TOM *The* Computer Simulator

For Windows 95, 98, NT

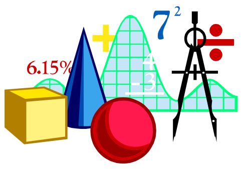
A self study guide

Version 2.0

Keylink Computers Ltd,
2 Woodway House,
Common Lane,
Kenilworth CV8 2ES

Tel: +44 1926 50909
Fax: +44 1926 864128

CONTENTS



1 Introduction

TOM stands for Thoroughly Obedient Moron and is a computer simulation designed to teach first time users how a computer works. It is entirely mouse driven, so no keyboard skills are required at all. TOM allows simple programs to be input and executed whilst the memory and registers are visible. This gives a very immediate feel of what programming is all about. Concepts such as 'flow of control' can be immediately appreciated because the execution path is made apparent.

TOM has 20 instructions, a manageably small number which nevertheless enables real programs to be written. The entire status of TOM is visible at all times giving instant feedback on success or problems. Programs may be executed, single stepped, saved, restored and modified. Saved programs are entirely readable, so they may be printed for the student's later revision.

TOM is **entirely** mouse driven and offers a multi-tasking, friendly and intuitive interface.

This latest version of TOM gives an interactive visualisation of how the actual underlying machine operates. It shows how the various components of a computer, the registers, the memory, the Arithmetic Logic Unit, the Address Decoder and the Bus are connected together and execute programs. In addition there are interactive screens that show: how logic gates are used to construct flip-flops and how these are used to build memory, even how the Address Decoder works. TOM is designed to give the student a complete grasp of the fundamental operation of a machine.

TOM enables inexperienced users to get to grips with the concepts of machine code while ignoring the complications of using a real assembler or development environment.

TOM addresses computing studies syllabuses that have topics which include computer (machine) architecture and operation. In addition it makes an ideal introduction to high level languages as it teaches the fundamental components of an algorithm and how a computer stores and processes data.

The scope of this manual has been extended to cover a great deal of new material. It is hoped that it will provide a useful teaching document as well as just being a manual on how to use TOM.

If you have difficulties with any of the terms use, please note that there is a glossary of terms in one of the appendices and in the on-line help file..

TOM was conceived and first implemented by Alan Chantler of Coventry University where it was used successfully for many years in the introduction to the Computer Science degree course. This implementation was designed and coded by Rob Lucas of Keylink Computers Ltd. At the request of Alan Chantler and in lieu of royalties a donation of a proportion of the purchase price is made to the Imperial Cancer Research Fund.

If you have any comments or suggestions for improvements then please let us at Keylink know.

Rob Lucas

Kenilworth, November 1995

2 Installation

2.1 Copying TOM's files

TOM is now distributed with an installation script which must be used to install TOM. As the installation files are stored in compressed form, the files cannot simply be copied from the floppy to a hard disk and then executed. For the same reason TOM will not run from the supplied floppy. The following installation instructions must be followed.

2.2 Creating a TOM application

Start Windows, and make the Program Manager the active window. Click on the Files option and select the Run option from the Files menu. Type 'a:setup.exe' (substitute for 'a' if you are installing from a different floppy drive). Now click on OK. The set-up program will create an application group called TOM.

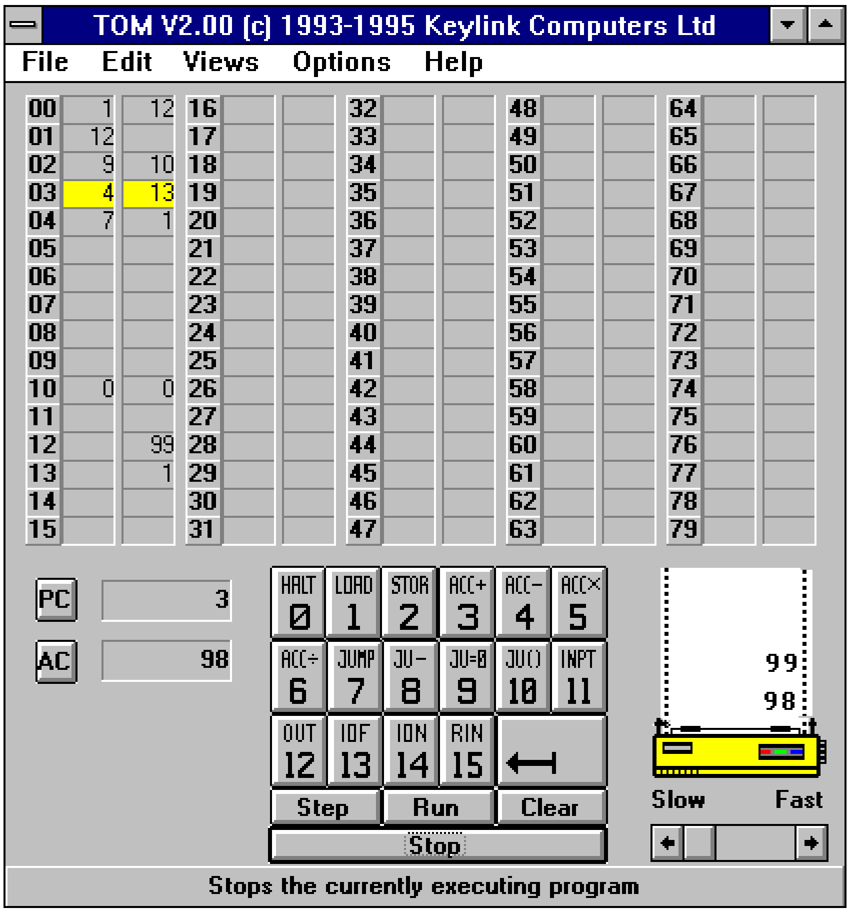
TOM is now installed and has an icon which looks like an abacus. Leave the installation disk in the drive. Double click on the TOM application group to open it and then double click on the TOM icon to start TOM. TOM will report that it has no initialisation file present in its directory, but will copy it from the installation disk when you reply Yes to this dialogue. This dialogue will not appear again and there is no need to use the installation disk when running TOM again.

3 TOM's components

TOM’s main screen is shown below. Below the Copyright heading are the Menu options.

The File menu options allow loading and saving of TOM programs. The Edit option has delete, copy and paste functions. The Views options allow you to display TOM’s other screens. The Options menu allows you to select various display settings, such as Hexadecimal. The Help menu option gives you access to a comprehensive on-line Windows help facility.

The upper part of the screen consists of TOM’s **Memory locations** which are labelled **00 to 79.**



TOM V2.00 (c) 1993-1995 Keylink Computers Ltd

00	1	12	16			32			48			64		
01	12		17			33			49			65		
02	9	10	18			34			50			66		
03	4	13	19			35			51			67		
04	7	1	20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10	0	0	26			42			58			74		
11			27			43			59			75		
12		99	28			44			60			76		
13		1	29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 3

AC 98

```
99
98
```

Stops the currently executing program

The **Register area** consists of the **Program Counter** and the **Accumulator**. The **Program counter** is labelled **PC** and the **Accumulator** is labelled **AC**.

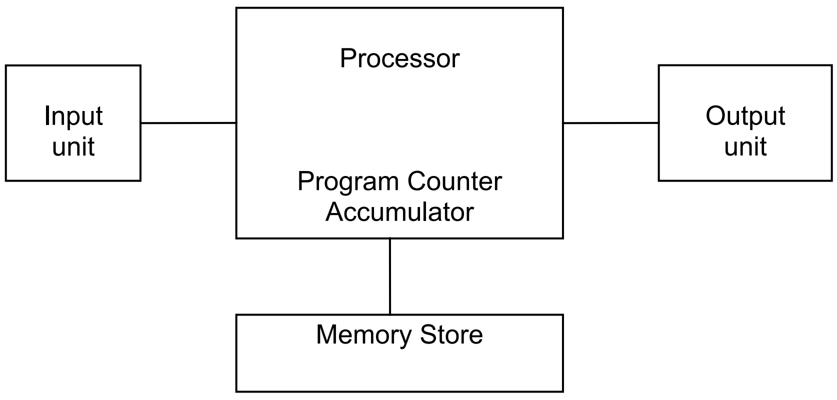
The **Keypad** is the central area of buttons, you use these to enter values into TOM's memory and to control the operation of TOM.

The **Output area** shows a printer icon and underneath this is the **Accelerator** which controls the speed at which TOM performs its calculations.

At the bottom of TOM's screen is a status bar. This displays informative messages that usually concern the component that the mouse is currently pointing to. When the mouse pointer is on TOM's memory, the status bar gives the numeric contents of the memory location in binary.

4 TOM's architecture

TOM has a single accumulator, a memory store of 80 locations and the means to input and output numbers. This can be described by the following diagram:



TOM simulates all of these components in simple ways. The function and operation of each component is described below.

4.1 Memory store

Each memory location has a numeric label to the left of it which is the address of that memory location. The address of a memory location is used in our programs to identify which memory location we mean in the same way as we might use a postal address to identify someone's house.

Each memory location consists of two words. The most significant word is the left of the two and the least significant is the right hand one.

Each word is exactly one byte (eight bits) which is precisely enough memory to store any of the numbers 0 to 255. You need not concern yourself with why this number is what it is for the moment, as it will be fully explained in a later chapter.

The contents of a memory location may be interpreted in two ways:

As data:

both words make up an integer value which is calculated from the most significant word * 256 + the least significant word. In practice, most of our integer values will be less than 256 so we can ignore the left hand byte when dealing with data values. Negative values are ignored until we get to a much more advanced stage.

Example:

12 | 2| 12|

this shows memory location 12 with 2 in the most significant word and 12 in the least significant word. When interpreted as a data value, this value is:

2 * 256 + 12 = 524

As instructions:

The most significant word is the instruction number and the least significant word is the operand, that is it is the argument that the instruction is to operate on (this is usually a memory address).

Example:

12 | 2| 12|

this shows memory location 12 with 2 in the most significant word and 12 in the least significant word. When interpreted as an instruction it means: apply instruction number 2 to the operand value 12. In TOM, instruction 2 means store the contents of the accumulator at the address given in the operand. So the example means: store the contents of the accumulator at memory location 12.

Notice that both examples were the same, but that they can mean different things depending on whether they are interpreted as data or an instruction/operand pair.

4.2 A Tom program

What we mean by a program is nothing other than a collection of instructions that can be entered into the memory locations and then executed. With TOM you will usually enter a program directly into memory by clicking on the memory cells and entering values using the keypad.

4.3 The Program Counter (PC)

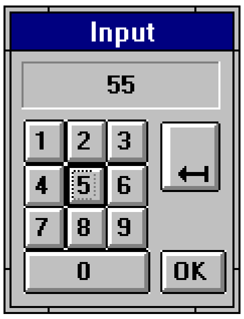
TOM, and any computer for that matter, works by repeatedly fetching instructions from memory and executing them. It keeps track of which instruction to execute next by storing its address in the **Program Counter**. That is, the Program Counter always contains the address of the next instruction to be executed. We usually start with the Program Counter set to 0, so the first instruction executed is that stored at memory location 0. Instructions are then normally executed in sequential order, i.e. execute instruction at location 0, followed by that at location 1, followed by that at location 2, and so on. The exception to this is when one of the **jump** instructions is executed which alter the flow of control of the program. See next chapter for details of jump instructions.

4.4 The Accumulator (AC)

The accumulator is used to store values that the computer is currently using. It is entirely general purpose and may contain a data value, an address or even an instruction number. It all depends on what the person who designed the program wants to store in it. TOM has several instructions which act on the Accumulator, including inputting a number which is stored in the Accumulator (see below), and adding to and subtracting from the Accumulator.

4.5 The Input Unit

The input unit allows values to be input to the executing program. In a real computer there would be many input units capable of supporting all kinds of input device (mouse, keyboard etc.). In TOM we have a simple form, which looks a little like a calculator:



You may enter a number by clicking on the buttons marked with digits, when you have completed your number, type OK. The last character input can be deleted by using the key with the arrow.

Whenever the instruction number (or **opcode**) 11 is executed, the dialogue box will appear on the screen and a number can be entered. If the number is too large, the number will be rejected.

4.6 The Output Unit

The output unit allows the program being executed to output values. In a real computer there would be output units corresponding to all the different kinds of output devices, such as the screen and the printer. In TOM we have one simple scrolling output area in the bottom right of TOM's window represented by a printer icon.

The output unit will output a number from the accumulator to the output area of TOM's screen.

Later chapters will show how a memory-mapped output device can also be used in TOM programs.

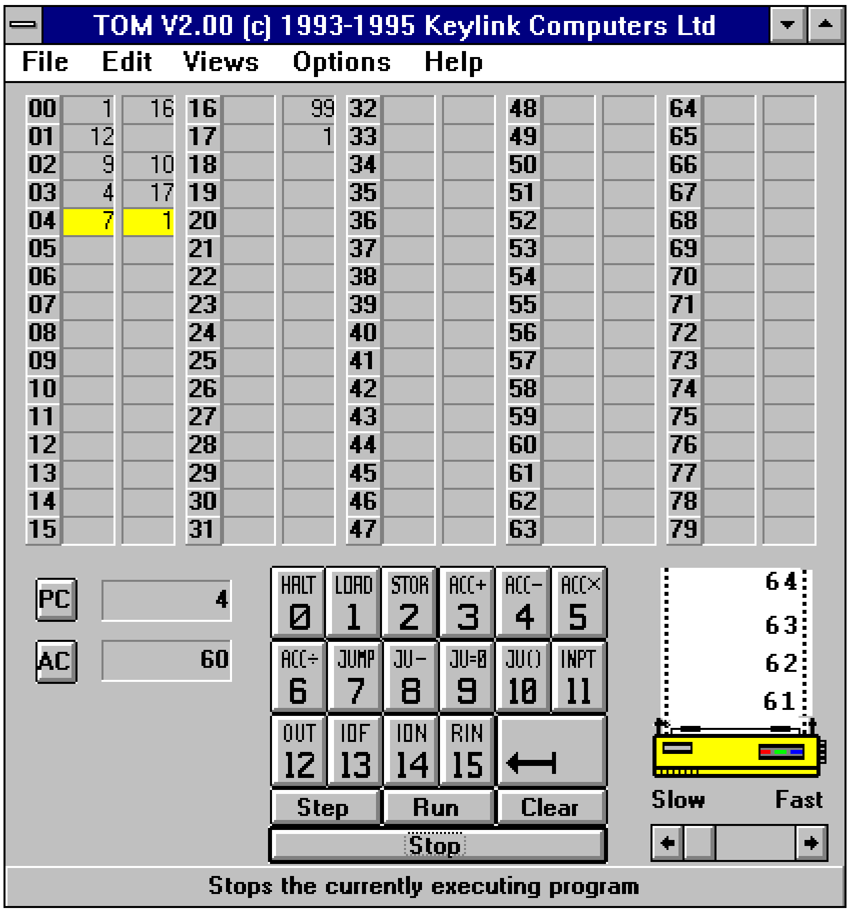
4.7 The Processor

The processor is the part of the computer that actually effects the execution of the instructions. It contains various internal registers, in particular the **Program Counter** and the **Accumulator**. To program TOM we don't need to know how this part works, but we do need to know the effects each of the possible instructions has. These effects are described in the next chapter which describes TOM's instruction set.

For those interested in how the processor works, TOM provides an interactive simulation of the internal workings of its processor, this is described in a later chapter.

Exercises A

Consider the following screen dump of TOM in a particular state mid-execution.



00	1	16	16		99	32			48			64		
01	12		17		1	33			49			65		
02	9	10	18			34			50			66		
03	4	17	19			35			51			67		
04	7	1	20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10			26			42			58			74		
11			27			43			59			75		
12			28			44			60			76		
13			29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 4

AC 60

```
64
63
62
61
```

Stops the currently executing program

(1) What are the contents of memory location 16?

