Module 12 – Software assemblers, editors and simulators

Overview

Stored program design (concept map)

Assembly language programming (concept map)

Objectives

At the completion of this module you will be able to:

- define software and define its use in applications, support and system programs
- explain the concept of an assembler and how it may be used to develop a program for a microprocessor
- explain the operation and features of the THRSim11 simulator software
- state the purpose of a simulator program.

12.1 Software

12.1.1 Introduction

Software is the term coined to describe the programs or set of instructions that ‘run’ the computer hardware. It is termed ‘software’ because traditionally it has been stored in a storage medium which allows easy alteration and modification without the change of any physical components. The introduction of ROM (read only memory) has altered this criteria slightly since programs held in ROM are permanent and unalterable. Once a program is held in ROM it could be argued that it comprises part of the ‘hardware’ instead of the ‘software’. When ROM is used to store programs, that program is often referred to as **firmware.**

Generally software can be split into three areas. These are:

- applications software
- support software
- system software.

Each of these areas will be discussed below but it should be realised that the application for which the computer is to be used will determine whether or not all three or only selected areas will exist with a particular system.

12.1.2 Applications software

These are the programs which make the computer perform a specific task for which it has been purchased. The programs written for dedicated microprocessor applications are 'Applications software', e.g. process control applications, data loggers, etc.

Such programs are usually machine dependent in that they are designed with a particular computer in mind and use the special features it provides, e.g. interrupt handling facilities. 'Application programs' once debugged and tested are usually stored in machine language form in ROM.

The majority of microprocessor based dedicated systems run 'applications software' exclusively. Some applications which require higher operating speed or extensive computation may use mini-computer based systems.

12.1.3 Support software

When developing 'Application Programs' it is virtually essential to provide development aides to the programmer. Programs which fall into this category are called**support software**. Programs developed, supplied and supported by the computer manufacturer for a particular make and type of machine which help in writing and testing 'application programs' are classed as 'support software'.

The most obvious examples are assemblers, compilers, text editors for typing in and correcting programs.

A second type of support software is available as packages of machine code subroutines which can be introduced as blocks of code into application programs. Typical examples are mathematical routines; e.g. floating point arithmetic, multiplication, division, trigonometric functions etc.; I/O handling and conversion routines, e.g. conversion from binary numbers to a string of ASCII characters etc. These packages are available to the programmer for inclusion in his application program and are known as utility routines.

In the microprocessor range two examples exist. The first is the 'microprocessor development system', which is designed to provide support for one microprocessor. Briefly this provides an editing facility for entering and changing the text of the program, an assembler or in some cases a compiler to translate the source code (program entered in assembler or higher level language) into object code (i.e. machine code), a simulator which is a program that 'simulates' the CPU operation and checks the program operation and a PROM programmer to store the debugged program in PROM. This system is used purely for Software Development.

The second example is the desktop microcomputer, which is based on a microprocessor system that runs **compiled** programs and may include an **interpreter**.

Such systems often can only be programmed in the high level language supported by the Compiler/Interpreter software, e.g. **Visual Basic**. An Interpreter may be considered part of the 'applications program' since both must operate together to provide useful program execution.

Microcomputer based systems extend across a broad range. At one end of the spectrum there is the stand-alone microcomputer used to run application type programs but with support facilities in the form of a word processor, assembler and compiler for developing its own applications programs.

At the other extreme there are microcomputer network systems used extensively for multi-user operation with multiple support systems. This server/client type system can run applications programs such as payroll, student records, etc. as well as providing software development aides to develop these and other programs along with providing an extensive range of utility packages.

12.1.4 System software

System Software is reserved for use on larger computer systems which have several I/O devices and a large number of support programs and packages usually stored on magnetic disc or tape. The system may be single user but more usually it is a multi-user system.

Any user developing his application program to run on this system (e.g. a ‘payroll program’ or ‘structural analysis program’) can write it so that it performs every task required of the computer. Many tasks such as file handling on disc or tape or timing control sequence for a particular I/O device do not differ from one application to the next. Such tasks can be provided by the system and are provided each time a system task is requested. Thus software, which is dedicated to the running and organising the computer system in a wider supervisory sense is classed as system software. In controlling the general operation of the computer system, the system software frees the application programs from standard system tasks.

