A SIMULATION TOOL FOR MC6811

A THESIS SUBMITTED TO
THE GRADUTE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

NAZLI (TUNCER) SARIKAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRIC ELECTRONICS ENGINEERING

DECEMBER 2004

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof.Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof.Dr. Hasan GÜRAN
Supervisor

Examining Committee Members

Assoc.Prof.Dr. Gözde BOZDAĞI AKAR	(METU, EE)	
Prof.Dr. Hasan GÜRAN	(METU, EE)	
Asst. Prof. Dr. Cüneyt BAZLAMAÇCI	(METU, EE)	
Dr. Ece GÜRAN	(METU, EE)	
Ms.Sc. Taner DURUCAN	(KAREL)	

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that as required by these rules and conduct I have fully cited and referenced all material and results that are not original to this work.

Nazlı (TUNCER) SARIKAN

ABSTRACT

A SIMULATION TOOL FOR MC6811

(Tuncer) Sarıkan, Nazlı
M.S., Department of Electrics and Electronics Engineering
Supervisor : Prof.Dr. Hasan GÜRAN
December 2004, 202 pages

The aim of this thesis study is to develop a simulator for an 8-bit microcontroller and the written document of this thesis study analyses the process of devoloping a software for simulating an 8 bit microcontroller, MC68HC11. In this simulator study a file processing including the parsing of the assembler code and the compilation of the parsed instructions is studied. Also all the instruction execution process containing the cycle and instruction execution and the interrupt routine execution is observed through a graphical user interface. Through this graphical user interface all the registries, address bus and data bus updates can also be observed. C++ programming language is used to implement the application. Object oriented programing techniques are used to provide easy of implementation and template usages.

Keywords: Microcontroller, Motorola, simulator, 8-bit

ÖZ

MC6811 İÇİN BİR BENZETİMCİ ARACI

(Tuncer) Sarıkan, Nazlı
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof.Dr. Hasan GÜRAN
Aralık 2004, 202 sayfa

Bu tez çalışmasının amacı 8 bit mikrokontrolör için bir simülatör geliştirmektir; bu yazılı döküman 8 bit mikrokontrolör için bir simülatör geliştirme sürecini incelemektedir. Bu simülatör çalışmasında bir dosyadan assemler kodunun ayrıştırılması ve derlenmesi işlemleri çalışılmıştır. Ayrıca kullanıcı arayüzünden cycle ve komut çalıştırılması; ayrıca "interrupt" işlenmesi süreçleri gözlenebilmektedir. Mikrokontrolörde bulunan bütün değişkenler, adres yolu ve veri yolu değerleri de bu grafik ara yüzünden gözlenebilmektedir. Bu uygulama geliştirilirken C++ programlama dili kullanılmıştır. Ayrıca şablon kullanımını ve uygulama geliştilmesini kolaylaştırmak adına nesne tabanlı programlama teknikleri kullanılmıştır.

Anahtar Kelimeler: Mikrokontrolör, Motorola, simülatör, 8-bit

To My Husband

and

To My Parents

ACKNOWLEDGEMENTS

The author wishes to express her deepest gratitude to her supervisor Prof.Dr. Hasan GÜRAN for his guidance, advice, criticism, encouragements and insight throughout the research.

The author would also like to thank her husband Alper SARIKAN for his patience, advices, help and technical assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The aim of this thesis is to develop a simulator for an 8-bit microcontroller. In this simulator a code written in the assembler language of the microcontroller and saved as an asm file is opened and compiled by the assembler compiler written in C++ programming language as a part of the thesis study. The compiled instructions are executed and the results obtained from instructions are shown in the graphical user interface of the microcontroller simulator.

A microprocessor is the integration of a number of useful functions into a single integrated circuit package. These functions are:

- The ability to execute a stored set of instructions to carry out user defined tasks.
- The ability to be able to access external memeory chips to both read and write data from and to the memory.

Basically, a microcontroller is a device which integrates a number of the components of a microprocessor system onto a single microchip. So a microcontroller combines onto the same microchip:

- The CPU core
- Memory (both ROM and RAM)
- Some parallel digital I/O

This simulator is developed to show the compilation and execution properties of a microcontroller. In order to achive this purpose, Motorola MC68HC11 which is an 8-bit microcontroller is chosen.

While developing the software of the microcontroller simulator, C++ programming language and Borland C++ Builder 5.0 Professional Edition compiler is used.

An introduction for the simulated microcontroller MC68HC11 is given in the second chapter. In this chapter, a brief explanation of the microcontroller is given and the block diagram and programmer's model is introduced very briefly.

The simulator which is the subject of this thesis study, is mainly related with the assembler code software compilation and execution properties of the microcontroller MC68HC11. In the third chapter, resets and interrupts which are the two main subjects related with the

assembler code software compilation and execution properties of the microcontroller are explained in detail.

The M68HC11 central processing unit (CPU), which is responsible for executing all software instructions in their programmed sequence, discussed in the fourth chapter. The internal registries and the roles of these registries during the program execution are also explained. Besides these, in this chapter, the instruction set of the microcontroller MC68HC11 is also explained as divided into functional groups of instructions.

In the fifth chapter the architecture of the software implemented as the subject of this thesis study is explained in detail. All the objects, structures, functions and the excution sequence developed in the project are explained.

CHAPTER 2

GENERAL DESCRIPTION

In this chapter a brief explanation of the microcontroller MC68HC11 is given, so the basic properties of the microcontroller is introduced [2], [3], [4].

2.1 General Description of the MC68HC11

The MC68HC11 is an advanced 8-bit MCU with highly sophisticated, on-chip peripheral capabilities. New design techniques were used to achieve a nominal bus speed of 2 MHz. In addition, the fully static design allows operation at frequencies down to dc, further reducing power consumption.

On-chip memory systems include 8 Kbytes of read-only memory (ROM), 512 bytes of electrically erasable programmable ROM (EEPROM), and 256 bytes of random-access memory (RAM).

Major peripheral functions are provided on-chip. An eight-channel analog-to-digital (A/D) converter is included with eight bits of resolution. An asynchronous serial communications interface (SCI) and a separate synchronous serial peripheral interface (SPI) are included. The main 16-bit, free-running timer system has three input-capture lines, five output-compare lines, and a real-time interrupt function. An 8-bit pulse accumulator subsystem can count external events or measure external periods.

Self-monitoring circuitry is included on-chip to protect against system errors. A computer operating properly (COP) watchdog system protects against software failures. A clock monitor system generates a system reset in case the clock is lost or runs too slow. An illegal opcode detection circuit provides a non-maskable interrupt if an illegal opcode is detected.

Two software-controlled power-saving modes, WAIT and STOP, are available to conserve additional power. These modes make the M68HC11 Family especially attractive for automotive and battery-driven applications.

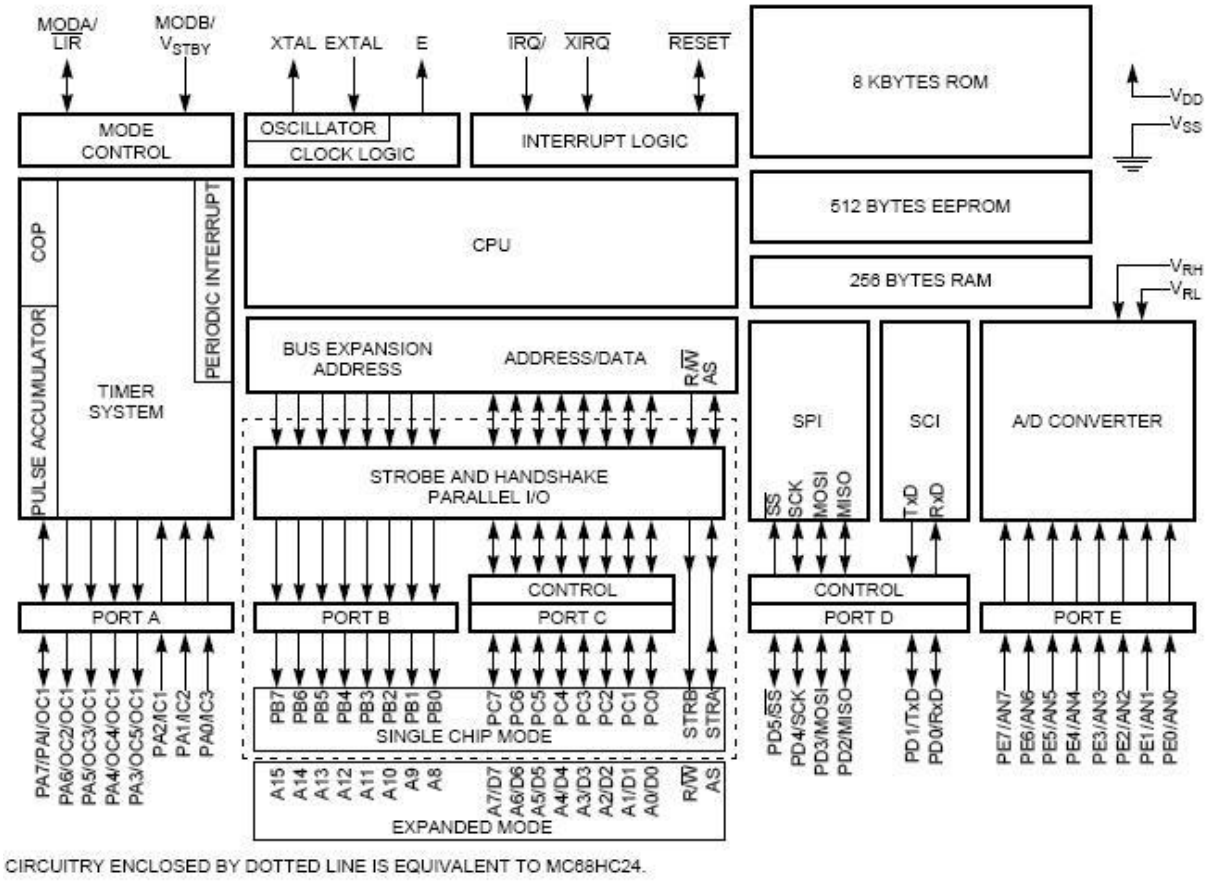

Figure 2.1 Block Diagram [2]

Figure 2.1 is a block diagram of the MC68HC11 MCU. This diagram shows the major subsystems and how they relate to the pins of the MCU. In the lower right-hand corner of this diagram, the parallel I/O subsystem is shown inside a dashed box. The functions of this subsystem are lost when the MCU is operated in expanded modes, but the MC68HC24 port replacement unit can be used to regain the functions that were lost. The functions are restored in such a way that the software programmer is unable to tell any difference between a single-chip system or an expanded system containing the MC68HC24. By using an expanded system containing an MC68HC24 and an external EPROM, the user can develop software intended for a single-chip application.

2.2 Programmer's Model

In addition to executing all M6800 and M6801 instructions, the M68HC11 instruction set includes 91 new opcodes. The nomenclature M68xx is used in conjunction with a specific CPU architecture and instruction set as opposed to the MC68HC11xx nomenclature, which

is a reference to a specific member of the M68HC11 Family of MCUs. Figure 2.2 shows the seven CPU registers available to the programmer. The two 8-bit accumulators (A and B) can be used by some instructions as a single 16-bit accumulator called the D register, which allows a set of 16-bit operations even though the CPU is technically an 8-bit processor.

The largest group of instructions added to involve the Y index register. Twelve bit manipulation instructions that can operate on any memory or register location were added. The exchange D with X and exchange D with Y instructions can be used to quickly get index values into the double accumulator (D) where 16-bit arithmetic can be used. Two 16-bit by 16-bit divide instructions are also included.

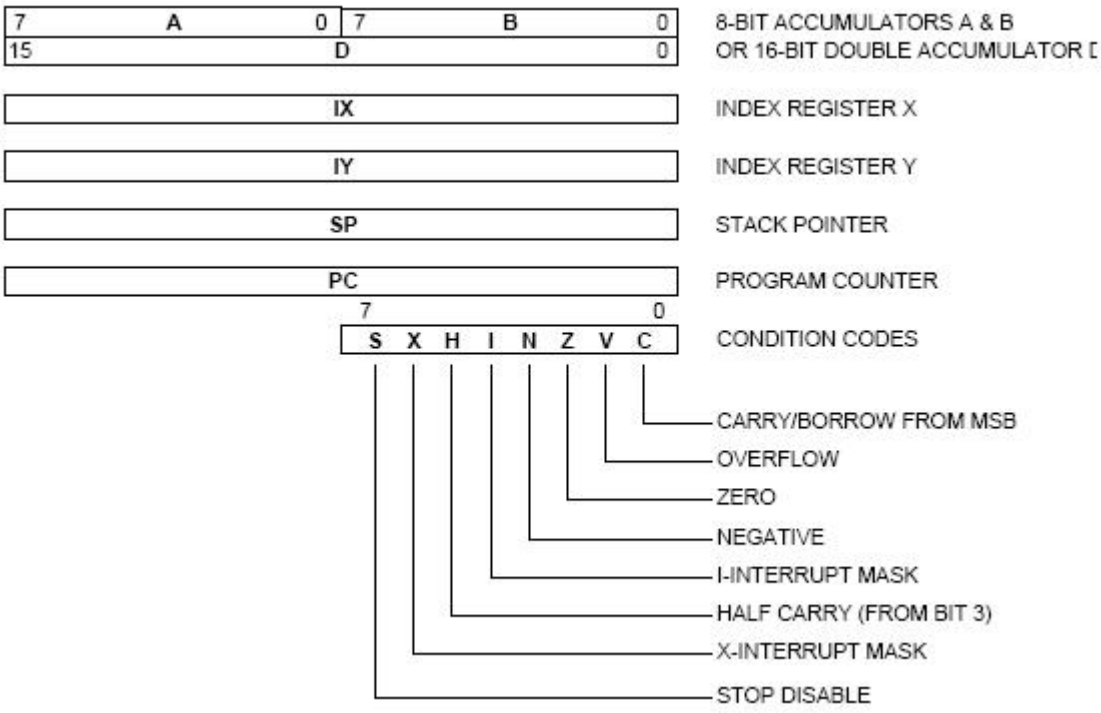

Figure 2.2 M68HC11 Programmer's Model [2]

CHAPTER 3

RESET AND INTERRUPT PROPERTIES OF MC68HC11

This chapter is mostly taken from the reference “HC11 M68HC11 Reference Manual, Motorola Inc., 1996” in order not to disturb the integrity of the thesis report.

The simulator which is the subject of this thesis study, is mainly related with the assembler code software compilation and execution properties of the microcontroller MC68HC11.

Resets and interrupts are two of the basic subjects related with the assembler code software compilation and execution properties of the microcontroller since these subjects include initialization process of the microcontroller and registry changes occuring during these processes. In this chapter mainly resets and interrupts processed by the microcontroller are explained in detail [2], [3], [4].

3.1 RESETS AND INTERRUPTS

Reset and interrupt operations are often discussed together because they share the common concept of vector fetching to force a new starting point for further central processing unit (CPU) operations. The reset structure in the MC68HC11, which is quite different from other MCUs, is presented in this section. This reset system can generate a reset output if reset-causing conditions are detected by internal systems. The on-chip electrically erasable programmable read-only memory (EEPROM) also places extra demands on external circuitry connected to the RESET pin.

The MC68HC11 includes 18 separate interrupt sources. On-chip peripheral systems generate maskable interrupts, which are recognized only if the global interrupt mask bit (I) in the condition code register (CCR) is clear. Three interrupt sources considered non-maskable will be discussed in detail in this section.

Maskable interrupts are prioritized according to a default arrangement; however, any one source may be elevated to the highest maskable priority position by a software accessible control register. This highest priority interrupt (HPRIO) register may be written at any time provided the I-bit in the CCR is set.

When interrupt conditions occur in an on-chip peripheral system, an interrupt status flag is set to indicate the condition. When the user's program has properly responded to this interrupt request, the status flag must be cleared. The method of clearing varies from one system to another, depending on the requirements of the system.

3.1.1 Initial Conditions Established During Reset

Reset is used to force the microcontroller unit (MCU) to assume a set of initial conditions and to begin executing instructions from a predetermined starting address. For most practical applications, the initial conditions take effect almost immediately after applying an active-low level to the RESET pin. Some reset conditions cannot take effect until/unless a clock is applied to the external clock input (EXTAL) pin. One example is port B, which acts as an address output port in the expanded modes and as a general-purpose output port in the single-chip modes. During reset in expanded mode, these pins would be $FF because this is the high-order half of $FFFE. During reset in single-chip mode, these pins would be $00. Since the mode pins are pipelined into the MCU, a clock is needed for the MCU to recognize the mode selected.

If no clock is present, the port B pins could be in the wrong state due to the inability of the MCU to recognize the correct mode of operation. If no clock is present, the MCU cannot advance out of the reset condition since internal reset is a clocked sequence; thus, the MCU cannot advance past the first step of this sequence. Even with no clock present, a RESET signal will cause some changes. Most important, an unclocked RESET signal resets the clock divider circuitry so the on-chip oscillator will start. If an application includes external clock circuitry driving the EXTAL pin, the RESET signal should force this external clock to resume oscillation.

3.1.1.1 System Initial Conditions

Once the reset condition is recognized, internal registers and control bits are forced to an initial state. These initial states, in turn, control on-chip peripheral systems to force them to known start-up states. Most of the initial conditions are independent of the operating mode.

3.1.1.1.1 CPU

After reset, the CPU fetches the restart vector from locations $FFFE, $FFFF ($BFFE, $BFFF if in special test or bootstrap mode) during the first three cycles and begins executing instructions. The stack pointer and other CPU registers are indeterminate immediately after reset; however, the X and I interrupt mask bits in the CCR are set to mask any interrupt requests. Also, the S bit in the CCR is set to disable the STOP mode.

3.1.1.1.2 Memory Map

After reset, the RAM and I/O mapping (INIT) register is initialized to $01, putting the 256 bytes of random-access memory (RAM) at locations $0000–$00FF and the control registers at locations $1000–$103F. The 8-Kbyte read-only memory (ROM) and/or the 512-byte EEPROM may or may not be present in the memory map because the two bits that enable them in the configuration control (CONFIG) register are EEPROM cells not affected by reset or power-down.

3.1.1.1.3 Parallel I/O

When a reset occurs in expanded-multiplexed operating mode, the 18 pins used for parallel I/O are dedicated to the expansion bus. If a reset occurs in the single-chip operating mode, the strobe A flag (STAF), strobe A interrupt (STAI), and handshake (HNDS) control bits in the parallel input/output control (PIOC) register are cleared so that no interrupt is pending or enabled, and the simple strobed mode (rather than full handshake mode) of parallel I/O is selected. The port C wired-OR mode (CWOM) bit in PIOC is cleared. Port C is initialized as an input port (data direction register for port C, DDRC $00); port B is a general-purpose output port with all bits cleared. STRA is the edge-sensitive strobe A input, and the active edge is initially configured to detect rising edges (edge select for strobe A (EGA) bit in PIOC is set). Port C, port D (bits [5:0]), port A (bits 0, 1, 2, and 7), and port E are configured as general-purpose high impedance inputs. Port B and bits [6:3] of port A have their directions fixed as outputs, and their reset state is logic zero.

3.1.1.1.4 Timer

During reset, the timer system is initialized to a count of $0000. The prescaler bits are cleared, and all output-compare registers are initialized to $FFFF. All input-capture registers are indeterminate after reset. The output-compare 1 (OC1M) mask register is cleared so that successful OC1 compares do not affect any I/O pins. The other four output compares are configured to not affect any I/O pins on successful compares. All three input-capture edge-detector circuits are configured for capture-disabled operation. The timer overflow interrupt flag and all eight timer function interrupt flags are cleared. All nine timer interrupts are disabled since their mask bits are cleared.

3.1.1.1.5 Real-Time Interrupt

The real-time interrupt flag is cleared, and automatic hardware interrupts are masked. The rate control bits are cleared after reset and may be initialized by software before the real-time interrupt system is used.

3.1.1.1.6 Pulse Accumulator

The pulse accumulator system is disabled at reset so that the pulse accumulator input (PAI) pin defaults to being a general-purpose input pin.

3.1.1.1.7 COP Watchdog

The computer operating properly (COP) watchdog system is enabled if the NOCOP control bit in the CONFIG register (EEPROM cell) is clear and disabled if NOCOP is set. The COP rate is set for the shortest duration time-out.

3.1.1.1.8 Serial Communications Interface (SCI)

The reset condition of the SCI system is independent of the operating mode. At reset, the SCI baud rate is indeterminate and must be established by a software write to the BAUD register. All transmit and receive interrupts are masked, and both the transmitter and receiver are disabled so the port pins default to being general-purpose I/O lines. The SCI frame format is initialized to an 8-bit character size. The send break and receiver wake-up functions are disabled. The transmit data register empty (TDRE) and transmit complete (TC) status bits in the SCI status register are both set, indicating that there is no transmit data in either the transmit data register or the transmit serial shift register. The receive data register full (RDRF), IDLE, overrun (OR), and framing error (FE) receive-related status bits are all cleared. Upon reset in special bootstrap mode, execution begins in the 192-byte bootstrap ROM, which changes some of the initial conditions by the time the bootloading process is finished. This firmware sets port D to wired-OR mode, establishes a baud rate, and enables the SCI receiver and transmitter.

3.1.1.1.9 Serial Peripheral Interface (SPI)

The SPI system is disabled by reset. The port pins associated with this function default to being general-purpose I/O lines.

3.1.1.1.10 Analog-to-Digital (A/D) Converter

The A/D converter system configuration is indeterminate after reset. The conversion complete flag is cleared by reset. The A/D power-up (ADPU) bit is cleared by reset, disabling the A/D system.

3.1.1.1.11 Other System Controls

The EEPROM programming controls are all disabled so the memory system is configured for normal read operation. The highest priority I bit interrupt defaults to being the external interrupt request (IRQ) pin by PSEL[3:0] equal to 0:1:0:1. The IRQ pin is configured for level-sensitive operation (for wired-OR systems). The read bootstrap ROM (RBOOT), special mode (SMOD), and mode A (MDA) bits in the HPRIO register reflect the status of the mode B (MODB) and MODA inputs at the rising edge of reset. The enable oscillator start-up delay (DLY) control bit is set to specify that an oscillator startup delay is imposed upon recovery from STOP mode. The clock monitor system is disabled by clock monitor enable (CME) equals zero.

The MC68HC11 has three internal sources that can cause reset as well as the external application of a low level to the RESET pin. No matter which of these sources causes reset, the entire MCU is reset. The RESET pin is driven low as a result of any of the reset sources. The only distinction that is made between the causes of reset is the reset vector, which is used to tell the CPU the starting address for execution when reset is released.

A few registers are not forced to a start-up condition as a result of reset. Since these registers do not affect the starting conditions at MCU pins, it is not important to force them to a start-up state during reset. One such example is the main-timer input-capture registers. Since these registers are not useful until after an input capture occurs, it is not important to force them to a start-up state during reset.

3.1.1.2 CONFIG Register Allows Flexible Configuration

The M68HC11 includes a nonvolatile CONFIG register, which controls a number of options typically controlled by mask options or by additional mode selection choices in other MCUs. By using a nonvolatile EEPROM-based register, it is possible to achieve the same effects as if the options were mask programmed and, at the same time, allow users to change these features after the MCU is manufactured. The most important aspect of this method of selecting options is that the selections automatically take effect on any power-up or reset without any software intervention. Two classes of features can be controlled in this manner. First, there are configuration choices that must inherently be made before the reset vector is even fetched. For example, the ROM enable must be decided so that the reset vector can be fetched out of the correct memory as the MCU comes out of reset. The COP watchdog timer enable is an example of the second class of features that can be controlled by an EEPROM bit. The COP watchdog timer is intended to detect software failures; thus, it is important to enable or disable this feature without any software intervention. If software could disable or was required to enable the COP watchdog, the COP watchdog timer could not detect a failure of that software.

The CONFIG register controls the presence or absence of ROM and/or EEPROM, enables/disables the COP watchdog timer, and engages/disengages the security option. The features enabled by the CONFIG register can be thought of as mask-programmed options that do not require software service.

3.1.1.3 Program Counter Loaded with Reset Vector

As reset is released, the CPU program counter is loaded with the reset vector that points to the first instruction in the user's program. Depending on the cause of reset and the mode of operation, the reset vector may be fetched from any of six possible locations. In older Motorola MCUs, there was only one reset vector at $FFFE, FFFF.

3.1.2 Causes Of Reset

In the MC68HC11, there are on-chip systems that can detect MCU system failures and generate a low level out the RESET pin to reinitialize other peripherals in the system. To distinguish between these causes, separate reset Vectors are used. The primary reset vector is used when the cause of reset is the internal power-on reset circuit or application of a low level to the RESET pin. In normal expanded and normal single chip modes, this vector is located at $FFFE, FFFF. If the oscillator input stops or is running too slow, the clock monitor circuit will generate a reset (provided the clock monitor is enabled). Time-out of the internal COP watchdog timer will generate a reset (provided the COP system is enabled). Table 2.1 summarizes the reset-vector locations versus the cause of reset and mode of operation.

Table 3.1 Reset Vector vs. Cause and MCU Mode [2]

Cause of Reset	Normal Mode Vector	Special Test or Bootstrap Vector
POR or RESET Pin	$FFFE, FFFF	$BFFE, BFFF
Clock Monitor Fail	$FFFC, FFFD	$BFFC, BFFD
COP Watchdog Time-Out	$FFFA, FFFB	$BFFA, BFFB

In special test and bootstrap modes, MCU vectors are located at $BFC0–$BFFF rather than the normal $FFC0–$FFFF area. The primary reason for this change is to be sure the reset vector can be supplied from an external source in special test mode. The normal reset vector is located at $FFFE, FFFF, which can be internal ROM or external memory space (depending on whether the internal ROM is enabled). The special test mode reset vector is at $BFFE, BFFF, which is always an external access independent of other system conditions.

This alternate mapping is important to the operation of bootstrap mode because it allows reset and other vectors to be located within the 192-byte bootloader ROM. As the MCU comes out of reset in special bootstrap mode, the reset vector is fetched out of the bootloader ROM, and execution begins at the start of the bootloader program. While in bootstrap mode, interrupts can be vectored to locations in the bootloaded program in RAM rather than vectoring to the routines specified in the internal ROM program.

The M68HC11 MCU is capable of distinguishing between an external reset and resets from the internal COP and clock monitor systems. When the COP watchdog timer times out or the clock monitor detects a clock failure, the COP and clock monitor status is temporarily saved. The RESET pin is then driven low for about four E-clock cycles and is released. Two E-clock cycles later, the RESET input is sampled. If RESET is high (has risen to logic one within the two cycles since it was released), the source of reset is presumed to be either the COP or clock monitor system. If RESET is still low, the source is presumed to be an external reset request, and the temporarily saved status from the COP and clock monitor systems is erased. Although there would rarely be more than one cause for a particular reset sequence, the three reset vectors are prioritized. If an external reset request drives the RESET pin low for less than four E-clock cycles, the differentiation logic could assume the source of reset was the internal COP or clock monitor system; however, as long as neither of these causes was indicated by the temporarily latched status, the normal reset vector would still be used by default. Although this MCU can differentiate between different reset causes, the most common implementation would direct all reset vectors to the same initialization software, regardless of the cause of reset.

There are four possible sources of reset in the MC68HC11. An internal circuit detects the rising edge on VDD and initiates a power-on reset. An on-chip COP watchdog timer monitors proper software execution; if software does not service this timer within its time-out period, a system reset is generated. Another on-chip circuit monitors the MCU clock frequency. If the MCU clock stops or is running too slow, a system reset is generated. Finally, a user can initiate an external reset by momentarily driving the RESET pin low. The COP and clock monitor features can be disabled. The power-on reset and external reset share the normal reset vector; whereas, the COP and clock monitor reset each have their own vector. The four causes of reset are described in greater detail in the following paragraphs.

3.1.2.1 Power-On Reset (POR)

The POR is only intended to initialize internal MCU circuits. As VDD is applied to the MCU, the POR circuit triggers and initiates a reset sequence. POR triggers an internal timing circuit that holds the RESET pin low for 4064 cycles of the internal PH2 clock. The MCU does not advance past this reset condition until a clock is present at the EXTAL pin long enough for these 4064-cycle PH2 clocks to be detected. The internal POR circuit will not retrigger unless VDD has discharged to 0 V; therefore, the internal POR circuit is not suitable as a power-loss detector.

In almost all M68HC11 systems, there will be an external circuit to hold the RESET pin low whenever VDD is below normal operating level. This external voltage-level detector or other external reset circuits are the normal source of reset in a system; the internal POR circuit only serves to initialize internal control circuitry during cold starts.

In some unusual applications, it may be desirable to hold RESET low long enough for the oscillator to reach stable operating frequency. This stable operating frequency is not a requirement of the MCU because the M68HC11 is a fully static design, which can operate correctly even when the oscillator has not reached stable operating frequency. If the oscillator has not reached stable operating frequency by the time RESET is released, software and timed delays will be longer than expected since these delays are based on the oscillator frequency. In most applications, such errors within the first few milliseconds of operation are of no concern, and no external power-on delay is necessary. In cases where timing is critical immediately out of RESET, an external POR circuit must be provided. The required amount of delay depends upon the oscillator startup time, which varies with the frequency and design of the oscillator as well as such things as VDD rise time. In a typical M68HC11 design with an E-clock frequency of 2 MHz, the internal POR will only hold RESET low for about 2 ms after oscillator start. With an 8-MHz crystal, the M68HC11 oscillator will typically start when VDD reaches about 1 V. For a typical VDD rise time, the internal POR times out well before VDD reaches an acceptable level. Thus, POR alone is rarely able to provide for all reset needs, and some external reset circuitry will be required.

3.1.3 Interrupt Process

The CPU in a microcontroller sequentially executes instructions. In many applications, it is necessary to execute sets of instructions in response to requests from various peripheral devices. These requests are often asynchronous to the execution of the main program. Interrupts provide a way to temporarily suspend normal program execution so the CPU can

be freed to service these requests. After an interrupt has been serviced, the main program resumes as if there had been no interruption.

The instructions executed in response to an interrupt are called the interrupt service routine. These routines are much like subroutines except that they are called through the automatic hardware interrupt mechanism rather than by a subroutine call instruction, and all CPU registers are saved on the stack rather than just saving the program counter. An interrupt (provided it is enabled) causes normal program flow to be suspended as soon as the currently executing instruction finishes. The interrupt logic then pushes the contents of all CPU registers onto the stack so the CPU context can be restored after the interrupt is finished. After stacking the CPU registers, the vector for the highest priority pending interrupt source is loaded into the program counter, and execution continues with the first instruction of the interrupt service routine. An interrupt is concluded with a return from interrupt (RTI) instruction, which causes all CPU registers and the return address to be recovered from the stack so that the interrupted program can resume as if there had been no interruption.

Interrupts can be enabled or disabled by mask bits (X and I) in the CCR and by local enable mask bits in the on-chip peripheral control registers. A few important interrupt sources that are always enabled are called non-maskable interrupts. The nonmaskable interrupt request (XIRQ) pin is effectively a non-maskable interrupt source except that it is disabled immediately after reset. Very special logic is associated with the interrupt mask bit (X) for XIRQ in the CCR to overcome classic problems associated with a non-maskable interrupt while allowing all of the benefits of such an interrupt. The remaining interrupt sources are maskable by the interrupt mask bit (I) in the CCR.

The interrupt mask bits in the CCR provide a means of controlling the nesting of interrupts. In rare cases, it may be useful to allow an interrupt routine to be interrupted (nesting of interrupts). Nesting of interrupts is discouraged because it greatly complicates a system and rarely improves system performance. By default, the interrupt structure inhibits interrupts during the interrupt entry sequence by setting the interrupt mask bit(s) in the CCR. As the CCR is recovered from the stack during the RTI instruction, the CCR bits return to the enabled state so additional interrupts can be serviced. If nesting of interrupts is desired, it must be specifically allowed by clearing the interrupt mask bit(s) after entering the interrupt service routine. Care must be taken to specifically mask (disable) the present interrupt with a local enable mask bit or to clear the interrupt source flag before clearing the mask bit in the CCR; otherwise, the same source would immediately interrupt, and an infinite loop could result.

Upon reset, both the X and I bit are set to inhibit all maskable interrupts and XIRQ. After minimum system initialization, software may clear the X bit by a transfer accumulator A to CCR (TAP) instruction, thus enabling XIRQ. Thereafter, software cannot set the X bit; thus, an XIRQ is effectively a non-maskable interrupt. Since the operation of the I-bit-related interrupt structure has no effect on the X bit, the external XIRQ pin remains effectively non-maskable. In the interrupt priority logic, XIRQ is a higher priority than any source that is maskable by the I-bit. All I-bit-related interrupts operate normally with their own priority relationship. When an I-bit-related interrupt occurs, the I-bit is automatically set by hardware after stacking the CCR byte, but the X bit is not affected. When an XIRQ occurs, both the X and I bits are automatically set by hardware after stacking the CCR. An RTI instruction restores the X and I bits to their pre-interrupt request state.

3.1.3.1 Interrupt Recognition and Stacking Registers

An interrupt can be recognized at any time provided it is enabled by its local mask (if any) and by the global mask bit in the CCR. Once any interrupt source is recognized, the CPU will respond at the completion of the currently executing instruction. Instructions cannot be interrupted; rather, the CPU decides whether to fetch another instruction or process an interrupt. In calculating the latency time from the actual interrupt request to the CPU response to that request, the user must consider the possibility that the CPU had just started a long instruction as the interrupt was requested. Most instructions are two to four cycles long, but the multiply (MUL) and integer divide (IDIV) or fractional divide (FDIV) instructions are 10 and 41 cycles, respectively.

When the CPU decides to service an interrupt, the contents of CPU registers are pushed (stored) on the stack in the order PCL, PCH, IYL, IYH, IXL, IXH, ACCA, ACCB, CCR. After the CCR value is stacked, the I-bit in the CCR (and the X bit if XIRQ is pending) is set to inhibit further interrupts. The interrupt sequence then proceeds to the priority resolution step.

3.1.3.2 Selecting Interrupt Vectors

After the CCR has been stacked, the CPU evaluates all pending interrupt requests to determine which source has the highest priority. Since the priority resolution step occurs several cycles after the original decision to service an interrupt, a higher priority source could become pending after the stacking operation started but before the priority is resolved. In such a case, the interrupt that is serviced can be different from the source that initiated the interrupt sequence. This subtle aspect means that the latency from an interrupt request to when it is serviced can be shorter than expected.

Interrupts obey a fixed hardware-priority circuit to resolve simultaneous requests; however, one I-bit-related interrupt source may be elevated to the highest I bit priority position in the resolution circuit. The first six interrupt sources are not masked by the I-bit in the CCR and have the fixed priority interrupt relationship: reset, clock monitor fail, COP fail, illegal opcode, and XIRQ. Each of these sources is an input to the priority resolution circuit. Software interrupt (SWI) is actually an instruction and has the highest priority other than reset because, once the SWI opcode is fetched, no other interrupt can be honored until the SWI vector has been fetched. The highest I-bit-related priority input is assigned under software control (of the HPRIO register) to be connected to any one of the remaining I-bit-related interrupt sources. To avoid timing races, the HPRIO register may only be written while the I-bit-related interrupts are inhibited (I bit in CCR = 1). An interrupt that is assigned to this highest priority position is still subject to masking by any associated control bits or by the I-bit in the CCR. The interrupt vector address is not affected by assigning a source to this highest priority position.

3.1.3.3 Return from Interrupt

When an interrupt has been serviced as needed, the RTI instruction terminates interrupt processing and returns to the program that was running at the time of the interruption. During servicing of the interrupt, some or all of the CPU registers will have changed. To continue the former program as if it had not been interrupted, the registers must be restored to the values present at the time the former program was interrupted. The RTI instruction accomplishes this by pulling (loading) the saved register values from the stack memory. The last value to be pulled from the stack is the program counter, which causes processing to resume where it was interrupted.

3.1.4 Non-Maskable Interrupts

This subsection discusses the illegal opcode fetch interrupt, the SWI instruction, and the XIRQ input pin. The illegal opcode fetch interrupt is a non-maskable interrupt source intended to improve system integrity. Although it performs like an interrupt, SWI is an instruction rather than an asynchronous interrupt. The XIRQ input is an updated version of the non-maskable interrupt (NMI) input of earlier MCUs.

3.1.4.1 Non-Maskable Interrupt Request (XIRQ)

Non-maskable interrupts are useful because they can always interrupt CPU operation. The most common use for such an interrupt is for very serious system problems, such as program runaway or power failure. The XIRQ mechanism over-comes two significant

problems with an NMI input while retaining the important capabilities associated with a non-maskable source.

The first NMI problem is as follows: What if an NMI is requested before the stack pointer has been initialized? If this request happens, the register stacking operation causes register values to be written to a random area of memory. If the stack pointer is pointing to some unimplemented memory area or to a read-only area, there will be no way to return to the program in progress at the time of the interrupt. If the stack pointer is pointing at a data area in memory, the register values will be written over the data (thus corrupting it). Since this situation is not desirable, the NMI had to be externally inhibited after reset until the stack pointer could be initialized.

The second NMI problem is as follows: What if the NMI signal bounces so that NMI is nested? If nesting occurs, the stack can be filled with several copies of the register values, possibly filling the stack beyond its allotted space. Nesting in this way would also cause excessive latency from the request until the resulting program actions are executed.

The M68HC11 solves both these problems with the X bit in the CCR. The X bit is very similar to the I-bit except that there are special restrictions on setting and clearing of the X bit. Since X can only be cleared by a software instruction, the programmer has control over when the XIRQ input becomes enabled. The two software instructions that can clear the X bit are TAP and RTI (provided the stacked CCR value has a zero in the X bit position). The two hardware conditions that can set the X bit are system reset and the recognition of an XIRQ.

Immediately after any reset, the X bit is set; thus, XIRQ is inhibited. When software has established initial conditions, including setting the stack pointer, the X bit may be cleared with a TAP instruction to enable XIRQ. These two steps overcome the first NMI problem. Since software cannot set the X bit, the XIRQ can be considered a nonmaskable source at this point. When an XIRQ occurs, the CCR value is stacked (with the X bit clear); the X bit is then automatically set to inhibit additional interrupts. This step overcomes the second NMI problem. When an RTI instruction is executed, the CCR is restored to the stacked value (which had the X bit clear). A common misconception is that the X bit can be set by executing an RTI instruction with a one in the X bit position of the stacked CCR value. In reality, the X bit is implemented as a set-reset flip-flop rather than a D-type flip-flop. The set input is connected to the OR of reset and XIRQ acknowledge. The reset input is connected to the AND of a CCR write and data bit 6 equals zero. If an attempt is made to TAP or unstack a one to the X bit, neither the set nor the clear input to the X bit flip-flop will be activated, and the X bit will remain unchanged.

The M68HC11 supports a STOP mode where all clocks are stopped to reduce power consumption to a few microamps. Recovery to active mode is accomplished by a reset or an interrupt (IRQ or XIRQ). Depending upon the state of the X bit in the CCR, the XIRQ input offers a choice of two recovery methods. If X is zero, XIRQ interrupts are enabled, and recovery leads to register stacking and normal interrupt service. If X is one, XIRQ interrupts are inhibited, but the XIRQ pin can still be used for recovery from the STOP mode. Rather than resuming operation with service of an interrupt (XIRQ), the clocks start and processing resumes with the next opcode after the STOP opcode. This technique can be thought of as a STOP-continue mechanism.

Some M68HC11 MCUs were manufactured with a subtle defect that can cause failure to properly recover from STOP with an interrupt input (IRQ or XIRQ). If the opcode immediately preceding the STOP opcode came from column 4 or 5 of the opcode map, recovery was incorrect. Column 4 and 5 opcodes are accumulator instructions, such as negate A (NEGA) or decrement B (DECB), which seldom appear immediately before a STOP instruction; therefore, a long time elapsed before the problem was discovered. A simple NOP instruction before the STOP opcode assures proper recovery from STOP in all cases.

3.1.4.2 Software Interrupt

The SWI is executed in the same manner as other instructions and takes precedence over pending interrupts only if the other interrupts are masked (I and X bits in the CCR set). The SWI instruction is executed in a manner similar to other maskable interrupts in that it sets the I-bit, CPU registers are stacked, etc. SWI is not inhibited by the global interrupt mask bits (X or I) in the CCR.

NOTE: The SWI instruction will not be fetched if any other interrupt is pending. However, once an SWI instruction begins, no other interrupt can be honored until the SWI vector has been fetched.

SWI instructions are commonly used in debug monitors to transfer control from a user program to the debug monitor. For example, while operating under monitor control, a designer can specify a breakpoint at some address in the user program being debugged. The monitor will replace the user's opcode at this address with the opcode for an SWI instruction. When the user's program is running and this SWI opcode is encountered, the monitor, recognizing that this is a breakpoint, will take control. The SWI opcodes are

usually placed into the user's program just before the program is run, and these locations are restored to the original opcode when the debug monitor regains control.

3.1.5 Maskable Interrupts

The remaining twenty interrupt sources in the MC68HC11 are subject to masking by a global interrupt mask bit (I bit in CCR). In addition to the global I bit, all of these sources except the external interrupt (IRQ pin) are subject to local enable bits in control registers. Most interrupt sources in the M68HC11 have separate interrupt vectors; thus, there is usually no need for software to poll control registers to determine the cause of an interrupt. The maskable interrupt sources respond to a fixed-priority relationship except that any one source can be dynamically elevated to the highest priority position of any maskable source.

This subsection discusses the maskable interrupt structure rather than the specific interrupts from individual internal peripheral subsystems. The interrupts associated with the internal subsystems are discussed throughout this manual during the discussion of each peripheral system.

3.1.5.1 I Bit in the Condition Code Register

The I-bit in the CCR acts as a primary enable control for all maskable interrupts. When the I-bit is set, interrupts can become pending but will not be honored. When the I-bit is clear, interrupts are enabled to interrupt normal program flow when an interrupt source requests service.

The I-bit is set during reset to prevent interrupts from being honored until minimum system initialization has been performed. Part of this minimum initialization would be to load the stack pointer so it points to an appropriate area of RAM. The I-bit is also automatically set during entry into any interrupt service routine to prevent an infinite source of interrupts from overwhelming the CPU. Software can also set the interrupt mask bit to inhibit interrupts during sensitive operations.