(2) What is the value of the contents of memory location 4 when interpreted as a number?

(3) In the same diagram, given that instruction number 12 means output the contents of the accumulator, what will be the output if the instruction at location 1 is executed immediately after the instruction at memory location 0?

(4) What memory location is being indicated by the program counter?

(5) What is the address of a memory location for?

(6) What is stored in the program counter?

(7) What are the two kinds of interpretation that can be applied to numbers stored in the memory of a computer?

(8) What do we mean by a program?

5 TOM's Basic instruction set

5.1 Introduction

TOM has quite a comprehensive instruction set. If all of these instructions were to be described at once you would probably become rather confused. So below are described the basic instructions which will nevertheless enable you to write some substantial programs. More of the instruction set is introduced as and when needed and there is an entire list with their descriptions in Appendix A.

5.2 Mnemonics

Instructions are represented by simple integer numeric values. This is how the computer likes them to be represented. However, it is difficult to understand a program which is just a list of numbers. For this reason we associate a short hand alphabetic code with each instruction which gives us a much better idea of what the instruction is. For example the instruction that loads the accumulator with a value from a memory location is 12, we represent this for convenience as LDA (LoaD Accumulator), these are called mnemonics.

The numeric form of a program is often called machine code, whereas the mnemonic form is commonly called assembler. The Options menu gives you the choice of representing programs in machine code and assembler. The use of assembler mnemonics will be returned to when we actually start programming.

In the list of instructions given below, the numeric value of the instruction is given followed by its mnemonic form in brackets.

5.3 Basic Instructions

There are thirteen basic instructions numbered 0-12. These numbers are frequently referred to as instruction codes, machine codes, operation codes or opcodes for short. Remember that, when dealing with instructions, the left-hand word in a memory location holds the **opcode**, while the right-hand word holds the **operand**, or argument, and, depending on the opcode, the operand may represent a memory location or a data value. Some opcodes do not have an operand, for example HALT.

In the description of TOM's opcodes below, the number of the opcode corresponds to the number on the 'keypad' in the bottom middle of TOM's screen which is used for input of instructions and data.

0 (HLT) HALT

Causes execution of the TOM program to cease. No operand.

1 (LDA) LOAD ACCUMULATOR

Place a copy of the contents of the memory location given as the operand into the accumulator. The contents of the memory location are unchanged.

2 (STO) STORE ACCUMULATOR

Place a copy of the contents of the accumulator into the memory location given as the operand. The contents of the accumulator are unchanged.

3 (AC+) ADD TO ACCUMULATOR

Place into the accumulator the arithmetic sum of the contents of the memory location given as the operand and the current contents of the accumulator. The contents of the memory location are unchanged.

4 (AC-) SUBTRACT FROM ACCUMULATOR

Place into the accumulator the result of subtracting the contents of the memory location given as the operand from the current contents of the accumulator. The contents of the memory location are unchanged.

5 (ACX) MULTIPLY ACCUMULATOR

Place into the accumulator the product of the contents of the memory location given as the operand and the current contents of the accumulator. The contents of the memory location are unchanged.

6 (AC/) DIVIDE ACCUMULATOR

Divide the current contents of the accumulator by the contents of the memory location given as the operand and place the resulting quotient into the accumulator. The remainder is lost and the original contents of the store location are not affected.

7 (JMP) JUMP UNCONDITIONALLY

Control is transferred to the memory location whose address is given as the operand. Jump instructions alter the sequence in which a program is executed.

8 (JM-) JUMP IF ACCUMULATOR NEGATIVE

If the contents of the accumulator is a negative number then control is transferred to the memory location whose address is given as the operand, otherwise program execution continues at the next sequential instruction.

9 (JM0) JUMP IF ACCUMULATOR IS ZERO

If the contents of the accumulator is zero then control is transferred to the store location whose address given as the operand, otherwise program execution continues at the next sequential instruction.

10 J() JUMP INDIRECT

Jump unconditionally to the address stored at the address given by the operand. Only the least significant word is considered to form the address.

11 (INP) INPUT A NUMBER TO ACCUMULATOR

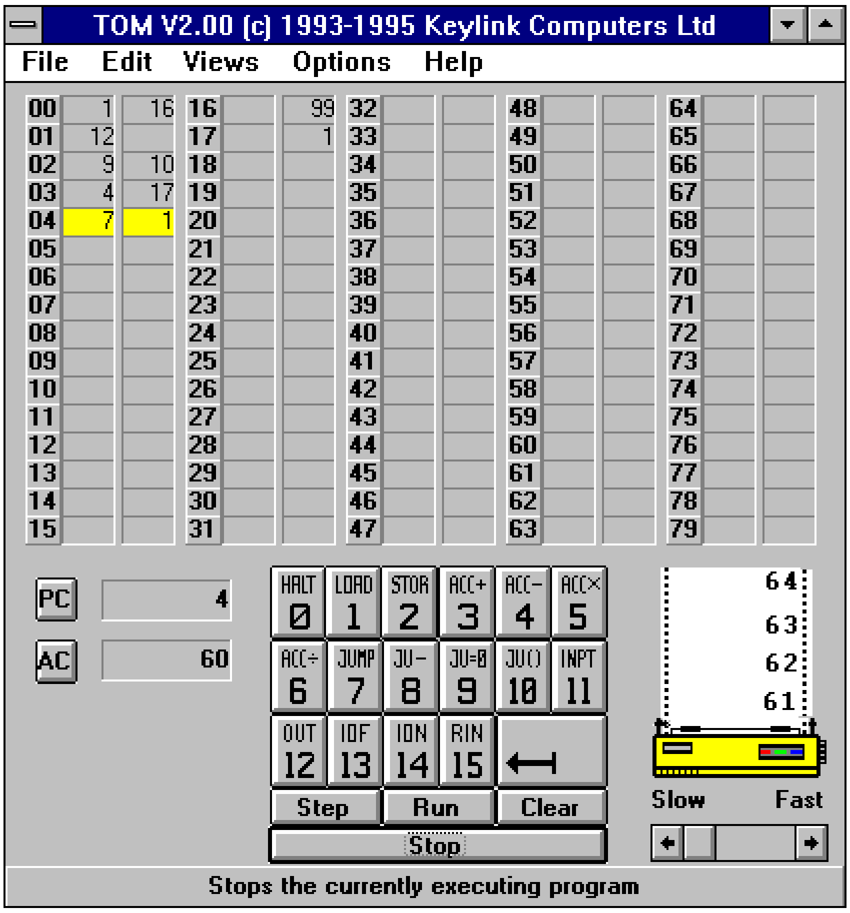
A dialogue box is displayed into which you can type in a number of up to six digits. If this is a legal number it is placed into the accumulator when you click the OK button. No operand.

12 (OUT) OUTPUT A NUMBER FROM THE ACCUMULATOR

The contents of the accumulator is output as a number to the output area of TOM's screen. No operand.

Exercises B

Consider the following screen dump of TOM in a particular state mid-execution.



00	1	16	16		99	32			48			64		
01	12		17		1	33			49			65		
02	9	10	18			34			50			66		
03	4	17	19			35			51			67		
04	7	1	20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10			26			42			58			74		
11			27			43			59			75		
12			28			44			60			76		
13			29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC	4
AC	60

```
64
63
62
61
```

Stops the currently executing program

(1) What do the contents of memory location 0 mean when interpreted as an instruction?

(2) If the instruction at memory location 0 were to be executed, what would the contents of the accumulator be?

(3) What do the contents of memory location 1 mean when interpreted as an instruction?

(4) If the instruction at memory location 1 were to be executed after the instruction at location 0, what would the contents of the accumulator be?

(5) What do the contents of memory location 2 mean when interpreted as an instruction?

6 Using TOM

6.1 Starting TOM

Start Windows and run TOM by double clicking on its icon (an abacus). TOM's screen will now appear. All memory locations are empty, and the Accumulator and Program Counter are both set to zero.

6.2 Stopping TOM

TOM may be exited by clicking on **File** in the top left of the screen and selecting **Exit** from the File menu.

6.3 Entering numbers into memory

To put a number into a memory cell, click on the memory cell's low or high word and then click on the keypad number(s) you want to enter. The contents of a memory cell can be removed by clicking on the arrow key at the bottom right of the keypad.

The keypad indicates which instructions correspond to which numbers. For example the key for 5 indicates that the corresponding instruction opcode is **ACCX** which is Multiply Accumulator. Clicking on this puts the number 5 into the current memory cell. The instruction mnemonics serve simply as a reminder of what numbers correspond to what instructions.

6.4 Single stepping

Clicking the Step button will cause a single instruction to be fetched from the address indicated by the current setting of the Program Counter and for this instruction to be executed. Successive clicks on the Step button will cause successive instructions to be fetched and executed. This is useful for debugging your TOM programs as it enables you to see easily the contents of memory locations, the Program Counter and Accumulator as each instruction is executed.

6.5 Running a TOM program

Clicking the Run button will start TOM executing from the instruction stored in the address indicated by the current setting of the Program Counter. The program will execute until the HALT statement is reached or an error occurs. Execution speed may be altered using the accelerator bar at the bottom right of the screen. Note that execution of an instruction other than a HALT or JUMP at location 79 will cause the program counter to be incremented to 80, causing TOM to 'go off the end' of memory. This will cause an 'invalid address' error.

6.6 Inputting a value

When the Input instruction (opcode 11) is executed a dialogue window will appear requiring a number of up to six digits to be input. Having entered a number, clicking the OK button will cause the number to be stored in the accumulator, provided it is a valid number, and for program execution to resume at the next sequential instruction.

6.7 Saving a TOM program

The current state of the memory (i.e. the current program) can be saved by selecting **Save** or **Save As** from the **File** menu. Click on File at the top left of the screen, and then on the Save option from the displayed menu. Enter a name for the file and click on the OK button, or the Cancel button to abort the operation. The saved file may be printed and is intelligible but it is recommended that you do not attempt to edit it. If you need to change a TOM program then load it into TOM, change it, and then re-save it.

Whenever you save a TOM program, the input and output sequence from the last time you clicked on Run will be saved with the program. This gives you a record of your program executing that you may wish to print. This part of the saved file is of no consequence when loading it into TOM.

6.8 Loading a TOM program

Click on File followed by Open from the File menu. Click on the name of the saved program you wish to load and click the OK button to load it or on Cancel to abort the operation. If the file is legal (i.e. it is a saved TOM program), the current state of TOM will be lost and the TOM file will be loaded. The new state of the memory will be displayed.

6.9 Printing a TOM program

Selecting Print from the file menu will cause the current TOM program to be output to the printer.

6.10 Resetting the Program Counter

The program counter can be reset to zero by clicking on the button labelled 'PC'.

6.11 Resetting the Accumulator

The Accumulator can be reset to zero by clicking on the button labelled 'AC'.

6.12 Halting an infinite loop

If you manage to code a program that won't exit, you can stop the program by clicking on the Stop button at the bottom centre of TOM's window.

6.13 Clearing TOM's memory

When you have completed an example and, possibly, saved your program to a file, you may wish to start another program. Clicking the Clear button will clear all memory locations and the output area and set the

Accumulator and Program Counter to zero. Use this option with caution! The same result can be achieved by clicking on New in the File menu.

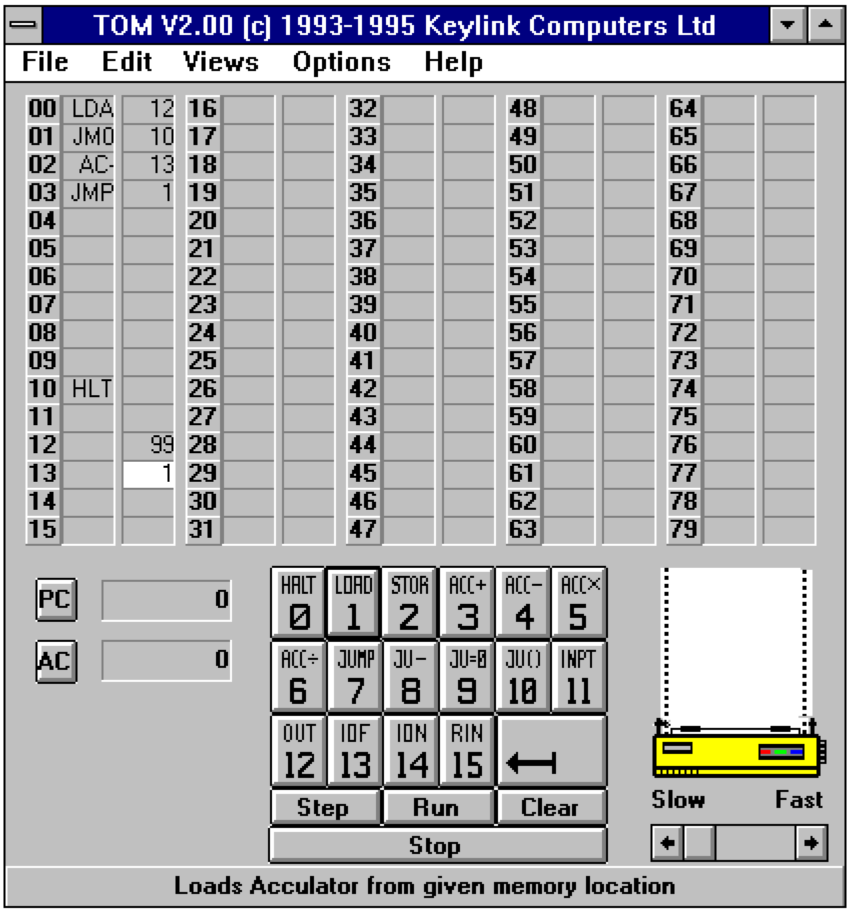
6.14 Altering execution speed

The speed of program execution may be controlled using the accelerator bar at the bottom right of the screen. Clicking the left-facing arrow will slow execution, clicking the right-facing arrow will increase speed. Dragging the accelerator slider left and right has the same effect.

6.15 Cutting and Pasting

If you need to insert one or more instructions between two adjacent memory locations, rather than rewrite a large part of your program, you may find it convenient to cut and paste code from one location to another. For example:

Consider the following state:



00	LDA	12	16			32			48			64		
01	JMO	10	17			33			49			65		
02	AC-	13	18			34			50			66		
03	JMP	1	19			35			51			67		
04			20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10	HLT		26			42			58			74		
11			27			43			59			75		
12		99	28			44			60			76		
13		1	29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 0

AC 0

Loads Acculator from given memory location

Here I've programmed a simple loop which runs from 99 down to 0, but I've forgotten to output the value. What I would like to do is insert the instruction 12 (Output) between the instruction at location 0 and the instruction at location 1.

I can do this by cutting and pasting.

