MC6811 MICROCONTROLLER SIMULATION TOOLKIT

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

BY

TOLGA TAŞKIN

IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

NOVEMBER 2005

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,EEE)	_____
Prof. Dr. Hasan GÜRAN	(METU,EEE)	_____
Asst. Prof. Dr. Cüneyt BAZLAMAÇCI	(METU,EEE)	_____
Dr. Ece SCHMIDT	(METU,EEE)	_____
M.Sc. Recep Çağrı YÜZBAŞIOĞLU	(HAVELSAN)	_____

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.

Name, Last name : Tolga TAŞKIN

Signature :

ABSTRACT

MC6811 MICROCONTROLLER SIMULATION TOOLKIT

TAŞKIN, Tolga

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Hasan GÜRAN

NOVEMBER 2005, 119 pages

The goal of this thesis study is to develop a simulator toolkit for Motorola's 8-bit microcontroller MC6811. The toolkit contains a cross-assembler to obtain object code from the source code and a simulator to run the object code. Written document of this thesis study describes the properties of the MC6811 microcontroller and its assembly language. In addition, the document describes the cross-assembler and simulator parts of the toolkit with details. In the cross-assembler part of the toolkit, parsing of the source code and processing of the parsed information is studied. The simulator part studies the execution of the object code generated by the cross-assembler. The execution of each instruction and main functions of the microcontroller can be observed from a Graphical User Interface (GUI). The Central Processing Unit (CPU), the busses, ports and interrupts of the microcontroller are included into the GUI. C++ programming language is used to develop and to implement the toolkit.

Keywords: 6811, cross-assembler, simulator, microcontroller.

ÖZ

MC6811 MİKROKONTROLCÜSÜ SİMÜLASYON ARAÇLARI

TAŞKIN, Tolga
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Hasan GÜRAN

KASIM 2005, 119 sayfa

Bu tez çalışmasının amacı Motorola MC6811 8-bit mikrokontrolcüsü için simülasyon araçları geliştirmektir. Bu araçlar kaynak kodundan nesne kodu elde etmeye yarıyan bir derleyici ve derlenmiş kaynak kodunu çalıştırmaya yarayan bir simülatörden oluşmaktadır. Bu tezin yazılı dokümanı MC6811 mikro kontrolcüsünün özelliklerini tanımlar. Aynı zamanda bu doküman derleyici ve simülatörün detaylarını içerir. Derleyici tarafında kaynak dosyanın okunması ve işlenmesi anlatılmıştır. Simülatör kısmında nesne kodunun çalıştırılması işlenmiştir. Grafik bir arayüz sayesinde mikrokontrolcünün bütün işlevleri, CPU, veri ve adres yolları, portları görülebilmektedir. Simülatör araçlarının geliştirilmesi ve yazımında C++ programlama dili kullanılmıştır.

Anahtar Kelimeler: 6811, çapraz derleyici, simülatör, mikrokontrolcü.

To My Family

To My Friends

and

To Myself

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to Mustafa Kemal ATATÜRK who is the founder of the Turkish Republic, gave freedom to my country, and gave me this chance to complete my thesis study under my own flag.

I also wish to express my gratitude to my supervisor Prof. Dr. Hasan GÜRAN for his guidance, advice, criticism, encouragement and insight through the research.

I would also like to thank especially to:

My family, İbrahim TAŞKIN my father, Hamiyet TAŞKIN my mother and Buğra TAŞKIN my little brother, for their understanding and support.

To my friends, brothers and sisters, for standing with me in my hardest and darkest days.

Finally to myself, for working hard enough and believing so much to complete this study.

TABLE OF CONTENTS

PLAGIARISM ..... **III**

ABSTRACT ..... **IV**

ÖZ ..... **V**

ACKNOWLEDGEMENTS ..... **VII**

LIST OF TABLES ..... **XIII**

LIST OF FIGURES ..... **XIV**

CHAPTERS

1. INTRODUCTION **1**

2. OVERVIEW OF MC6811 ..... **2**

2.1. HARDWARE 3

2.1.1. I/O INTEGRATED CIRCUITS AND PARALLEL I/O PORTS *4*

2.1.2. INTERRUPT SYSTEM *5*

2.1.2.1. SELECTING INTERRUPT VECTORS 6

2.1.2.2. RETURN FROM INTERRUPT 6

2.1.2.3. NON-MASKABLE INTERRUPTS 7

2.1.2.4. MASKABLE INTERRUPTS 8

2.1.2.5. THE INTERRUPT REQUEST 9

2.1.3. REAL TIME CLOCK *9*

2.1.4. THE PROGRAMMABLE TIMER *10*

2.1.5. PULSE ACCUMULATOR *10*
2.1.6. SERIAL COMMUNICATIONS INTERFACE *11*
2.1.7. ANALOG TO DIGITAL CONVERTER *11*
2.2. THE MC6811 COMPUTER OPERATION 12
2.2.1. PROGRAMMING MODEL *12*
2.2.2. INTERNAL COMPUTER OPERATIONS *14*
2.3. ADDRESSING MODES 16
2.4. INSTRUCTIONS 16

3. THE CROSS ASSEMBLER 18

3.1. MOTOROLA ASSEMBLY LANGUAGE 18
3.1.1. LABEL FIELD *19*
3.1.2. OPERATION FIELD *19*
3.1.2.1. OPCODES 20
3.1.2.2. DIRECTIVES 20
3.1.3. OPERAND FIELD *22*
3.1.3.1. OPERATORS 22
3.1.3.2. CONSTANTS 23
3.1.4. COMMENT FIELD *23*
3.2. ASSEMBLER PROCESS 24
3.2.1. FIRST PASS *26*
3.2.2. SECOND PASS *26*
3.2.3. INVOCATION OF THE ASSEMBLER *36*

4. THE SIMULATOR 37

4.1. SCOPE AND DESIGN OF THE SIMULATOR 38
4.1.1. INCLUDED AND EXCLUDED PARTS AND PROPERTIES OF MC6811 ..... *39*
4.1.1.1. CENTRAL PROCESSING UNIT 39
4.1.1.2. PORTS AND RELATED PROPERTIES 41
4.1.1.3. INTERRUPTS AND MEMORY 43

4.1.2. DESIGN OF SIM68TT11 ..... *44*
4.2. GRAPHICAL USER INTERFACE 46
4.2.1. MAIN WINDOW *46*
4.2.1.1. MICROCONTROLLER 46
4.2.1.2. FLOW CHART 58
4.2.2. MENU BAR *59*
4.2.2.1. LOAD BUTTON 59
4.2.2.2. OPTIONS 61
4.2.2.3. RUN, FWD, BACK and STOP 63
4.3. INVOCATION OF THE SIMULATOR 64
5. CONCLUSIONS 66
REFERENCES 68
APPENDICES
APPENDIX A: DATA TYPES AND SUBROUTINES OF SIM68TT11 69
A.1. DATA TYPES 69
A.1.1. SregistreS *69*
A.1.2. InxRegs *69*
A.1.3. AddressModes *70*
A.1.4. whbranch *70*
A.1.5. CCRclear *70*
A.1.6. shift *70*
A.1.7. arith *71*
A.1.8. logic *71*
A.2. SUBROUTINES 71
A.2.1. searchInterrupt *71*
A.2.2. exec_SABA *72*
A.2.3. execLDA_AB *72*
A.2.4. execSTA_AB *72*

A.2.5. exec_ABXY *72*
A.2.6. exec_ADCAB *73*
A.2.7. exec_ADDD *73*
A.2.8. exec_logic *73*
A.2.9. exec_ASLRD *73*
A.2.10. exec_BRCH *74*
A.2.11. exec_BIT *74*
A.2.12. exec_CMP *74*
A.2.13. exec_CBA *74*
A.2.14. exec_CLCIV *74*
A.2.15. execCLEAR *75*
A.2.16. exec_ARITH *75*
A.2.17. exec_LOAD *75*
A.2.18. exec_STORE *76*
A.2.19. exec_PUSH *76*
A.2.20. exec_PULL *76*
A.2.21. exec_RTI *76*
A.2.22. exec_JUMP *76*
A.2.23. exec_TABvC *77*
A.2.24. exec_TBvCA *77*
A.2.25. exec_CPD *77*
A.2.26. exec_arithSXY *77*
A.2.27. exec_JSR *77*
A.2.28. exec_RSR *78*
A.2.29. exec_SWI *78*
A.2.30. exec_TSXYS *78*
A.2.31. exec_XGDXY *78*
A.2.32. illegalOPCODE *79*
A.2.33. exec_DAA *79*
A.2.34. exec_NOP *79*

A.2.35. exec_BRCS *79*
A.2.36. exec_BCLSE *79*
A.2.37. exec_WAI *80*
A.2.38. exec_FIDIV *80*
A.2.39. exec_MUL *80*
APPENDIX B: VERIFICATION OF ASM68TT11 AND SIM68TT11 81
B.1. VERIFICATION OF ASM68TT1 81
B.2. VERIFICATION OF SIM68TT1 89
APPENDIX C: USER MANUALS 100
C.1. MANUAL OF ASM68TT11 100
C.2. MANUAL OF SIM68TT11 101
APPENDIX D: PROGRAMS 104

LIST OF TABLES

TABLE

1 Hardware Control Registers of SIM68TT11 44

LIST OF FIGURES

FIGURES

1 Block Diagram of the MC6811 MCU 3
2 TMSK2 and TFLG2 Registers 9
3 Pulse Accumulator 10
4 ADCTL Register 12
5 Programming Model of MC6811 13
6 Main flowchart 29
7 Parse flowchart 30
8 Instruction Mnemonic flowchart 1 31
9 Instruction Mnemonic flowchart 2 32
10 Instruction Mnemonic flowchart 3 33
11 Instruction Mnemonic flowchart 4 34
12 Instruction Mnemonic flowchart 5 35
13 Block Diagram of Microprocessor 40
14 Simulator Flowchart Part 1 47
15 Simulator Flowchart Part 2 48
16 SIM68TT11 Main Window 49
17 Registers and Internal Buses 51
18 Control Unit 52
19 Bus Interface Unit 53
20 Operation Information Box 53
21 Interrupt Control 54
22 Input Output Ports 55
23 Memory 1 56

24 Memory 2 57
25 Memory 3 58
26 Flowchart 59
27 Load Button 60
28 Load Menu 60
29 Options 61
30 Run Modes 62
31 Step Type 62
32 Continuous Mode Speed 63
33 Memory Mode 63
34 Info Box 65
35 Help Box 65
36 SIM68TT11 MENU 103

CHAPTER 1

INTRODUCTION

The aim of this thesis study is to develop a simulator toolkit for Motorola's 8-bit microcontroller MC6811. The toolkit contains a cross-assembler to obtain object code from the source code and a simulator to run the object code.

The HCMOS MC6811 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. The HCMOS technology used on the MC6811 combines smaller size and higher speeds with the low power and high noise immunity of CMOS.

The cross-assembler developed in this thesis is used to convert the source code, written according to the rules of Motorola Assembly Language, to object that can be understood and run by simulator tool. Simulator tool runs the object code either from the cross-assembler developed here or from any assembler produces object code in Motorola S-19 format.

The following chapters include the explanation and description of the microcontroller, assembler language, assemblers, loaders and cross-assembler developed within the toolkit. In addition, the developed simulator is introduced and explained through the chapters.

CHAPTER 2

OVERVIEW OF MC6811

In this chapter, general information about MC6811 is given to prepare reader for later chapters. The MC6811 is a single-chip microcomputer or microcontroller because it contains memory and input/output hardware. 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.

Figure 1 is a block diagram of the MC6811 MCU. This diagram shows the major subsystems and how they relate to the pins of the MCU. Details of the MC6811 are given in [1], [7], [8], [12], [13].

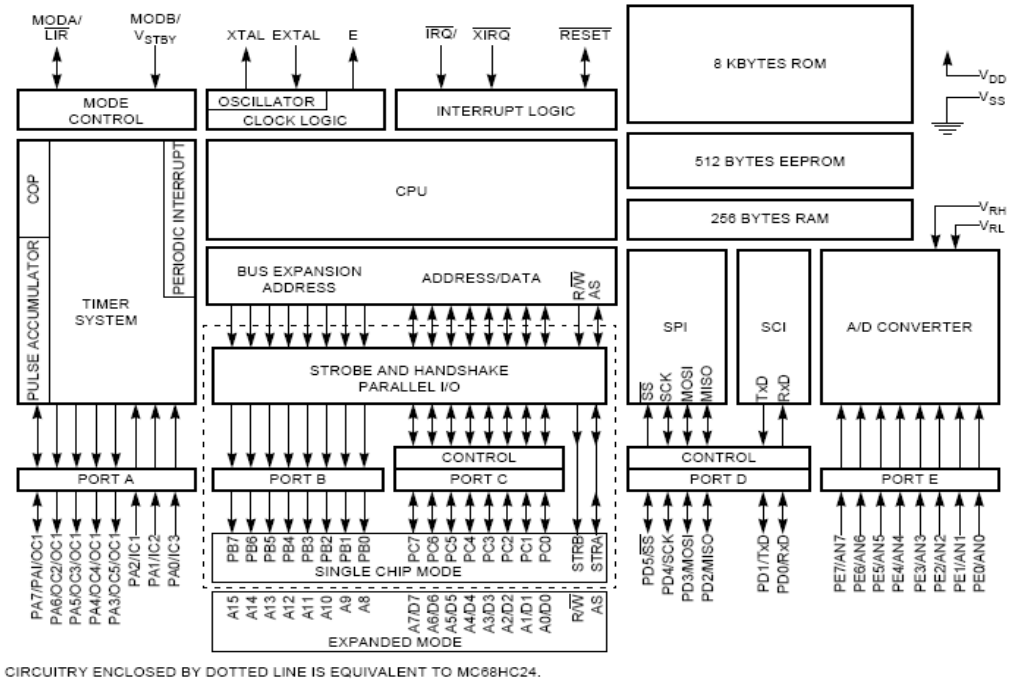

Figure 1 Block Diagram of the MC6811 MCU

2.1. HARDWARE

The MC6811 integrated circuit contains many input/output devices and several types of memory as mentioned previously. The I/O devices are frequently used control applications. This chapter discusses the hardware devices inside MC6811 chip. The details given here are based on the E9 model of the chip.

The MC6811 microprocessor executes a total of 307_{10} instruction codes. The chip has 12_{10} kilobytes of masked ROM, 512_{10} bytes of static RAM, 512_{10} EEPROM and has a control register that prevents erroneous changing of the EEPROM. The SCI asynchronous serial device allows communication to remote serial I/O devices. The SPI interface allows both control of other ICs in the microcomputer and communications to another nearby circuit board. Several parallel I/O ports, including some programmable ports, allow communication to external I/O devices. The timer device allows measurement of timing intervals and timed control of outputs. Eight channel A/D converter measures DC voltages. Including analog hardware in the chip allows sensors of many kinds to interface very easily. The pulse accumulator can both count pulses and measure the duration of pulses from I/O devices outside the computer. Computer operating properly, clock monitor and illegal instruction devices forces a safe hardware response when certain computer hardware or software has failed. Further information about the hardware of MC6811 can be found in [1], [7], [8], [12], [13].

2.1.1. I/O INTEGRATED CIRCUITS AND PARALLEL I/O PORTS

The ports are collections of signal wires or pins that the MC6811 uses to communicate with devices outside the chip. The port names are arbitrary because each port has several functions. The ports have different functions depending on the operating mode of the MC6811.

Software can program the MC6811 hardware to connect ports A, D, and E to input/output devices, such as an analog-to-digital converter, inside the MC6811. All the I/O port pins have either digital input or digital output capability. For example, the ports B and C used in examples in this book have an additional function.
A hardware reset initializes the input/output hardware in the MC6811. The reset also puts the chip into the correct operating mode. A hardware reset disables all input/output devices inside the MC6811. The effect is to program all available I/O

ports as parallel I/O ports. The reset clears all input and output registers, and the data direction register bits making any programmable port bits inputs. All data direction register bits make an input when they are 0 and make an output when they are 1.

2.1.2. INTERRUPT SYSTEM

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 interrupt system in the MC6811 supports several types of interrupts from many input/output devices. The MC6811 also has a hardware reset function that is similar to an interrupt. Most interrupts in the MC6811 have separate interrupt vectors. Each of them sends control to individual interrupt service routines. The interrupt vectors contain the addresses of the interrupt service routines for each device. Table 1 lists the many interrupt vectors in MC6811.

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. 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.
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. Details of the interrupt system are given in [1], [7], [8], [12], [13].

2.1.2.1. SELECTING INTERRUPT VECTORS

After the CCR has been stacked, the CPU evaluates all pending interrupt requests to determine which source has the highest priority. 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. Software interrupt (SWI) is actually an instruction and has the highest priority because, once the SWI opcode is fetched, no other interrupt can be honored until the SWI vector has been fetched.

2.1.2.2. 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. The RTI instruction pulls 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.

2.1.2.3. 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.

2.1.2.3.1. 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 MC6811 controls Non-maskable interrupts 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.

2.1.2.3.2. ILLEGAL OPCODE FETCH

Since not all possible opcodes or opcode sequences are defined, an illegal opcode detection circuit has been included. When an illegal opcode is detected, an interrupt is requested to the illegal opcode vector. The illegal opcode vector should never be left uninitialized. The stack pointer should be re-initialized as a result of an illegal opcode interrupt so repeated execution of illegal opcodes does not cause stack overruns. If the illegal opcode vector were left uninitialized, it could point to a

memory location that contained an illegal opcode. In such a case, there would be an infinite loop of repeated illegal opcodes and an infinite stack overflow, which would cause the register contents to be stored to all memory addresses in a very short time.

2.1.2.3.3. 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. The global interrupt mask bits (X or I) in the CCR do not inhibit SWI. 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.

2.1.2.4. MASKABLE INTERRUPTS

The remaining twenty interrupt sources in the MC6811 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 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.

2.1.2.5. THE INTERRUPT REQUEST

The IRQ interrupt is named from the phrase "interrupt request." The IRQ interrupt is the principal external interrupt line in the 68HC11. It has a single interrupt vector to send control to a single interrupt service routine. The maskable interrupt structure in the MC6811 can be extended to additional external interrupting sources through the IRQ input. The XIRQ pin provides for nonmaskable interrupts.

2.1.3. REAL TIME CLOCK

One way for a computer to track time is to use an I/O device that causes interrupts periodically. An oscillator sets a flag at the end of each period. The interrupt service routine then can count interrupts to track time.

The RTI function can be used to generate hardware interrupts at a fixed periodic rate. The real-time interrupt device causes an interrupt by setting a flag named RTIF for real-time interrupt flag. A program can examine this flag by testing bit 6 of the TFLG2 register at address 1025. Figure 2 shows TFLG2 register.

TMSK2 — Timer Interrupt Mask Register 2 **$1024**

	BIT 7	6	5	4	3	2	1	BIT 0
	TOI	RTII	PAOVI	PAII	0	0	PR1	PR0
RESET:	0	0	0	0	0	0	0	0

TFLG2 — Timer Interrupt Flag Register 2 **$1025**

	BIT 7	6	5	4	3	2	1	BIT 0
	TOF	RTIF	PAOVF	PAIF	0	0	0	0
RESET:	0	0	0	0	0	0	0	0

Figure 2 TMSK2 and TFLG2 Registers

2.1.4. THE PROGRAMMABLE TIMER

Control applications of computers usually require measurements of time. The performance of control systems is usually time-dependent. The MC6811 contains a hardware timer. The timer can measure time for both inputs and outputs. The timing schemes are different for inputs and outputs. The timer consists of three logical parts: a free-running counter, input-capture hardware, and output-compare hardware. Because many programmable options are available, this hardware is very complex. However, the fundamental principles of its operation are quite simple.

2.1.5. PULSE ACCUMULATOR

The pulse accumulator is an 8-bit counter/timer system that can be configured to operate in either of two basic modes. In the event counting mode, the 8-bit counter is clocked to increasing values at each active edge of the PAI pin. In the gated time accumulation mode, a free-running E divided by 64 clock subject to the PAI pin being active clocks the 8-bit counter. Figure 3 is a simplified block diagram of the pulse accumulator in each of these two possible modes.

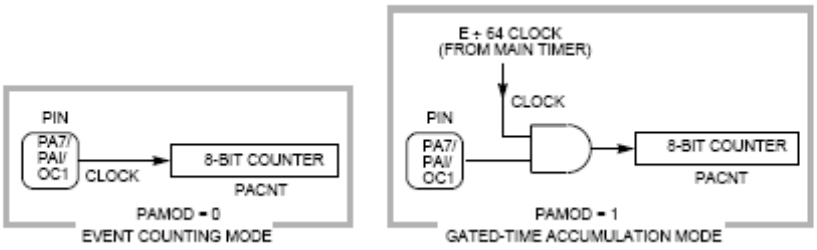

Figure 3 Pulse Accumulator

2.1.6. SERIAL COMMUNICATIONS INTERFACE

the serial communication interface, SCI, inside the MC6811 makes serial transmission and reception easy. The SCI device is very flexible, so it can adopt most applications. However, its sophistication means that the program must select many control options and control many data bits. Most programs for controlling the SCI use interrupt.

2.1.7. ANALOG TO DIGITAL CONVERTER

The analog-to-digital or A/D converter in the MC6811 makes 8-bit unsigned numbers representing external DC voltages. The A/D converter, by using an 8-channel multiplexer, can read voltages from eight different pins on the MC68HC 11 package. Port E is the collection of the eight input pins for the A/D converter. An A/D converter normally reads signals from analog sensors. Typical sensors measure temperature, pressure, or position.

The MC6811 chip contains an 8-bit successive-approximation A/D converter. No other analog input or output device is in the chip. The fact that the MC6811 chip contains both digital and analog circuits on the same chip is noteworthy.

The A/D converter cannot cause an interrupt. The conversion is so fast that the overhead of servicing an interrupt makes polling attractive. Figure 4 shows ADCTL register which controls the A/D converter functions. Details of the A/D converter and related functions are given in [1], [7], [8], [12], [13].

ADCTL — A/D Control/Status Register $1030

	BIT 7	6	5	4	3	2	1	BIT 0
	CCF	—	SCAN	MULT	CD	CC	CB	CA
RESET:	0	0	U	U	U	U	U	U

Figure 4 ADCTL Register

2.2. THE MC6811 COMPUTER OPERATION

The functions performed by machine instructions can be more easily understood if you understand the operation of the computer hardware. This section introduces the principal registers in the MC6811 and explains their uses during instruction fetch and execution, and discusses the CPU architecture, addressing modes, and the instruction set (by instruction types). Further information about the operation of MC6811 can be found in [1], [7], [8], [12], [13].

2.2.1. PROGRAMMING MODEL

Figure 5 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.

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).

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 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.

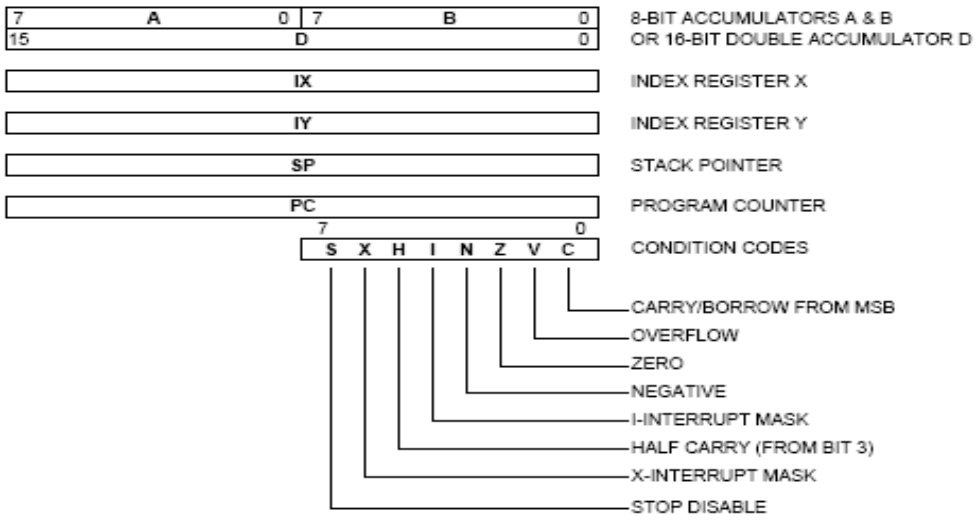

Figure 5 Programming Model of MC6811

The program counter is a 16-bit register that holds the address of the next instruction to be executed. The Condition Code 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. 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 H bit indicates a carry from bit 3 during an addition operation. The N bit reflects the state of the most significant bit (MSB) of a result. The Z bit is set when all bits of the result are zeros. The C bit is normally used to indicate if a carry from an addition or a borrow has occurred because of a subtraction. The STOP disable (S) bit is used to allow or disallow the STOP instruction. The interrupt request (IRQ) mask (I bit) is a global mask that disables all maskable interrupt sources. The XIRQ mask (X bit) is used to disable interrupts from the XIRQ pin.

2.2.2. INTERNAL COMPUTER OPERATIONS

This section discusses the operation of the microcomputer using the programming model. Examines fetch and execute phases of instructions.

The fetch phase of the instruction execution brings a copy of the next instruction code from memory into the microprocessor instruction register. The microprocessor enters the fetch phase each time it completes the previous instruction. When the microprocessor is turned on and starts running, the very first operation is a fetch. The microprocessor sends the number in the program counter register to the memory as an address and tells the memory to read. The program counter determines where an instruction will be fetched from, because it supplies the address to the memory.

So far, the fetch phase has brought the first byte of the instruction into the microprocessor and adjusted the program counter to point to the next memory location. The microprocessor has determined that three bytes must be fetched for this instruction. During the fetch phase of this instruction, the three bytes of the instruction were put into the instruction register and the program counter was adjusted. The entire instruction code has been fetched from memory into the microprocessor; therefore, the fetch phase of the instruction operation is complete.

The program counter now contains the address of the next location in memory after this instruction code. The next location must contain the next instruction in the program. The design of the computer requires that instructions be in order in the memory.