The I-bit can be cleared by software instructions or during the execution of an RTI instruction. In most applications, the I-bit remains set during interrupt service routines so other interrupts will not be honored until a current interrupt service routine finishes (i.e., nesting is not permitted). In more unusual applications, it is possible to allow nesting of interrupts by clearing the I-bit during an interrupt service routine. Since this procedure requires much expertise, it should not be attempted by a novice programmer. In some cases, worst-case interrupt latency can be reduced by allowing interrupt nesting, but usually the best procedure is to minimize the execution time of interrupt service routines.

Since the overhead associated with interrupt nesting usually violates this procedure, nesting is not recommended.

The operation of the I-bit during service of an interrupt proceeds as follows. When an enabled interrupt occurs and the I-bit is clear, the CPU completes the current instruction and begins the interrupt response sequence. The current contents of the CPU registers are pushed onto the stack (stored in stack RAM). The register values are saved one byte at a time in the following order: PCL, PCH, IYL, IYH, IXL, IXH, ACCA, ACCB, and CCR. After the CCR value is stacked, the I-bit in the CCR is set to inhibit further interrupts. Next, the vector for the highest priority pending interrupt is fetched, and processing continues with execution of the first instruction in the interrupt service routine. The last instruction in the interrupt service routine is the RTI instruction. This instruction causes the previously stacked register values to be loaded back into the registers in reverse order. Since the program counter is restored to its pre-interrupt value, the next instruction executed will be the instruction that would have been executed if the interrupt had not occurred.

A common error for new users is to put a set interrupt mask (SEI) instruction at the beginning of an interrupt service routine and a clear interrupt mask (CLI) instruction just before the RTI instruction. These instructions should not be used in this way because they are redundant. The automatic interrupt logic already sets the I-bit on the way into an interrupt and clears the I-bit during normal execution of the RTI instruction.

3.1.5.2 Special Considerations for I-Bit-Related Instructions

There are some special conditions associated with the I-bit that require additional consideration. The I-bit is actually a sequential logic circuit rather than a simple flip-flop. When the I-bit is set by an SEI or a TAP instruction, interrupts are inhibited immediately. An interrupt occurring while an SEI instruction is executing will not be honored until unless the I-bit is later cleared. When the I-bit is cleared by a CLI or TAP instruction, the actual clear operation is delayed for one bus cycle so the instruction following the CLI or TAP will always be executed. This procedure implies that the following loop can never be interrupted by a maskable interrupt:

```
LOOP  CLI        Enable Interrupts
      SEI        Disable Interrupts
      BRA LOOP   Repeat
```

The reason for this delayed clear operation can be seen in the next instruction sequence:

CLI Enable Interrupts
WAI Wait for an Interrupt

If there were not a delay in clearing the I-bit, it is possible the interrupt could be recognized between the CLI and WAI instructions. Upon return from the interrupt service routine, the WAI instruction would be executed, and the CPU would erroneously wait for the interrupt that was just serviced.

During execution of an RTI instruction, the first register to be restored from the stack is the CCR. In this situation, the one-cycle delay in clearing the I-bit expires long before the RTI instruction is finished; thus, a new interrupt sequence can be started even before a single instruction of the interrupted program is executed.

CHAPTER 4

PROGRAMMER INSTRUCTIONS

This chapter discusses the M68HC11 central processing unit (CPU), which is responsible for executing all software instructions in their programmed sequence [2], [3], [4]. The M68HC11 CPU can execute all M6800 and M6801 instructions (source and objectcode compatible) and more than 90 new instruction opcodes. Since more than 256 instruction opcodes exist, a multiple-page opcode map is used in which some new instructions are specified by a page-select prebyte before the opcode byte.

The architecture of the M68HC11 CPU causes all peripheral, I/O, and memory locations to be treated identically as locations in the 64-Kbyte memory map. Thus, there are no special instructions for I/O that are separate from those used for memory. This technique is sometimes called "memory-mapped I/O". In addition, there is no execution- time penalty for accessing an operand from an external memory location as opposed to a location within the MCU.

The M68HC11 CPU offers several new capabilities when compared to the earlier M6801 and M6800 CPUs. The biggest change is the addition of a second 16-bit index register (Y). Powerful, new bit-manipulation instructions are now included, allowing manipulation of any bit or combination of bits in any memory location in the 64-Kbyte address space. Two new 16-bit by 16-bit divide instructions are included. Exchange instructions allow the contents of either index register to be exchanged with the contents of the 16-bit double accumulator. Finally, several instructions have been upgraded to make full 16-bit arithmetic operations even easier than before.

4.1 Programmer’s Model

Figure 4.1 shows the programmer’s model of the M68HC11 CPU. The CPU registers are an integral part of the CPU and are not addressed as if they were memory locations. Each of these registers is discussed in the subsequent paragraphs.

4.1.1 Accumulators (A, B, and D)

Accumulators A and B are general-purpose 8-bit accumulators used to hold operands and results of arithmetic calculations or data manipulations. Some instructions treat the

combination of these two 8-bit accumulators as a 16-bit double accumulator (accumulator D).

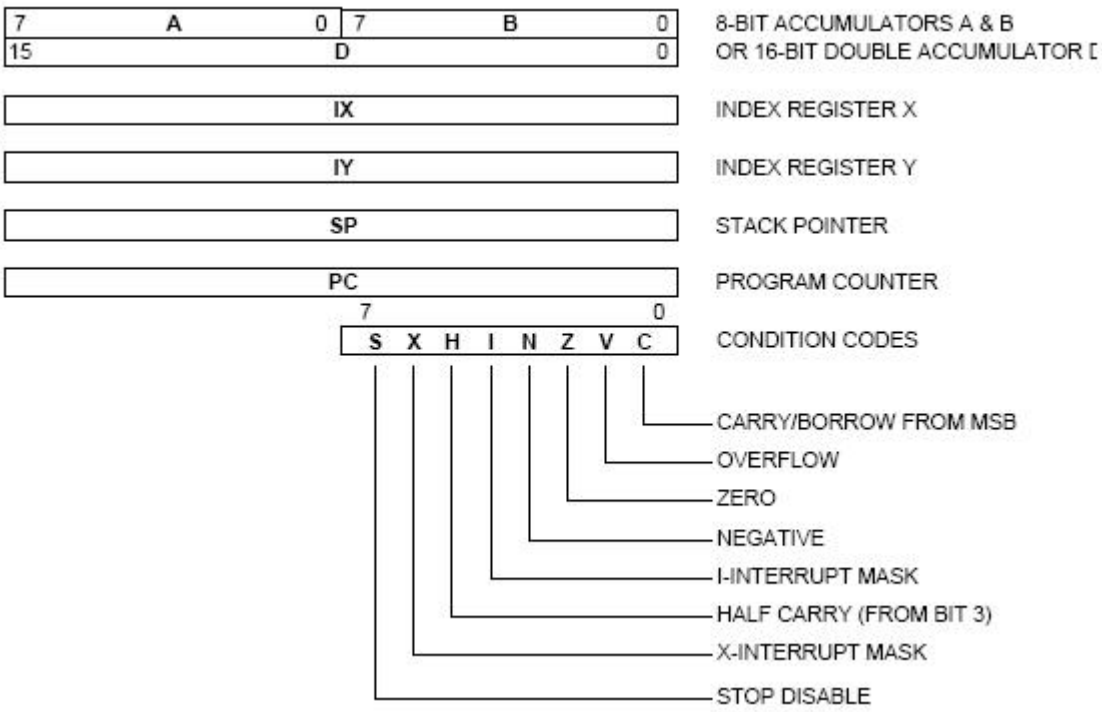

Figure 4.1 M68HC11 Programmer's Model [2]

Most operations can use accumulator A or B interchangeably; however, there are a few notable exceptions. The ABX and ABY instructions add the contents of the 8-bit accumulator B to the contents of the 16-bit index register X or Y, and there are no equivalent instructions that use A instead of B. The TAP and TPA instructions are used to transfer data from accumulator A to the condition code register or from the condition code register to accumulator A; however, there are no equivalent instructions that use B rather than A. The decimal adjust accumulator A (DAA) instruction is used after binary- coded decimal (BCD) arithmetic operations, and there is no equivalent BCD instruction to adjust B. Finally, the add, subtract, and compare instructions involving both A and B (ABA, SBA, and CBA) only operate in one direction; therefore, it is important to plan ahead so the correct operand will be in the correct accumulator.

4.1.2 Index Registers (X and Y)

The 16-bit index registers X and Y are used for indexed addressing mode. In the indexed addressing mode, the contents of a 16-bit index register are added to an 8-bit offset, which is included as part of the instruction, to form the effective address of the operand to be used

in the instruction. In most cases, instructions involving index register Y take one extra byte of object code and one extra cycle of execution time compared to the equivalent instruction using index register X. The second index register is especially useful for moves and in cases where operands from two separate tables are involved in a calculation. In the earlier M6800 and M6801, the programmer had to store the index to some temporary location so the second index value could be loaded into the index register.

The ABX and ABY instructions along with increment and decrement instructions allow some arithmetic operations on the index registers, but, in some cases, more powerful calculations are needed. The exchange instructions, XGDX and XGDY, offer a very simple way to load an index value into the 16-bit double accumulator, which has more powerful arithmetic capabilities than the index registers themselves.

It is very common to load one of the index registers with the beginning address of the internal register space (usually $1000), which allows the indexed addressing mode to be used to access any of the internal I/O and control registers. Indexed addressing requires fewer bytes of object code than the corresponding instruction using extended addressing. Perhaps a more important argument for using indexed addressing to access register space is that bit-manipulation instructions are available for indexed addressing but not for extended addressing.

4.1.3 Stack Pointer (SP)

The M68HC11 CPU automatically supports a program stack. This stack may be located anywhere in the 64-Kbyte address space and may be any size up to the amount of memory available in the system. Normally, the stack pointer register is initialized by one of the very first instructions in an application program. Each time a byte is pushed onto the stack, the stack pointer is automatically decremented, and each time a byte is pulled off the stack, the stack pointer is automatically incremented. At any given time, the stack pointer register holds the 16-bit address of the next free location on the stack. The stack is used for subroutine calls, interrupts, and for temporary storage of data values.

When a subroutine is called by a jump to subroutine (JSR) or branch to subroutine (BSR) instruction, the address of the next instruction after the JSR or BSR is automatically pushed onto the stack (low half first). When the subroutine is finished, a return from subroutine (RTS) instruction is executed. The RTS causes the previously stacked return address to be pulled off the stack, and execution continues at this recovered return address.

Whenever an interrupt occurs (provided it is not masked), the current instruction finishes normally, the address of the next instruction (the current value in the program counter) is pushed onto the stack, all of the CPU registers are pushed onto the stack, and execution continues at the address specified by the vector for the highest priority pending interrupt. After completing the interrupt service routine, a return from interrupt (RTI) instruction is executed. The RTI instruction causes the saved registers to be pulled off the stack in reverse order, and program execution resumes as if there had been no interruption.

Another common use for the stack is for temporary storage of register values. A simple example would be a subroutine using accumulator A. The user could push accumulator A onto the stack when entering the subroutine and pull it off the stack just before leaving the subroutine. This method is a simple way to assure a register(s) will be the same after returning from the subroutine as it was before starting the subroutine.

The most important aspect of the stack is that it is completely automatic. A programmer does not normally have to be concerned about the stack other than to be sure that it is pointing at usable random-access memory (RAM) and that there is sufficient space. To assure sufficient space, the user would need to know the maximum depth of subroutine or interrupt nesting possible in the particular application.

There are a few less common uses for the stack. The stack can be used to pass parameters to a subprogram, which is fairly common in high-level language compilers but is often overlooked by assembly-language programmers. There are two advantages of this technique over specific assignment of temporary or variable locations. First, the memory locations are only needed for the time the subprogram is being executed; they can be used for something else when the subprogram is completed. Second, this feature makes the subprogram re-entrant so that an interrupting program could call the same subprogram with a different set of values without disturbing the interrupted use of the subprogram.

In unusual cases, a programmer may want to look at or even manipulate something that is on the stack, which should only be attempted by an experienced programmer because it requires a detailed understanding of how the stack operates. Monitor programs like BUFFALO sometimes place items on a stack manually and then perform an RTI instruction to go to a user program. This technique is an odd use of the stack and RTI instruction because an RTI would normally correspond to a previous interrupt.

4.1.4 Program Counter (PC)

The program counter is a 16-bit register that holds the address of the next instruction to be executed.

4.1.5 Condition Code Register (CCR)

This register contains five status indicators, two interrupt masking bits, and a STOP disable bit. The register is named for the five status bits since that is the major use of the register. In the earlier M6800 and M6801 CPUs, there was no X interrupt mask and no STOP disable control in this register.

The five status flags reflect the results of arithmetic and other operations of the CPU as it performs instructions. The five flags are half carry (H), negative (N), zero (Z), overflow (V), and carry/borrow (C). The half-carry flag, which is used only for BCD arithmetic operations, is only affected by the add accumulators A and B (ABA), ADD, and add with carry (ADC) addition instructions (21 opcodes total). The N, Z, V, and C status bits allow for branching based on the results of a previous operation. Simple branches are included for either state of any of these four bits. Both signed and unsigned versions of branches are provided for the conditions <, ≤, =, ≠, ≥, or >.

The H bit indicates a carry from bit 3 during an addition operation. This status indicator allows the CPU to adjust the result of an 8-bit BCD addition so it is in correct BCD format, even though the add was a binary operation. This H bit, which is only updated by the ABA, ADD, and ADC instructions, is used by the DAA instruction to compensate the result in accumulator A to correct BCD format.

The N bit reflects the state of the most significant bit (MSB) of a result. For two's complement, a number is negative when the MSB is set and positive when the MSB is zero. The N bit has uses other than in two's-complement operations. By assigning an often tested flag bit to the MSB of a register or memory location, the user can test this bit by loading an accumulator.

The Z bit is set when all bits of the result are zeros. Compare instructions do an internal implied subtraction, and the condition codes, including Z, reflect the results of that subtraction. A few operations (INX, DEX, INY, and DEY) affect the Z bit and no other condition flags. For these operations, the user can only determine = and ≠.

The V bit is used to indicate if a two's-complement overflow has occurred as a result of the operation.

The C bit is normally used to indicate if a carry from an addition or a borrow has occurred as a result of a subtraction. The C bit also acts as an error flag for multiply and divide operations. Shift and rotate instructions operate with and through the carry bit to facilitate multiple-word shift operations.

In the M68HC11 CPU, condition codes are automatically updated by almost all instructions; thus, it is rare to execute any extra instructions to specifically update the condition codes. For example, the load accumulator A (LDAA) and store accumulator A (STAA) instructions automatically set or clear the N, Z, and V condition code flags. (In some other architectures, very few instructions affect the condition code bits; thus, it takes two instructions to load and test a variable.) The challenge in a Motorola processor lies in finding instructions that specifically do not alter the condition codes in rare cases where that is desirable. The most important instructions that do not alter conditions codes are the pushes, pulls, add B to X (ABX), add B to Y (ABY), and 16-bit transfers and exchanges.

The STOP disable (S) bit is used to allow or disallow the STOP instruction. Some users consider the STOP instruction dangerous because it causes the oscillator to stop; however, the user can set the S bit in the CCR to disallow the STOP instruction. If the STOP instruction is encountered by the CPU while the S bit is set, it will be treated like a no-operation (NOP) instruction, and processing continues to the next instruction.

The interrupt request (IRQ) mask (I bit) is a global mask that disables all maskable interrupt sources. While the I-bit is set, interrupts can become pending and are remembered, but CPU operation continues uninterrupted until the I-bit is cleared. After any reset, the I-bit is set by default and can only be cleared by a software instruction. When any interrupt occurs, the I-bit is automatically set after the registers are stacked but before the interrupt vector is fetched. After the interrupt has been serviced, an RTI instruction is normally executed, restoring the registers to the values that were present before the interrupt occurred. Normally, the I-bit would be zero after an RTI was executed. Although interrupts can be re-enabled within an interrupt service routine, to do so is unusual because nesting of interrupts becomes possible, which requires much more programming care than single-level interrupts and seldom improves system performance.

The XIRQ mask (X bit) is used to disable interrupts from the XIRQ pin. After any reset, X is set by default and can only be cleared by a software instruction. When XIRQ is recognized, the X bit (and I bit) are automatically set after the registers are stacked but before the

interrupt vector is fetched. After the interrupt has been serviced, an RTI instruction is normally executed, causing the registers to be restored to the values that were present before the interrupt occurred. It is logical to assume the X bit was clear before the interrupt; thus, the X bit would be zero after the RTI was executed. Although XIRQ can be re-enabled within an interrupt service routine, to do so is unusual because nesting of interrupts becomes possible, which requires much more programming care than single-level interrupts.

4.2 Addressing Modes

In the M68HC11 CPU, six addressing modes can be used to reference memory: immediate, direct, extended, indexed (with either of two 16-bit index registers and an 8- bit offset), inherent, and relative. Some instructions require an additional byte (a prebyte) before the opcode to accommodate a multiple-page opcode map.

Each of the addressing modes (except inherent) results in an internally generated, double-byte value referred to as the effective address. This value, which is the result of a statement operand field, is the value that appears on the address bus during the memory reference portion of the instruction. The addressing mode is an implicit part of every M68HC11 instruction.

Bit-manipulation instructions actually employ two or three addressing modes during execution but are classified by the addressing mode used to access the primary operand. All bit-manipulation instructions use immediate addressing to fetch a bit mask, and branch variations use relative addressing mode to determine a branch destination.

The following paragraphs provide a description of each addressing mode. In these descriptions, effective address is used to indicate the memory address from which the argument is fetched or stored or from which execution is to proceed.

4.2.1 Immediate (IMM)

In the immediate addressing mode, the actual argument is contained in the byte(s) immediately following the instruction in which the number of bytes matches the size of the register. These instructions are two, three, or four (if prebyte is required) bytes.

Machine-code byte(s) that follow the opcode are the value of the statement rather than the address of a value. In this case, the effective address of the instruction is specified by the character # sign and implicitly points to the byte following the opcode. The immediate value

is limited to either one or two bytes, depending on the size of the register involved in the instruction. Examples of several assembly-language statements using the immediate addressing mode are shown. Symbols and expression used in these statements are defined immediately after the examples.

The first three statements are assembler directives that set up values to be used in the remaining statements. The remaining nine statements are examples of immediate addressing. The value of each statement operand field appears in byte(s) immediately following the opcode. The operand field for immediate addressing begins with the character # sign. The character # sign is used by the assembler to detect the immediate mode of addressing. A very common programming error is to forget this character # sign.

Machine Code	**Label**	**Operation**	**Operand**	**Comments**
	CAT	EQU	7	CAT SAME AS 7
	ORG		$1000	SET LOCATION COUNTER
	REGS	EQU	*	ADDR(REGS) IS $1000
86 16		LDAA	#22	DECIMAL 22 ⇒ACCA ($16)
C8 34		EORB	#$34	XOR ($34,ACCB) ⇒ACCB
81 24		CMPA	#%100100	RIGHT ALIGNED BINARY
86 07		LDAA	#CAT	7 ⇒ACCA
CC 12 34		LDD	#$1234	
CC 00 07		LDD	#7	7 ⇒ACCA:ACCB
86 12		LDAA	#@22	OCTAL
86 41		LDAA	#'A	ASCII
CE 10 00		LDX	#REGS	ADDR(REGS) ⇒X

A variety of symbols and expressions can be used following the character # sign. Since not all assemblers use the same rules of syntax and special characters, the user should refer to the documentation for the particular assembler that will be used. Character prefixes used in the previous example statements are defined as follows:

Prefix	**Definition**
None	Decimal
$	Hexadecimal
@	Octal

% Binary

’ Single ASCII Character

4.2.2 Extended (EXT)

In the extended addressing mode, the effective address of the instruction appears explicitly in the two bytes following the opcode. Therefore, the length of most instructions using the extended addressing mode is three bytes: one for the opcode and two for the effective address. The last two bytes of the instruction contain the absolute address of the operand. These instructions are three or four (if prebyte is required) bytes: one or two for the opcode and two for the effective address. Instructions from the second, third, and fourth opcode map pages require a page-select prebyte prior to the opcode byte. Only four extended addressing mode instructions involving index register Y require this extra prebyte.

Examples of assembly-language statements that use extended addressing mode are grouped with direct addressing mode examples and appear after the discussion of the direct addressing mode.

4.2.3 Direct (DIR)

In the direct addressing mode, the least significant byte of the effective address of the instruction appears in the byte following the opcode. The high-order byte of the effective address is assumed to be $00 and is not included as an instruction byte (saves program memory space and execution time). This fact limits the use of direct addressing mode to operands in the $0000–$00FF area of memory (called the direct page). The direct addressing mode is sometimes called zero-page addressing mode. The length of most instructions using the direct addressing mode is two bytes: one for the opcode and one for the effective address. Instructions from the second, third, and fourth opcode-map pages require a page-select prebyte prior to the opcode byte. Only four direct addressing mode instructions involving index register Y require this extra prebyte.

Direct addressing allows the user to access $0000–$00FF, using instructions that take one less byte of program memory space than the equivalent instructions using extended addressing. By eliminating the additional memory access, execution time is reduced by one cycle. In the course of a large program, this savings can be substantial. For most applications, the default memory map of the microcontroller unit (MCU), which places internal random-access memory (RAM) in the $0000–$00FF area, is a good choice because the designer can assign these locations to frequently referenced data variables. In some MCU applications, it is desirable to locate the internal registers in this premium

memory space. This arrangement might be desirable in an I/O-intensive application in which the program space savings are important or in the case of some very critical timing requirement in which the extra cycle for extended addressing mode is undesirable. In the M68HC11 MCU, software can configure the memory map so that internal RAM, and/or internal registers, or external memory space can occupy these addresses.

There are some instructions that provide for extended addressing mode but not direct addressing mode. These instructions, which are members of a group called read-modify-write instructions, operate directly on memory (opcodes $40–$7F except jump (JMP) and test for zero or minus (TST) on all opcode pages) and have the following form:

<operation>M ⇒M

The increment memory byte (INC), decrement memory byte (DEC), clear memory byte (CLR), and one's complement memory byte (COM) instructions are members of this group, and each supports extended addressing mode but not direct addressing mode. The following example shows the direct and extended addressing modes.

Machine Code	**Label**	**Operation**	**Operand**	**Comments**
B3 00 12		SUBD	CAT	FWD REF TO CAT
	CAT	EQU	$12	DEFINE CAT=$12
93 12		SUBD	CAT	BKWD REF TO CAT
7F 00 12		CLR	CAT	EXTENDED ONLY

In the previous example, the first reference to the CAT label is a forward reference, and the assembler selected the extended addressing mode. The second reference, which is a backward reference, enabled the assembler to know the symbol value when processing the statement, and the assembler selected the direct addressing mode. The last reference to CAT is also a backward reference to a symbol in the direct addressing area, but the extended addressing mode was selected because there is no direct addressing mode variation of that particular instruction. Some assemblers allow the direct or extended addressing modes to be forced (by preceding the operand field with < or >, respectively), even when other conditions would suggest the other mode.

4.2.4 Indexed (INDX, INDY)

In the indexed addressing mode, either index register X or Y is used in calculating the effective address. In this case, the effective address is variable and depends on the current

contents of index register X or Y and a fixed, 8-bit, unsigned offset contained in the instruction. This addressing mode can be used to reference any memory location in the 64-Kbyte address space. These instructions are usually two or three bytes (if prebyte is required) — the opcode and the 8-bit offset.

In microprocessor-based systems, instructions usually reside in read-only memory (ROM). Therefore, the offset in the instruction should be considered a fixed value that is determined at assembly time rather than during program execution. The use of dynamic single-byte offsets is facilitated with the use of the add accumulator B to index register X (ABX) instruction. More complex address calculations are aided by the arithmetic capability of the 16-bit accumulator D and the XGDX and XGDY instructions.

If no offset is specified or desired, the machine code will contain $00 in the offset byte. The offset is an unsigned single-byte value that, when added to the current value in the index register, yields the effective address of the operand, leaving the index register unchanged. Because the offset byte is unsigned, only positive offsets in the range 0–255 can be specified. To use the indexed addressing mode to access on-chip registers in the MC68HC11, it is best to initialize the index register to the starting address of the register block (usually $1000) and use an 8-bit offset ($00–$3F) in the instructions that access registers. This method is preferred over loading the index register with the 16-bit address of a register and then specifying a zero offset in the instruction. This latter method requires modification of the index register for each register access; whereas, the former method does not.

Examples of the indexed addressing mode are shown (EA indicates effective address):

Machine Code	**Label**	**Operation**	**Operand**	**Comments**
E3 00		ADDD	X	EA=(X)
E3 00		ADDD ,X		EA=(X)
E3 00		ADDD	0,X	EA=(X)
E3 04		ADDD	4,X	EA=(X)+4
	CAT	EQU	7	DEFINE CAT=7
E3 07		ADDD	CAT,X	EA=(X)+7
E3 22		ADDD	$22,X	EA=(X)+$22
E3 22		ADDD	CAT*8/2+6,X	EA=(X)+(CAT*8÷2+6)

Bit-manipulation instructions support direct and indexed addressing modes but not extended addressing mode. The indexed addressing mode becomes very important for these instructions because the direct addressing mode only permits access to the first 256

memory locations; whereas, the indexed addressing mode allows access to any memory location in the 64-Kbyte memory map.

The second index register (Y) improves the efficiency of move operations and operations involving data from more than one table. Most instructions involving index register Y require two-byte opcodes, thus requiring one extra byte of program memory space and one extra cycle of execution time compared to the equivalent index register X instruction.

4.2.5 Inherent (INH)

In the inherent addressing mode, everything needed to execute the instruction is inherently known by the CPU. The operands (if any) are CPU registers and thus are not fetched from memory. These instructions are usually one or two bytes.

Many M68HC11 MCU instructions use one or more registers as operands. For instance, the ABA instruction causes the CPU to add the contents of accumulators A and B and place the result in accumulator A. The INCB instruction causes the contents of accumulator B to be incremented by one. Similarly, the INX instruction causes the index register X to be incremented by one. These three assembly-language statements are examples of the inherent addressing mode:

Machine Code	Label	Operation	Operand	Comments
1B		ABA		A+B→A
5C		INCB		B+1→B
08		INX		X+1→X

4.2.6 Relative (REL)

The relative addressing mode is used only for branch instructions. Branch instructions, other than the branching versions of bit-manipulation instructions, generate two machine code bytes: one for the opcode and one for the relative offset. Because it is desirable to branch in either direction, the offset byte is a signed two's-complement offset with a range of –128 to +127 bytes (with respect to the address of the instruction immediately following the branch instruction). If the branch condition is true, the contents of the 8-bit signed byte following the opcode (offset) are added to the contents of the program counter to form the effective branch address; otherwise, control proceeds to the instruction immediately following the branch instruction.

The offset byte is always the last byte of a branch instruction. If the offset byte is zero, execution will proceed to the instruction immediately following the branch instruction, regardless of the test involved. A branch always (BRA) instruction with an offset of $FE will result in an infinite loop back to itself. Direct or indexed X addressing mode branch if bit clear (BRCLR) and branch if bit set (BRSET) instructions are four-byte instructions; therefore, an offset byte of $FC will cause the instruction to execute repeatedly until the bit test becomes false. Indexed Y addressing mode BRCLR and BRSET instructions are five-byte instructions; thus, an offset byte of $FB will cause the instruction to execute repeatedly until the bit test becomes false.

Examples of the relative addressing mode are shown in the following assembly-language statements:

Machine Code	**Label**	**Operation**	**Operand**	**Comments**
20 00	THERE	BRA	WHERE	FORWARD BRANCH
22 FC	WHERE	BHI	THERE	BACKWARD BRANCH
24 04		BCC	LBCC	L-O-N-G BCC
27 FE	HANG	BEQ	HANG	BRANCH TO SELF
27 FE		BEQ	*	"*" MEANS "HERE"
7E 10 00	LBCC	JMP	$1000	
8D F7		BSR	HANG	

4.3 M68HC11 Instruction Set

The instruction set is divided into functional groups of instructions. Some instructions will appear in more than one functional group. For example, transfer accumulator A to CCR (TAP) appears in the CCR group and in the load/store/transfer subgroup of accumulator/memory instructions.

To expand the number of instructions used in the M68HC11 CPU, a prebyte mechanism that affects certain instructions has been added. Most of the instructions affected are associated with index register Y. Instructions that do not require a prebyte reside in page 1 of the opcode map. Instructions requiring a prebyte reside in pages 2, 3, and 4 of the opcode map. The opcode-map prebyte codes are $18 for page 2, $1A for page 3, and $CD for page 4. A prebyte code applies only to the opcode immediately following it. That is, all instructions are assumed to be single-byte opcodes unless the first byte of the instruction happens to correspond to one of the three prebyte codes rather than a page 1 opcode.

4.3.1 Accumulator and Memory Instructions

Most of these instructions use two operands. One operand is either an accumulator or an index register; whereas, the second operand is usually obtained from memory using one of the addressing modes discussed earlier. These accumulator memory instructions can be divided into six subgroups: 1) loads, stores, and transfers, 2) arithmetic operations, 3) multiply and divide, 4) logical operations, 5) data testing and bit manipulation, and 6) shifts and rotates. These instructions are discussed in the following tables and paragraphs.

4.3.1.1 Loads, Stores, And Transfers

Almost all MCU activities involve transferring data from memories or peripherals into the CPU or transferring results from the CPU into memory or I/O devices. The load, store, and transfer instructions associated with the accumulators are summarized in the following table. There are additional load, store, push, and pull instructions associated with the index registers and stack pointer.

Table 4.1 Loads, Stores, And Transfers [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Clear Memory Byte	CLR			X	X	X	
Clear Accumulator A	CLRA						X
Clear Accumulator B	CLRB						X
Load Accumulator A	LDAA	X	X	X	X	X	
Load Accumulator B	LDAB	X	X	X	X	X	
Load Double Accumulator D	LDD	X	X	X	X	X	
Pull A from Stack	PULA						X
Pull B from Stack	PULB						X
Push A onto Stack	PSHA						X
Push B onto Stack	PSHB						X
Store Accumulator A	STAA	X	X	X	X	X	
Store Accumulator	STAB	X	X	X	X	X	
Store Double Accumulator D	STD	X	X	X	X	X	
Transfer A to B	TAB						X
Transfer A to CCR	TAP						X
Transfer B to A	TBA						X
Transfer CCR to A	TPA						X
Exchange D with X	XGDX						X
Exchange D with Y	EGDY						X

4.3.1.2 Arithmetic Operations

This group of instructions supports arithmetic operations on a variety of operands; 8- and 16-bit operations are supported directly and can easily be extended to support multiple-word operands. Two's-complement (signed) and binary (unsigned) operations are supported directly. BCD arithmetic is supported by following normal arithmetic instruction sequences, using the DAA instruction, which restores results to BCD format. Compare instructions perform a subtract within the CPU to update the condition code bits without altering either operand. Although test instructions are provided, they are seldom needed since almost all other operations automatically update the condition code bits.

Table 4.2 Arithmetic Operations [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulators	ABA						X
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Add with Carry to A	ADCA	X	X	X	X	X	
Add with Carry to B	ADCB	X	X	X	X	X	
Add Memory to A	ADDA	X	X	X	X	X	
Add Memory to B	ADDB	X	X	X	X	X	
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X	
Compare A to B	CBA						X
Compare A to Memory	CMPA	X	X	X	X	X	
Compare B to Memory	CMPB	X	X	X	X	X	
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X	
Decimal Adjust A (for BCD)	DAA						X
Decrement Memory Byte	DEC			X	X	X	
Decrement Accumulator A	DECA						X
Decrement Accumulator B	DECB						X
Increment Memory Byte	INC			X	X	X	
Increment Accumulator A	INCA						X
Increment Accumulator B	INCB						X
Two’s Complement Memory Byte	NEG			X	X	X	
Two’s Complement Accumulator A	NEGA						
Two’s Complement Accumulator B	NEGB						
Subtract with Carry from A	SBCA	X	X	X	X	X	
Subtract with Carry from B	SBCB	X	X	X	X	X	
Subtract Memory from A	SUBA	X	X	X	X	X	
Subtract Memory from B	SUBB	X	X	X	X	X	
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X	
Test for Zero or Minus	TST			X	X	X	
Test for Zero or Minus A	TSTA						X
Test for Zero or Minus B	TSTB						X

4.3.1.3 Multiply and Divide

One multiply and two divide instructions are provided. The 8-bit by 8-bit multiply produces a 16-bit result. The integer divide (IDIV) performs a 16-bit by 16-bit divide, producing a 16-bit result and a 16-bit remainder. The fractional divide (FDIV) divides a 16-bit numerator by a larger 16-bit denominator, producing a 16bit result (a binary weighted fraction between 0 and 0.99998) and a 16-bit remainder. FDIV can be used to further resolve the remainder from an IDIV or FDIV operation.

Table 4.3 Multiply and Divide [2]

Function	Mnemonic	INH
Multiply (A X B →D)	MUL	X
Fractional Divide (D ÷ X →X; r →D)	FDIV	X
Integer Divide (D ÷X →X; r →D)	IDIV	X

4.3.1.4 Logical Operations

This group of instructions is used to perform the boolean logical operations AND, inclusive OR, exclusive OR, and one's complement.

Table 4.4 Logical Operations [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
AND A with Memory	ANDA	X	X	X	X	X	
AND B with Memory	ANDB	X	X	X	X	X	
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
One's Complement Memory Byte	COM			X	X	X	
One's Complement A	COMA						X
One's Complement B	COMB						X
OR A with Memory (Exclusive)	EORA	X	X	X	X	X	
OR B with Memory (Exclusive)	EORB	X	X	X	X	X	
OR A with Memory (Inclusive)	ORAA	X	X	X	X	X	
OR B with Memory (Inclusive)	ORAB	X	X	X	X	X	

4.3.1.5 Data Testing and Bit Manipulation

This group of instructions is used to operate on operands as small as a single bit, but these instructions can also operate on any combination of bits within any 8-bit location in the 64-Kbyte memory space. The bit test (BITA or BITB) instructions perform an AND operation within the CPU to update condition code bits without altering either operand. The BSET and BCLR instructions read the operand, manipulate selected bits within the operand, and write the result back to the operand address. Some care is required when read-modify-write instructions such as BSET and BCLR are used on I/O and control register locations because the physical location read is not always the same as the location written.

Table 4.5 Data Testing and Bit Manipulation [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY
Bit(s) Test A with Memory	BITA	X	X	X	X	X
Bit(s) Test B with Memory	BITB	X	X	X	X	X
Clear Bit(s) in Memory	BCLR		X		X	X
Set Bit(s) in Memory	BSET		X		X	X
Branch if Bit(s) Clear	BRCLR		X		X	X
Branch if Bit(s) Set	BRSET		X		X	X

4.3.1.6 Shifts and Rotates

All the shift and rotate functions in the M68HC11 CPU involve the carry bit in the CCR in addition to the 8- or 16-bit operand in the instruction, which permits easy extension to multiple-word operands. Also, by setting or clearing the carry bit before a shift or rotate instruction, the programmer can easily control what will be shifted into the opened end of an operand. The arithmetic shift right (ASR) instruction maintains the original value of the MSB of the operand, which facilitates manipulation of two's-complement (signed) numbers.

Table 4.6 Shifts and Rotates [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Arithmetic Shift Left Memory	ASL			X	X	X	
Arithmetic Shift Left A	ASLA						X
Arithmetic Shift Left B	ASLB						X
Arithmetic Shift Left Double	ASLD						X
Arithmetic Shift Right Memory	ASR			X	X	X	
Arithmetic Shift Right A	ASRA						X
Arithmetic Shift Right	ASRB						X
(Logical Shift Left Memory)	(LSL)			X	X	X	
(Logical Shift Left A)	(LSLA)						X
(Logical Shift Left B)	(LSLB)						X
(Logical Shift Left Double)	(LSLD)						X
Logical Shift Right Memory	LSR			X	X	X	
Logical Shift Right A	LSRA						X
Logical Shift Right B	LSRB						X
Logical Shift Right D	LSRD						X
Rotate Left Memory	ROL			X	X	X	
Rotate Left A	ROLA						X
Rotate Left B	ROLB						X
Rotate Right Memory	ROR			X	X	X	
Rotate Right A	RORA						X
Rotate Right B	RORB						X

The logical-left-shift instructions are shown in parentheses because there is no difference between an arithmetic and a logical left shift. Both mnemonics are recognized by the assembler as equivalent, but having both instruction mnemonics makes some programs easier to read.

4.3.2 Stack and Index Register Instructions

The following table summarizes the instructions available for the 16-bit index registers (X and Y) and the 16-bit stack pointer.

Table 4.7 Stack and Index Register Instructions [2]

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Compare X to Memory (16 Bit)	CPX	X	X	X	X	X	
Compare Y to Memory (16 Bit)	CPY	X	X	X	X	X	
Decrement Stack Pointer	DES						X
Decrement Index Register X	DEX						X
Decrement Index Register Y	DEY						X
Increment Stack Pointer	INS						X
Increment Index Register X	INX						X
Increment Index Register Y	INY						X
Load Index Register X	LDX	X	X	X	X	X	
Load Index Register Y	LDY	X	X	X	X	X	
Load Stack Pointer	LDS	X	X	X	X	X	
Pull X from Stack	PULX						X
Pull Y from Stack	PULY						X
Push X onto Stack	PSHX						X
Push Y onto Stack	PSHY						X
Store Index Register X	STX	X	X	X	X	X	
Store Index Register Y	STY	X	X	X	X	X	
Store Stack Pointer	STS	X	X	X	X	X	
Transfer SP to X	TSX						X
Transfer SP to Y	TSY						X
Transfer X to SP	TXS						X
Transfer Y to SP	TYS						X
Exchange D with X	XGDX						X
Exchange D with Y	XGDY						X

The exchange D with X (XGDX) and exchange D with Y (XGDY) provide a simple way of transferring a pointer value from a 16-bit index register to accumulator D, which has more powerful 16-bit arithmetic capabilities than the 16-bit index registers. Since these are bidirectional exchanges, the original value of accumulator D is automatically preserved in the index register while the pointer is being manipulated in accumulator D. When pointer calculations are finished, another exchange simultaneously updates the index register and restores accumulator D to its former value.

The transfers between an index register and the stack pointer deserve additional comment. The stack pointer always points at the next free location on the stack as opposed to the last item that was pushed onto the stack. The usual reason for transferring the stack pointer value into an index register is to allow indexed addressing access to information that was formerly pushed onto the stack. In such cases, the address pointed to by the stack pointer is of no value since nothing has yet been stored at that location. This fact explains why the value in the stack pointer is incremented during transfers to an index register. There is a corresponding decrement of a 16-bit value as it is transferred from an index register to the stack pointer.

4.3.3 Condition Code Register Instructions

These instructions allow a programmer to manipulate bits in the CCR.

Table 4.8 Condition Code Register Instructions [2]

Function	Mnemonic	INH
Clear Carry Bit	CLC	X
Clear Interrupt Mask Bit	CLI	X
Clear Overflow Bit	CLV	X
Set CarryBit	SEC	X
Set Interrupt Mask Bit	SEI	X
Set Overflow Bit	SEV	X
Transfer A to CCR	TAP	X
Transfer CCR to A	TPA	X

Initially, it may appear that there should be a set and a clear instruction for each of the eight bits in the CCR; however, these instructions are present for only three of the eight bits (C, I, and V). Upon closer consideration, good reasons exist for not including the set and clear instructions for the other five bits. The stop disable (S) bit is an unusual case because this bit is intended to lock out the STOP instruction for those who view it as an undesirable function in their application. Providing set and clear instructions for this bit would make it easier to enable STOP when it was not wanted or disable STOP when it was wanted. The TAP instruction provides a way to change the S bit but reduces the chance of an undesirable change to S because the value of accumulator A at the time the TAP instruction is executed determines whether the S bit will actually change.

The XIRQ mask (X bit) is another unusual case. The definition of this bit specifically states that software shall not be allowed to change X from zero to one; in fact, this change is even

prohibited by hardware logic. This feature immediately eliminates a need for a set X instruction. For arguments similar to those used for the S bit, the TAP instruction is preferred over a clear X instruction to clear X because TAP makes it a little less likely that X will become cleared before the programmer intended.

The half-carry (H) bit needs no set or clear instructions because this condition code bit is only used by the DAA instruction to adjust the result of a BCD add or subtract. Since the H bit is not used as a test condition for any branches, it would not be useful to be able to set or clear this bit.

This leaves only the negative (N) and zero (Z) condition code bits. In contrast to S, X, and H, it is often useful to be able to easily set or clear these flag bits. A clear accumulator instruction, such as CLRB, will clear the N and set the Z condition code bits. The load instruction, LDAA#$80, causes N to be set and Z to be cleared. Since there are so many simple instructions that can set or clear N and Z, it is not necessary to provide specific set and clear instructions for N and Z in this group.

4.3.4 Program Control Instructions

This group of instructions, which is used to control the flow of a program rather than to manipulate data, has been divided into five subgroups: 1) branches, 2) jumps, 3) subroutine calls and returns, 4) interrupt handling, and 5) miscellaneous,

4.3.4.1 Branches

These instructions allow the CPU to make decisions based on the contents of the condition code bits. All decision blocks in a flow chart would correspond to one of the conditional branch instructions summarized in the following table.

Table 4.9 Branches [2]

Function	Mnemonic	REL	DIR	INDX	INDY	Comments
Branch if Carry Clear	BCC	X				C = 0 ?
Branch if Carry Set	BCS	X				C = 1 ?
Branch if Equal Zero	BEQ	X				Z = 1 ?
Branch if Greater Than or Equal	BGE	X				Signed ≥
Branch if Greater Than	BGT	X				Signed >
Branch if Higher	BHI	X				Unsigned >
Branch if Higher or Same (same as BCC)	BHS	X				Unsigned ≥
Branch if Less Than or Equal	BLE	X				Signed ≤
Branch if Lower (same as BCS)	BLO	X				Unsigned <
Branch if Lower or Same	BLS	X				Unsigned ≤
Branch if Less Than	BLT	X				Signed <
Branch if Minus	BMI	X				N= 1 ?
Branch if Not Equal	BNE	X				Z = 0 ?
Branch if Plus	BPL	X				N = 0 ?
Branch if Bit(s) Clear in Memory Byte	BRCLR		X	X	X	Bit Manipulation
Branch Never	BRN	X				3-cycle NOP
Branch if Bit(s) Set in Memory Byte	BRSET		X	X	X	Bit Manipulation
Branch if Overflow Clear	BVC	X				V = 0 ?
Branch if Overflow Set	BVS	X				V = 1 ?