A special system program called the **executive** program manages the complete computer system and organises the allocation of all resources provided by the system between the multiple users, i.e. allocates I/O devices, memory, system software, support software, etc.

The system executive and the supporting system software is referred to as the **operating system**.

The function of the ‘operating system’ in a multi-user system is as follows:

- **Processor scheduling** – Allocation of processing time to ‘jobs’ waiting to be run. Various techniques can be employed. One method is to run each job (i.e. program) for a fixed period called a **‘time slice’**. If the program completes within this period the job is complete, if the job is not complete it returns to the queue of jobs awaiting processing. Similarly if the program requires access to an I/O device which is being used elsewhere it is placed in another queue awaiting the availability of that device.

  The aim of the ‘processor scheduling’ is to get the maximum program throughout without large programs ‘hogging’ the system at the expense of small programs or longer programs being equally well ‘stalled’ by smaller programs.
- **Job scheduling** – Allocation of jobs awaiting processing. Normally a large computer system has multiple users each with a different ‘priority’. The ‘job scheduler’ will organise the queue of jobs according to priority. Jobs can be organised in **batch mode** where programs are run overnight at the convenience of the system or **interactive mode** where the actual user is requesting program execution immediately and is waiting for the results **on-line**.
- **Resource management** – This is the general term used here to include

- allocation of support programs as requested
- allocation of I/O devices and related system software
- allocation of memory space for jobs in process
- transfer of programs between main memory and backing store, i.e. magnetic disc/tape.

All large computer systems based on microcomputer and main frame computers have an operating system. The complexity and major features provided varies considerably and is closely related to price. The cost of the system software often exceeds the cost of the hardware alone.

Unfortunately time does not permit a full consideration of the complete computer range together with details of the software systems as categorised above.

In the time available in this course our attention will be restricted to the major features of the assembler and the software developments for microprocessor based systems.

Self assessment 12.1

1. What is computer software?
2. How is system software different from application software?

12.2 Assemblers

12.2.1 Introduction

The basic purpose of an assembler is to translate assembly language mnemonics into binary machine language code (source code or opcode). An assembler is a software program, which runs on a host computer. Most assemblers offer: labelling, comments and symbol tables.

Assembler statements consist of ‘fields’

i.e.

Label	Mnemonics	Operand	Comment

Most microprocessor assemblers use what is called ‘free formatting’ which means the individual fields can be separated by a ‘delimiter’ put there by the programmer. Typical delimiters are the space, colon, slash and comma.

12.2.2 Assembler fields

The first field is the label field. Labels should be used often and most assemblers will allow a certain number of characters per label such as six or eight. Labels assist in understanding the algorithms used by a particular programmer to solve a problem.

The next field is the mnemonic field. Normal mnemonics are used in this field and it is the only field which **must** have an entry in every line.

The assembler software simply compares the operation code with a 'lookup table' until it finds a match. The match will then yield the **binary code** for that particular mnemonic.

These lookup tables are usually stored in a read only memory.

12.2.3 Pseudo operations

These are **directives** to the assembler only and are ignored by the computer itself. They appear in the mnemonic field but are not translated into binary code.

They may assign program and data to areas of memory, define symbols, allocate space for variables, generate fixed tables or mark the end of a program. Typical examples are: origin (ORG), equate (EQU), reserve (RES), data (DAT), end (END).

12.2.4 Example

To assist with the understanding of an assembler operation, consider the following example:

Label field	Mnemonic field	Operand field	Comment field
BILL	EQU	$0000	
JACK	EQU	$0001	
	ORG	$0010	
START	LDA	#$50	Set Counter
	"	"	
	"	"	
	DECA		Decrement count
	BEQ	START	If zero go to START
	LDAA	BILL	Get Bill
	BPL	HERE	is it -ve
	NEGA		Yes → negate
HERE	CMPA	JACK	
	"	"	
	etc.	"	

The assembler is to code this program into machine code. Designed into the assembler is a table which contains the following:

Mnemonic	Machine code in hex	Instruction length (bytes)

ADDA #	8B	2
CLRA	4F	1
"	"	"
"	"	"
"	"	"

This allows the assembler to examine the **mnemonic field** and the 'address mode' and code the program by 'table look up' methods.