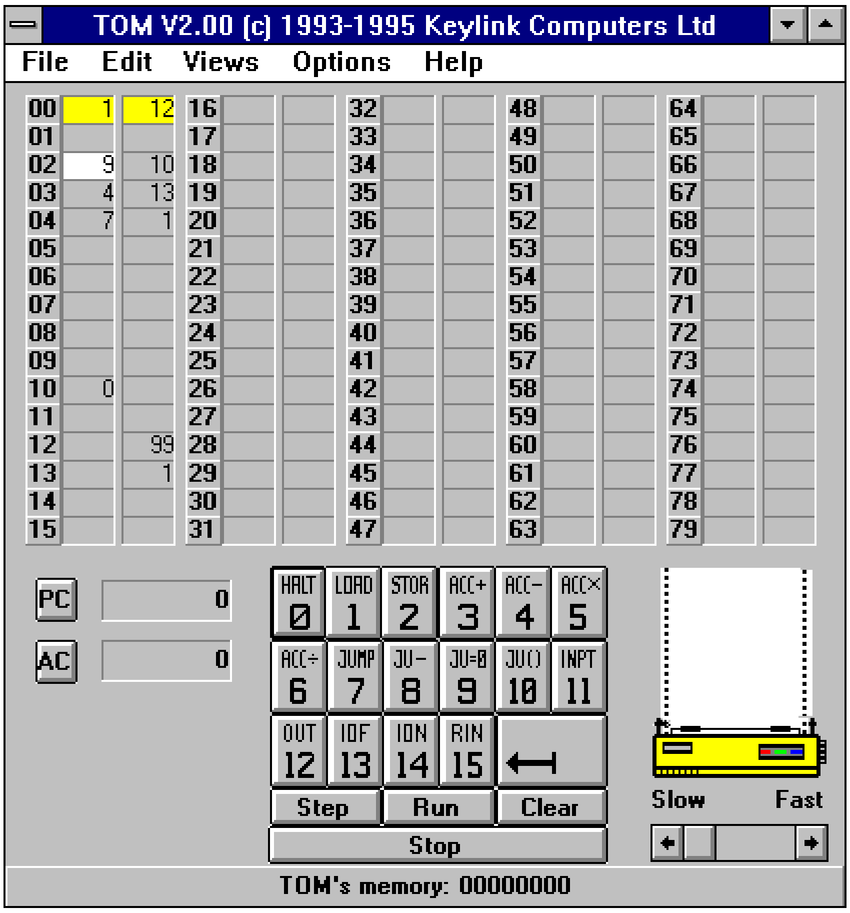
First I highlight the cells that I need to move. I do this by holding down the shift key and clicking on the cells I want to move. i.e. I hold down the shift key and click on the locations at addresses 01, 02 and 03. This displays the highlighted cells in green.

I then select **Cut** from the **Edit** menu. At this point the contents of cells 01, 02 and 03 are cleared.

I now click on the start address of the destination, which is at address 02.

I then select **Paste** from the **Edit** menu.

At this point I am looking at:



TOM V2.00 (c) 1993-1995 Keylink Computers Ltd

File Edit Views Options Help

00	1	12	16			32			48			64		
01			17			33			49			65		
02	9	10	18			34			50			66		
03	4	13	19			35			51			67		
04	7	1	20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10	0		26			42			58			74		
11			27			43			59			75		
12		99	28			44			60			76		
13		1	29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 0

AC 0

TOM's memory: 00000000

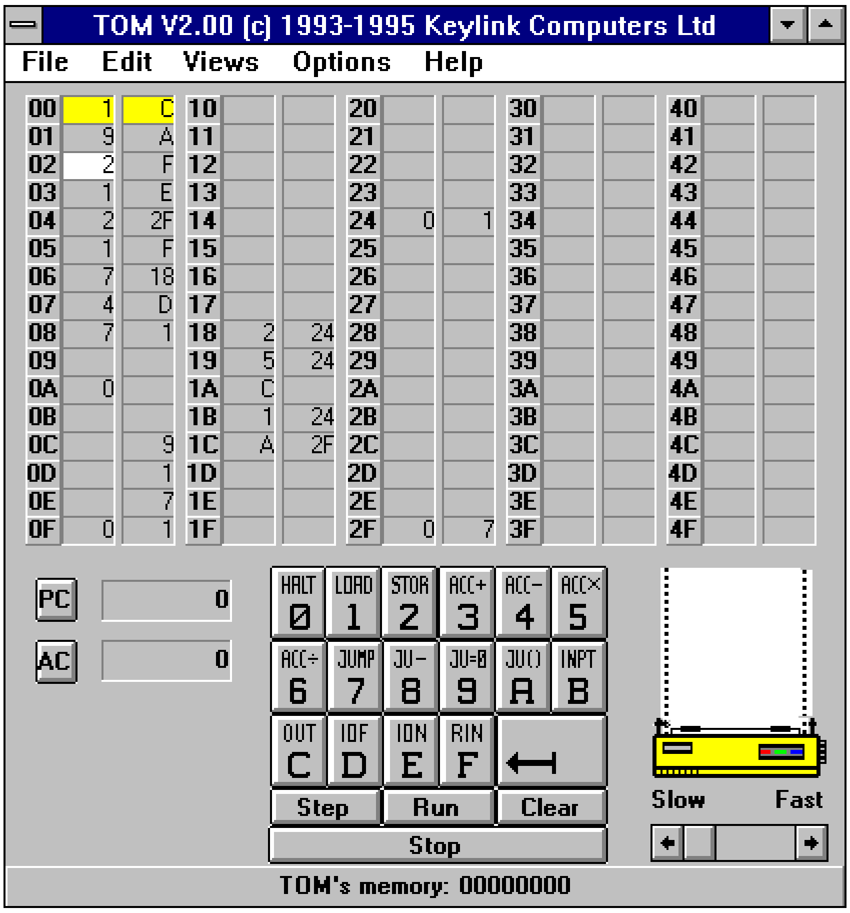
All that remains is for me to insert instruction 12 into location 01, which I do in the normal way.

6.16 Hexadecimal

It is possible to represent the numbers in the registers and memory in either decimal (default) or hexadecimal. Select either Decimal or Hexadecimal from the Options menu. When in hexadecimal mode, the keypad and memory labels are also altered to hexadecimal representation.

See the chapter on Number systems for an explanation of the decimal and hexadecimal numeric representations.

This shows the memory, keypad, and memory labels in hexadecimal mode:



00	1	C	10			20			30			40		
01	9	A	11			21			31			41		
02	2	F	12			22			32			42		
03	1	E	13			23			33			43		
04	2	2F	14			24	0	1	34			44		
05	1	F	15			25			35			45		
06	7	18	16			26			36			46		
07	4	D	17			27			37			47		
08	7	1	18	2	24	28			38			48		
09			19	5	24	29			39			49		
0A	0		1A	C		2A			3A			4A		
0B			1B	1	24	2B			3B			4B		
0C		9	1C	A	2F	2C			3C			4C		
0D		1	1D			2D			3D			4D		
0E		7	1E			2E			3E			4E		
0F	0	1	1F			2F	0	7	3F			4F		

PC 0

AC 0

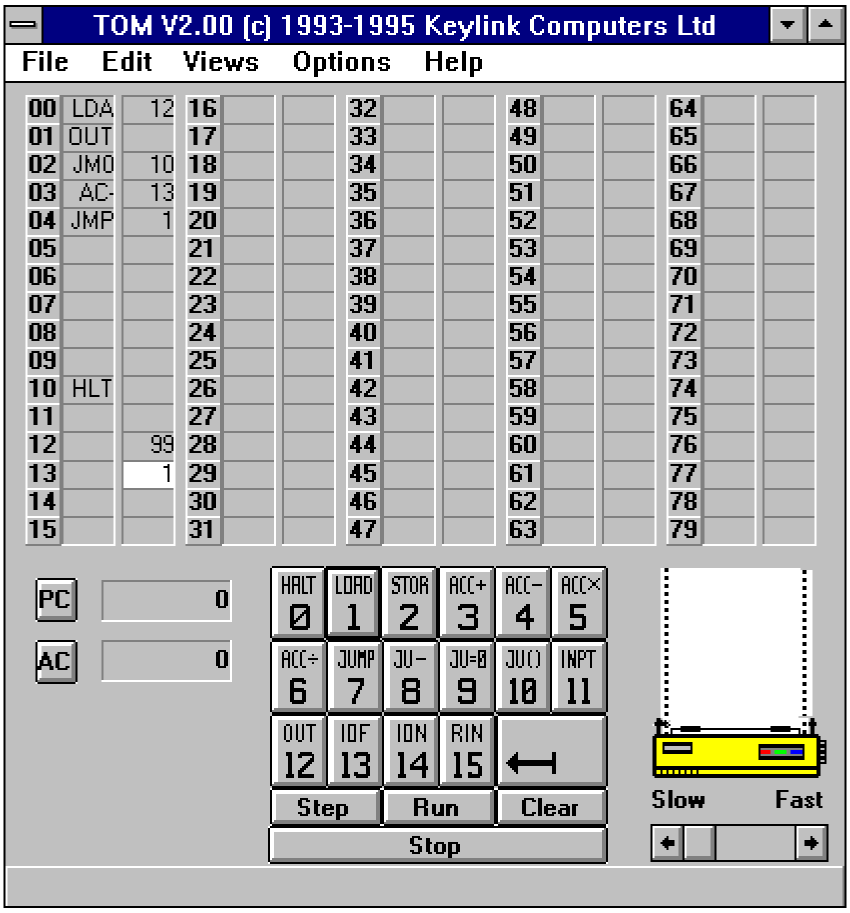
TOM's memory: 00000000

6.17 Disassembling

A program can be much more readable when it is presented as a combination of mnemonics and numbers rather than just plain numbers. The mnemonics used correspond closely to those on the keypad.

An assembler is commonly used as a mean of programming a micro processor. These always use mnemonics in place of the actual instruction names. They also provide other useful features. Hence, we call the translation process from instructions represented as mnemonics, to instructions represented as numbers as **assembly**. The reverse process of translating numbers to mnemonics as **disassembly**.

The following screen shot shows the above program displayed in its disassembled form (the missing instruction has been inserted):



00	LDA	12	16			32			48			64		
01	OUT		17			33			49			65		
02	JMO	10	18			34			50			66		
03	AC-	13	19			35			51			67		
04	JMP	1	20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10	HLT		26			42			58			74		
11			27			43			59			75		
12		99	28			44			60			76		
13		1	29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 0

AC 0

To translate the current contents of memory into mnemonics choose **Disassemble** from the **Options** menu. TOM will automatically convert the translated instructions back to numbers when it comes to executing them. It will also perform this assembly when the Machine view is selected from the **Views** menu

You can force Assembly by choosing **Assemble** from the **Options** menu.

7 Worked examples

In the following examples the contents of the memory are given in the same format as that used by Save and Print. That is: the memory location is the first number followed by the contents of the most and least significant word of the memory location. Memory locations which are not used are not listed. Each field is separated by a '|'. The algorithm is annotated on the right. To save space the accumulator is abbreviated to @ and a memory location is denoted by #. For example #24 means memory location number 24. In addition we use brackets to mean the contents of, as in: (#24) which means the contents of memory location 24.

Worked example 1

Load the number 55 from memory location 24 into the accumulator and output it.

This is a simple example to help you get used to using TOM. First of all, put the value 55 into the rightmost memory cell at location 24 by clicking on it and then clicking twice on keypad number 5.

Put the load accumulator instruction (opcode 1) into the most significant word of memory location 0 (click on the left cell at location 0 and click on 1 on the keypad). Put its operand value, 24, into the least significant word (click on the right hand cell at location 0 and then click on 2 and 4 on the keypad).

Put the instruction for output accumulator (12) into the most significant word at address 1. This does not need an operand.

Finally put the halt instruction, 0, into the most significant word at address 2. TOM's memory should now look like:

```
 0|  1| 24|  load @ from #24
 1| 12|   |  output @
 2|  0|   |  halt
24|   | 55|  value 55 at #24
```

Execute the program by clicking on the Step button three times. At each Step watch very carefully to see the program counter's value change from 0 to 1 to 2, watch how the Accumulator's value becomes 55 after execution of the first instruction, and how its value is 'printed' after execution of the second instruction.

Re-run the program by setting the program counter to 0 by clicking on the 'PC' button and clicking the Step button again until the HALT is executed, or, once you understand the flow of execution, press the Run button.

Save the program by using the Save menu option as described above, use the name ‘example1.tom’.

Worked example 2

Input two numbers, add them and output the result.

This demonstrates the use of the input unit, the use of memory for storing values via the STORE instruction, the use of the ADD instruction, and the use of the output unit. Clear TOM’s memory by clicking on Clear then input the following values into memory using the same method as for example 1.

```
0| 11|   |  input a value to @
1|  2| 12|  store @ at #12
2| 11|   |  input another number
3|  3| 12|  add (#12) to @
4| 12|   |  output @
5|  0|   |  halt
```

When you run this program you will be prompted for a number on two occasions when the instruction being executed is Input value (11). When you are prompted for a number, enter it using the keypad and click on the OK button. Use Step to execute the program for the first time, and Run when you understand how it works.

Worked example 3

Output the numbers 9 to 1 by starting with 9 and decrementing (subtracting) by 1 until 0 is reached.

This example shows how a simple loop can be implemented. When programs start to be sufficiently complicated that they need to have loops and decisions then we find it very beneficial to plan our program using a form of 'Pidgin English' we call pseudo code. This gives us an abstract way of describing the program without worrying too much about the actual details of the code. In this case our pseudo code is:

```
for X = 9 to 1
   output X
repeat
```

The loop is implemented by loading a count into the accumulator, and using a jump if accumulator zero to determine when to exit the loop. We obviously need to subtract one from the count each time through the loop. We've indented the line between for and repeat to make it especially clear what statements are repeated.

```
0|  1| 12|  load @ with (#12)
1|  9| 10|  jump if @ zero to #10 (HALT)
```

2	12		output @
3	4	13	subtract (#13) from @
4	7	1	jump unconditionally to #1
10	0		halt
12		9	data value 9 stored at #12
13		1	data value 1 for decrement

The pseudo code may not have helped greatly here because our program was virtually all loop, but when the program is a bit more complicated, it becomes extremely convenient to think of a loop in the pseudo code way knowing that we can code it in actual instructions by using the above translation. After all, which do you find easier to understand, the pseudo code or TOM's machine instructions?

With the contents of memory as described above and the Program Counter set to zero, click on Step to single step through the program or Run. You should see the program looping, performing the output, and finally exiting.

Exercises C

(1) Write a program to input a pair of integers and output them with the largest first.

Think of the solution in pseudo code as:

```
input X
input Y
if X > Y then
   output X
   output Y
else
   output Y
   output X
endif
```

where 'input X' can be translated into a TOM input followed by a store instruction (a different address can be used to differentiate the X from the Y, i.e. store X at one location and Y at another). The 'if' statement can be translated by using a Subtract from Accumulator followed by one of the JUMP instructions. Remember that after a Store Accumulator, the contents of the Accumulator are unchanged. The 'else' statement may be coded either as a Jump Unconditional (to a HALT instruction), or in this example, as there is nothing to do after it, directly as a HALT. We've indented between the 'if', 'else' and 'endif' to make it especially clear which statements correspond to what conditions.

Translate this into TOM's instructions and test that your solution works. If it doesn't then single step through the execution of the program and determine where it is going wrong. Make a correction and try again.

(2) Write a program to input a set of positive integers terminated by zero and output the arithmetic mean.

The following pseudo code should be used as a model for your solution:

```
count = 0
sum = 0
input X
while X <> 0
   sum = sum + X
   count = count + 1
repeat
mean = sum / count
output mean
```

(3) Write a program to input a positive number, N, followed by N further numbers, and print out N followed by the sum of the N numbers. You should describe the program in pseudo code before coding it.

(4) Write a program to input a set of positive integers terminated by zero and output the largest, the smallest, and the range. You should describe the program in pseudo code before coding it.

8 Advanced examples

8.1 Worked example 4

Output the squares of the numbers 9 through 1.