The instruction register holds the instruction code as it awaits the execute phase of the instruction. The fetch phase of an instruction never changes the accumulators or any memory registers.

The microprocessor automatically enters the execute phase of instruction operation at the completion of the fetch phase. The execute phase carries out the function specified by the instruction code. The right two bytes of the instruction register are sent to the memory as an address, and the memory is told to read. The right two bytes of the instruction register contain the address part of the instruction code and thus point to the data number in memory. During the execute phase, the microprocessor copied the addressed data number into the accumulator, destroying the original number there. The execute phase of this instruction does not affect the program counter. Only a few instructions affect the program counter during the execute phase.

Both the fetch and execute phases of the instruction operation are now finished. The microprocessor automatically begins a new fetch at the next tick of the clock. After the instruction operation is finished, the number in the program counter specifies the address of the next instruction to be fetched. The number in the instruction register is no longer useful. The accumulators hold data numbers including the one that resulted from this instruction.

2.3. ADDRESSING MODES

In the MC6811 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.

2.4. INSTRUCTIONS

This section is intended to explain the basic capabilities and organization of the instruction set. For this discussion, 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 MC6811 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.

CHAPTER 3

THE CROSS ASSEMBLER

In this chapter, the cross-assembler part of toolkit will be discussed. A program called an *assembler* translates the symbols to binary numbers that can be loaded into computer memory. The assembler puts together or assembles the complete machine code from the source code, which includes op-code and operand. A *cross-assembler* is an assembler, which runs on one type of processor (the host) and produces machine code for another (the target). This chapter discusses the Motorola Assembly Language for the MC6811 and the operation of the cross-assembler. [9], [10], [11] are references for Motorola Assembly Language and "how an assembler works?".

3.1. MOTOROLA ASSEMBLY LANGUAGE

Assembly language is a symbolic representation of operations (machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. Programs written in assembly language consist of a sequence of source statements.

An assembly language source statement is a single line of a source program. This section lists and describes the characteristics of standard Motorola Assembly Language source statement. Each source statement may include up to four fields: a *label* (or '*' for a comment line), an *operation* (instruction mnemonic or assembler directive), an *operand* and a *comment*. Some fields may be left blank. The fields have variable lengths and are delimited by spaces. The items in the fields must be separated at least one space. You may use more spaces without changing the meaning of the statement.

3.1.1. LABEL FIELD

The label field occurs as the first field of a source statement and represents the memory address of the labeled item. The EQU directive, which will be discussed later, overrides the assignment of an address value to the symbol by assigning its own value. The label field can take one of the following forms:

- An asterisk (*) as the first character indicates that the rest of he the source statement is a comment. Comments are ignored by the assembler.
- A white space character (blank or tab) as the first character indicates that the label field is empty. The line has no label and is not a comment.
- A symbol character as the first character indicates that the line has a label. Symbol characters are the upper or lower case letters a-z, digits 0-9 and the special characters, period (.), dollar sign ($) and underscore (_). Symbols consist of one to eight characters the first of which must be alphabetic or the special characters, period (.) or underscore (_). All characters are significant, upper, and lower case sensitive.

A symbol may occur only once in the label field. If a symbol occurs more than once in a label field, then each reference to that symbol will be flagged with an error. A label may appear on line by itself. The assembler interprets this as ‘set the value of the label equal to the current value of the program counter’.

3.1.2. OPERATION FIELD

The operation field occurs after the label field, and must be preceded by at least one whitespace character. The operation field must contain a legal opcode mnemonic or an assembler directive. Lower case characters in this field are converted to upper case before being checked as a legal mnemonic. Thus, ‘LDAA’, ‘ldaa’ and ‘LDaA’ are recognized as same mnemonic. Entries in the operation field may be one of the two types.

3.1.2.1. OPCODES

These correspond directly to the machine instructions. The M68HC11 family of microcontrollers uses 8-bit opcodes. Each opcode identifies a particular instruction and associated addressing mode to the CPU. Several opcodes are required to provide each instruction with a range of addressing capabilities. A four-page opcode map has been implemented to expand the number of instructions. An additional byte, called a prebyte, directs the processor from page 0 of the opcode map to one of the other three pages. As its names implies, the additional byte precedes the opcode. If the operation codes includes any register name associated with the instruction. These register names must not be separated from the opcode with any white space characters.

3.1.2.2. DIRECTIVES

Directives are symbolic commands in the source program that control the operation of the assembler program. Directives are placed in the operation field although they do not direct the assembler to generate instruction codes. Six directives are implemented in the cross-assembler.

-ORG Directive: The origin directive, ORG, changes the program counter to the value specified by the expressions in the operand field. Subsequent stamens are assembled into memory locations starting with the new program counter value. If no ORG directive is encountered in the source program, the program counter is initialized to zero. Expressions cannot contain forward references or undefined symbols.

-EQU Directive: The equal directive assigns the value of a numerical value in the operand field to a symbol in the label. The label cannot be redefined anywhere else in the program. The expression cannot contain forward references or undefined symbols. Equates with forward references are flagged with errors.

-FCB Directive: FCB is an acronym for form constant-byte. This directive may have one or more operands separated by commas. The value of the each operand is truncated to eight bits, and is stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one ore more of them can be null (two adjacent commas), in this case a single byte of zero will be assigned for that operand. An error will occur if he upper eight bits of the evaluated operands' values are not all ones or all zeros.

-FDB Directive: FDB is an acronym for form double-byte. This directive may have one or more operands separated by commas. The 16-bit value corresponding to each operand is stored into two successive bytes of the object program. The storage begins at the current program counter and the label is assigned to the first 16-bit value. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one ore more of them can be null (two adjacent commas), in this case a single byte of zero will be assigned for that operand.

-RMB Directive: RMB is an acronym for reserve memory bytes. This directive causes the location counter to be advanced by the value of the expression in the operand field. This directive reserves a block of memory whose length in bytes is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a table area in the memory.

-END Directive: this directive marks the end of the source program and needs no label or expression. The END statement tells the assembler it has reached the last source statement. The assembler will not read any more source lines after the END directive. If any additional source statements are beyond the END directive, they will

be ignored. When the assembler encounters the END it proceeds to generate the object module. The END directive is not same as the STOP instruction in the program. STOP is an instruction that controls the target microcontroller on the other hand END is an directive that controls the assembler.

-Asterisk '*': The assembler treats an asterisk anywhere other than column one or in a comment as another directive that tells the assembler to substitute the current address for the symbol for the symbol '*'. Asterisk means the current value of the program counter for the assembler.

3.1.3. OPERAND FIELD

The content of the operand field varies depend on the operator used in the operation field. If it is needed the operand field must follow the operation field and must be preceded by at least one whitespace character, but no spaces are allowed in the operand field. Extra spaces may cause your program to be in correct, while the statement is correct for the assembler. The operand field may contain a symbol, an expression or a combination of symbols and expressions separated by commas.

3.1.3.1. OPERATORS

The operators are the same as in C programming language:

+: add

- : subtract

*: multiply

/: divide

% : remainder after division

& : bitwise AND

| : bitwise OR

^ :bitwise exclusive OR

3.1.3.2. CONSTANTS

Constants represent quantities of data that do not vary in value during the execution of a program. Constants are presented to the assembler in one of five formats: decimal, hexadecimal, binary, and octal or ASCII. The programmer should indicate the number format to the assembler with the following prefixes:

$: HEX

% : BINARY

@ : OCTAL

‘ : ASCII

Unprefixed constants are received as decimal. A decimal constant consists of a string of numeric digits. The value of a decimal constant must fall in the range [0-65535]. A hexadecimal constant consists of a maximum of four characters from the set of digits (0-9) and the upper case alphabetic letters (A-F), and is preceded by a dollar sign ($). Hexadecimal constants must be in the range $0000 to $FFFF. A binary constant consists of a maximum of 16 ones or zeros preceded by a percent sign (%). An octal constant consists of maximum of six numeric digits. These digits are from 0 to 7. Octal constants are preceded by a commercial at-sign (@) and must in the ranges @0 to @177777. A single ASCII character can be used as a constant in expressions. ASCII constants are preceded by single quote (‘). Any character, including the singe quote, can be used as a character constant.

BSET, BCLR, BRSET and BRCLR instructions have more than one operand including MASK and ADDRESS information. These operands must be separated by colons (‘:’).

3.1.4. COMMENT FIELD

The comment field is the remainder of the line to the right of the operand field. The comment field is separated from the operand field, if there is no need to any operand

then from operation field, by at least one whitespace character. The comment field can contain any printable ASCII characters. The comment field may be left blank. The assembler ignores comments when generating object code. Since the comment filed starts right after the space character following the operand or operation field, avoid putting spaces in these fields.

3.2. ASSEMBLER PROCESS

This section discusses the cross-assembler part of the toolkit. The design and running details of the cross-assembler are given in this section. Assembly language is a symbolic representation of the instructions and data numbers in a program. This representation forms the source code of a program. An assembler (or more correctly, a symbolic assembly program) is a system that assists the programmer in the preparation of machine code programs. A program as executed by the processor consists usually of binary information in consecutive registers of the core. The assembler allows the programmer to write symbolic representations of the contents of store registers, and converts these into binary words with information for the loader. A loader program reads the load module and places the binary numbers into the memory of the target computer. Some assembler programs generate object module that are also load modules, so the name load module is sometimes an alternative to the name object module.

The assembler program also keeps the track of cross references within the program, and facilitates the combining of subprograms to form larger programs. The need for assemblers and loaders arises from the fact that the computer requires its instructions to be in the form of groups of binary digits, while the programmer likes to use alphabetical symbols and decimal numbers.

A cross-assembler is an assembler, which runs on one type of processor, called host, and produces machine code for another one, called target. The cross-assembler tool in this toolkit works on a two-pass basis, that is, it scans the source code twice. The first pass builds up a symbol table by collecting all the symbol definitions and the second pass does the actual translation, using the symbol values accumulated in the first pass. The cross-assembler must distinguish three sorts of information in the source program. First, there are those items, such as operation codes and numerical constants that have a value, which does not depend on the position of the routine in the store. They are thus independent of the way this routine is combined with other routines. Then there are symbols, which name instructions or working stores within the routine, which are not referred to by other routines. Although the actual store address of such items will depend on the position of the routine in the store, and hence on the way in which it is combined with other routines, the position relative to the start of the routine is known uniquely. Such information is said to be relocatable, since it is only necessary to add datum corresponding to the star of the routine in store to produce absolute address. Finally, there are cross reference symbols, which are defined, and/or referres to in other routines, whose values are unknown until the whole program is put together.

The input of the cross-assembler must follow the rules described in the above sections and extra rules given in the section 3.2.3. In addition, the cross-assembler outputs three files to the system. An OUX file, which is the object code to be loaded to simulator program. A LST file that contain the source code. In addition, a file that contains the symbols in the program and their values.

3.2.1. FIRST PASS

The aim of the first pass of the cross-assembler is to build up a symbol table by collecting all the symbol definitions. The symbol table constructed here is used in the second pass for generating the object code.
The cross-assembler parses the source code line by line and search for a label in each line during the first pass. If the label field is not empty, the program starts a validation process. The rules of this validation are described in section 3.1.1. If the label passes from the validation process, the program searches for the label in symbol table to avoid duplication of the label. If the label is not used previously it is added to the symbol table with related address.

Symbol table is the place where valid labels are kept for the second pass. A symbol table consists of a number of entries, each of which associates a symbol with a value. Alphabetical indexing is used for sort and search in this table. This method presupposes that the symbols are arranged in alphabetical order of the left most character. An auxiliary table is constructed which points to the start of each section in the main table. Once the appropriate sub table has been determined from the initial letter, it is linearly searched in the sub table.

If the symbol is used with EQU directive, the value of the operand is added to the symbol table instead of address. The first pass of the assembler gives errors if an invalid label is used, a duplicate label is used and forward reference is used with EQU directive.

3.2.2. SECOND PASS

Second pass of the assembler does the actual translation, using the symbol values accumulated in the first pass. The cross-assembler parses the source code line by line. The label field is ignored, not processed, in the second pass. The first processed field in a line is operation field. The cross-assembler reads the data here and determines

whether it is an assembler directive or an instruction mnemonic. There are two look up tables in the program, one is for directives and the other is for instruction mnemonics. In the assembler directive table, there are only the names of the directives. The cross-assembler compares the operation field data with entry in the table. If there is a match, the routine for the related directive is activated. If there is no match, the cross-assembler checks the instruction mnemonics table.

The instruction mnemonic table has four entries. The first entry is contains the mnemonic of the instruction to use in comparison. Second field is the valid addressing modes for the instruction. Third field is the hexadecimal opcode for related addressing modes. Fourth entry is the length of the instruction according to the addressing modes. The cross-assembler scans the table using alphabetical indexing. When it finds a match, the mnemonic is flagged to use its opcode after finding the addressing mode from the operand field. The lower case letters are converted into upper case, so all the comparisons during the operation field process is done wit upper case letters. If no match occurs for operation field, an invalid operation error returns to user.

The next step is to check operand field to determine addressing mode for the instruction. If the expression in the operand field is valid, the cross-assembler extracts the addressing mode. '#' sign as the first character states its immediate addressing mode. ',X', ',Y' at the end of the expression means the addressing mode is indexed. If the calculated value of the operand is less than or equal to FF then the addressing mode is direct, if it is greater than FF the addressing mode is extended. Inherent addressing mode needs no operand. In addition, branching instructions use no other addressing modes.

Once the addressing mode is decided and the operand is calculated, the flagged instruction is written to the memory location pointed by the location counter. The location counter keeps the trace of the memory locations of the instruction and their

opcodes. It works same as the program counter in the CPU. The opcode of the instruction decided by the addressing mode is added to memory. Than the operand is written to memory. Moreover, the cross-assembler passes to the next line of the source code. After processing of each line related data is added to the output files. A more detailed explanation of the cross-assembler process is in the following flowcharts. Figure 6 gives the details of the main loop of the ASM68TT11. Figure 7 shows the flowchart of the parsing part. Figure 8, Figure 9, Figure 10, Figure 11 and Figure 12 show how instructions and their operands are treated during first and second passes.