Because 'symbols' or 'labels' are convenient for use when writing assembly language programs, the assembler must keep track of these and insert the preset values or calculate the 'offsets' accordingly.

This initially looks simple enough, the assembler scans the program a line at a time and

- keeps track of the 'memory address' occupied by each instruction. This is done by setting a '**location counter**' to the value set by the 'ORG' statement and incrementing it for each instruction according to the Instruction Length given in the **mnemonic table**.
- constructs a **symbol table** of entries in the label field. When a 'symbol or label' is encountered in the 'Label Field' it is entered in the symbol table together with the value it is assigned for 'EQU directives' or the value of the 'Location Counter' elsewhere in the program.

e.g.	BILL	$0000
	JACK	$0001
	START	$0010
	etc.	

- Coding the program directly into machine code can now proceed a line at a time as it is entered.

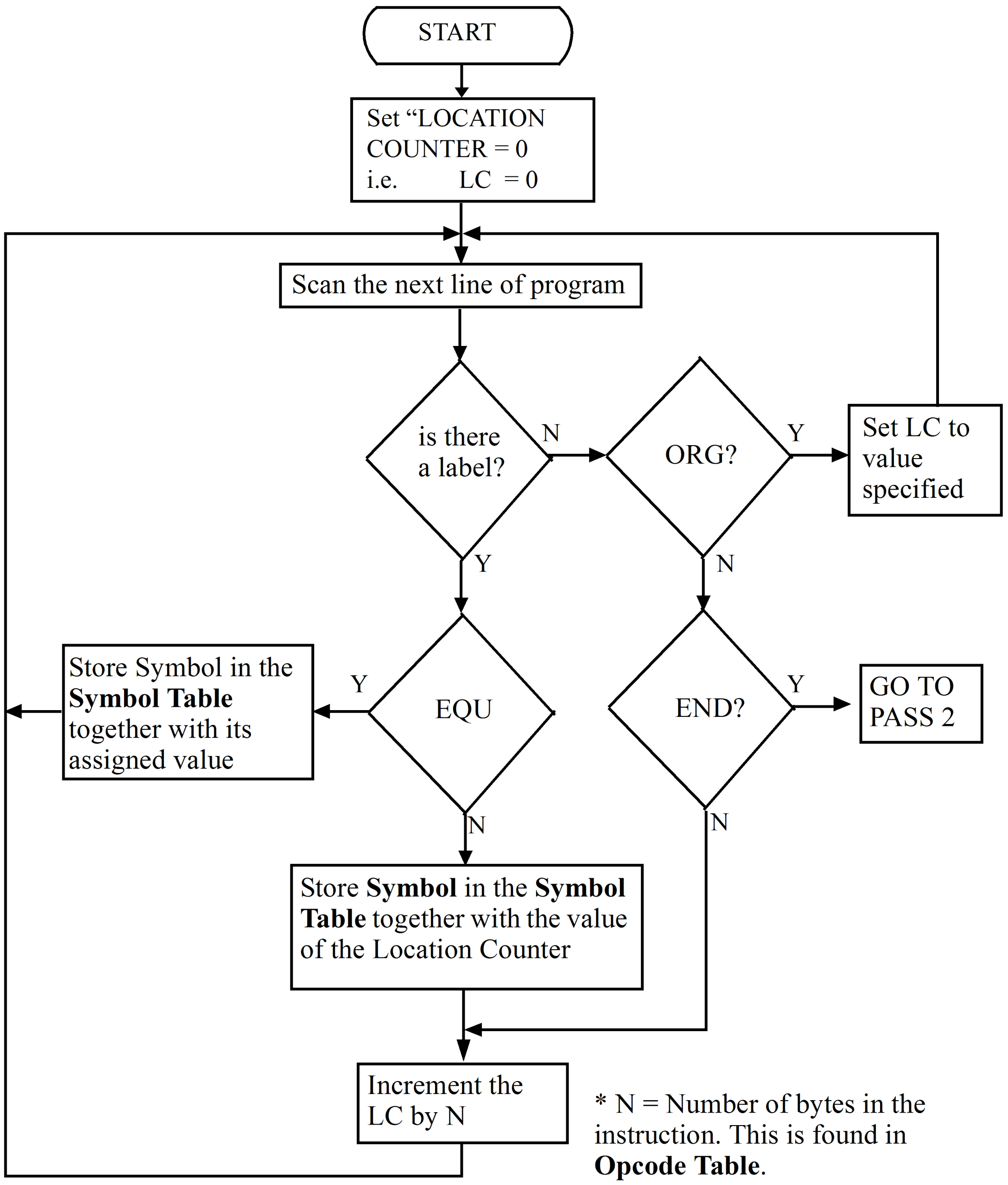
  When a 'label/symbol' is encountered in the 'operand field', e.g. as in 'BEQ START', the address or value assigned to that symbol is obtained from the **symbol** table.