We use the same technique as in the last worked example but use a 'subroutine' to calculate and output the square. The pseudo code for the main part of the solution is:

```
for X = 9 to 1
   square(X)
repeat
```

and:

```
Z = X * X
output Z
```

for the subroutine. This is a little trickier than it first appears because we must be careful not to leave unwanted values in the accumulator. So when we return from our subroutine by jumping into the main loop we must first restore the accumulator to its value when the subroutine was called.

```
 0|  1| 12|   load @ with (#12)
 1|  9| 10|   jump if @ zero to #10 (HALT)
 2|  7| 24|   jump to #24 to calc square
 3|  4| 13|   subtract (#13) from @
 4|  7|  1|   jump unconditionally to #1
10|  0|   |   halt
12|   |  9|   data value 9 stored at #12
13|   |  1|   data value 1 for decrement
24|  2| 36|   make a copy of @ at #36
25|  5| 36|   multiply @ by its copy
26| 12|   |   output result of multiply
27|  1| 36|   restore @ to old value
28|  7|  3|   jump back into main loop
```

As an example of what happens if the accumulator is not restored to its initial value by the subroutine, change the contents of location 27 to:

```
12|  7| 28|
```

which simply transfers control to the next instruction (i.e. it effectively does nothing). Now single step through the program and watch what happens.

8.2 Worked example 5

Output the squares of the numbers 9 through 1 using a subroutine that can jump back to an address stored in a specific location.

This shows a fairly advanced use of jumping using indirect addressing to achieve a primitive version of a conventional subroutine's return statement. It can safely be ignored by the more timid.

The 'subroutine' approach whereby we jump to an address to achieve some useful affect and then jump back is often very useful, as we can jump to it from many different places in our program. But if we want to jump back from different places then how is the subroutine to know where to jump back to? We arrange for this by storing the return address at a location that the subroutine will then use to do its jump. This requires that the address used for the jump is not given directly as the operand to the JUMP instruction. We use the JUMP INDIRECT instruction which uses its operand to store the address at which the address for the jump is stored.

So before we jump to our subroutine we put the return address somewhere where the subroutine can find it, say address 47. We need to store the return address at location 14 so that we can load it into the accumulator with the LOAD instruction. One further complexity is that we need to remember the old value of the accumulator before storing the return address, by storing it to an unused location and then restoring it with load.

So the program becomes:

```
 0|  1| 12|  load @ with (#12)
 1|  9| 10|  jump if @ zero to #10 (HALT)
 2|  2| 15|  store @ at #15 temporarily
 3|  1| 14|  load @ with return address
 4|  2| 47|  store return address at #47
 5|  1| 15|  restore @ from #15
 6|  7| 24|  jump to #24 to calc square
 7|  4| 13|  subtract (#13) from @
 8|  7|  1|  jump unconditionally to #1
10|  0|   |  Halt
12|   |  9|  data value 9 stored at #12
13|   |  1|  data value 1 for decrement
14|   |  7|  call has return address = 7
24|  2| 36|  make a copy of @ at #36
25|  5| 36|  multiply @ by its copy
26| 12|   |  output result of multiply
27|  1| 36|  restore @ to old value
28| 10| 47|  jump indirect into main loop
```

Notice that the instruction at location 28 is the indirect jump which transfers control to the instruction at the address stored at address 47.

Exercises D

(1) Write a program to output the first five natural numbers and their squares.

(2) Write a program that counts the number of 1's given in an input sequence of positive numbers terminated by zero.

(3) Write a program to input two numbers and calculate the remainder when one is divided by the other.

(4) Write a program to output the positive odd numbers in descending order which are less than any input number.

(5) Write a program that calculates and outputs all the factors for any positive inputted number.

9 Number systems

9.1 Introduction

We all have an intuitive idea of how the normal decimal number system works, mainly because we have all used it for as long as we can remember. The decimal number system is based on counting things in groups of ten, and almost certainly came about because we have ten fingers. Unfortunately, the number ten has no such significance for a computer, whereas the number two does.

Before we proceed to look at base 2 (or binary) arithmetic we'll take a closer look at the decimal system. From an understanding of the decimal system we can easily make the adjustment to doing arithmetic in any base.

9.2 Base 10

A base 10 (or decimal) number is a series of digits taken from the ordered set:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Note that this is a set of ten distinct symbols. i.e. it allows us to label up to ten objects with a distinct label from the set.

The position of a digit determines the power of ten that it represents.

For example:

$529 = 5*10^2 + 2 *10^1 + 9*10^0$

$= 500 + 20 + 9$

Note that any number to the power zero is one.

This has described how we represent numbers in base 10. We normally just take this for granted, but it is very useful to understand the mechanics of it because the same methods apply to any number system.

9.3 Base 2 (Binary)

A number is a series of digits taken from the ordered set:

{0, 1}

Notice that we have two distinguishable symbols.

The position of each digit determines the power of 2 it represents.

For example:

10110 = $1*2^4 + 0*2^3 + 1*2^{\wedge 2} + 1*21 + 0*2^0$

= 16 + 0 + 4 + 2 + 0   (in decimal)

= 22 (in decimal)

9.4 Base 16 (Hexadecimal)

A number is a series of digits taken from the ordered set:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Notice that we have sixteen distinguishable symbols. We use the same digits as with decimal but extend them with the first six characters from the alphabet .

The position of each digit determines the power of 16 it represents (this should be beginning to sound rather familiar).

For example:

F9A = $F*16^2 + 9*16^1 + A*16^0$

= 15*256 + 9 *16 + 10 (in decimal)

= 3994 (in decimal)

9.5 Relationship between binary and hexadecimal

Computers are built from two-state devices, i.e. transistors which can be on or off. This makes binary the natural choice. Unfortunately binary numbers are rather unpleasant to work with. When written they tend to be long, hard to compare and recognise. This is where the usefulness of hexadecimal comes in, hexadecimal turns out to be a very useful and compact way of writing binary numbers.

Here is a conversion table that shows the binary representation of each hexadecimal digit:

Binary	Hexadecimal
0	0
1	1
10	2
11	3

100	4
101	5
110	6
111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

From this table we can see that four bits are equivalent to one hexadecimal digit. Put another way, four bits are capable of representing exactly the same numbers as a single hexadecimal digit.

We often think of a single hexadecimal digit as four binary digits and we often add the necessary leading zeroes to make up the four bits.

But what is amazingly convenient is that we can split a binary number of any length into four bit parts and convert each part independently (using the above table if you wish).

For example:

Take the binary number:

1111101010000010

What is the equivalent hexadecimal? This looks pretty nasty, but given our trick the problem is incredibly simple. Firstly split the number into four bit groups working from the right:

1111   1010   1000   0010

Now convert each group to get:

FA82

That's the answer!

If the binary number does not split into an exact number of groups of four bits then just left pad it with enough zeroes so it does.

Why does this method work? It's because sixteen, on which hexadecimal is based, is a power of two.

For example, start with the hexadecimal number and express it as powers of two:

FA82 = $F*16^3 + A*16^2 + 8*16^1 + 2*16^0$

This is by definition of what a hexadecimal number means. Now using the fact that 16 = 2^3 we can rewrite this as:

FA82 = $F * 2^{12} + A*2^8 + 8*2^4 +2*2^0$

This is because:

16^3 = 2^{12}

16^2 = 2^8

16^0 = 2^0

Continuing with our calculation:

FA82 = $1111 * 2^{12} + 1010*2^8 + 1000 * 2^4 + 0010 * 2^0$

This last stage has just used our conversion table given above. Our final stage gives us:

FA82 = $1*2^{15} + 1*2^{14} + 1*2^{13} + 1*2^{12} + 1*2^{11} + 0*2^{10}$...

Which yields the binary number we started with.

9.6 Bytes and bits

One byte is a collection of eight bits.

The range of numbers is binary is:

00000000 to 11111111

and in hexadecimal is:

00 to FF

Each addressable memory location of TOM consists of two bytes. The left byte, which is the more significant byte, and the right byte, which is the least significant byte.

This means that a single byte can store any number between 0 and FF, and each memory location can store any number between 0 and FFFF.

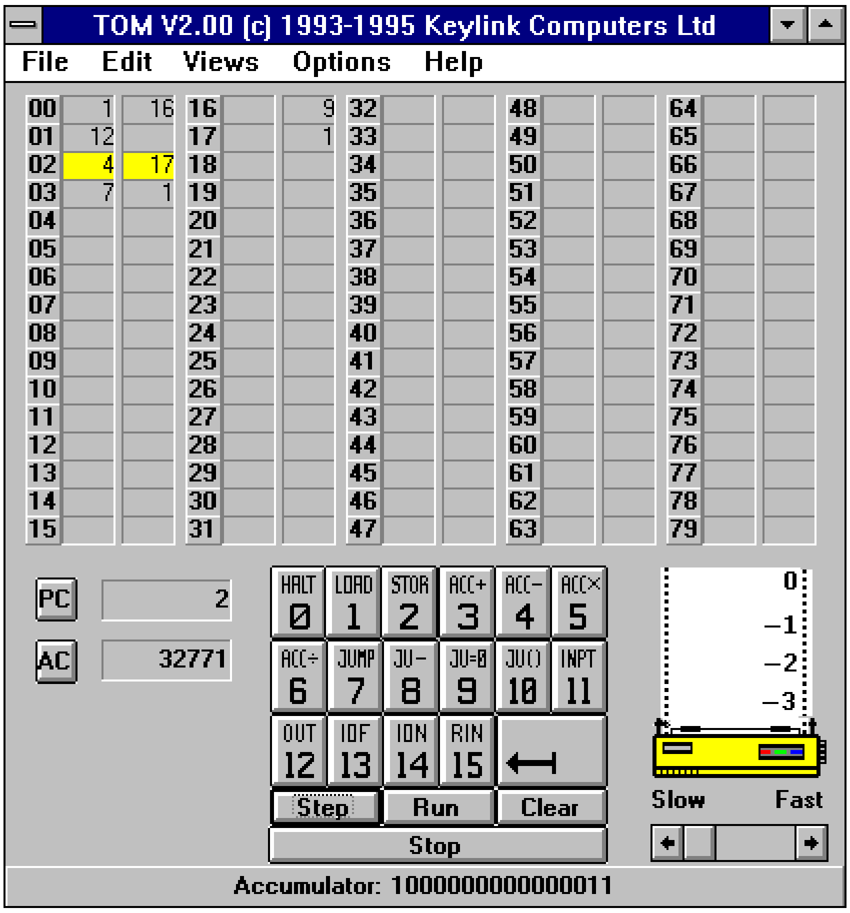
9.7 Representing negative numbers

Each of TOM's memory locations is 16 bits or two bytes long. This allows for 256*256 = 65536 different numbers. So it might seem obvious that we can represent the numbers 0 through 65536, and so we could if we are happy to ignore negative numbers. But if we want to be able to calculate with negative numbers then we must have some scheme for representing them.

TOM can store negative numbers and it uses the most significant bit to represent whether the number is positive or negative. If the bit is not set then the number is positive and if the bit is set then the number is negative. That leaves 15 bits for the actual number. This halves the range of numbers we can represent to 0 through 32767.

This is the simplest way of representing a negative number, make one of the bits a sign bit, and make all the other bits the absolute value. This is not actually the most common method but it is by far the most direct which is why we use it in TOM. It is easy to spot that a number is negative by looking at its binary form and noting whether the left most bit is set.

TOM recognises a negative number by virtue of the leftmost bit value and processes it accordingly. The output area is also aware of negative numbers and will represent the output with the normal minus sign followed by the absolute value. Consider the program on this screen shot:



00	1	16	16		9	32			48			64		
01	12		17		1	33			49			65		
02	4	17	18			34			50			66		
03	7	1	19			35			51			67		
04			20			36			52			68		
05			21			37			53			69		
06			22			38			54			70		
07			23			39			55			71		
08			24			40			56			72		
09			25			41			57			73		
10			26			42			58			74		
11			27			43			59			75		
12			28			44			60			76		
13			29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 2

AC 32771

```
 0
-1
-2
-3
```

Accumulator: 1000000000000011

This shows a simple loop starting at 9 and constantly subtracting 1. The loop carries on executing when it gets to 0, and negative numbers are displayed on the printer. Note the value displayed in the accumulator, this value has just been output as -3, which is precisely what it represents.

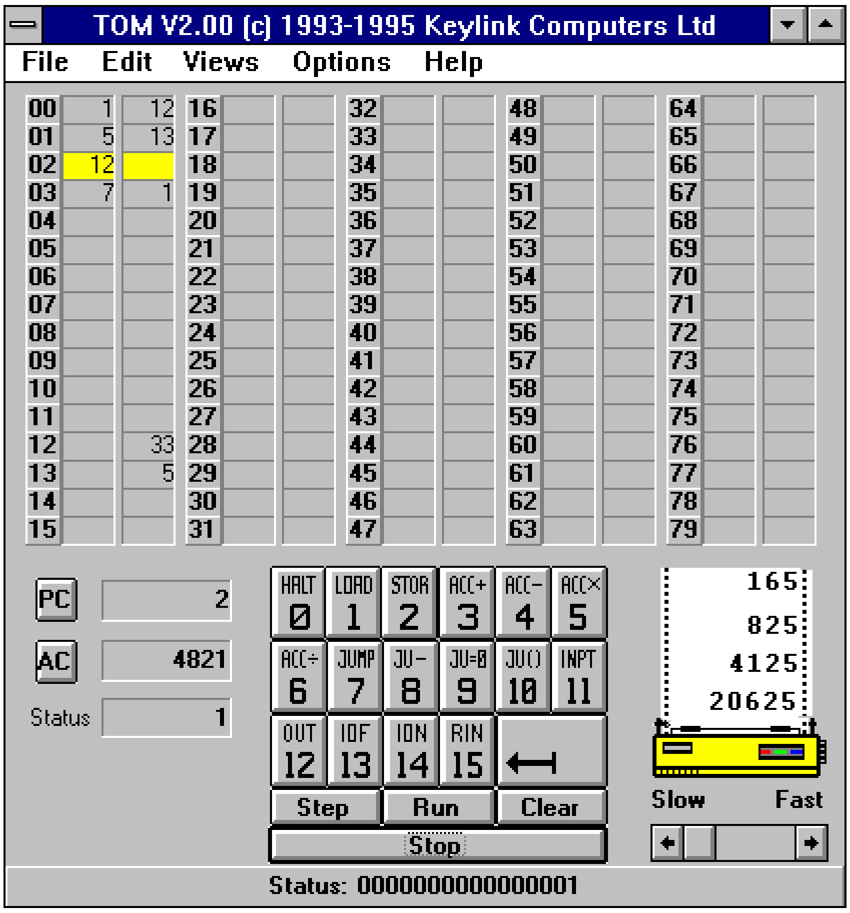
In the Status bar at the bottom of the screen, the value of the accumulator is displayed in binary (you can achieve this by moving the mouse pointer over the accumulator register). It is clear that the leftmost, or most significant bit of this number is set. The value of this binary number without the leftmost bit is ‘11’ which is binary for decimal 3. If you were to calculate the decimal equivalent of the entire binary number, that is including the leftmost bit, you would get the number 32771, which is what is displayed in the accumulator.

The following table gives TOM’s number representation, showing the binary and decimal values:

Decimal	Binary	TOM value (decimal)
65536	1111111111111111	-32768
.	.	.
.	.	.
32769	1000000000000001	-1
32768	1000000000000000	0
32767	0111111111111111	32767
.	.	.
.	.	.
1	0000000000000001	1
0	0000000000000000	0

9.8 Status register and arithmetical overflow

TOM has a Status register which can be displayed by clicking on **Status On** in the **Options** menu. The Status register appears below the accumulator:



```
00   1  12
01   5  13
02  12
03   7   1
12      33
13       5
```

```
PC       2
AC    4821
Status   1
```

```
  165
  825
 4125
20625
```

```
Status: 0000000000000001
```

The above program keeps multiplying the accumulator by 5, until the inevitable happens: the result overflows what TOM can represent. If you have the Status register visible, then the result will simply be that the lowest bit of the Status register is set. The binary form can be seen by placing the mouse over the Status register. If you have not got the Status register visible, then TOM will produce a warning and stop the program from continuing.