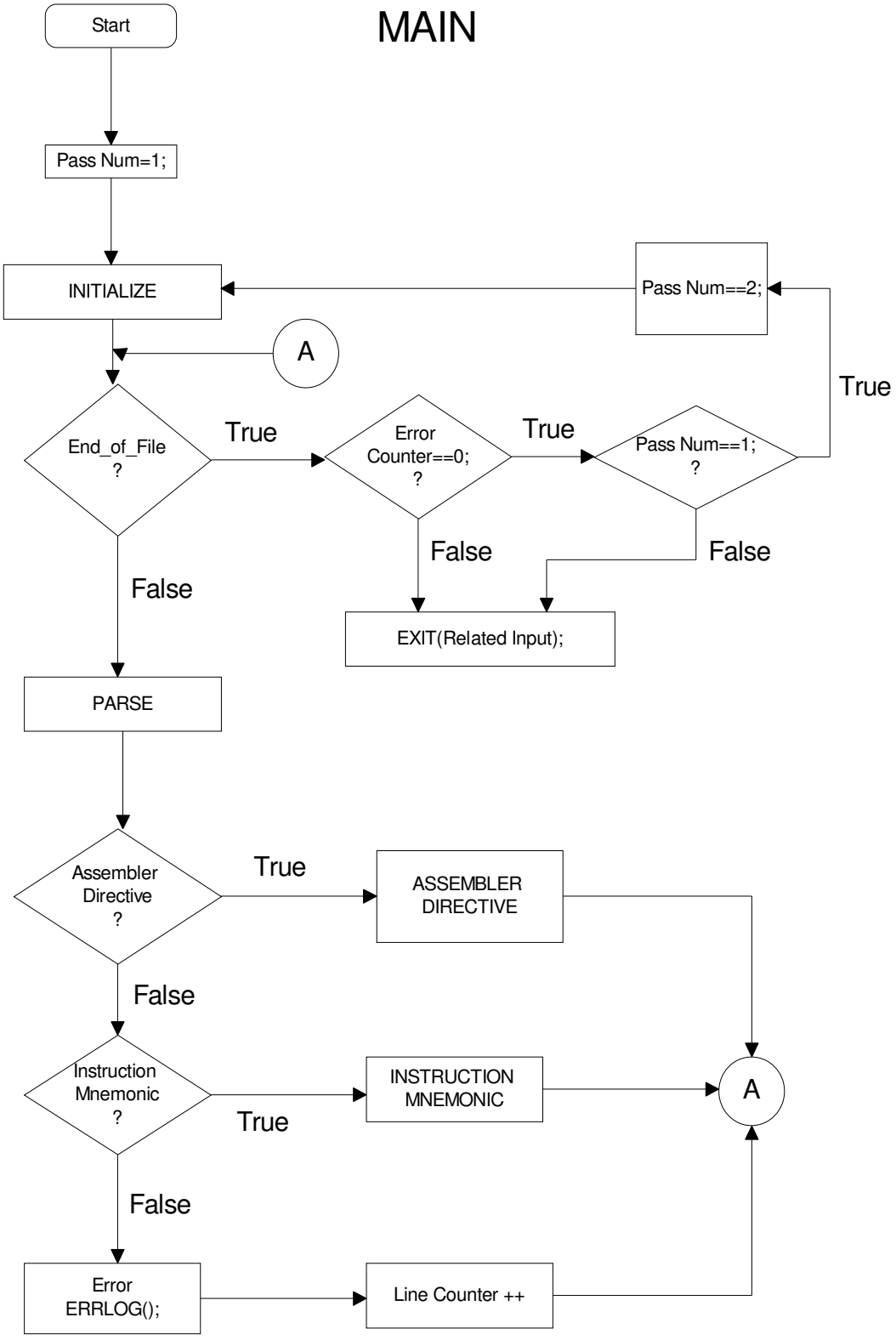

Figure 6 Main flowchart

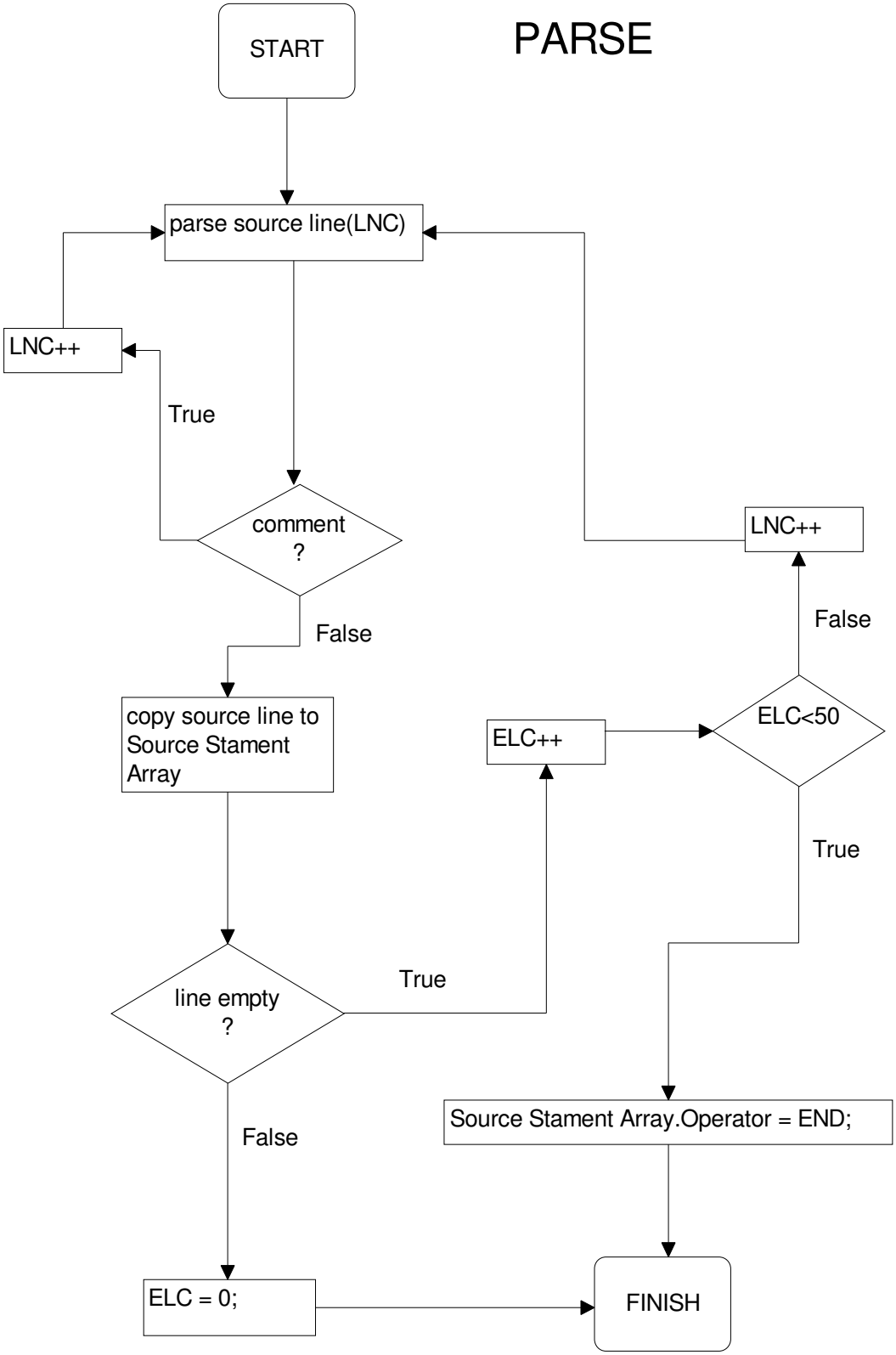

Figure 7 Parse flowchart

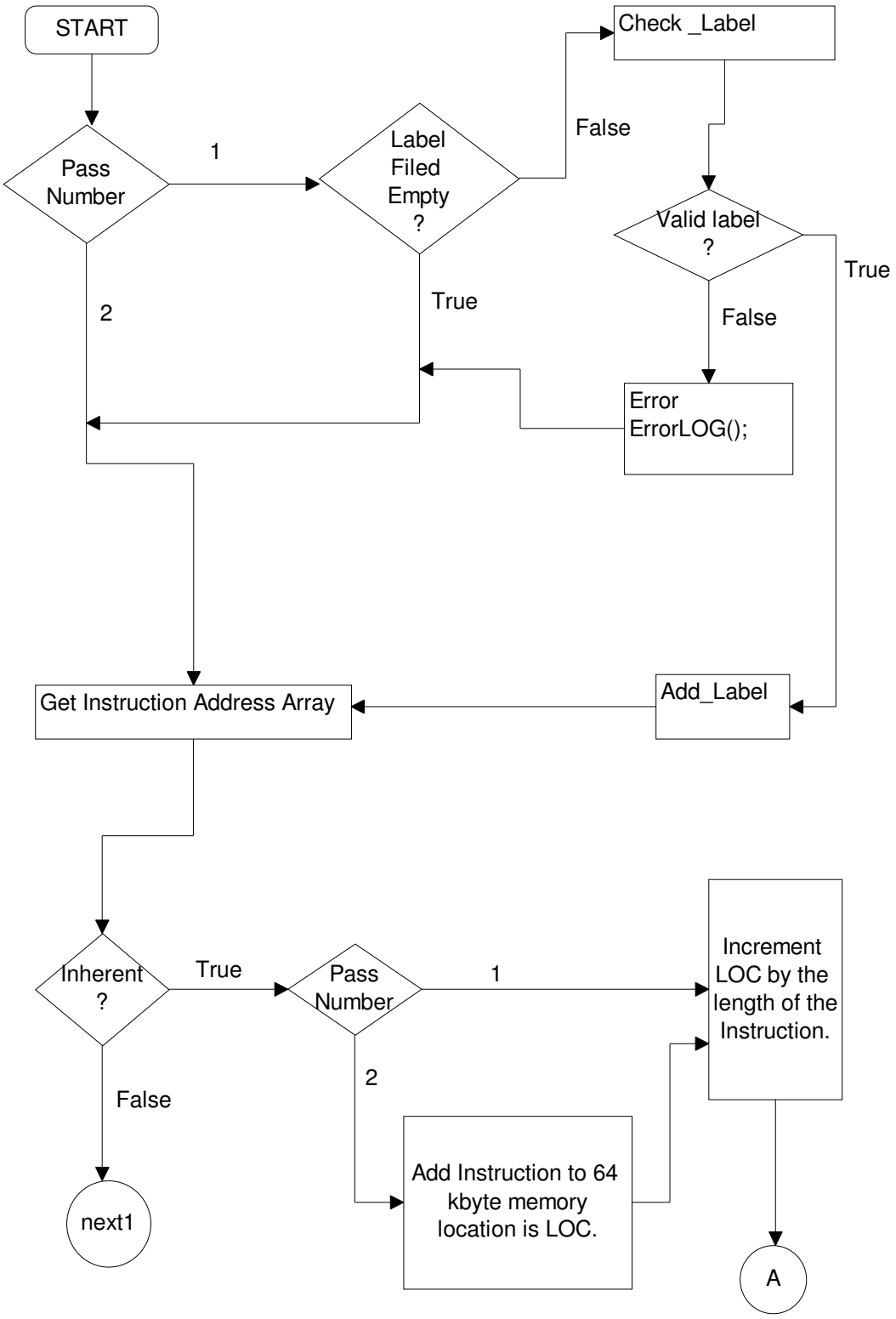

Figure 8 Instruction Mnemonic flowchart 1

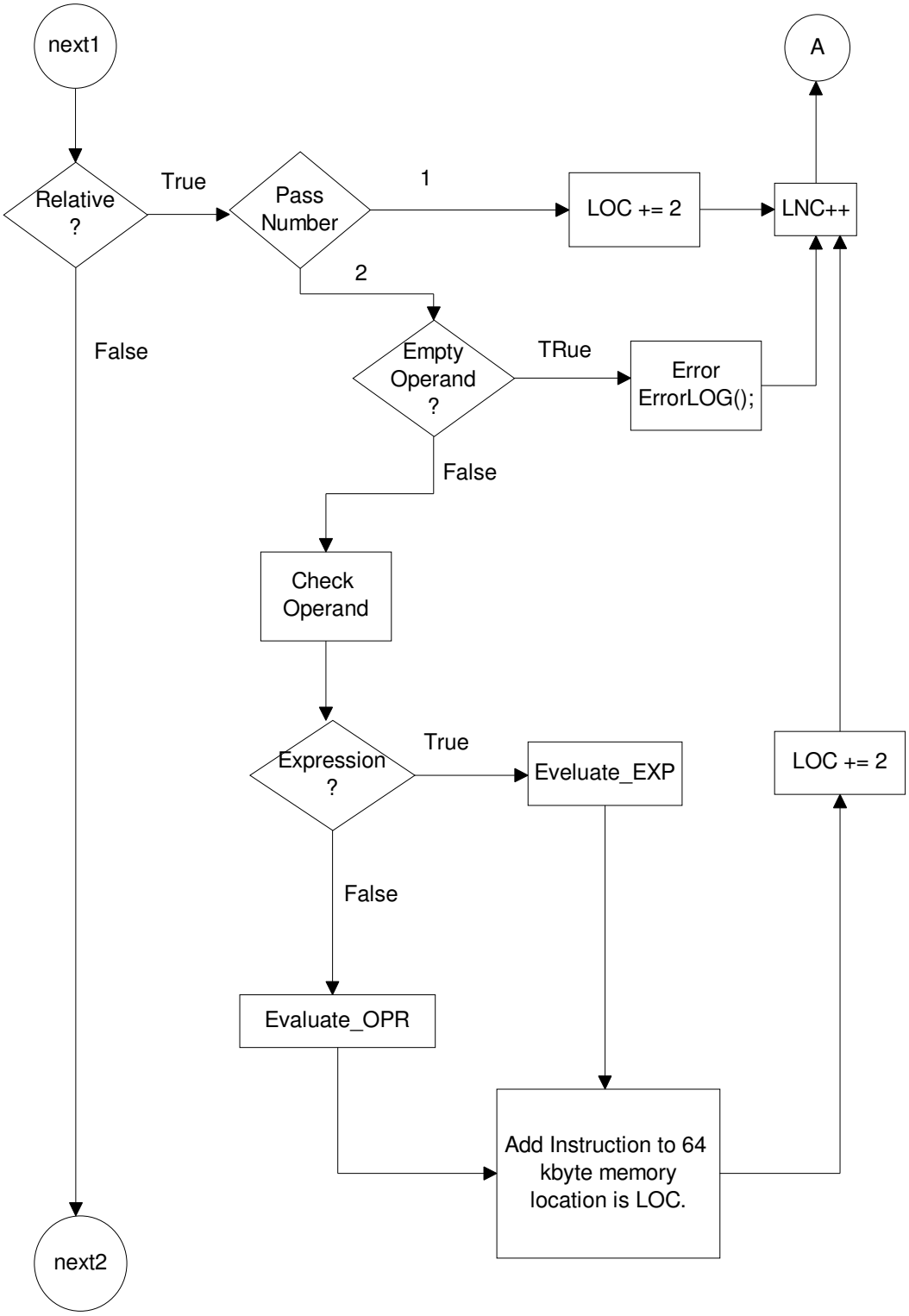

Figure 9 Instruction Mnemonic flowchart 2

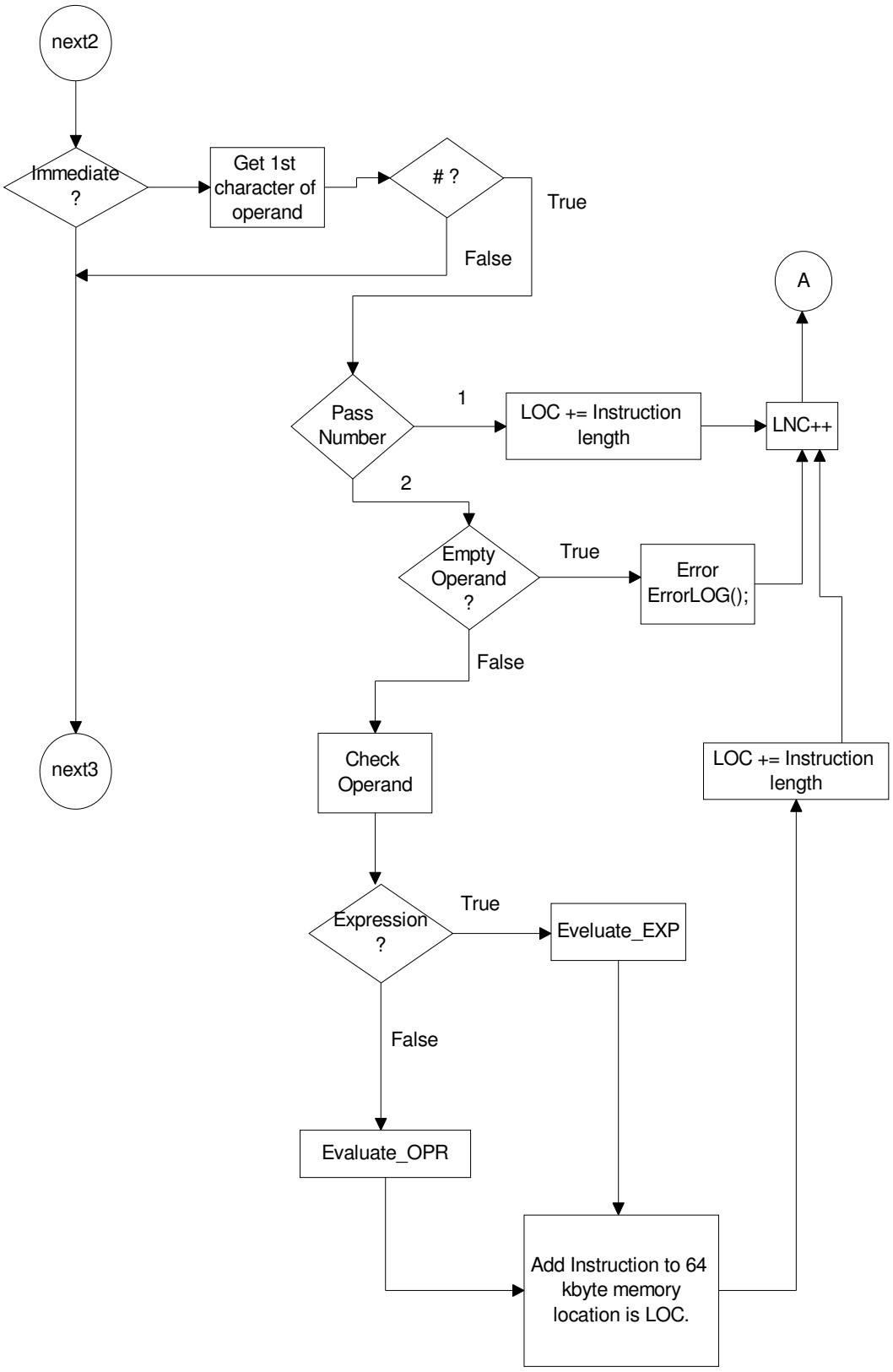

Figure 10 Instruction Mnemonic flowchart 3

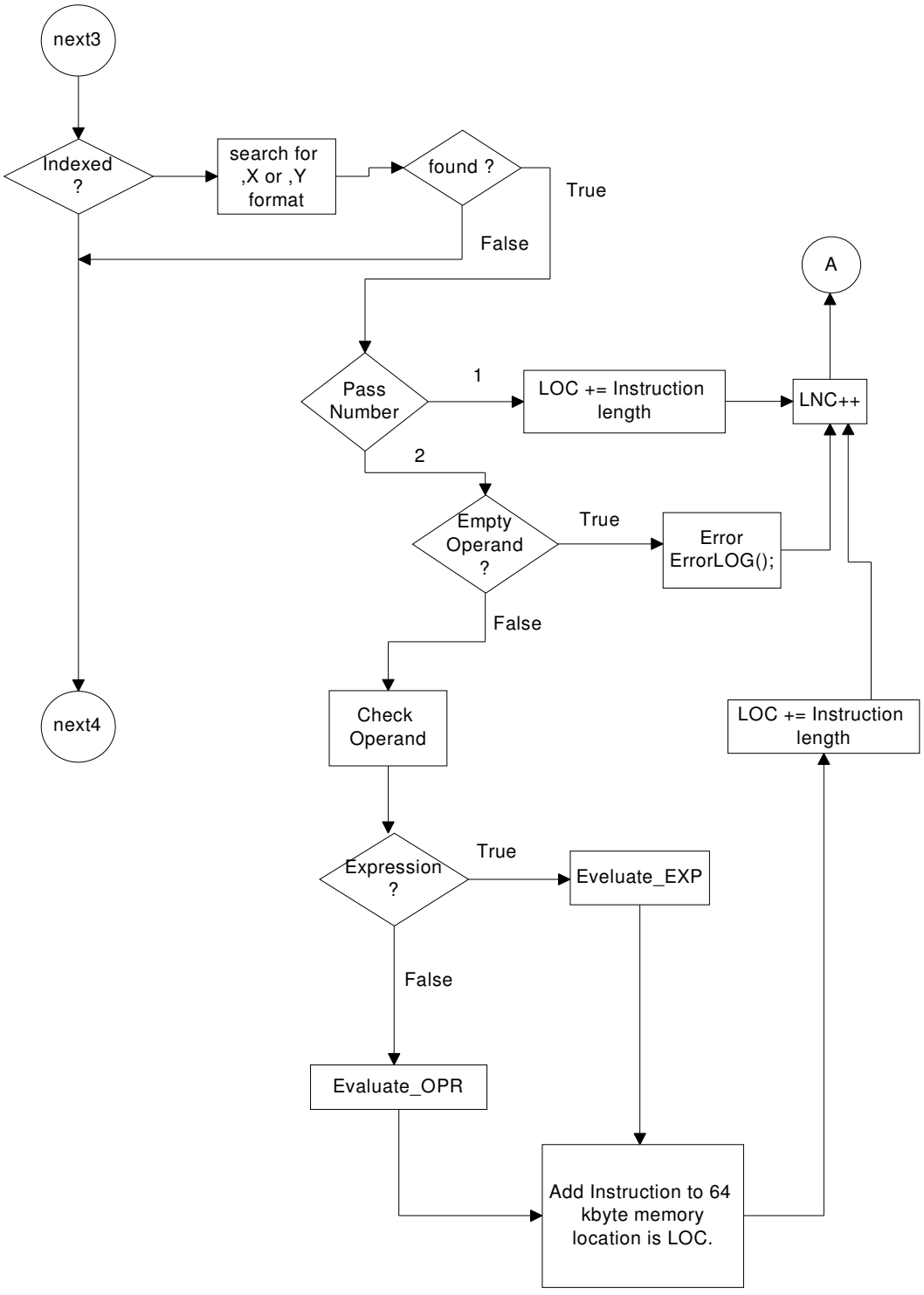

Figure 11 Instruction Mnemonic flowchart 4

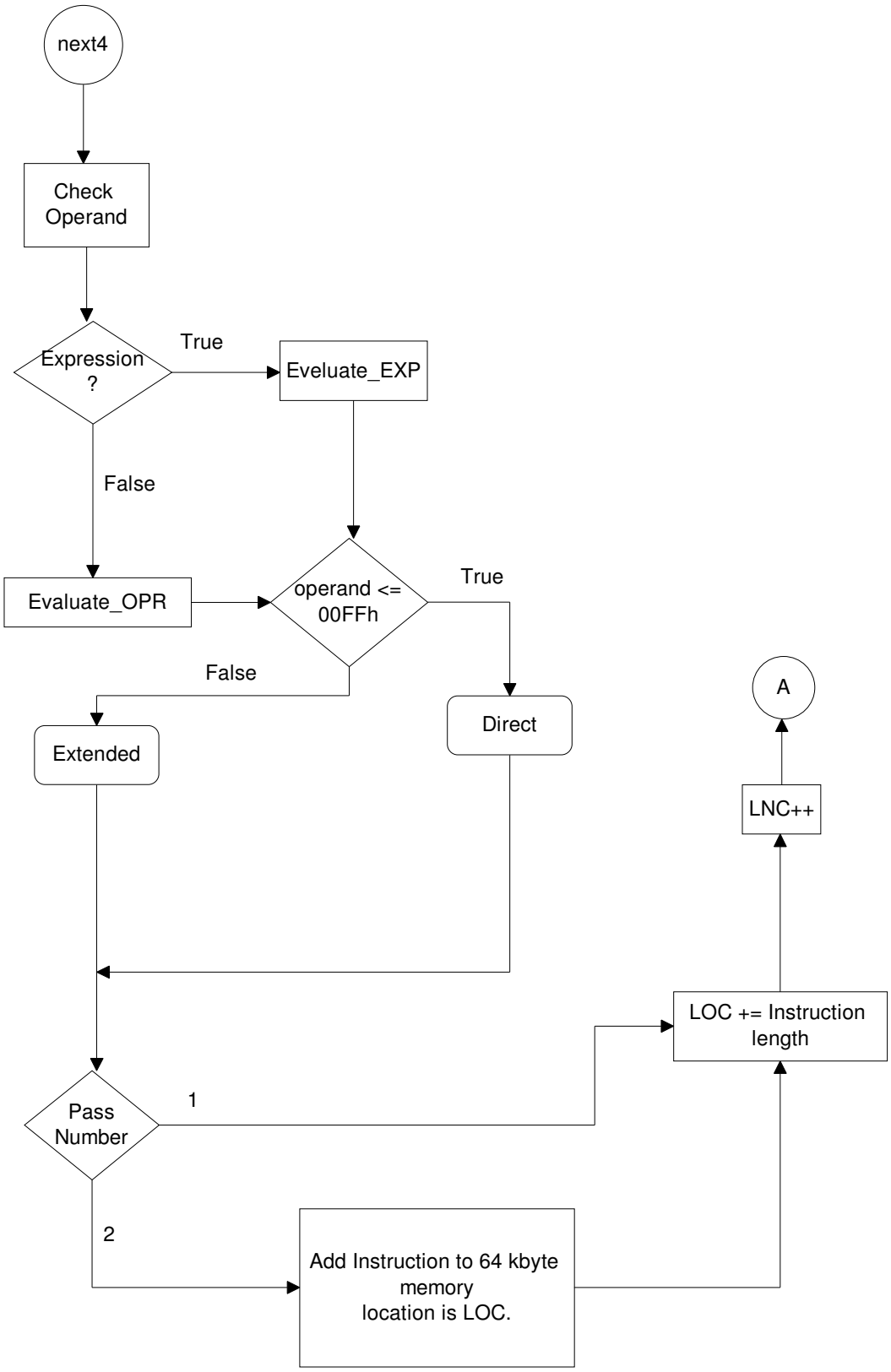

Figure 12 Instruction Mnemonic flowchart 5

3.2.3. INVOCATION OF THE ASSEMBLER

The cross-assembler is named as 'ASM68TT11.exe'. To generate MC6811 code runs the ASM68TT11.exe in command prompt. To run the cross-assembler enters the following command line:

ASM68TT11 file1.asm

There are no options for the program. Once the command line is written, press enter. The cross-assembler produces the output files related to the input file. The input file and the cross-assembler must be placed in the same directory.

The output files are DEN file, OUX file and LIT files. OUX file is the output of the cross-assembler for the simulator; this output file will be used by the simulator. DEN file contains the source code and the related object code; it can help the user to see the results of the source code and the hex code of the micro controller. LIT keeps the track of the variables and labels with their hexadecimal values. Cross-assembler gives the outputs at every successful process and places them in the same directory with itself.

The DEN output is very useful to verify the result of ASM68TT11 because it contains the input (source code) and output (hex code) in the same file. A verification chapter is included in Appendix B. During the verification of the ASM68TT11 an input source code, which contains all the possible combinations of instructions, is used. Then the DEN output of the ASM68TT11 is checked manually to verify the result.

The ASM68TT11 executable file and its user manual is given in a CD-ROM in Appendix D.

CHAPTER 4

THE SIMULATOR

In this chapter, simulator part of the toolkit is described. This chapter also discuses, which parts of the microcontroller are covered by the simulator and the details of the Graphical User Interface.

A simulator is a device, computer program or system used during software verification, which behaves or operates like a given system when provided with a set of controlled inputs. Name of the simulator discussed in this chapter is "SIM68TT11". The SIM68TT11 is designed to run the object codes created by the cross-assembler ASM68TT11. Information about the ASM68TT11 is given in Chapter 3. In addition to the ASM68TT11's 'OUX' format, SIM68TT11 can run the Motorola 'S19' file formatted object codes. SIM68TT11 has Graphical User Interface from where the inside structure of the Central Processing Unit, ports A to E, address and data busses, the interrupts XIRQ,IRQ and RESET, memory and memory control signals can be observed.

SIM68TT11 runs in two basic modes. First, one is the "Continuous Mode"; in this mode, the simulator executes the loaded code without stopping until a 'STOP' command is given by the user. Second running mode of the simulator is "Step by Step' mode. In this mode, SIM68TT11 executes the loaded code part by part, and between these parts, waits for the user to tell the simulator go on. The parts in this mode are defined by two options: "Instruction by Instruction" and "Cycle by Cycle".

The “Instruction by Instruction” option executes a single instruction at a time. The “Cycle by Cycle” option executes a cycle of a specific instruction at a time.

4.1. SCOPE AND DESIGN OF THE SIMULATOR

SIM68TT11 covers the Central Processing Unit, Input/Output ports A, B, C, D and E, Internal Memory and Interrupt Control with IRQ, XIRQ and RESET signals. On the other hand, SIM68TT11 excludes the special properties of the Input/Output ports associated with them, such as Analog Digital Converter property of Port E. SIM68TT11 also does not cover Real Time Interrupt system. The details of the simulator’s scope are given in the following sections.

Since the peripherals of the Central Processing Unit, except Input/Output ports and memory, are excluded in SIM68TT11, the Central Processing Unit is left as the main part of the simulator. During the design and implementation phases of SIM68TT11 most of the attention is paid to Central Processing Unit.

Seven main registers, an Arithmetic Logic Unit, a Control Unit and a Bus Interface Unit, represent the Central Processing Unit of MC6811 in SIM68TT11. The main issue of the design of Central Processing Unit is the internal bus structure. A double bus structure is used to connect the registers and other units inside the Central Processing Unit. This structure is shown in Figure 13. There are two internal data buses connecting the registers, Control Unit, Arithmetic Logic Unit and Bus Interface Unit. These two internal buses are called “Internal Data Bus Low” and “Internal Data Bus High” in the Central Processing Unit of SIM68TT11. The 16-bit registers are divided into two 8-bit registers and represented as low and high bytes. As an example of this, Program Counter is represented with two 8-bit registers called Program Counter high byte and Program Counter low byte. The Internal Data Bus Low connects Accumulator A, Index Register X low byte, Index Register Y low byte, Stack Pointer low byte and Program Counter low byte. In addition, the Internal Data

Bus High connects Accumulator B, Index Register X high byte, Index Register Y high byte, Stack Pointer high byte and Program Counter high byte. These two buses are also connected to the Arithmetic Logic Unit and Bus Interface Unit. The Control Unit connects two internal buses and provides data connection between these buses.

4.1.1. INCLUDED AND EXCLUDED PARTS AND PROPERTIES OF MC6811

SIM68TT11 does not cover all parts and properties of MC6811. This section discusses the included and excluded parts and properties of MC6811 by examining the controller part by part.

4.1.1.1. CENTRAL PROCESSING UNIT

The Central Processing Unit of MC6811 has seven main registers. These registers are: two 8-bit general purpose registers Accumulator A and B, two 16-bit registers for indexed addressing mode Index Register X and Y, a 16-bit register for stacking operations Stack Pointer, a 16-bit register that shows the next instruction to be executed Program Counter and an 8-bit Condition Code Register used for control of the interrupts and branch operations. All of these registers are included in SIM68TT11. An Arithmetic Logic Unit is included, but the operations of this unit are not shown for simplicity. User can see only the inputs and the output of the related operation.

The Control Unit of Central Processing Unit is a synchronous sequential network that sends control signals to the memory, and other parts of the system. Control Unit has a three-byte instruction register that holds an instruction code number as it is fetched from memory. The instruction register is said to be transparent because its operations cannot be seen.

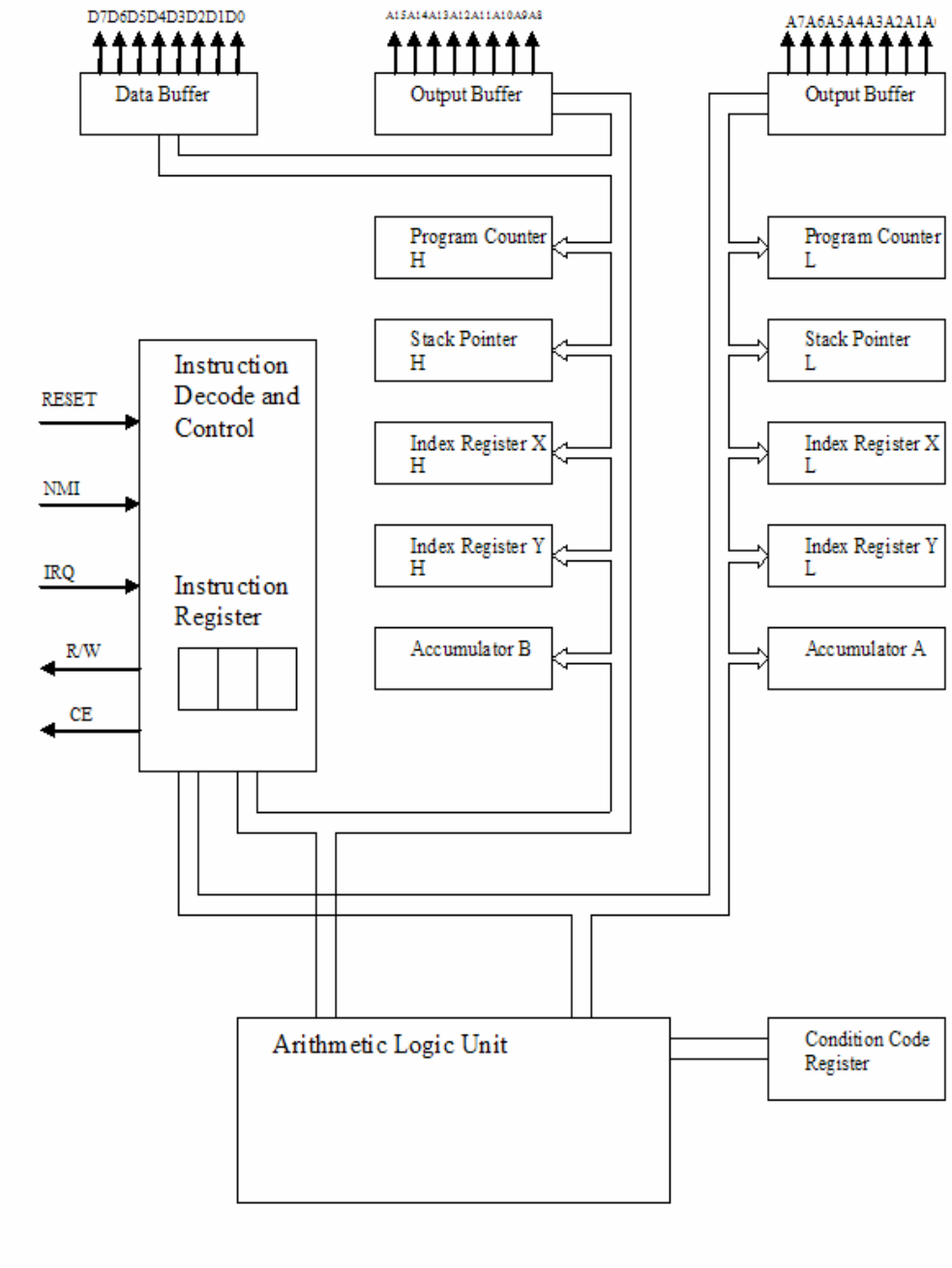

Figure 13 Block Diagram of Microprocessor

Therefore, instructions cannot control the instruction register. The programming model usually does not show the instruction register. SIM68TT11 on the other hand, has a control unit with an instruction register in it. The instruction register is shown in SIM68TT11 to give a better understanding to the user. The three-byte instruction register divided in to two parts in SIM68TT11. First part is the one-byte instruction register and the second part is the two-byte data register. The names imply the functions of the bytes in the instruction register of MC6811. Instruction Register holds instruction opcode while Data Register keeps data related to the instruction such as address information of an extended addressing mode instruction.

Another part of the Central Processing Unit is the Bus Interface Unit. This unit helps the processor to load or read address and data buses. It includes a Memory Address Register to store the 16-bit address and a Memory Buffer Register to store 8-bit data. SIM68TT11 has the Bus Interface Unit in the same manner. The Bus Interface Unit has also two control signal “Read/Write” and “Chip Enable”. These two signals control the memory module to read and write.

The internal data bus system is explained in previous section. The Central Processing Unit has also an internal control bus to control the registers and other units. This bus is shown in SIM68TT11’s Graphical User Interface but has no function. Instead of showing the control signals, they are explained in information boxes in SIM68TT11.

4.1.1.2. PORTS AND RELATED PROPERTIES

The MC6811 has five ports for digital input output operation and other specific operations. For simplicity, only the digital input output operations are implemented in SIM68TT11. The programmable timer and the pulse accumulator use port A pins addition to the parallel Input/Output functions. The Programmable Timer and the Pulse Accumulator are not functional. The PORTA register address 1000 controls the digital input and output functions. Pins PA0, PA1 and PA2 are input only pins. Pins

PA4, PA5 and PA6 are output only pins. PA7 and PA3 are bidirectional pins. By using data direction pins DDR3 and DDR7 in the PACTL register user can control the direction.

Port B is always an output only digital port as it is in MC6811. No other devices use the port B pins. Port C is always a programmable Input/Output port. The data direction register DDRC at address 1007 determines whether a pin is an input or an output. The program may either read or write the Port C register at address 1003. No other devices use the port C pins. SIM68TT11 uses the port C in the same manner.