  i.e. The assembler determines that START = $0010 and from this value and that of the location counter it can calculate the 'offset' for the 'BEQ' instruction.
  Note: The Assembler ignores all comments in the comment field.

  This procedure works well until the 'BPL HERE' instruction is encountered. Because the Label 'HERE' has not been reached yet in the 'label field', there is no entry for it in the **symbol** table and direct translation into machine code cannot continue. This is called the **forward reference** problem and is the reason for having a 2-pass assembler.

In a 2-pass assembler, on the **first pass**, i.e. the first time the program is run through the assembler, a complete **symbol table** is constructed. The way this is achieved is shown in the following flowchart.

First PASS


START
Set "LOCATION COUNTER = 0
i.e. LC = 0
Scan the next line of program
is there a label?
N
Y
ORG?
Y
N
Set LC to value specified
EQU
Y
N
Store Symbol in the Symbol Table together with its assigned value
END?
Y
N
GO TO PASS 2
Store Symbol in the Symbol Table together with the value of the Location Counter
Increment the LC by N
* N = Number of bytes in the instruction. This is found in Opcode Table.

Note: The only assembler directives illustrated here are END, EQU and ORG.

At the completion of PASS 1 all the symbols used will have an entry and an assigned value in, the **symbol table**.

On PASS 2 the program can now be scanned a line at a time and coded directly into machine code. When a label or symbol is encountered in the operand field its value can simply be determined from the **symbol table** and the translation continued.

It is possible to have a one-pass assembler, however, these are complex and require a large memory space. In PASS 2 of a two-pass system, as the line is scanned and the code produced the machine code can be output directly onto file. In PASS 1 only the symbol table has to be stored. In a one-pass system the complete coded program has to be stored with special features to 'mark' where the forward reference problem occurred. At the conclusion of reading in the source program (i.e. assembly language program) the assembler can then return to insert the correct values for each '**mark**' and finally output the machine language program (i.e. object code).

Self assessment 12.2

1. What is the purpose of an assembler program?
2. How does assembly language differ from machine language?

12.3 Software development for microprocessors

12.3.1 Introduction

To assist in the development of software for a specific microprocessor, manufacturers often produce development kits. These kits provide sufficient hardware and software to enable the developer to program and test the application software. Typically, development kits include a microprocessor board that includes a serial interface and LED display, management software to run on the host computer (the assembler/debugger), and an interface cable to connect the microprocessor board to the host computer.

With the development kit connected to the host computer, a program can be written on the host in assembly language then assembled into machine code, and the machine code then downloaded to the development kit where it can be run. Some advanced assemblers provide debugging facilities, allowing the developer to monitor microprocessor registers, etc as the program runs. The THRSim11 software interfaced to the EZ-micro development board is a typical example.

12.3.2 The THRSim11 simulator

The following text is an extract from the THRSim11 website, <http://www.hc11.demon.nl/thrsim11/thrsim11.htm>.