The limited range of branches (–128/+127 locations) is more than adequate for most (but not all) situations. In cases where this range is too short, a jump instruction must be used. For every branch, there is a branch for the opposite condition; thus, it is simple to replace a branch having an out-of-range destination with a sequence consisting of the opposite branch around a jump to the out-of-range destination. For example, if a program contained the following instruction

BHI TINBUK2 Unsigned >

where TINBUK2 was out of the –128/+127 location range, the following instruction sequence could be substituted:

```
                BLS     AROUND Unsigned ≤
                JMP     TINBUK2 Still go to TINBUK2 if >
AROUND          EQU     *
```

4.3.4.2 Jumps

The jump instruction allows control to be passed to any address in the 64-Kbyte memory map.

Table 4.10 Jumps [2]

Function	Mnemonic	DIR	EXT	INDX	INDY	INH
Jump	JMP	X	X	X	X	

4.3.4.3 Subroutine Calls And Returns (BSR, JSR, RTS)

These instructions provide an easy way to divide a programming task into manageable blocks called subroutines. The CPU automates the process of remembering the address in the main program where processing should resume after the subroutine is finished. This address is automatically pushed onto the stack when the subroutine is called and is pulled off the stack during the RTS instruction that ends the subroutine.

Table 4.11 Subroutine Calls And Returns (BSR, JSR, RTS) [2]

Function	Mnemonic	REL	DIR	EXT	INDX	INDY	INH
Branch to Subroutine	BSR	X					
Jump to Subroutine	JSR		X	X	X	X	
Return from Subroutine	RTS						X

4.3.4.4 Interrupt Handling (RTI, SWI, WAI)

This group of instructions is related to interrupt operations.

Table 4.12 Interrupt Handling (RTI, SWI, WAI) [2]

Function	Mnemonic	INH
Return from Interrupt	RTI	X
Software Interrupt	SWI	X
Wait for Interrupt	WAI	X

The software interrupt (SWI) instruction is similar to a JSR instruction, except the contents of all working CPU registers are saved on the stack rather than just the return address. SWI

is unusual in that it is requested by the software program as opposed to other interrupts that are requested asynchronously to the executing program.

Wait for interrupt (WAI) has two main purposes. WAI is executed to place the MCU in a reduced power-consumption standby state (WAIT mode) until some interrupt occurs. It is also used to reduce the latency time to some important interrupt. The reduction of latency occurs because the time-consuming task of storing the CPU registers on the stack is performed as soon as the WAI instruction begins executing. When the interrupt finally occurs, the CPU is ready to fetch the appropriate vector so the delay associated with register stacking is eliminated from latency calculations.

4.3.4.5 Miscellaneous (NOP, STOP, TEST)

NOP, which can be used to introduce a small time delay into the flow of a program, is often useful in meeting the timing requirements of slow peripherals. By incorporating NOP instructions into loops, longer delays can be produced.

Table 4.13 Miscellaneous (NOP, STOP, TEST) [2]

Function	Mnemonic	INH
No Operation (2-cycle delay)	NOP	X
Stop Clocks	STOP	X
Test	TEST	X

During debugging, it is common to replace various instructions with NOP opcodes to effectively remove an unwanted instruction without having to rearrange the rest of the program. By using the memory modify function of a debug monitor, the instruction can easily be removed and restored to see the effect.

Occasionally, a programmer is faced with the problem of fine-tuning the delays through various paths in his program. In such cases, it is sometimes useful to use a branch never (BRN) instruction as a three-cycle NOP. It is also possible to fine-tune execution time by choosing alternate addressing-mode variations of instructions to change the execution time of an instruction sequence without changing the program's function.

STOP is an unusual instruction because it causes the oscillator and all MCU clocks to freeze. This frozen state is called STOP mode, and power consumption is dramatically reduced in this mode. The operation of this instruction is also dependent upon the S condition code bit because the STOP mode is not appropriate for all applications. If S is

one, the STOP instruction is treated as a NOP instruction, and processing continues to the next instruction.

The TEST instruction is used only during factory testing and is treated as an illegal opcode in normal operating modes of the MCU. This instruction causes unusual behavior on the address bus (counts backwards), which prevents its use in any normal system.

CHAPTER 5

SOFTWARE ARCHITECTURE

The source code is implemented by using C++ language. Borland C++ Builder 5.0 is used for compilation and linking. The source code of the application is given in APPENDIX B. Object orinted programming is used in order to use templates and class structures [1], [6].

Program can run on Windows 98 and higher operating systems. The program is started by running the 6811.exe file. Application creates a windows graphical user interface form.

In this chapter the architecture of the software implemented as the subject of this thesis study is explained in detail. All the objects, structures, functions and the excution sequence developed in the project are explained.

5.1 User Interface and Form Objects

Main form of the user interface is shown in Figure 5.1.

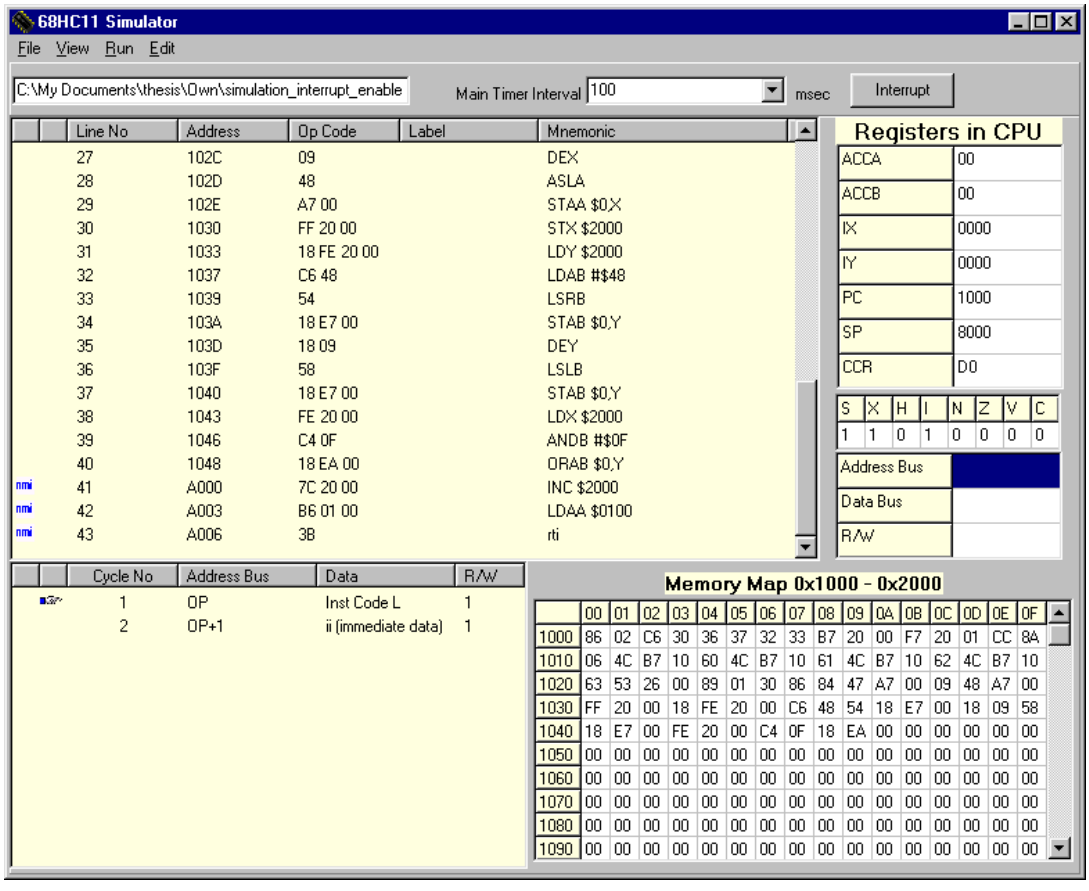

Figure 5.1 Main Form

In the form, File, View and Run main menus exist. From the file menu user can open an asm file for compilation and debugging. There exist two listviews on the main form. ListView object is used to manage and display a list of items in a form. The items can be displayed in columns with column headers and sub-items, vertically or horizontally, with small or large icons.

5.1.1 CodeView Object

The first ListView object (CodeView) in our study is used to display the lines in the asm file. The content of the asm file is not directly shown in the CodeView. During the processing of the file empty lines are removed; instructions and labels are parsed. Corresponding line numbers and the memory locations of the instructions and labels are inserted to the listview. In Figure 5.2, columns of the CodeView object are shown. First two columns are empty columns. These two columns are used to highlight the line which will be executed. Column with “Line No” header shows the corresponding line numbers in the asm file. The value shown in this column is the physical line number in the asm file for all the non-empty lines. “Address” column is used to show the starting memory location containing the instruction in this line. “Op Code” column is the hexadecimal equivalent of the instruction. “Label” column is used to show the label if there exists one for the corresponding line. “Mnemonic” column is used to show the alphanumeric representation of the instruction and its arguments.

Figure 5.2 CodeView Object

5.1.2 CycleView Object

Second listview (CycleView) is used to display the cycle list of the highlighted instruction in the first listview (CodeView). In Figure 5.3, columns of the CycleView object are shown. First two columns are empty columns. These two columns are used to highlight the cycle

which will be executed. Column with “Cycle No” header shows the corresponding cycle numbers for the corresponding instruction shown in CodeView object. “Cycle Operation” column is used to show the word definition of the operation processed in the cycle.

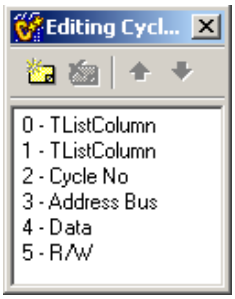

Figure 5.3 CycleView Object

There are three StringGrid objects on the main form. StringGrid object is used to present textual data in a tabular format.

1. RegisterGrid
2. CCRGrid
3. AddrDataGrid

5.1.3 RegisterGrid Object

This object is used to show the contents of the CPU registers during execution. Abbreviations “ACCA”, “ACCB”, “IX”, “IY”, “PC”,”SP” and “CCR” stand for accumulator A, accumulator B, index register X, index register Y, program counter, stack pointer and condition code register respectively. In Figure 5.4 content of the RegisterGrid object is shown.

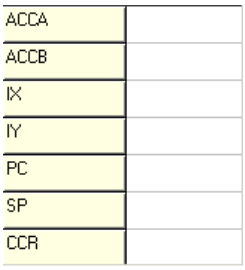

Figure 5.4 RegisterGrid Object

5.1.4 CCRGrid Object

This object is used to show the bit contents of the condition code register.

- S : Stop Disable, bit 7
- X : X Interrupt mask, bit 6
- H : Half carry, bit 5
- I : I interrupt mask, bit 4
- N : Negative Indicator, bit 3
- Z : Zero Indicator, bit 2
- V : Two's complement overflow indicator, bit 1
- C : Carry/Borrow, bit 0

In Figure 5.5 content of the CCRGrid object is shown.

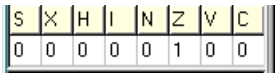

Figure 5.5 CCRGrid Object

5.1.5 AddrDataGrid Object

This object is used to show the contents of the address bus, data bus and R/W content. In Figure 5.6 content of the CCRGrid object is shown.

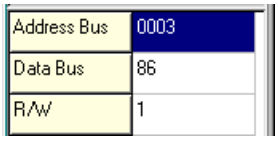

Figure 5.6 AddrDataGrid Object

There is an image object on the form. Image object is used to display a graphical image on a form.

5.1.6 AddrDataGUI Object

This object is used to show the contents of the address bus, data bus and R/W content. Labels are embedded on the image in order to show the dynamic data. Each bit of the address, data bus and R/W are represented with different labels. When the data content of the labels are changed the changed label's background color is changed to red during one cycle operation. In Figure 5.7 content of the AddrDataGUI object is shown.

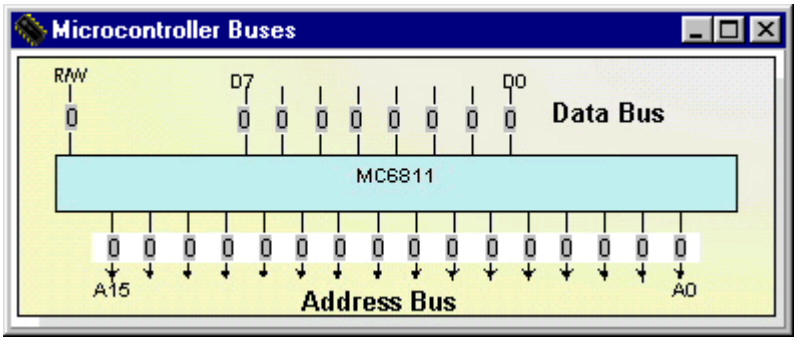

Figure 5.7 AddrDataGUI Object

There is a timer object on the form. Timer object is used to simplify calling the Windows API timer functions SetTimer and KillTimer, and to simplify processing the WM_TIMER messages.

5.1.7 ExecutionTimer Object

This object is enabled with timing interval 1 milliseconds at the beginning of the program execution. ExecutionTimer is never stopped. In each timer expiration ExecutionTimerTimer function is automatically executed. A state machine is created in this function. Following code stands for the state machine handler in the ExecutionTimerTimer function:

```
case TIMER_IDLE:
    break;
case TIMER_EXECUTE_ALL:
    ExecuteFile();
    break;
case TIMER_EXECUTE_INST:
    ExecuteCurrentPC();
    HighlightNextExecution(Motorola.PC.value());
    MainTimer.State=TIMER_EXECUTE_WAITKEY;
    break;
case TIMER_EXECUTE_CYCLE:
    ExecuteCurrentCycle();
    MainTimer.State=TIMER_EXECUTE_WAITKEY;
    break;
case TIMER_EXECUTE_WAITKEY:
    break;
```

In TIMER_IDLE state, nothing is done. This state represents the end of execution by the halt of the system. This state is entered by HALT or STOP instruction.

In TIMER_EXECUTE_ALL state, all the instructions are executed. This state is initiated by F9 key or main menu item “Run → Execute All”. Execution continues until the end of instruction list is reached or F9 key is stroked.
In TIMER_EXECUTE_INST state, current highlighted instruction is executed. This state is initiated by F8 key or main menu item “Run → Execute Instruction”. After the execution process timer state is changed to TIMER_EXECUTE_WAITKEY.
In TIMER_EXECUTE_CYCLE state, current highlighted cycle is executed. This state is initiated by F7 key or main menu item “Run → Execute Cycle”. After the execution process timer state is changed to TIMER_EXECUTE_WAITKEY.
In TIMER_EXECUTE_WAITKEY state, nothing is done. This state represents the system is waiting for a keystroke (F7, F8 or F9 or their corresponding menu items located in “Run” main menu).

Timer state flowchart is given below.

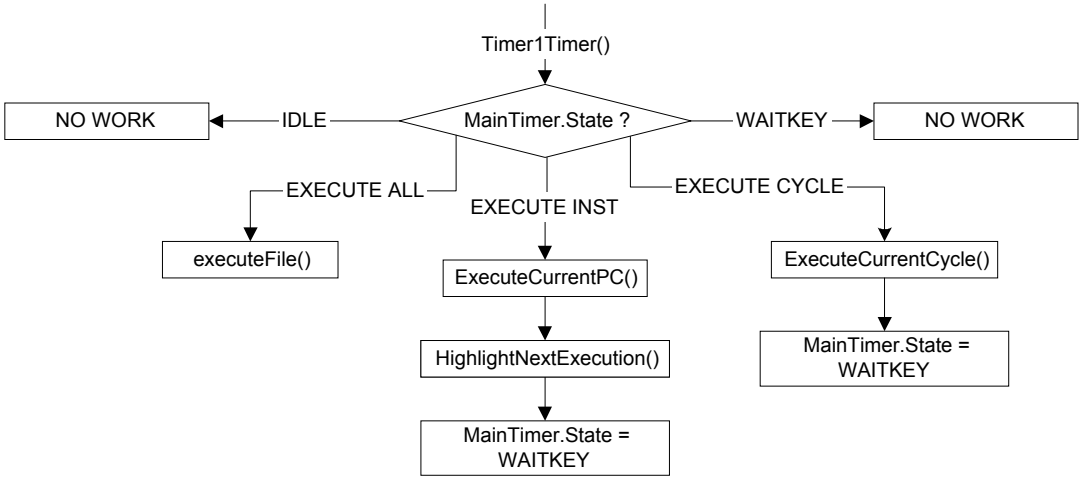

Figure 5.8 Timer State Flowchart

There is a MainMenu object on the form. MainMenu object is used to provide the main menu for a form. MainMenu introduces properties and methods for merging the drop-down menus of the main menu with the main menu of another form, and for assisting in the menu negotiation process of an OLE container.

5.1.8 MainMenu1 Object

This menu object consists “File”, “View”, “Run” and “Edit” items, which introduce user interface for the software. From “File “ menu asm file can be loaded. From “View” menu content of the whole memory or the asm file can be viewed. From “Run” menu execution

modes of the loaded asm file can be selected. Three different execution modes are available; Execution of all instructions, execution of single instruction and execution of single cycle. Shortcut keys are defined for these execution modes. Since main menu object is available over all the form, shortcut keystrokes are directed to the same functions independent of focused item. From the “Edit” menu item the stack pointer, start vector and interrupt vector of the microcontroller can be set to other values than the default ones.

There is an OpenDialog object on the form. OpenDialog displays a modal Windows dialog box for selecting and opening files. The dialog does not appear at runtime until it is activated by a call to the Execute method.

5.1.9 OpenDialog1 Object

When the open command from “File” menu is selected OpenDialog1 is executed with .asm and .txt file filters. By using this dialog user can browse through the folders and select the appropriate file for execution.

5.2 OBJECTS and STRUCTURES

This section introduces the objects and structures used in the project.

5.2.1 asmLine Object

This object is defined in asmline.cpp file. Object is used with LinkedList object. The initiation of the LinkedList object created with asmLine object is done while reading the asm file. Each valid line of the file is inserted into the LinkedList object as an asmLine object. This object is implemented in order to define each line in the .asm file. During execution of the loaded file properties of this object is used. The asmLine object contains another manually created object called “OpCode “ near the predefined data types. Content of the asmLine object is given below.

```
class asmLine :public OpCode
{
private:
    BOOL itsInterrupt;
    UINT itsExecLineNo;
    UCHAR itsDebugMode;
    LONGINT itsFilePtr;
public:
    char itsLineStr[256];
```

```
    UCHAR id;
    UINT LViewId;
    UINT itsMemLoc;
    OpCode *itsOpCode;
    UINT addrMode;
    BOOL itsLabel;
    char itsLbl[100];
    char itsJmpLbl[100];
    UINT itsJmpAddr;
    cmd_str hcodes;
    asmLine(UINT lno, char *linestr);
    UINT assignMemLoc(UINT data);
    void assignLabel(char *data);
    UINT queryExecLineNo(){return itsExecLineNo;};
    void OutStr (char *dest);
    ~asmLine();//destructor
};
```

The properties of this object are used to define the parts of the line of the .asm file. The properties of the object are :

- **itsInterrupt :** (BOOL) Defines if there is an interrupt in that line or not
- **itsExecLineNo :** (UINT) Execution Line Number, the line number neglecting the empty lines
- **itsDebugMode :** (UCHAR) Defines if the line can be debuggable or not
- **itsFilePtr :** (LONGINT) The location of the line in the text file
- **itsLineStr :** (char) The string contained in that line.
- **id :** (UCHAR) Instruction table index for the found item
- **LViewId :** (UINT) The location of the instruction in the view where all the .asm code is printed.
- **itsMemLoc :** (UINT) The location of the instruction in that line in the internal memory of the microprocessor.
- **ItsOpCode :** (OpCode *) The property of the .asm line which holds all the instruction, source and destination data about the line.
- **addrMode :** (UINT) The addressing mode of the instruction contained in this asm line.

```
#define ADDR_IMMEDIATE      0x0001
#define ADDR_INDEXEDX       0x0002
#define ADDR_INDEXEDY       0x0004
#define ADDR_DIRECT         0x0008
```

```
#define ADDR_EXTENDED       0x0010
#define ADDR_INHERENT       0x0020
```

- **itsLabel :** (BOOL) Defines if there is a Label in that line or not
- **itsLbl :** (Char) Holds the label at that line
- **itsJmpLbl :** (char) Holds the jump label at that line
- **hcodes :** (cmd_str) Holds the hexadecimal version of the code existing at that line, both the instruction an the parameters.

5.2.2 OpCode Object

This object is defined in opcode.cpp file. Object is used with asmLine object. The initiation of the OpCode object is done while reading the asm file. Each valid instruction and its arguments are inserted into the OpCode. This object is implemented in order to hold the instruction, source and destination data about the line. Content of the OpCode object is given below.

```
class OpCode
{
private:

public:
        instruction *itsInstruction;
        opDescriptor *source;
        opDescriptor *dest;
        opDescriptor *mask;
        opDescriptor *rel;

        UCHAR QueryPCIncrement(UINT addrmode);
        OpCode();
        void OutStr(char *deststr);
        void OutStrCurr(opDescriptor *curr, char *dest);
        void GetHexStr(char *OutsStr, UINT adr);
        ~OpCode();
};
```

- **itsInstruction :** (instruction *) Used to hold the instruction data type properties of a line in the .asm file.
- **source :** (opDescriptor *) Used to hold the opDescriptor data type properties of the source part of the instruction located in a line in the .asm file.

- **dest :** (opDescriptor *) Used to hold the opDescriptor data type properties of the destination part of the instruction located in a line in the .asm file.
- **mask :** (opDescriptor *) Used to hold the opDescriptor data type properties of the mask part of the instruction located in a line in the .asm file.
- **rel :** (opDescriptor *) Used to hold the opDescriptor data type properties of the relative addressing part of the instruction located in a line in the .asm file.

5.2.3 instruction Structure

This structure is defined in typedef.h file. Structure is used with OpCode object. A new instruction is created for all OpCode objects. “instTable” is a constant variable declaration with instruction type. This structure is implemented in order to hold the all information about an instruction included in the instruction set of MC68HC11. This “instruction” structure is used in the instTable.h file, in the array that contains a list of all the instructions of MC68HC11 and their properties. Content of the instruction structure is given below.

```
typedef struct instruction{
    char *mnemonic;
    unsigned int branch;
    unsigned int hexcode[6];
    int cycle[6];
    int pccount[6];
    int cyclePtr[6];
    int (*exec)(asmLine *);
} instruction;
```

```
const instruction instTable[] = {
  {"ABA", 0, {0,0,0,0,0,0x1B}, {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1},
              {0xff,0xff,0xff,0xff,0xff,0}, instExec_ABA},
.
.
.
.
};
```

- **mnemonic :** (char *) Alphabetic representation of the instruction.
- **branch :** (unsigned int) Defines if the instruction is a branch instruction or not. (0 - 1)

- **hexcode :** (unsigned int [6]) The hexadecimal representation of the instruction for all the addressing modes.
- **cycle :** (int [6]) The number of cycles of the instruction for all the addressing modes.
- **pccount :** (int [6]) The number of increments in the program counter for that instruction for all the addressing modes.
- **cyclePtr :** (int [6]) The index number of the instruction showing which cycle group the instruction is related to.
- **exec :** (int) The executable function related to the instruction containing the operation to be done with this instruction.

5.2.4 opDescriptor Structure

This structure is defined in typedef.h file. Structure is used with OpCode object. Four new opDescriptors are created for each OpCode object. For each argument of an instruction a separate opDescriptor structure holds the properties and value of the argument. Content of the opDescriptor structure is given below.

```
typedef struct {
      long int data;
      char backRef;
      char defined;
} opDescriptor;
```

- **data :** (long int) Value of the argument.
- **backRef :** (char) Represents if the value is referring to a backward memory location.
- **defined :** (char) Specifies if the opDescriptor is defined for the corresponding argument.

5.2.5 cmd_str Structure

This structure is defined in typedef.h file. Structure is used with asmLine object. Structure holds the data of the hexadecimal equivalent of the instruction located in the owner asmLine class. Content of the opDescriptor structure is given below.

```
typedef struct cmd_str {
    unsigned char cmd[20];
    unsigned char len;
} cmd_str;
```

- **cmd :** (unsigned char[20]) Content of the hexadecimal code
- **len :** (unsigned char) Length of the hexadecimal code.

5.2.6 LinkedList Object

This object is defined in linked.cpp file. Object nodes are implemented with template declaration. This object is a single directional list. Content of the LinkedList object is given below.

```
template <class T>
class LinkedList{
      private:
          Node<T> *front;
          Node<T> *getNode(const T& item, Node<T> *ptrNext =
      NULL);
          void freeNode(Node<T> *p);
      public:
          Node<T> *current;
          void resetCurrent();
          LinkedList();
          ~LinkedList();
          int listSize();
          int ListEmpty();
          void Insert(const T& item);
          void PrintList(TRichEdit *PWindow);
};
```

- **front :** (Node<T> *) The pointer pointing to the node in the front of the linked list.
- **current :** (Node<T> *) The pointer pointing to the node active currently in the linked list.
- **LinkedList() :** Constructor of the class, “front” is set to NULL.
- **~LinkedList() :** Destructor of the class , all the nodes are freed, i.e. deleted from the memory.
- **freeNode (Node<T> *p) :** The node passed as parameter is deleted, i.e. the memory is emptied.
- **resetCurrent() :** Current pointer is set to the front pointer.
- **listSize() :** The size of the linked list is returned as integer.

- **ListEmpty() :** Returns boolean value if the front pointer is NULL or not, i.e. the linked list is empty or not
- **Insert(const T& item) :** Inserts a new node to the linked list containing the data in “item”. The node is inserted to the end of the linked list.
- **PrintList(TRichEdit *PWindow) :** The function is used to print all the members of the list to a window defined by “PWindow”.

5.2.7 Node Object

This object is defined in linked.h file. Object is used in LinkedList object. This object forms the data part of the list. Content of the Node object is given below.

```
template <class T>
class Node {
      private:
            Node<T> *next;
      public:
            T data;
            void print (TRichEdit *PWindow);
            Node (const T& item, Node<T> *ptrnext = NULL) :
    data(item), next(ptrnext){};
            void insertAfter(Node<T> *p);
            Node<T> deleteAfter();
            Node<T> *nextNode();
            void AssignNextNode(Node<T> *nextNode);
};
```

- **next :** (Node<T> *) The pointer pointing to the node next to the current node..
- **data :** (T) Represents the data contained in the node object.
- **insertAfter(Node<T> *p) :** The function inserts the “p” node after the current node.
- **deleteAfter() :** The function deletes the node after the current node. If the current node is the last one then NULL is returned.
- **nextNode() :** The function returns the next node after the current node.
- **AssignNextNode(Node<T> *nextNode) :** The function inserts the node “nextNode” after the current node.

5.2.8 Micro Object

This object is defined in micro.cpp file. Object holds all the properties of the microcontroller. An object named as Motorola is created in the project and this object is accessed from different parts of the project. Most important part of the object is the memory part. All the program memory content, data memory content is accessed through this array. Content of the Micro object is given below.

```
class Micro{
      private:
      public:
          UCHAR ACCA, ACCB;
          Reg16Bit ACCD;
          RegCCR CCR;
          Reg16Bit IX, IY;
          Reg16Bit PC;
          Reg16Bit SP;
          Reg16Bit addrBus;
          UCHAR dataBus;
          UCHAR R_W;
          UCHAR itsMem[0x10000];
          UCHAR ii,jj,kk,hh,ll,dd,mm,ff,rr,result,MM,Nxtop;
          UINT SWI_vector;
          Reg16Bit DPTRa;
          Reg16Bit DPTRb;
          Reg16Bit DPTRc;
          UCHAR mode;
          Micro();
          void UpdatePC(asmLine *curLine);
          ~Micro(){delete []itsMem;};//destructor
};
```

5.2.9 RegCCR Object

This object is defined in regCCR.h file. Object holds all the properties of the condition code register. This object is used with the Micro object in order to define the condition code register. The main aim of this object is to handle the bits of the condition code register. Content of the RegCCR object is given below.

```
class RegCCR {
```

```
        private:
            UCHAR data;
        public:
            bool GetBit(UCHAR bitpos);
            void AssignBit(UCHAR bitpos, bool val);
            void Carry8bit(UINT op1,UINT op2,UINT result);
            void Carry8Negate(UINT result);
            void OverFlow8bit(UINT op1,UINT op2,UINT result);
            void OverFlow8bitExor(bool op1, bool op2);
            void Zero8bit(UINT result);
            void Negative8bit(UINT result);
            void  HalfCarry8bit(UINT  op1,  UINT  op2,  UINT
        result);
            void Carry16bit(UINT op1,UINT op2,UINT result);
            void OverFlow16bit(UINT op1,UINT op2,UINT result);
            void Zero16bit(UINT result);
            void Negative16bit(UINT result);
            bool operator=(UCHAR val);
            UCHAR value(void);
            void AssignData(UCHAR val);
            RegCCR(){};//constructor
            ~RegCCR(){};//destructor
    };
```

- **data :** (UCHAR) 8 bit data each bit corresponds to a bit in the CCR.
- **GetBit(UCHAR bitpos) :** Returns the bit of CCR at the “bitpos” position.
- **AssignBit(UCHAR bitpos, bool val) :** Assigns the bit at the “bitpos” position the value “val.”
- **value(void) :** Returns the value of the CCR
- **AssignData(UCHAR val) :** Assigns data in “val” to the CCR.

  CCR : Condition Code Register

  S - X - H - I - N - Z - V - C

  S : Stop Disable, bit 7

  X : X Interrupt mask, bit 6

  H : Half carry, bit 5

  I : I interrupt mask, bit 4

  N : Negative Indicator, bit 3

  Z : Zero Indicator, bit 2

  V : Two's complement overflow indicator, bit 1

C : Carry/Borrow, bit 0

5.2.10 Reg16Bit Object

This object is defined in reg16bit.cpp file. Object is implemented to define bit addressable 16 bit registers of the microcontroller. This object also handles all the functions of the 16 bit registers. Content of the Reg16Bit object is given below.

```
class Reg16Bit
{
private:
    UINT data;
public:
    Reg16Bit &operator=(UINT );
    Reg16Bit &operator=(const Reg16Bit &);
    UINT operator+(UINT );
    Reg16Bit &operator+(const Reg16Bit &);
    Reg16Bit &operator-(UINT );
    Reg16Bit &operator-(const Reg16Bit &);
    UINT value(){return data;};
    UINT valueH(){return (data/256);};
    UINT valueL(){return (data%256);};
   void assignH(UINT val){data&=0x00ff;data+=(val&0xff)<<8;};
    void assignL(UINT val){data&=0xff00;data+=val&0xff;};
    Reg16Bit(){};//constructor
    ~Reg16Bit(){};//destructor
};
```

5.3 Lookup Tables

All lookup tables are defined in instTable.h file.

5.3.1 instTable Table

The main lookup table is instTable which is constructed by using instruction structure. The detailed instruction table is given in APPENDIX B. In this table:

- mnemonic: of the instruction
- branch: property
- hexcode[6]: hexadecimal equivalent of the instruction for six different addressing modes.

- cycle[6]: number of cycles of the instruction for six different addressing modes.
- pccount[6]: program counter incremention value of the instruction for six different addressing modes.
- cyclePtr[6]: pointer of the instruction to the CycleTable for six different addressing modes.
- (*exec)(asmLine *): Corresponding execution function of the instruction. Addressing modes are handled in the functions.

5.3.2 CycleTable Table

CycleTable is constructed by using MachineCycleType structure. The details of the cycles of each instruction is given in APPENDIX A. Also the cycle table structure is given in the source code part, APPENDIX B. In this table:

- val[16]: Defines the cycle numbers of the corresponding entry. Cycle numbers are defined between 0-121 in execute.cpp.

5.3.3 MDefCycleTable Table

MDefCycleTable is constructed by using MachineCycleDefStruct structure. In this table:

- id: Cycle number
- address: String which explains address bus content for the cycle which is defined with id index.
- data: String which explains data bus content for the cycle which is defined with id index.
- r_w: String which explains read/write pin content for the cycle which is defined with id index.

5.4 Execution Process Sequence

Program execution can be examined in five steps.

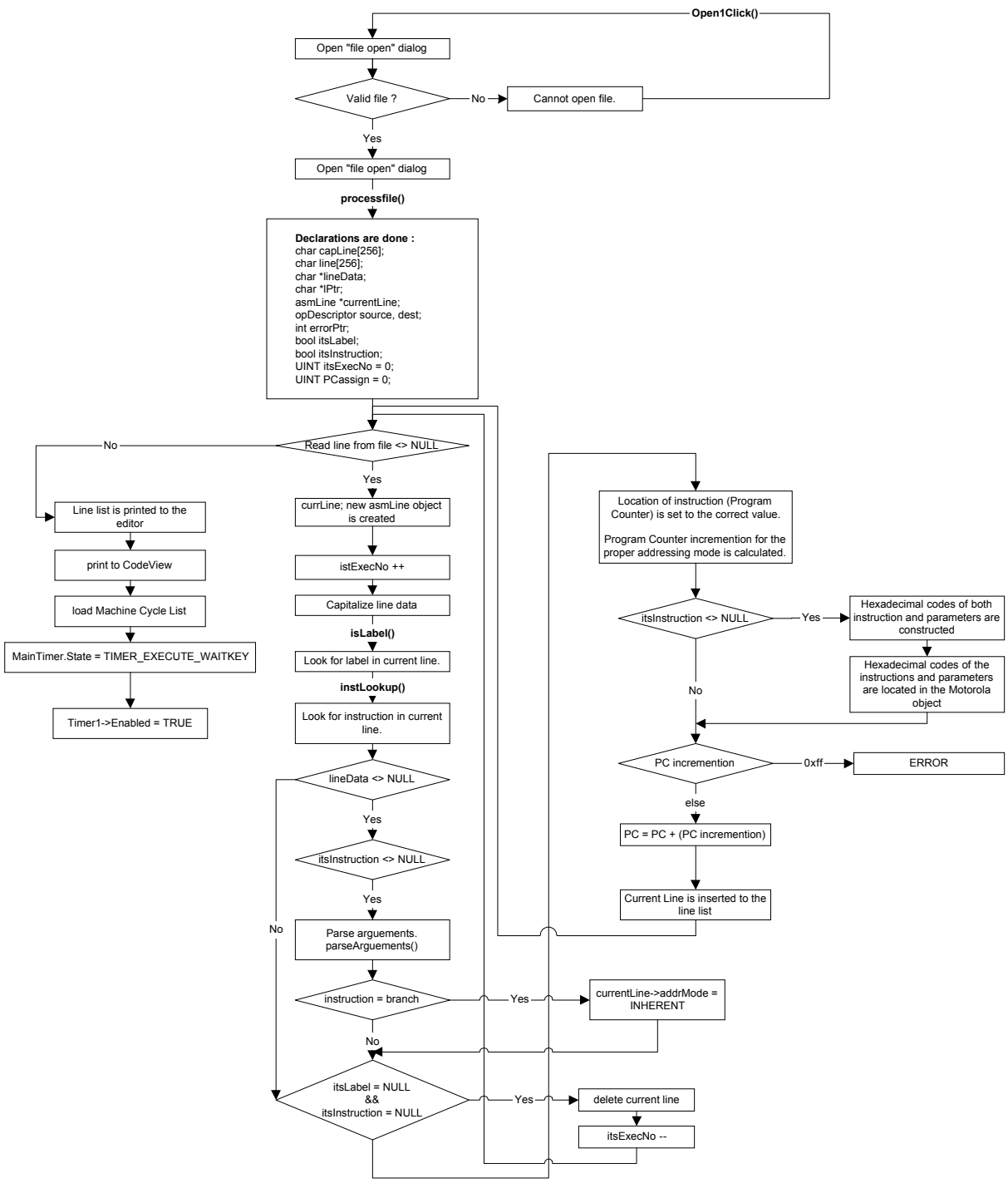

Figure 5.9 Flowchart of the Application

5.4.1 Startup and initiations

This step consists creation of main form creation. Once the exe file is ran main form is created and CycleView, CodeView, CCRGrid, AddrDataGrid, RegisterGrid objects are initialized. The column name data of these objects are inserted and properties are adjusted.

The constant tables are created. These constant tables are instTable Table, CycleTable Table, MDefCycleTable Table and these tables are mentioned in the chapter 5.3.

Motorola object is created from Micro class. If interrupt vector, stack pointer and start vector of the microcontroller object are not set by using the corresponding menu items under the "Edit" menu, then:

- `UINT IntVectorAddress=0xa000`
- `UINT SPVector=0x8000`
- `UINT PCVector=0x1000`

declerations are done during the initialization process.

LineList object is created from LinkedList class with asmLine template.

ExecutionTimer is created and state is initiated.

5.4.2 Open File and Parse File

OpenDialog1 object is executed. Dialog waits user for file selection. After the user selects a valid file, selected file is opened with read only attribute and processfile function is called. ExecutionTimer object is enabled with time interval 1 milliseconds.

5.4.2.1 ProcessFile Function

ProcessFile function is defined in funcs2.cpp file.

1. Opened file is read line by line till the end of file is reached.
2. For each read line an asmLine object is created.
3. Labels are picked from the line with IsLabel function.
4. Instruction search is done by instLookup function on the rest of the string.
5. In case of a valid instruction argument parse operation is made by using parseArguments function.
6. Hexadecimal codes for the found instruction is obtained by using querySourceCodes and obtained results are inserted into asmLine object by using insertSourceCodes function.
7. Finally created asmLine object is inserted into LineList object.

8. In case an instruction is not found in the read line created asmLine object is deleted.

5.4.2.1.1 IsLabel Function

IsLabel function is defined in funcs2.cpp file.

1. Line is searched until the read character is either an alphanumeric one, '.', '_' or '$'.
2. All the characters are put into a string until a character other than described above is read. When this stiuation occurs an "end of string (/0)" character is put at the end of this string.
3. For the rest of the line; if the next character is the same as the start of the line (which is the situation that we are still at the beginning of the line), or if the next character is ":" (which is the situation of the end of the label), or if the next character is "*" (which is the situation that the rest of the line is just a comment); the string where the characters are collected is set as the label for that line.
4. If the label is found to be "`NMINTERRUPT`" then the rest of the instructions until the RTI instruction is parsed from the file is thought to be interrupt routine to be entered by pressing the button simulating the hardware interrupt during the normal execution process.
5. For the rest of the line, after the spaces are cleared, the next character is set as the start point of the rest of the asmLine where instruction is searched.

5.4.2.1.2 instLookup Function

instLookup function is defined in funcs2.cpp file.

1. If the rest of the asmLine is not NULL, then the instruction is searched in the line.
2. The line is serched until the end of the mnemonic of the instruction is reached.
3. The obtained mnemonic is searched in the instTable, which is the contant instruction table described in section 6.
4. The result obtained from this searched is assigned to the "asmLPtr→itsOpCode→itsInstruction" property of the asmLine object.

5.4.2.1.3 parseArguments Function

parseArguments functions is defined in the funcs2.cpp. In this process, digital design considerations and computer system architecture are taken into account [8], [9].

1. source, dest, mask and rel properties of the "curLine→itsOpCode" object is reseted at the begining of the function.
2. A temprorary opDescriptor structure is reseted also.

3. Until the rest of the asmLine object becomes NULL, i.e. the function reaches the end of the line, all the arguements of the instruction is parsed by using opParse function and the temporary opDescriptor structure as a parameter.
4. Then all the arguements parsed by using the opParse function is assigned to the arguements of the current asmLine's "curLine→itsOpCode" object.
5. "Defined" property of source, dest, mask and rel structures of "currLine→itsOpCode" object is set to 1 if they are parsed by the function.

5.4.2.1.4 querySourceCodes Function

querySourceCodes function is defined in the funs2.cpp file.

1. Hexcode of the instruction is examined if it is bigger than 0xff, i.e it is bigger than one byte. Depending on the situation currLine→hcodes.cmd and currLine→hcodes.len properties of the current asmLine object are updated.
2. All the arguments of the instruction are examined one by one if they are parsed. If they are parsed, they are examined if they are bigger than one byte or not. Depending on the situation currLine->hcodes.cmd and currLine→hcodes.len properties of the current asmLine opbject is updated accordingly.

5.4.2.1.5 insertSourceCodes Function

insertSourceCodes function is defined in the funcs2.cpp file.

1. For each instruction in the asm file "currLine→hcodes.len" is used to send the hexadecimal representation of the instruction byte by byte to Motorola object in order to be printed out to the corresponding form.

5.4.3 Wait for Keystroke

ExecutionTimer Object starts to wait in the TIMER_EXECUTE_WAITKEY status, so it starts to wait a keystroke. Unless the application is stopped by the halt of the system, i.e. TIMER_EXECUTE_IDLE status, pressing one of F7, F8, F9 keys or the corresponding menu items makes the application reacts to the keystroke.

As explained above, F7 means "single cycle execution", F8 means "single instruction execution" and F9 means "execute all".