Port D is a 6-bit programmable Input/Output port. The PORTD register at address 1008 is used to read or write data to the port. The data direction register DDRD at address 1009 controls the direction of each pin. The Serial Communication Interface uses port D pins to for communication. Since this property of the MC6811 is not implemented in SIM68TT11, port D pins function as only digital Input/Output pins. The port E pins are both an input only digital port and the inputs to the Analog to Digital Converter. The digital input port is read by reading the PORTE register at address 100A. SIM68TT11 includes port E only as digital Input/Output pins. No Analog to Digital Converter function is implemented.

SIM68TT11 has two buses between the Central Processing Unit and its peripherals. These are called External Data Bus and External Address Bus in SIM68TT11. Data bus is used to transfer data between memory and Central Processing Unit. Address Bus carries the address of the data. There are also two control signals for read and write operations of memory.

As mentioned before Analog to Digital Converter, Serial Communication Interface, Pulse Accumulator and Timer, Real Time Interrupt system, Serial Peripheral Interface and the related properties of these systems are not implemented. The

interrupts caused by these systems and the control registers in the memory are not effective in SIM68TT11.

4.1.1.3. INTERRUPTS AND MEMORY

Reset and Interrupt system of MC6811 was described in previous chapters. For these interrupts, XIRQ and IRQ are implemented and functional. In addition, Illegal Opcode Fetch Interrupt is functional. A reset button in the Graphical User Interface acts as the RESET pin of MC6811.

MC6811 has 64 kilobytes of memory. This memory space divided in to ROM, RAM and EEPROM partition in MC6811. SIM68TT11 has 64 kilobytes of memory but there is neither RAM nor EEPROM nor ROM partitions. All of the memory can be accessed and used by the user. The memory space of MC6811 also contains Hardware Control Registers. The place of these registers in memory can be changed. However, SIM68TT11 includes only the hardware control register related with Input/Output ports and operations. SIM68TT11 does not include all of the hardware components of MC6811. Because of this some of the Hardware control registers are not implemented. This registers acts as normal memory words. Implemented Hardware Control Registers are shown in Table 1.

When designing and implementing the simulator it is assumed that MC6811 is always running in “Normal Single Chip Mode”. TEST instruction is not implemented because it is an illegal opcode in this mode.

Also for the STOP instruction the following information which is given in [8]: “An error in some mask sets of the M68HC11 caused incorrect recover from STOP under very specific unusual conditions. If the opcode of the instruction before the STOP instruction came from column 4 or 5 of the opcode map, the STOP instruction was incorrectly interpreted as a two-byte instruction. A simple way to avoid this potential

problem is to put a NOP instruction (which is a column 0 opcode) immediately before any STOP instruction." is ignored in SIM68TT11.

Table 1 Hardware Control Registers of SIM68TT11

Name	Address
PORTA	$1000
PORTC	$1003
PORTB	$1004
DDRC	$1007
PORTD	$1008
DDRD	$1009
PORTE	$100A
PACTL	$1026

E_Cycle information is given only for the execution of instructions in the reference manual. Therefore, SIM6811TT does not increment the E_Cycle when it is not executing an instruction listed in valid instructions i.e. while stacking the registers before interrupt routines. The cycle in "Cycle by Cycle" operation and E_Cycle are different concepts.

4.1.2. DESIGN OF SIM68TT11

SIM68TT11 is designed in order to work in the same way as MC6811. MC6811 Central Processing Unit uses a micro program that is stored in its control unit to run the processor in a predefined way. SIM68TT11 does the same thing by using the algorithms in it. The flowcharts in Figures 14 and 15 explain the algorithm behind SIM68TT11 operation. When micro controller is invoked by a reset or power up it fetches the start vector from the predefined location. Then initializes the hardware and goes on executing the stored program as explained in the flowcharts in Figures 14 and 15.

There is a main routine in the SIM68TT11 program. This main routine is responsible of fetching and executing the instructions. It also checks the interrupt requests before each fetch cycle and serves the interrupt. Fetch cycle is same for all instructions but during the execution cycles, the main routine calls subroutines for each instruction. Subroutines are special functions having similar names with mnemonics (e.g. exec_PSH, exec_RTI). Some similar instructions use same subroutines. For example ADDA, ADDB, SUBA, SUBB use 'exec_Arith' subroutine. This subroutine treats the instructions in the same way but uses Accumulator A or B and adds or subtracts according to its inputs. There are enumerated data types defining the registers and arithmetic operations. These data types used as inputs to tell the common subroutines which register will be used with which operation.

After the subroutine execution, main routine fetches another instruction and execute it. Before the main routine fetches the next instruction it checks for the interrupts and if there is an interrupt needs to be served, main routine calls the related subroutine, which will process the related interrupt. The interrupt subroutine prepares the micro controller for interrupt subroutine and then passes the control to the main routine again. Main routine fetches the next instruction and executes it.

To sum up the discussion about the software architecture of SIM68TT11; SIM68TT11 has a main routine that controls the main functions of the program and calls subroutines for interrupts and execution phase of the instructions. There are data structures, which represents the registers in the CPU, data and address busses and memory. Enumerated data types are used for register names and arithmetic and logic operations. These data types make the program code more understandable and easy to read. Appendix A discusses some enumerated data types and main subroutines used in the program. The details of the microcontroller working principles and peripherals are extracted from the references [1], [2], [3], [6], [7], [8], [12] and [13].

A function writes all the register and any memory locations, which have been reached during the execution of an instruction, to a file. This function is enabled only for verification of the simulator and no longer functional. The out file also contains the executed source code. This output file is given in Appendix C.

4.2. GRAPHICAL USER INTERFACE

Graphical User Interface is the face of the program to the user. While routines of simulator algorithms work in the background, the user can see the result of them in the Graphical User Interface. Graphical User Interface window consists of a main window that displays the microcontroller parts, and a menu bar, where user controls the program. Figure 16 shows the Graphical User Interface.

4.2.1. MAIN WINDOW

The main window of the Graphical User Interface displays the CPU (Central Processing Unit). Port A, Port B, Port C, Port D and Port E, Interrupts and Memory Block with its control signals address and data busses. It also displays the flowchart of the running program. This window consists of buttons, labels, edit boxes, animated buses.

4.2.1.1. MICROCONTROLLER

Microcontroller part of the simulator shows implemented parts of MC6811 to the user. Following sections will introduce parts of the microcontroller.

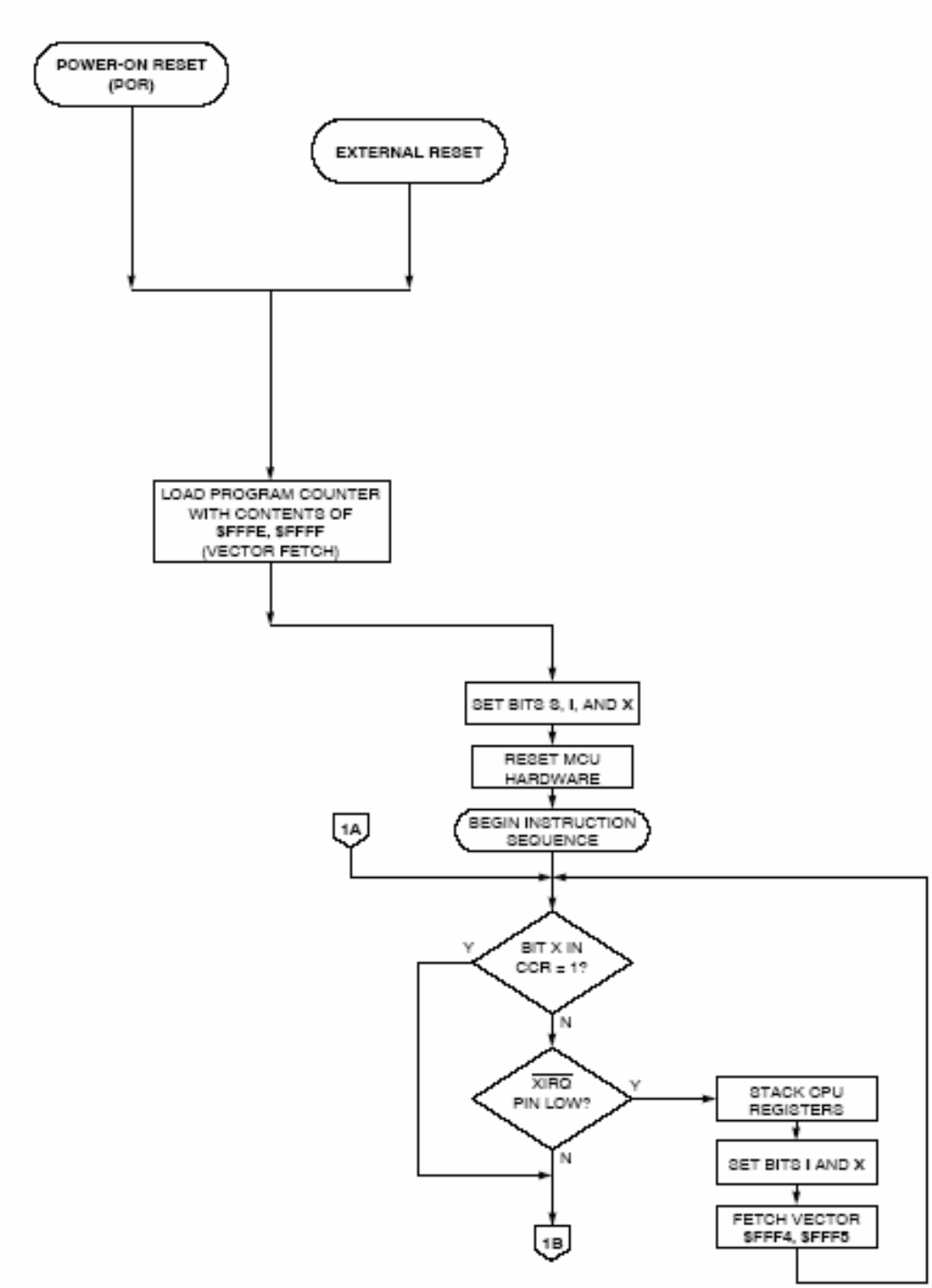

Figure 14 Simulator Flowchart Part 1

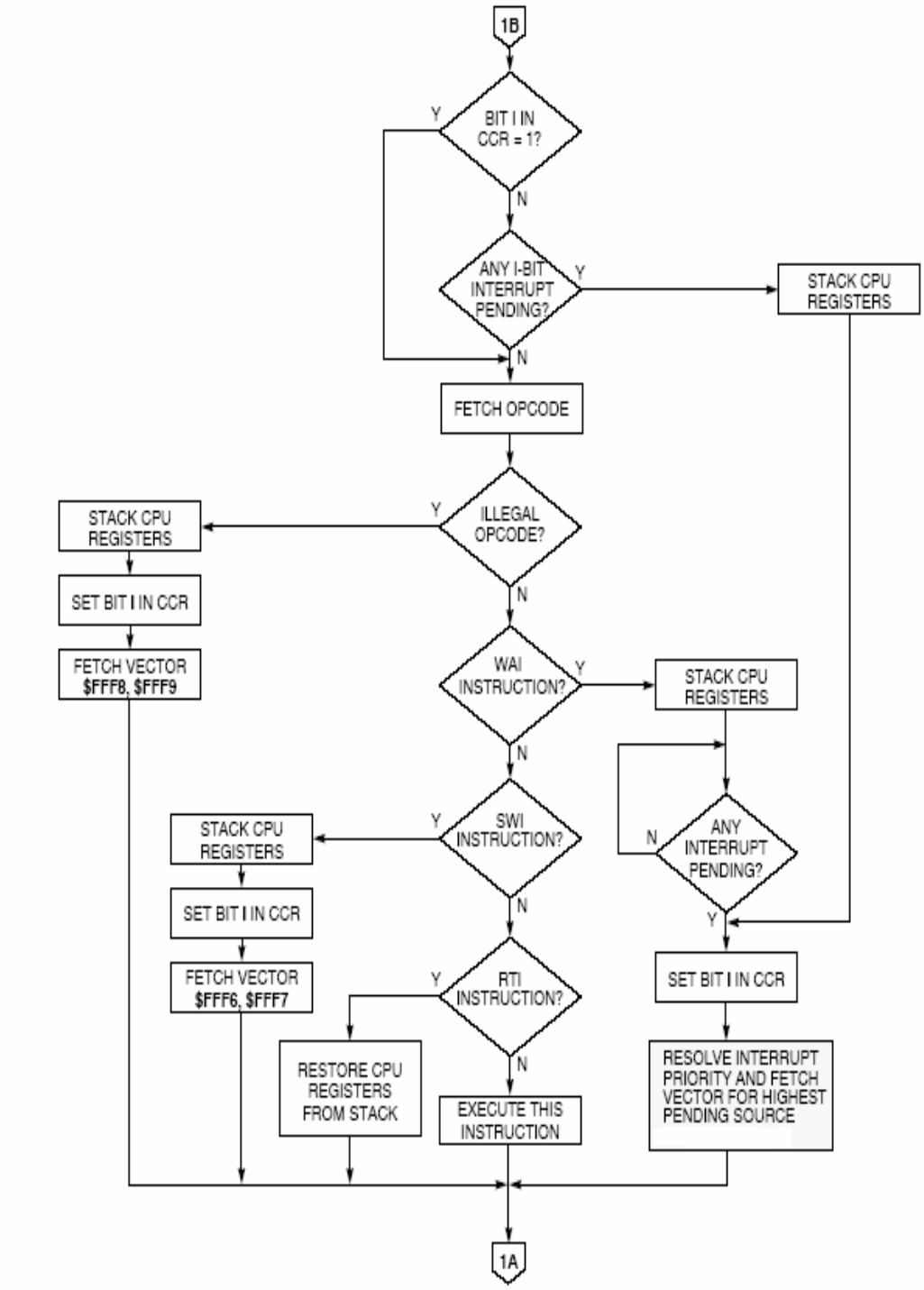

Figure 15 Simulator Flowchart Part 2

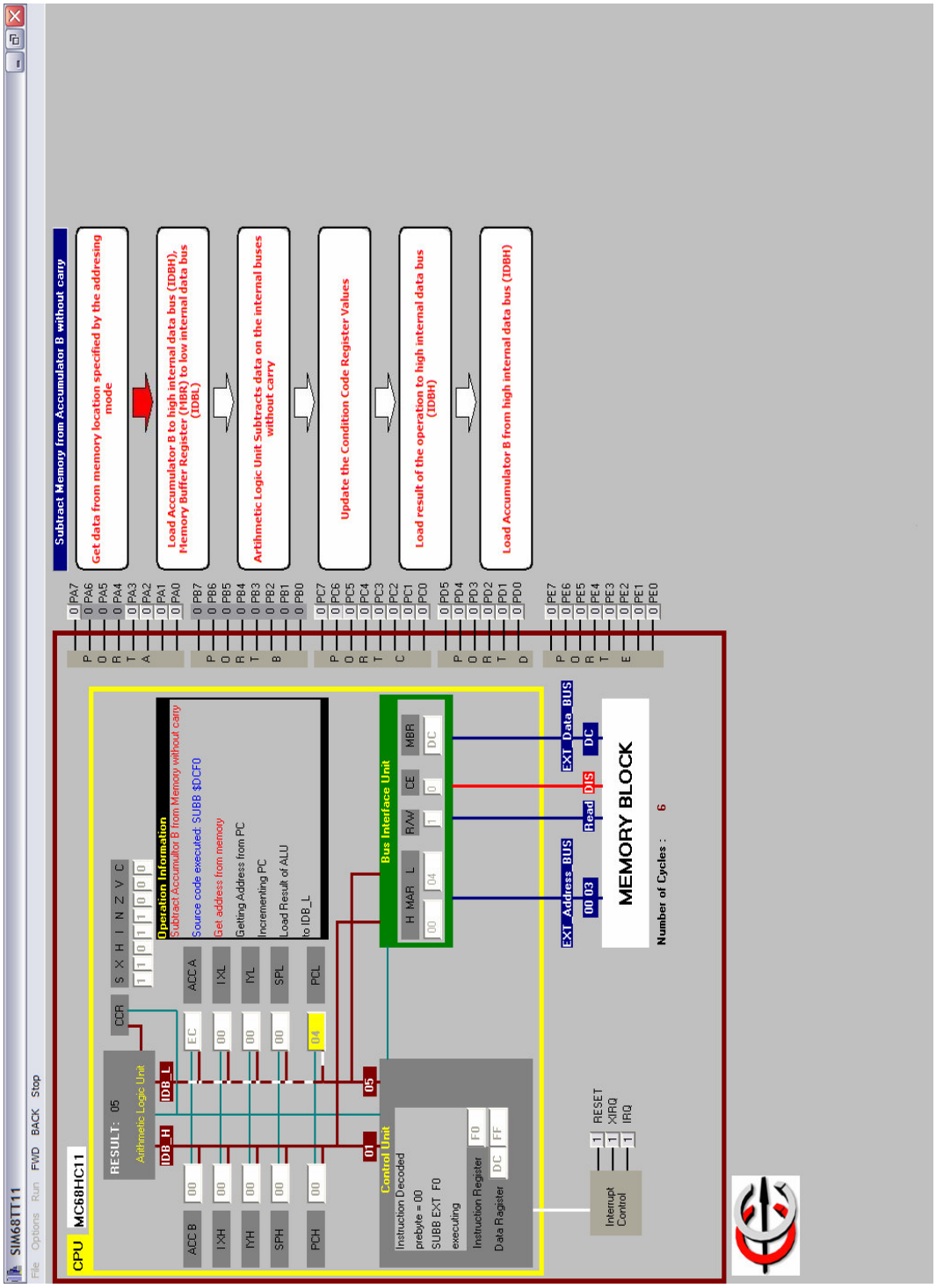

Figure 16 SIM68TT11 Main Window

4.2.1.1.1 CENTRAL PROCESSING UNIT

Central Processing Unit part of the main window is the heart of the Graphical user Interface. The entire microcontroller Central Processing Unit registers is displayed here with the Central Processing Unit's Control Unit, Bus Interface Unit and operation info boxes.

4.2.1.1.1.1. REGISTERS AND INTERNAL BUSES

The Central Processing Unit has two internal data buses and an internal control bus. The Internal Data Buses, low and high are connected to the Central Processing Unit registers. The Central Processing Unit registers are the 8-bit Accumulator A (ACCA), Accumulator B (ACCB); 16-bit Index Register X (IX), Index Register Y (IY), Stack Pointer (SP), Program Counter (PC). The 16-bit registers are separated in two equal 8-bit parts, a low byte and a high byte. Therefore, the Internal Data Bus Low connects Accumulator A (ACCA), Index Registers X low byte (IXL), Index Register Y low byte (IYL), Stack Pointer low byte (SPL), Program Counter low byte (PCL), Bus Interface Unit, Arithmetic Logic Unit and Control Unit of the Central Processing Unit. On the other hand, Internal Data Bus High connects Accumulator B (ACCB), Index Registers X high byte (IXL), Index Register Y high byte (IYL), Stack Pointer high byte (SPL), Program Counter high byte (PCL) Bus Interface Unit, Arithmetic Logic Unit and Control Unit of the Central Processing Unit. When a data will be moved from a register to another it is loaded to related data bus. In addition, the bus shows the way by highlighting the connection between the registers. Moreover, destination is painted to yellow while departure is blue. The registers Accumulator A, Accumulator B, Index Register X, Index Register Y, Stack Pointer, Program Counter are registers from programmers model. Figure 17 shows these parts.

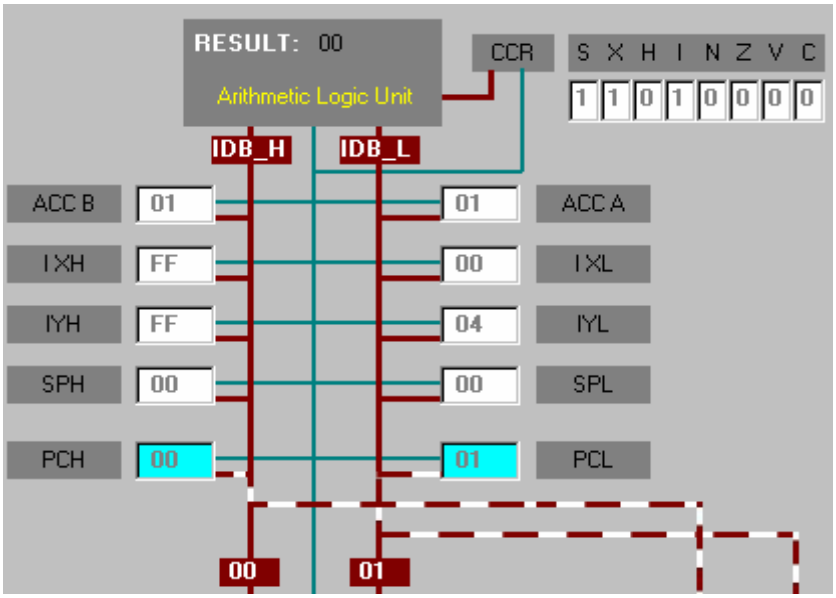

Figure 17 Registers and Internal Buses

4.2.1.1.1.2. CONTROL UNIT

The Control Unit of the Central Processing Unit contains a micro program and controls all the operations in the Central Processing unit and microcontroller. Figure 18 shows the Control Unit. It is figured out with an Instruction Register (IR) two Data Registers (DR) and an info field. The Instruction Register is the place where the micro program fetches the opcode of the instruction from memory to decode and execute. Data Registers are for the fetched operands or other data except opcodes. The micro program runs the microcontroller using these registers. The info field displays information about current Control Unit activity. This field has four lines for information. First and fourth line from the top tells whether an instruction is in fetching or executing phase. Second line shows the current opcode map by displaying pre byte of the instruction. Third line displays brief information about the current instruction. First part of this line is the mnemonic of the instruction (LDS: Load Stack Pointer as in Figure 20), second part is the addressing mode (IMM: Immediate as in Figure 20) and last part gives the opcode information to the user. The Control

Unit displays this information box only in “Step by Step” mode of SIM68TT11. In “Instruction by Instruction” and “Continuous” modes, no such information is displayed. In these modes, “Operation Information Filed” is used for instruction information.

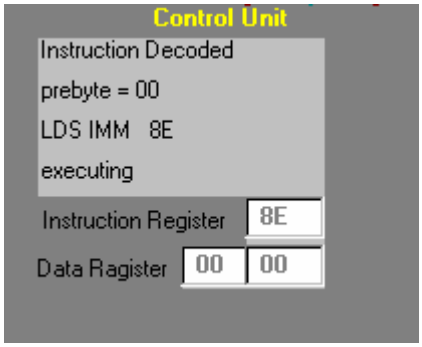

Figure 18 Control Unit

4.2.1.1.1.3. BUS INTERFACE UNIT

The Bus Interface Unit (BUI), shown in Figure 19, is the communication part of the Central Processing Unit with rest of the microcontroller. It has a Memory Address Register (MAR), which is a 16-bit register and separated into two 8-bit by 8-bit and labeled as MARH, MARL. There is also an 8-bit Memory Buffer Register (MBR). In addition, two control signals for read/write (R/W) and chip enable (CE). Memory Address Register stores the address of the data that will be processed. Memory Buffer Register holds the incoming and outgoing data. Control signals are only for memory to select read write and enable the chip.

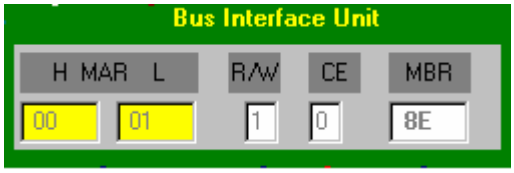

Figure 19 Bus Interface Unit

4.2.1.1.1.4. OPERATION INFORMATION BOX

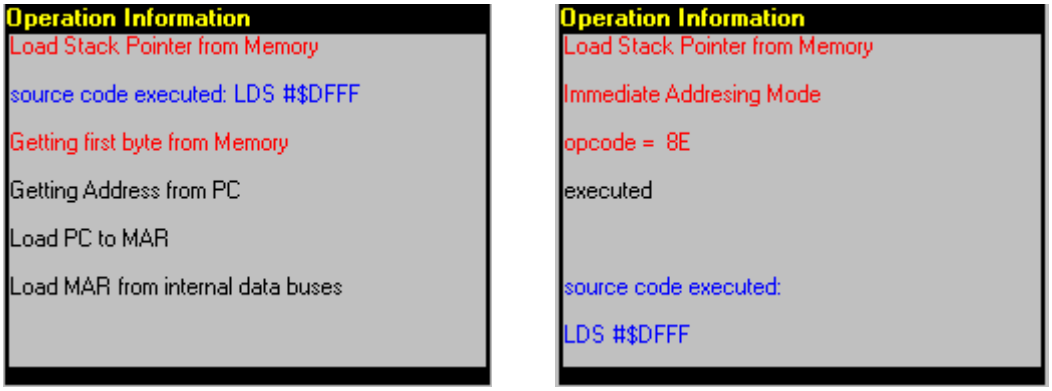

Figure 20 Operation Information Box

The Operation Information field gives related information about the Central Processing Unit operations. While the Central Processing Unit executes or fetches an instruction the detailed information is given in this field at every step of the process. Therefore, the user can follow the Central Processing Unit operations from this info field.

The Operation Information Field's functions are different in "Cycle by Cycle" operation and "Instruction by Instruction" operation. Figure 20 shows this field on both modes. The left hand side figure is for "Cycle by Cycle" mode. There are seven lines in this field and first line is the description of the current instruction. Second

line displays the source code that is currently in process and letters in this line are blue. The rest of the lines give information about the current operation of the Central Processor. User can follow what is going on in the processor.

In the “Instruction by Instruction” mode, the first line again displays the description of the current instruction. Second line gives the addressing mode and third line shows the opcode of the current instruction. Lines six and seven are the source code that is currently in process and letters in these lines are blue.