The Motorola 68HC11 microcontroller is a popular microcontroller used in many applications. With the THRSim11 program you can edit, assemble, simulate and debug programs for the 68HC11 on your windows PC. You can also use THRSim11 to debug the program on your target EVM or EVB compatable board. The simulator simulates the CPU, ROM, RAM, and all memory mapped I/O ports. It also simulates the on board peripherals such as:

- *timer (including pulse accumulator),*
- *analog to digital converter,*
- *parallel ports (including handshake),*
- *serial port,*

- *I/O pins (including analog and interrupt pins).*

While debugging the graphical user interface makes it possible to view and control every register (CPU registers and I/O registers), memory location (data, program, and stack), and pin of the simulated microcontroller. Even when the program is running! It is possible to stop the simulation at any combination of events. For example: stop when RxD becomes low and RAM location $003F contains $BD or I/O register TCNT is greater than $3456.

A number of (simulated) external components can be connected to the pins of the simulated 68HC11 while debugging. For example:

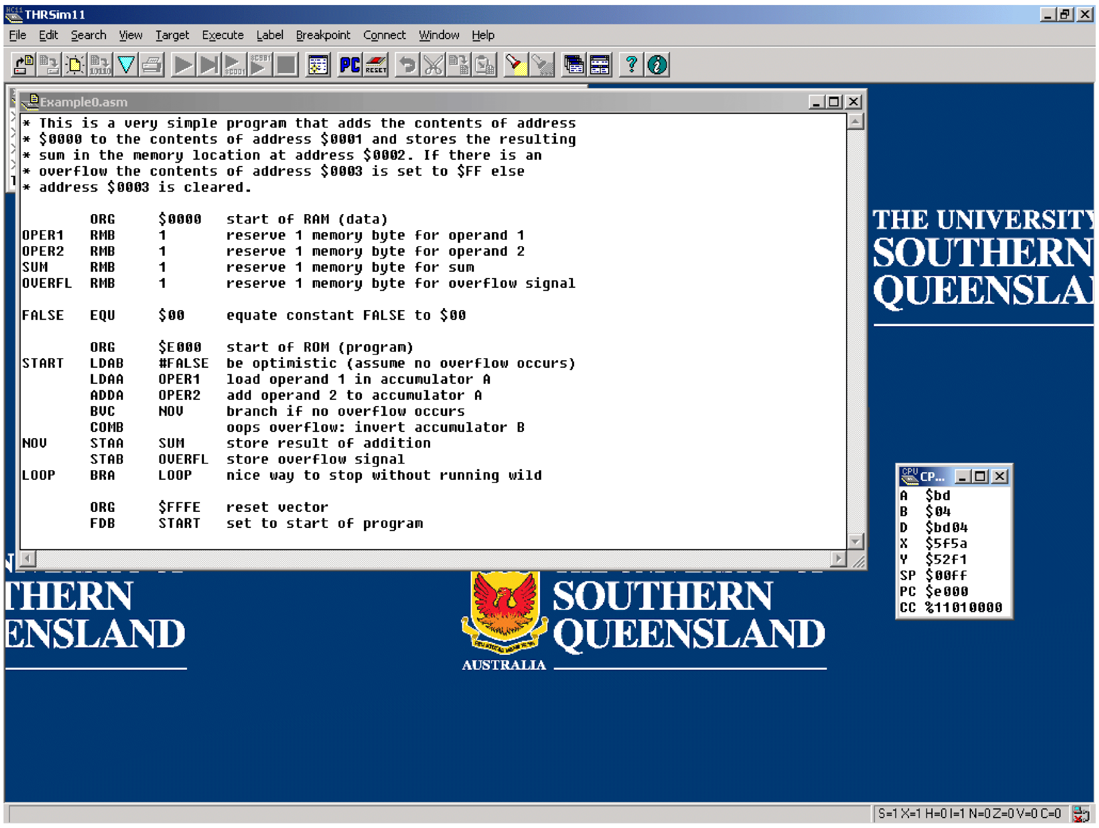
- *LED's,*
- *switches,*
- *analog sliders (variable voltage potential).*
- *serial transmitter and receiver.*
- *many more... see the THRSim11 components page.*

There is also a 4 x 20 LCD character display mapped in the address space of the 68HC11.