5.4.4 Execute given command or commands

According to the key pressed, or the menu item selected one of the below operations are done:

1. Execute Current Cycle
2. Execute Current PC
3. Execute All Intructions

All the flowcharts relating to these operations are given below :

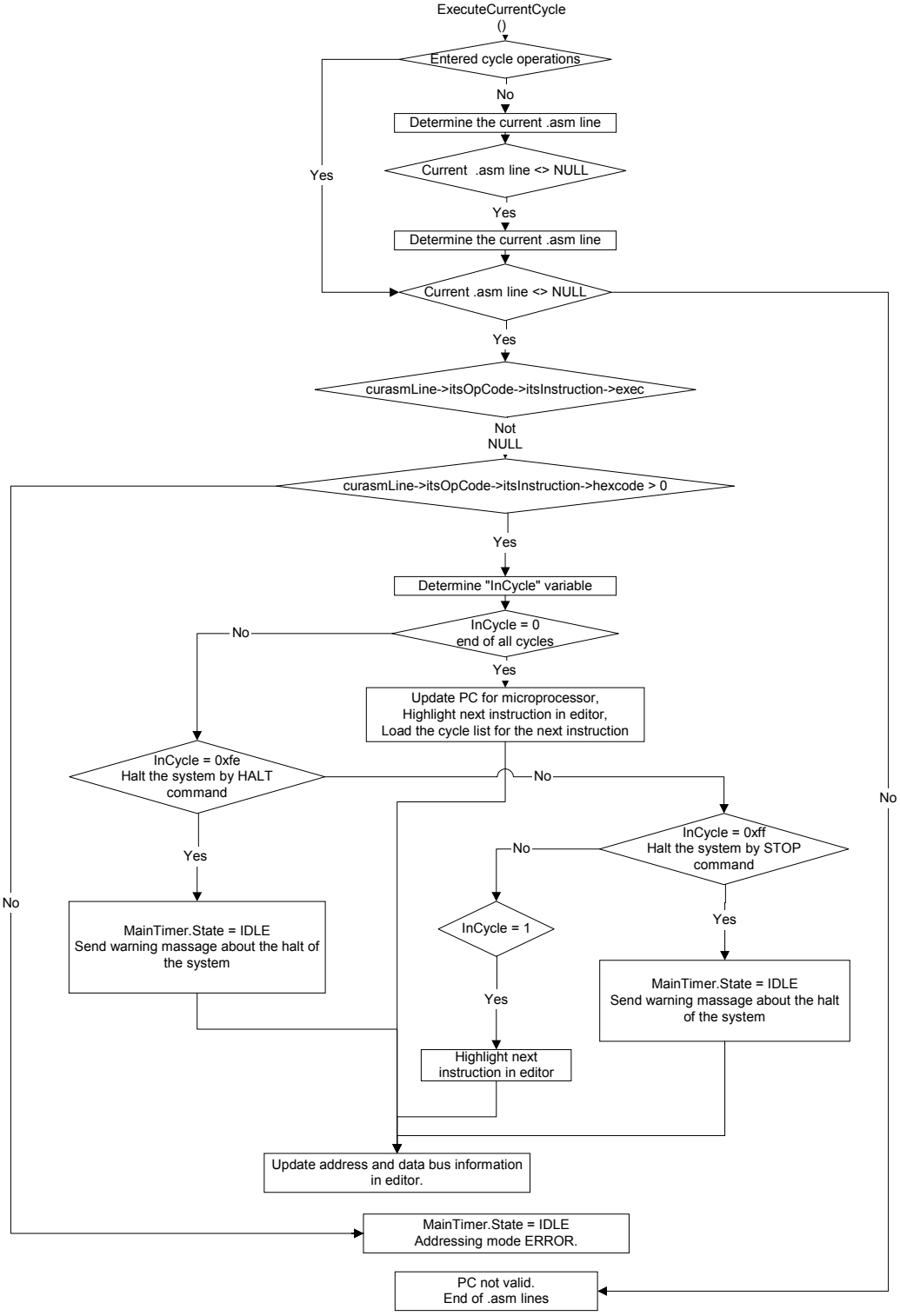

Figure 5.10 Flowchart of ExecuteCurrentCycle Function

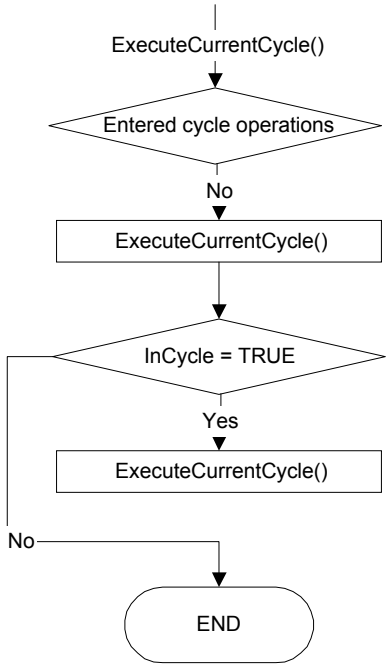

Figure 5.11 Flowchart of ExecuteCurrentPC Function

The base operation of all those three operations is “execute cycle” operation. All other operations are containing this operation.

ExecuteCurrentCycle function firstly controls if asmLine object is valid or not. Then addressing mode of the instruction is controlled if it is valid for that specific instruction ot not. Then after these controls, the function checks if the instruction has a valid execution funtion determining the processes specific for that instruction. If there is a valid execution function for the instruction, this function is called. When the execution function of the instruction is called, a specific function for this instruction is called from execute.cpp file. In this function, the cycles of the instruction called one by one by the function machineExecute(). After this process is completed, the function returns to the function ExecuteCurrentCycle().

5.4.5 Refresh screen

When a keystroke sensed and corresponding applications are executed by the application, the user interface of the application must be refreshed in order to present the new conditions.

When a cycle of an instruction is executed by pressing F7, if there are more cycles for the same instruction then the next cycle of the instruction is highlighted by the HighlightNextCycle() function. If there are no cycles left for the same instruction then the next instruction is highlighted by the HighlightNextExecution() function and the cycle list of the newly highlighted instruction is loaded to the user interface by the LoadMachineCycleList() function.

When an instruction is executed by the application, same operations are carried out but since the timer interval of the application is very small, the cycle highlight operation can not be observed. If the instruction is not the last instruction in the file then, what can be observed on the user interface to be refreshed is the highlight operation of the next instruction. Also the cycle list of the current instruction is also loaded to the user interface. If it is the last instruction in the file then “End of All Items” warning message is given by the application.

CHAPTER 6

CONCLUSION

The aim of this thesis study is to develop a simulator for an 8-bit microcontroller. In this simulator a code written in the assembler language of the microcontroller and saved as an asm file is opened and compiled by the assembler compiler written in C++ programming language. The compiled instructions are executed and the results obtained from instructions are shown in the graphical user interface of the microcontroller simulator.

By using the software implemented, the file containing the assembler code of the microcontroller is opened. The code contained in the file is parsed, all the instructions in the file are found out with all the labels and interrupt routines included in the file. All the instructions parsed from the file are distinguished with all their properties by using the lookup tables. The instructions are executed cycle by cycle using the execution functions written for each instruction seperately. During the execution of the instructions, update of the registries, address bus and data bus of the microcontroller can be observed by the graphical user interface.

Besides, the software implemented can simulate the hardware interrupt property of the microcontroller. On the graphical user interface, there is button for this simulation purpose; by pressing this button the XIRQ is sensed by the microcontroller and the program counter of the microcontroller is set to the interrupt vector. Until RTI instruction is processed, the microcontroller executes the interrupt routine and processes the generic interrupt procedures.

All through these operations the updates occurring in the registries, data bus and address bus of the microcontroller can be observed through the graphical user interface. The operations can be executed either cycle by cycle or intruction by instruction or all the intructions at a time.

This tool can be used for educational purposes referring to the aim of the thesis project, since it is easy and understandable for a student to observe the behaviour of the microcontroller from a graphical user interface showing all the registries, data bus and address bus of the microcontroller. Since cycle operation is also implemented in the simulator, the behaviour of an instruction from the instruction set of the microcontroller can easily be observed cycle by cycle with all the changes occurring.

Also by improving the software accordingly, the application can be turned into a simulator showing the behaviours of a basic computer since a similar principle is working at the background of a basic computer.

As a result of this thesis study, the operation principle of a microcontroller becomes clear for the user and the concepts included in the subject of the microcontroller can be observed in detail with all the changes occurring during the execution process.

REFERENCES

1. Deitel, H. M., Deitel P. J., C++, Prentice-Hall, 2001
2. HC11 M68HC11 Reference Manual, Motorola Inc., 1996
3. HC11 M68HC11L6 Technical Data, Motorola Inc., 1992
4. HC11 M68HC11 E Series, Motorola Inc., 1995
5. Holub, A. I., Compiler Design in C, Prentice-Hall, C1990
6. Ford, W., Topp, W., Data Structures with C++, Prentice-Hall, 1996
7. Lewis II, P. M., Rosenkrantz, D.J., Streans R.E., Compiler Design Theory, Addison-Wesley Pub. Co., C1976,
8. Mano, M. M., Digital Design, Prentice-Hall, C1984
9. Mano, M. M., Computer System Architecture, Prentice-Hall, C1976

APPENDIX A - Machine Cycles

Table A.1 ABA Cycle Table

Cycle	ABA (INH)			
	Addr	Data	R/W	Own Code
1	OP	1B	1	20
2	OP+1	-	1	23
{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				

Table A.2 ABX Cycle Table

Cycle	ABX			
	Addr	Data	R/W	Own Code
1	OP	3A	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}				

Table A.3 ABY Cycle Table

Cycle	ABX			
	Addr	Data	R/W	Own Code
1	OP	1B	1	20
2	OP+1	3A	1	27
3	OP+2	-	1	39
4	FFFF	-	1	8
{20,27,39,8,0xf4,0xf5,0xf6,0xf7}				

Table A.4 ADCA Cycle Table

Cycle	ADCA (IMM)				ADCA (DIR)				ADCA (EXT)				ADCA (INDX)				ADCA (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	89	1	20	OP	99	1	20	OP	B9	1	20	OP	A9	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A9	1	27
3					00dd	00dd	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	hhll	1	13	X+ff	X+ff	1	47	FFFF	-	1	8
5																	Y+ff	Y+ff	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.5 ADCB Cycle Table

Cycle	ADCB (IMM)				ADCB (DIR)				ADCB (EXT)				ADCB (INDX)				ADCB (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C9	1	20	OP	D9	1	20	OP	F9	1	20	OP	E9	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	E9	1	27
3					00dd	00dd	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	hhll	1	13	X+ff	X+ff	1	47	FFFF	-	1	8
5																	Y+ff	Y+ff	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.6 ADDA Cycle Table

Cycle	ADDA (IMM)				ADDA (DIR)				ADDA (EXT)				ADDA (INDX)				ADDA (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	BB	1	20	OP	9B	1	20	OP	BB	1	20	OP	AB	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AB	1	27
3					00dd	00dd	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4									hhll	hhll	1	13	X+ff	X+ff	1	47	FFFF	-	1	8
5																	Y+ff	Y+ff	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.7 ADDB Cycle Table

Cycle	ADDB (IMM)				ADDB (DIR)				ADDB (EXT)				ADDB (INDX)				ADDB (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	CB	1	20	OP	DB	1	20	OP	FB	1	20	OP	EB	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	EB	1	27
3					00dd	00dd	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	hhll	1	13	X+ff	X+ff	1	47	FFFF	-	1	8
5																	Y+ff	Y+ff	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.8 ABX Cycle Table

Cycle	ADDD (IMM)				ADDD (DIR)				ADDD (EXT)				ADDD (INDX)				ADDD (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	CB	1	20	OP	DB	1	20	OP	FB	1	20	OP	EB	1	20	OP	18	1	20
2	OP+1	jj	1	29	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	EB	1	27
3	OP+2	kk	1	36	00dd	00dd	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4	FFFF	-	1	8	00dd+1	00dd+1	1	3	hhll	hhll	1	13	X+ff	X+ff	1	47	FFFF	-	1	8
5					FFFF	-	1	8	hhll+1	hhll+1	1	15	X+ff+1	X+ff+1	1	48	Y+ff	Y+ff	1	54
6									FFFF	-	1	8	FFFF	-	1	8	Y+ff+1	Y+ff+1	1	55
7																	FFFF	-	1	8
	{20,29,36,8,0xf4,0xf5,0xf6,0xf7}				{20,24,1,3,8,0xf5, 0xf6,0xf7}				{20,26,37,13,15,8,0xf6,0xf7}				{20,25,8,47,48,8,0xf6,0xf7}				{20,27,35,8,54,55,8,0xf7}			

Table A.9 ANDA Cycle Table

Cycle	ANDA (IMM)				ANDA (DIR)				ANDA (EXT)				ANDA (INDX)				ANDA (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	84	1	20	OP	94	1	20	OP	B4	1	20	OP	A4	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	Dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AB	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.10 ANDB Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(INDX)				(INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C4	1	20	OP	D4	1	20	OP	F4	1	20	OP	E4	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	Dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	EB	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.11 ASLA Cycle Table

Cycle	ASLA			
	Addr	Data	R/W	Code
1	OP	48	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.12 ASLB Cycle Table

Cycle	ASLB			
	Addr	Data	R/W	Code
1	OP	58	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.13 ASL Cycle Table

Cycle	ASL (EXT)				ASL (INDX)				ASL (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	78	1	20	OP	68	1	20	OP	18	1	20
2	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	68	1	27
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFFhhll	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46,0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.14 ASLD Cycle Table

Cycle	ASLD (INH)			
	Addr	Data	R/W	Code
1	OP	05	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.15 ASRA Cycle Table

Cycle	ASRA (INH)			
	Addr	Data	R/W	Code
1	OP	47	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.16 ASRB Cycle Table

Cycle	ASRB (INH)			
	Addr	Data	R/W	Code
1	OP	57	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.17 ASR Cycle Table

Cycle	ASR (EXT)				ASR (INDX)				ASR (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	77	1	20	OP	67	1	20	OP	18	1	20
2	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	68	1	27
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46,0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.18 BCC Cycle Table

Cycle	BCC (INH)			
	Addr	Data	R/W	Code
1	OP	24	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.19 BCLR Cycle Table

Cycle	BCLR (DIR)				BCLR (IND,X)				BCLR (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	15	1	20	OP	1D	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	Ff	1	25	OP+1	1D	1	27
3	00dd	(00dd)	1	1	FFFF	-	1	8	OP+2	ff	1	35
4	OP+2	MM	1	38	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	OP+2	MM	1	38	(IY)+ff	(IY+ff)	1	54
6	00dd	result	0	6	FFFF	-	1	8	OP+3	mm	1	43
7					X+ff	result	0	46	FFFF	-	1	8
8									Y+ff	result	0	53
	{20,24,1,38,8,6,0xf6,0xf7}				{20,25,8,47,38,8,46,0xf7}				{20,27,35,8,54,43,8,53}			

Table A.20 BCS Cycle Table

Cycle	BCS (REL)			
	Addr	Data	R/W	Code
1	OP	25	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.21 BEQ Cycle Table

Cycle	BEQ (REL)			
	Addr	Data	R/W	Code
1	OP	27	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.22 BGE Cycle Table

Cycle	BGE (REL)			
	Addr	Data	R/W	Code
1	OP	2C	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.23 BGT Cycle Table

Cycle	BGT (REL)			
	Addr	Data	R/W	Code
1	OP	2E	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.24 BHI Cycle Table

Cycle	BHI (REL)			
	Addr	Data	R/W	Code
1	OP	22	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.25 BHS Cycle Table

Cycle	BHS (REL)			
	Addr	Data	R/W	Code
1	OP	24	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.26 BITA Cycle Table

Cycle	BITA (IMM)				BITA (DIR)				BITA (EXT)				BITA (INDX)				BITA (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	85	1	20	OP	95	1	20	OP	B5	1	20	OP	A5	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AB	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.27 BITB Cycle Table

Cycle	BITB (IMM)				BITB (DIR)				BITB (EXT)				BITB (INDX)				BITB (INDY)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C5	1	20	OP	D5	1	20	OP	F5	1	20	OP	E5	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AB	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.28 BLE Cycle Table

Cycle	BLE (REL)			
	Addr	Data	R/W	Code
1	OP	2F	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.29 BLO Cycle Table

Cycle	BLO (REL)			
	Addr	Data	R/W	Code
1	OP	25	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.30 BLS Cycle Table

Cycle	BLS (REL)			
	Addr	Data	R/W	Code
1	OP	23	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.31 BLT Cycle Table

Cycle	BLT (REL)			
	Addr	Data	R/W	Code
1	OP	2D	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.32 BMI Cycle Table

Cycle	BMI (REL)			
	Addr	Data	R/W	Code
1	OP	2B	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.33 BNE Cycle Table

Cycle	BNE (REL)			
	Addr	Data	R/W	Code
1	OP	26	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.34 BPL Cycle Table

Cycle	BPL (REL)			
	Addr	Data	R/W	Code
1	OP	2A	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.35 BRA Cycle Table

Cycle	BRA (REL)			
	Addr	Data	R/W	Code
1	OP	20	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.36 BRCLR Cycle Table

Cycle	BRCLR (DIR)				BRCLR (IND,X)				BRCLR (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	13	1	20	OP	1F	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	ff	1	25	OP+1	1F	1	27
3	00dd	(00dd)	1	1	FFFF	-	1	8	OP+2	ff	1	35
4	OP+2	mm	1	38	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	OP+3	rr	1	44	OP+2	mm	1	38	(IY)+ff	(IY+ff)	1	54
6	FFFF	-	1	8	OP+3	rr	1	44	OP+3	mm	1	43
7					FFFF	-	1	8	OP+4	rr	1	45
8									FFFF	-	1	8
	{20,24,1,38,44,8,0xf6,0xf7}				{20,25,8,47,38,44,8,0xf7}				{20,27,35,8,54,43,45,8}			

Table A.37 BRN Cycle Table

Cycle	BRN (REL)			
	Addr	Data	R/W	Code
1	OP	21	1	20
2	OP+1	rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.38 BRSET Cycle Table

Cycle	BRSET (DIR)				BRSET (IND,X)				BRSET (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	12	1	20	OP	1E	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	Ff	1	25	OP+1	1E	1	27
3	00dd	(00dd)	1	1	FFFF	-	1	8	OP+2	ff	1	35
4	OP+2	mm	1	38	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	OP+3	Rr	1	44	OP+2	Mm	1	38	(IY)+ff	(IY+ff)	1	54
6	FFFF	-	1	8	OP+3	Rr	1	44	OP+3	mm	1	43
7					FFFF	-	1	8	OP+4	rr	1	45
8									FFFF	-	1	8
	{20,24,1,38,44,8,0xf6,0xf7}				{20,25,8,47,38,44,8,0xf7}				{20,27,35,8,54,43,45,8}			

Table A.39 BSET Cycle Table

Cycle	BSET (DIR)				BSET (IND,X)				BSET (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	14	1	20	OP	1E	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	Ff	1	25	OP+1	1E	1	27
3	00dd	(00dd)	1	1	FFFF	-	1	8	OP+2	ff	1	35
4	OP+2	mm	1	38	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	OP+2	Mm	1	38	(IY)+ff	(IY+ff)	1	54
6	00dd	result	0	2	FFFF	-	1	8	OP+3	mm	1	43
7					X+ff	result	0	46	FFFF	-	1	8
8									IY+ff	result	0	53
	{20,24,1,38,8,2,0xf6,0xf7}				{20,25,8,47,38,8,46,0xf7}				{20,27,35,8,54,43,8,53}			

Table A.40 BSR Cycle Table

Cycle	BSR (REL)			
	Addr	Data	R/W	Code
1	OP	8D	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
4	SUB	Nxt op	1	60
5	SP	Rtn lo	0	61
6	SP-1	Rtn hi	0	62
	{20,30,8,60,61,62,0xf6,0xf7}			

Table A.41 BVC Cycle Table

Cycle	BVC (REL)			
	Addr	Data	R/W	Code
1	OP	28	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.42 BVS Cycle Table

Cycle	BVS (REL)			
	Addr	Data	R/W	Code
1	OP	29	1	20
2	OP+1	Rr	1	30
3	FFFF	-	1	8
	{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.43 CBA Cycle Table

Cycle	CBA (INH)			
	Addr	Data	R/W	Code
1	OP	11	1	20
2	OP+1	Rr	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.44 CLC Cycle Table

Cycle	CLC (INH)			
	Addr	Data	R/W	Code
1	OP	11	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.45 CLI Cycle Table

Cycle	CLI (INH)			
	Addr	Data	R/W	Code
1	OP	0E	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.46 CLRA Cycle Table

Cycle	CLRA (INH)			
	Addr	Data	R/W	Code
1	OP	4F	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.47 CLRB Cycle Table

Cycle	CLRB (INH)			
	Addr	Data	R/W	Code
1	OP	5F	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.48 CLR Cycle Table

Cycle												
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	7F	1	20	OP	7F	1	20	OP	7F	1	20
2	OP+1	hh	1	26	OP+1	hh	1	25	OP+1	hh	1	27
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ll	1	35
4	Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	Hhll	00	0	12	X+ff	00	0	49	FFFF	-	1	8
7									Y+ff	00	0	52
	{20,26,37,13,8,12,0xf6,0xf7}				{20,25,8,47,8,49, 0xf6,0xf7}				{20,27,35,8,54,8,52,0xf7}			

Table A.49 CLV Cycle Table

Cycle	CLV (INH)			
	Addr	Data	R/W	Code
1	OP	0A	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.50 CMPA Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	81	1	20	OP	91	1	20	OP	B1	1	20	OP	A1	1	20	OP	18	1	20
2	OP+1	Ii	1	29	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A1	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff		35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,29,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.51 CMPB Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C1	1	20	OP	D1	1	20	OP	F1	1	20	OP	E1	1	20	OP	18	1	20
2	OP+1	Ii	1	29	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	E1	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff		35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,29,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.52 COMA Cycle Table

Cycle	COMA(INH) (INH)			
	Addr	Data	R/W	Code
1	OP	43	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.53 COMB Cycle Table

Cycle	COMB (INH)			
	Addr	Data	R/W	Code
1	OP	53	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.54 COM Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	73	1	20	OP	63	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	Ff	1	25	OP+1	63	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	Result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.55 CPD Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	1A	1	20	OP	1A	1	20	OP	1A	1	20	OP	1A	1	20	OP	CD	1	20
2	OP+1	83	1	27	OP+1	93	1	27	OP+1	B3	1	27	OP+1	A3	1	27	OP+1	A3	1	27
3	OP+2	Jj	1	40	OP+2	dd	1	41	OP+2	Hh	1	34	OP+2	ff	1	35	OP+2	ff	1	35
4	OP+3	Kk	1	42	00dd	(00dd)	1	1	OP+3	Ll	1	111	FFFF	-	1	8	FFFF	-	1	8
5	FFFF	-	1	8	00dd+1	(00dd+1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	Y+ff	(Y+ff)	1	54
6					FFFF	-	1	8	Hhll +1	(hhll +1)	1	15	X+ff +1	(X+ff +1)	1	48	Y+ff +1	(Y+ff +1)	1	55
7									FFFF	-	1	8	FFFF	-	1	8	FFFF	-	1	8
	{20,27,40,42,8,0xf5,0xf6,0xf7}				{20,27,41,1,3,8,0xf6,0xf7}				{20,27,34,111,13,15,8,0xf7}				{20,27,35,8,47,48,8,0xf7}				{20,27,35,8,54,55,8}			

Table A.56 CPX Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	8C	1	20	OP	9C	1	20	OP	BC	1	20	OP	AC	1	20	OP	CD	1	20
2	OP+1	Jj	1	29	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	AC	1	27
3	OP+2	Kk	1	36	00dd	(00dd)	1	1	OP+2	Hh (ll olmalı)	1	34	FFFF	-	1	8	OP+2	Ff	1	35
4	FFFF	-	1	8	00dd +1	(00dd +1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5					FFFF	-	1	8	Hhll +1	(hhll +1)	1	15	X+ff +1	(X+ff +1)	1	48	Y+ff	(Y+ff)	1	54
6									FFFF	-	1		FFFF	-	1	8	Y+ff +1	(Y+ff +1)	1	55
7																	FFFF	-	1	8
	{20,29,36,8, 0xf4, 0xf5, 0xf6,0xf7}				{20,24,1,3, 8, 0xf5, 0xf6,0xf7}				{20,26,34,13, 15, 0xf5, 0xf6,0xf7}				{20,25,8,47,48,8, 0xf6,0xf7}				{20,27,35,8,54,55,8,0xf7}			

Table A.57 CPY Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	18	1	20	OP	18	1	20	OP	BC	1	20	OP	1A	1	20	OP	18	1	20
2	OP+1	8C	1	27	OP+1	9C	1	27	OP+1	Hh	1	27	OP+1	AC	1	27	OP+1	AC	1	27
3	OP+2	Jj	1	40	OP+2	dd	1	41	OP+2	Hh	1	34	OP+2	ff	1	35	OP+2	ff	1	35
4	OP+3	Kk	1	42	00dd	(00dd)	1	1	OP+3	Ll	1	111	FFFF	-	1	8	FFFF	-	1	8
5	FFFF	-	1	8	00dd+1	(00dd+1)	1	3	Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	Y+ff	(Y+ff)	1	54
6					FFFF	-	1	8	hhll+1	(hhll+1)	1	15	X+ff+1	(X+ff+1)	1	48	Y+ff+1	(Y+ff+1)	1	55
7									FFFF	-	1	8	FFFF	-	1	8	FFFF	-	1	8
	{20,27,40,42,8,0xf5,0xf6,0xf7}				{20,27,41,1,3,8,0xf6,0xf7}				{20,27,34,111,13,15,8,0xf7}				{20,27,35,8,47,48,8,0xf7}				{20,27,35,8,54,55,8}			

Table A.58 DAA Cycle Table

Cycle	(INH)			
	Addr	Data	R/W	Code
1	OP	19	1	20
2	OP+1	-	1	23
3				
4				
5				
6				
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.59 DECA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	4A	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.60 DECB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	5A	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.61 DEC Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	7A	1	20	OP	7A	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	Hh	1	25	OP+1	6A	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	Hhll	Result	1	14	X+ff	result	1	46	FFFF	-	1	8
7									Y+ff	result	1	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46,0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.62 DES Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	34	1	20
2	OP+1	-	1	23
3	SP	-	1	63
	{20,23,63,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.63 DEX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	09	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.64 DEY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	09	1	27
3	OP+2	-		39
4	FFFF	-	1	8
	{20,27,39,8,0xf4,0xf5,0xf6,0xf7}			

Table A.65 EORA Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	88	1	20	OP	98	1	20	OP	B8	1	20	OP	A8	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A8	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.66 EORB Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C8	1	20	OP	D8	1	20	OP	F8	1	20	OP	E8	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	E8	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.67 FDIV Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	03	1	20
2	OP+1	-	1	23
3-41	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.68 IDIV Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	02	1	20
2	OP+1	-	1	23
3-41	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.69 INCA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	4C	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.70 INCB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	5C	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.71 INC Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	7C	1	20	OP	6C	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	Ff	1	25	OP+1	6C	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	result	0	14	X+ff	Result	0	46	FFFF	-	1	8
7									Y+ff	result	1	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46,0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.72 INS Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	31	1	20
2	OP+1	-	1	23
3	SP	-	1	63
	{20,23,63,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.73 INX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	08	1	20
2	OP+1	-	1	23
3	SP	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.74 INY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	08	1	27
3	OP+2	-	1	39
4	FFFF	-	1	8
	{20,27,39,8,0xf4,0xf5,0xf6,0xf7}			

Table A.75 JMP Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	7E	1	20	OP	6E	1	20	OP	18	1	20
2	OP+1	hh	1	26	OP+1	Ff	1	25	OP+1	6E	1	27
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4									FFFF	-		8
	{20,26,37,0xf 3,0xf4,0xf5,0xf6,0xf7}				{20,25,8,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,0xf4,0xf5,0xf6,0xf7}			

Table A.76 JSR Cycle Table

Cycle	DIR				EXT				IND,X				IND,Y			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	9D	1	20	OP	BD	1	20	OP	AD	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	Ff	1	25	OP+1	AD	1	27
3	00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	SP	Rtn lo	0	61	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	SP-1	Rtn hi	0	62	SP	Rtn lo	0	61	SP	Rtn lo	0	61	Y+ff	(Y+ff)	1	54
6					SP-1	Rtn hi	0	62	SP-1	Rtn hi	0	62	SP	Rtn lo	0	61
7													SP-1	Rtn hi	0	62
	{20,24,1,61,62,0xf5,0xf6,0xf7}				{20,26,37,13,61,62,0xf6,0xf7}				{20,25,8,47,61,62,0xf6,0xf7}				{20,27,35,8,54,61,62,0xf7}			

Table A.77 LDAA Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	86	1	20	OP	96	1	20	OP	B6	1	20	OP	A6	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	A6	1	27
3					00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)		54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.78 LDAB Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C6	1	20	OP	D6	1	20	OP	F6	1	20	OP	E6	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	E6	1	27
3					00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)		54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.79 LDD Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	CC	1	20	OP	DC	1	20	OP	FC	1	20	OP	EC	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	Dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	E6	1	27
3	OP+2	Kk	1	36	00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4					00dd +1	(00dd +1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5									Hhll+1	(Hhll +1)	1	15	X+ff +1	(X+ff +1)	1	48	Y+ff	(Y+ff)	1	54
6																	Y+ff +1	(Y+ff +1)	1	55
	{20,28,36,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,15,0xf5,0xf6,0xf7}				{20,25,8,47,48,0xf5,0xf6,0xf7}				{20,27,35,8,54,55,0xf6,0xf7}			

Table A.80 LDS Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	BE	1	20	OP	9E	1	20	OP	EE	1	20	OP	AE	1	20	OP	18	1	20
2	OP+1	Jj	1	29	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AE	1	27
3	OP+2	Kk	1	36	00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4					00dd +1	(00dd +1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5									Hhll +1	(hhll +1)	1	15	X+ff +1	(X+ff +1)	1	48	Y+ff	(Y+ff)	1	54
6																	Y+ff 1	(Y+ff +1)	1	55
	{20,29,36,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,15,0xf5,0xf6,0xf7}				{20,25,8,47,48,0xf5,0xf6,0xf7}				{20,27,35,8,54,55,0xf6,0xf7}			

Table A.81 LDX Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	CE	1	20	OP	DE	1	20	OP	FE	1	20	OP	EE	1	20	OP	CD	1	20
2	OP+1	Jj	1	29	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	EE	1	27
3	OP+2	Kk	1	36	00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4					00dd+1	(00dd+1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5									Hhll+1	(hhll+1)	1	15	X+ff+1	(X+ff+1)	1	48	Y+ff	(Y+ff)	1	54
6																	Y+ff+1	(Y+ff+1)	1	55
	{20,29,36,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,15,0xf5,0xf6,0xf7}				{20,25,8,47,48,0xf5,0xf6,0xf7}				{20,27,35,8,54,55,0xf6,0xf7}			

Table A.82 LDY Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	18	1	20	OP	18		20	OP	18	1	20	OP	18	1	20	OP	18	1	20
2	OP+1	CE	1	27	OP+1	DE		27	OP+1	FE	1	27	OP+1	EE	1	27	OP+1	EE	1	27
3	OP+2	Jj	1	40	OP+2	dd		41	OP+2	hh	1	34	OP+2	ff		35	OP+2	Ff	1	35
4	OP+3	Kk		42	00dd	(00dd)	1	1	OP+3	ll	1	111	FFFF	-	1	8	FFFF	-	1	8
5					00dd+1	(00dd+1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	Y+ff	(Y+ff)	1	54
6									Hhll+1	(hhll+1)	1	15	X+ff+1	(X+ff+1)	1	48	Y+ff+1	(Y+ff+1)	1	55
	{20,27,40,42,0xf4,0xf5,0xf6,0xf7}				{20,27,41,1,3,0xf5,0xf6,0xf7}				{20,27,34,111,13,15,0xf6,0xf7}				{20,27,35,8,47,48,0xf6,0xf7}				{20,27,35,8,54,55,0xf6,0xf7}			

Table A.83 LSLA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	48	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.84 LSLB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	58	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.85 LSL Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	48	1	20	OP	68	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	68	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	Ff		35
4	Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	Hhll	Result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.86 LSLD Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	05	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.87 LSRA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	44	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.88 LSRB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	54	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.89 LSR Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	74	1	20	OP	64	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	64	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	Result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	Result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.90 LSRD Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	04	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.91 MUL Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	3D	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.92 NEGA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	40	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.93 NEGB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	50	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.94 NEG Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	70	1	20	OP	60	1	20	OP	18	1	20
2	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	60	1	27
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	result	0	14	X+ff	result	0	46	FFFF	-	1	8
7									Y+ff	result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.95 NOP Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	01	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.96 ORAA Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	8A	1	20	OP	9A	1	20	OP	BA	1	20	OP	AA	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	AA	1	27
3					00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.97 ORAB Cycle Table

Cycle	(IMM)				(DIR)				(EXT)				(IND,X)				(IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	CA	1	20	OP	DA	1	20	OP	FA	1	20	OP	EA	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	EA	1	27
3					00dd	(00dd)	1	1	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.98 PSHA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	36	1	20
2	OP+1	-	1	23
3	SP	(A)	0	64
4				
5				
6				
	{20,23,64,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.99 PSHB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	37	1	20
2	OP+1	-	1	23
3	SP	(B)	0	65
	{20,23,65,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.100 PSHX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	3C	1	20
2	OP+1	-	1	23
3	SP	(IXL)	0	66
4	SP-1	(IXH)	0	67
	{20,23,66,67,0xf4,0xf5,0xf6,0xf7}			

Table A.101 PSHY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	3C	1	27
3	OP+2	-		39
4	SP	(IXL)	0	68
5	SP-1	(IXH)	0	69
	{20,27,39,68,69,0xf5,0xf6,0xf7}			

Table A.102 PULA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	32	1	20
2	OP+1	-	1	23
3	SP	-	1	63
4	SP+1	Get A	1	70
	{20,23,63,70,0xf4,0xf5,0xf6,0xf7}			

Table A.103 PULB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	33	1	20
2	OP+1	-	1	23
3	SP	-	1	63
4	SP+1	Get B	1	71
	{20,23,63,71,0xf4,0xf5,0xf6,0xf7}			

Table A.104 PULX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	38	1	20
2	OP+1	-	1	23
3	SP	-	1	63
4	SP+1	Get IXH	1	72
5	SP+2	Get IXL	1	73
	{20,23,63,72,73,0xf5,0xf6,0xf7}			

Table A.105 PULY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	38	1	27
3	OP+2	-	1	39
4	SP	-	1	63
5	SP+1	Get IXH	1	74
6	SP+2	Get IXL	1	75
	{20,27,39,63,74,75,0xf6,0xf7}			

Table A.106 ROLA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	49	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.107 ROLB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	59	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.108 ROL Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	79	1	20	OP	69	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	69	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	Result	0	14	X+ff	Result	0	46	FFFF	-	1	8
7									Y+ff	Result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.109 RORA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	46	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.110 RORB Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	56	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.111 ROR Cycle Table

Cycle	EXT				IND,X				IND,Y			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	76	1	20	OP	66	1	20	OP	18	1	20
2	OP+1	Hh	1	26	OP+1	ff	1	25	OP+1	66	1	27
3	OP+2	Ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	hhll	result	0	14	X+ff	Result	0	46	FFFF	-	1	8
7									Y+ff	Result	0	53
	{20,26,37,13,8,14,0xf6,0xf7}				{20,25,8,47,8,46, 0xf6,0xf7}				{20,27,35,8,54,8,53,0xf7}			

Table A.112 RTI Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	3B	1	20
2	OP+1	-	1	23
3	SP	-	1	63
4	SP+1	Get CC	1	76
5	SP+2	Get B	1	77
6	SP+3	Get A	1	78
7	SP+4	Get IXH	1	79
8	SP+5	Get IXL	1	80
9	SP+6	Get IYH	1	81
10	SP+7	Get IYL	1	82
11	SP+8	Rtn hi	1	83
12	SP+9	Rtn lo	1	84
	{20,23,63,76,77,78,79,80,81,82,83,84}			

Table A.113 RTS Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	39	1	20
2	OP+1	-	1	23
3	SP	-	1	63
4	SP+1	Rtn hi	1	85
5	SP+2	Rtn lo	1	86
	{20,23,63,85,86,0xf5,0xf6,0xf7}			

Table A.114 SBA Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	10	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.115 SBCA Cycle Table

Cycle	SBCA (IMM)				SBCA (DIR)				SBCA (EXT)				SBCA (IND,X)				SBCA (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	82	1	20	OP	92	1	20	OP	B2	1	20	OP	A2	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A2	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.116 SBCB Cycle Table

Cycle	SBCB (IMM)				SBCB (DIR)				SBCB (EXT)				SBCB (IND,X)				SBCB (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C2	1	20	OP	D2	1	20	OP	F2	1	20	OP	E2	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	E2	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.117 SEC Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	0D	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.118 SEI Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	0F	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.119 SEV Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	0B	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.120 STAA Cycle Table

Cycle	STAA (DIR)				STAA (EXT)				STAA (IND,X)				STAA (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	97	1	20	OP	B7	1	20	OP	A7	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A7	1	27
3	00dd	(A)	1	4	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4					hhll	(A)	1	11	X+ff	(A)	1	56	FFFF	-	1	8
5													Y+ff	(A)	0	57
	{20,24,4,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,11,0xf4,0xf5,0xf6,0xf7}				{20,25,8,56,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,57,0xf5,0xf6,0xf7}			

Table A.121 STAB Cycle Table

Cycle	STAB (DIR)				STAB (EXT)				STAB (IND,X)				STAB (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	D7	1	20	OP	F7	1	20	OP	E7	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	Ff	1	25	OP+1	E7	1	27
3	00dd	(B)	1	5	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4					hhll	(B)	1	10	X+ff	(B)	1	58	FFFF	-	1	8
5													Y+ff	(B)	0	59
	{20,24,5,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,10,0xf4,0xf5,0xf6,0xf7}				{20,25,8,58,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,59,0xf5,0xf6,0xf7}			

Table A.122 STD Cycle Table

Cycle	STD (DIR)				STD (EXT)				STD (IND,X)				STD (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	DD	1	20	OP	FD	1	20	OP	ED	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	ED	1	27
3	00dd	(A)	1	4	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	00dd+1	(B)	1	7	hhll	(A)	1	11	X+ff	(A)	1	56	FFFF	-	1	8
5					hhll+1	(B)	1	17	X+ff+1	(B)	1	50	Y+ff	(A)	1	57
6													Y+ff+1	(B)	1	51
	{20,24,4,7,0xf4,0xf5,0xf6,0xf7}				{20,26,37,11,17,0xf5,0xf6,0xf7}				{20,25,8,56,50,0xf5,0xf6,0xf7}				{20,27,35,8,57,51,0xf6,0xf7}			

Table A.123 STOP Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	CF	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.124 STS Cycle Table

Cycle	STS (DIR)				STS (EXT)				STS (IND,X)				STS (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	9F	1	20	OP	BF	1	20	OP	AF	1	20	OP	18	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	AF	1	27
3	00dd	(SPH)	0	87	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	00dd+1	(SPL)	0	90	hhll	(SPH)	0	93	X+ff	(SPH)	0	99	FFFF	-	1	8
5					hhll+1	(SPL)	0	96	X+ff+1	(SPL)	0	102	Y+ff	(SPH)	0	105
6													Y+ff+1	(SPL)	0	108
	{20,24,87,90,0xf4,0xf5,0xf6,0xf7}				{20,26,37,93,96,0xf5,0xf6,0xf7}				{20,25,8,99,102,0xf5,0xf6,0xf7}				{20,27,35,8,105,108,0xf6,0xf7}			

Table A.125 STX Cycle Table

Cycle	STX (DIR)				STX (EXT)				STX (IND,X)				STX (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	DF	1	20	OP	FF	1	20	OP	EF	1	20	OP	CD	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	EF	1	27
3	00dd	(IXH)	0	88	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	00dd+1	(IXL)	0	91	hhll	(IXH)	0	94	X+ff	(IXH)	0	100	FFFF	-	1	8
5					hhll+1	(IXL)	0	97	X+ff+1	(IXL)	0	103	Y+ff	(IXH)	0	106
6													Y+ff+1	(IXL)	0	109
	{20,24,88,91,0xf4,0xf5,0xf6,0xf7}				{20,26,37,94,97,0xf5,0xf6,0xf7}				{20,25,8,100,103,0xf5,0xf6,0xf7}				{20,27,35,8,106,109,0xf6,0xf7}			

Table A.126 STY Cycle Table

Cycle	STY (DIR)				STY (EXT)				STY (IND,X)				STY (IND,Y)			
	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	DF	1	20	OP	FF	1	20	OP	EF	1	20	OP	CD	1	20
2	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	EF	1	27
3	00dd	(IYH)	0	89	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4	00dd+1	(IYL)	0	92	hhll	(IYH)	0	95	X+ff	(IYH)	0	101	FFFF	-	1	8
5					hhll+1	(IYL)	0	98	X+ff+1	(IYL)	0	104	Y+ff	(IYH)	0	107
6													Y+ff+1	(IYL)	0	110
	{20,24,89,92,0xf4,0xf5,0xf6,0xf7}				{20,26,37,95,98,0xf5,0xf6,0xf7}				{20,25,8,101,104,0xf5,0xf6,0xf7}				{20,27,35,8,107,110,0xf6,0xf7}			

Table A.127 SUBA Cycle Table

Cycle	SUBA (IMM)				SUBA (DIR)				SUBA (EXT)				SUBA (IND,X)				SUBA (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	80	1	20	OP	90	1	20	OP	B0	1	20	OP	A0	1	20	OP	18	1	20
2	OP+1	Ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	ff	1	25	OP+1	A0	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ff	1	35
4									Hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.128 SUBB Cycle Table

Cycle	SUBB (IMM)				SUBB (DIR)				SUBB (EXT)				SUBB (IND,X)				SUBB (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	C0	1	20	OP	D0	1	20	OP	F0	1	20	OP	E0	1	20	OP	18	1	20
2	OP+1	ii	1	28	OP+1	dd	1	24	OP+1	hh	1	26	OP+1	Ff	1	25	OP+1	E0	1	27
3					00dd	(00dd)	1	1	OP+2	ll	1	37	FFFF	-	1	8	OP+2	ff	1	35
4									hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5																	Y+ff	(Y+ff)	1	54
	{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7}				{20,26,37,13,0xf4,0xf5,0xf6,0xf7}				{20,25,8,47,0xf4,0xf5,0xf6,0xf7}				{20,27,35,8,54,0xf5,0xf6,0xf7}			