9.9 Binary Coded Decimal (BCD)

Four bits can represent the numbers 0 to 15. So an exceptionally easy way to represent a decimal number is to allow each group of four bits to represent a single decimal digit, and we ignore the fact that each four bits could actually represent rather more. For example we can represent the number 5071 as:

5071 = 0101 0000 0111 0001

We can add two numbers represented in this fashion in a very straightforward manner. Just sum the individual groups and use carrying over exactly as for decimal.

This gives us an exceptionally easy method of conversion between a decimal number and its BCD form. But a major disadvantage is that it needs rather more storage for a given range of decimal. The above 16 bits can only represent a number up to 9999 in decimal, whereas TOM with 16 bits represents a much greater range than this both positive and negative.

Another problem is that the logic required to perform decimal arithmetic on BCD is rather more complex than for binary. This point will become rather more clear when you reach the chapter on Logic gates.

Exercises E

(1) What is the hexadecimal number: FA3 in decimal?

(2) What is the binary number: 1001001001 in decimal?

(3) Convert the following numbers from binary to hexadecimal

(a) 0101100011110110

(b) 111110000

(c) 10101010101

(4) Convert the following hexadecimal numbers to binary:

(a) FF

(b) AA

(c) 10F0

(d) 1111

(5) In TOM's representation which of this binary numbers is negative:

(a)1000100111000010

(b) 0000010100100111

(c) 1111110000000111

(d) 1000000000000000

(e) 0000000000000000

(6) What numbers do (d) and (e) represent?

10 Subroutines

10.1 Introduction

We have already met the idea of a subroutine in Worked Example 5, which I repeat here:

Output the squares of the numbers 9 through 1 using a subroutine.

In the original solution I had to arrange for the subroutine to jump back to where it had been called from by explicitly storing the return address at some fixed location. Although this is perfectly feasible TOM, in common with most processors, has specific instructions for handling subroutine calls which have the advantage of being rather more flexible. The instructions are:

JSB (19)	Jump to subroutine
RTN (20)	Return from subroutine

There is also a dedicated register for the use of subroutine calls: the Stack Pointer.

The stack is an area of memory starting at address 64 which is used for storing return addresses of subroutine calls. The stack pointer is a register that holds the address of the next free entry in the stack. The reason we need a stack is that:

- our program may call a subroutine from address 12 say
- this subroutine may call another subroutine from address 45 say.

Now, what we want to happen is:

- when the second subroutine finishes we return from the second subroutine into the first at address 46;
- when the first subroutine completes, we want to return to the address 13.

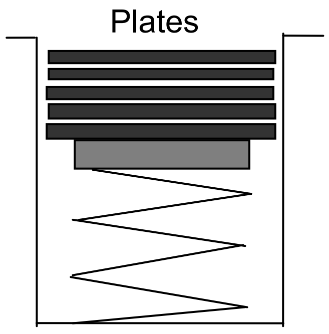
If we have only stored one address for returning to, we clearly cannot do the above which requires two addresses. Hence we use a stack.

A stack is generally taken to be something we take values from in reverse order to how they have been stored. Sometimes such a stack is referred to as a last in, first out stack (LIFO), to emphasise this point.

A stack is much the same as the contraption you see at self-service canteens for delivering plates:

Canteen plate

delivery system



The last plate in is the first plate out.

The subroutine situation is one that most definitely requires a stack. When we make a subroutine call, we store the address to return to, and when that subroutine returns, it must return to the last value stacked.

10.2 Using the Subroutine instructions

This is precisely what the subroutine instructions do when they are executed:

> **JSB (19) JUMP SUBROUTINE**
>
> Puts the program counter + 1 (i.e. the address to return to) into the the memory location given by the stack pointer, Increments the stack pointer and jumps to the address given as the operand.
>
> **RTN (20) RETURN FROM SUBROUTINE**
>
> Decrements the stack pointer and puts the contents of the memory location given by the stack pointer into the Program Counter.

We are now going to see how the above program can be written using the **JSB** and **RTN** instructions.

As with the original solution, the main part of the program executes a loop. It first loads the initial value from address 12 then the loop starts.

```
 0|LDA| 12|   load @ with (#12)
 1|JM0| 10|   jump if @ zero to #10
 2|STO| 15|   store @ at 15
 3|JSB| 32|   Jump sub
 4|LDA| 15|   restore @
 5|AC-| 13|   decrement loop counter
 6|JMP|  1|   jump to start of loop
10|HLT|   |   halt
```

The next few locations are the data values needed by the program:

```
12|   |  9|  loop index
13|   |  1|  loop decrement value
15|   |  1|  temp storage for subroutine argument
```

This is the subroutine:

```
32|ACX| 15|  multiply @ by argument
33|OUT|   |  output the result
34|RTN|   |  return to main loop
```

Memory location 64 has the return address in, but note, you do not have to enter this, it is automatically stored there by the JSB instruction.

```
64|HLT|  4|
```

This is quite obviously a bit easier than when we solved it before. The simplicity carries over when we have more complex situations involving one subroutine calling another. There is no need to keep track of return addresses within our programs, just use the JSB and RTN instructions for all your subroutine calls and returns and the addressing will be looked after for you.

10.3 Recursion

Recursion is when a subroutine calls itself. This may seem quite a bizarre notion but it is in fact rather useful.

Consider the problem of calculating the factorial of a number which is usually represented by an exclamation mark, for example, factorial 4 is denoted by 4!. You calculate the factorial of a number by multiplying the number with 1 less, and 2 less, and so on down to 1. So 4! = 4 * 3 * 2 * 1 = 24.

A precise why of describing the factorial of any number, n, is:

n! = n * (n - 1)!

0! = 1

These two statements are entirely adequate for calculating the factorial of any number. Take the example of 4!.

Using the first line of the definition we can rewrite 4! as follows:

4! = 4 * 3!

4! = 4 * 3 * 2!

4! = 4 * 3 * 2 * 1!

4! = 4 * 3 * 2 * 1 * 0!

Now using the second line we can eliminate the 0! to get:

4! = 4 * 3 * 2 * 1 * 1

4! = 24

Which is the same as our previous result.

Our recursive subroutine will work by taking the number to calculate the factorial of, subtracting one, doing the multiplication and then calling itself to calculate the factorial of the number which is one less.

The first part of the code loads the accumulator with the value that we want to find the factorial of, we store this value at location 12.

```
 0|LDA| 12|
 1|JSB| 16|
 2|OUT|   |
 3|HLT|   |
12|   |  3| the number we want the factorial of
```

After loading the accumulator it jumps to the first subroutine, outputs the result, which the subroutine will store in the accumulator, and then halts.

The first subroutine is used to organise the storage structure for the actual routine that does the calculation. Our storage scheme uses location 48 for the result of the multiplication so far, location 49 stores the decrement value which is 1, and location 50 will be the number to multiply by, which starts at the number we are finding the factorial of and is decremented to zero.

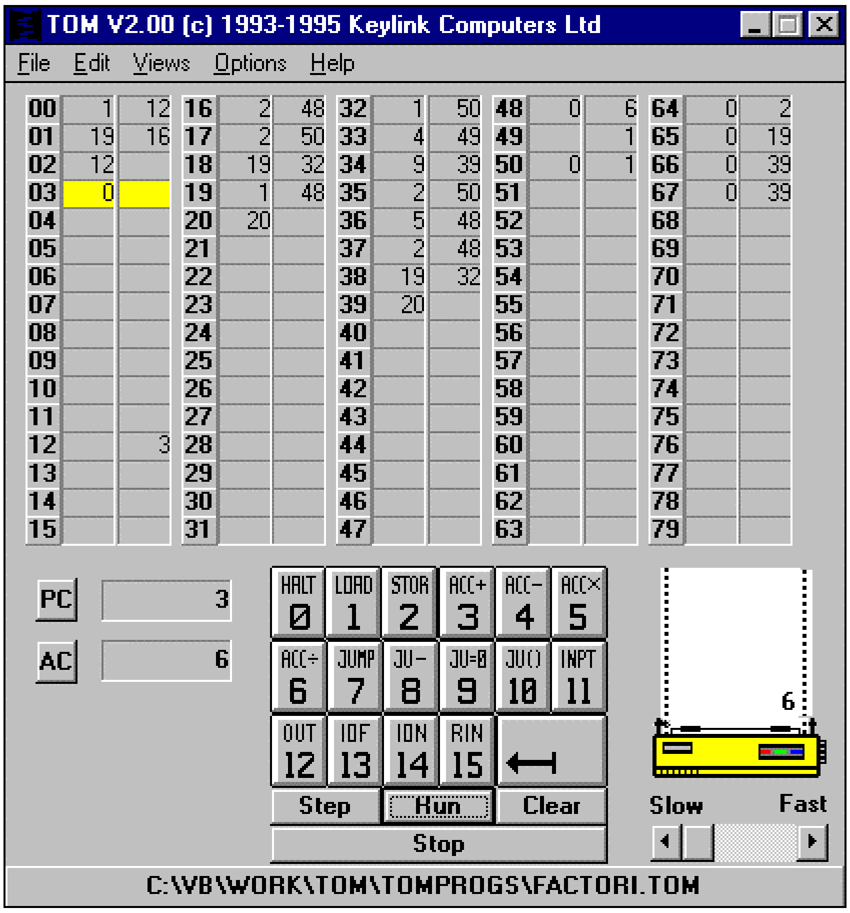
```
16|STO| 48| store @ at 48 - result of multiplications
17|STO| 50| store @ at 50 - the multiplication factor
18|JSB| 32| jump to the factorial subroutine
19|LDA| 48| result is in 48, put this in @
20|RTN|   | return to the main routine
```

Now for the factorial subroutine itself.

```
32|LDA| 50| load the current factor into @
33|AC-| 49| subtract 1
34|JM0| 39| if zero jump to the end of the subroutine
35|STO| 50| store the new factor at 50
36|ACX| 48| multiply the new factor by result so far
37|STO| 48| store the result back in result so far
38|JSB| 32| make the recursive call
39|RTN|   | return
49|   |  1| the decrement value
```

When you watch this execute you will see the subroutine stack grow from location 64. When the factor reaches zero you will see a series of return statements executed which shows the recursion unwinding.

This is the state of TOM when the execution has finished:



00	1	12	16	2	48	32	1	50	48	0	6	64	0	2
01	19	16	17	2	50	33	4	49	49		1	65	0	19
02	12		18	19	32	34	9	39	50	0	1	66	0	39
03	0		19	1	48	35	2	50	51			67	0	39
04			20	20		36	5	48	52			68		
05			21			37	2	48	53			69		
06			22			38	19	32	54			70		
07			23			39	20		55			71		
08			24			40			56			72		
09			25			41			57			73		
10			26			42			58			74		
11			27			43			59			75		
12		3	28			44			60			76		
13			29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC 3

AC 6

6

C:\VB\WORK\TOM\TOMPROGS\FACTORI.TOM

Note the values in the subroutine stack, the first is the return from the subroutine that starts at address 16, the second is the return from the first call to the factorial subroutine, and the remaining two are returns from the recursive calls.

10.4 Stack overflow

Just how big a stack do we need? This can readily be calculated, for each active subroutine call we quite clearly need to be able to store a return address. So the number of locations the stack needs is exactly the same as the most subroutines that can be called at the same time.

TOM's stack is from address 64 to address 68, hence TOM can support five simultaneous calls to subroutines (as it can remember a total of five addresses to return to). This is not a great deal and you will find real computers will allows very much more.

In fact, it is seldom a hardware limit, but something that is set in software. Using a modern language such as C or Pascal, you may be required to tell the system how big a stack you want, or accept some default value that is provided for you. In any event, whatever language you use, be it BASIC or ADA, there is a limit.

What happens when a program exceeds the number of subroutine calls supported by the size of the stack? This all depends on the program. In the simplest situation, the stack starts to grow into an area of memory that is used for something else, and corrupts whatever the something else is.

In TOM the memory from 69 can be used for an output device, this is explored in the next chapter. If your stack grows to address 69 and beyond, and you are using the output device, then its output is going to be corrupted by the stack values. An example of this happening is given in the next chapter.

It is possible for software to trap the fact that the stack is about to exceed the limit set for it. This is done by something called a stack probe, which examines the current setting of the stack pointer before each subroutine call and raises an error if the stack is about to trash (illegally overwrite) memory not allocated to it. Clearly having stack probes means that the executable task will take longer to run. Hence, it is not unusual, to enable stack probes during program development and then to disable them once the program has been fully developed and tested.

Exercises F

(1) Write a subroutine that calculates the factorial of a number stored in the accumulator and outputs it. A factorial is calculated by multiplying the given number by each number between it and 1. For example: 5! = 5*4*3*2*1 = 120.

(2) Call the above subroutine from inside a loop which runs from 1 to 9.

(3) Write a subroutine that finds the result of X mod 7, that is given some number, X placed in the accumulator, it places the remainder when as many sevens as possible have been subtracted. The result is to be placed in the accumulator.

11 Memory mapped output

11.1 Introduction

Memory mapped output is a technique where a given section of memory is dedicated to an output device, such as a screen. Each location on memory will directly represent some part of the screen image. The kind of screen you commonly use on your PC will be memory mapped from some considerable chunk of memory.

Changes made to the memory in this area immediately change what is displayed on the screen. Electronic circuitry constantly scans this area of memory and refreshes the screen.

A Super VGA image on a PC uses 800 by 600 pixels and can represent 16 colours. A pixel is the smallest element of the screen image, each pixel can be at one of 16 colours. That needs 4 bits of memory for each pixel (2^4 = 16). So the total memory requirement for the screen image is:

800*600*4 = 1,280,000 bits

Or 160,000 bytes

TOM only has 80 memory locations, so its memory mapping capability is very limited. In fact we use only 10 memory locations and rather than map these to pixels (you couldn't make much of an image with just 10 pixels!) TOM maps them to ASCII characters.

11.2 ASCII Characters

ASCII stands for American Standard Code for Information Interchange and is a simple collating sequence that maps numbers to characters. These are some of the mappings:

Number (Decimal)	Character
32	Space
.	.
48	0

49	1
.	.
57	9
.	.
65	A
66	B
.	.
90	Z
.	.
97	a
98	b
.	.
122	z
.	.

The numeric, uppercase and lowercase letters are all in their natural sequence, so you should be able to calculate what the ASCII code for any letter or any number is. If you find this difficult then there is a complete ASCII table in an appendix.

There are two sorts of ASCII character:

- Printable
- Non-printable

Printable characters are like those given in the above table. Non-printable characters are sometimes known as control characters, they are generally used to control an action of an output device such as a screen or a printer.

Some examples of non-printable characters are:

ASCII	Meaning
10	Line Feed
13	Carriage Return

These are clearly intended for controlling a printer and the more primitive types of terminal. Such a device would interpret a Line Feed followed by a Carriage Return as the codes necessary to start a new line.

TOM is only capable of interpreting the printable type of character. Trying to print all other types of character results in a space being printed.

11.3 Using Memory Mapped Output

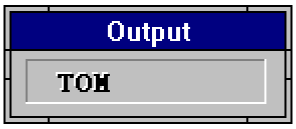
The memory which is mapped to the output window is from address 69 to address 78.

Select **Output Device** from the **Options** menu and the memory mapped output device will be displayed. The device will display characters mapped from the addresses 69 to 78 at all times, i.e. not just when a TOM program is running.

For example put the following values into memory at the indicated locations:

```
69|    |84|
70|    |79|
71|    |77|
```

The output device should now look like:



Far more commonly we would use a program to display values on our output device.

11.4 Indirect addressing

If we try writing a program to do almost anything with memory mapped output, such as copying characters from one part of memory to the memory mapped area, we find that we have a problem. Consider writing a program to transfer the memory locations from 24 to 26 to 69 to 71:

```
00|LDA|24|
01|STO|69|
02|LDA|25|
03|STO|38|
04|LDA|26|
05|STO|39|

24|    |84|
25|    |79|
26|    |77|
```

This copies “TOM” to the memory mapped output area. As you can see each character is copied with a LDA,STO pair, which soon becomes rather long winded when copying strings greater in length than this. What we need is to be able to indirectly address memory, loading and storing values from memory locations stored in other memory locations.