THRSim11 can communicate with the Motorola EVM and EVB boards or with any other board running the BUFFALO monitor program. When your assembly program is loaded into the target board the graphical user interface makes it possible to view and control every register (CPU registers and I/O registers) and memory location (data, program, and stack) of the real microcontroller. It is possible to stop the execution at any address and inspect or change the registers and memory.

This software will be used to program and simulate the operation of the 68HC11 microcontroller. The software and installation instructions are provided to you on the resource CD included in this study package.

All program examples which are provided hereafter are able to be assembled and simulated using this software.


THRSim11
File Edit Search View Target Execute Label Breakpoint Connect Window Help
Example0.asm
CP...
S=1 X=1 H=0 I=1 N=0 Z=0 V=0 C=0

```
* This is a very simple program that adds the contents of address
* $0000 to the contents of address $0001 and stores the resulting
* sum in the memory location at address $0002. If there is an
* overflow the contents of address $0003 is set to $FF else
* address $0003 is cleared.

        ORG     $0000   start of RAM (data)
OPER1   RMB     1       reserve 1 memory byte for operand 1
OPER2   RMB     1       reserve 1 memory byte for operand 2
SUM     RMB     1       reserve 1 memory byte for sum
OVERFL  RMB     1       reserve 1 memory byte for overflow signal

FALSE   EQU     $00     equate constant FALSE to $00

        ORG     $E000   start of ROM (program)
START   LDAB    #FALSE  be optimistic (assume no overflow occurs)
        LDAA    OPER1   load operand 1 in accumulator A
        ADDA    OPER2   add operand 2 to accumulator A
        BVC     NOV     branch if no overflow occurs
        COMB            oops overflow: invert accumulator B
NOV     STAA    SUM     store result of addition
        STAB    OVERFL  store overflow signal
LOOP    BRA     LOOP    nice way to stop without running wild

        ORG     $FFFE   reset vector
        FDB     START   set to start of program
```

```
A  $bd
B  $04
D  $bd04
X  $5f5a
Y  $52f1
SP $00ff
PC $e000
CC %11010000
```

(Source: THRSimm11 simulator software)

Figure 12.1: The THRSim11 graphical user interfave

12.3.3 Example program

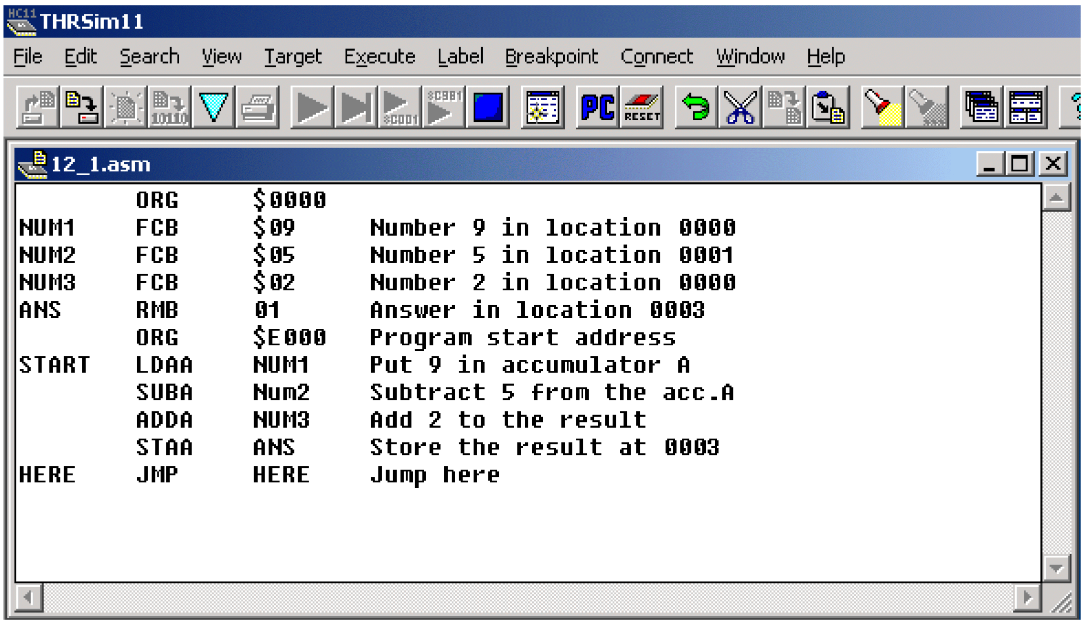
The program is created by selecting ‘New’ from the ‘File’ menu. The program is divided into a set of ‘fields’ as we have seen in previous work and, although not shown on the screen, they are named as shown below. Note that each field is separated by a field delimiter, in this case the ‘Tab’ key.