Table A.129 SUBD Cycle Table

Cycle	SUBD (IMM)				SUBD (DIR)				SUBD (EXT)				SUBD (IND,X)				SUBD (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C	A	D	RW	C	A	D	RW	C
1	OP	83	1	20	OP	93	1	20	OP	F0	1	20	OP	E0	1	20	OP	18	1	20
2	OP+1	jj	1	29	OP+1	Dd	1	24	OP+1	hh	1	26	OP+1	Ff	1	25	OP+1	E0	1	27
3	OP+2	kk	1	36	00dd	(00dd)	1	1	OP+2	ll	1	34	FFFF	-	1	8	OP+2	ff	1	35
4	FFFF	-	1	8	00dd +1	(00dd +1)	1	3	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5					FFFF	-	1	8	Hhll +1	(hhll +1)	1	15	X+ff +1	(X+ff +1)	1	48	Y+ff	(Y+ff)	1	54
6									FFFF	-	1	8	FFFF	-	1	8	Y+ff +1	(Y+ff +1)		55
7																	FFFF	-	1	8
	{20,29,36,8,0xf4,0xf5,0xf6,0xf7}				{20,24,1,3,8,0xf5,0xf6,0xf7}				{20,26,34,13,15,8,0xf6,0xf7}				{20,25,8,47,48,8,0xf6,0xf7}				{20,27,35,8,54,55,8,0xf7}			

Table A.130 SWI Cycle Table

Cycle	SWI (INH)			
	Addr	Data	R/W	Code
1	OP	3F	1	
2	OP+1	-	1	
3	SP	Rtn lo	0	
4	SP-1	Rtn hi	0	
5	SP-2	(IYL)	0	
6	SP-3	(IYH)	0	
7	SP-4	(IXL)	0	
8	SP-5	(IXH)	0	
9	SP-6	(A)	0	
10	SP-7	(B)	0	
11	SP-8	(CCR)	0	
12	SP-8	(CCR)	1	
13	Vec hi	Svc hi	1	
14	Vec lo	Svc lo	1	

Table A.131 TAB Cycle Table

Cycle	TAB (INH)			
	Addr	Data	R/W	Code
1	OP	16	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.132 TAP Cycle Table

Cycle	TAP (INH)			
	Addr	Data	R/W	Code
1	OP	06	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.133 TBA Cycle Table

Cycle	TBA (INH)			
	Addr	Data	R/W	Code
1	OP	17	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.134 TEST Cycle Table

Cycle	TEST (INH)			
	Addr	Data	R/W	Code
1	OP	00	1	
2	OP+1	-	1	
3	OP+2	-	1	
4	OP+3	-	1	
5	PREV-1	(PREV-1)	1	

Table A.135 TPA Cycle Table

Cycle	TPA (INH)			
	Addr	Data	R/W	Code
1	OP	00	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.136 TSTA Cycle Table

Cycle	TSTA (INH)			
	Addr	Data	R/W	Code
1	OP	4D	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.137 TSTB Cycle Table

Cycle	TSTB (INH)			
	Addr	Data	R/W	Code
1	OP	5D	1	20
2	OP+1	-	1	23
	{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.138 TST Cycle Table

Cycle	TST (EXT)				TST (IND,X)				TST (IND,Y)			
	Addr	Data	R/W	Code	A	D	RW	C	A	D	RW	C
1	OP	7D	1	20	OP	7D	1	20	OP	7D	1	20
2	OP+1	hh	1	26	OP+1	hh	1	26	OP+1	hh	1	26
3	OP+2	ll	1	37	FFFF	-	1	8	OP+2	Ll	1	37
4	hhll	(hhll)	1	13	X+ff	(X+ff)	1	47	FFFF	-	1	8
5	FFFF	-	1	8	FFFF	-	1	8	Y+ff	(Y+ff)	1	54
6	FFFF	-	1	8	FFFF	-	1	8	FFFF	-	1	8
7									FFFF	-	1	8
	{20,26,37,13,8,8,0xf6,0xf}				{20,26,8,47,8,8,0xf6,0xf}				{20,26,37,8,54,8,8,0xf}			

Table A.139 TSX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	30	1	20
2	OP+1	-	1	23
3	SP	-	1	63
	{20,23,63,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.140 TSY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	30	1	27
3	OP+2	-	1	39
4	SP	-	1	63
	{20,27,39,63,0xf4,0xf5,0xf6,0xf7}			

Table A.141 TXS Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	35	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.142 TYS Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	35	1	27
3	OP+2	-	1	39
4	FFFF	-	1	8
	{20,27,39,8,0xf4,0xf5,0xf6,0xf7}			

Table A.143 WAI Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	3E	1	
2	OP+1	-	1	
3	SP	Rtn lo	0	
4	SP-1	Rtn hi	0	
5	SP-2	(IYL)	0	
6	SP-3	(IYH)	0	
7	SP-4	(IXL)	0	
8	SP-5	(IXH)	0	
9	SP-6	(A)	0	
10	SP-7	(B)	0	
11	SP-8	(CCR)	0	
12	SP-8	(CCR)	1	
13	Vec hi	Svc hi	1	
14	Vec lo	Svc lo	1	

Table A.144 XGDX Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	8F	1	20
2	OP+1	-	1	23
3	FFFF	-	1	8
	{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7}			

Table A.145 XGDY Cycle Table

Cycle				
	Addr	Data	R/W	Code
1	OP	18	1	20
2	OP+1	8F	1	27
3	OP+2	-	1	39
4	FFFF	-	1	8
	{20,27,39,8,0xf4,0xf5,0xf6,0xf7}			

APPENDIX B – Source Code

prj.cpp

```
#include <vcl.h>
#include <stdio.h>
#pragma hdrstop

#include "prj.h"
#include "editor.h"
#include "MemoryContent.h"
#include "splash.h"
#include "Getinput.h"
#include "StackView.h"

#include "typedef.h"
#include "asmLine.h"
#include "OpCode.h"
#include "OpParam.h"
#include "funcs.h"
#include "linked.h"
#include "micro.hpp"
#include "reg16bit.h"

Micro Motorola;
UCHAR MachineCycle = 0;
UCHAR InCycle = 0;
UINT focusedLine=0;
UINT focusedCycle=0;
UINT IntVectorAddress=0xa000;
bool InterruptRequest=false;
bool InitInterrupt=false;
UINT SPVector=0x8000;
UINT PCVector=0x1000;
UINT *GetInputdest;

//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
extern PACKAGE TRAMPool *RAMPool;
extern PACKAGE TEditorForm *EditorForm;
extern TSplashForm *SplashForm;
extern TForm3 *Form3;
extern TStackViewer *StackViewer;

FILE *inFile;       /* Input file */
LinkedList<asmLine> LineList;
TimerStruct MainTimer;

//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    fclose (inFile);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ExecutionTimerTimer(TObject *Sender)
{
    switch (MainTimer.State){
        case TIMER_IDLE:
            break;
        case TIMER_EXECUTE_ALL:
            ExecuteFile();
            break;
        case TIMER_EXECUTE_INST:
            ExecuteCurrentPC();
            HighlightNextExecution(Motorola.PC.value());
            MainTimer.State=TIMER_EXECUTE_WAITKEY;
            break;
        case TIMER_EXECUTE_CYCLE:
            ExecuteCurrentCycle();
            MainTimer.State=TIMER_EXECUTE_WAITKEY;
            break;
        case TIMER_EXECUTE_WAITKEY:
            break;
    }
    UpdateMemoryLocations();
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Open1Click(TObject *Sender)
{
    if (OpenDialog1->Execute()){
        Edit1->Text = OpenDialog1->FileName;
        inFile = fopen(OpenDialog1->FileName.c_str(), "r");
```

```
          if (inFile == NULL)
            EditorForm->Editor->Lines->Add ("Can not open file");
        else{
            EditorForm->Editor->Lines->Clear();
            EditorForm->Editor->Lines->Add("File Opened");

            processFile();
            MainTimer.State = TIMER_EXECUTE_WAITKEY;
            ExecutionTimer->Enabled = true;
        }
    }
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //[col][row]

    SplashForm = new TSplashForm(Application);
    SplashForm->Update();
    Sleep(1500);

    TGridRect myRect;
          myRect.Left = -1;
          myRect.Top = -1;
          myRect.Right = -1;
          myRect.Bottom = -1;
          RegisterGrid->Selection = myRect;
          CCRGrid->Selection = myRect;


    RegisterGrid->Cells[0][0] = "ACCA";
    RegisterGrid->Cells[0][1] = "ACCB";
    RegisterGrid->Cells[0][2] = "IX";
    RegisterGrid->Cells[0][3] = "IY";
    RegisterGrid->Cells[0][4] = "PC";
    RegisterGrid->Cells[0][5] = "SP";
    RegisterGrid->Cells[0][6] = "CCR";
    RegisterGrid->ColWidths[0]=82;
    RegisterGrid->ColWidths[1]=82;

    AddrDataGrid->Cells[0][0] = "Address Bus";
    AddrDataGrid->Cells[0][1] = "Data Bus";
    AddrDataGrid->Cells[0][2] = "R/W";
    AddrDataGrid->ColWidths[0]=82;
    AddrDataGrid->ColWidths[1]=82;

    CCRGrid->Cells[0][0] = "S";
    CCRGrid->Cells[1][0] = "X";
    CCRGrid->Cells[2][0] = "H";
    CCRGrid->Cells[3][0] = "I";
    CCRGrid->Cells[4][0] = "N";
    CCRGrid->Cells[5][0] = "Z";
    CCRGrid->Cells[6][0] = "V";
    CCRGrid->Cells[7][0] = "C";
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Memory1Click(TObject *Sender)
{
    if (RAMPool->Visible == false){
        DumpMemorytoGrid();
        RAMPool->Visible = true;
    }
}
//---------------------------------------------------------------------------
TTextAttributes *__fastcall TForm1::CurrText(void)
{
    return EditorForm->Editor->SelAttributes;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Editor1Click(TObject *Sender)
{
    EditorForm->Visible=true;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ExecuteCycle1Click(TObject *Sender)
{
    if (MainTimer.State==TIMER_EXECUTE_WAITKEY)
        MainTimer.State=TIMER_EXECUTE_CYCLE;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ExecuteInstruction1Click(TObject *Sender)
{
    if (MainTimer.State==TIMER_EXECUTE_WAITKEY)
        MainTimer.State=TIMER_EXECUTE_INST;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ExecuteAll1Click(TObject *Sender)
{
    if (MainTimer.State==TIMER_EXECUTE_WAITKEY)
        MainTimer.State=TIMER_EXECUTE_ALL;
    else if (MainTimer.State==TIMER_EXECUTE_ALL)
```

```
        MainTimer.State=TIMER_EXECUTE_WAITKEY;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ExitClick(TObject *Sender)
{
    Form1->Close();
}
//---------------------------------------------------------------------------

void __fastcall TForm1::NMIAddress1Click(TObject *Sender)
{
    Form3->Show();
    //Visible = true;
    Form3->Caption = "Interrupt Vector Address";
    Form3->Edit1->Text = "0x"+IntToHex((int)IntVectorAddress,4);
    GetInputdest=&IntVectorAddress;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (InterruptRequest==false && InitInterrupt==false){
        InitInterrupt=true;
        ExecuteCurrentPC();
        Button1->Enabled = false;
    }

}
//---------------------------------------------------------------------------

void __fastcall TForm1::StackAddress1Click(TObject *Sender)
{
    Form3->Show();
    //Visible = true;
    Form3->Caption = "Stack Pointer Address";
    Form3->Edit1->Text = "0x"+IntToHex((int)SPVector,4);
    GetInputdest=&SPVector;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::StartVector1Click(TObject *Sender)
{
    Form3->Show();
    //Visible = true;
    Form3->Caption = "Stack Pointer Address";
    Form3->Edit1->Text = "0x"+IntToHex((int)PCVector,4);
    GetInputdest=&PCVector;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Stack1Click(TObject *Sender)
{
    UINT I, J, K;
    int rcount=SPVector-Motorola.SP.value();
    StackViewer->StringGrid1->RowCount = rcount;
    StackViewer->StringGrid1->Height = StackViewer->StringGrid1->DefaultRowHeight*(rcount+1);
    StackViewer->Panel1->Height = StackViewer->StringGrid1->Height;

    K=0;
    for (J = SPVector; J >=Motorola.SP.value(); J--){
        StackViewer->StringGrid1->Cells[0][K++] = IntToHex((int)J,4);
    }

    K=0;
    for (J = SPVector; J >=Motorola.SP.value(); J--){
        StackViewer->StringGrid1->Cells[1][K++] = IntToHex(*(Motorola.itsMem+J),2);
    }
    StackViewer->Show();
}
//---------------------------------------------------------------------------
```

funcs2.cpp

```
#include "prj.h"
#include "MemoryContent.h"
#include "editor.h"


#include <StdCtrls.hpp>
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "def.h"
#include "typedef.h"
#include "linked.h"
#include "asmLine.h"

#include "funcs.h"
#include "Micro.hpp"

extern Micro Motorola;
extern UCHAR MachineCycle;
extern UCHAR InCycle;
extern UINT focusedLine;
extern UINT focusedCycle;
extern UINT IntVectorAddress;
extern bool InterruptRequest;
extern bool InitInterrupt;
extern UINT PCVector;

#include "instTbl.h"

short int tableSize = sizeof(instTable)/sizeof(instruction);
bool NMIInterrupt=false;

extern PACKAGE TForm1 *Form1;
extern PACKAGE TRAMPool *RAMPool;
extern PACKAGE TEditorForm *EditorForm;

extern FILE *inFile;                    /* Input file */
extern LinkedList<asmLine> LineList;
extern TimerStruct MainTimer;

extern bool IsLabel(char **line, asmLine *asmLPtr);
extern asmLine * SearchCurrentPCLine(LinkedList<asmLine> *LList, UINT currentPC);
extern void PrintToCodeView(LinkedList<asmLine> *LList);
extern void LoadMachineCycleList(UINT currentPC);
extern void HighlightNextCycle(UINT mID);
extern void UpdateAddressDataBusView(void);

//---------------------------------------------------------------------------

int       processFile()
{
char capLine[256];
char line[256];
char *lineData;
char *lPtr;
asmLine *currentLine;
opDescriptor source, dest;
int errorPtr;
bool itsLabel;
bool itsInstruction;
UINT itsExecNo = 0;
UINT PCassign = PCVector;
UINT Vectorassign = IntVectorAddress;


    MachineCycle=0;

    while(fgets(line, 255, inFile)){
        currentLine = new asmLine(itsExecNo, (char *)&line);
        itsExecNo++;
        currentLine->hcodes.len=0;


        errorPtr = 0;
        strcap(capLine, line);
        lineData = (char *)&capLine;
        itsLabel = IsLabel (&lineData, currentLine);
        itsInstruction = instLookup(&lineData, currentLine);

        if (lineData){
            if (itsInstruction){
                parseArguments(lineData, &errorPtr, currentLine);
                if (currentLine->itsOpCode->itsInstruction->branch==1)
                    currentLine->addrMode = ADDR_INHERENT;
                currentLine->itsNMIInterrupt=NMIInterrupt;
            }
        }

        if (!itsLabel && !itsInstruction){
            delete currentLine;
            itsExecNo--;
        }
        else{
            if (NMIInterrupt == false){
```

```
                UINT i = currentLine->assignMemLoc(PCassign);
                if (itsInstruction){
                    querySourceCodes(currentLine);
                    insertSourceCodes(currentLine,PCassign);
                }

                if (i == 0xff)
                    ;
                else{
                    PCassign += i;
                }
            }//end of non interrupt instructions
            else{
                UINT i = currentLine->assignMemLoc(Vectorassign);
                if (itsInstruction){
                    querySourceCodes(currentLine);
                    insertSourceCodes(currentLine,Vectorassign);
                }
                if (i == 0xff)
                    ;
                else{
                    Vectorassign += i;
                }
            }
            if (itsInstruction)
                if (!strcmp(currentLine->itsOpCode->itsInstruction->mnemonic,"RTI"))
                    NMIInterrupt=0;
            LineList.Insert(*currentLine);
        }
    }/*end while*/

    LineList.PrintList(EditorForm->Editor);
    PrintToCodeView(&LineList);
    LoadMachineCycleList(Motorola.PC.value());
          return 1;
}

//---------------------------------------------------------------------------
int       strcap(char *d, char *s)
{
char capFlag;

          capFlag = -1;
          while (*s) {
                    if (capFlag)
                              *d = toupper(*s);
                    else
                              *d = *s;
                    if (*s == '\'')
                              capFlag = (char) !capFlag;
                    d++;
                    s++;
                    }
          *d = '\0';

          return 1;

}
//---------------------------------------------------------------------------
char *skipSpace(char *p)
{
          while (isspace(*p))
                    p++;
          return p;
}
//---------------------------------------------------------------------------
void ownTrimRight(char *p)
{
    while (*p != '\0'){
        if (isspace(*p) || (*p == '\r') || (*p == '\n') || (*p == ','))
            *p = '\0';
        else
            p++;
    }
}
//---------------------------------------------------------------------------
bool IsLabel(char **line, asmLine *asmLPtr)
{
char *p, *start;
UINT i;
char label[LABELSIGCHARS+1];

          p = start = skipSpace(*line);
          if (*p && *p != '*'){
                    i = 0;
                    do {
                              if (i < LABELSIGCHARS)
                                        label[i++] = *p;
                              p++;
                    } while (isalnum(*p) || *p == '.' || *p == '_' || *p == '$');
                    label[i] = '\0';

                    if ((isspace(*p) && start == *line) || *p == ':' || *p == '*' ) {
                              if (*p == ':')
                                        p++;
                              p = skipSpace(p);
            *line = p;
            if (!strcmp(label, "NMINTERRUPT")){
```

```
                NMIInterrupt=true;
                return 0;
            }
            else{
                asmLPtr->assignLabel(label);
            }
            return 1;
        }
    }
    return 0;
}
//---------------------------------------------------------------------------
bool instLookup(char **line, asmLine *asmLPtr)
{
    char *p, *start;
    char opcode[8];
    int i, hi, lo, mid, cmp;

          p = start = skipSpace(*line);
    if (p == NULL)
        return 0;
          i = 0;
          do {
                    if (i < 7)
                              opcode[i++] = *p;
                    p++;
          } while (isalpha(*p));
          opcode[i] = '\0';

              lo = 0;
        hi = tableSize;
          hi = int (tableSize - 1);
              do {
                    mid = (hi + lo) / 2;
                    cmp = strcmp(opcode, instTable[mid].mnemonic);
                        if (cmp > 0)
                                  lo = mid + 1;
                    else if (cmp < 0)
                                hi = mid - 1;
          } while (cmp && (hi >= lo));
              if (!cmp) {
            asmLPtr->itsOpCode->itsInstruction = (instruction *)&instTable[mid];
            asmLPtr->id = mid;
            *line = p;
                        return 1;
        }
    *line = start;
          return 0;
}
//---------------------------------------------------------------------------
void parseArguments(char *p, int *errorPtr, asmLine *curLine)
{
    AnsiString temp;

    opDescriptor tempopdesc;

    *errorPtr = 0;
    resetOpDescriptor (curLine->itsOpCode->source);
    resetOpDescriptor (curLine->itsOpCode->dest);
    resetOpDescriptor (curLine->itsOpCode->mask);
    resetOpDescriptor (curLine->itsOpCode->rel);
    resetOpDescriptor (&tempopdesc);
    p = opParse(p, errorPtr, &tempopdesc, curLine, 0);
    *curLine->itsOpCode->source = tempopdesc;

    if (*errorPtr > ERROR)
        return;
    if (p == NULL)
        return;

    if (*p != ',' && *p != ' ') {//' ' is for BCLR
        NEWERROR(*errorPtr, SYNTAX);
        return;
    }
    p++;
    resetOpDescriptor (&tempopdesc);
    p = opParse(p, errorPtr, &tempopdesc, curLine, 1);
    *curLine->itsOpCode->dest = tempopdesc;

    if (*errorPtr > SEVERE)
        return;

    if (*p != ' ') {
        NEWERROR(*errorPtr, SYNTAX);
        return;
    }

    if (p == NULL)
        return;
    p++;
    resetOpDescriptor (&tempopdesc);
    p = opParse(p, errorPtr, &tempopdesc, curLine, 2);
    *curLine->itsOpCode->mask = tempopdesc;

    if (p == NULL)
        return;
    p++;
    resetOpDescriptor (&tempopdesc);
```

```
    p = opParse(p, errorPtr, &tempopdesc, curLine, 3);
    *curLine->itsOpCode->rel = tempopdesc;
}
//---------------------------------------------------------------------------
void resetOpDescriptor(opDescriptor *dest)
{
    dest->data = OPDESC_DEF_DATA;
    dest->backRef = OPDESC_DEF_BACKREF;
    dest->defined = 0;
}
//---------------------------------------------------------------------------
void ExecuteFile(void)
{
    ExecuteCurrentPC();
}
//---------------------------------------------------------------------------
void UpdateMemoryLocations(void)
{
    int I, J, K;
    Form1->RegisterGrid->Cells[1][0] = IntToHex(Motorola.ACCA,2);
    Form1->RegisterGrid->Cells[1][1] = IntToHex(Motorola.ACCB,2);
    Form1->RegisterGrid->Cells[1][2] = IntToHex((int)Motorola.IX.value(),4);
    Form1->RegisterGrid->Cells[1][3] = IntToHex((int)Motorola.IY.value(),4);
    Form1->RegisterGrid->Cells[1][4] = IntToHex((int)Motorola.PC.value(),4);
    Form1->RegisterGrid->Cells[1][5] = IntToHex((int)Motorola.SP.value(),4);
    Form1->RegisterGrid->Cells[1][6] = IntToHex(Motorola.CCR.value(),2);

    Form1->CCRGrid->Cells[0][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_STOP),1);
    Form1->CCRGrid->Cells[1][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_XINTERRUPT),1);
    Form1->CCRGrid->Cells[2][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_HALFCARRY),1);
    Form1->CCRGrid->Cells[3][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_INTERRUPT),1);
    Form1->CCRGrid->Cells[4][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_NEGATIVE),1);
    Form1->CCRGrid->Cells[5][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_ZERO),1);
    Form1->CCRGrid->Cells[6][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_OVERFLOW),1);
    Form1->CCRGrid->Cells[7][1] = IntToHex((int)Motorola.CCR.GetBit(BIT_CARRY),1);
}
//---------------------------------------------------------------------------
void ExecuteCurrentCycle(void)
{
    UINT currentPC = Motorola.PC.value();
    static asmLine * curasmLine;
    static asmLine * nextasmLine;
    char errstr[200];

    if (!InCycle){//The place for the first process
        MachineCycle = 0;
        curasmLine = SearchCurrentPCLine(&LineList, currentPC);
        if (curasmLine != NULL){
            updateMicroParameters(curasmLine);
        }
    }

    if (curasmLine != NULL){
        if (curasmLine->itsOpCode->itsInstruction->exec)
            if (curasmLine->itsOpCode->itsInstruction->hexcode[ConvertModeVal(curasmLine->addrMode)]>0){
                InCycle = curasmLine->itsOpCode->itsInstruction->exec(curasmLine);
                if (!InCycle){//end of all cycles
                    Motorola.UpdatePC(curasmLine);
                    InterruptLookup();
                    HighlightNextExecution(Motorola.PC.value());
                    LoadMachineCycleList(Motorola.PC.value());
                }
                else if (InCycle == 0xfe){//Halt the system by TEST command
                    MainTimer.State = TIMER_IDLE;
                    sprintf (errstr, "System Halted By Test Operation");
                    ErrorMessage(errstr, "System Halted");
                }
                else if (InCycle == 0xff){//Halt the system by STOP command
                    MainTimer.State = TIMER_IDLE;
                    sprintf (errstr, "System Halted By Stop Operation");
                    ErrorMessage(errstr, "System Halted");
                }
                else if (InCycle == 1){
                    HighlightNextCycle(MachineCycle);
                }
                UpdateAddressDataBusView();
            }
            else{
                MainTimer.State = TIMER_IDLE;
                sprintf (errstr, "Mnemonic: %s \nAddressing Mode: %s",\
                        curasmLine->itsOpCode->itsInstruction->mnemonic,\
                        ConvertModeValtoStr(curasmLine->addrMode));
                ErrorMessage(errstr, "Addressing Mode Error");
            }
    }
    else{
    }
}
//---------------------------------------------------------------------------
void ExecuteCurrentPC(void)
{
    if (!InCycle)//If not entered any cycle, make it enter cycle
        ExecuteCurrentCycle();
    while (InCycle)
        ExecuteCurrentCycle();
}
//---------------------------------------------------------------------------
asmLine * SearchCurrentPCLine(LinkedList<asmLine> *LList, UINT currentPC)
```

```
{
    LList->resetCurrent();
    while(LList->current != NULL){
        if (LList->current->data.itsMemLoc == currentPC){
            if (LList->current->data.itsOpCode->itsInstruction != NULL)//label + instruction or only instruction
                return &LList->current->data;
            else{//only label
                ;
            }
        }
        LList->current = LList->current->nextNode();
    }
    return NULL;
}
//---------------------------------------------------------------------------
asmLine * SearchLabel(char *srchLbl)
{
    LinkedList<asmLine> *LList = &LineList;
    LList->resetCurrent();
    while(LList->current != NULL){
        if (LList->current->data.itsLabel){//Label exists
            if (!strcmp (LList->current->data.itsLbl, srchLbl)){//searched string??
                return &LList->current->data;
            }
        }
        LList->current = LList->current->nextNode();
    }
    return NULL;
}
//---------------------------------------------------------------------------
void ASMOutput(LinkedList<asmLine> *LList)
{
    LList->resetCurrent();
    while(LList->current != NULL){
        LList->current = LList->current->nextNode();
    }
}
//---------------------------------------------------------------------------
UCHAR ConvertModeVal(UCHAR mode)
{
    switch (mode){
        case ADDR_IMMEDIATE:
            return 0;
        case ADDR_INDEXEDX:
            return 3;
        case ADDR_INDEXEDY:
            return 4;
        case ADDR_DIRECT:
            return 1;
        case ADDR_EXTENDED:
            return 2;
        case ADDR_INHERENT:
            return 5;
    }
    return 0xff;
}
//---------------------------------------------------------------------------
char *ConvertModeValtoStr(UCHAR mode)
{
    switch (mode){
        case ADDR_IMMEDIATE:
            return "Immediate";
        case ADDR_INDEXEDX:
            return "IndexedX";
        case ADDR_INDEXEDY:
            return "IndexedY";
        case ADDR_DIRECT:
            return "Direct";
        case ADDR_EXTENDED:
            return "Extended";
        case ADDR_INHERENT:
            return "Inherent";
    }
    return "Not Defined";
}
//---------------------------------------------------------------------------
void updateMicroParameters(asmLine * curasmLine)
{
    Motorola.DPTRa = curasmLine->itsOpCode->source->data;
    Motorola.DPTRb = curasmLine->itsOpCode->dest->data;
    Motorola.DPTRc = curasmLine->itsOpCode->mask->data;
    Motorola.ii = Motorola.DPTRa.valueL();//One byte immediate data

    Motorola.jj = Motorola.DPTRa.valueH();//High order byte of 16-bit immediate data
    Motorola.kk = Motorola.DPTRa.valueL();//Low order byte of 16-bit immediate data

    Motorola.hh = Motorola.DPTRa.valueH();//High order byte of 16-bit extended address
    Motorola.ll = Motorola.DPTRa.valueL();//Low order byte of 16-bit extended address

    Motorola.dd = Motorola.DPTRa.valueL();//Low order 8-bits of direct address

    Motorola.mm = Motorola.DPTRb.valueL();//8-bit mask (set bits correspond to operand bits which will be
affected)

    Motorola.ff = Motorola.DPTRa.valueL();//8-bit forward offset $00 (0) to $FF (255) (is added to index)
    Motorola.rr = Motorola.DPTRa.valueL();//Signed relative offset $80 (-128) to $7F (+127) (offset relative to
address following machine code offset byte)
    if (!strcmp(curasmLine->itsOpCode->itsInstruction->mnemonic,"BRCLR")||\
```

```
        !strcmp(curasmLine->itsOpCode->itsInstruction->mnemonic,"BRSET"))
        Motorola.rr = Motorola.DPTRc.valueL();//Signed relative offset $80 (-128) to $7F (+127) (offset relative
to address following machine code offset byte)

    Motorola.MM = 0xff;
    if (!strcmp(curasmLine->itsOpCode->itsInstruction->mnemonic,"BCLR")){
        if (curasmLine->addrMode == ADDR_DIRECT)
            Motorola.MM = curasmLine->itsOpCode->dest->data;
        else
            Motorola.MM = curasmLine->itsOpCode->mask->data;
    }
}
//---------------------------------------------------------------------------
void ErrorMessage(char *errstr, char *caption)
{
    if ( Application->MessageBox(errstr, caption, MB_OK) == IDOK)
        exit(1);
}
//---------------------------------------------------------------------------
void UpdateGridLocation(UCHAR *src)
{
    UINT destGrid = UINT(src-(UCHAR *)&Motorola.itsMem);
    UINT destRow = destGrid/16 + 1;
    UINT destCol = destGrid%16 + 1;
    RAMPool->StringGrid1->Cells[destCol][destRow] = IntToHex(*(Motorola.itsMem+destGrid),2);
}
//---------------------------------------------------------------------------
void UpdateGridLocation2(UCHAR *src, UCHAR data)
{
    UINT destGrid = UINT(src-(UCHAR *)&Motorola.itsMem);
    UINT destRow = destGrid/16 + 1;
    UINT destCol = destGrid%16 + 1;
    RAMPool->StringGrid1->Cells[destCol][destRow] = data;
}
//---------------------------------------------------------------------------
void DumpMemorytoGrid(void)
{
    int I, J, K;
    for (J = 1; J < RAMPool->StringGrid1->RowCount; J++)
        for (I = 1; I < RAMPool->StringGrid1->ColCount; I++){
            K = (J-1)*16 + (I-1);
            RAMPool->StringGrid1->Cells[I][J] = IntToHex(*(Motorola.itsMem+K),2);
        }
}
//---------------------------------------------------------------------------
void querySourceCodes(asmLine *currLine)
{
    UINT hexcode = currLine->itsOpCode->itsInstruction->hexcode[ConvertModeVal(currLine->addrMode)];
    if (hexcode>0xff)
        currLine->hcodes.cmd[currLine->hcodes.len++]=hexcode/0x100;
    currLine->hcodes.cmd[currLine->hcodes.len++]=hexcode%0x100;
    if(currLine->itsOpCode->source->defined==1){
        if (currLine->itsOpCode->source->data>0xff)
            currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->source->data/0x100;
        currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->source->data%0x100;
    }
    if(currLine->itsOpCode->dest->defined==1){
        if (currLine->itsOpCode->dest->data>0xff)
            currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->dest->data/0x100;
        currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->dest->data%0x100;
    }
    if(currLine->itsOpCode->mask->defined==1){
        if (currLine->itsOpCode->mask->data>0xff)
            currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->mask->data/0x100;
        currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->mask->data%0x100;
    }
    if(currLine->itsOpCode->rel->defined==1){
        if (currLine->itsOpCode->rel->data>0xff)
            currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->rel->data/0x100;
        currLine->hcodes.cmd[currLine->hcodes.len++]=currLine->itsOpCode->rel->data%0x100;
    }
}
//---------------------------------------------------------------------------
void insertSourceCodes(asmLine *currLine, UINT PCassign)
{
    for (UINT count=0;count<currLine->hcodes.len;count++)
        Motorola.itsMem[PCassign+count]=currLine->hcodes.cmd[count];
}
//---------------------------------------------------------------------------
void PrintToCodeView(LinkedList<asmLine> *LList)
{
    TListItem *newItem;
    UINT lcount=1;
    AnsiString tstr;

    LList->resetCurrent();
    while(LList->current != NULL){
        LList->current->data.LViewId = Form1->CodeView->Items->Count;
        newItem = Form1->CodeView->Items->Add();//1.col

        newItem->ImageIndex = LList->current->data.itsNMIInterrupt?7:-1;
        newItem->SubItems->Add("");
        newItem->SubItemImages[0] = -1;

        newItem->SubItems->Add(IntToStr(lcount++));
        newItem->SubItemImages[1] = -1;

        newItem->SubItems->Add(IntToHex((int)LList->current->data.itsMemLoc, 4));
```

```
        newItem->SubItemImages[2] = -1;

        tstr = "";
        for (UINT count=0;count<LList->current->data.hcodes.len;count++){
            tstr = tstr+IntToHex((int)LList->current->data.hcodes.cmd[count], 2);
            tstr = tstr+" ";
        }
        newItem->SubItems->Add(tstr);
        newItem->SubItemImages[3] = -1;

        if (LList->current->data.itsLabel)
            newItem->SubItems->Add(LList->current->data.itsLbl);
        else
            newItem->SubItems->Add("");
        newItem->SubItemImages[4] = -1;

        if (LList->current->data.itsOpCode->itsInstruction != NULL){
            tstr = LList->current->data.itsLineStr;
            newItem->SubItems->Add(tstr.Trim());
        }
        else
            newItem->SubItems->Add("");
        newItem->SubItemImages[5] = -1;

        LList->current = LList->current->nextNode();
    }
    HighlightNextExecution(Motorola.PC.value());
}
//---------------------------------------------------------------------------
void HighlightNextExecution(UINT currentPC)
{
    asmLine * curasmLine;

    Form1->CodeView->Items->Item[focusedLine]->SubItemImages[0]=-1;
    curasmLine = SearchCurrentPCLine(&LineList, currentPC);
    if (!curasmLine){
        MainTimer.State=TIMER_IDLE;
        if ( Application->MessageBox("End of All Items", "Warning", MB_OK) == IDOK)
            ;
    }
    else{
        focusedLine=curasmLine->LViewId;
        if (focusedLine>Form1->CodeView->Items->Count){
            MainTimer.State=TIMER_IDLE;
            if ( Application->MessageBox("End of All Items", "Warning", MB_OK) == IDOK)
                ;
        }
        else
            Form1->CodeView->Items->Item[focusedLine]->SubItemImages[0]=1;
    }
}
//---------------------------------------------------------------------------
void LoadMachineCycleList(UINT currentPC)
{
    char* szBuffer = new char[512];
    asmLine * curLine;

    curLine = SearchCurrentPCLine(&LineList, currentPC);

    TListItem *newItem;
    UINT id;
    UINT mode;
    UINT mID;

    if (!curLine){
        MainTimer.State=TIMER_IDLE;
        Form1->CycleView->Items->Clear();
        return;
    }
    id = curLine->id;
    mode = ConvertModeVal(curLine->addrMode);

    if (Form1->CycleView->Items->Count)
        Form1->CycleView->Items->Clear();
    for (UINT count=0;count<instTable[id].cycle[mode];count++){
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[count];
        newItem = Form1->CycleView->Items->Add();//1.col
        newItem->ImageIndex = -1;
        newItem->SubItems->Add("");
        newItem->SubItemImages[0] = -1;

        newItem->SubItems->Add(IntToStr(count+1));
        newItem->SubItemImages[1] = -1;

        newItem->SubItems->Add(MDefCycleTable[mID].address);   //cycle def
        newItem->SubItemImages[2] = -1;

        newItem->SubItems->Add(MDefCycleTable[mID].data);//addr
        newItem->SubItemImages[3] = -1;

        newItem->SubItems->Add(MDefCycleTable[mID].r_w);//data
        newItem->SubItemImages[4] = -1;

        newItem->SubItems->Add("");//r/w
        newItem->SubItemImages[5] = -1;
    }
    focusedCycle = 0;
    HighlightNextCycle(0);
```

```
}
//---------------------------------------------------------------------------
void HighlightNextCycle(UINT mID)
{
    Form1->CycleView->Items->Item[focusedCycle]->SubItemImages[0]=-1;
    if (!InCycle)//end of all cycles
        mID = 0;
    Form1->CycleView->Items->Item[mID]->SubItemImages[0]=1;

    focusedCycle = mID;
}
//---------------------------------------------------------------------------
void displayOperation(TLabel *dest, UINT value, UINT andValue, UINT lastaddr)
{
    dest->Caption = IntToStr((bool)(value&andValue));
    if ((lastaddr&andValue)!=(value&andValue)){
        dest->Color = clRed;
        dest->Font->Color = clWhite;
    }
    else{
        dest->Color=clBtnFace;
        dest->Font->Color = clBlack;
    }

}
//---------------------------------------------------------------------------
void UpdateAddressDataBusView(void)
{
    static UINT lastaddr;
    static UINT lastdata;
    static UINT lastRW;
    UINT addrValue = Motorola.addrBus.value();
    UINT dataValue = Motorola.dataBus;
    UINT RWValue = Motorola.R_W;

    Form1->AddrDataGrid->Cells[1][0] = IntToHex((int)Motorola.addrBus.value(),4);
    Form1->AddrDataGrid->Cells[1][1] = IntToHex((int)Motorola.dataBus,2);
    Form1->AddrDataGrid->Cells[1][2] = IntToHex((int)Motorola.R_W,1);

    displayOperation(Form1->Label2,addrValue,0x8000,lastaddr);
    displayOperation(Form1->Label3,addrValue,0x4000,lastaddr);
    displayOperation(Form1->Label4,addrValue,0x2000,lastaddr);
    displayOperation(Form1->Label5,addrValue,0x1000,lastaddr);
    displayOperation(Form1->Label6,addrValue,0x0800,lastaddr);
    displayOperation(Form1->Label7,addrValue,0x0400,lastaddr);
    displayOperation(Form1->Label8,addrValue,0x0200,lastaddr);
    displayOperation(Form1->Label9,addrValue,0x0100,lastaddr);
    displayOperation(Form1->Label10,addrValue,0x0080,lastaddr);
    displayOperation(Form1->Label11,addrValue,0x0040,lastaddr);
    displayOperation(Form1->Label12,addrValue,0x0020,lastaddr);
    displayOperation(Form1->Label13,addrValue,0x0010,lastaddr);
    displayOperation(Form1->Label14,addrValue,0x0008,lastaddr);
    displayOperation(Form1->Label15,addrValue,0x0004,lastaddr);
    displayOperation(Form1->Label16,addrValue,0x0002,lastaddr);
    displayOperation(Form1->Label17,addrValue,0x0001,lastaddr);

    displayOperation(Form1->Label18,dataValue,0x0080,lastdata);
    displayOperation(Form1->Label19,dataValue,0x0040,lastdata);
    displayOperation(Form1->Label20,dataValue,0x0020,lastdata);
    displayOperation(Form1->Label21,dataValue,0x0010,lastdata);
    displayOperation(Form1->Label22,dataValue,0x0008,lastdata);
    displayOperation(Form1->Label23,dataValue,0x0004,lastdata);
    displayOperation(Form1->Label24,dataValue,0x0002,lastdata);
    displayOperation(Form1->Label25,dataValue,0x0001,lastdata);

    displayOperation(Form1->Label26,RWValue,0x0001,lastRW);
    lastaddr=addrValue;
    lastdata=dataValue;
    lastRW=RWValue;
}
//---------------------------------------------------------------------------
bool InterruptLookup(void)
{
    if (InitInterrupt==true){
        UCHAR *memPtr=&Motorola.itsMem[Motorola.SP.value()];
        *memPtr--=Motorola.PC.valueL();
        *memPtr--=Motorola.PC.valueH();
        *memPtr--=Motorola.IY.valueL();
        *memPtr--=Motorola.IY.valueH();
        *memPtr--=Motorola.IX.valueL();
        *memPtr--=Motorola.IX.valueH();
        *memPtr--=Motorola.ACCA;
        *memPtr--=Motorola.ACCB;
        *memPtr--=Motorola.CCR.value();
        Motorola.SP = Motorola.SP - 0x0009;
        Motorola.CCR.AssignBit(BIT_INTERRUPT, 1);
        Motorola.CCR.AssignBit(BIT_XINTERRUPT, 1);
        Motorola.PC = IntVectorAddress;
        InitInterrupt=false;
        InterruptRequest=true;
        return 1;
    }
    return 0;
}
//---------------------------------------------------------------------------
```

opParse2.cpp

```
#include <vcl.h>
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "typedef.h"
#include "asmLine.h"
#include "funcs.h"
#include "def.h"

//---------------------------------------------------------------------------
char *opParse(char *p, int *errorPtr, opDescriptor *opDesc, asmLine *curLine, UCHAR argId)
{
UINT param1,param2;
bool jmpCheck = false;
char *start;
          p = start = skipSpace(p);
          /* Check for branch instruction */
    if (curLine->itsOpCode->itsInstruction->branch==1){
        if (!strcmp(curLine->itsOpCode->itsInstruction->mnemonic,"BRCLR")){
            if (curLine->addrMode == ADDR_DIRECT && argId == 2)
                jmpCheck = true;
            else if (argId == 3)
                jmpCheck = true;
        }
        else
            jmpCheck = true;
        if (jmpCheck == true){
            strcpy (curLine->itsJmpLbl, p);
            ownTrimRight(curLine->itsJmpLbl);
            p = eval(p, &(opDesc->data), &(opDesc->backRef), errorPtr, &(opDesc->defined));
            /* If expression evaluates OK, then return */
            if (*errorPtr < ERROR){
                curLine->itsJmpAddr = opDesc->data;
            }
            return NULL;
        }
    }
          /* Check for immediate mode */
          if (p[0] == '#') {
                    p = eval(++p, &(opDesc->data), &(opDesc->backRef), errorPtr, &(opDesc->defined));
                    /* If expression evaluates OK, then return */
                    if (*errorPtr < SEVERE) {
                              if (isTerm(*p)) {
                                        curLine->addrMode = ADDR_IMMEDIATE;
                                        return p;}
                              else {
                                        NEWERROR(*errorPtr, SYNTAX);
                                        return NULL;}
                    }
                    else
                              return NULL;
    }
          if (p[0] == 'X'){
        if (isTerm(p[1])){
            curLine->addrMode = ADDR_INDEXEDX;
            return (p + 1);}
        else {
            NEWERROR(*errorPtr, SYNTAX);
            return NULL;}
        return NULL;
    }
          /* Check for indexed mode Y*/
          if (p[0] == 'Y') {
        if (isTerm(p[1])){
            curLine->addrMode = ADDR_INDEXEDY;
            return (p + 1);}
        else {
            NEWERROR(*errorPtr, SYNTAX);
            return NULL;}
        return NULL;
    }
          /* Check for direct or extended*/
          p = eval(p, &(opDesc->data), &(opDesc->backRef), errorPtr, &(opDesc->defined));
          if (*errorPtr < SEVERE) {
                    // Check for absolute
                    if (isTerm(p[0])) {
            if (!opDesc->backRef || opDesc->data >= 0)
                if (opDesc->data <0x100){//0-00ff
                    if (curLine->addrMode == ADDR_INHERENT)
                        curLine->addrMode = ADDR_DIRECT;
                }
                else{
                    if (curLine->addrMode == ADDR_INHERENT)
                        curLine->addrMode = ADDR_EXTENDED;
                }
                    return p;
        }
        else{
              NEWERROR(*errorPtr, SYNTAX);
              return NULL;}
          return NULL;
    }
          NEWERROR(*errorPtr, SYNTAX);
          return NULL;
```

```
}
```