This method of indirectly addressing memory is of enormous benefit when coding anything that deals with a range of memory cells. We can arrange to store the start and end addresses actually in memory and use these to store our values indirectly.

The instruction for loading the accumulator from a memory location given in another memory location is 17 with mnemonic L(). Its argument specifies the memory address of a location where the actual value is stored. For example:

```
00|L()| 12|

12|   | 25|

25|   | 79|
```

The Load Indirect instruction at memory location 0, loads the value 79 into the accumulator. The big advantage of this instruction is that if we now want this occurrence of Load Indirect to load from a different location we only have to change the value stored in location 12.

Similarly, the instruction for storing the accumulator to a memory location given in another memory location is 18 with mnemonic S(). Its argument specifies the memory address of a location where the actual value is to be stored. For example:

```
00|S()| 12|

12|   | 25|

25|   |   |
```

The Store Indirect instruction at memory location 0, stores the current value of the accumulator to memory location 25. The big advantage of this instruction is that if we now want this occurrence of Store Indirect to store to a different location we only have to change the value stored in location 12.

Consider the following TOM program which outputs the letters ‘ABCDEFGHIJ’ to the output device:

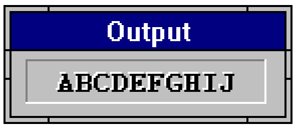
```
 0|LDA| 12|  Load @ with number of letters
 1|JM0| 24|  finish when @ = 0
 2|AC-| 13|  decrement loop counter
 3|STO| 12|  and put it back where it came from
 4|AC+| 14|  add 69 to @, this is the address for
output
 5|STO| 15|  store the address at 15
 6|LDA| 16|  load @ with ASCII offset value (65 = A)
 7|AC+| 12|  add the loop index
 8|S()| 15|  copy (@) to memory mapped area (address at
15)
```

```
 9|JMP|   0|  jump to start of loop

12|   |  10|  number of letters to be output
13|   |   1|  loop decrement value
14|   |  69|
15|HLT|   0|  Stores address in memory mapped area
16|   |  65|

24|HLT|    |
```

Notice how the letters in the output window:

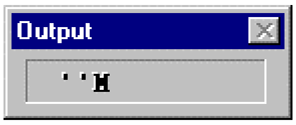

Output
ABCDEFGHIJ

are output in reverse, starting with ‘J’, which is ASCII 65 + 9 going to memory location 78, and finishing with ‘A’, which is ASCII 65 + 0, going to memory location 69.

11.4 Memory corruption

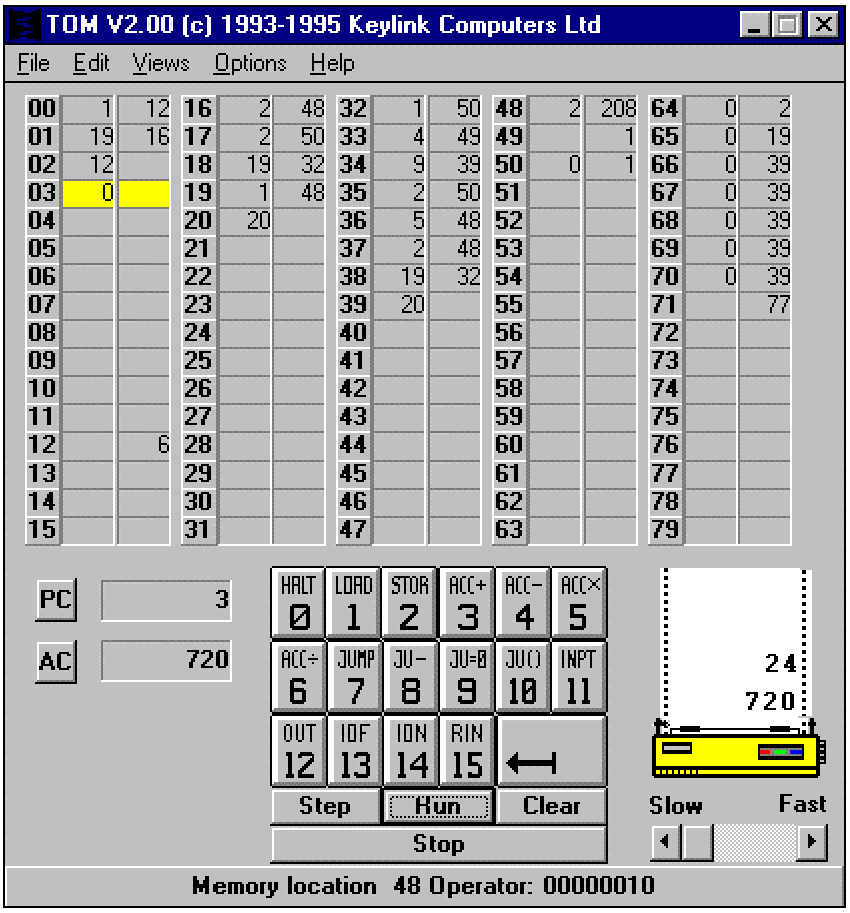
Dedicating an area of memory to an output device means that this area must not be used for anything else. The stack used for subroutines starts at address 64 and grows upwards in memory. If we run the factorial example, given in the previous chapter, and give it a starting value of 6, then there will be a total of 7 subroutine calls active at the deepest point of the recursion. This will cause overwriting of some of the memory mapped area.

The following full screen shot shows the end result of running the factorial program to calculate the factorial of 6, whilst the memory mapped output starts off by displaying TOM, it ends up displaying:


Output
''M

The two quotes are the result of the stack growing into the memory mapped output area at addresses 69 and 70.

This corruption of memory can be easily seen on the full screen shot:



00	1	12	16	2	48	32	1	50	48	2	208	64	0	2
01	19	16	17	2	50	33	4	49	49		1	65	0	19
02	12		18	19	32	34	9	39	50	0	1	66	0	39
03	0		19	1	48	35	2	50	51			67	0	39
04			20	20		36	5	48	52			68	0	39
05			21			37	2	48	53			69	0	39
06			22			38	19	32	54			70	0	39
07			23			39	20		55			71		77
08			24			40			56			72		
09			25			41			57			73		
10			26			42			58			74		
11			27			43			59			75		
12		6	28			44			60			76		
13			29			45			61			77		
14			30			46			62			78		
15			31			47			63			79		

PC	3
AC	720

```
24
720
```

Memory location 48 Operator: 00000010

The number 39 is the return address from the subroutine which starts at address 32. If you look in the ASCII table you'll see that 39 is also the number for the single quote character.

Exercises G

(1) What is the ASCII collating sequence number for the letter ‘w’?

(2) Find out what resolution your PC screen is using and how many colours it can display. Calculate what the memory requirement for your screen is.

(3) Display the Memory Mapped output device and by putting values directly into memory make the output display your name.

(4) Write a subroutine that takes the value of the accumulator as the start address and copies ten locations starting at that address to the memory mapped output area. Use indirect addressing.

12 Interrupts

12.1 Introduction

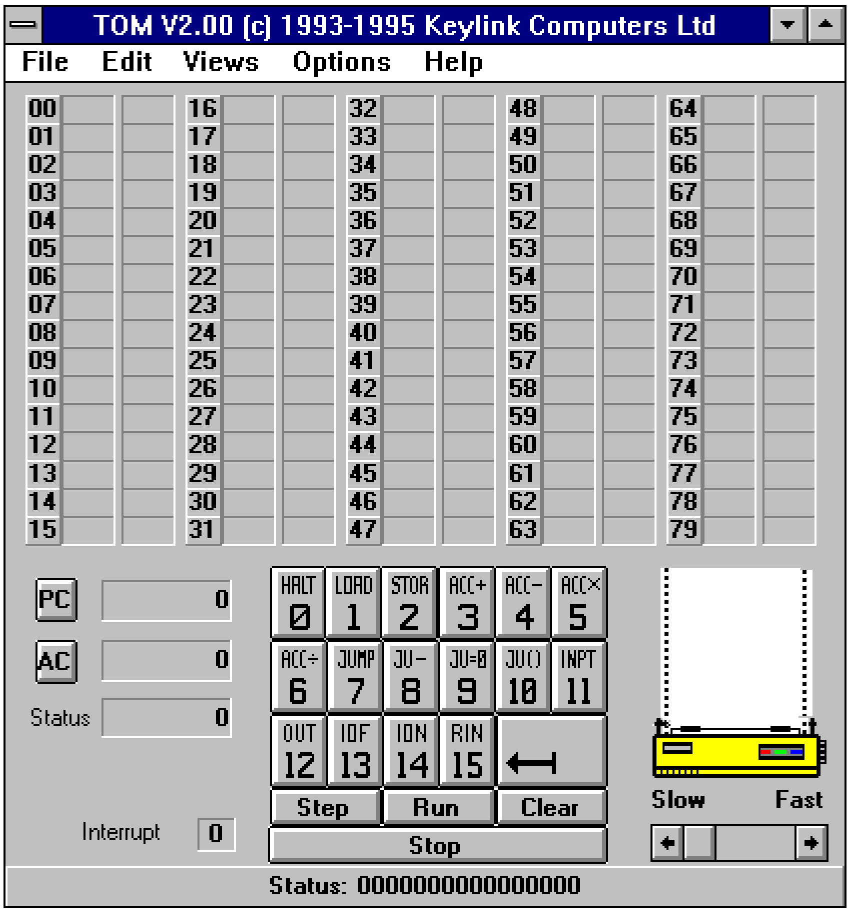
When an input device, such as a keyboard, makes an input we want the machine to stop what it is doing and process the input character. In most situations we do not want the computer to be constantly waiting for a keyboard character, which would waste a great deal of computer time, but only process the input as and when it is available.

This is achieved by the use of an **interrupt**. Within the processor's execution logic a check is made before each instruction is executed as to whether any device has posted an interrupt. If an interrupt has been set, it is then necessary for this interrupt to be **serviced,** i.e. processed before the program that was executing can be returned to.

The routine that services an interrupt is much the same as any other piece of code that resides in memory and can be jumped to, however, there are a few extra instructions available for processing interrupts and an extra register, the **interrupt register.** In addition there is an **interrupt mask bit** in the Status register, this is the second most significant bit.

12.2 Using the Interrupt option

With TOM running choose **Interrupts** from the **Options** menu and the screen will look like:

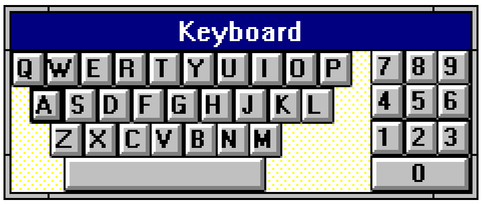


Notice the extra one bit register at the bottom left of the screen. The Status register is automatically displayed when Interrupts On is selected. The Interrupt Mask bit must be set to 1 for an interrupt to be noticed by TOM. Its normal state is set to zero, i.e. all interrupts are normally ignored. The Interrupt Mask bit is normally set by the executing program, hence there are instructions for setting and unsetting it.

The Interrupt register is set by the device wanting to cause the interrupt.

When **both** the Interrupt register **and** the Interrupt Mask bit are set an interrupt occurs.

Also displayed is a second form which represents a typical input device, a QWERTY keyboard with numeric pad:



Whenever a key is pressed on the Keyboard the Interrupt register is set. This may or may not cause an interrupt depending on the state of the Interrupt Mask bit.

The following table lists the new instructions which are available:

Instruction number	Mnemonic	Description
13	IOF	Set Mask to 0
14	ION	Set Mask to 1
15	RIN	Read Interrupt Input
16	RTI	Return from Interrupt

12.3 Programming for Interrupts

When you want your program to be capable of programming interrupts execute the **ION**, Interrupt on, instruction. Now you must have defined an Interrupt Service Routine (also known as an Interrupt Handler).

When an Interrupt occurs, i.e. when a key on the keyboard form is pressed, control will be passed to the Interrupt Handler you have defined.

TOM expects the starting address of your Interrupt Handler to be located at address 79. That is the memory at location 79 must contain the starting address for the handler.

What TOM does when it encounters an interrupt is:

- Saves the current value of the Program Counter.
- Saves the current value of the Accumulator.
- Transfers control to the address that is stored in location 79.
- Sets the Interrupt Mask bit to 0.

TOM saves the Program Counter and Accumulator so that it can continue processing where it left off, it will do this when it encounters the **RTI**, Return from Interrupt, instruction (this will always be the last instruction of your interrupt handler).

It transfers control to the start of the Interrupt handler in order to service the interrupt.

TOM sets the Interrupt Mask bit to 0 in order to disable any further interrupts until this one has been processed.

The **RTI** instruction does the following:

- Returns the value of the Program Counter to what was saved.
- Returns the value of the Accumulator to what was saved.
- Sets the Interrupt Register to 0.

The Program Counter and Accumulator are restored to enable processing to continue precisely as it would have done had the interrupt never happened.

The Interrupt Register is set to 0 because the interrupt has been processed. Note that the mask remains off, it is up to the program to reset this mask if it so desires to process another interrupt.

The **RIN**, Read Interrupt Input, instruction allows the value being input from the device to be read into the Accumulator. You would expect every interrupt handler to use this instruction.

What actual code goes into an interrupt handler is down to you, but here is a typical example of a program that will loop indefinitely outputting zero to the printer, but every time a key is pressed it outputs the ASCII value of the key to the printer.

The first part of the program is a simple loop that switches on interrupt handling and outputs the contents of memory location 12 to the printer repeatedly:

```
 0|ION|    | Turn interrupt mask on
 1|LDA|  12| Load @ with (12)
 2| 12|   0| Output @
 3|  7|   0| Jump to 0
```

The data value at location 12 is initialised to zero:

```
12|   |   0|
```

This next part is the interrupt handler itself:

```
24|IOF|    | Turn off interrupt handling
25|RIN|    | Read keyboard char into @
26|STO|  12| Store @ to 12
27|RTI|    | Return from handler
```

Finally we have placed the address of the start of the Interrupt Handler at location 79.

```
79|   |  24|
```

Exercises H

(1) Write an interrupt handler that stores input character codes at successive memory locations.

(2) With the memory mapped output device displayed, write a program, with an interrupt handler that writes input characters to the output device. Each character should go to the first position of the output device.

(3) Modify the above program to display input characters, from left to right on the output device.

13 Logic gates

13.1 Introduction

A fundamental component of a computer is the logic gate. Although there are many different types of gate we will only deal with three, the **AND** gate, the **OR** gate and the **NOT** gate. These are in fact sufficient to build any logic circuit, and it is possible to build any of the other types of gate out of just these three.

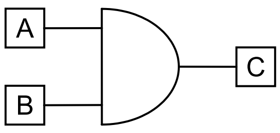
The purpose of a logic gate is to take some logical input and provide a result which depends on the input value or values.

When dealing with gates we only consider two possible values, True and False, but we far more often use the numbers 1 and 0 to represent these values. In the actual circuitry which implements these gates, the value of 1 will often be represented by a high voltage and 0 will be a low voltage.

13.2 The AND gate

The AND gate has two inputs. Its purpose is to give a logical 1 as output when both of its logical inputs is 1. In all other cases its output value is 0.

An AND gate is often represented by a semi-circular shape, like this:



Here, the inputs have been labelled A and B and the output has been labelled C. The AND gate result is 1 when both A **and** B are 1, in all other cases it is 0.

The function of a gate is often described by a truth table which lists all the possible combinations of input and the corresponding output.