4.2.1.1.2. OTHER HARDWARE IN THE CONTROLLER

4.2.1.1.2.1. INTERRUPT CONTROL

The simulator has three interrupts; these are RESET, XIRQ and IRQ. The interrupts are represented by buttons. The user can interrupt the system by clicking the related button. Figure 21 shows these buttons. When user presses on an interrupt or reset button, the label turns to red and the value on the line turns one to zero. When the related interrupt or reset is serviced, the interrupt controls turn to their original states.

Figure 21 Interrupt Control

4.2.1.1.2.2. I/O PORTS

There are five ports in the simulator as in the microcontroller. These ports are Port A, Port B, Port C, Port D and Port E. Toggle buttons and labels represent the ports. Labels mean the pin of the port is output and cannot be altered by the user. The toggle buttons are for the input pins of the ports. If a 1 is displayed in the button, the pin has 1 as an input, and if a 0 is displayed in the button, the pin has 0 as an input. If user clicks the button, button toggles it value from 1 to 0 or 0 to 1 as well as the input to related pin. Figure 22 displays the Port A of SIM68TT11. In this figure PA7, PA3, PA2, PA1, PA0 are inputs while PA6, PA5, PA4 are outputs. Other ports use the same visualization. The special abilities of the ports are not functional. The labels on the right of the figure are the pin names of the ports.

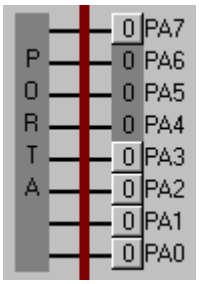

Figure 22 Input Output Ports

4.2.1.1.2.3. MEMORY

Memory of the simulator, shown in Figure 23, has two parts. First part is the address, data buses and memory control signals. The address bus is named as EXT_Address_BUS and keeps the address of the data. The data bus is called

EXT_Data_BUS and keeps the data to be processed. Current values of these buses can be seen on the screen.

The control signals are Read/Write and Chip Enable signals. When reading from memory this line is blue and when writing to memory, the line goes to red. In the same manner, the Chip Enable line is blue when memory is enabled, red for disable.

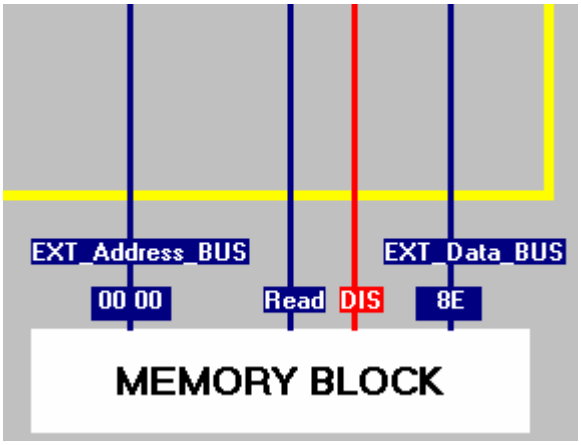

Figure 23 Memory 1

When user clicks on the Memory Block in SIM68TT11 the internal view of the memory opens in another window, which is shown in Figure 24 and Figure 25. User can follow the memory contents from this window. Memory window has two forms. First one is shown in Figure 24 and has a form of 255 by 255 matrix. Columns of this memory matrix form the upper byte of a memory address while the rows form the lower byte. The address can be reached by writing the column and row values successively. The internal view of the memory is functional in all modes of operation. Second memory mode works in the same with the first one, only difference is the size of the matrix. Second one has a 1 by 65535 matrix. Both memory windows give user the opportunity to find and view any location in the memory by 'FIND' button.

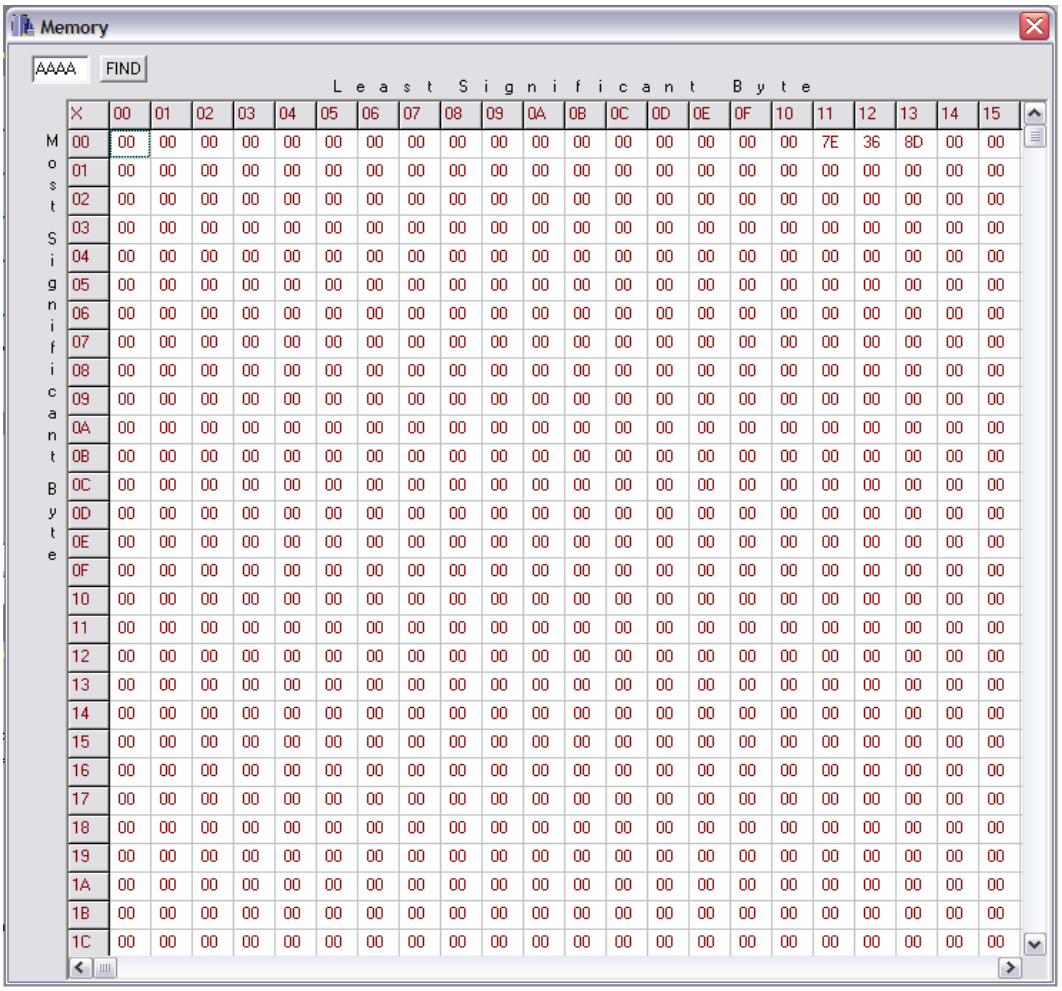

Figure 24 Memory 2

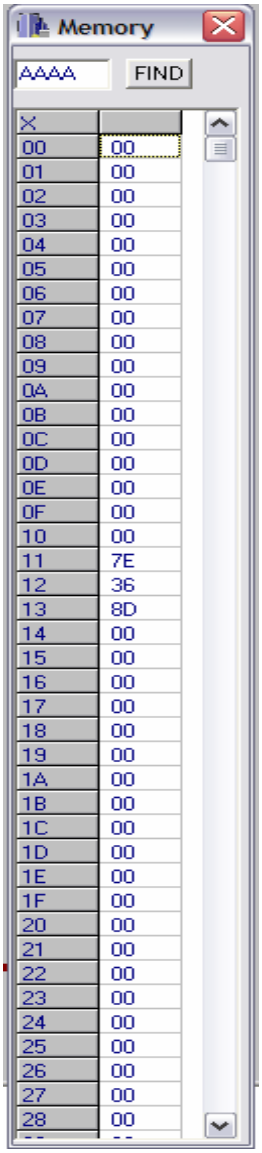

Figure 25 Memory 3

4.2.1.2. FLOW CHART

A flow chart helps the user to understand the operation of the micro controller. Figure 26 shows a sample of this flow chart. Flow chart is displayed only in “Cycle by Cycle” mode. The arrows turn in to red from white while the program progresses.

The user can follow the flow of the program with the help of these arrows on the flow chart.

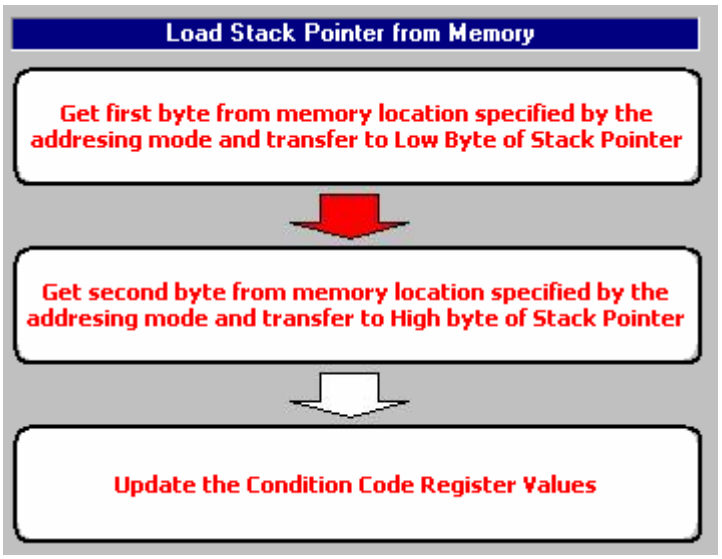

Figure 26 Flowchart

4.2.2. MENU BAR

Menu bar contains menu buttons. These buttons provides user to load a program to memory, set running modes and run a stored program. Functions of these buttons are explained in following sections.

4.2.2.1. LOAD BUTTON

To run a program on the microcontroller user must load an object code to the memory of the simulator. To load a file into the memory user should use the Load

button under the File button as in Figure 27. When Load button is clicked, the file dialog in Figure 28 is displayed. User can choose a file to load into the memory.

There are two formats supported by the simulator these are ‘OUX’ and ’S19’ formats. ‘OUX’ is the output of the ASM68TT11, which is developed and implemented as a part of this thesis study. ‘S19’ format is Motorola’s standard format for micro controllers and is the output of AS11 cross-assembler. Motorola defined 3 format types to adapt to the growing hunger for memory. The basic format is the so called S19 format, which has a 16 bit address field and can be used for files of up to 64kb.

Figure 27 Load Button

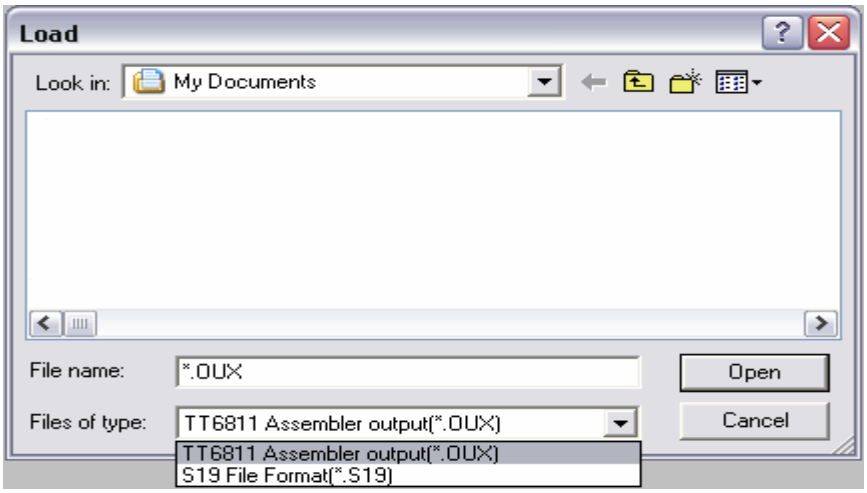

Figure 28 Load Menu

4.2.2.2. OPTIONS

From the options menu user can select the run mode of the simulator, step type for the ‘step by step’ mode and continuous mode’s running speed. Figure 29 shows the options menu.

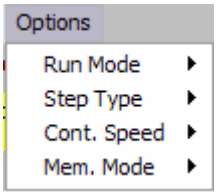

Figure 29 Options

4.2.2.2.1. RUN MODE

The simulator runs in two basic modes. First one is the ‘continuous mode’; in this mode, the simulator executes the loaded code without stopping until the user gives a ‘STOP’ command. Second running mode of the simulator is ‘step by step’ mode. In this mode, the simulator executes the loaded code part by part, and between these parts, waits for the user to tell the simulator go on. The parts in this mode are defined by two options; ‘instruction by instruction’ and ‘cycle by cycle’. The ‘instruction by instruction’ option executes a single instruction at a time. The ‘cycle by cycle’ option executes a cycle of a specific instruction at a time. The cycle is not same as the E_Cycle given in the Motorola’s reference documents. The cycle here is each step of the program progressed after user clicks FWD button and is not same with the E_Cycle information. Figure 30 shows the Run Mode option of SIM68TT11.

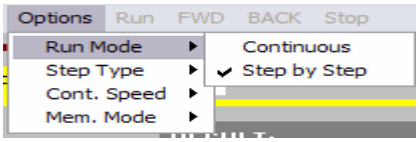

Figure 30 Run Modes

4.2.2.2.2. STEP TYPE

From this menu user can select the step type of the “Step by Step” mode. If “Cycle by Cycle” is, selected SIM68TT11 executes a cycle in each step. If “Instr. By Instr. “is selected SIM68TT11 executes an instruction in each step Figure 31 shows the Step Type option of SIM68TT11.

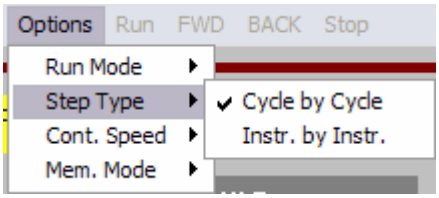

Figure 31 Step Type

4.2.2.2.3. CONTINUOUS MODE SPEED

In continuous mode, the simulator executes the loaded code without stopping until the user gives a ‘STOP’ command. This option determines the speed of this mode.
If “1 Instr. Per Sec.” is selected then one instruction will be executed in each second. If user selects “Fast”, the speed of SIM68TT11 is the top speed that is allowed by the computer, which SIM68TT11 runs on. Figure 32 shows the Cont. Speed option of SIM68TT11.

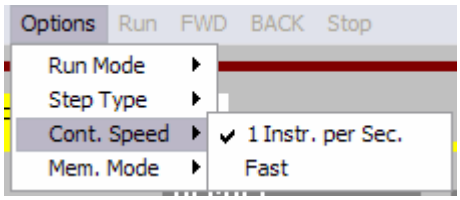

Figure 32 Continuous Mode Speed

4.2.2.2.4. MEMORY MODE

Memory Mode selects the type of the memory window. Memory window shows the contents of the memory to the user. It has two modes '255x255' or '1x65535'. These modes define the size of the memory grid. Figure 32 shows the memory mode options.

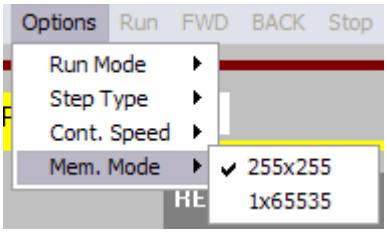

Figure 33 Memory Mode

4.2.2.3. RUN, FWD, BACK and STOP

The RUN button is used to start a program, which is loaded to memory. FWD button provides forward processing during program run. BACK button is for backward processing. BACK button works only in "Cycle by Cycle" mode of "Step by Step"

mode. When this button is pressed, it turns the program to the first operation of the current instruction that the SIM68TT11 executes.

4.3. INVOCATION OF THE SIMULATOR

The C++ programming language is used for the development and implementation of the simulator. The program runs on Windows 98 or higher operating systems. Screen resolution should be at least 1152 by 864 pixels for a better and whole view of SIM68TT11 main window. To run the simulator correctly user must save the "SIM68TT11.exe", "movs" folder and "pix" folder in the same folder. Otherwise, the program cannot find the related pictures and animations used in the simulator.

After the needed files are copied to the same directory and folder. User can start the program by executing the SIM68TT11.exe. When the simulator runs the main window in Figure 16 is displayed. To run a code in the simulator user must load the code using the LOAD button in the menu bar. User should select the running mode and related topics from the options menu. After loading an object code SIM68TT11 activates the RUN button. Pressing the Run button initializes the microcomputer and starts the program. According to the selected mode, SIM68TT11 executes the code in the memory. If the mode is, "Step by Step" user must use the FWD and BACK button to progress through the stored program. BACK button works only in the 'Cycle by Cycle' mode and turns the program to the first step of the execution phase of the current instruction. While the code is being executed by, the SIM68TT11 user can follow the program from the information boxes introduced in previous chapters.

During the program executing user can use 'F' key from the keyboard instead of FWD button and 'B' key from the keyboard instead of BACK button. User can also display the memory space by clicking on the memory block. There are also info and help boxes in the program. If user points a label by mouse, an info box appears as in Figure 34. Info boxes give user brief information about the related item. If user wants

detailed information, he/she should click on the item to display the help box as in Figure 35.

The SIM68TT11 executable file and it's user manual is given in a CD-ROM in Appendix D.

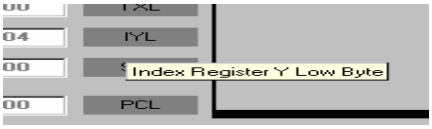

Figure 34 Info Box

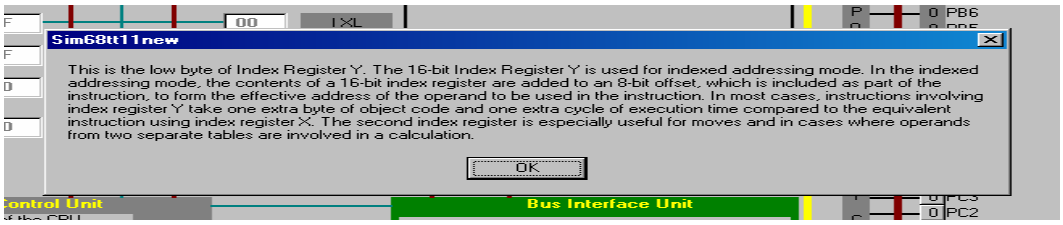

Figure 35 Help Box

CHAPTER 5

CONCLUSIONS

The aim of this thesis study is to develop a simulator toolkit for Motorola's 8-bit micro controller MC6811. The toolkit contains a cross-assembler, ASM68TT11, to obtain object code from the source code and a simulator, SIM68TT11, to run the object code. The code generated by using the Motorola Assembly Language rules is translated in to machine code by the cross-assembler, so the simulator can run the code.

The cross-assembler runs in command prompt and produces the object code for the simulator. Each line of the source code is traced and translated into related object code. The output of the cross-assembler is loaded to the simulator program. The simulator executes the code as the MC6811 does. The registers, address and data busses inside the CPU can be observed from the Graphical User Interface as well as the interrupts, I/O ports of the micro controller.

The simulator has two running modes. Continuous modes execute the code continuously until user stops the program. 'Step by step' mode executes the code either 'instruction by instruction' or 'cycle by cycle'. During these phases, the Graphical User Interface displays related hardware of the micro controller.

SIM68TT11 and ASM68TT11 are verified by comparing the outputs of the programs by desired results. Motorola manuals give the hex code related with an instruction and define the execution of same instruction. So the desired results are driven from these knowledge and compared manually.

The toolkit is designed to teach the user every basic micro controller operation. So it can be used for educational purposes. By the help of the Graphical User Interface a student, even with too little information about micro controllers, can understand the working principles of the microcomputer and test his/her knowledge.

On the other hand, simulator's continuous and instruction by instruction running modes help advanced users to develop and test their software.

Appendix D includes a CD-ROM with SIM68TT11 and ASM68TT11 executable files and user manuals.

Both ASM68TT11 and SIM68TT11 have some parts to be improved as a future work. The ASM68TT11 cross assembler can only understand basic assembly language commands. ASM68TT11 can be developed to understand macro program structures such as 'begin' and 'end'. SIM68TT11 does not include the all the functions of the MC6811 micro controller. The Analog to Digital Converter, Serial Communication Interface, and Serial Peripheral Interface and Real Time Interrupt systems can be implemented to the simulator as a further work. In addition, some peripheral devices such as LCD, keyboard or a device to test and understand the serial communication issues can be included to the toolkit for better understanding of the micro controller operations in a system.

By the help of this thesis study, a new user can learn much about the micro controllers and advanced users can use the simulator toolkit to develop their software. Therefore, this simulator toolkit is useful in both educational and research areas.

REFERENCES

[1] Peter SPASOV, Microcontroller Technology: The 68HC11, Prentice-Hall, 2002.

[2] Ronald J. TOCCI, Frank J. AMBROSIO, Microprocessors and Microcomputers Hardware and Software, Prentice-Hall, 2003.

[3] William C. Wray, Joseph D. Greenfield, Ross T. Bannatyne, Using Microprocessors and Microcomputers The Motorola Family, Prentice-Hall, 1999

[4] Nicolai M. JOSUTTIS, The C++ Standard Library A Tutorial and Handbook, Addison-Wesley, 1999

[5] Stephen PRATA, C++ Primer Plus, SAMS,2001

[6] I. Scott MACKENZIE, The 68000 Microprocessor, Prentice-Hall, 1995

[7] Motorola Inc., M68HC11E/D HC11 MC6811 E Series Technical Data, 1995

[8] Motorola Inc., M68HC11RM/AD HC11 MC6811 Reference Manual, 1995

[9] Prof.Dr. E. SCHWARTZ, Prof.Dr A. Arroyo, Assembly Language Notes, 1998

[10] John J. DONOVAN, Systems Programming, McGraw-Hill Book Company, 1972

[11] D. W. BARRON, Assemblers and Loaders, MacDonald Co., 1969

[12] Gene H. MILLER, Microcomputer Engineering, Prentice-Hall, 1998

[13] Kenneth HINTZ, Daniel TABAK, Microcontrollers: Architecture, Implementation & Programming, McGraw-Hill, 1992

APPENDIX A: DATA TYPES AND SUBROUTINES OF SIM68TT11

A.1. DATA TYPES

This section of the appendix deals with the enumerated data types. Gives the name of the data types and their brief explanation.

A.1.1. SregistreS

The following line shows the definition of the SregisterS and their names.

```
enum SregisterS {ACC_A, ACC_B, PC_H, PC_L, SPH, SPL, IXH, IXL, IYH, IYL, IR, DRH, DRL, MARH, MARL, MBR, CCR, ALUN};
```

SregisterS defines the names of the registers used in the Central Processing Unit of the SIM68TT11. SregisterS tells the functions which register will be used durng the processing of an instruction.

A.1.2. InxRegs

The following line shows the definition of the InxRegs and their names.

```
enum InxRegs {INX,INY};
```

InxRegs defines the index registers in the Central Processing Unit of the SIM68TT11. InxRegs tells the functions, which index register, will be used during the processing of an instruction with indexed addressing mode.

A.1.3. AddressModes

The following line shows the definition of the AddressModes and their names.

```
enum AddressModes {IMM,DIR,EXT,IND,REL,INH};
```

AddressModes defines the addressing mode of an instruction. According to this addressing mode subroutines processes the instruction.

A.1.4. whbranch

The following line shows the definition of the whbranch and their names.

```
enum whbranch  {BCC, BCLR, BCS, BEQ, BGE, BGT, BHI, BHS, BLE, BLO, BLS, BLT, BMI, BNE, BPL, BRA, BRCLR,
BRN, BRSET, BSR, BVC, BVS, BSET, JSR};
```

whbranch defines which branch condition will be checked for a branch instruction.

A.1.5. CCRclear

The following line shows the definition of the CCRclear and their names.

```
enum CCRclear  {CLC,CLI,CLV,CLR} ;
```

CCRclear defines which one of the Condition Code Register registers will be processed by the instruction.

A.1.6. shift

The following line shows the definition of the shift and their names.

```
enum shift {right,left};
```

shift defines the direction of a shift instruction.

A.1.7. arith

The following line shows the definition of the arith and their names.

```
enum arith {DECR, INCR, COMP, NEG, ROL, ROR, MUL, SUB, TOP, TST, LSL, LSR, ASR, FDIV, IDIV};
```

arith defines the arithmetic operation for an instruction.

A.1.8. logic

The following line shows the definition of the logic and their names.

```
enum logic {AND,ORA,EOR,SET,CLE};
```

logic defines the logic operation for an instruction

A.2. SUBROUTINES

This section of the appendix explains the subroutines in the SIM68TT11. Gives their definitions and discusses functions of them. The functions of the below subroutines are taken from cycle information of the instructions from [8].

A.2.1. searchInterrupt

```
bool searchInterrupt();
```

This subroutine checks the interrupts sources for any demand. If there is an interrupt demending a service the subroutine checks if the interrupt is allowed or not. If the interrupt is allowed then the subroutine prepares the CPU for interrupt subroutine. It stores the registers in stack, sets the flag to avoid furhter interrupts and fetch the related interrupt vector for the related interrupt subroutine. The searchInterrupt has

no inputs but a boolean output tells the main program an interrupt routine has been started to run.

A.2.2. exec_SABA

```
void exec_SABA(arith);
```

This subroutine executes the ABA and SBA instructions. Has no output but needs to know which operation will be executed between the two Accumulators. So it takes an arith type input.

A.2.3. execLDA_AB

```
void execLDA_AB(AddressModes ,SregisterS , InxRegs );
```

This subroutine executes the LDAA and LDAB instructions. Has no output but needs to know the addressing mode and the Accumulator will be loaded. Therefore, it takes an AddressModes, SregisterS and InxRegs type inputs.

A.2.4. execSTA_AB

```
void execSTA_AB(AddressModes ,SregisterS , InxRegs );
```

This subroutine executes the STAA and STAB instructions. Has no output but needs to know the addressing mode and the Accumulator will be stored. Therefore, it takes an AddressModes, SregisterS and InxRegs type inputs.