Eval.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "typedef.h"
#include "asmLine.h"
#include "funcs.h"

#define INTLIMIT 0xFFFF
#define LONGLIMIT 0xFFFFFFFF
#define STACKMAX 5
#define MAXHASH 26

static symbolDef *htable[MAXHASH+1];

char      *eval(char *p, long *valuePtr, char *refPtr, int *errorPtr, char *defptr)
{
long      valStack[STACKMAX];
char      opStack[STACKMAX-1];
int       valPtr = 0;
int       opPtr = 0;
long      t;
int       prec;
long      i;
char      evaluate, backRef;
int       status;
UCHAR pass2 = 1;


evaluate = TRUE;// Assume that the expression is to be evaluated, at least until an undefined symbol is found
*refPtr = TRUE;// Assume initially that all symbols are backwards references
while (TRUE) {// Loop until terminator character is found (loop is exited via return)
    //EXPECT AN OPERAND                 *
                  status = OK;// Use evalNumber to read in a number or symbol */
                  p = evalNumber(p, &t, &backRef, &status, defptr);
                  NEWERROR(*errorPtr, status);
                  if (!backRef && (status > ERROR || status == INCOMPLETE)) {
           evaluate = FALSE;// Stop evaluating the expression */
           *refPtr = FALSE;
       }
                  else if (*errorPtr > SEVERE)
           return NULL;// Pass any other error to the caller */
                  else {// If OK or WARNING, push the value on the stack */
                           if (evaluate)
           valStack[valPtr++] = t;// Set *refPtr to reflect the symbol just parsed */
                           *refPtr = (char) (*refPtr && backRef);
       }

                   //                EXPECT AN OPERATOR
                  // Handle the >> and << operators
                  if (*p == '>' || *p == '<') {
                           p++;
                           if (*p != *(p-1)) {
           NEWERROR(*errorPtr, SYNTAX);
           return NULL;
                           }
       }
                  prec = precedence(*p);// Do all stacked operations that are of higher precedence than the
operator just examined. */
                  while (opPtr && evaluate && (prec <= precedence(opStack[opPtr-1]))) {// Pop operands and
operator and do the operation */
       t = valStack[--valPtr];
                  i = valStack[--valPtr];
                  status = doOp(i, t, opStack[--opPtr], &t);
                           if (status != OK) {// Report error from doOp */
                                    if (pass2) {
               NEWERROR(*errorPtr, status);
               }
                                    else
               NEWERROR(*errorPtr, INCOMPLETE);
           evaluate = FALSE;
           *refPtr = FALSE;
           }
                           else// Otherwise push result on the stack */
           valStack[valPtr++] = t;
       }
                  if (prec) {
                           if (evaluate)// If operator is valid, push it on the stack */
           opStack[opPtr++] = *p;
                           p++;
       }
                  else if (*p == ',' || *p == '(' || *p == ')' || !(*p) || isspace(*p)) {// If the character
terminates the expression, then return the various results needed. */
                           if (evaluate){
           *valuePtr = valStack[--valPtr];
           *defptr = 1;
           }
                           else
                           *valuePtr = 0;
       return p;
       }
                  else {// Otherwise report the syntax error */
       NEWERROR(*errorPtr,  SYNTAX);
                  return NULL;
                  }
```

```
                    }

          return NULL;//NORMAL;
}
//---------------------------------------------------------------------------
char      *evalNumber(char *p, long *numberPtr, char *refPtr, int *errorPtr, char *defptr)
{
int       status;
long      base;
long      x;
char      name[SIGCHARS+1];
symbolDef *symbol;
int       i;
char      endFlag;
UCHAR pass2 = 1;

          *refPtr = TRUE;
          if (*p == '-') {
                    // Evaluate unary minus operator recursively */
                    p = evalNumber(++p, &x, refPtr, errorPtr, defptr);
                    *numberPtr = -x;
                    return p;
                    }
          else if (*p == '~') {
                    // Evaluate one's complement operator recursively */
                    p = evalNumber(++p, &x, refPtr, errorPtr, defptr);
                    *numberPtr = ~x;
                    return p;
                    }
          else if (*p == '(') {
                    // Evaluate parenthesized expressions recursively */
                    p = eval(++p, &x, refPtr, errorPtr, defptr);
                    if (*errorPtr > SEVERE)
                              return NULL;
                    else if (*p != ')') {
                              NEWERROR(*errorPtr, SYNTAX);
                              return NULL;
                              }
                    else {
                              *numberPtr = x;
                              return ++p;
                              }
                    }//end *p == '('
          else if (*p == '$' && isxdigit(*(p+1))) {
                    // Convert hex digits until another character is found. (At least one hex digit is present.)
                    x = 0;
                    while (isxdigit(*++p)) {
                              if ((unsigned long)x > (unsigned long)LONGLIMIT/16)
                                        NEWERROR(*errorPtr, NUMBER_TOO_BIG);
                              if (*p > '9')
                                        x = 16 * x + (*p - 'A' + 10);
                              else
                                        x = 16 * x + (*p - '0');
                              }
                    *numberPtr = x;
                    return p;
                    }//end *p == '$' && isxdigit(*(p+1))) {
          else if (*p == '%' || *p == '@' || isdigit(*p)) {
// Convert digits in the appropriate base (binary, octal, or decimal) until an invalid digit is found. */
                    if (*p == '%') {
                              base = 2;
                              p++;
                              }
                    else if (*p == '@') {
                              base = 8;
                              p++;
                              }
                    else base = 10;
                    // Check that at least one digit is present */
                    if (*p < '0' || *p >= '0' + base) {
                              NEWERROR(*errorPtr, SYNTAX);
                              return NULL;
                              }
                    x = 0;
                    // Convert the digits into an integer */
                    while (*p >= '0' && *p < '0' + base) {
                              if (x > (LONGLIMIT - (*p - '0')) / base) {
                                        NEWERROR(*errorPtr, NUMBER_TOO_BIG);
               //Form1->Memo1->Lines->Add("number is too big\n");
                                        //printf ("number is too big\n");
                                        }
                              x = (long) ( (long) ((long) base * x) + (long) (*p - '0') );
                              p++;
                              }
                    *numberPtr = x;
                    return p;
                    }//end *p == '%' || *p == '@' || isdigit(*p)) {
          else if (*p == '\'') {
                    endFlag = FALSE;
                    i = 0;
                    x = 0;
                    p++;
                    while (!endFlag) {
                              if (*p == '\'')
                                        if (*(p+1) == '\'') {
                                                  x = (x << 8) + *p;
                                                  i++;
                                                  p++;
```

```
                                        }
                                else
                                        endFlag = TRUE;
                        else {
                                x = (x << 8) + *p;
                                i++;
                                }
                        p++;
                        }
                if (i == 0) {
                        NEWERROR(*errorPtr, SYNTAX);
                        return NULL;
                        }
                else if (i == 3)
                        x = x << 8;
                else if (i > 4)
                        NEWERROR(*errorPtr, ASCII_TOO_BIG);
                *numberPtr = x;
                return p;
                }//end *p == '\'') {
        else if (isalpha(*p) || *p == '.') {
                // Determine the value of a symbol */
                i = 0;
                // Collect characters of the symbol's name (only SIGCHARS characters are significant) */
                do {
                        if (i < SIGCHARS)
                                name[i++] = *p;
                        p++;
                } while (isalnum(*p) || *p == '.' || *p == '_' || *p == '$');
                name[i] = '\0';
                // Look up the name in the symbol table, resulting in a pointer to the symbol table entry */
                status = OK;
                symbol = lookup(name, FALSE, &status);
                if (status == OK)
                // If symbol was found, and it's not a register list symbol, then return its value */
                        if (!(symbol->flags & REG_LIST_SYM)) {
                                *numberPtr = symbol->value;
                                if (pass2)
                                        *refPtr = (symbol->flags & BACKREF);
                                }
                        else {
                                // If it is a register list symbol, return error */
                                *numberPtr = 0;
                                NEWERROR(*errorPtr, REG_LIST_SPEC);
                                }
                else {
                        // Otherwise return an error */
                        if (pass2) {
                                NEWERROR(*errorPtr, UNDEFINED);
                                }
                        else
                                NEWERROR(*errorPtr, INCOMPLETE);
                        *refPtr = FALSE;
                        }
                return p;
                }//end *p == '\'') {
        else {
                // Otherwise, the character was not a valid operand */
                NEWERROR(*errorPtr, SYNTAX);
                return NULL;
    }
        return NULL;//NORMAL;
}
//---------------------------------------------------------------------------
int precedence(char op)
{
        /* Compute the precedence of an operator. Higher numbers indicate
           higher precedence, e.g., precedence('*') > precedence('+').
           Any character which is not a binary operator will be assigned
           a precedence of zero. */
        switch (op) {
                case '+' :
                case '-' : return 1;
                case '&' :
                case '!' : return 3;
                case '>' :
                case '<' : return 4;
                case '*' :
                case '/' :
                case '\\': return 2;
                default  : return 0;
                }
        return 1;//NORMAL;
}
//---------------------------------------------------------------------------
int doOp(long val1, long val2, char op, long *result)
{
        /* Performs the operation of the operator on the two operands.
           Returns OK or DIV_BY_ZERO. */
        switch (op) {
                case '+' : *result = val1 + val2;  return OK;
                case '-' : *result = val1 - val2;  return OK;
                case '&' : *result = val1 & val2;  return OK;
                case '!' : *result = val1 | val2;  return OK;
                case '>' : *result = val1 >> val2; return OK;
                case '<' : *result = val1 << val2; return OK;
                case '*' : *result = val1 * val2;  return OK;
                case '/' : if (val2 != 0) {
```

```
                                        *result = val1 / val2;
                                        return OK;
                                        }
                                else
                                        return DIV_BY_ZERO;
                        case '\\': if (val2 != 0) {
                                        *result = val1 % val2;
                                        return OK;
                                        }
                                else
                                        return DIV_BY_ZERO;
                        default  : break;//ourcommand
                        }
                return 1;//NORMAL;
}
//---------------------------------------------------------------------------
symbolDef *lookup(char *sym, int create, int *errorPtr)
{
int h, cmp;
symbolDef *s, *last, *t;
static char initialized = FALSE;

                if (!initialized) {
                        for (h = 0; h <= MAXHASH; h++)
                                htable[h] = 0;
                        initialized = TRUE;
                        }

                h = hash(sym);
                if (s = htable[h]) {
                        /* Search the linked list for a matching symbol */
                        last = NULL;
                        while (s && (cmp = strcmp(s->name, sym)) < 0) {
                                last = s;
                                s = s->next;
                                }
                        /* If a match was found, return pointer to the structure */
                        if (s && !cmp) {
                                if (create)
                                        NEWERROR(*errorPtr, MULTIPLE_DEFS);
                                t = s;
                                }
                        else if (create)
                                if (last) {
                                        t = (symbolDef *) malloc(sizeof(symbolDef));
                                        last->next = t;
                                        t->next = s;
                                        strcpy(t->name, sym);
                                        }
                                else {
                                        t = (symbolDef *) malloc(sizeof(symbolDef));
                                        t->next = htable[h];
                                        htable[h] = t;
                                        strcpy(t->name, sym);
                                        }
                        else
                                NEWERROR(*errorPtr, UNDEFINED);
                        }
                else if (create) {
                        t = (symbolDef *) malloc(sizeof(symbolDef));
                        htable[h] = t;
                        t->next = NULL;
                        strcpy(t->name, sym);
                        }
                else
                        NEWERROR(*errorPtr, UNDEFINED);
                return t;
}
//---------------------------------------------------------------------------
int hash(char *symbol)
{
int sum;

                sum = 0;
                while (*symbol) {
                        sum += (isupper(*symbol)) ? (*symbol - 'A') : 26;
                        symbol++;
                        }
                return (sum % 27);
}
//---------------------------------------------------------------------------
symbolDef *define(char *sym, long value, int check, int *errorPtr)
{
symbolDef *symbol;

                symbol = lookup(sym, !check, errorPtr);
                if (*errorPtr < ERROR)
                        if (check) {
                                if (symbol->value != value)
                                        NEWERROR(*errorPtr, PHASE_ERROR);
                                symbol->flags |= BACKREF;
                                }
                        else {
                                symbol->value = value;
                                symbol->flags = 0;
                                }
                return symbol;
}
```

Execute.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "prj.h"
#include "linked.h"
#include "typedef.h"
#include "asmLine.h"

#include "micro.hpp"
#include "reg16bit.h"
#include "funcs.h"

#include "OpCode.h"
#include "instTbl.h"

extern PACKAGE TForm1 *Form1;
extern LinkedList<asmLine> LineList;
extern asmLine * SearchCurrentPCLine(LinkedList<asmLine> *LList, UINT currentPC);
extern Micro Motorola;
extern UCHAR MachineCycle;
extern bool InterruptRequest;
extern UINT SPVector;

//---------------------------------------------------------------------------
int instExec_ABA(asmLine *curLine)
{   //ACCA = (ACCA) + (ACCB)
    UINT id = curLine->id;
    UCHAR *op1, *op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;

        UINT data = *op1 + *op2;

        Motorola.CCR.Carry8bit(*op1, *op2, data);
        Motorola.CCR.OverFlow8bit(*op1, *op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.HalfCarry8bit(*op1, *op2, data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ABX(asmLine *curLine)
{   //IX = (IX) + (ACCB)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        Motorola.IX = Motorola.IX + Motorola.ACCB;
        return 0;
    }
    else
        return 1;
}
////////////////////////////////////
int instExec_ABY(asmLine *curLine)
{   //IY = (IY) + (ACCB)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        Motorola.IY = Motorola.IY + Motorola.ACCB;
        return 0;
    }
    else
        return 1;
}

////////////////////////////////////
int instExec_ADCA(asmLine *curLine)
{   // ACCA = (ACCA) + (M) + (C)
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
```

```
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        }
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];


        UINT data = *op1 + Motorola.CCR.GetBit(BIT_CARRY) + op2;

        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.HalfCarry8bit(*op1, op2, data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ADCB(asmLine *curLine)
{   //ACCB <- (ACCB) + (M) + (C)
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCB;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        }
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        UINT data = *op1 + Motorola.CCR.GetBit(BIT_CARRY) + op2;
        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.HalfCarry8bit(*op1, op2, data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ADDA(asmLine *curLine)
{   //ACCA <- (ACCA) + (M)
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];


        UINT data = *op1 + op2;
        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
```

```
        Motorola.CCR.HalfCarry8bit(*op1, op2, data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ADDB(asmLine *curLine)
{   //ACCB <- (ACCB) + (M)
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        UINT data = *op1 + op2;
        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.HalfCarry8bit(*op1, op2, data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ADDD(asmLine *curLine)
{   //ACCD <- (ACCD) + (M : M + 1)
    UINT id = curLine->id;
    UINT result;
    UINT *op1,op2;
    UINT ACCD = (Motorola.ACCA*0x100)+Motorola.ACCB;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &ACCD;
        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=curLine->itsOpCode->source->data;
        else if (curLine->addrMode == ADDR_DIRECT){
            result=Motorola.itsMem[curLine->itsOpCode->source->data]*256;
            result+=Motorola.itsMem[curLine->itsOpCode->source->data+1];
            op2=result;
        }
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data]*256+\
                        Motorola.itsMem[curLine->itsOpCode->source->data+1];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2=Motorola.itsMem[Motorola.IX+curLine->itsOpCode->source->data]*256+\
                        Motorola.itsMem[Motorola.IX+curLine->itsOpCode->source->data+1];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2=Motorola.itsMem[Motorola.IY+curLine->itsOpCode->source->data%256]*256+\
                        Motorola.itsMem[Motorola.IY+curLine->itsOpCode->source->data%256+1];


        UINT data = *op1 + op2;
        Motorola.CCR.Carry16bit(*op1, op2, data);
        Motorola.CCR.OverFlow16bit(*op1, op2, data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.Negative16bit(data);
        *op1 = data;
        Motorola.ACCA = ACCD/0x100;
        Motorola.ACCB = ACCD%0x100;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ANDA(asmLine *curLine)
{   //ACCA <- (ACCA) . (M)
    UINT id = curLine->id;
    UCHAR *op1,op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
```

```
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCA;

        UINT data = *op1 & op2;
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ANDB(asmLine *curLine)
{   //ACCB <- (ACCB) + (M)
    UINT id = curLine->id;
    UCHAR *op1, op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCB;

        UINT data = *op1 & op2;
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ASLA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;

        UINT data = *op1<<1;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ASLB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
```

```
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        UINT data = *op1<<1;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ASL(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();

        UINT data = *op1<<1;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));
        Motorola.result = data;
        *op1 = data;
        UpdateGridLocation(op1);

        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_ASLD(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        UINT ACCD = (Motorola.ACCA*0x100)+Motorola.ACCB;
        op1 = &ACCD;
        UINT data = *op1<<1;

        Motorola.CCR.Negative16bit(data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x8000));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
        Motorola.ACCA = ACCD/0x100;
        Motorola.ACCB = ACCD%0x100;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ASRA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
```

```
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;

        UCHAR data = *op1>>1;
        data = data + ((*op1)&0x80);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ASRB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1,op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        UCHAR data = *op1>>1;
        data = data + ((*op1)&0x80);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ASR(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();

        UINT data = *op1>>1;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));
        Motorola.result = data;
        *op1 = data;
        UpdateGridLocation(op1);
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_BCC(asmLine *curLine)
{   //PC <- (PC) + $0002 + Rel      if (C) = 0
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//first operation:determine relative address
```

```
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!Motorola.CCR.GetBit(BIT_CARRY)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_BCLR(asmLine *curLine)
{   //M = (M) • (PC + 2)
    //M = (M) • (PC + 3)    (for IND,Y address mode only)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    UCHAR *op1;
    UCHAR mask;

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){
        if (curLine->addrMode == ADDR_DIRECT){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
            mask = curLine->itsOpCode->dest->data;
        }
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
            mask = curLine->itsOpCode->mask->data;
        }
        else if (curLine->addrMode == ADDR_INDEXEDY){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();
            mask = curLine->itsOpCode->mask->data;
        }

        *op1 = *op1&mask;
        Motorola.result = *op1;
        UpdateGridLocation(op1);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.Zero8bit(*op1);
        Motorola.CCR.Negative8bit(*op1);
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_BCS(asmLine *curLine)
{   //PC = (PC)+ $0002 + Rel        if (C) = 1
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address determination
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_CARRY)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
```

```
        return 0;
    }
    else
        return 1;
}
////////////////////////////////
int instExec_BEQ(asmLine *curLine)
{   //PC = (PC)+ $0002 + Rel        if (Z) = 1
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_ZERO)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}



////////////////////////////////

int instExec_BGE(asmLine *curLine)
{   //PC <- (PC)+ $0002 + Rel       if (N) EXOR (V) = 0
    //i.e., if (ACCX)<=(M)      (two's-complement signed numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }


    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_NEGATIVE) ^ Motorola.CCR.GetBit(BIT_OVERFLOW))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}

////////////////////////////////

int instExec_BGT(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (Z) + [(N) EXOR (V)] = 0
    //i.e., if (ACCX)>(M)       (two's-complement signed numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
```

```
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_ZERO)|| ((Motorola.CCR.GetBit(BIT_NEGATIVE) ^
Motorola.CCR.GetBit(BIT_OVERFLOW))))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}

//////////////////////////////////

int instExec_BHI(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (C) + (Z) = 0
    //i.e., if (ACCX)>(M)       (unsigned binary numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_CARRY) || (Motorola.CCR.GetBit(BIT_ZERO)))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BHS(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (C) = 0
    //i.e., if (ACCX)>=(M)      (unsigned binary numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_CARRY))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
```

```
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BITA(asmLine *curLine)
{   //(ACCA) . (M)
    UINT id = curLine->id;
    UCHAR *op1,op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2 = (curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCA;

        UINT data = *op1 & op2;
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BITB(asmLine *curLine)
{   //(ACCB) + (M)
    UINT id = curLine->id;
    UCHAR *op1, op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCB;

        UINT data = *op1 & op2;
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BLE(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (Z) + [(N) EXOR (V)] = 1
    //i.e., if (ACCX)<=(M)       (two's-complement signed numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
```

```
        if (Motorola.CCR.GetBit(BIT_ZERO)|| ((Motorola.CCR.GetBit(BIT_NEGATIVE) ^
Motorola.CCR.GetBit(BIT_OVERFLOW)))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//-------------------------------------------------------------------------
int instExec_BLO(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (C) = 1
    //i.e., if (ACCX)<(M)       (unsigned binary numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_CARRY)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}

////////////////////////////////

int instExec_BLS(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      (C) + (Z) = 1
    //i.e., if (ACCX)<=(M)      (unsigned binary numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_CARRY) || Motorola.CCR.GetBit(BIT_ZERO)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//-------------------------------------------------------------------------
int instExec_BLT(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (N) EXOR (V)= 1
    //i.e., if (ACCX)<(M)       (two's complement signed numbers)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
```

```
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_NEGATIVE) ^ Motorola.CCR.GetBit(BIT_OVERFLOW)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//-------------------------------------------------------------------------------
int instExec_BMI(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (N) = 1
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_NEGATIVE)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//-------------------------------------------------------------------------------
int instExec_BNE(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (Z) = 0
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }


    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_ZERO))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
```

```
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_BPL(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (N) = 0
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        //jmpLine = SearchLabel(curLine->itsJmpLbl);

        if (!(Motorola.CCR.GetBit(BIT_NEGATIVE))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_BRA(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_BRCLR(asmLine *curLine)
{   //PC <- (PC)+ $0004 + Rel   if (M) • (PC + 2) = 0
    //PC <- (PC)+ $0005 + Rel   if (M) • (PC + 3) = 0 (for IND,Y address
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    UCHAR *op1;
    UCHAR mask;
    UCHAR result;
    UCHAR offset;
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
```

```
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        if (curLine->addrMode == ADDR_DIRECT){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
            mask = curLine->itsOpCode->dest->data;
            offset = 0x0004;
        }
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
            mask = curLine->itsOpCode->mask->data;
            offset = 0x0005;
        }
        else if (curLine->addrMode == ADDR_INDEXEDY){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();
            mask = curLine->itsOpCode->mask->data;
            offset = 0x0005;
        }

        result = *op1&mask;
        if (!result)
            Motorola.PC = Motorola.PC + offset + destaddr;
        else
            Motorola.PC = Motorola.PC + offset;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BRN(asmLine *curLine)
{   //PC=(PC)+ $0002
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

                Motorola.PC = Motorola.PC + 2;
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}

////////////////////////////////////
int instExec_BRSET(asmLine *curLine)
{   //M = (M) • (PC + 2)
    //M = (M) • (PC + 3)    (for IND,Y address mode only)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    UCHAR *op1;
    UCHAR mask;
    UCHAR result;
    UCHAR offset;
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        if (curLine->addrMode == ADDR_DIRECT){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
            mask = curLine->itsOpCode->dest->data;
            offset = 0x0004;
        }
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
            mask = curLine->itsOpCode->mask->data;
            offset = 0x0005;
        }
        else if (curLine->addrMode == ADDR_INDEXEDY){
```

```
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();
            mask = curLine->itsOpCode->mask->data;
            offset = 0x0005;
        }

        result = ~(*op1)&mask;
        if (!result)
            Motorola.PC = Motorola.PC + offset + destaddr;
        else
            Motorola.PC = Motorola.PC + offset;
        return 0;
    }
    else
        return 1;
}
////////////////////////////////////
int instExec_BSET(asmLine *curLine)
{   //M = (M) • (PC + 2)
    //M = (M) • (PC + 3)    (for IND,Y address mode only)
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    UCHAR *op1;
    UCHAR mask;

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){
        if (curLine->addrMode == ADDR_DIRECT){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data;
            mask = curLine->itsOpCode->dest->data;
        }
        else if (curLine->addrMode == ADDR_INDEXEDX){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IX.value();
            mask = curLine->itsOpCode->mask->data;
        }
        else if (curLine->addrMode == ADDR_INDEXEDY){
            op1 = Motorola.itsMem+curLine->itsOpCode->source->data + Motorola.IY.value();
            mask = curLine->itsOpCode->mask->data;
        }

        *op1 = *op1|mask;
        Motorola.result = *op1;
        UpdateGridLocation(op1);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.Zero8bit(*op1);
        Motorola.CCR.Negative8bit(*op1);
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//-----------------------------------------------------------------------------
extern int instExec_BSR(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr){
            destaddr = curLine->itsJmpAddr;
            jmpLine = SearchCurrentPCLine(&LineList, destaddr);
        }
        else
            ;
        Motorola.Nxtop = jmpLine->itsInstruction->hexcode[jmpLine->addrMode];
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        if (curLine->itsJmpAddr)
            Motorola.PC = Motorola.PC + destaddr + 2;
        else
            Motorola.PC = destaddr;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_BVC(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (V) = 0
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
```

```
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (!(Motorola.CCR.GetBit(BIT_OVERFLOW))){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_BVS(asmLine *curLine)
{   //PC=(PC)+ $0002 + Rel      if (V) = 1
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }


    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (Motorola.CCR.GetBit(BIT_OVERFLOW)){
            if (curLine->itsJmpAddr)
                Motorola.PC = Motorola.PC + destaddr + 2;
            else
                Motorola.PC = destaddr;
        }
        else
            ;//PC is incremented by 2 automatically by updatePC function PC = PC+2;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_CBA(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        UCHAR *op1, *op2;
        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;

        UINT data = *op1-*op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.OverFlow8bit(*op2,data,*op1);
        Motorola.CCR.Carry8bit(*op2,data,*op1);

        return 0;//[if (instTable[id].cycle......]'s return value
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_CLC(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
```

```
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_CARRY, 0);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_CLI(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_INTERRUPT, 0);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_CLRA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        *op1 = 0;

        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);
        Motorola.CCR.AssignBit(BIT_ZERO,1);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_CLRB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        *op1 = 0;

        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);
        Motorola.CCR.AssignBit(BIT_ZERO,1);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_CLR(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());
```

```
        *op1 = 0;
        UpdateGridLocation(op1);

        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);
        Motorola.CCR.AssignBit(BIT_ZERO,1);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_CLV(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_CMPA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCA;
        UCHAR data = *op1 - op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.OverFlow8bit(op2, data, *op1);
        Motorola.CCR.Carry8bit(op2, data, *op1);

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_CMPB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCB;
        UCHAR data = *op1 - op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.OverFlow8bit(op2, data, *op1);
        Motorola.CCR.Carry8bit(op2, data, *op1);
```

```
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_COMA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        data = (~(*op1)) + (0xff) - (*op1);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.AssignBit(BIT_CARRY,1);

        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_COMB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        data = (~(*op1)) + (0xff) - (*op1);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.AssignBit(BIT_CARRY,1);

        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_COM(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = (~(*op1)) + (0xff) - (*op1);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.AssignBit(BIT_CARRY,1);
        Motorola.result = data;
        *op1=data;
        UpdateGridLocation(op1);

        return 1;
    }
    else{
```

```
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//------------------------------------------------------------------------
extern int instExec_CPD(asmLine *curLine)
{
    UINT ACCD=(Motorola.ACCA*0x100)+(Motorola.ACCB);
    UCHAR *op2;
    UINT *op1, data, result;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &ACCD;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }

        result = *op1 - data;

        Motorola.CCR.Negative16bit(result);
        Motorola.CCR.Zero16bit(result);
        Motorola.CCR.OverFlow16bit(*op1,data,result);
        Motorola.CCR.Carry16bit(data, result, *op1);
        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------
extern int instExec_CPX(asmLine *curLine)
{
    UCHAR *op2;
    Reg16Bit *op1;
    UINT data, result;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IX;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }

        result = op1->value() - data;

        Motorola.CCR.Negative16bit(result);
        Motorola.CCR.Zero16bit(result);
        Motorola.CCR.OverFlow16bit(op1->value(),data,result);


        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------
extern int instExec_CPY(asmLine *curLine)
{
    UCHAR *op2;
    Reg16Bit *op1;
    UINT data, result;
```

```
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IY;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }

        result = op1->value() - data;

        Motorola.CCR.Negative16bit(result);
        Motorola.CCR.Zero16bit(result);
        Motorola.CCR.OverFlow16bit(op1->value(),data,result);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
extern int instExec_DAA(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        UCHAR accH, accL;
        accH = Motorola.ACCA/16;
        accL = Motorola.ACCA%16;
        if (!Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<10 && accL<10)
            ;
        else if (!Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<9 && accL>9){
            Motorola.ACCA+=6;
            Motorola.CCR.AssignBit(BIT_CARRY, 0);
        }
        else if (!Motorola.CCR.GetBit(BIT_CARRY) && Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<10 && accL<4){
            Motorola.ACCA+=6;
            Motorola.CCR.AssignBit(BIT_CARRY, 0);
        }
        else if (!Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH>9 && accL<10){
            Motorola.ACCA+=0x60;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }
        else if (!Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH>8 && accL>9){
            Motorola.ACCA+=0x66;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }
        else if (!Motorola.CCR.GetBit(BIT_CARRY) && Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH>9 && accL<4){
            Motorola.ACCA+=0x66;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }
        else if (Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<3 && accL<10){
            Motorola.ACCA+=0x60;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }
        else if (Motorola.CCR.GetBit(BIT_CARRY) && !Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<3 && accL>9){
            Motorola.ACCA+=0x66;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }
        else if (Motorola.CCR.GetBit(BIT_CARRY) && Motorola.CCR.GetBit(BIT_HALFCARRY) && \
            accH<4 && accL<4){
            Motorola.ACCA+=0x66;
            Motorola.CCR.AssignBit(BIT_CARRY, 1);
        }

        Motorola.CCR.Negative8bit(Motorola.ACCA);
        Motorola.CCR.Zero8bit(Motorola.ACCA);
        return 0;
    }
```

```
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_DECA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        data = *op1 - (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                );

        *op1=data;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_DECB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        data = *op1 - (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                    );
       *op1=data;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_DEC(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UCHAR data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = *op1 - (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                    );
        *op1 = data;
        Motorola.result = data;
        UpdateGridLocation(op1);

        return 1;
```

```
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//-----------------------------------------------------------------------------
int instExec_DES(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.SP;
        *op1 = *op1 - (0x0001);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_DEX(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IX;
        *op1 = *op1- (0x0001);
        data = op1->value();

        Motorola.CCR.Zero16bit(data);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_DEY(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IY;
        *op1 = *op1- (0x0001);
        data = op1->value();

        Motorola.CCR.Zero16bit(data);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_EORA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
```

```
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCA;
        data = *op1 ^ op2;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_EORB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        op1 = &Motorola.ACCB;
        data = *op1 ^ op2;
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_FDIV(asmLine *curLine)//(ACCD)/(IX); IX <- Quotient, ACCD <- Remainder
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID;
    if (MachineCycle<3)
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    else
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[2];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        UINT ACCD = (Motorola.ACCA*0x100)+Motorola.ACCB;
        UINT numerator = ACCD;
        UINT divisor = Motorola.IX.value();
        UINT result;
        UINT remainder;
        bool tempbit;

        if (divisor){
            result = ACCD/divisor;
            remainder = ACCD%divisor;
        }

        Motorola.IX = result;
        ACCD = remainder;
        Motorola.ACCA = ACCD/0x100;
        Motorola.ACCB = ACCD%0x100;
        Motorola.CCR.Zero16bit(result);
        tempbit = bool(numerator<=divisor);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, ~tempbit);
        tempbit = bool(divisor);
        tempbit = !tempbit;
        Motorola.CCR.AssignBit(BIT_CARRY, tempbit);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_IDIV(asmLine *curLine)
{
```

```
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID;
    if (MachineCycle<3)
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    else
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[2];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        UINT ACCD = (Motorola.ACCA*0x100)+Motorola.ACCB;
        UINT numerator = ACCD;
        UINT divisor = Motorola.IX.value();
        UINT result;
        UINT remainder;
        bool tempbit;

        if (divisor){
            result = ACCD/divisor;
            remainder = ACCD%divisor;
        }

        Motorola.IX = result;
        ACCD = remainder;
        Motorola.ACCA = ACCD/0x100;
        Motorola.ACCB = ACCD%0x100;
        Motorola.CCR.Zero16bit(result);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        tempbit = bool(divisor);
        tempbit = !tempbit;
        Motorola.CCR.AssignBit(BIT_CARRY, tempbit);

        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------------
int instExec_INCA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        data = *op1 + (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                    );
        *op1=data;
        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------------
int instExec_INCB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        data = *op1 + (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                    );
        *op1=data;
        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------------
int instExec_INC(asmLine *curLine)
```

```
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = *op1 + (0x01);

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, \
            ~bool(data&0x80)&bool(data&0x40)&bool(data&0x20)&bool(data&0x10) \
            &bool(data&0x08)&bool(data&0x04)&bool(data&0x02)&bool(data&0x01)\
                    );
        *op1 = data;
        Motorola.result = data;
        UpdateGridLocation(op1);

        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_INS(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    op1 = &Motorola.SP;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        *op1 = *op1 + (0x0001);
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_INX(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IX;
        *op1 = *op1 + (0x0001);
        data = op1->value();

        Motorola.CCR.Zero16bit(data);
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_INY(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IY;
        *op1 = *op1 + (0x0001);
```

```
        data = op1->value();

        Motorola.CCR.Zero16bit(data);
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
extern int instExec_JMP(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT op2;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.PC;
        if (curLine->addrMode == ADDR_EXTENDED)
            op2 = (curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = (curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = (curLine->itsOpCode->source->data + Motorola.IY.value());

        *op1 = op2;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
extern int instExec_JSR(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    asmLine * jmpLine;
    static int destaddr;

    if (!MachineCycle){//relative address
        jmpLine = SearchLabel(curLine->itsJmpLbl);
        if (curLine->itsJmpAddr)
            destaddr = curLine->itsJmpAddr;
        else
            destaddr = jmpLine->itsMemLoc;
        Motorola.rr = destaddr;
    }

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle>3){
        if (curLine->addrMode == ADDR_DIRECT)
            Motorola.PC = Motorola.PC + 0x0002;
        else if (curLine->addrMode == ADDR_EXTENDED)
            Motorola.PC = Motorola.PC + 0x0003;
        else if (curLine->addrMode == ADDR_INDEXEDX)
            Motorola.PC = Motorola.PC + 0x0002;
        else if (curLine->addrMode == ADDR_INDEXEDY)
            Motorola.PC = Motorola.PC + 0x0003;
    }

    if (instTable[id].cycle[mode] == MachineCycle){
        if (curLine->itsJmpAddr)
            Motorola.PC = Motorola.PC + destaddr + 2;
        else
            Motorola.PC = destaddr;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_LDAA(asmLine *curLine)
{   //ACCX <- (M)
    UINT id = curLine->id;
    UCHAR *op1,op2;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
```

```
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);

        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LDAB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);

        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LDD(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op2;
    UINT *op1;
    UINT data;
    UINT ACCD;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &ACCD;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
```

```
        *op1 = data;
        Motorola.ACCA = ACCD/0x100;
        Motorola.ACCB = ACCD%0x100;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
extern int instExec_LDS(asmLine *curLine)
{
    UCHAR *op2;
    Reg16Bit *op1;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.SP;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }


        Motorola.CCR.Negative16bit(data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;
        SPVector=data;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
extern int instExec_LDX(asmLine *curLine)
{
    UCHAR *op2;
    Reg16Bit *op1;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1=&Motorola.IX;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2=&Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2=&Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2=&Motorola.itsMem[Motorola.IX+curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2=&Motorola.itsMem[Motorola.IY+curLine->itsOpCode->source->data%256];
            data = *(op2)*0x100 + *(op2+1);
        }

        Motorola.CCR.Negative16bit(data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
extern int instExec_LDY(asmLine *curLine)
{
    UCHAR *op2;
```

```
    Reg16Bit *op1;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1=&Motorola.IY;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2=&Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2=&Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2=&Motorola.itsMem[Motorola.IX+curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2=&Motorola.itsMem[Motorola.IY+curLine->itsOpCode->source->data%256];
            data = *(op2)*0x100 + *(op2+1);
        }

        Motorola.CCR.Negative16bit(data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSLA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        data = *op1<<1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSLB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        data = *op1<<1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSL(asmLine *curLine)
```

```
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){
        op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        data = *op1<<1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x80));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
        Motorola.result = data;
        UpdateGridLocation(op1);

        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }

}
//-----------------------------------------------------------------------------
int instExec_LSLD(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT ACCD;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        ACCD = (Motorola.ACCA*0x100)+Motorola.ACCB;
        data = ACCD<<1;

        Motorola.CCR.Negative16bit(data);
        Motorola.CCR.Zero16bit(data);
        Motorola.CCR.AssignBit(BIT_CARRY, bool(ACCD&0x8000));
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        Motorola.ACCA = data/0x100;
        Motorola.ACCB = data%0x100;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSRA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;

        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x01));
        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);

        data = *op1>>1;

        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
```

```
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSRB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x01));
        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);

        data = *op1>>1;

        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_LSR(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op1&0x01));
        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);

        data = *op1>>1;

        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,\
        Motorola.CCR.GetBit(BIT_NEGATIVE)^Motorola.CCR.GetBit(BIT_CARRY));

        *op1 = data;
        Motorola.result = data;
        UpdateGridLocation(op1);

        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//-----------------------------------------------------------------------------
int instExec_LSRD(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, *op2;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;
        data = (Motorola.ACCA*0x100) + Motorola.ACCB;
        data =data>>1;

        Motorola.CCR.AssignBit(BIT_CARRY, bool(*op2&0x01));
        Motorola.CCR.AssignBit(BIT_NEGATIVE, 0);
```

```
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, bool(data&0x01));

        Motorola.ACCA = data/0x100;
        Motorola.ACCB = data%0x100;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_MUL(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1,*op2;
    UINT data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID;

    if (MachineCycle<3)
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    else
        mID = CycleTable[instTable[id].cyclePtr[mode]].val[2];

    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;
        data = (*op1) * (*op2);

        *op1 = data/256;
        *op2 = data%256;

        Motorola.CCR.AssignBit(BIT_CARRY, Motorola.ACCB&0x80);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_NEGA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UCHAR data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        *op1 = 0 - *op1;
        data= *op1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.Carry8Negate(data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_NEGB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1;
    UCHAR data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        *op1 = 0 - *op1;
        data= *op1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.Carry8Negate(data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
```

```
//---------------------------------------------------------------------------
int instExec_NEG(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);;
    UCHAR data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op1 = 0 - *op1;
        data= *op1;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        Motorola.CCR.Carry8Negate(data);
        *op1 = data;
        Motorola.result = data;
        UpdateGridLocation(op1);
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_NOP(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ORAA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCA;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        UCHAR data;
        data= *op1 | op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ORAB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
```

```
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        UCHAR data;
        data= *op1 | op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW, 0);

        *op1 = data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_PSHA(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UCHAR *op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.SP;
        *op1 = *op1 - (0x0001);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_PSHB(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_PSHX(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_PSHY(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
```

```
}
//-----------------------------------------------------------------------------
int instExec_PULA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_PULB(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
////////////////////////////////////
int instExec_PULX(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
////////////////////////////////////
int instExec_PULY(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ROLA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = &Motorola.ACCA;
    UCHAR destreg, Cbit;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x80);
        destreg = destreg<<1;
        destreg += Cbit;
        *op1 = destreg;

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ROLB(asmLine *curLine)
```

```
{
    UINT id = curLine->id;
    UCHAR *op1 = &Motorola.ACCB;
    UCHAR destreg, Cbit;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x80);
        destreg = destreg<<1;
        destreg += Cbit;
        *op1 = destreg;

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
    return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_ROL(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
    UCHAR destreg, Cbit;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (MachineCycle == (instTable[id].cycle[mode]-1)){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x80);
        destreg = destreg<<1;
        destreg += Cbit;
        *op1 = destreg;
        UpdateGridLocation(op1);

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
        Motorola.result = destreg;
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//-----------------------------------------------------------------------------
int instExec_RORA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = &Motorola.ACCA;
    UCHAR destreg, Cbit;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);
        Cbit = Cbit<<7;

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x01);
        destreg = destreg>>1;
        destreg += Cbit;
        *op1 = destreg;

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
    return 0;
    }
```

```
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_RORB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = &Motorola.ACCB;
    UCHAR destreg, Cbit;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);
        Cbit = Cbit<<7;

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x01);
        destreg = destreg>>1;
        destreg += Cbit;
        *op1 = destreg;

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_ROR(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
    UCHAR destreg, Cbit;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        destreg = *op1;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);
        Cbit = Cbit<<7;

        Motorola.CCR.AssignBit(BIT_CARRY,destreg&0x01);
        destreg = destreg>>1;
        destreg += Cbit;
        *op1 = destreg;
        UpdateGridLocation(op1);

        Motorola.CCR.Negative8bit(destreg);
        Motorola.CCR.Zero8bit(destreg);
        Motorola.CCR.OverFlow8bitExor(Motorola.CCR.GetBit(BIT_NEGATIVE),\
                                      Motorola.CCR.GetBit(BIT_CARRY));
        Motorola.result = destreg;
        return 1;
    }
    else{
        if (instTable[id].cycle[mode] == MachineCycle)
            return 0;
        return 1;
    }
}
//---------------------------------------------------------------------------
int instExec_RTI(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Form1->Button1->Enabled = true;
        InterruptRequest=false;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_RTS(asmLine *curLine)
{
```

```
    UINT id = curLine->id;
    Reg16Bit *op1,*op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SBA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, *op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;

        UINT data = *op1 - *op2;

        Motorola.CCR.Carry8bit(*op1, *op2, data);
        Motorola.CCR.OverFlow8bit(*op1, *op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SBCA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2 , Cbit;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2 = (curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        data = *op1 - (op2 + Cbit);
        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SBCB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2 , Cbit;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;
        Cbit = Motorola.CCR.GetBit(BIT_CARRY);

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2 = (curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
```

```
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = *(Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        data = *op1 - (op2 + Cbit);
        Motorola.CCR.Carry8bit(*op1, op2, data);
        Motorola.CCR.OverFlow8bit(*op1, op2, data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Negative8bit(data);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SEC(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_CARRY,1);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SEI(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_INTERRUPT,1);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SEV(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_OVERFLOW,1);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_STAA(asmLine *curLine)
{   //(M) <- (ACCX)
    UCHAR *op1, *op2;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;

        if (curLine->addrMode == ADDR_DIRECT)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op2 = *op1;
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.Zero8bit(*op1);
        Motorola.CCR.Negative8bit(*op1);
        return 0;
    }
```

```
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_STAB(asmLine *curLine)
{
    UCHAR *op1, *op2;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCB;

        if (curLine->addrMode == ADDR_DIRECT)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op2 = *op1;
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.Zero8bit(*op1);
        Motorola.CCR.Negative8bit(*op1);

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_STD(asmLine *curLine)
{
    UCHAR *op1, *op2, *op3;
    UINT ACCD;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        op2 = &Motorola.ACCB;
        ACCD = ((*op1)*0x100) + *op2;

        if (curLine->addrMode == ADDR_DIRECT)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op3 = *op1;
        *(op3+1) = *op2;
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.Zero16bit(ACCD);
        Motorola.CCR.Negative16bit(ACCD);

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_STOP(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        if (Motorola.CCR.GetBit(BIT_STOP))
            return 0;//nothing to do NOP
        else
            return 0xff;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_STS(asmLine *curLine)
{
    UCHAR *op1, *op2, *op3;
    Reg16Bit SP;
```

```
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        SP = Motorola.SP;

        if (curLine->addrMode == ADDR_DIRECT)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op3 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op3 = Motorola.SP.valueH();
        *(op3+1) = Motorola.SP.valueL();

        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.Zero16bit(SP.value());
        Motorola.CCR.Negative16bit(SP.value());

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_STX(asmLine *curLine)
{
    UCHAR *op1;
    Reg16Bit IX;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        IX = Motorola.IX;

        if (curLine->addrMode == ADDR_DIRECT)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op1 = Motorola.IX.valueH();
        *(op1+1) = Motorola.IX.valueL();

        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.Zero16bit(IX.value());
        Motorola.CCR.Negative16bit(IX.value());

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_STY(asmLine *curLine)
{
    UCHAR *op1;
    Reg16Bit IY;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        IY = Motorola.IY;

        if (curLine->addrMode == ADDR_DIRECT)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_EXTENDED)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IX.value());
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op1 = (Motorola.itsMem + curLine->itsOpCode->source->data + Motorola.IY.value());

        *op1 = Motorola.IY.valueH();
        *(op1+1) = Motorola.IY.valueL();

        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
```

```
        Motorola.CCR.Zero16bit(IY.value());
        Motorola.CCR.Negative16bit(IY.value());

        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_SUBA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1,op2,data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCA;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = *op1 - op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Carry8bit(data, op2, *op1);
        Motorola.CCR.OverFlow8bit(data, op2, *op1);

        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_SUBB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1,op2,data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCB;

        if (curLine->addrMode == ADDR_IMMEDIATE)
            op2=(curLine->itsOpCode->source->data);
        else if (curLine->addrMode == ADDR_DIRECT)
            op2=Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = *op1 - op2;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.Carry8bit(data, op2, *op1);
        Motorola.CCR.OverFlow8bit(data, op2, *op1);

        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//---------------------------------------------------------------------------
int instExec_SUBD(asmLine *curLine)
{
    UINT ACCD=((Motorola.ACCA)*0x100 + Motorola.ACCB);
    UINT *op1, data, result;
    UCHAR *acca, *accb, *op2;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
```

```
        acca = &Motorola.ACCA;
        accb = &Motorola.ACCB;
        op1 = &ACCD;

        if (curLine->addrMode == ADDR_IMMEDIATE){
            data = curLine->itsOpCode->source->data;
        }
        else{
            if (curLine->addrMode == ADDR_DIRECT)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_EXTENDED)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data];
            else if (curLine->addrMode == ADDR_INDEXEDX)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
            else if (curLine->addrMode == ADDR_INDEXEDY)
                op2 = &Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];
            data = *(op2)*0x100 + *(op2+1);
        }

        result = *op1 - data;

        Motorola.CCR.Negative16bit(result);
        Motorola.CCR.Zero16bit(result);
        Motorola.CCR.OverFlow16bit(result,data,*op1);
        Motorola.CCR.Carry16bit(result, data, *op1);

        *acca = result/256;
        *accb = result%256;


        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_SWI(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    if (!MachineCycle)
        Motorola.PC = Motorola.PC + 0x0001;
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        Motorola.CCR.AssignBit(BIT_INTERRUPT,1);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TAB(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, *op2, data;

    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1=&Motorola.ACCA;
        op2=&Motorola.ACCB;
        data=*op1;
        *op2=data;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TAP(asmLine *curLine)
{
    UCHAR *op1;
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.ACCA;
        Motorola.CCR.AssignData(*op1);
        return 0;
    }
    else
        return 1;
}
```

```
//-----------------------------------------------------------------------------
int instExec_TBA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, *op2;
    UINT data;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1=&Motorola.ACCA;
        op2=&Motorola.ACCB;
        data=*op2;
        *op1=data;

        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(data);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TEST(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        if (Motorola.CCR.GetBit(BIT_STOP))
            return 0;//nothing to do NOP
        else
            return 0xfe;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TPA(asmLine *curLine)
{
    UCHAR op1,*op2;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op2 = &Motorola.ACCA;
        op1 = Motorola.CCR.value();
        *op2=op1;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TSTA(asmLine *curLine)
{
    UINT id = curLine->id;
    UCHAR *op1, op2;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCA;
        op2 = *op1 - (0x00);

        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);
        Motorola.CCR.Negative8bit(op2);
        Motorola.CCR.Zero8bit(*op1);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TSTB(asmLine *curLine)
{
    UCHAR *op1, op2;
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;
```

```
    if (instTable[id].cycle[mode] == MachineCycle){
        op1 = &Motorola.ACCB;
        op2 = *op1 - (0x00);

        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);
        Motorola.CCR.Negative8bit(op2);
        Motorola.CCR.Zero8bit(*op1);

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TST(asmLine *curLine)
{
    UCHAR op2, data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        if (curLine->addrMode == ADDR_EXTENDED)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data];
        else if (curLine->addrMode == ADDR_INDEXEDX)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IX.value()];
        else if (curLine->addrMode == ADDR_INDEXEDY)
            op2 = Motorola.itsMem[curLine->itsOpCode->source->data + Motorola.IY.value()];

        data = op2 - (0x00);
        Motorola.CCR.AssignBit(BIT_OVERFLOW,0);
        Motorola.CCR.AssignBit(BIT_CARRY,0);
        Motorola.CCR.Negative8bit(data);
        Motorola.CCR.Zero8bit(op2);
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TSX(asmLine *curLine)
{
    Reg16Bit *op1, *op2;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1=&Motorola.IX;
        op2=&Motorola.SP;

        data = *op2 + (0x0001);
        *op1=data;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TSY(asmLine *curLine)
{
    Reg16Bit *op1, *op2;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IY;
        op2 = &Motorola.SP;
        data = *op2 + (0x0001);
        *op1 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TXS(asmLine *curLine)
{
    Reg16Bit *op1, *op2;
    UINT data;
```

```
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IX;
        op2 = &Motorola.SP;
        data = op1->value() - (0x0001);
        *op2 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_TYS(asmLine *curLine)
{
    Reg16Bit *op1, *op2;
    UINT data;

    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IY;
        op2 = &Motorola.SP;
        data = op1->value() - (0x0001);
        *op2 = data;
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_WAI(asmLine *curLine)
{
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    if (!MachineCycle)
        Motorola.PC = Motorola.PC + 0x0001;
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_XGDX(asmLine *curLine)
{
    UINT id = curLine->id;
    Reg16Bit *op1;
    UINT op2, temp;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){

        op1 = &Motorola.IX;
        op2 = (Motorola.ACCA*0x100) + Motorola.ACCB;
        temp = op1->value();
        *op1 = op2;
        op2 = temp;

        Motorola.ACCA = op2/0x100;
        Motorola.ACCB = op2%0x100;

        return 0;
    }
    else
        return 1;
}
//-----------------------------------------------------------------------------
int instExec_XGDY(asmLine *curLine)
{
    UINT op2, temp;
    Reg16Bit *op1;
    UINT id = curLine->id;
    UINT mode = ConvertModeVal(curLine->addrMode);
    UINT mID = CycleTable[instTable[id].cyclePtr[mode]].val[MachineCycle];
    machineExecute(mID, id, mode);
    MachineCycle++;

    if (instTable[id].cycle[mode] == MachineCycle){
```

```
        op1 = &Motorola.IY;
        op2 = (Motorola.ACCA*0x100) + Motorola.ACCB;
        temp = op1->value();
        *op1 = op2;
        op2 = temp;

        Motorola.ACCA = op2/0x100;
        Motorola.ACCB = op2%0x100;

        return 0;
    }
    else
        return 1;
}
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
void machineExecute(UINT mId, UINT instId, UINT adrMode)//machine ID + ID
{
    Motorola.R_W = 1;
    switch (mId){
        case 1://00dd         00dd      1
            Motorola.addrBus = Motorola.dd;
            Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
            break;
        case 2://00dd         Result    0
            Motorola.addrBus = Motorola.dd;
            Motorola.dataBus = Motorola.result;
            Motorola.R_W = 0;
            break;
        case 3://00dd+1       00dd+1              1
            Motorola.addrBus = Motorola.dd+1;
            Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
            break;
        case 4://00dd         (A)       1
            Motorola.addrBus = Motorola.dd;
            Motorola.dataBus = Motorola.ACCA;
            break;
        case 5://00dd         (B)       1
            Motorola.addrBus = Motorola.dd;
            Motorola.dataBus = Motorola.ACCB;
            break;
        case 6://00dd+1       (A)       1
            Motorola.addrBus = Motorola.dd+1;
            Motorola.dataBus = Motorola.ACCA;
            break;
        case 7://00dd+1       (B)       1
            Motorola.addrBus = Motorola.dd+1;
            Motorola.dataBus = Motorola.ACCB;
            break;
        case 8://0xFFFF       -         1
            Motorola.addrBus = 0xFFFF;
            break;
        case 9://Reserved
            break;
        case 10://hhll        (B)       1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll;
            Motorola.dataBus = Motorola.ACCB;
            break;
        case 11://hhll        (A)       1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll;
            Motorola.dataBus = Motorola.ACCA;
            break;
        case 12://hhll        (00)      0
            Motorola.addrBus = Motorola.hh*256+Motorola.ll;
            Motorola.dataBus = 0;
            Motorola.R_W = 0;
            break;
        case 13://hhll        (hhll)    1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll;
            Motorola.dataBus = *(Motorola.itsMem+Motorola.addrBus.value());
            break;
        case 14://hhll        Result    0
            Motorola.addrBus = Motorola.hh*256+Motorola.ll;
            Motorola.dataBus = Motorola.result;
            Motorola.R_W = 0;
            break;
        case 15://hhll+1      (hhll+1)    1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
            Motorola.dataBus = *(Motorola.itsMem+Motorola.addrBus.value());
            break;
        case 16://hhll+1    (A)     1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
            Motorola.dataBus = Motorola.ACCA;
            break;
        case 17://hhll+1      (B)           1
            Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
            Motorola.dataBus = Motorola.ACCB;
            break;
        case 20://OP    Inst Code L 1
            Motorola.addrBus = Motorola.PC;
            Motorola.dataBus = instTable[instId].hexcode[adrMode]&0xff;
            break;
        case 21://OP    Inst Code H 1
            Motorola.addrBus = Motorola.PC;
            Motorola.dataBus = instTable[instId].hexcode[adrMode]>>8;
            break;
```

```
case 23://OP+1  -   1
    Motorola.addrBus = Motorola.PC + 1;
    break;
case 24://OP+1        dd (direct data)    1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.dd;
    break;
case 25://OP+1        ff        1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.ff;
    break;
case 26://OP+1        hh        1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.hh;
    break;
case 27://OP+1        Inst Code L         1
    Motorola.addrBus = Motorola.PC+1;
    Motorola.dataBus = instTable[instId].hexcode[adrMode]&0xff;
    break;
case 28://OP+1        ii (immediate data) 1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.ii;
    break;
case 29://OP+1        jj        1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.jj;
    break;
case 30://OP+1        rr        1
    Motorola.addrBus = Motorola.PC + 1;
    Motorola.dataBus = Motorola.rr;
    break;
case 34://OP+2        hh        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.hh;
    break;
case 35://OP+2        ff        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.ff;
    break;
case 36://OP+2        kk        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.kk;
    break;
case 37://OP+2        ll        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.ll;
    break;
case 38://OP+2        MM        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.MM;
    break;
case 39://OP+2        -         1
    Motorola.addrBus = Motorola.PC + 2;
    break;
case 40://OP+2        jj        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.jj;
    break;
case 41://OP+2        dd        1
    Motorola.addrBus = Motorola.PC + 2;
    Motorola.dataBus = Motorola.dd;
    break;
case 42://OP+3        kk        1
    Motorola.addrBus = Motorola.PC + 3;
    Motorola.dataBus = Motorola.kk;
    break;
case 43://OP+3        mm        1
    Motorola.addrBus = Motorola.PC + 3;
    Motorola.dataBus = Motorola.mm;
    break;
case 44://OP+3        rr        1
    Motorola.addrBus = Motorola.PC + 3;
    Motorola.dataBus = Motorola.rr;
    break;
case 45://OP+4        rr        1
    Motorola.addrBus = Motorola.PC + 4;
    Motorola.dataBus = Motorola.rr;
    break;
case 46://X+ff        Result    0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff;
    Motorola.dataBus = Motorola.result;
    Motorola.R_W = 0;
    break;
case 47://X+ff        X+ff      1
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    break;
case 48://X+ff+1      X+ff+1    1
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    break;
case 53:
    Motorola.R_W = 0;
    break;
case 54:
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    break;
```

```
case 55:
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    break;
case 60:
    Motorola.dataBus = Motorola.Nxtop;
    Motorola.addrBus = Motorola.rr;
    break;
case 61:
    Motorola.dataBus = Motorola.PC.valueL();
    Motorola.addrBus = Motorola.SP;
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 62:
    Motorola.dataBus = Motorola.PC.valueH();
    Motorola.addrBus = Motorola.SP;
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 63://SP    -   1
    Motorola.addrBus = Motorola.SP;
    break;
case 64://SP          (A)       0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.ACCA;
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 65://SP          (B)       0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus=Motorola.ACCB;
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 66://SP          (IXL)     0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus=Motorola.IX.valueL();
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 67://SP          (IXH)     0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus=Motorola.IX.valueH();
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 68://SP          (IYL)     0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus=Motorola.IY.valueL();
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 69://SP          (IYH)     0
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus=Motorola.IY.valueH();
    Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
    UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
    Motorola.SP = Motorola.SP - 0x0001;
    Motorola.R_W = 0;
    break;
case 70://SP+1  get A   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.ACCA = Motorola.dataBus;
    break;
case 71://SP+1  get B   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.ACCB = Motorola.dataBus;
    break;
case 72://SP+1        Get IXH   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IX.assignH(Motorola.dataBus);
    break;
case 73://SP+2        Get IXL   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IX.assignL(Motorola.dataBus);
```

```
    break;
case 74://SP+1        Get IYH   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IY.assignH(Motorola.dataBus);
    break;
case 75://SP+2        Get IYL   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IY.assignL(Motorola.dataBus);
    break;
case 76://SP+1        Get CC    1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.CCR.AssignData(Motorola.dataBus);
    break;
case 77://SP+2        Get B     1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.ACCB = Motorola.dataBus;
    break;
case 78://SP+3        Get A     1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.ACCA = Motorola.dataBus;
    break;
case 79://SP+4        Get IXH   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IX.assignH(Motorola.dataBus);
    break;
case 80://SP+5  Get IXL 1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IX.assignL(Motorola.dataBus);
    break;
case 81://SP+6        Get IYH   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IY.assignH(Motorola.dataBus);
    break;
case 82://SP+7        Get IYL   1
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.IY.assignL(Motorola.dataBus);
    break;
case 83:
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.PC.assignH(Motorola.dataBus);
    break;
case 84:
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.PC.assignL(Motorola.dataBus);
    break;
case 85:
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.PC.assignH(Motorola.dataBus);
    break;
case 86:
    Motorola.SP = Motorola.SP + 0x0001;
    Motorola.addrBus = Motorola.SP;
    Motorola.dataBus = Motorola.itsMem[Motorola.addrBus.value()];
    Motorola.PC.assignL(Motorola.dataBus);
    break;
case 87://00dd        (SPH)     0
    Motorola.addrBus = Motorola.dd;
    Motorola.dataBus = Motorola.SP.valueH();
    Motorola.R_W = 0;
    break;
case 88://00dd        (IXH)     0
    Motorola.addrBus = Motorola.dd;
    Motorola.dataBus = Motorola.IX.valueH();
    Motorola.R_W = 0;
    break;
case 89://00dd        (IYH)     0
    Motorola.addrBus = Motorola.dd;
    Motorola.dataBus = Motorola.IY.valueH();
    Motorola.R_W = 0;
    break;
case 90://00dd+1      (SPL)     1
    Motorola.addrBus = Motorola.dd+1;
    Motorola.dataBus = Motorola.SP.valueL();
    break;
```

```
case 91://00dd+1      (IXL)     0
    Motorola.addrBus = Motorola.dd+1;
    Motorola.dataBus = Motorola.IX.valueL();
    Motorola.R_W = 0;
    break;
case 92://00dd+1      (IYL)     0
    Motorola.addrBus = Motorola.dd+1;
    Motorola.dataBus = Motorola.IY.valueL();
    Motorola.R_W = 0;
    break;
case 93://hhll        (SPH)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll;
    Motorola.dataBus = Motorola.SP.valueH();
    Motorola.R_W = 0;
    break;
case 94://hhll        (IXH)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll;
    Motorola.dataBus = Motorola.IX.valueH();
    Motorola.R_W = 0;
    break;
case 95://hhll        (IYH)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll;
    Motorola.dataBus = Motorola.IY.valueH();
    Motorola.R_W = 0;
    break;
case 96://Hhll+1      (SPL)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
    Motorola.dataBus = Motorola.SP.valueL();
    Motorola.R_W = 0;
    break;
case 97://Hhll+1      (IXL)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
    Motorola.dataBus = Motorola.IX.valueL();
    Motorola.R_W = 0;
    break;
case 98://Hhll+1      (IYL)     0
    Motorola.addrBus = Motorola.hh*256+Motorola.ll+1;
    Motorola.dataBus = Motorola.IY.valueL();
    Motorola.R_W = 0;
    break;
case 99://X+ff        (SPH)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff;
    Motorola.dataBus = Motorola.SP.valueH();
    Motorola.R_W = 0;
    break;
case 100://X+ff       (IXH)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff;
    Motorola.dataBus = Motorola.IX.valueH();
    Motorola.R_W = 0;
    break;
case 101://X+ff       (IYH)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff;
    Motorola.dataBus = Motorola.IY.valueH();
    Motorola.R_W = 0;
    break;
case 102://X+ff+1     (SPL)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.SP.valueL();
    Motorola.R_W = 0;
    break;
case 103://X+ff+1     (IXL)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.IX.valueL();
    Motorola.R_W = 0;
    break;
case 104://X+ff+1     (IYL)     0
    Motorola.addrBus = Motorola.IX.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.IY.valueL();
    Motorola.R_W = 0;
    break;
case 105://Y+ff       (SPH)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff;
    Motorola.dataBus = Motorola.SP.valueH();
    Motorola.R_W = 0;
    break;
case 106://Y+ff       (IXH)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff;
    Motorola.dataBus = Motorola.IX.valueH();
    Motorola.R_W = 0;
    break;
case 107://Y+ff       (IYH)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff;
    Motorola.dataBus = Motorola.IY.valueH();
    Motorola.R_W = 0;
    break;
case 108://Y+ff+1     (SPL)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.SP.valueL();
    Motorola.R_W = 0;
    break;
case 109://Y+ff+1     (IXL)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.IX.valueL();
    Motorola.R_W = 0;
    break;
case 110://Y+ff+1     (IYL)     0
    Motorola.addrBus = Motorola.IY.value()+Motorola.ff+1;
    Motorola.dataBus = Motorola.IY.valueL();
```

```
            Motorola.R_W = 0;
            break;
        case 111://OP+3       ll        1
            Motorola.addrBus = Motorola.PC + 3;
            Motorola.dataBus=Motorola.ll;
            break;
        case 112://SP-2       (IYL)     0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.IY.valueL();
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 113://SP-3       (IYH)     0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.IY.valueH();
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 114://SP-4       (IXL)     0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.IX.valueL();
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 115://SP-5       (IXH)     0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.IX.valueH();
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 116://SP-6       (A)       0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.ACCA;
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 117://SP-7       (B)       0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.ACCB;
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.SP = Motorola.SP - 0x0001;
            Motorola.R_W = 0;
            break;
        case 118://SP-8       (CCR)     0
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.CCR.value();
            Motorola.itsMem[Motorola.addrBus.value()] = Motorola.dataBus;
            UpdateGridLocation(&Motorola.itsMem[Motorola.addrBus.value()]);
            Motorola.R_W = 0;
            break;
        case 119://SP-8       (CCR)     1
            Motorola.addrBus = Motorola.SP;
            Motorola.dataBus=Motorola.CCR.value();
            Motorola.SP = Motorola.SP - 0x0001;
            break;
        case 120://Vec hi     Svc hi    1
            Motorola.PC.assignH(Motorola.SWI_vector/0x100);
            break;
        case 121://Vec lo     Svc lo    1
            Motorola.PC.assignL(Motorola.SWI_vector%0x100);
            break;
        default:
            break;
    }
}
//---------------------------------------------------------------------------
```