When typing in a program you must define your data and any special areas of memory first. This is achieved using ‘**directives or pseudo mnemonics**’.

Pseudo mnemonic	Meaning	Usage	Definition definition
EQU ORG	equate origin	BUFFER EQU $1200 ORG $0010	defines the value of a label sets starting address of next byte of code
FCB	form constant byte	FCB $09 FCB $0976 FCB $A045FF	makes next byte=$09 makes next two bytes =$0976 makes next three bytes = $A045FF
RMB	reserve memory byte	DATA RMB n	leaves a decimal n byte gap in code for later use
PAGE	paginate	PAGE	instruction for use by printout routine only. Forces page-break in listing.

In this example we will use the basic program we have developed and coded previously. That is, developing a program to execute the problem: 9 - 5 + 2 = ANS. As before we will give the numbers 9, 5 and 2 labels as NUM 1, NUM2 and NUM3 respectively, and the result will be stored in the location labelled as ANS.

Editor listing


THRSim11
File Edit Search View Target Execute Label Breakpoint Connect Window Help
12_1.asm

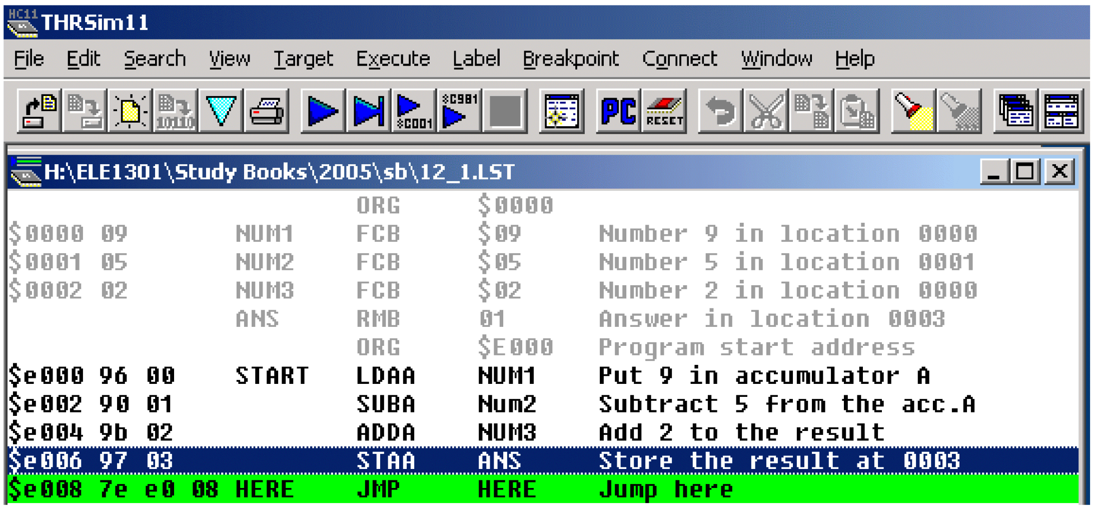
```
        ORG     $0000
NUM1    FCB     $09     Number 9 in location 0000
NUM2    FCB     $05     Number 5 in location 0001
NUM3    FCB     $02     Number 2 in location 0000
ANS     RMB     01      Answer in location 0003
        ORG     $E000   Program start address
START   LDAA    NUM1    Put 9 in accumulator A
        SUBA    Num2    Subtract 5 from the acc.A
        ADDA    NUM3    Add 2 to the result
        STAA    ANS     Store the result at 0003
HERE    JMP     HERE    Jump here
```

Note the use of ‘ORG’ to define the starting address at location $0000 for data and $E000 for the program.