A.2.5. exec_ABXY

```
void exec_ABXY (InxRegs);
```

This subroutine executes the ABX and ABY instructions. Has no output but needs to know which index register will be used. Therefore, it takes and InxRegs type inputs.

A.2.6. exec_ADCAB

```
void exec_ADCAB (SregisterS , int , AddressModes , InxRegs ,arith );
```

This subroutine executes the ADCA, ADDA, SBCA, SUBA, ADCB, ADDB, SUBB and SBCB instructions. Has no output but needs to know the addressing mode, the Accumulator will be loaded, operation between the Accumulator and memory and to use carry bit or not. Therefore, it takes an AddressModes, SregisterS, arith and InxRegs type inputs.

A.2.7. exec_ADDD

```
void exec_ADDD (SregisterS , AddressModes , InxRegs,arith);
```

This subroutine executes the ADDD and SUBD instructions. Has no output but needs to know the addressing mode, operation between the Accumulator and memory. Therefore, it takes an AddressModes, SregisterS, arith and InxRegs type inputs.

A.2.8. exec_logic

```
void exec_logic (SregisterS ,AddressModes , InxRegs ,logic );
```

This subroutine executes the ANDA, ORA, EORA, ANDB, ORB and EORB instructions. Has no output but needs to know the addressing mode and the operation. Therefore, it takes an AddressModes, SregisterS, logic and InxRegs type inputs.

A.2.9. exec_ASLRD

```
void exec_ASLRD (shift );
```

This subroutine executes the LSLD and LSRD instructions. Has no output but needs to know the direction of the shift. Therefore, it takes a shift type input.

A.2.10. exec_BRCH

```
void exec_BRCH (whbranch );
```

This subroutine executes all of the branch instructions except BRCLR and BRSET. Has no output but needs to know the branching condition. Therefore, it takes a whbranch type input.

A.2.11. exec_BIT

```
void exec_BIT (SregisterS ,AddressModes , InxRegs );
```

This subroutine executes the BITA and BITB instructions. Has no output but needs to know the addressing mode and the Accumulator. Therefore, it takes an AddressModes, SregisterS, and InxRegs type inputs.

A.2.12. exec_CMP

```
void exec_CMP (SregisterS ,AddressModes , InxRegs );
```

This subroutine executes the CMPA and CMPB instructions. Has no output but needs to know the addressing mode and the Accumulator. Therefore, it takes an AddressModes, SregisterS, and InxRegs type inputs.

A.2.13. exec_CBA

```
void exec_CBA( );
```

This subroutine executes the CBA instructions. Has no output and input.

A.2.14. exec_CLCIV

```
void exec_CLCIV (CCRclear ,logic);
```

This subroutine executes the CLC, CLI, CLV, SEC, SEI and SEV instructions. Has no output but needs to know the operation and which bit of the Condition Code Register will be used. Therefore, it takes a CCRclear and logic type inputs.

A.2.15. execCLEAR

```
void execCLEAR (SregisterS ,AddressModes ,InxRegs );
```

This subroutine executes the CLRA, CLRB and CLR instructions. Has no output but needs to know the addressing mode and the Accumulator. Therefore, it takes an AddressModes, SregisterS, and InxRegs type inputs.

A.2.16. exec_ARITH

```
void exec_ARITH (SregisterS ,AddressModes ,InxRegs ,arith );
```

This subroutine executes the COMA, DECA, INCA, NEGA, ROLA, RORA, LSLA, LSRA, TSTA, ASRA, COMB, DECB, INCB, NEGB, ROLB, RORB, LSLB, LSRB, TSTB, ASRB, COM, DEC, INC, NEG, ROL, ROR, LSL, LSR, TST and ASR instructions. Has no output but needs to know the addressing mode, the operation and the Accumulator. Therefore, it takes an AddressModes, SregisterS, arith and InxRegs type inputs.

A.2.17. exec_LOAD

```
void exec_LOAD (SregisterS ,AddressModes ,InxRegs  );
```

This subroutine executes the LDS, LDD, LDX and LDY instructions. Has no output but needs to know the addressing mode and the register. Therefore, it takes an AddressModes, SregisterS and InxRegs type inputs.

A.2.18. exec_STORE

```
void exec_STORE (SregisterS ,AddressModes ,InxRegs  );
```

This subroutine executes the STS, STD, STX and STY instructions. Has no output but needs to know the addressing mode and the register. Therefore, it takes an AddressModes, SregisterS and InxRegs type inputs.

A.2.19. exec_PUSH

```
void exec_PUSH (SregisterS,int);
```

This subroutine executes the PSHA, PSHB, PSHX and PSHY instructions. Has no output but needs to know the register. Therefore, it takes a SregisterS type input.

A.2.20. exec_PULL

```
void exec_PULL (SregisterS,int);
```

This subroutine executes the PULA, PULB, PULX and PULY instructions. Has no output but needs to know the register. Therefore, it takes a SregisterS type input.

A.2.21. exec_RTI

```
void exec_RTI ( );
```

This subroutine executes the RTI instruction. Has no output and input.

A.2.22. exec_JUMP

```
void exec_JUMP (AddressModes,InxRegs);
```

This subroutine executes the JMP instruction. Has no output but needs to know the addressing mode. Therefore, it takes an AddressModes and InxRegs type inputs.

A.2.23. exec_TABvC

```
void exec_TABvC (SregisterS);
```

This subroutine executes the TAB and TAP instructions. Has no output but needs to know the register. Therefore, it takes a SregisterS type input.

A.2.24. exec_TBvCA

```
void exec_TBvCA (SregisterS);
```

This subroutine executes the TBA and TPA instructions. Has no output but needs to know the register. Therefore, it takes a SregisterS type input.

A.2.25. exec_CPD

```
void exec_CPD (SregisterS,AddressModes,InxRegs);
```

This subroutine executes the CPD, CPX and CPY instructions. Has no output but needs to know the addressing mode and the register. Therefore, it takes an AddressModes, SregisterS and InxRegs type inputs.

A.2.26. exec_arithSXY

```
void exec_arithSXY (SregisterS,arith);
```

This subroutine executes the DES, DEX, DEY, INS, INX and INY instructions. Has no output but needs to know the operation and the register. Therefore, it takes an arith and SregisterS type inputs.

A.2.27. exec_JSR

```
void exec_JSR (AddressModes,InxRegs,whbranch );
```

This subroutine executes the JSR instruction. Has no output but needs to know the addressing mode. Therefore, it takes an AddressModes and InxRegs type inputs.

A.2.28. exec_RSR

```
void exec_RSR ( );
```

This subroutine executes the RSR instruction. Has no output and inputs.

A.2.29. exec_SWI

```
void exec_SWI( );
```

This subroutine executes the SWI instruction. Has no output and input.

A.2.30. exec_TSXYS

```
void exec_TSXYS(SregisterS,SregisterS);
```

This subroutine executes the TSX, TSY, TXS and TYS instructions. Has no output but needs to know the destination and departure registers. Therefore, it takes two SregisterS type inputs.

A.2.31. exec_XGDXY

```
void exec_XGDXY (InxRegs );
```

This subroutine executes the XGDX and XGDY instructions. Has no output but needs to know the destination and departure registers. Therefore, it takes an InxRegs type input.

A.2.32. illegalOPCODE

```
void illegalOPCODE ( );
```

This subroutine is the illegal opcode trap. When an illegal opcode is fetched, this routine is called by the main routine. And fetches the illegal opcode vector from memory. This starts the illegal opcode routine from the main memory of the stored program. Has no output and input.

A.2.33. exec_DAA

```
void exec_DAA ( );
```

This subroutine executes the DAA instruction. Has no output or inputs.

A.2.34. exec_NOP

```
void exec_NOP ();
```

This subroutine executes the NOP instruction. Has no output or inputs.

A.2.35. exec_BRCS

```
void exec_BRCS (logic ,AddressModes ,InxRegs);
```

This subroutine executes the BRSET and BRCLR instructions. Has no output but needs to know the addressing mode, set or clear memory bits. Therefore, it takes an InxRegs, AddressModes and logic type inputs.

A.2.36. exec_BCLSE

```
void exec_BCLSE(logic ,AddressModes ,InxRegs);
```

This subroutine executes the BSET and BCLR instructions. Has no output but needs to know the addressing mode, set or clear memory bits. Therefore, it takes an InxRegs, AddressModes and logic type inputs.

A.2.37. exec_WAI

```
void exec_WAI( );
```

This subroutine executes the WAI instruction. Has no output or inputs.

A.2.38. exec_FIDIV

```
void exec_FIDIV (arith );
```

This subroutine executes the FDIV and IDIV instructions. . Has no output but needs to know the division type. Therefore, it takes an arith type input.

A.2.39. exec_MUL

```
void exec_MUL ( );
```

This subroutine executes the MUL instruction. Has no output or inputs.

APPENDIX B: VERIFICATION OF ASM68TT11 AND SIM68TT11

This appendix proves that the cross-assembler and the simulator are working correctly. First part of the appendix gives DEN output of the ASM68TT11 as a proof. In this output format source code can be seen with the related hex output. All the outputs are verified according to [8]. Second part of the appendix gives a special file delivered by the SIM68TT11. This file is not available in the current version of the simulator. It shows all the register and related memory contents of the SIM68TT11during the execution of the code in first part of the appendix. The code used to test the software is not a meaningful code but contains all the possible instructions for MC6811. All values in both sections are checked manually.

B.1. verification ASM68TT11

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00001	->	0000	->				
Line:	00002	->	0000	->		ZAL2	EQU	@53777
Line:	00003	->	0000	->		ZAL3	EQU	$3C
Line:	00004	->	0000	->		ZAL4	EQU	128
Line:	00005	->	0000	->		ZAL1	EQU	%10011011
Line:	00006	->	0000	->		.atl	EQU	$3599
Line:	00007	->	0000	->		_der	EQU	$35D6
Line:	00008	->	0000	->			ORG	211
Line:	00009	->	00d3	->	7e 36 8d		JMP	%11011010001101
Line:	00010	->	00d6	->				
Line:	00011	->	00d6	->				
Line:	00012	->	00d6	->			ORG	$FFFE
Line:	00013	->	fffe	->	01 01		FDB	$34ff
Line:	00014	->	0000	->			ORG	$FFF6
Line:	00015	->	fff6	->	37 d3		FDB	$37d3

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00016	->	fff8	->			ORG	$FFF2
Line:	00017	->	fff2	->	37 e8		FDB	$37e8
Line:	00018	->	fff4	->			ORG	$FFF4
Line:	00019	->	fff4	->	37 e8		FDB	$37e8
Line:	00020	->	fff6	->			ORG	$FFF8
Line:	00021	->	fff8	->	37 91		FDB	$3791
Line:	00022	->	fffa	->				
Line:	00023	->	fffa	->				
Line:	00024	->	fffa	->			ORG	$34FF
Line:	00025	->	34ff	->				
Line:	00026	->	34ff	->				
Line:	00027	->	34ff	->	86 a3		LDAA	#$A3
Line:	00028	->	3501	->	c6 5d		LDAB	#$5D
Line:	00029	->	3503	->	ce 77 ee		LDX	#$77EE
Line:	00030	->	3506	->	18 ce ab cd		LDY	#$ABCD
Line:	00031	->	350a	->	8e f4 35		LDS	#$F435
Line:	00032	->	350d	->				
Line:	00033	->	350d	->	1b	AAA1	ABA	
Line:	00034	->	350e	->	3a	AAA2	ABX	
Line:	00035	->	350f	->	18 3a	AAA3	ABY	
Line:	00036	->	3511	->	89 36	AAA4	ADCA	#54
Line:	00037	->	3513	->	99 56	AAA5	ADCA	$56
Line:	00038	->	3515	->	99 11	AAA6	ADCA	%10001
Line:	00039	->	3517	->	a9 22	AAA7	ADCA	$22,X
Line:	00040	->	3519	->	18 a9 80	AAA8	ADCA	ZAL4,Y
Line:	00041	->	351c	->	c9 36	AAA9	ADCB	#54
Line:	00042	->	351e	->	d9 5f	AAA10	ADCB	$5E+@1
Line:	00043	->	3520	->	f9 58 fe	AAA11	ADCB	ZAL2+255
Line:	00044	->	3523	->	e9 6b	AAA12	ADCB	$1D+78,X
Line:	00045	->	3525	->	18 e9 6b	AAA13	ADCB	$1D+78,Y
Line:	00046	->	3528	->	8b 36	BBB14	ADDA	#54
Line:	00047	->	352a	->	9b 5f	BBB15	ADDA	$5E+@1
Line:	00048	->	352c	->	bb 58 fe	BBB16	ADDA	ZAL2+255
Line:	00049	->	352f	->	ab 6b	BBB17	ADDA	$1D+78,X
Line:	00050	->	3531	->	18 ab 6b	BBB18	ADDA	$1D+78,Y
Line:	00051	->	3534	->	cb 36	BBB19	ADDB	#54
Line:	00052	->	3536	->	db 5f	BBB20	ADDB	$5E+@1
Line:	00053	->	3538	->	fb 58 fe	BBB21	ADDB	ZAL2+255
Line:	00054	->	353b	->	eb 6b	BBB22	ADDB	$1D+78,X
Line:	00055	->	353d	->	18 eb 6b	BBB23	ADDB	$1D+78,Y
Line:	00056	->	3540	->	c3 00 36	CCC24	ADDD	#54
Line:	00057	->	3543	->	d3 5f	CCC25	ADDD	$5E+@1
Line:	00058	->	3545	->	f3 58 fe	CCC26	ADDD	ZAL2+255
Line:	00059	->	3548	->	e3 6b	CCC27	ADDD	$1D+78,X
Line:	00060	->	354a	->	18 e3 6b	CCC28	ADDD	$1D+78,Y
Line:	00061	->	354d	->	84 56	CCC29	ANDA	#$56
Line:	00062	->	354f	->	94 5f	CCC30	ANDA	$5E+@1
Line:	00063	->	3551	->	b4 58 fe	CCC31	ANDA	ZAL2+255
Line:	00064	->	3554	->	a4 6b	CCC32	ANDA	$1D+78,X

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00065	->	3556	->	18 a4 6b	CCC33	ANDA	$1D+78,Y
Line:	00066	->	3559	->	c4 56	CCC34	ANDB	#$56
Line:	00067	->	355b	->	d4 5f	CCC35	ANDB	$5E+@1
Line:	00068	->	355d	->	f4 58 fe	DDD36	ANDB	ZAL2+255
Line:	00069	->	3560	->	e4 6b	DDD37	ANDB	$1D+78,X
Line:	00070	->	3562	->	18 e4 6b	DDD38	ANDB	$1D+78,Y
Line:	00071	->	3565	->	78 58 fe	DDD39	ASL	ZAL2+255
Line:	00072	->	3568	->	68 6b	DDD40	ASL	$1D+78,X
Line:	00073	->	356a	->	18 68 6b	DDD41	ASL	$1D+78,Y
Line:	00074	->	356d	->	48	DDD42	ASLA	
Line:	00075	->	356e	->	58	DDD43	ASLB	
Line:	00076	->	356f	->	05	DDD44	ASLD	
Line:	00077	->	3570	->	77 03 3a	DDD45	ASR	765+@75
Line:	00078	->	3573	->	67 6b	DDD46	ASR	$1D+78,X
Line:	00079	->	3575	->	18 67 6b	DDD47	ASR	$1D+78,Y
Line:	00080	->	3578	->	47	DDD48	ASRA	
Line:	00081	->	3579	->	57	DDD49	ASRB	
Line:	00082	->	357a	->	24 00	DDD50	BCC	DDD51
Line:	00083	->	357c	->	15 5f 33	DDD51	BCLR	$5E+@1:$33
Line:	00084	->	357f	->	1d 6b 37	EEE52	BCLR	$1D+78,X:55
Line:	00085	->	3582	->	18 1d 6b 12	EEE53	BCLR	$1D+78,Y:@22
Line:	00086	->	3586	->	25 00	EEE54	BCS	EEE55
Line:	00087	->	3588	->	27 00	EEE55	BEQ	EEE56
Line:	00088	->	358a	->	2c 00	EEE56	BGE	EEE57
Line:	00089	->	358c	->	2e 00	EEE57	BGT	EEE58
Line:	00090	->	358e	->	22 00	EEE58	BHI	EEE59
Line:	00091	->	3590	->	24 0e	EEE59	BHS	FFF66
Line:	00092	->	3592	->	85 56	EEE60	BITA	#$56
Line:	00093	->	3594	->	95 5f	EEE61	BITA	$5E+@1
Line:	00094	->	3596	->	b5 03 3a	EEE62	BITA	765+@75
Line:	00095	->	3599	->	a5 6b	EEE63	BITA	$1D+78,X
Line:	00096	->	359b	->	18 a5 6b	EEE64	BITA	$1D+78,Y
Line:	00097	->	359e	->	c5 56	EEE65	BITB	#$56
Line:	00098	->	35a0	->	d5 5f	FFF66	BITB	$5E+@1
Line:	00099	->	35a2	->	f5 03 3a	FFF67	BITB	765+@75
Line:	00100	->	35a5	->	e5 08	FFF68	BITB	%1001-@1,X
Line:	00101	->	35a7	->	18 e5 6b	FFF69	BITB	$1D+78,Y
Line:	00102	->	35aa	->	2f 00	FFF70	BLE	FFF71
Line:	00103	->	35ac	->	25 00	FFF71	BLO	FFF72
Line:	00104	->	35ae	->	23 00	FFF72	BLS	FFF73
Line:	00105	->	35b0	->	2d 00	FFF73	BLT	FFF74
Line:	00106	->	35b2	->	2b 00	FFF74	BMI	FFF75
Line:	00107	->	35b4	->	26 00	FFF75	BNE	FFF76
Line:	00108	->	35b6	->	2a 00	FFF76	BPL	FFF77
Line:	00109	->	35b8	->	20 00	FFF77	BRA	FFF78
Line:	00110	->	35ba	->	13 11 45 09	FFF78	BRCLR	%10110-5:$45:GGG81
Line:	00111	->	35be	->	1f 08 13 05	FFF79	BRCLR	%1001-@1,X:@23:GGG81
Line:	00112	->	35c2	->	18 1f 6b f0 00	GGG80	BRCLR	$1D+78,Y:%11110000:GGG81
Line:	00113	->	35c7	->	21 1f	GGG81	BRN	GGG93

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00114	->	35c9	->	12 11 ee 09	GGG82	BRSET	%10110-5:$EE:GGG85
Line:	00115	->	35cd	->	1e 08 11 05	GGG83	BRSET	%1001-@1,X:$11:GGG85
Line:	00116	->	35d1	->	18 1e 6b 00 00	GGG84	BRSET	$1D+78,Y:$00:GGG85
Line:	00117	->	35d6	->	14 11 c9	GGG85	BSET	%10110-5:%11001001
Line:	00118	->	35d9	->	1c 08 e1	GGG86	BSET	%1001-@1,X:$E1
Line:	00119	->	35dc	->	18 1c 08 58	GGG87	BSET	%1001-@1,Y:88
Line:	00120	->	35e0	->	8d 00	GGG88	BSR	GGG89
Line:	00121	->	35e2	->	28 00	GGG89	BVC	GGG90
Line:	00122	->	35e4	->	29 00	GGG90	BVS	GGG91
Line:	00123	->	35e6	->	11	GGG91	CBA	
Line:	00124	->	35e7	->	0c	GGG92	CLC	
Line:	00125	->	35e8	->	0e	GGG93	CLI	
Line:	00126	->	35e9	->	7f 03 3a	HHH94	CLR	765+@75
Line:	00127	->	35ec	->	6f 08	HHH95	CLR	%1001-@1,X
Line:	00128	->	35ee	->	18 6f 08	HHH96	CLR	%1001-@1,Y
Line:	00129	->	35f1	->	4f	HHH97	CLRA	
Line:	00130	->	35f2	->	5f	HHH98	CLRB	
Line:	00131	->	35f3	->	0a	HHH99	CLV	
Line:	00132	->	35f4	->	81 56	HHH100	CMPA	#$56
Line:	00133	->	35f6	->	91 11	HHH101	CMPA	%10110-5
Line:	00134	->	35f8	->	b1 03 3a	HHH102	CMPA	765+@75
Line:	00135	->	35fb	->	a1 08	HHH103	CMPA	%1001-@1,X
Line:	00136	->	35fd	->	18 a1 08	HHH104	CMPA	%1001-@1,Y
Line:	00137	->	3600	->	c1 56	HHH105	CMPB	#$56
Line:	00138	->	3602	->	d1 11	HHH106	CMPB	%10110-5
Line:	00139	->	3604	->	f1 03 3a	HHH107	CMPB	765+@75
Line:	00140	->	3607	->	e1 08	HHH108	CMPB	%1001-@1,X
Line:	00141	->	3609	->	18 e1 08	HHH109	CMPB	%1001-@1,Y
Line:	00142	->	360c	->	73 03 3a	HHH10	COM	765+@75
Line:	00143	->	360f	->	63 08	III111	COM	%1001-@1,X
Line:	00144	->	3611	->	18 63 08	III112	COM	%1001-@1,Y
Line:	00145	->	3614	->	43	III113	COMA	
Line:	00146	->	3615	->	53	III114	COMB	
Line:	00147	->	3616	->	1a 83 00 56	III115	CPD	#$56
Line:	00148	->	361a	->	1a 93 11	III116	CPD	%10110-5
Line:	00149	->	361d	->	1a b3 03 3a	III117	CPD	765+@75
Line:	00150	->	3621	->	1a a3 08	III118	CPD	%1001-@1,X
Line:	00151	->	3624	->	cd a3 08	III119	CPD	%1001-@1,Y
Line:	00152	->	3627	->	8c 00 11	III120	CPX	#%10001
Line:	00153	->	362a	->	9c 11	III121	CPX	%10110-5
Line:	00154	->	362c	->	bc 03 3a	III122	CPX	765+@75
Line:	00155	->	362f	->	ac 08	III123	CPX	%1001-@1,X
Line:	00156	->	3631	->	cd ac 08	III124	CPX	%1001-@1,Y
Line:	00157	->	3634	->	18 8c 00 11	III125	CPY	#%10001
Line:	00158	->	3638	->	18 9c 11	KKK126	CPY	%10110-5
Line:	00159	->	363b	->	18 bc 03 3a	KKK127	CPY	765+@75
Line:	00160	->	363f	->	1a ac 08	KKK128	CPY	%1001-@1,X
Line:	00161	->	3642	->	18 ac 08	KKK129	CPY	%1001-@1,Y
Line:	00162	->	3645	->	19	KKK130	DAA	

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00163	->	3646	->	7a 57 ff	KKK131	DEC	ZAL2
Line:	00164	->	3649	->	6a 3c	KKK132	DEC	ZAL3,X
Line:	00165	->	364b	->	18 6a 08	KKK133	DEC	%1001-@1,Y
Line:	00166	->	364e	->	4a	KKK134	DECA	
Line:	00167	->	364f	->	5a	KKK35	DECB	
Line:	00168	->	3650	->	34	KKK136	DES	
Line:	00169	->	3651	->	09	KKK137	DEX	
Line:	00170	->	3652	->	18 09	KKK38	DEY	
Line:	00171	->	3654	->	88 11	KKK39	EORA	#%10001
Line:	00172	->	3656	->	98 11	JJJ140	EORA	%10110-5
Line:	00173	->	3658	->	b8 57 ff	JJJ41	EORA	ZAL2
Line:	00174	->	365b	->	a8 3c	JJJ142	EORA	ZAL3,X
Line:	00175	->	365d	->	18 a8 08	JJJ143	EORA	%1001-@1,Y
Line:	00176	->	3660	->	c8 11	JJJ144	EORB	#%10001
Line:	00177	->	3662	->	d8 11	JJJ145	EORB	%10110-5
Line:	00178	->	3664	->	f8 57 ff	JJJ146	EORB	ZAL2
Line:	00179	->	3667	->	e8 3c	JJJ147	EORB	ZAL3,X
Line:	00180	->	3669	->	18 e8 08	JJJ148	EORB	%1001-@1,Y
Line:	00181	->	366c	->	03	JJJ149	FDIV	
Line:	00182	->	366d	->	02	JJJ150	IDIV	
Line:	00183	->	366e	->	7c 57 ff	JJJ151	INC	ZAL2
Line:	00184	->	3671	->	6c 3c	LLL152	INC	ZAL3,X
Line:	00185	->	3673	->	18 6c 08	LLL153	INC	%1001-@1,Y
Line:	00186	->	3676	->	4c	LLL154	INCA	
Line:	00187	->	3677	->	5c	LLL155	INCB	
Line:	00188	->	3678	->	31	LLL156	INS	
Line:	00189	->	3679	->	08	LLL157	INX	
Line:	00190	->	367a	->	18 08	LLL158	INY	
Line:	00191	->	367c	->	7e 36 7f	LLL159	JMP	$367f
Line:	00192	->	367f	->	ce 36 80	sawq1z	LDX	#$3680
Line:	00193	->	3682	->	6e 04	LLL160	JMP	$04,X
Line:	00194	->	3684	->	18 ce 36 80	saq1z	LDY	#@33200
Line:	00195	->	3688	->	18 6e 0b	LLL161	JMP	$0b,Y
Line:	00196	->	368b	->	9d d3	LLL162	JSR	$D3
Line:	00197	->	368d	->	bd 36 90	LLL63	JSR	13968
Line:	00198	->	3690	->	ad 12	LLL164	JSR	$12,X
Line:	00199	->	3692	->	18 ad 15	LLL65	JSR	$15,Y
Line:	00200	->	3695	->	86 11	MMM166	LDAA	#%10001
Line:	00201	->	3697	->	96 11	MMM167	LDAA	%10110-5
Line:	00202	->	3699	->	b6 57 ff	MMM68	LDAA	ZAL2
Line:	00203	->	369c	->	a6 3c	MMM169	LDAA	ZAL3,X
Line:	00204	->	369e	->	18 a6 3c	MMM170	LDAA	ZAL3,Y
Line:	00205	->	36a1	->	c6 11	MMM171	LDAB	#%10001
Line:	00206	->	36a3	->	d6 11	MMM172	LDAB	%10110-5
Line:	00207	->	36a5	->	f6 57 ff	MMM173	LDAB	ZAL2
Line:	00208	->	36a8	->	e6 3c	MMM174	LDAB	ZAL3,X
Line:	00209	->	36aa	->	18 e6 3c	MMM175	LDAB	ZAL3,Y
Line:	00210	->	36ad	->	cc 00 11	MMM176	LDD	#%10001
Line:	00211	->	36b0	->	dc 5a	MMM177	LDD	$5A