typedef.h

```
#if !defined(TypedefH)
#define TypedefH

#include <stdio.h>
#include <vcl.h>

typedef unsigned char UCHAR;
typedef unsigned int UINT;
typedef long int LONGINT;
class asmLine;

/* Flag values for the "flags" field of a symbol */

#define BACKREF               0x01      /* Set when the symbol is defined on the 2nd pass */
#define REDEFINABLE           0x02      /* Set for symbols defined by the SET directive */
#define REG_LIST_SYM          0x04      /* Set for symbols defined by the REG directive */

/* Status values */

#define OK                    0x00

/* Severe errors */
#define SEVERE                0x400
#define SYNTAX                0x401
#define INV_OPCODE  0x402
#define INV_ADDR_MODE         0x403
#define LABEL_REQUIRED        0x404
#define PHASE_ERROR 0x405

/* Errors */
#define ERROR                 0x300
#define UNDEFINED   0x301
#define DIV_BY_ZERO 0x302
#define MULTIPLE_DEFS         0x303
#define REG_MULT_DEFS         0x304
#define REG_LIST_UNDEF        0x305
#define INV_FORWARD_REF       0x306
#define INV_LENGTH  0x307

/* Minor errors */
#define MINOR                 0x200
#define INV_SIZE_CODE         0x201
#define INV_QUICK_CONST       0x202
#define INV_VECTOR_NUM        0x203
#define INV_BRANCH_DISP 0x204
#define INV_DISP    0x205
#define INV_ABS_ADDRESS       0x206
#define INV_8_BIT_DATA        0x207
#define INV_16_BIT_DATA       0x208
#define ODD_ADDRESS 0x209
#define NOT_REG_LIST          0x20A
#define REG_LIST_SPEC         0x20B
#define INV_SHIFT_COUNT       0x20C

/* Warnings */
#define WARNING               0x100
#define ASCII_TOO_BIG         0x101
#define NUMBER_TOO_BIG        0x102
#define INCOMPLETE  0x103

/* TIMER STATES */
#define TIMER_IDLE                    0
#define TIMER_EXECUTE_ALL       1
#define TIMER_EXECUTE_INST      2
#define TIMER_EXECUTE_CYCLE     3
#define TIMER_EXECUTE_WAITKEY   4

#define SEVERITY    0xF00

/* The NEWERROR macros updates the error variable var only if the
   new error code is more severe than all previous errors.  Throughout
   ASM this is the standard means of reporting errors. */

#define NEWERROR(var, code)   if ((code & SEVERITY) > var) var = code
/* Symbol table definitions */
/* Significant length of a symbol */
#define SIGCHARS 8
/* Structure for operand descriptors */
typedef struct {
          long int data;      /* Immediate value, displacement, or absolute address */
          char backRef;       /* True if data field is known on first pass */
    char defined;
          } opDescriptor;
    /* Structure for a symbol table entry */
typedef struct symbolEntry {
          long int value;                       /* 32-bit value of the symbol */
          struct symbolEntry *next;     /* Pointer to next symbol in linked list */
          char flags;                           /* Flags (see below) */
          char name[SIGCHARS+1];                /* Name */
          } symbolDef;

/* Status values */

#define OK                    0x00
```

```
#define NEWERROR(var, code)   if ((code & SEVERITY) > var) var = code

#define isDelim(c)   ((c == '\r') || (c == '\n') || (c == ',') || c == '\0')
#define isTerm(c)   (isspace(c) || (c == ',') || c == '\0')
#define isRegNum(c) ((c >= '0') && (c <= '7'))
/* Addressing mode codes/bitmasks */
#define ADDR_IMMEDIATE      0x0001
#define ADDR_INDEXEDX       0x0002
#define ADDR_INDEXEDY       0x0004
#define ADDR_DIRECT         0x0008
#define ADDR_EXTENDED       0x0010
#define ADDR_INHERENT       0x0020

#define DnDirect                        0x00001
#define AnDirect                        0x00002
#define AnInd                                   0x00004
#define AnIndPost                       0x00008
#define AnIndPre                        0x00010
#define AnIndDisp                       0x00020
#define AnIndIndex              0x00040
#define AbsShort                        0x00080
#define AbsLong                             0x00100
#define PCDisp                                  0x00200
#define PCIndex                             0x00400
#define Immediate                       0x00800
#define SRDirect                        0x01000
#define CCRDirect                       0x02000
#define USPDirect                       0x04000
#define SFCDirect                       0x08000
#define DFCDirect                       0x10000
#define VBRDirect                       0x20000
/* Register and operation size codes/bitmasks */
#define BYTE        ((int) 1)
#define WORD        ((int) 2)
#define LONG        ((int) 4)
#define SHORT       ((int) 8)
/* OpDescriptor default values */
#define OPDESC_DEF_MODE     (0)
#define OPDESC_DEF_DATA     (0xffff)
#define OPDESC_DEF_REG      (0xff)
#define OPDESC_DEF_INDEX    (0xff)
#define OPDESC_DEF_SIZE     (0xff)
#define OPDESC_DEF_BACKREF  (0)
#define OPDESC_DEF_REG (0xff)
typedef struct cmd_str{
    unsigned char cmd[20];
    unsigned char len;
          } cmd_str;

typedef struct{
          unsigned int val[8];
          } cycleType;

typedef struct{
          unsigned int val[16];
          } MachineCycleType;

typedef struct instruction{
          char *mnemonic;               /* Mnemonic */
          unsigned int branch;
          unsigned int hexcode[6];
          int cycle[6];
          int pccount[6];
          int cyclePtr[6];
          int (*exec)(asmLine *);//opDescriptor *, opDescriptor *, UINT);
          } instruction;

typedef enum {
    BIT_CARRY=0,
    BIT_OVERFLOW=1,
    BIT_ZERO=2,
    BIT_NEGATIVE=3,
    BIT_INTERRUPT=4,
    BIT_HALFCARRY=5,
    BIT_XINTERRUPT=6,
    BIT_STOP=7
}ccrbits;

typedef struct TimerStruct{
          unsigned int State;
          unsigned int Key;
          } TimerStruct;

typedef struct MachineCycleDefStruct{
          unsigned int id;
          char *address;                /* Mnemonic */
          char *data;                   /* Mnemonic */
          char *r_w;                    /* Mnemonic */
          } MachineCycleDefStruct;

typedef struct IODisplayType{
    AnsiString *AddrBus[16];
    AnsiString *DataBus[8];
          } IODisplayType;
#endif
```

OpCode.h

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "typedef.h"
#include "asmLine.h"

asmLine::asmLine(UINT lno, char *linestr)//constructor
{
    itsMemLoc = 0xffff;
    itsLabel = 0;
    itsSWInterrupt=false;
    itsNMIInterrupt=false;
    itsExecLineNo = lno;
    itsDebugMode = 0;
    itsFilePtr = 0;
    itsJmpLbl[0] = NULL;
    itsJmpAddr = 0;
    itsOpCode = new OpCode;
    strcpy ((char *)&itsLineStr, linestr);
    addrMode = ADDR_INHERENT;
    sourceDefined = false;
    destDefined = false;
    maskDefined = false;
    relDefined = false;
    LViewId = 0xffff;
}

void asmLine::assignLabel(char *data)
{
    itsLabel = 1;
    strcpy (itsLbl,data);
}

UINT asmLine::assignMemLoc(UINT data)
{
    itsMemLoc = data;
    return itsOpCode->QueryPCIncrement(addrMode);
}

asmLine::~asmLine()//destructor
{
    delete itsOpCode;
    delete []itsLineStr;
}

void asmLine::OutStr (char *dest)
{
    char *cmp;
    char numstr[25];
    cmp = (char *)&itsLineStr;
        do{
            if (*cmp == '\n'){
                *cmp = '\0';
                break;
            }
            cmp++;
        }while(*cmp != '\0');

    strcpy (dest, itsLineStr);
}
```

OpCode.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include <StdCtrls.hpp>

#include "typedef.h"
#include "OpCode.h"

OpCode::OpCode()//Constructor
{
    itsInstruction = NULL;
    source = new opDescriptor;
    dest = new opDescriptor;
    mask = new opDescriptor;
    rel = new opDescriptor;
    source->data = 0;
    dest->data = 0;
}
//---------------------------------------------------------------------------
void OpCode::OutStr(char *deststr)
{
    strcat (deststr, "\nItsSource -- ");
    this->OutStrCurr(source, deststr);
    strcat (deststr, "\nItsDest -- ");
    this->OutStrCurr(dest, deststr);
    strcat (deststr, "\nItsMask -- ");
    this->OutStrCurr(mask, deststr);
    strcat (deststr, "\nItsRel -- ");
    this->OutStrCurr(rel, deststr);
}
//---------------------------------------------------------------------------
UCHAR OpCode::QueryPCIncrement(UINT addrmode)
{
    UCHAR result = 0xff;
    if (itsInstruction == NULL)
        return 0;//Line da sadece label varsa
    switch (addrmode){
        case 0:
            break;
        case ADDR_IMMEDIATE:
            result = itsInstruction->pccount[0];
            break;
        case ADDR_INDEXEDX:
            result = itsInstruction->pccount[3];
            break;
        case ADDR_INDEXEDY:
            result = itsInstruction->pccount[4];
            break;
        case ADDR_DIRECT:
            result = itsInstruction->pccount[1];
            break;
        case ADDR_EXTENDED:
            result = itsInstruction->pccount[2];
            break;
        case ADDR_INHERENT:
            result = itsInstruction->pccount[5];
            break;
    }
    return result;
}
//---------------------------------------------------------------------------
OpCode::~OpCode()//Destructor
{
    delete source;
    delete dest;
    delete mask;
    delete rel;
}
//---------------------------------------------------------------------------
void OpCode::GetHexStr(char *OutsStr, UINT adr)
{
    UCHAR addrmode=0;
    char deststr[10];
    char srcstr[10];
    switch (adr){
        case 0:
            break;
        case ADDR_IMMEDIATE:
                addrmode = 0;
                strcat ((char *)&srcstr, IntToHex((int)source->data,2).c_str());
            break;
        case ADDR_INDEXEDX:
                addrmode = 3;
            break;
        case ADDR_INDEXEDY:
                addrmode = 4;
            break;
        case ADDR_DIRECT:
                addrmode = 1;
                strcat ((char *)&srcstr, IntToHex((int)source->data,2).c_str());
            break;
        case ADDR_EXTENDED:
                addrmode = 2;
                strcat ((char *)&srcstr, IntToHex((int)source->data,4).c_str());
```

```
            break;
        case ADDR_INHERENT:
                addrmode = 5;
            break;
    }
    if (itsInstruction->hexcode[addrmode]<0x100)
        strcat (OutsStr, IntToHex((int)itsInstruction->hexcode[addrmode],2).c_str());
    else{
        strcat (OutsStr, IntToHex((int)itsInstruction->hexcode[addrmode],4).c_str());
    }
    strcat (OutsStr, IntToHex((int)source->data,2).c_str());
}
//---------------------------------------------------------------------------
```

asmLine.h

```
#if !defined(asmLineH)
#define asmLineH

#include "OpCode.h"

class asmLine :public OpCode
{
private:
    UINT itsExecLineNo;
    UCHAR itsDebugMode;
    LONGINT itsFilePtr;//text file location
public:
    BOOL itsSWInterrupt;
    BOOL itsNMIInterrupt;
    char itsLineStr[256];
    UCHAR id;//Instruction table index for the found item
    UINT LViewId;
    UINT itsMemLoc;
    OpCode *itsOpCode;
    UINT addrMode;
    BOOL itsLabel;
    char itsLbl[100];
    char itsJmpLbl[100];
    UINT itsJmpAddr;
    cmd_str hcodes;
    bool sourceDefined;
    bool destDefined;
    bool maskDefined;
    bool relDefined;
    asmLine(UINT lno, char *linestr);//constructor
    UINT assignMemLoc(UINT data);
    void assignLabel(char *data);
    UINT queryExecLineNo(){return itsExecLineNo;};
    void OutStr (char *dest);
    ~asmLine();//destructor
};
#endif    // asmLineH
```

asmLine.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "typedef.h"
#include "asmLine.h"

asmLine::asmLine(UINT lno, char *linestr)//constructor
{
    itsMemLoc = 0xffff;
    itsLabel = 0;
    itsSWInterrupt=false;
    itsNMIInterrupt=false;
    itsExecLineNo = lno;
    itsDebugMode = 0;
    itsFilePtr = 0;
    itsJmpLbl[0] = NULL;
    itsJmpAddr = 0;
    itsOpCode = new OpCode;
    strcpy ((char *)&itsLineStr, linestr);
    addrMode = ADDR_INHERENT;
    sourceDefined = false;
    destDefined = false;
    maskDefined = false;
    relDefined = false;
    LViewId = 0xffff;
}

void asmLine::assignLabel(char *data)
{
    itsLabel = 1;
    strcpy (itsLbl,data);
}

UINT asmLine::assignMemLoc(UINT data)
{
    itsMemLoc = data;
    return itsOpCode->QueryPCIncrement(addrMode);
}

asmLine::~asmLine()//destructor
{
    delete itsOpCode;
    delete []itsLineStr;
}

void asmLine::OutStr (char *dest)
{
    char *cmp;
    char numstr[25];
    cmp = (char *)&itsLineStr;
        do{
            if (*cmp == '\n'){
                *cmp = '\0';
                break;
            }
            cmp++;
        }while(*cmp != '\0');

    strcpy (dest, itsLineStr);
}
```

linked.h

```
#if !defined(LinkedH)
#define LinkedH

#include <StdCtrls.hpp>
#include <ComCtrls.hpp>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

template <class T>
class Node {
private:
    Node<T> *next;
public:
    T data;
    void print (TRichEdit *PWindow){
        char outstr[256];
        data.OutStr((char *)&outstr);
        PWindow->Lines->Add(outstr);
    };
    Node (const T& item, Node<T> *ptrnext = NULL):data(item),next(ptrnext){};
    void insertAfter(Node<T> *p){p->next = next; next = p;};
    Node<T> deleteAfter(){Node<T> &temp = next; if (next == NULL) return NULL;\
                          next = temp->next; return temp;};
    Node<T> *nextNode() {return next;};
    void AssignNextNode(Node<T> *nextNode) {next = nextNode;};
};
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
template <class T>
class LinkedList{
private:
    Node<T> *front;
    Node<T> *getNode(const T& item, Node<T> *ptrNext = NULL){
        Node<T> *newnode;
        newnode = new Node<T>(item,ptrNext);
        return newnode;
    };
    void freeNode(Node<T> *p){delete p;};
public:
    Node<T> *current;
//---------------------------------------------------------------------------
    void resetCurrent() {current = front;};
//---------------------------------------------------------------------------
    LinkedList(){front = NULL;};
//---------------------------------------------------------------------------
    ~LinkedList(){
        Node<T> *curpos, *nextpos;
        curpos = front;
        while(curpos != NULL){
            nextpos = curpos->nextNode();
            freeNode(curpos);
            curpos = nextpos;
        }
    };
//---------------------------------------------------------------------------
    int listSize() {
        Node<T> *temp = front;
        int size = 0;
        if (front)
        size = 1;
            while(temp->nextNode()){
            temp = temp->nextNode();
            size++;
            }
    return size;
    };
//---------------------------------------------------------------------------
    int ListEmpty() const {return (!front);};
//---------------------------------------------------------------------------
    void Insert(const T& item){
        Node<T> *temp = front;
        if (!front)
            front = getNode(item, NULL);
        else{
            while(temp->nextNode())
                temp = temp->nextNode();
            temp->AssignNextNode(getNode(item, NULL));
        }
    };
//---------------------------------------------------------------------------
    void PrintList(TRichEdit *PWindow)
    {
        Node<T> *curpos, *nextpos;
        curpos = front;
        while(curpos != NULL){
            nextpos = curpos->nextNode();
            curpos->print(PWindow);
            curpos = nextpos;
        }
    };

};//end of the class declaration
```

micro.hpp

```
#if !defined(MicroH)
#define MicroH

#include "reg16Bit.h"
#include "regCCR.h"
#include "typedef.h"

class Micro
{
private:
public:
    UCHAR ACCA, ACCB;
    RegCCR CCR;
    Reg16Bit IX, IY;
    Reg16Bit PC;
    Reg16Bit SP;
    Reg16Bit addrBus;
    UCHAR dataBus;
    UCHAR R_W;
    UCHAR itsMem[0x10000];

    UCHAR ii,jj,kk,hh,ll,dd,mm,ff,rr,result,MM,Nxtop;
    UINT SWI_vector;
    Reg16Bit DPTRa;
    Reg16Bit DPTRb;
    Reg16Bit DPTRc;
    UCHAR mode;


    Micro();
    void UpdatePC(asmLine *curLine);
    ~Micro(){delete []itsMem;};//destructor
};
#endif    // MicroH
```

micro.cpp

```
#include <stdio.h>
#include <iostream.h>
#include <ctype.h>

#include "typedef.h"
#include "asmLine.h"

#include "micro.hpp"
#include "regCCR.h"

extern UINT SPVector;
extern UINT PCVector;

Micro::Micro()//Constructor
{

    ACCA = ACCB =0;
    CCR = 0;

    IX = IY = 0;
    PC = PCVector;
    SP = SPVector;
    R_W = 0;
    addrBus = 0xffff;
    dataBus = 0;
    SWI_vector = 0xffff;
};//constructor

void Micro::UpdatePC(asmLine *curLine)
{
    UCHAR incVal = curLine->itsOpCode->QueryPCIncrement(curLine->addrMode);
    if (incVal == 0xff)//error
        ;
    else{
        if (!curLine->itsOpCode->itsInstruction->branch==1)//PC is determined in execute for branch
            PC=PC+incVal;
    }
}
```

reg16Bit.h

```
#if !defined(Reg16bitH)
#define Reg16bitH

#include "typedef.h"

class Reg16Bit
{
private:
    UINT data;
public:
    Reg16Bit &operator=(UINT );
    Reg16Bit &operator=(const Reg16Bit &);
    UINT operator+(UINT );
    Reg16Bit &operator+(const Reg16Bit &);
    Reg16Bit &operator-(UINT );
    Reg16Bit &operator-(const Reg16Bit &);
    UINT value(){return data;};
    UINT valueH(){return (data/256);};
    UINT valueL(){return (data%256);};
    void assignH(UINT val){data&=0x00ff;data+=(val&0xff)<<8;};
    void assignL(UINT val){data&=0xff00;data+=val&0xff;};
    Reg16Bit(){};//constructor
    ~Reg16Bit(){};//destructor
};
#endif    // Reg16bitH
```

reg16bit.cpp

```
#include "reg16Bit.h"

Reg16Bit &Reg16Bit::operator=(UINT rhs)
{
    if (this->data != rhs){
        data = rhs;
    }
    return *this;
}

Reg16Bit &Reg16Bit::operator=(const Reg16Bit &rhs)
{
    if (this != &rhs){
        data = rhs.data;
    }
    return *this;
}

UINT Reg16Bit::operator+(UINT rhs)
{
    return (this->data+rhs);
}

Reg16Bit &Reg16Bit::operator+(const Reg16Bit &rhs)
{
    data += rhs.data;
    return *this;
}
Reg16Bit &Reg16Bit::operator-(UINT rhs)
{
    data -= rhs;
    return *this;
}

Reg16Bit &Reg16Bit::operator-(const Reg16Bit &rhs)
{
    data -= rhs.data;
    return *this;
}
```

regCCR.h