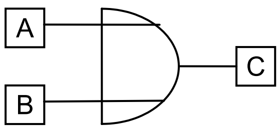
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

For example, to find out what the output value will be when A is 1 and B is 0, read the third row which gives the result that C, the output, is 0.

13.3 The Or gate

The OR gate has two inputs. Its purpose is to give a logical 1 as output when either of its logical inputs is 1. In all other cases its output value is 0.

An OR gate is often represented by a semi-circular shape, like this:



Here, the inputs have been labelled A and B and the output has been labelled C. The OR gate result is 1 when either A **or** B are 1, in all other cases it is 0.

The function of a gate is often described by a truth table which lists all the possible combinations of input and the corresponding output.

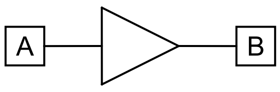
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

For example, to find out what the output value will be when A is 1 and B is 0, read the third row which gives the result that C, the output, is 1.

13.4 The NOT gate

The NOT gate has one input. Its purpose is to negate its logical input. That is, when the input is 1, the output is 0, and when the input is 0, the output is 1.

An Not gate is often represented by a triangular shape, like this:



Here, the input has been labelled A and the output has been labelled B. The NOT gate result is 1 when both A is 0, in all other cases it is 1.

The function of a gate is often described by a truth table which lists all the possible combinations of input and the corresponding output.

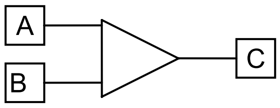
A	B
0	1
1	0

For example, to find out what the output value will be when A is 1 , read the second row which gives the result that B, the output, is 0.

13.5 Putting Logic Gates together

As an example of how logic gates can be connected together we will show how a NAND gate can be constructed from an AND gate and a NOT gate. A NAND gate gives the opposite result of an AND gate. That is the result is 0 whenever either of its inputs is 1, and 1 in all other cases.

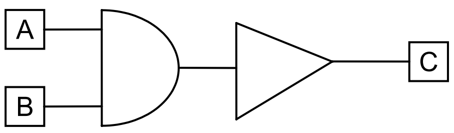
A NAND gate can be represented by a triangle, but it differs from the NOT gate because it has two inputs:



The truth table for a NAND gate is:

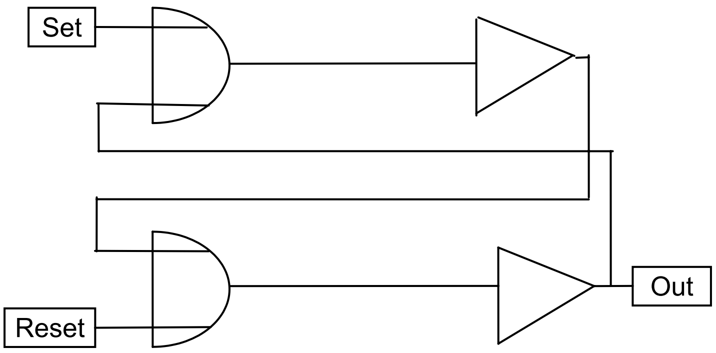
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

The NAND gate is easily constructed by taking the result of an AND gate and negating by using a NOT gate:



13.6 The Flip-Flop

The signals that pass though the circuits that comprise logic gates are usually very fleeting. What we require when implementing memory circuits is a way to hold onto a logical 1 or 0, indefinitely. A flip-flop is a cunning arrangement of gates that allows a state to be set and then tested for at any time after it has been set. This is achieved by arranging a feedback loop within the flip-flop which keeps it in this steady state.

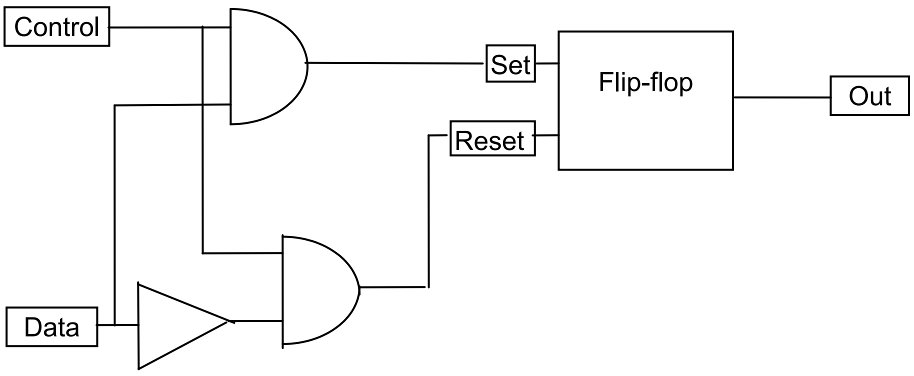


A short duration logical 1 at Set will provide a continuous logical 1 at Out. A short duration logical 1 at Reset will provide a continuous logical 0 at Out.

Consider the Set operation. A logical 1 is applied at Set, this means that the result of the upper OR gate must be 1 regardless of whatever the other input to this gate is. This result of 1 is then notted which must result in logical 0. As Set and Reset signals are only applied for a short duration, we can assume that the Reset is at 0, therefore the result of the second OR gate must be 0, this is then notted to produce the Output value of 1. **But** most importantly, this result is then sent back to the input of the top OR gate where is now acts as another Set signal. At this point the Set signal can be removed and the flip-flop circuit will remember the value of 1 indefinitely. In a similar fashion applying a short duration 1 to the Reset will cause a steady Out of 0 to be achieved.

13.7 The Controlled Flip-Flop

A flip-flop provides us with the central component of memory. But we find it more useful presented in a slightly different way. What we need it to be able to apply a value of 0 or 1 to a memory device and have it remembered indefinitely. But a pulse of 0 is really no pulse at all, so we need to add a control line to tell the memory unit when it should be listening. This is called a controlled flip-flop and is constructed from a normal flip-flop with a few extra gates:



When the Control is at 0, then the state of the flip-flop will not change. When the Control is at 1 then the state of the flip-flop will change according to the data value at Data.

13.8 Half-adder

Clearly computers perform arithmetic, but how they perform it is perhaps rather less clear. The simplest arithmetic operation is the half-adder. This allows just two bits to be added together. As adding a ‘1’ to a ‘1’ overflows a single bit (the result is ‘10’ in binary), it is necessary to not only have a result bit, but also a carry bit.

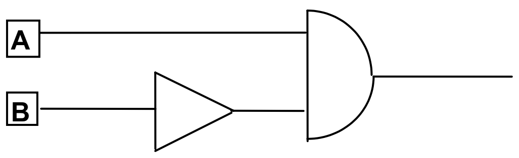
The truth table for such a device is easy enough to specify, we simply fill in the table with the required results. If our two input bits are A and B, and the result is S with carry bit C. Then clearly adding 0 to 0 results in 0 with 0 carry bit, and so on. Here is the table:

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

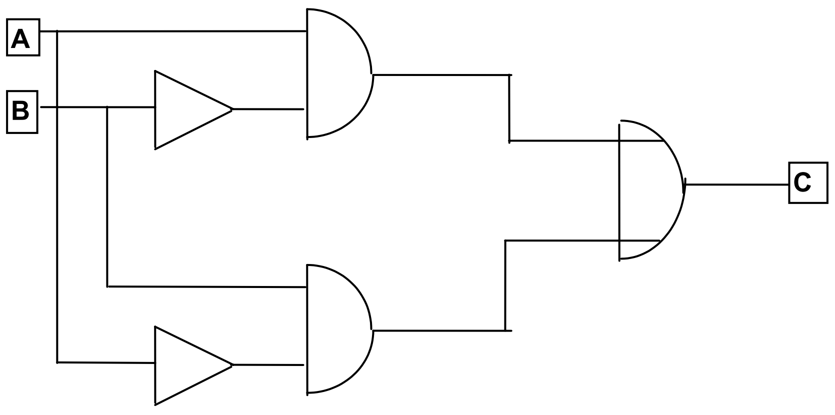
The Carry bit is the result of A **AND** B. This should be clear as we only set the carry bit when both input bits are set.

The Sum bit is either A **AND NOT** B **OR NOT** A **AND** B. This is a little more difficult. We only set the sum bit when either: A is set but B is not set, or when B is set and A is not set. Putting this another way, we only set S when one or other of A or B are set, but not when both are.

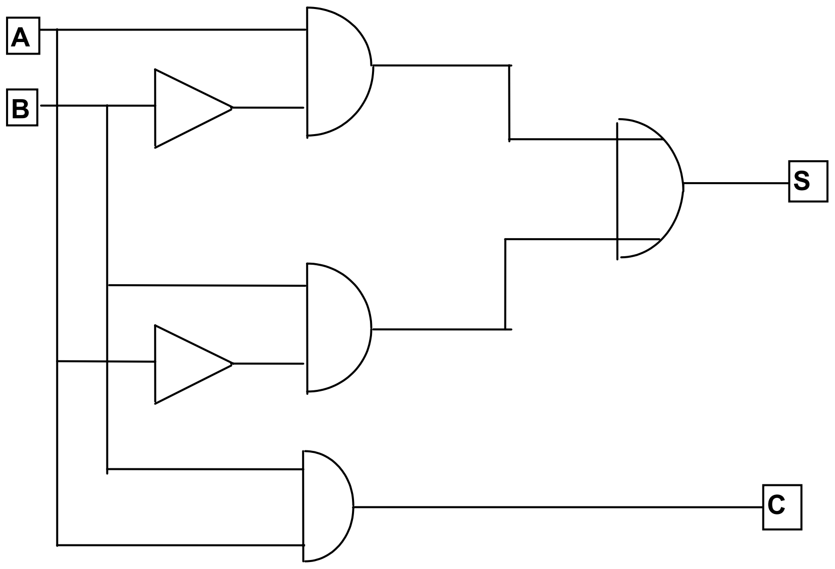
We can build the logic diagram for the half-adder in a number of stages. First the diagram for A **AND NOT** B:



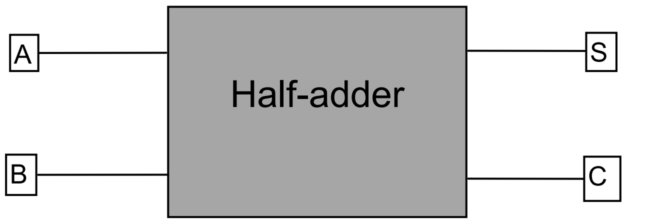
We now need to OR this with NOT A and B:



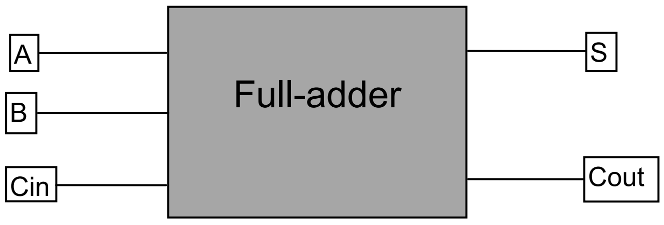
Now it just remains to add the logic for the carry bit, which is just a result of A **AND** B:



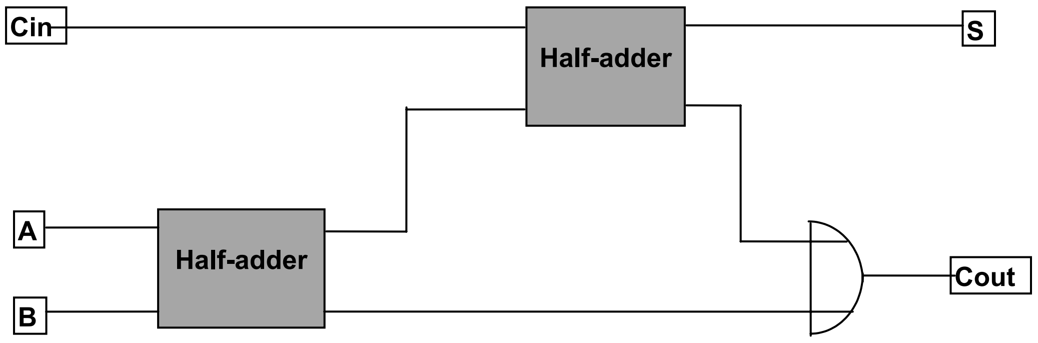
It is convenient to represent a half-adder as a simple box. This helps us when constructing more complex logic, and, of course, it matters not a jot, just as long as we do know how to actually implement one as the above diagram shows us. So for clarity we will from now on represent a half-adder with:



The half-adder is not quite what we need to add together lots of binary digits because we need to be able to cope with carrying in as well as carrying out. What we need is a full-adder. This is very much like a half-adder but has a carry in bit as well as a carry out bit.



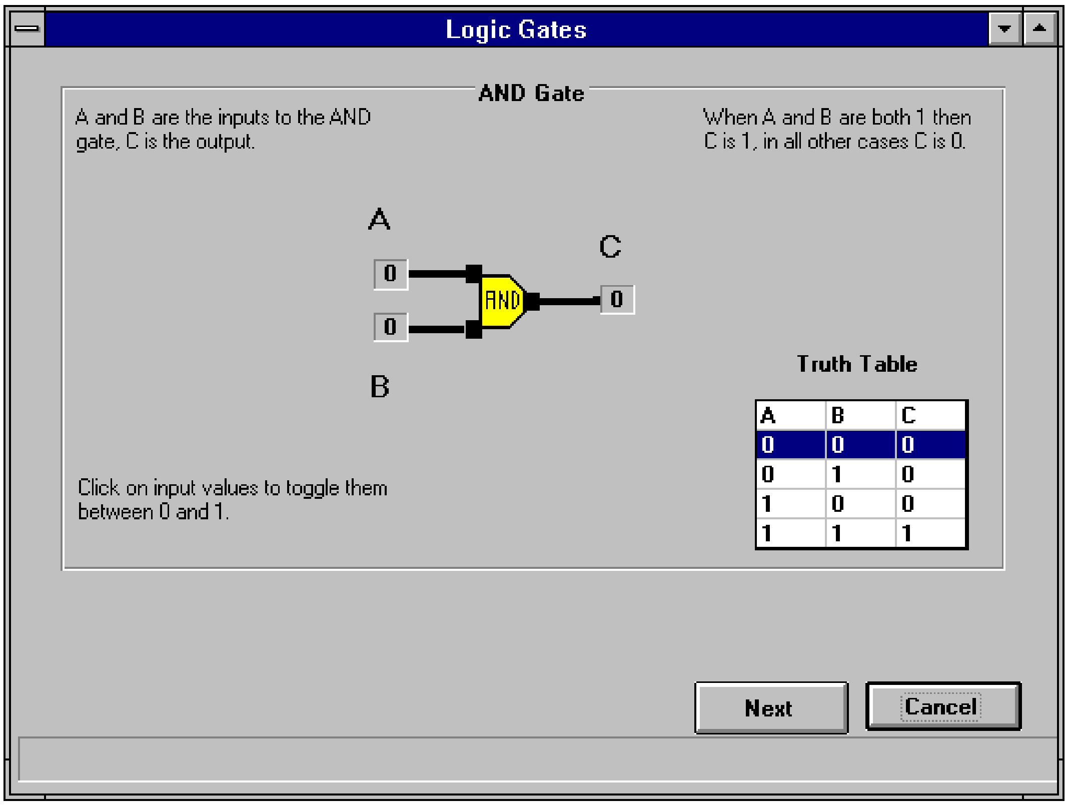
We can make one full-adder out of two half-adders by inputting A and B to the first adder, then inputting the result of this along with the Carry into the second half-adder. The sum is then the result of the second half-adder and the carry out is the result of OR-ing the carry out from both half-adders:



With the full-adder we now have the capability of adding together arbitrary numbers of digits. Clearly TOM would need to use sixteen full-adders to perform arithmetic on the contents of the accumulator and a full memory location. This can be achieved by using a full-adder to add the zero position bits from both numbers, the carry out is then input to the carry in for the next full-adder which adds the bits in position 1, and so until the bits in position 15. The carry out in this case will indicate arithmetic overflow and can be used to set the Status register's lowest bit.