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00212	->	36b2	->	fc 56 80	MMM178	LDD	$5678+%1000
Line:	00213	->	36b5	->	ec 3c	MMM179	LDD	ZAL3,X
Line:	00214	->	36b7	->	18 ec 3c	MMM80	LDD	ZAL3,Y
Line:	00215	->	36ba	->	8e 00 12	MMM181	LDS	#@22
Line:	00216	->	36bd	->	9e 5a	MMM182	LDS	$5A
Line:	00217	->	36bf	->	be 56 80	MMM183	LDS	$5678+%1000
Line:	00218	->	36c2	->	ae 3c	NNN84	LDS	ZAL3,X
Line:	00219	->	36c4	->	18 ae 3c	NNN185	LDS	ZAL3,Y
Line:	00220	->	36c7	->	ce 00 12	NNN186	LDX	#@22
Line:	00221	->	36ca	->	de 5a	NNN87	LDX	$5A
Line:	00222	->	36cc	->	fe 56 80	NNN188	LDX	$5678+%1000
Line:	00223	->	36cf	->	ee 3c	NNN189	LDX	ZAL3,X
Line:	00224	->	36d1	->	cd ee 3c	NNN190	LDX	ZAL3,Y
Line:	00225	->	36d4	->	18 ce 00 12	NNN191	LDY	#@22
Line:	00226	->	36d8	->	18 de 5a	NNN192	LDY	$5A
Line:	00227	->	36db	->	18 fe 56 80	NNN193	LDY	$5678+%1000
Line:	00228	->	36df	->	1a ee 3c	NNN94	LDY	ZAL3,X
Line:	00229	->	36e2	->	18 ee 3c	NNN95	LDY	ZAL3,Y
Line:	00230	->	36e5	->	78 56 80	NNN196	LSL	$5678+%1000
Line:	00231	->	36e8	->	68 3c	NNN197	LSL	ZAL3,X
Line:	00232	->	36ea	->	18 68 3c	NNN198	LSL	ZAL3,Y
Line:	00233	->	36ed	->	48	NNN199	LSLA	
Line:	00234	->	36ee	->	58	OOO200	LSLB	
Line:	00235	->	36ef	->	05	OOO201	LSLD	
Line:	00236	->	36f0	->	74 56 80	OOO202	LSR	$5678+%1000
Line:	00237	->	36f3	->	64 3c	OOO203	LSR	ZAL3,X
Line:	00238	->	36f5	->	18 64 3c	OOO204	LSR	ZAL3,Y
Line:	00239	->	36f8	->	44	OOO205	LSRA	
Line:	00240	->	36f9	->	54	OOO206	LSRB	
Line:	00241	->	36fa	->	04	OOO207	LSRD	
Line:	00242	->	36fb	->	3d	OOO208	MUL	
Line:	00243	->	36fc	->	70 56 80	OOO209	NEG	$5678+%1000
Line:	00244	->	36ff	->	60 3c	OOO210	NEG	ZAL3,X
Line:	00245	->	3701	->	18 60 3c	OOO211	NEG	ZAL3,Y
Line:	00246	->	3704	->	40	PPP212	NEGA	
Line:	00247	->	3705	->	50	PPP213	NEGB	
Line:	00248	->	3706	->	01	PPP214	NOP	
Line:	00249	->	3707	->	8a 12	PPP215	ORAA	#@22
Line:	00250	->	3709	->	9a 5a	PPP216	ORAA	$5A
Line:	00251	->	370b	->	ba 56 80	PPP217	ORAA	$5678+%1000
Line:	00252	->	370e	->	aa 3f	PPP218	ORAA	ZAL3+3,X
Line:	00253	->	3710	->	18 aa 3c	PPP219	ORAA	ZAL3,Y
Line:	00254	->	3713	->	ca 12	PPP220	ORAB	#@22
Line:	00255	->	3715	->	da 5a	PPP221	ORAB	$5A
Line:	00256	->	3717	->	fa 56 80	PPP222	ORAB	$5678+%1000
Line:	00257	->	371a	->	ea 3f	PPP223	ORAB	ZAL3+3,X
Line:	00258	->	371c	->	18 ea 3c	PPP224	ORAB	ZAL3,Y
Line:	00259	->	371f	->	36	PPP225	PSHA	
Line:	00260	->	3720	->	37	PPP226	PSHB	

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00261	->	3721	->	3c	QQQ227	PSHX	
Line:	00262	->	3722	->	18 3c	QQQ228	PSHY	
Line:	00263	->	3724	->	32	QQQ229	PULA	
Line:	00264	->	3725	->	33	QQQ230	PULB	
Line:	00265	->	3726	->	38	QQQ231	PULX	
Line:	00266	->	3727	->	18 38	QQQ232	PULY	
Line:	00267	->	3729	->	79 56 80	QQQ233	ROL	$5678+%1000
Line:	00268	->	372c	->	69 3f	QQQ234	ROL	ZAL3+3,X
Line:	00269	->	372e	->	18 69 1c	QQQ235	ROL	ZAL3-4/2,Y
Line:	00270	->	3731	->	49	QQQ236	ROLA	
Line:	00271	->	3732	->	59	QQQ237	ROLB	
Line:	00272	->	3733	->	76 56 80	QQQ238	ROR	$5678+%1000
Line:	00273	->	3736	->	66 3f	QQQ239	ROR	ZAL3+3,X
Line:	00274	->	3738	->	18 66 1c	RRR240	ROR	ZAL3-4/2,Y
Line:	00275	->	373b	->	46	RRR241	RORA	
Line:	00276	->	373c	->	56	RRR242	RORB	
Line:	00277	->	373d	->	8e a0 00		LDS	#$A000
Line:	00278	->	3740	->	ce 37 49		LDX	#$3749
Line:	00279	->	3743	->	3c		PSHX	
Line:	00280	->	3744	->	36		PSHA	
Line:	00281	->	3745	->	3c		PSHX	
Line:	00282	->	3746	->	3c		PSHX	
Line:	00283	->	3747	->	3c		PSHX	
Line:	00284	->	3748	->	3b	RRR243	RTI	
Line:	00285	->	3749	->	ce 37 4e		LDX	#$374e
Line:	00286	->	374c	->	3c		PSHX	
Line:	00287	->	374d	->	39	RRR244	RTS	
Line:	00288	->	374e	->	10	RRR245	SBA	
Line:	00289	->	374f	->	82 80	RRR246	SBCA	#ZAL4
Line:	00290	->	3751	->	92 9b	RRR247	SBCA	ZAL1
Line:	00291	->	3753	->	b2 56 80	RRR248	SBCA	$5678+%1000
Line:	00292	->	3756	->	a2 3f	RRR249	SBCA	ZAL3+3,X
Line:	00293	->	3758	->	18 a2 1c	RRR250	SBCA	ZAL3-4/2,Y
Line:	00294	->	375b	->	c2 80	RRR251	SBCB	#ZAL4
Line:	00295	->	375d	->	d2 9b	RRR252	SBCB	ZAL1
Line:	00296	->	375f	->	f2 ab cd	SSS253	SBCB	$ABCD
Line:	00297	->	3762	->	e2 3f	SSS254	SBCB	ZAL3+3,X
Line:	00298	->	3764	->	18 e2 1c	SSS255	SBCB	ZAL3-4/2,Y
Line:	00299	->	3767	->	0d	SSS256	SEC	
Line:	00300	->	3768	->	0f	SSS257	SEI	
Line:	00301	->	3769	->	0b	SSS258	SEV	
Line:	00302	->	376a	->	97 9b	SSS259	STAA	ZAL1
Line:	00303	->	376c	->	b7 ab cd	SSS260	STAA	$ABCD
Line:	00304	->	376f	->	a7 3f	SSS261	STAA	ZAL3+3,X
Line:	00305	->	3771	->	18 a7 1c	SSS262	STAA	ZAL3-4/2,Y
Line:	00306	->	3774	->	d7 9b	SSS263	STAB	ZAL1
Line:	00307	->	3776	->	f7 ab cd	SSS264	STAB	$ABCD
Line:	00308	->	3779	->	e7 3f	TTT265	STAB	ZAL3+3,X
Line:	00309	->	377b	->	18 e7 1c	TTT266	STAB	ZAL3-4/2,Y

	Line Number		Progran Counter		Hex Code			Source Code
Line:	00310	->	377e	->	dd 9e	TTT267	STD	ZAL1+%011
Line:	00311	->	3780	->	fd ab cd	TTT268	STD	$ABCD
Line:	00312	->	3783	->	ed 3f	TTT269	STD	ZAL3+3,X
Line:	00313	->	3785	->	18 ed 1c	TTT270	STD	ZAL3-4/2,Y
Line:	00314	->	3788	->	86 ff		LDAA	#$FF
Line:	00315	->	378a	->	06		TAP	
Line:	00316	->	378b	->	cf	TTT271	STOP	
Line:	00317	->	378c	->	9f 9e	TTT272	STS	ZAL1+%011
Line:	00318	->	378e	->	bf ab cd	TTT273	STS	$ABCD
Line:	00319	->	3791	->	af 3f	TTT274	STS	ZAL3+3,X
Line:	00320	->	3793	->	18 af 1c	UUU275	STS	ZAL3-4/2,Y
Line:	00321	->	3796	->	df 9e	UUU276	STX	ZAL1+%011
Line:	00322	->	3798	->	ff ab cd	UUU277	STX	$ABCD
Line:	00323	->	379b	->	ef 3f	UUU278	STX	ZAL3+3,X
Line:	00324	->	379d	->	cd ef 1c	UUU279	STX	ZAL3-4/2,Y
Line:	00325	->	37a0	->	18 df 9e	UUU280	STY	ZAL1+%011
Line:	00326	->	37a3	->	18 ff ab cd	UUU281	STY	$ABCD
Line:	00327	->	37a7	->	1a ef 3f	UUU282	STY	ZAL3+3,X
Line:	00328	->	37aa	->	18 ef 1c	UUU283	STY	ZAL3-4/2,Y
Line:	00329	->	37ad	->	80 80	UUU284	SUBA	#ZAL4
Line:	00330	->	37af	->	90 9e	UUU285	SUBA	ZAL1+%011
Line:	00331	->	37b1	->	b0 ab cd	UUU286	SUBA	$ABCD
Line:	00332	->	37b4	->	a0 3f	VVV287	SUBA	ZAL3+3,X
Line:	00333	->	37b6	->	18 a0 1c	VVV288	SUBA	ZAL3-4/2,Y
Line:	00334	->	37b9	->	c0 80	VVV289	SUBB	#ZAL4
Line:	00335	->	37bb	->	d0 9e	VVV290	SUBB	ZAL1+%011
Line:	00336	->	37bd	->	f0 ab cd	VVV291	SUBB	$ABCD
Line:	00337	->	37c0	->	e0 3f	VVV292	SUBB	ZAL3+3,X
Line:	00338	->	37c2	->	18 e0 1c	VVV293	SUBB	ZAL3-4/2,Y
Line:	00339	->	37c5	->	83 00 80	VVV294	SUBD	#ZAL4
Line:	00340	->	37c8	->	93 9e	VVV295	SUBD	ZAL1+%011
Line:	00341	->	37ca	->	b3 ab cd	VVV296	SUBD	$ABCD
Line:	00342	->	37cd	->	a3 3f	VVV297	SUBD	ZAL3+3,X
Line:	00343	->	37cf	->	18 a3 1c	VVV298	SUBD	ZAL3-4/2,Y
Line:	00344	->	37d2	->	3f	VVV299	SWI	
Line:	00345	->	37d3	->	16	VVV300	TAB	
Line:	00346	->	37d4	->	06	VVV301	TAP	
Line:	00347	->	37d5	->	17	XXX302	TBA	
Line:	00348	->	37d6	->	07	XXX304	TPA	
Line:	00349	->	37d7	->	7d ab cd	XXX305	TST	$ABCD
Line:	00350	->	37da	->	6d 3f	XXX306	TST	ZAL3+3,X
Line:	00351	->	37dc	->	18 6d 1c	XXX307	TST	ZAL3-4/2,Y
Line:	00352	->	37df	->	4d	XXX308	TSTA	
Line:	00353	->	37e0	->	5d	XXX309	TSTB	
Line:	00354	->	37e1	->	30	XXX310	TSX	
Line:	00355	->	37e2	->	18 30	YYY311	TSY	
Line:	00356	->	37e4	->	35	YYY312	TXS	
Line:	00357	->	37e5	->	18 35	YYY313	TYS	
Line:	00358	->	37e7	->				

```
Line:   00359   ->  37e7    ->  8f              YYY315      XGDX
Line:   00360   ->  37e8    ->  18 8f           YYY316      XGDY
Line:   00361   ->  37ea    ->
```

B.2. VERIFICATION OF SIM68TT1

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
34FF	ED	ED	1	1	0	1	0	0	0	0	3680	3680	F431	00	00	00	0	
3501	A3	ED	1	1	0	1	1	0	0	0	3680	3680	F431	86	A3	C6	2	LDAA #$A3
3503	A3	5D	1	1	0	1	0	0	0	0	3680	3680	F431	C6	5D	CE	4	LDAB #$5D
3506	A3	5D	1	1	0	1	0	0	0	0	77EE	3680	F431	77	EE	18	7	LDX #$77EE
3507	A3	5D	1	1	0	1	0	0	0	0	77EE	3680	F431	X	X	X	8	
350A	A3	5D	1	1	0	1	1	0	0	0	77EE	ABCD	F431	AB	CD	8E	11	LDY #$ABCD
350D	A3	5D	1	1	0	1	1	0	0	0	77EE	ABCD	F435	X	X	X	14	LDS #$F435
350E	00	5D	1	1	0	1	0	1	0	1	77EE	ABCD	F435	X	X	X	16	ABA
350F	00	5D	1	1	0	1	0	1	0	1	784B	ABCD	F435	X	X	X	19	ABX
3510	00	5D	1	1	0	1	0	1	0	1	784B	ABCD	F435	X	X	X	20	
3511	00	5D	1	1	0	1	0	1	0	1	784B	AC2A	F435	X	X	X	23	ABY
3513	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	89	36	99	25	ADCA #$36
3515	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	00	00	00	28	ADCA $56
3517	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	00	00	00	31	ADCA $11
3519	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	00	00	00	35	ADCA $22, X
351A	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	X	X	X	36	
351C	37	5D	1	1	0	1	0	0	0	0	784B	AC2A	F435	00	00	00	40	ADCA $80,Y
351E	37	93	1	1	0	1	1	0	1	0	784B	AC2A	F435	C9	36	D9	42	ADCB #$36
3520	37	93	1	1	0	1	1	0	0	0	784B	AC2A	F435	00	00	00	45	ADCB $5F
3523	37	93	1	1	0	1	1	0	0	0	784B	AC2A	F435	00	00	00	49	ADCB $58FE
3525	37	93	1	1	0	1	1	0	0	0	784B	AC2A	F435	00	00	00	53	ADCB $6B, X
3526	37	93	1	1	0	1	1	0	0	0	784B	AC2A	F435	X	X	X	54	
3528	37	93	1	1	0	1	1	0	0	0	784B	AC2A	F435	00	00	00	58	ADCB $6B,Y
352A	6D	93	1	1	0	1	0	0	0	0	784B	AC2A	F435	8B	36	9B	60	ADDA #$36
352C	6D	93	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	63	ADDA $5F
352F	6D	93	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	67	ADDA $58FE
3531	6D	93	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	71	ADDA $6B, X
3532	6D	93	1	1	1	1	0	0	0	0	784B	AC2A	F435	X	X	X	72	
3534	6D	93	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	76	ADDA $6B,Y
3536	6D	C9	1	1	0	1	1	0	0	0	784B	AC2A	F435	CB	36	DB	78	ADDB #$36
3538	6D	C9	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	00	00	81	ADDB $5F
353B	6D	C9	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	00	00	85	ADDB $58FE
353D	6D	C9	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	00	00	89	ADDB $6B, X

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
353E	6D	C9	1	1	1	1	1	0	0	0	784B	AC2A	F435	X	X	X	90	
3540	6D	C9	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	00	00	94	ADDB $6B,Y
3543	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	36	D3	98	ADDD #$0036
3545	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	103	ADDD $5F
3548	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	109	ADDD $58FE
354A	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	115	ADDD $6B, X
354B	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	X	X	X	116	
354D	6D	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	00	00	122	ADDD $6B,Y
354F	44	FF	1	1	1	1	0	0	0	0	784B	AC2A	F435	84	56	94	124	ANDA #$56
3551	00	FF	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	127	ANDA $5F
3554	00	FF	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	131	ANDA $58FE
3556	00	FF	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	135	ANDA $6B, X
3557	00	FF	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	136	
3559	00	FF	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	140	ANDA $6B,Y
355B	00	56	1	1	1	1	0	0	0	0	784B	AC2A	F435	C4	56	D4	142	ANDB #$56
355D	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	145	ANDB $5F
3560	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	149	ANDB $58FE
3562	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	153	ANDB $6B, X
3563	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	154	
3565	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	158	ANDB $6B,Y
3568	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	164	ASL $58FE
356A	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	170	ASL $6B, X
356B	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	171	
356D	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	177	ASL $6B,Y
356E	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	179	ASLA
356F	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	181	ASLB
3570	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	184	LSLD
3573	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	190	ASR $033A
3575	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	196	ASR $6B, X
3576	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	197	
3578	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	203	ASR $6B,Y
3579	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	205	ASRA
357A	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	207	ASRB
357C	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	24	00	15	210	BCC 00
357F	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	216	BCLR $5F mask: 33
3582	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	223	BCLR $6B, X mask: 37
3583	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	224	
3586	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	231	BCLR $6B,Y mask: 12

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
3588	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	234	BCC 00
358A	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	27	00	2C	237	BEQ 00
358C	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	2C	00	2E	240	BGE 00
358E	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	243	BGT 00
3590	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	246	BHI 00
35A0	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	24	0E	85	249	BCC 0E
35A2	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	252	BITB $5F
35A5	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	256	BITB $033A
35A7	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	260	BITB 08, X
35A8	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	261	
35AA	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	00	265	BITB 6B,Y
35AC	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	2F	00	25	268	BLE 00
35AE	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	271	BCC 00
35B0	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	23	00	2D	274	BLS 00
35B2	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	277	BLT 00
35B4	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	280	BMI 00
35B6	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	283	BNE 00
35B8	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	2A	00	20	286	BPL 00
35BA	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	20	00	13	289	BRA 00
																		BRCLR $11
																		mask: 45
35C7	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	45	09	1F	295	Rel.Adr.: 09
35C9	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	298	BRN 1F
																		BRSET $11
																		mask: EE
35CD	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	11	EE	09	304	Rel.Adr.: 09
																		BRSET $08,
																		X mask: 11
35D1	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	08	11	05	311	Rel.Adr.: 05
35D2	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	X	X	X	312	
																		BRSET $6B,Y
																		mask: 00
35D6	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F435	00	00	14	319	Rel.Adr.: 00
																		BSET $11
35D9	00	00	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	C9	00	325	mask: C9
																		BSET $08, X
35DC	00	00	1	1	1	1	1	0	0	0	784B	AC2A	F435	00	E1	00	332	mask: E1
35DD	00	00	1	1	1	1	1	0	0	0	784B	AC2A	F435	X	X	X	333	
																		BSET $08,Y
35E0	00	00	1	1	1	1	0	0	0	0	784B	AC2A	F435	00	58	00	340	mask: 58
35E2	00	00	1	1	1	1	0	0	0	0	784B	AC2A	F433	00	35	E2	346	BSR
35E4	00	00	1	1	1	1	0	0	0	0	784B	AC2A	F433	28	00	29	349	BVC 00

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
35E6	00	00	1	1	1	1	0	0	0	0	784B	AC2A	F433	X	X	X	352	BVS 00
35E7	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F433	X	X	X	354	CBA
35E8	00	00	1	1	1	1	0	1	0	0	784B	AC2A	F433	X	X	X	356	CLC
35E9	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	358	CLI
35EC	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	364	CLR $ 033A
35EE	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	370	CLR $ 08, X
35EF	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	371	
35F1	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	377	CLR $ 08,Y
35F2	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	379	CLRA
35F3	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	381	CLRB
35F4	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	383	CLV
35F6	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	81	56	91	385	CMPA #$56
35F8	00	00	1	1	1	0	0	0	0	1	784B	AC2A	F433	00	C9	00	388	CMPA $11
35FB	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	392	CMPA $033A
35FD	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	396	CMPA $08, X
35FE	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	397	
3600	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	401	CMPA $08,Y
3602	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	C1	56	D1	403	CMPB #$56
3604	00	00	1	1	1	0	0	0	0	1	784B	AC2A	F433	00	C9	00	406	CMPB $11
3607	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	410	CMPB $033A
3609	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	414	CMPB $08, X
360A	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	X	X	X	415	
360C	00	00	1	1	1	0	0	1	0	0	784B	AC2A	F433	00	00	00	419	CMPB $08,Y
360F	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FF	00	425	COM $033A
3611	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FF	00	431	COM $08, X
3612	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	432	
3614	00	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FF	00	438	COM $08,Y
3615	FF	00	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	440	COMA
3616	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	442	COMB
3617	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	443	
361A	FF	FF	1	1	1	0	1	0	0	0	784B	AC2A	F433	83	00	56	447	CPD #$0056
361B	FF	FF	1	1	1	0	1	0	0	0	784B	AC2A	F433	X	X	X	448	
361D	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	93	11	1A	453	CPD $11
361E	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	X	X	X	454	
3621	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	B3	03	3A	460	CPD $033A
3622	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	X	X	X	461	
3624	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	A3	08	CD	467	CPD $08, X
3625	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	X	X	X	468	
3627	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	A3	08	8C	474	CPD $08,Y
362A	FF	FF	1	1	1	0	0	0	1	0	784B	AC2A	F433	8C	00	11	478	CPX #$0011
362C	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	9C	11	BC	483	CPX $11

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
362F	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	BC	03	3A	489	CPX $033A
3631	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	AC	08	CD	495	CPX $08, X
3632	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	X	X	X	496	
3634	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	AC	08	18	502	CPX $08,Y
3635	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	X	X	X	503	
3638	FF	FF	1	1	1	0	1	0	0	0	784B	AC2A	F433	8C	00	11	507	CPY #$0011
3639	FF	FF	1	1	1	0	1	0	0	0	784B	AC2A	F433	X	X	X	508	
363B	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	9C	11	18	513	CPY $11
363C	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	514	
363F	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	BC	03	3A	520	CPY $033A
3640	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	521	
3642	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	AC	08	18	527	CPY $08, X
3643	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	X	X	X	528	
3645	FF	FF	1	1	1	0	0	0	1	1	784B	AC2A	F433	AC	08	19	534	CPY $08,Y
3646	FF	FF	1	1	1	0	1	0	1	1	784B	AC2A	F433	X	X	X	536	DAA
3649	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FF	00	542	DEC $57FF
364B	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FF	00	548	DEC $3C, X
364C	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	549	
364E	FF	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	00	FE	00	555	DEC $08,Y
364F	FE	FF	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	557	DECA
3650	FE	FE	1	1	1	0	1	0	0	1	784B	AC2A	F433	X	X	X	559	DECB
3651	FE	FE	1	1	1	0	1	0	0	1	784B	AC2A	F432	X	X	X	562	DES
3652	FE	FE	1	1	1	0	1	0	0	1	784A	AC2A	F432	X	X	X	565	DEX
3653	FE	FE	1	1	1	0	1	0	0	1	784A	AC2A	F432	X	X	X	566	
3654	FE	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	X	X	X	569	DEY
3656	EF	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	88	11	98	571	EORA #$11
3658	26	FE	1	1	1	0	0	0	0	1	784A	AC29	F432	00	C9	00	574	EORA $11
365B	D9	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	00	FF	00	578	EORA $57FF
365D	D9	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	00	00	FF	582	EORA $3C, X
365E	D9	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	X	X	X	583	
3660	D9	FE	1	1	1	0	1	0	0	1	784A	AC29	F432	00	00	FE	587	EORA $08,Y
3662	D9	EF	1	1	1	0	1	0	0	1	784A	AC29	F432	C8	11	D8	589	EORB #$11
3664	D9	26	1	1	1	0	0	0	0	1	784A	AC29	F432	00	C9	00	592	EORB $11
3667	D9	D9	1	1	1	0	1	0	0	1	784A	AC29	F432	00	FF	00	596	EORB $57FF
3669	D9	D9	1	1	1	0	1	0	0	1	784A	AC29	F432	00	00	FF	600	EORB $3C, X
366A	D9	D9	1	1	1	0	1	0	0	1	784A	AC29	F432	X	X	X	601	
366C	D9	D9	1	1	1	0	1	0	0	1	784A	AC29	F432	00	00	FE	605	EORB $08,Y
366D	D9	D9	1	1	1	0	1	0	1	0	FFFF	AC29	F432	X	X	X	646	FDIV
366E	D9	D9	1	1	1	0	1	0	0	0	0000	AC29	F432	X	X	X	646	IDIV
3671	D9	D9	1	1	1	0	0	1	0	0	0000	AC29	F432	00	00	00	652	INC $57FF
3673	D9	D9	1	1	1	0	0	0	0	0	0000	AC29	F432	00	01	00	658	INC $3C, X