```
#if !defined(RegCCRH)
#define RegCCRH
#include "typedef.h"
#include "math.h"
class RegCCR
{
private:
    UCHAR data;
public:
    bool GetBit(UCHAR bitpos){
        return bool(data & UCHAR(pow(2, bitpos)));
    };
    void AssignBit(UCHAR bitpos, bool val){
    UCHAR rhs = val<<bitpos;
        data &= ~UCHAR(pow(2, bitpos));
        data += rhs;
    };
//---------------------------------------------------------------------------
    void Carry8bit(UINT op1,UINT op2,UINT result){
        this->AssignBit(BIT_CARRY,
                        bool(op1&0x80)&bool(op2&0x80)|\
                        bool(op2&0x80)&~bool(result&0x80)|\
                        bool(op1&0x80)&~bool(result&0x80)\
                                    );
    }
//---------------------------------------------------------------------------
    void Carry8Negate(UINT result){
        this->AssignBit(BIT_CARRY,
                        bool(result&0x80)|bool(result&0x40)|\
                        bool(result&0x20)|bool(result&0x10)|\
                        bool(result&0x08)|bool(result&0x04)|\
                        bool(result&0x02)|bool(result&0x01)
                                    );
    }
//---------------------------------------------------------------------------
    void OverFlow8bit(UINT op1,UINT op2,UINT result){
        this->AssignBit(BIT_OVERFLOW, \
                bool(op1&0x80)&bool(op2&0x80)&~bool(result&0x80)|\
                ~bool(op1&0x80)&~bool(op2&0x80)&bool(result&0x80)\
                );
    }
//---------------------------------------------------------------------------
    void OverFlow8bitExor(bool op1, bool op2){
        this->AssignBit(BIT_OVERFLOW, op1^op2);
    }
//---------------------------------------------------------------------------
    void Zero8bit(UINT result){
        this->AssignBit(BIT_ZERO, ~bool(result));
    }
//---------------------------------------------------------------------------
    void Negative8bit(UINT result){
        this->AssignBit(BIT_NEGATIVE, bool(result&0x80));
    }
//---------------------------------------------------------------------------
    void HalfCarry8bit(UINT op1, UINT op2, UINT result){
        this->AssignBit(BIT_HALFCARRY, \
                bool(op1&0x08)&bool(op2&0x08)|\
                bool(op2&0x08)&bool(result&0x08)|\
                bool(op1&0x08)&bool(result&0x08)
                );
    }
//---------------------------------------------------------------------------
    void Carry16bit(UINT op1,UINT op2,UINT result){
        this->AssignBit(BIT_CARRY,
                        bool(op1&0x8000)&bool(op2&0x8000)|\
                        bool(op2&0x8000)&~bool(result&0x8000)|\
                        bool(op1&0x8000)&~bool(result&0x8000)\
                                    );
    }
//---------------------------------------------------------------------------
    void OverFlow16bit(UINT op1,UINT op2,UINT result){
        this->AssignBit(BIT_OVERFLOW, \
                bool(op1&0x8000)&bool(op2&0x8000)&~bool(result&0x8000)|\
                ~bool(op1&0x8000)&~bool(op2&0x8000)&bool(result&0x8000)\
                );
    }
//---------------------------------------------------------------------------
    void Zero16bit(UINT result){
        this->AssignBit(BIT_ZERO, ~bool(result));
    }
//---------------------------------------------------------------------------
    void Negative16bit(UINT result){
        this->AssignBit(BIT_NEGATIVE, bool(result&0x8000));
    }
//---------------------------------------------------------------------------
    bool operator=(UCHAR val){data = val;return 1;};
//---------------------------------------------------------------------------
    UCHAR value(void){return data;};
    void AssignData(UCHAR val){data = val;};
//---------------------------------------------------------------------------
    RegCCR(){};//constructor
    ~RegCCR(){};//destructor
};
#endif    // RegBitAddressable
//---------------------------------------------------------------------------
```

instTable.h

```
//mnemonic + hexcode[6] + cycle[6] + pccount[6]
//Immediate + Direct + Extended + IndexedX + IndexedY + Inherent
const instruction instTable[] = {
//{{0,1,2,3,4,5,6,7},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5,6,7},{0,1,2,3,4,5,6,7}}
{ "ABA",  0, {0,0,0,0,0,0x1B}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}  ,
instExec_ABA },
    { "ABX",  0, {0,0,0,0,0,0x3A}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_ABX  },
    { "ABY",  0, {0,0,0,0,0,0x1B3A}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_ABY  },
    { "ADCA", 0, {0x89,0x99,0xB9,0xA9,0x18A9,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ADCA },
    { "ADCB", 0, {0xC9,0xD9,0xF9,0xE9,0x18E9,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ADCB },
    { "ADDA", 0, {0xBB,0x9B,0xBB,0xAB,0x18AB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ADDA },
    { "ADDB", 0, {0xCB,0xDB,0xFB,0xEB,0x18EB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ADDB },
    { "ADDD", 0, {0xC3,0xD3,0xF3,0xE3,0x18E3,0}, {4,5,6,6,7,-1}    , {3,2,3,2,3,0}, {89,13,47,31,69,0xff}
,instExec_ADDD },
    { "ANDA", 0, {0x84,0x94,0xB4,0xA4,0x18AB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ANDA },
    { "ANDB", 0, {0xC4,0xD4,0xF4,0xE4,0x18EB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ANDB },
    { "ASL",  0, {0,0,0x78,0x68,0x1868,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_ASL  },
    { "ASLA", 0, {0,0,0,0,0,0x48}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ASLA },
    { "ASLB", 0, {0,0,0,0,0,0x58}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ASLB },
    { "ASLD", 0, {0,0,0,0,0,0x05}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_ASLD },
    { "ASR",  0, {0,0,0x77,0x67,0x1868,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_ASR  },
    { "ASRA", 0, {0,0,0,0,0,0x47}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ASRA },
    { "ASRB", 0, {0,0,0,0,0,0x57}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ASRB },
    { "BCC",  1, {0,0,0,0,0,0x24}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2}, {0,0,0,0,0,91}
,instExec_BCC  },
    { "BCLR", 0, {0,0x15,0,0x1D,0x181D,0}      , {-1,6,-1,7,8,-1}  , {0,3,0,3,4,0}, {0xff,16,0xff,29,67,0xff}
,instExec_BCLR },
    { "BCS",  1, {0,0,0,0,0,0x25}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BCS  },
    { "BEQ",  1, {0,0,0,0,0,0x27}              , {-1,-1,-1,-1,-1,3}, {0,0,2,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BEQ  },
    { "BGE",  1, {0,0,0,0,0,0x2C}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BGE  },
    { "BGT",  1, {0,0,0,0,0,0x2E}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BGT  },
    { "BHI",  1, {0,0,0,0,0,0x22}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BHI  },
    { "BHS",  1, {0,0,0,0,0,0x24}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BHS  },
    { "BITA", 0, {0x85,0x95,0xB5,0xA5,0x18AB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_BITA },
    { "BITB", 0, {0xC5,0xD5,0xF5,0xE5,0x18EB,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_BITB },
    { "BLE",  1, {0,0,0,0,0,0x2F}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BLE  },
    { "BLO",  1, {0,0,0,0,0,0x25}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BLO  },
    { "BLS",  1, {0,0,0,0,0,0x23}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BLS  },
    { "BLT",  1, {0,0,0,0,0,0x2D}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BLT  },
    { "BMI",  1, {0,0,0,0,0,0x2B}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BMI  },
    { "BNE",  1, {0,0,0,0,0,0x26}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BNE  },
    { "BPL",  1, {0,0,0,0,0,0x2A}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BPL  },
    { "BRA",  1, {0,0,0,0,0,0x20}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BRA  },
    { "BRCLR",1, {0,0x13,0,0x1F,0x181F,0}      , {-1,6,-1,7,8,-1}  , {0,4,0,5,5,0}, {0xff,14,0xff,28,66,0xff}
,instExec_BRCLR},
    { "BRN",  1, {0,0,0,0,0,0x21}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BRN  },
    { "BRSET",1, {0,0x12,0,0x1E,0x181E,0}      , {-1,6,-1,7,8,-1}  , {0,2,0,2,3,0}, {0xff,14,0xff,28,66,0xff}
,instExec_BRSET},
    { "BSET", 0, {0,0x14,0,0x1C,0x181C,0}      , {-1,6,-1,7,8,-1}  , {0,2,0,2,3,0}, {0xff,15,0xff,29,67,0xff}
,instExec_BSET },
    { "BSR",  1, {0,0,0,0,0,0x8D}              , {-1,-1,-1,-1,-1,6}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,92} ,instExec_BSR  },
    { "BVC",  1, {0,0,0,0,0,0x28}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BVC  },
    { "BVS",  1, {0,0,0,0,0,0x29}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,91} ,instExec_BVS  },
    { "CBA",  0, {0,0,0,0,0,0x11}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CBA  },
    { "CLC",  0, {0,0,0,0,0,0x0c}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CLC  },
    { "CLI",  0, {0,0,0,0,0,0x0e}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CLI  },
```

```
    { "CLR",  0, {0,0,0x7F,0x6F,0x186F,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,49,34,72,0xff}
,instExec_CLR  },
    { "CLRA", 0, {0,0,0,0,0,0x4F}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CLRA },
    { "CLRB", 0, {0,0,0,0,0,0x5F}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CLRB },
    { "CLV",  0, {0,0,0,0,0,0x0a}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_CLV  },
    { "CMPA", 0, {0x81,0x91,0xB1,0xA1,0x18A1,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {87,11,45,27,65,0xff}
,instExec_CMPA },
    { "CMPB", 0, {0xC1,0xD1,0xF1,0xE1,0x18E1,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {87,11,45,27,65,0xff}
,instExec_CMPB },
    { "COM",  0, {0,0,0x73,0x63,0x1863,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_COM  },
    { "COMA", 0, {0,0,0,0,0,0x43}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_COMA },
    { "COMB", 0, {0,0,0,0,0,0x53}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_COMB },
    { "CPD",  0, {0x1A83,0x1A93,0x1AB3,0x1AB3,0xCDA3,0}, {5,6,7,7,7,-1},{3,3,4,3,3,0},{82,84,58,64,70,0xff}
,instExec_CPD  },
    { "CPX",  0, {0x8C,0x9C,0xBC,0xAC,0xCDAC,0}, {4,5,6,6,7,-1}    , {2,2,3,2,3,0}, {89,13,39,31,69,0xff}
,instExec_CPX  },
    { "CPY",  0, {0x188C,0x189C,0x18BC,0x18AC,0x18AC,0}, {5,6,7,7,7,-1},{4,3,4,3,3,0},{82,84,58,64,70,0xff}
,instExec_CPX  },
    { "DAA",  0, {0,0,0,0,0,0x19}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_DAA  },
    { "DEC",  0, {0,0,0x7A,0x6A,0x186A,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_DEC  },
    { "DECA", 0, {0,0,0,0,0,0x4A}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_DECA },
    { "DECB", 0, {0,0,0,0,0,0x53}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_DECB },
    { "DES",  0, {0,0,0,0,0,0x34}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,1}
,instExec_DES  },
    { "DEX",  0, {0,0,0,0,0,0x09}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_DEX  },
    { "DEY",  0, {0,0,0,0,0,0x1809}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_DEY  },
    { "EORA", 0, {0x88,0x98,0xB8,0xA8,0x18A1,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_EORA },
    { "EORB", 0, {0xC8,0xD8,0xF8,0xE8,0x18E8,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_EORB },
    { "FDIV",  0  ,  {0,0,0,0,0,0x03}    , {-1,-1,-1,-1,-1,41}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,10}
,instExec_FDIV },
    { "IDIV",  0  ,  {0,0,0,0,0,0x02}    , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,10}
,instExec_IDIV },
    { "INC",  0, {0,0,0x7C,0x6C,0x186C,0}      , {-1,-1,6,6,7,-1}  , {0,0,6,6,7,0}, {0xff,0xff,50,33,73,0xff}
,instExec_INC  },
    { "INCA", 0, {0,0,0,0,0,0x4C}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_INCA },
    { "INCB", 0, {0,0,0,0,0,0x5C}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_INCB },
    { "INS",  0, {0,0,0,0,0,0x31}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,1}
,instExec_INS  },
    { "INX",  0, {0,0,0,0,0,0x08}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_INX  },
    { "INY",  0, {0,0,0,0,0,0x1808}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_INY  },
    { "JMP",  1, {0,0,0x7E,0x6E,0x186E,0}      , {-1,-1,3,3,4,-1}  , {0,0,3,2,3,0}, {0xff,0xff,41,24,59,0xff}
,instExec_JMP  },
    { "JSR",  1, {0,0x9D,0xBD,0xAD,0x18AD,0}   , {-1,5,6,6,7,-1}   , {0,2,3,2,3,0}, {0xff,17,48,32,71,0xff}
,instExec_JSR  },
    { "LDAA", 0, {0x86,0x96,0xB6,0xA6,0x18A6,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_LDAA },
    { "LDAB", 0, {0xC6,0xD6,0xF6,0xE6,0x18E6,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_LDAB },
    { "LDD",  0, {0xCC,0xDC,0xFC,0xEC,0x18EC,0}, {3,4,5,5,6,-1}    , {2,2,3,2,3,0}, {86,12,46,30,68,0xff}
,instExec_LDD  },
    { "LDS",  0, {0xBE,0x9E,0xEE,0xAE,0x18AE,0}, {3,4,5,5,6,-1}    , {2,2,3,2,3,0}, {88,12,46,30,68,0xff}
,instExec_LDS  },
    { "LDX",  0, {0xCE,0xDE,0xFE,0xEE,0xCDEE,0}, {3,4,5,5,6,-1}    , {2,2,3,2,3,0}, {88,12,46,30,68,0xff}
,instExec_LDX  },
    { "LDY",  0, {0x18CE,0x18DE,0x18FE,0x1AEE,0x18EE,0},{4,5,6,6,6,-1},{3,3,4,3,3,0},{81,83,57,63,68,0xff}
,instExec_LDY  },
    { "LSL",  0, {0,0,0x78,0x68,0x1868,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_LSL  },
    { "LSLA", 0, {0,0,0,0,0,0x48}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_LSLA },
    { "LSLB", 0, {0,0,0,0,0,0x58}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_LSLB },
    { "LSLD", 0, {0,0,0,0,0,0x05}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_LSLD },
    { "LSR",  0, {0,0,0x74,0x64,0x1864,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_LSR  },
    { "LSRA", 0, {0,0,0,0,0,0x44}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_LSRA },
    { "LSRB", 0, {0,0,0,0,0,0x54}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_LSRB },
    { "LSRD", 0, {0,0,0,0,0,0x05}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_LSRD },
    { "MUL",  0, {0,0,0,0,0,0x3D}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_MUL  },
    { "NEG",  0, {0,0,0x70,0x60,0x1860,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_NEG  },
    { "NEGA", 0, {0,0,0,0,0,0x40}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_NEGA },
    { "NEGB", 0, {0,0,0,0,0,0x50}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_NEGB },
```

```
    { "NOP",  0, {0,0,0,0,0,0x01}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_NOP  },
    { "ORAA", 0, {0x8A,0x9A,0xBA,0xAA,0x18AA,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ORAA },
    { "ORAB", 0, {0xCA,0xDA,0xFA,0xEA,0x18EA,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_ORAB },
    { "PSHA", 0, {0,0,0,0,0,0x36}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,7}
,instExec_PSHA },
    { "PSHB", 0, {0,0,0,0,0,0x37}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,8}
,instExec_PSHB },
    { "PSHX", 0, {0,0,0,0,0,0x3C}              , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,9}
,instExec_PSHX },
    { "PSHY", 0, {0,0,0,0,0,0x183C}            , {-1,-1,-1,-1,-1,5}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,79} ,instExec_PSHY },
    { "PULA", 0, {0,0,0,0,0,0x32}              , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,2}
,instExec_PULA },
    { "PULB", 0, {0,0,0,0,0,0x33}              , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,3}
,instExec_PULB },
    { "PULX", 0, {0,0,0,0,0,0x3C}              , {-1,-1,-1,-1,-1,5}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,4}
,instExec_PULX },
    { "PULY", 0, {0,0,0,0,0,0x183C}            , {-1,-1,-1,-1,-1,6}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,78} ,instExec_PULY },
    { "ROL",  0, {0,0,0x79,0x69,0x1869,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_ROL  },
    { "ROLA", 0, {0,0,0,0,0,0x49}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ROLA },
    { "ROLB", 0, {0,0,0,0,0,0x59}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_ROLB },
    { "ROR",  0, {0,0,0x76,0x66,0x1866,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,50,33,73,0xff}
,instExec_ROR  },
    { "RORA", 0, {0,0,0,0,0,0x46}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_RORA },
    { "RORB", 0, {0,0,0,0,0,0x56}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_RORB },
    { "RTI",  0, {0,0,0,0,0,0x3B}              , {-1,-1,-1,-1,-1,12},{0,0,0,0,0,0/*Not to increment PC by 1*/},
{0xff,0xff,0xff,0xff,0xff,5}  ,instExec_RTI  },
    { "RTS",  0, {0,0,0,0,0,0x39}              , {-1,-1,-1,-1,-1,5}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,6}
,instExec_RTS  },
    { "SBA",  0, {0,0,0,0,0,0x10}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_SBA  },
    { "SBCA", 0, {0x82,0x92,0xB2,0xA2,0x18A2,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_SBCA },
    { "SBCB", 0, {0xC2,0xD2,0xF2,0xE2,0x18E2,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_SBCB },
    { "SEC",  0, {0,0,0,0,0,0x0D}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_SEC  },
    { "SEI",  0, {0,0,0,0,0,0x0F}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_SEI  },
    { "SEV",  0, {0,0,0,0,0,0x0B}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_SEV  },
    { "STAA", 0, {0,0x97,0xB7,0xA7,0x18A7,0}   , {-1,3,4,4,5,-1}   , {0,2,3,2,3,0}, {0xff,18,43,35,74,0xff}
,instExec_STAA },
    { "STAB", 0, {0,0xD7,0xF7,0xE7,0x18E7,0}   , {-1,3,4,4,5,-1}   , {0,2,3,2,3,0}, {0xff,20,42,37,76,0xff}
,instExec_STAB },
    { "STD",  0, {0,0xDD,0xFD,0xED,0x18ED,0}   , {-1,4,5,5,6,-1}   , {0,2,3,2,3,0}, {0xff,19,44,36,75,0xff}
,instExec_STD  },
    { "STOP", 0, {0,0,0,0,0,0xCF}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_STOP },
    { "STS",  0, {0,0x9F,0xBF,0xAF,0x18AF,0}   , {-1,4,5,5,6,-1}   , {0,2,3,2,3,0}, {0xff,21,53,38,60,0}
,instExec_STS  },
    { "STX",  0, {0,0xDF,0xFF,0xEF,0xCDEF,0}   , {-1,4,5,5,6,-1}   , {0,2,3,2,3,0}, {0xff,22,54,25,61,0}
,instExec_STX  },
    { "STY",  0, {0,0xDF,0xFF,0xEF,0xCDEF,0}   , {-1,4,5,5,6,-1}   , {0,2,3,2,3,0}, {0xff,23,55,26,62,0}
,instExec_STY  },
    { "SUBA", 0, {0x80,0x90,0xB0,0xA0,0x18A0,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_SUBA },
    { "SUBB", 0, {0xC0,0xD0,0xF0,0xE0,0x18E0,0}, {2,3,4,4,5,-1}    , {2,2,3,2,3,0}, {85,11,45,27,65,0xff}
,instExec_SUBB },
    { "SUBD", 0, {0x83,0x93,0xB3,0xA3,0x18A3,0}, {4,5,6,6,7,-1}    , {3,2,3,2,3,0}, {90,12,40,31,69,0xff}
,instExec_SUBD },
    { "SWI",  1, {0,0,0,0,0,0x3F}              , {-1,-1,-1,-1,-1,14},{0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,93} ,instExec_SWI  },
    { "TAB",  0, {0,0,0,0,0,0x16}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TAB  },
    { "TAP",  0, {0,0,0,0,0,0x06}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TAP  },
    { "TBA",  0, {0,0,0,0,0,0x17}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TBA  },
    { "TEST", 0, {0,0,0,0,0,0x17}              , {-1,-1,-1,-1,-1,5}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,94} ,instExec_TEST },
    { "TPA",  0, {0,0,0,0,0,0x07}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TPA  },
    { "TST",  0, {0,0,0x7D,0x6D,0x186D,0}      , {-1,-1,6,6,7,-1}  , {0,0,3,2,3,0}, {0xff,0xff,51,50,52,0xff}
,instExec_TST  },
    { "TSTA", 0, {0,0,0,0,0,0x4D}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TSTA },
    { "TSTB", 0, {0,0,0,0,0,0x5D}              , {-1,-1,-1,-1,-1,2}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,0}
,instExec_TSTB },
    { "TSX",  0, {0,0,0,0,0,0x30}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1}, {0xff,0xff,0xff,0xff,0xff,1}
,instExec_TSX  },
    { "TSY",  0, {0,0,0,0,0,0x1830}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,77} ,instExec_TSY  },
    { "TXS",  0, {0,0,0,0,0,0x35}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_TXS  },
    { "TYS",  0, {0,0,0,0,0,0x1835}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_TYS  },
    { "WAI",  1, {0,0,0,0,0,0x3E}              , {-1,-1,-1,-1,-1,14},{0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,93} ,instExec_WAI  },
```

```
    { "XGDX", 0, {0,0,0,0,0,0x8F}              , {-1,-1,-1,-1,-1,3}, {0,0,0,0,0,1},
{0xff,0xff,0xff,0xff,0xff,10} ,instExec_XGDX },
    { "XGDY", 0, {0,0,0,0,0,0x188F}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_XGDY },
    { "NMINTERRUPT", 0, {0x12,0x34,0x56,0x78,0x9a,0xab}            , {-1,-1,-1,-1,-1,4}, {0,0,0,0,0,2},
{0xff,0xff,0xff,0xff,0xff,80} ,instExec_XGDY },
};

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------

const MachineCycleType CycleTable[]={
{20,23,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},  // /00/  ASLA (INH), ASLB (INH),
        //ASRA (INH), ASRB (INH), CBA (INH), CLC (INH), CLI (INH), CLRA (INH),
        //CLRB (INH), CLV (INH), COMA (INH), COMB (INH), DAA (INH), DECA (INH),
        //DECB (INH), INCA (INH), INCB (INH), LSLA (INH), LSLB (INH), LSRA (INH),
        //LSRB (INH), NEGA (INH), NEGB (INH), NOP (INH), ROLA (INH), ROLB (INH),
        //RORA (INH), RORB (INH), SBA (INH), SEC (INH), SEI (INH), SEV (INH),
        //STOP (INH), TAB (INH), TAP (INH), TBA (INH), TPA (INH), TSTA (INH),
        //TSTB (INH), ABA (INH)
{20,23,63,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /01/  DES (INH), INS
(INH),TSX (INH)
{20,23,63,70,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /02/  PULA (INH)
{20,23,63,71,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /03/  PULB (INH)
{20,23,63,72,73,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /04/  PULX (INH)
{20,23,63,76,77,78,79,80,81,82,83,84,0xfc,0xfd,0xfe,0xff},  // /05/  RTI (INH)
{20,23,63,85,86,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /06/  RTS (INH)
{20,23,64,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /07/  PSHA (INH)
{20,23,65,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /08/  PSHB (INH)
{20,23,66,67,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /09/  PSHX (INH)
{20,23,8,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /10/  ASLD (INH), DEX (INH),
        //FDIV (INH), IDIV (INH), INX (INH), LSLD (INH), LSRD (INH),
        //MUL (INH), TXS  (INH), XGDX (INH), ABX (INH)
{20,24,1,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /11/  ANDA (DIR), ANDB (DIR),
        //BITA (DIR), BITB (DIR), CMPA (DIR), CMPB (DIR), EORA (DIR),
        //EORB (DIR), LDAA (DIR), LDAB (DIR), ORAA (DIR), ORAB (DIR),
        //SBCA (DIR), SBCB (DIR), SUBA (DIR), SUBB (DIR), ADCA (DIR),
        //ADDA (DIR), ADCB (DIR), ADDB (DIR)
{20,24,1,3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /12/  LDD (DIR), LDS (DIR),
LDX (DIR)
{20,24,1,3,8,0xf5, 0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /13/  ADDD (DIR), CPX (DIR),
SUBD (DIR)
{20,24,1,38,44,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},            // /14/  BRCLR (DIR), BRSET
(DIR)
{20,24,1,38,8,2,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},             // /15/  BSET (DIR)
{20,24,1,38,8,6,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},             // /16/  BCLR (DIR)
{20,24,1,61,62,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /17/  JSR (DIR)
{20,24,4,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /18/  STAA (DIR)
{20,24,4,7,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /19/  STD (DIR)
{20,24,5,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /20/  STAB (DIR)
{20,24,87,90,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /21/  STS (DIR)
{20,24,88,91,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /22/  STX (DIR)
{20,24,89,92,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /23/  STY (DIR)
{20,25,8,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /24/  JMP (IND,X)
{20,25,8,100,103,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /25/  STX (IND,X)
{20,25,8,101,104,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /26/  STY (IND,X)
{20,25,8,47,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /27/  ANDA (INDX), ANDB
(INDX),
        //BITA (IND,X), BITB (IND,X), CMPA (IND,X), CMPB (IND,X), EORA (IND,X),
        //EORB (IND,X), LDAA (IND,X), LDAB (IND,X), ORAA (IND,X), ORAB (IND,X),
        //SBCA (IND,X), SBCB (IND,X), SUBA (IND,X), SUBB (IND,X), ADCA (INDX),
        //ADCB (INDX), ADDA (INDX), ADDB (INDX)
{20,25,8,47,38,44,8,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /28/  BRCLR (IND,X), BRSET
(IND,X)
{20,25,8,47,38,8,46,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /29/  BCLR (IND,X), BSET
(IND,X)
{20,25,8,47,48,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /30/  LDD (IND,X), LDS
(IND,X), LDX (IND,X)
{20,25,8,47,48,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},            // /31/  CPX (IND,X), ADDD
(INDX), SUBD (IND,X)
{20,25,8,47,61,62,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /32/  JSR (IND,X)
{20,25,8,47,8,46,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},            // /33/  COM (IND,X), LSL
(IND,X),
        //LSR (IND,X), NEG (IND,X), ROL (IND,X), ROR (IND,X), ASL (INDX),
        //ASR (INDX), DEC (IND,X), INC (IND,X)
{20,25,8,47,8,49,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},            // /34/  CLR (IND,X)
{20,25,8,56,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /35/  STAA (IND,X)
{20,25,8,56,50,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /36/  STD (IND,X)
{20,25,8,58,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /37/  STAB (IND,X)
{20,25,8,99,102,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /38/  STS (IND,X)
{20,26,34,13,15,0xf5, 0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /39/  CPX (EXT)
{20,26,34,13,15,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /40/  SUBD (EXT)
{20,26,37,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /41/  JMP (EXT)
{20,26,37,10,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /42/  STAB (EXT)
{20,26,37,11,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /43/  STAA (EXT)
{20,26,37,11,17,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /44/  STD (EXT)
{20,26,37,13,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /45/  ANDA (EXT), ANDB (EXT),
        //BITA (EXT), BITB (EXT), CMPA (EXT), CMPB (EXT), EORA (EXT),
        //EORB (EXT), LDAA (EXT), LDAB (EXT), ORAA (EXT), ORAB (EXT),
        //SBCA (EXT), SBCB (EXT), SUBA (EXT), SUBB (EXT), ADCA (EXT),
        //ADCB (EXT), ADDA (EXT), ADDB (EXT)
{20,26,37,13,15,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /46/  LDD (EXT), LDS (EXT),
LDX (EXT)
{20,26,37,13,15,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /47/  ADDD (EXT)
{20,26,37,13,61,62,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},          // /48/  JSR (EXT)
{20,26,37,13,8,12,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /49/  CLR (EXT)
{20,26,37,13,8,14,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /50/  ASL (EXT), ASR (EXT),
COM (EXT),
```

```
        //DEC (EXT), INC (EXT), LSL (EXT), LSR (EXT), NEG (EXT), ROL (EXT), ROR (EXT)
{20,26,37,13,8,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},             // /51/  TST (EXT)
{20,26,37,8,54,8,8,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},                // /52/  TST (IND,Y)
{20,26,37,93,96,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /53/  STS (EXT)
{20,26,37,94,97,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /54/  STX (EXT)
{20,26,37,95,98,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /55/  STY (EXT)
{20,26,8,47,8,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /56/  TST (IND,X)
{21,27,34,111,13,15,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /57/  LDY (EXT)
{21,27,34,111,13,15,8,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},            // /58/  CPD (EXT), CPY (EXT)
{21,27,35,8,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /59/  JMP (IND,Y)
{21,27,35,8,105,108,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /60/  STS (IND,Y)
{21,27,35,8,106,109,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /61/  STX (IND,Y)
{21,27,35,8,107,110,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /62/  STY (IND,Y)
{21,27,35,8,47,48,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /63/  LDY (IND,X)
{21,27,35,8,47,48,8,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /64/  CPD (IND,X), CPY
(IND,X)
{21,27,35,8,54,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /65/  ANDA (INDY), ANDB
(INDY),
        //BITA (IND,Y), BITB (IND,Y), CMPA (IND,Y), CMPB (IND,Y), EORA (IND,Y),
        //EORB (IND,Y), LDAA (IND,Y), LDAB (IND,Y), ORAA (IND,Y), ORAB (IND,Y),
        //SBCA (IND,Y), SBCB (IND,Y), SUBA (IND,Y), SUBB (IND,Y), ADCA (INDY),
        //ADCB (INDY), ADDA (INDY), ADDB (INDY)
{21,27,35,8,54,43,45,8,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},                // /66/  BRCLR (IND,Y), BRSET
(IND,Y)
{21,27,35,8,54,43,8,53,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},                // /67/  BCLR (IND,Y), BSET
(IND,Y)
{21,27,35,8,54,55,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /68/  LDD (IND,Y), LDS
(IND,Y), LDX (IND,Y), LDY (IND,Y)
{21,27,35,8,54,55,8,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /69/  ADDD (INDY), CPX
(IND,Y), SUBD (IND,Y)
{21,27,35,8,54,55,8,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},                   // /70/  CPD (IND,Y), CPY
(IND,Y)
{21,27,35,8,54,61,62,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},             // /71/  JSR (IND,Y)
{21,27,35,8,54,8,52,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /72/  CLR (IND,Y)
{21,27,35,8,54,8,53,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},              // /73/  ASL (INDY), ASR (INDY),
        //COM (IND,Y), DEC (IND,Y), INC (IND,Y), LSL (IND,Y), LSR (IND,Y),
        //NEG (IND,Y), ROL (IND,Y), ROR (IND,Y)
{21,27,35,8,57,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /74/  STAA (IND,Y)
{21,27,35,8,57,51,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /75/  STD (IND,Y)
{21,27,35,8,59,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /76/  STAB (IND,Y)
{21,27,39,63,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /77/  TSY  (INH)
{21,27,39,63,74,75,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},          // /78/  PULY (INH)
{21,27,39,68,69,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},        // /79/  PSHY (INH)
{21,27,39,8,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /80/  DEY (INH), INY (INH),
TYS (INH),
        //XGDY (INH), ABY (INH)
{21,27,40,42,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},      // /81/  LDY (IMM)
{21,27,40,42,8,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},         // /82/  CPD (IMM), CPY (IMM)
{21,27,41,1,3,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},          // /83/  LDY (DIR)
{21,27,41,1,3,8,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},             // /84/  CPD (DIR), CPY (DIR)
{20,28,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},  // /85/  ANDA (IMM), ANDB (IMM),
        //BITA (IMM), BITB (IMM), EORA (IMM), EORB (IMM), LDAA (IMM), LDAB (IMM),
        //ORAA (IMM), ORAB (IMM), SBCA (IMM), SBCB (IMM), SUBA (IMM), SUBB (IMM),
        //ADCA (IMM), ADCB (IMM), ADDB (IMM), ADDA (IMM)
{20,28,36,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /86/  LDD (IMM)
{20,29,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},  // /87/  CMPA (IMM), CMPB (IMM)
{20,29,36,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},    // /88/  LDS (IMM), LDX (IMM)
{20,29,36,8,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /89/  CPX (IMM), ADDD (IMM)
{20,29,36,8,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},       // /90/  SUBD (IMM)
{20,30,8,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},     // /91/  BCC (INH), BCS (REL),
BEQ (REL),
        //BGE (REL), BGT (REL), BHI (REL), BHS (REL), BLE (REL), BLO (REL),
        //BLS (REL), BLT (REL), BMI (REL), BNE (REL), BPL (REL), BRA (REL),
        //BRN (REL), BVC (REL), BVS (REL)
{20,30,8,60,61,62,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff},           // /92/  BSR (REL)
{20,23,61,62,112,113,114,115,116,117,118,119,120,121,0xfe,0xff},                // /93/  SWI (INH),  WAI (INH)
{20,23,39,122,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff}      // /94/  TEST (INH)

};
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
const MachineCycleDefStruct MDefCycleTable[] = {
{         0         ,         "Bos"          ,     "00dd"    ,         "1"       },
{         1         ,         "00dd"         ,     "00dd"    ,         "1"       },
{         2         ,         "00dd"         ,     "Result"  ,         "0"       },
{         3         ,         "00dd+1"  ,          "00dd+1"  ,         "1"       },
{         4         ,         "00dd"         ,     "(A)"     ,         "1"       },
{         5         ,         "00dd"         ,     "(B)"     ,         "1"       },
{         6         ,         "00dd+1"  ,          "(A)"     ,         "1"       },
{         7         ,         "00dd+1"  ,          "(B)"     ,         "1"       },
{         8         ,         "0xFFFF"  ,          "-"       ,         "1"       },
{         9         ,         "Bos"     ,          "-"       ,         "1"       },
{         10        ,         "hhll"         ,     "(B)"     ,         "1"       },
{         11        ,         "hhll"         ,     "(A)"     ,         "1"       },
{         12        ,         "hhll"         ,     "0"       ,         "0"       },
{         13        ,         "hhll"         ,     "hhll"    ,         "1"       },
{         14        ,         "hhll"         ,     "Result"  ,         "0"       },
{         15        ,         "hhll+1"  ,          "hhll+1"  ,         "1"       },
{         16        ,         "hhll+1"  ,          "(A)"     ,         "1"       },
{         17        ,         "hhll+1"  ,          "(B)"     ,         "1"       },
{         18        ,         "Bos"          ,     "00dd"    ,         "1"       },
{         19        ,         "Bos"          ,     "00dd"    ,         "1"       },
{         20        ,         "OP"           ,     "Inst Code L"       ,         "1"       },
{         21        ,         "OP"           ,     "Inst Code H"       ,         "1"       },
{         22        ,         "Bos"          ,     "00dd"    ,         "1"       },
{         23        ,         "OP+1"         ,     "-"       ,         "1"       },
{         24        ,         "OP+1"         ,     "dd (direct data)"  ,         "1"       },
{         25        ,         "OP+1"         ,     "ff"      ,         "1"       },
```

```
{       26      ,       "OP+1"      ,   "hh"        ,       "1"     },
{       27      ,       "OP+1"      ,   "Inst Code H"       ,       "1"     },
{       28      ,       "OP+1"      ,   "ii (immediate data)"       ,       "1"     },
{       29      ,       "OP+1"      ,   "jj"        ,       "1"     },
{       30      ,       "OP+1"      ,   "rr"        ,       "1"     },
{       31      ,       "Bos"       ,   "00dd"      ,       "1"     },
{       32  ,   "Bos"       ,   "00dd"      ,       "1"     },
{       33      ,       "Bos"       ,   "00dd"      ,       "1"     },
{       34      ,       "OP+2"      ,   "hh"        ,       "1"     },
{       35      ,       "OP+2"      ,   "ff"        ,       "1"     },
{       36      ,       "OP+2"      ,   "kk"        ,       "1"     },
{       37      ,       "OP+2"      ,   "ll"        ,       "1"     },
{       38      ,       "OP+2"      ,   "MM"        ,       "1"     },
{       39      ,       "OP+2"      ,   "-"         ,       "1"     },
{       40      ,       "OP+2"      ,   "jj"        ,       "1"     },
{       41      ,       "OP+2"      ,   "dd"        ,       "1"     },
{       42      ,       "OP+3"      ,   "kk"        ,       "1"     },
{       43      ,       "OP+3"      ,   "mm"        ,       "1"     },
{       44      ,       "OP+3"      ,   "rr"        ,       "1"     },
{       45      ,       "OP+4"      ,   "rr"        ,       "1"     },
{       46      ,       "X+ff"      ,   "Result"    ,       "0"     },
{       47      ,       "X+ff"      ,   "X+ff"      ,       "1"     },
{       48      ,       "X+ff+1"    ,   "X+ff+1"    ,       "1"     },
{       49      ,       "X+ff"      ,   "0"         ,       "0"     },
{       50      ,       "X+ff+1"    ,   "(B)"       ,       "1"     },
{       51      ,       "Y+ff+1"    ,   "(B)"       ,       "1"     },
{       52      ,       "Y+ff"      ,   "0"         ,       "0"     },
{       53      ,       "Y+ff"      ,   "Result"    ,       "0"     },
{       54      ,       "Y+ff"      ,   "Y+ff"      ,       "1"     },
{       55      ,       "Y+ff+1"    ,   "Y+ff+1"    ,       "1"     },
{       56      ,       "X+ff"      ,   "(A)"       ,       "1"     },
{       57      ,       "Y+ff"      ,   "(A)"       ,       "1"     },
{       58      ,       "X+ff"      ,   "(B)"       ,       "1"     },
{       59      ,       "Y+ff"      ,   "(B)"       ,       "1"     },
{       60      ,       "Sub"       ,   "Nxt op"    ,       "1"     },
{       61      ,       "SP"        ,   "Rtn lo"    ,       "0"     },
{       62      ,       "SP-1"      ,   "Rtn hi"    ,       "0"     },
{       63      ,       "SP"    ,       "-"         ,       "1"     },
{       64      ,       "SP"    ,       "(A)"       ,       "0"     },
{       65      ,       "SP"    ,       "(B)"       ,       "0"     },
{       66      ,       "SP"    ,       "(IXL)"     ,       "0"     },
{       67      ,       "SP"    ,       "(IXH)"     ,       "0"     },
{       68      ,       "SP"    ,       "(IYL)"     ,       "0"     },
{       69      ,       "SP"    ,       "(IYH)"     ,       "0"     },
{       70      ,       "SP+1"  ,       "get A"     ,       "1"     },
{       71      ,       "SP+1"  ,       "get B"     ,       "1"     },
{       72      ,       "SP+1"  ,       "Get IXH"   ,       "1"     },
{       73      ,       "SP+2"  ,       "Get IXL"   ,       "1"     },
{       74      ,       "SP+1"  ,       "Get IYH"   ,       "1"     },
{       75      ,       "SP+2"  ,       "Get IYL"   ,       "1"     },
{       76      ,       "SP+1"  ,       "Get CC"    ,       "1"     },
{       77      ,       "SP+2"  ,       "Get B"     ,       "1"     },
{       78      ,       "SP+3"  ,       "Get A"     ,       "1"     },
{       79      ,       "SP+4"      ,   "Get IXH"   ,       "1"     },
{       80      ,       "SP+5"      ,   "Get IXL"   ,       "1"     },
{       81      ,       "SP+6"      ,   "Get IYH"   ,       "1"     },
{       82      ,       "SP+7"      ,   "Get IYL"   ,       "1"     },
{       83      ,       "SP+8"      ,   "Rtn hi = Get PCH"  ,       "1"     },
{       84      ,       "SP+9"      ,   "Rtn lo = Get PCL"  ,       "1"     },
{       85      ,       "SP+1"      ,   "Rtn hi"    ,       "1"     },
{       86      ,       "SP+2"      ,   "Rtn lo"    ,       "1"     },
{       87      ,       "00dd"      ,   "(SPH)"     ,       "0"     },
{       88      ,       "00dd"      ,   "(IXH)"     ,       "0"     },
{       89      ,       "00dd"      ,   "(IYH)"     ,       "0"     },
{       90      ,       "00dd+1"    ,   "(SPL)"     ,       "1"     },
{       91      ,       "00dd+1"    ,   "(IXL)"     ,       "0"     },
{       92      ,       "00dd+1"    ,   "(IYL)"     ,       "0"     },
{       93      ,       "hhll"      ,   "(SPH)"     ,       "0"     },
{       94      ,       "hhll"      ,   "(IXH)"     ,       "0"     },
{       95      ,       "hhll"      ,   "(IYH)"     ,       "0"     },
{       96      ,       "Hhll+1"    ,   "(SPL)"     ,       "0"     },
{       97      ,       "Hhll+1"    ,   "(IXL)"     ,       "0"     },
{       98      ,       "Hhll+1"    ,   "(IYL)"     ,       "0"     },
{       99      ,       "X+ff"      ,   "(SPH)"     ,       "0"     },
{       100     ,       "X+ff"      ,   "(IXH)"     ,       "0"     },
{       101     ,       "X+ff"      ,   "(IYH)"     ,       "0"     },
{       102     ,       "X+ff+1"    ,   "(SPL)"     ,       "0"     },
{       103     ,       "X+ff+1"    ,   "(IXL)"     ,       "0"     },
{       104     ,       "X+ff+1"    ,   "(IYL)"     ,       "0"     },
{       105     ,       "Y+ff"      ,   "(SPH)"     ,       "0"     },
{       106     ,       "Y+ff"      ,   "(IXH)"     ,       "0"     },
{       107     ,       "Y+ff"      ,   "(IYH)"     ,       "0"     },
{       108     ,       "Y+ff+1"    ,   "(SPL)"     ,       "0"     },
{       109     ,       "Y+ff+1"    ,   "(IXL)"     ,       "0"     },
{       110     ,       "Y+ff+1"    ,   "(IYL)"     ,       "0"     },
{       111     ,       "OP+3"      ,   "ll"        ,       "1"     },
{       112     ,       "SP-2"      ,   "(IYL)"     ,       "0"     },
{       113     ,       "SP-3"      ,   "(IYH)"     ,       "0"     },
{       114     ,       "SP-4"      ,   "(IXL)"     ,       "0"     },
{       115     ,       "SP-5"      ,   "(IXH)"     ,       "0"     },
{       116     ,       "SP-6"      ,   "(A)"       ,       "0"     },
{       117     ,       "SP-7"      ,   "(B)"       ,       "0"     },
{       118     ,       "SP-8"      ,   "(CCR)"     ,       "0"     },
{       119     ,       "SP-8"      ,   "(CCR)"     ,       "1"     },
{       120     ,       "Vec hi"    ,   "Svc hi"    ,       "1"     },
{       121     ,       "Vec lo"    ,   "Svc lo"    ,       "1"     },
{       122     ,       "OP+3"      ,   "-"         ,       "1"     }
};
```