If the display is satisfactory then the appropriate SAVE option is chosen from the ‘File’ menu.

The program is then be assembled by selecting ‘Assemble’ from the ‘File’ menu. Assembly will then occur and if any errors are present, they will be reported on the screen.

If there are no errors, the assembler output will appear in a new dialogue box. The following is an example of the assembler output.


THRSim11
File Edit Search View Target Execute Label Breakpoint Connect Window Help
H:\ELE1301\Study Books\2005\sb\12_1.LST

```
                         ORG    $0000
$0000 09        NUM1     FCB    $09     Number 9 in location 0000
$0001 05        NUM2     FCB    $05     Number 5 in location 0001
$0002 02        NUM3     FCB    $02     Number 2 in location 0000
                ANS      RMB    01      Answer in location 0003
                         ORG    $E000   Program start address
$e000 96 00     START    LDAA   NUM1    Put 9 in accumulator A
$e002 90 01              SUBA   Num2    Subtract 5 from the acc.A
$e004 9b 02              ADDA   NUM3    Add 2 to the result
$e006 97 03              STAA   ANS     Store the result at 0003
$e008 7e e0 08 HERE      JMP    HERE    Jump here
```

Once the program is assembled it can be simulated by selecting ‘Run’ from the ‘Execute’ menu. Toolbar buttons are also available for these functions.

Table 12.1: THRSim11 simulator default memory map

THRSim11 – 68HC11 Simulator

Memory type	Address	Labels	Description	
RAM (Data)	0000		***Use for program data***	Total available memory space in this processor
	|			
	00FF	STACK	Default stack address	
Not used	0100			
	|			
	0FFF			
I/O Registers	1000	PRA	Port A data	
	|			
	1003	PRC	Port C data	
	1004	PRB	Port B data	
	1006	DDRB	Port B direction	
	1007	DDRC	Port C direction	
	1008	PRD	Port D data	
	1009	DDRD	Port D direction	
	|			
	1040	LCDD	LCD data	
	1041	LCDC	LCD control	
RAM	B600		Used to simulate 68HC11 internal EEPROM	
	|			
	B7FF			
ROM (Program)	E000		***Use for program code***	
	|			
	FFF2	IRQ	IRQ pin interrupt vector	
	|			
	FFFE	RESET	Reset button vector	
	FFFF			

Activity 12.1

Refer to the ELE1301 course page on ‘Study Desk’ and complete the following experiments:

3. Home experiment 12-1 – Introduction to the THRSim11 software
4. Home experiment 12-2 – Introduction to assembly language programming.
5. Home experiment 12-3 – Directives in assembly language programming
6. Home experiment 12-4 – Introduction to program simulation.

Self assessment 12.3

1. Using the instruction set provided for the 68HC11 microcontroller, write a program that will sum the contents of memory addresses $0000 through $0004 and store the result in address $0005.
   - i. Start the program at address $E000
   - ii. Use indexed addressing to access the data
   - iii. Add assembler directives as required
   - iv. Set out the program using the following column heading.

     Address Opcode Label Mnemonic Operand Comments

     Values to be summed:

     $0000 - $01

     $0001 - $02

     $0002 - $03

     $0003 - $04

     $0004 - $05
2. Draw a flow chart for the above program.
3. Explain the operation of the above program.
4. Write a fully coded program, using the instruction set provided, to successively subtract the decimal number 15 from the decimal number 75 and when the result is negative store it to memory. Neatly set out your work to provide:
   - i. Flow chart
   - ii. Memory map
   - iii. Assembly language program
   - iv. Machine coded program