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
3674	D9	D9	1	1	1	0	0	0	0	0	0000	AC29	F432	X	X	X	659	
3676	D9	D9	1	1	1	0	0	0	0	0	0000	AC29	F432	00	01	FE	665	INC $08,Y
3677	DA	D9	1	1	1	0	1	0	0	0	0000	AC29	F432	X	X	X	667	INCA
3678	DA	DA	1	1	1	0	1	0	0	0	0000	AC29	F432	X	X	X	669	INCB
3679	DA	DA	1	1	1	0	1	0	0	0	0000	AC29	F433	X	X	X	672	INS
367A	DA	DA	1	1	1	0	1	0	0	0	0001	AC29	F433	X	X	X	675	INX
367B	DA	DA	1	1	1	0	1	0	0	0	0001	AC29	F433	X	X	X	676	
367C	DA	DA	1	1	1	0	1	0	0	0	0001	AC2A	F433	X	X	X	679	INY
367F	DA	DA	1	1	1	0	1	0	0	0	0001	AC2A	F433	36	7F	CE	682	JMP $367F
3682	DA	DA	1	1	1	0	0	0	0	0	3680	AC2A	F433	36	80	6E	685	LDX #$3680
3684	DA	DA	1	1	1	0	0	0	0	0	3680	AC2A	F433	6E	04	18	688	JMP $04, X
3685	DA	DA	1	1	1	0	0	0	0	0	3680	AC2A	F433	X	X	X	689	
3688	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F433	36	80	18	692	LDY #$3680
3689	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F433	X	X	X	693	
368B	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F433	6E	0B	9D	696	JMP $0B,Y
00D3	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F431	00	36	8B	701	JMP $D3
368D	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F431	36	8D	00	704	JMP $368D
3690	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F42F	00	36	8D	710	JMP $3690
3692	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F42D	00	36	90	716	JSR $12, X
3693	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F42D	X	X	X	717	
3695	DA	DA	1	1	1	0	0	0	0	0	3680	3680	F42B	00	36	93	723	JSR $15,Y
3697	11	DA	1	1	1	0	0	0	0	0	3680	3680	F42B	86	11	96	725	LDAA #$11
3699	C9	DA	1	1	1	0	1	0	0	0	3680	3680	F42B	00	C9	00	728	LDAA $11
369C	00	DA	1	1	1	0	0	1	0	0	3680	3680	F42B	00	00	00	732	LDAA $57FF
369E	12	DA	1	1	1	0	0	0	0	0	3680	3680	F42B	00	12	9E	736	LDAA $3C, X
369F	12	DA	1	1	1	0	0	0	0	0	3680	3680	F42B	X	X	X	737	
36A1	12	DA	1	1	1	0	0	0	0	0	3680	3680	F42B	00	12	9E	741	LDAA $3C,Y
36A3	12	11	1	1	1	0	0	0	0	0	3680	3680	F42B	C6	11	D6	743	LDAB #$11
36A5	12	C9	1	1	1	0	1	0	0	0	3680	3680	F42B	00	C9	00	746	LDAB $11
36A8	12	00	1	1	1	0	0	1	0	0	3680	3680	F42B	00	00	00	750	LDAB $57FF
36AA	12	12	1	1	1	0	0	0	0	0	3680	3680	F42B	00	12	9E	754	LDAB $3C, X
36AB	12	12	1	1	1	0	0	0	0	0	3680	3680	F42B	X	X	X	755	
36AD	12	12	1	1	1	0	0	0	0	0	3680	3680	F42B	00	12	9E	759	LDAB $3C,Y
36B0	00	11	1	1	1	0	0	0	0	0	3680	3680	F42B	00	11	DC	762	LDD #$0011
36B2	00	00	1	1	1	0	0	1	0	0	3680	3680	F42B	00	00	00	766	LDD $5A
36B5	00	00	1	1	1	0	0	1	0	0	3680	3680	F42B	00	00	00	771	LDD $5680
36B7	12	9E	1	1	1	0	0	0	0	0	3680	3680	F42B	12	9E	5A	776	LDD $3C, X
36B8	12	9E	1	1	1	0	0	0	0	0	3680	3680	F42B	X	X	X	777	
36BA	12	9E	1	1	1	0	0	0	0	0	3680	3680	F42B	12	9E	5A	782	LDD $3C,Y
36BD	12	9E	1	1	1	0	0	0	0	0	3680	3680	0012	X	X	X	785	LDS #$0012
36BF	12	9E	1	1	1	0	0	1	0	0	3680	3680	0000	00	00	00	789	LDS $5A

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
36C2	12	9E	1	1	1	0	0	1	0	0	3680	3680	0000	00	00	00	794	LDS $5680
36C4	12	9E	1	1	1	0	0	0	0	0	3680	3680	129E	12	9E	5A	799	LDS $3C, X
36C5	12	9E	1	1	1	0	0	0	0	0	3680	3680	129E	X	X	X	800	
36C7	12	9E	1	1	1	0	0	0	0	0	3680	3680	129E	12	9E	5A	805	LDS $3C,Y
36CA	12	9E	1	1	1	0	0	0	0	0	0012	3680	129E	00	12	DE	808	LDX #$0012
36CC	12	9E	1	1	1	0	0	1	0	0	0000	3680	129E	00	00	00	812	LDX $5A
36CF	12	9E	1	1	1	0	0	1	0	0	0000	3680	129E	00	00	00	817	LDX $5680
36D1	12	9E	1	1	1	0	0	0	0	0	0100	3680	129E	01	00	00	822	LDX $3C, X
36D2	12	9E	1	1	1	0	0	0	0	0	0100	3680	129E	X	X	X	823	
36D4	12	9E	1	1	1	0	0	0	0	0	129E	3680	129E	12	9E	5A	828	LDX $3C,Y
36D5	12	9E	1	1	1	0	0	0	0	0	129E	3680	129E	X	X	X	829	
36D8	12	9E	1	1	1	0	0	0	0	0	129E	0012	129E	00	12	18	832	LDY #$0012
36D9	12	9E	1	1	1	0	0	0	0	0	129E	0012	129E	X	X	X	833	
36DB	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	00	00	00	837	LDY $5A
36DC	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	X	X	X	838	
36DF	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	00	00	00	843	LDY $5680
36E0	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	X	X	X	844	
36E2	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	00	00	00	849	LDY $3C, X
36E3	12	9E	1	1	1	0	0	1	0	0	129E	0000	129E	X	X	X	850	
36E5	12	9E	1	1	1	0	0	0	0	0	129E	0100	129E	01	00	00	855	LDY $3C,Y
36E8	12	9E	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	861	ASL $5680
36EA	12	9E	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	867	ASL $3C, X
36EB	12	9E	1	1	1	0	0	1	0	0	129E	0100	129E	X	X	X	868	
36ED	12	9E	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	874	ASL $3C,Y
36EE	24	9E	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	876	ASLA
36EF	24	3C	1	1	1	0	0	0	1	1	129E	0100	129E	X	X	X	878	ASLB
36F0	48	78	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	881	LSLD
36F3	48	78	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	887	LSR $5680
36F5	48	78	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	893	LSR $3C, X
36F6	48	78	1	1	1	0	0	1	0	0	129E	0100	129E	X	X	X	894	
36F8	48	78	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	900	LSR $3C,Y
36F9	24	78	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	902	LSRA
36FA	24	3C	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	904	LSRB
36FB	12	1E	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	907	LSLD
36FC	02	1C	1	1	1	0	0	0	0	0	129E	0100	129E	X	X	X	917	MUL
36FF	02	1C	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	923	NEG $5680
3701	02	1C	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	929	NEG $3C, X
3702	02	1C	1	1	1	0	0	1	0	0	129E	0100	129E	X	X	X	930	
3704	02	1C	1	1	1	0	0	1	0	0	129E	0100	129E	00	00	00	936	NEG $3C,Y
3705	FE	1C	1	1	1	0	1	0	0	1	129E	0100	129E	X	X	X	938	NEGA
3706	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	X	X	X	940	NEGB

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
3707	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	X	X	X	942	NOP
3709	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	8A	12	9A	944	ORA #$12
370B	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	947	ORA $5A
370E	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	951	ORA $5680
3710	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	955	ORA $3F, X
3711	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	X	X	X	956	
3713	FE	E4	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	960	ORA $3C,Y
3715	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	CA	12	DA	962	ORB #$12
3717	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	965	ORB $5A
371A	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	969	ORB $5680
371C	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	973	ORB $3F, X
371D	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	X	X	X	974	
371F	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129E	00	00	00	978	ORB $3C,Y
3720	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129D	00	FE	00	981	PSHA
3721	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129C	00	F6	FE	984	PSHB
3722	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129A	00	12	9E	988	PSHX
3723	FE	F6	1	1	1	0	1	0	0	1	129E	0100	129A	X	X	X	989	
3724	FE	F6	1	1	1	0	1	0	0	1	129E	0100	1298	00	01	00	993	PSHY
3725	01	F6	1	1	1	0	1	0	0	1	129E	0100	1299	00	01	00	997	PULA
3726	01	00	1	1	1	0	1	0	0	1	129E	0100	129A	01	00	12	1001	PULB
3727	01	00	1	1	1	0	1	0	0	1	129E	0100	129C	12	9E	F6	1006	PULX
3728	01	00	1	1	1	0	1	0	0	1	129E	0100	129C	X	X	X	1007	
3729	01	00	1	1	1	0	1	0	0	1	129E	F6FE	129E	F6	FE	00	1012	PULY
372C	01	00	1	1	1	0	0	0	0	0	129E	F6FE	129E	00	01	00	1018	ROL $5680
372E	01	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	00	00	00	1024	ROL $3F, X
372F	01	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	X	X	X	1025	
3731	01	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	00	00	00	1031	ROL $1C,Y
3732	02	00	1	1	1	0	0	0	0	0	129E	F6FE	129E	X	X	X	1033	ROLA
3733	02	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	X	X	X	1035	ROLB
3736	02	00	1	1	1	0	0	1	1	1	129E	F6FE	129E	00	00	00	1041	ROR $5680
3738	02	00	1	1	1	0	1	0	1	0	129E	F6FE	129E	00	80	00	1047	ROR $3F, X
3739	02	00	1	1	1	0	1	0	1	0	129E	F6FE	129E	X	X	X	1048	
373B	02	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	00	00	00	1054	ROR $1C,Y
373C	01	00	1	1	1	0	0	0	0	0	129E	F6FE	129E	X	X	X	1056	RORA
373D	01	00	1	1	1	0	0	1	0	0	129E	F6FE	129E	X	X	X	1058	RORB
3740	01	00	1	1	1	0	1	0	0	0	129E	F6FE	A000	X	X	X	1061	LDS #$A000
3743	01	00	1	1	1	0	0	0	0	0	3749	F6FE	A000	37	49	3C	1064	LDX #$3749
3744	01	00	1	1	1	0	0	0	0	0	3749	F6FE	9FFE	00	37	49	1068	PSHX
3745	01	00	1	1	1	0	0	0	0	0	3749	F6FE	9FFD	00	01	37	1071	PSHA
3746	01	00	1	1	1	0	0	0	0	0	3749	F6FE	9FFB	00	37	49	1075	PSHX
3747	01	00	1	1	1	0	0	0	0	0	3749	F6FE	9FF9	00	37	49	1079	PSHX

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
3748	01	00	1	1	1	0	0	0	0	0	3749	F6FE	9FF7	00	37	49	1083	PSHX
3749	37	49	0	0	1	1	0	1	1	1	4937	4901	A000	37	49	00	1095	RTI
374C	37	49	0	0	1	1	0	0	0	1	374E	4901	A000	37	4E	3C	1098	LDX #$374E
374D	37	49	0	0	1	1	0	0	0	1	374E	4901	9FFE	01	37	4E	1102	PSHX
374E	37	49	0	0	1	1	0	0	0	1	374E	4901	A000	37	4E	00	1107	RTS
374F	EE	49	0	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1109	SBA
3751	6D	49	0	0	1	1	0	0	0	0	374E	4901	A000	82	80	92	1114	SBCA #$80
3753	6D	49	0	0	1	1	0	0	0	0	374E	4901	A000	00	00	00	1117	SBCA $9B
3756	6D	49	0	0	1	1	0	0	0	0	374E	4901	A000	00	00	00	1121	SBCA $5680
3758	CF	49	0	0	1	1	1	0	1	1	374E	4901	A000	9F	9E	BF	1125	SBCA $3F, X
3759	CF	49	0	0	1	1	1	0	1	1	374E	4901	A000	X	X	X	1126	
375B	CE	49	0	0	1	1	1	0	0	0	374E	4901	A000	00	00	00	1130	SBCA $1C,Y
375D	CE	C9	0	0	1	1	1	0	1	1	374E	4901	A000	C2	80	D2	1135	SBCB #$80
375F	CE	C8	0	0	1	1	1	0	0	0	374E	4901	A000	00	00	00	1138	SBCB $9B
3762	CE	C8	0	0	1	1	1	0	0	0	374E	4901	A000	00	00	00	1142	SBCB $ABCD
3764	CE	2A	0	0	1	1	0	0	0	0	374E	4901	A000	9F	9E	BF	1146	SBCB $3F, X
3765	CE	2A	0	0	1	1	0	0	0	0	374E	4901	A000	X	X	X	1147	
3767	CE	2A	0	0	1	1	0	0	0	0	374E	4901	A000	00	00	00	1151	SBCB $1C,Y
3768	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1153	SEC
3769	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1155	SEI
376A	CE	2A	0	0	1	1	0	0	1	1	374E	4901	A000	X	X	X	1157	SEV
376C	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	00	CE	00	1160	STAA $9B
376F	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	00	CE	00	1164	STAA $ABCD
3771	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	9F	CE	BF	1168	STAA $3F, X
3772	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1169	
3774	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	00	CE	00	1173	STAA $1C,Y
3776	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	00	2A	00	1176	STAB $9B
3779	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	00	2A	00	1180	STAB $ABCD
377B	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	9F	2A	BF	1184	STAB $3F, X
377C	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1185	
377E	CE	2A	0	0	1	1	0	0	0	1	374E	4901	A000	00	2A	00	1189	STAB $1C,Y
3780	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	CE	2A	00	1193	STD $9E
3783	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	CE	2A	00	1198	STD $ABCD
3785	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	CE	2A	AB	1203	STD $3F, X
3786	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1204	
3788	CE	2A	0	0	1	1	1	0	0	1	374E	4901	A000	CE	2A	00	1209	STD $1C,Y
378A	FF	2A	0	0	1	1	1	0	0	1	374E	4901	A000	86	FF	06	1211	LDAA #$FF
378B	FF	2A	1	0	1	1	1	1	1	1	374E	4901	A000	X	X	X	1213	TAP
378C	FF	2A	1	0	1	1	1	1	1	1	374E	4901	A000	X	X	X	1215	STOP
378E	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	A0	00	00	1219	STS $CE
3790	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1222	BPL AB

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
3791	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1223	
3791	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	37	91	00	1223	
3793	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	A0	00	AB	1228	STS $3F, X
3794	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1229	
3796	FF	2A	1	0	1	1	1	0	0	1	374E	4901	A000	A0	00	00	1234	STS $1C,Y
3798	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	37	4E	00	1238	STX $9E
379B	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	37	4E	00	1243	STX $ABCD
379D	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	37	4E	AB	1248	STX $3F, X
379E	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1249	
37A0	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	37	4E	00	1254	STX $1C,Y
37A1	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1255	
37A3	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	49	01	00	1259	STY $9E
37A4	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1260	
37A7	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	49	01	00	1265	STY $ABCD
37A8	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1266	
37AA	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	49	01	AB	1271	STY $3F, X
37AB	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	X	X	X	1272	
37AD	FF	2A	1	0	1	1	0	0	0	1	374E	4901	A000	49	01	00	1277	STY $1C,Y
37AF	7F	2A	1	0	1	1	0	0	0	0	374E	4901	A000	80	80	90	1279	SUBA #$80
37B1	36	2A	1	0	1	1	0	0	0	0	374E	4901	A000	00	49	01	1282	SUBA $9E
37B4	ED	2A	1	0	1	1	1	0	0	1	374E	4901	A000	00	49	01	1286	SUBA $ABCD
37B6	A4	2A	1	0	1	1	1	0	0	0	374E	4901	A000	9F	49	01	1290	SUBA $3F, X
37B7	A4	2A	1	0	1	1	1	0	0	0	374E	4901	A000	X	X	X	1291	
37B9	5B	2A	1	0	1	1	0	0	1	0	374E	4901	A000	00	49	01	1295	SUBA $1C,Y
37BB	5B	AA	1	0	1	1	1	0	1	1	374E	4901	A000	C0	80	D0	1297	SUBB #$80
37BD	5B	61	1	0	1	1	0	0	1	0	374E	4901	A000	00	49	01	1300	SUBB $9E
37C0	5B	18	1	0	1	1	0	0	0	0	374E	4901	A000	00	49	01	1304	SUBB $ABCD
37C2	5B	CF	1	0	1	1	1	0	0	1	374E	4901	A000	9F	49	01	1308	SUBB $3F, X
37C3	5B	CF	1	0	1	1	1	0	0	1	374E	4901	A000	X	X	X	1309	
37C5	5B	86	1	0	1	1	1	0	0	0	374E	4901	A000	00	49	01	1313	SUBB $1C,Y
37C8	5B	06	1	0	1	1	0	0	0	0	374E	4901	A000	83	00	80	1317	SUBD #$0080
37CA	12	05	1	0	1	1	0	0	0	0	374E	4901	A000	00	49	01	1322	SUBD $9E
37CD	C9	04	1	0	1	1	1	0	0	1	374E	4901	A000	00	49	01	1328	SUBD $ABCD
37CF	80	03	1	0	1	1	1	0	0	0	374E	4901	A000	9F	49	01	1334	SUBD $3F, X
37D0	80	03	1	0	1	1	1	0	0	0	374E	4901	A000	X	X	X	1335	
37D2	37	02	1	0	1	1	0	0	1	0	374E	4901	A000	00	49	01	1341	SUBD $1C,Y
37D3	37	02	1	0	1	1	0	0	1	0	374E	4901	9FF7	00	B2	02	1355	SWI
37D4	37	37	1	0	1	1	0	0	0	0	374E	4901	9FF7	X	X	X	1357	TAP
37D5	37	37	0	0	1	1	0	1	1	1	374E	4901	9FF7	X	X	X	1359	TAP
37D6	37	37	0	0	1	1	0	0	0	1	374E	4901	9FF7	X	X	X	1361	TBA
37D7	31	37	0	0	1	1	0	0	0	1	374E	4901	9FF7	X	X	X	1363	TPA

P.CNT.	ACCA	ACCB	S	X	H	I	Z	N	V	C	IX	IY	SP	M-1	M	M+1	E.CYC.	Source Code
37DA	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	00	49	01	1369	TST $ABCD
37DC	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	9F	49	01	1375	TST $3F, X
37DD	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	X	X	X	1376	
37DF	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	00	49	01	1382	TST $1C,Y
37E0	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	X	X	X	1384	TSTA
37E1	31	37	0	0	1	1	0	0	0	0	374E	4901	9FF7	X	X	X	1386	TSTB
37E2	31	37	0	0	1	1	0	0	0	0	9FF8	4901	9FF7	X	X	X	1389	TSX
37E3	31	37	0	0	1	1	0	0	0	0	9FF8	4901	9FF7	X	X	X	1390	
37E4	31	37	0	0	1	1	0	0	0	0	9FF8	9FF8	9FF7	X	X	X	1393	TSY
37E5	31	37	0	0	1	1	0	0	0	0	9FF8	9FF8	9FF7	X	X	X	1396	TXS
37E6	31	37	0	0	1	1	0	0	0	0	9FF8	9FF8	9FF7	X	X	X	1397	
37E7	31	37	0	0	1	1	0	0	0	0	9FF8	9FF8	9FF7	X	X	X	1400	TYS
37E8	9F	F8	0	0	1	1	0	0	0	0	3137	9FF8	9FF7	X	X	X	1403	XGDX
37E9	9F	F8	0	0	1	1	0	0	0	0	3137	9FF8	9FF7	X	X	X	1404	
37EA	9F	F8	0	0	1	1	0	0	0	0	3137	9FF8	9FF7	X	X	X	1407	XGDY

APPENDIX C: USER MANUALS

This appendix is consists of the short user manuals for ASM68TT11 and SIM68TT11 for a quick start to a new user.

C.1. MANUAL OF ASM68TT11

The cross-assembler is named as 'ASM68TT11.exe'. To generate MC6811 code runs the ASM68TT11.exe in command prompt. To run the cross-assembler enters the following command line:

ASM68TT11 file1.asm

There are no options for the program. Once the command line is written, press enter. The cross-assembler produces the output files related to the input file. The input file and the cross-assembler must be placed in the same directory.

The output files are DEN file, OUX file and LIT files. OUX file is the output of the cross-assembler for the simulator; the simulator will use this output file. DEN file contains the source code and the related object code; it can help the user to see the results of the source code and the hex code of the micro controller. LIT keeps the track of the variables and labels with their hexadecimal values. Cross-assembler gives the outputs at every successful process and places them in the same directory with itself.

User must pay attention to the BCLR, BSET, BRSET and BRCLR instructions when writing the source code. BSET, BCLR, BRSET and BRCLR instructions have a more than one operand including MASK and ADDRESS information. These operands must be separated by colons (':').

C.2. MANUAL OF SIM68TT11

To run the simulator correctly user must save the “SIM68TT11.exe”, “movs” folder and “pix” folder in the same folder. Otherwise, the program cannot find the related pictures and animations used in the simulator.

After the needed files are copied to the same directory and folder. User can start the program by executing the SIM68TT11.exe. When the simulator runs the main window in Figure 16 is displayed. To run a code in the simulator user must load the code using the LOAD button in the menu bar. SIM68TT11 can run both ASM68TT11 and AS11 output files. The output of these programs are ‘*.OUX’ and ‘*.S19’ respectively.

After user loads an object code to the memory of the SIM68TT11 user should select the running mode and related topics from the options menu. The menu tree of SIM68TT11 is shown in Figure 36.

First option is ‘Run Mode’ which has two sub options. These sub options select the running mode of the program. SIM68TT11 has two main running modes; continuous and step by step. In continuous mode simulator executes the object code without a break and until user clicks STOP button. In the step by step mode simulator executes a certain amount of the object code (this could be an instruction or a cycle at a time) and waits user to click FWD button to execute the next part. The sub options in the ‘Run Mode’ selects these run modes of the SIM68TT11.

Step by step mode can execute a program by ‘instruction by instruction’ or ‘cycle by cycle’. The ‘instruction by instruction’ option executes a single instruction at a time. The ‘cycle by cycle’ option executes a cycle of a specific instruction at a time. The cycle is not same as the E_Cycle given in the Motorola’s reference documents. The cycle here is each step of the program progressed after user clicks FWD button and

represents smaller steps than the E_Cycle information. For example, MC6811 fetches a word from memory to a register in one E_Cycle but SIM68TT11 fetches a word in four cycles. These four cycles are loading data from memory to external data bus, loading data from external data bus to Memory Buffer Register, loading data from Memory Buffer Register to internal data bus and loading data from internal data bus to related register.

In continuous mode, the simulator executes the loaded code without stopping until the user gives a 'STOP' command. The 'CONT. SPEED' option determines the speed of this mode. If "1 Instr. Per Sec." is selected then one instruction will be executed in each second. If user selects "Fast", the speed of SIM68TT11 is the top speed that is allowed by the computer, which SIM68TT11 runs on.

Memory Mode selects the type of the memory window. Memory window shows the contents of the memory to the user. It has two modes '255x255' or '1x65535'. These modes define the size of the memory grid.

After loading an object code SIM68TT11 activates the RUN button. Pressing the Run button initializes the microcomputer and starts the program. According to the selected mode, SIM68TT11 executes the code in the memory. If the mode is, "Step by Step" user must use the FWD and BACK button to progress through the stored program. BACK button works only in the 'Cycle by Cycle' mode and turns the program to the first step of the execution phase of the current instruction. While the code is being executed by, the SIM68TT11 user can follow the program from the information boxes introduced in previous chapters.

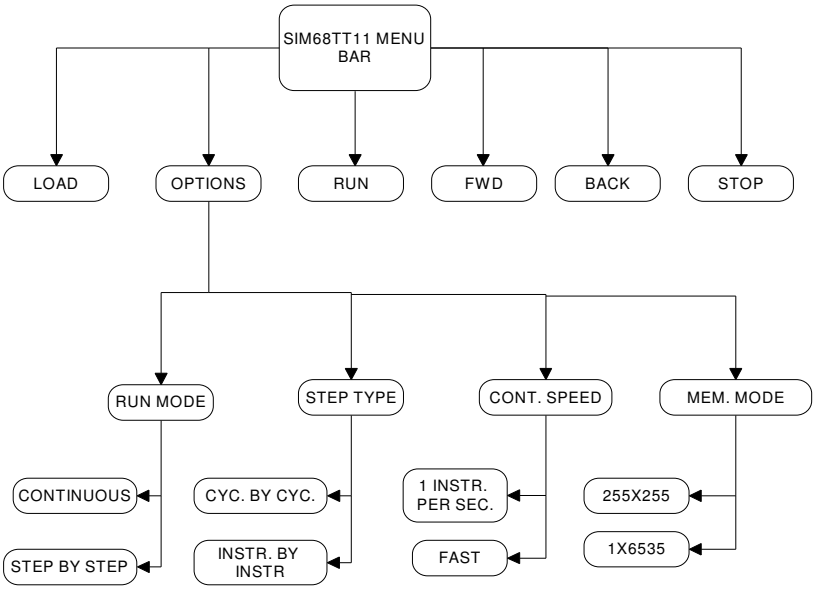

Figure 36 SIM68TT11 MENU

During the program executing user can use ’F’ key from the keyboard instead of FWD button and ’B’ key from the keyboard instead of BACK button. User can also display the memory space by clicking on the memory block. There are also info and help boxes in the program. If user points a label by mouse, an info box appears. Info boxes give user brief information about the related item. If user wants detailed information, he/she should click on the item to display the help box.

APPENDIX D: PROGRAMS

This appendix is a CD-ROM which contains the SIM68TT11 and ASM68TT11 executable files with their user manuals. The CD-ROM also contains the Motorola Inc., M68HC11RM/AD HC11 MC6811 Reference Manual. The CD-ROM is attached to the back cover of thesis.