13.9 Logic Gates in TOM

To display the logic gates screen select Logic Gates from the Views menu. The first in the Gate sequence shows the AND gate like this:



A and B are the inputs to the AND gate, C is the output.

When A and B are both 1 then C is 1, in all other cases C is 0.

Click on input values to toggle them between 0 and 1.

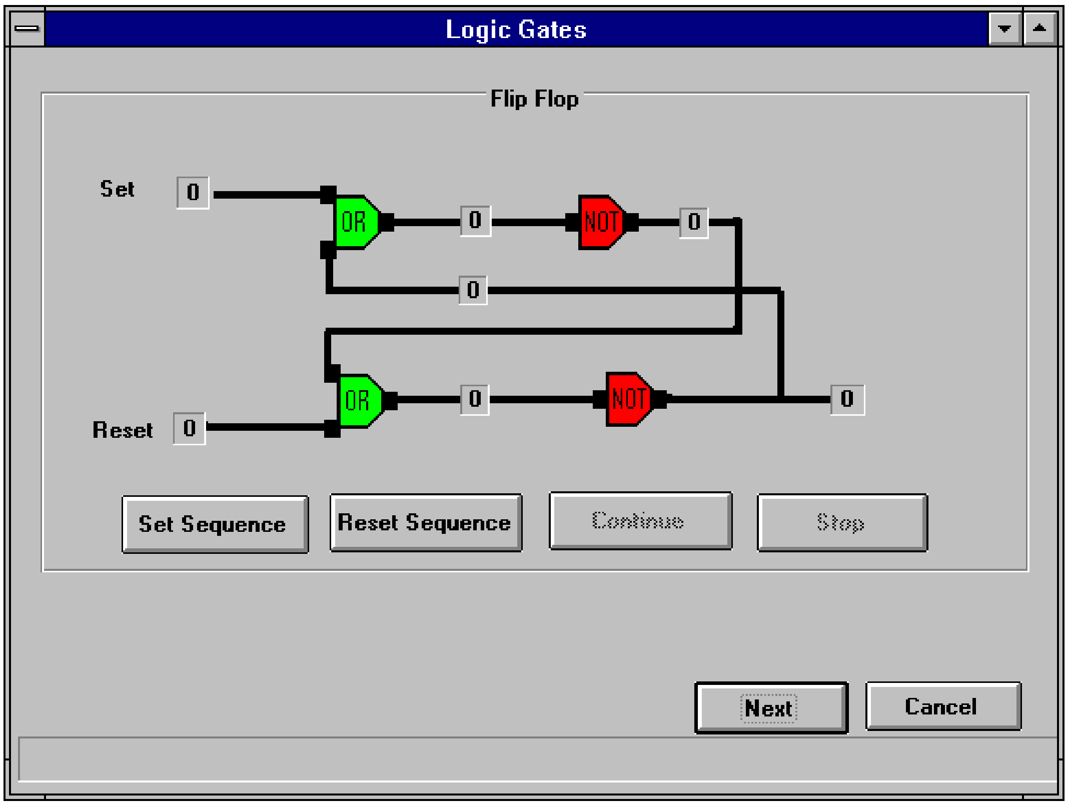
Truth Table

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

You can click on either of the inputs, A and B to toggle their values between 0 and 1. The output at C will change accordingly and the relevant row in the truth table will be highlighted.

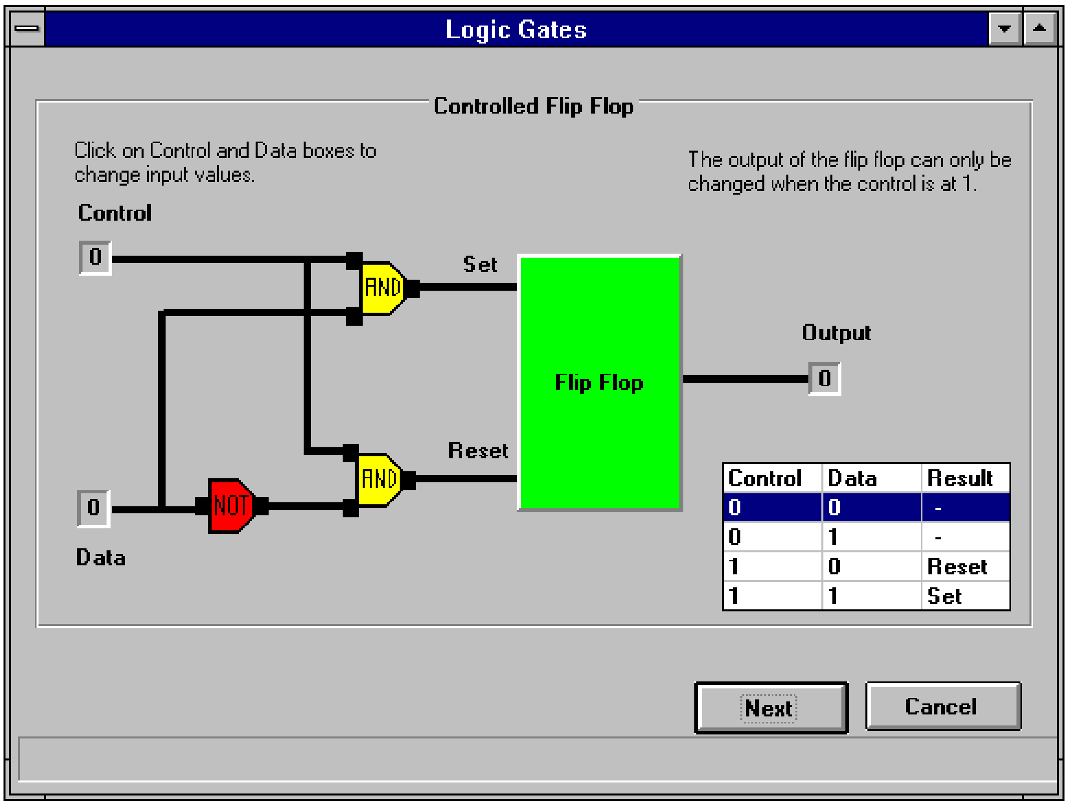
There are similar screens for the OR and NOT gates which are displayed by clicking on **Next**.

The flip-flop screen allows the **Set** and **Reset** sequence to be seen in detail. The screen shows:



To see the Set sequence first click on Set Sequence and then keep pressing Continue until you have seen enough of the sequence to satisfy yourself how it works. Then press Stop. You can view the Reset sequence in a similar fashion by first clicking on Reset Sequence.

Pressing **Next** displays the Controlled Flip-flop screen:



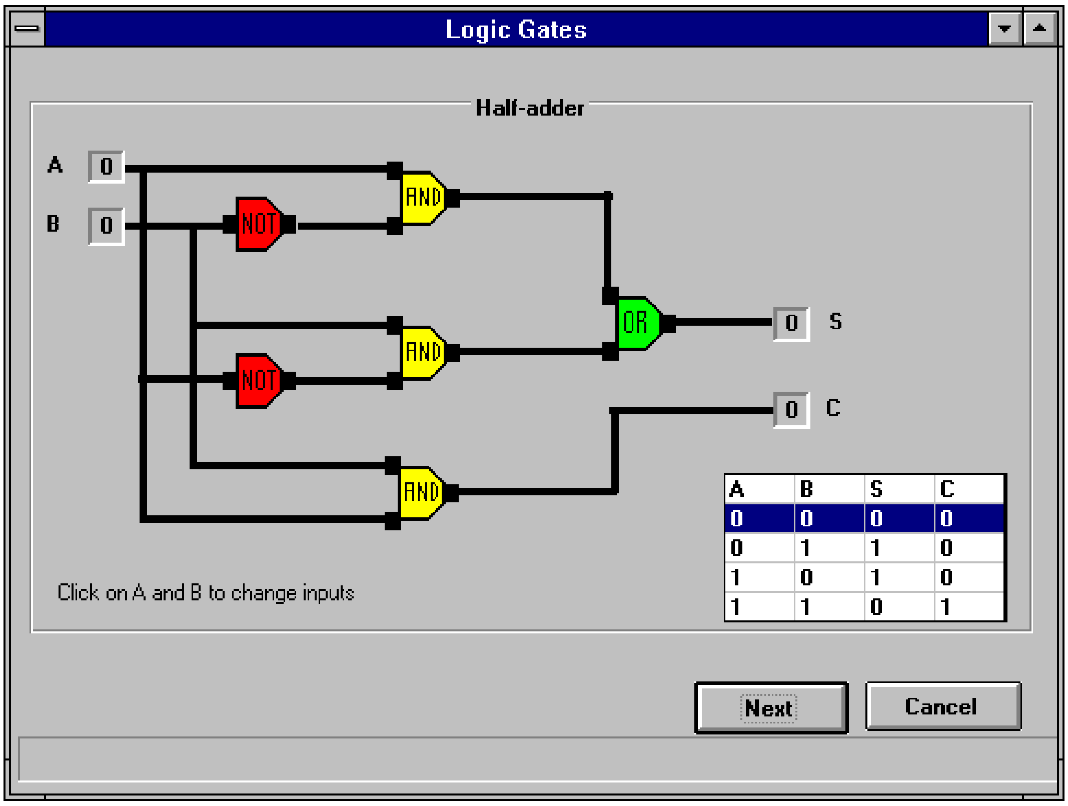
Click on Control and Data boxes to change input values.

The output of the flip flop can only be changed when the control is at 1.

Control	Data	Result
0	0	-
0	1	-
1	0	Reset
1	1	Set

Here, you can click on the Control and Data inputs to toggle their values. The output will change accordingly and the relevant line in the truth table will be highlighted.

Clicking on **Next** will display the half-adder screen:

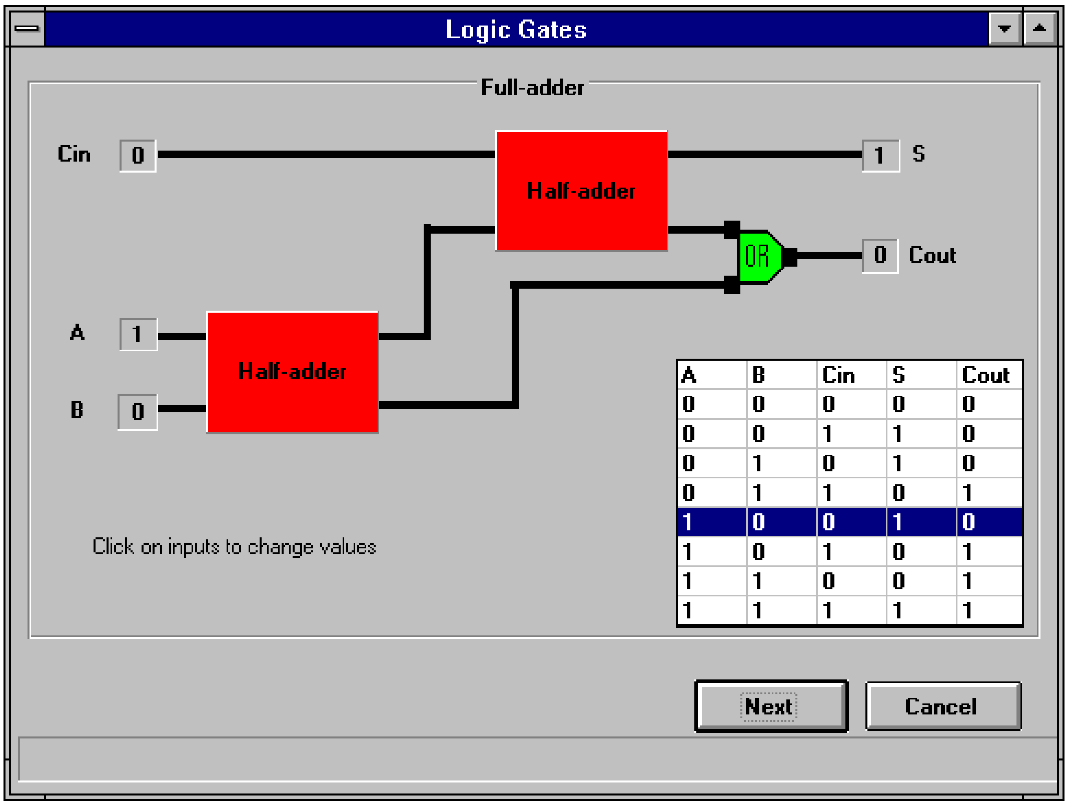


Click on A and B to change inputs

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Again, the inputs can be toggled by clicking on them.

Clicking on **Next** will display the full-adder screen:



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

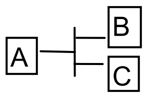
It is possible to 'drill down' to previous screens by double clicking on any of the logical components on the screen. For example, the OR gate screen can be displayed by double clicking one of the OR gates on the above screen.

13.10 How Logic Gates are constructed

As you are no doubt aware, computers are constructed from millions of transistors. The transistors are vital components of Logic Gates. The fact that transistors can be super-miniaturised is overwhelmingly the reason for their use, but additional factors are also that they are cheap and extremely reliable.

A transistor is really nothing other than a switch. When we apply a current to its input, it allows current to flow between its other two connections.

This is how we represent a transistor:



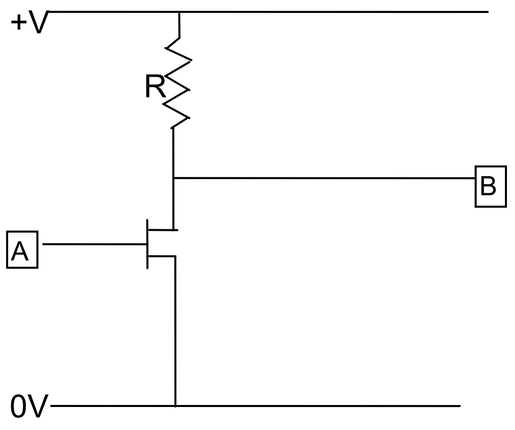
A transistor has three connections which I have labelled as A , B and C. A is called the base, B is the collector, and C is the emitter. When a voltage is applied to the base, i.e. to A, the Emitter and Collector become connected.

So, there are two distinct states that the transistor can be in:

- When no voltage is applied to A, B and C are isolated.

- When a voltage is applied to A , B and C are connected.

This is the simplest circuit that shows how a transistor can be used as a switch:

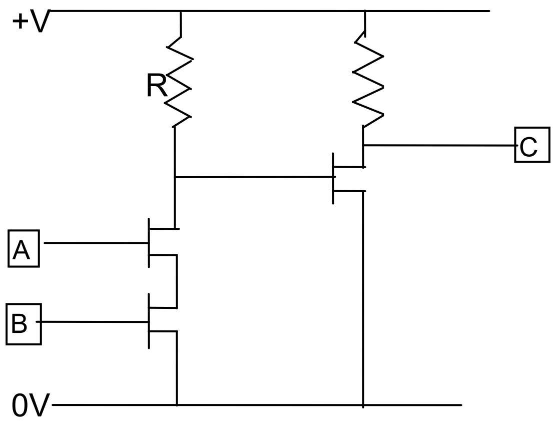


The bottom wire is at zero voltage and the top wire is at some arbitrary positive voltage. When there is no voltage applied to A then the transistor is in an off state, this means that B is effectively connected to the +V wire (via the resistor, R, whose purpose will be explained soon). This means that B is at a relatively high voltage when compared with the bottom wire, i.e. it is equivalent to a logical 1.

When a voltage is applied to A, the transistor is switched on and B is then effectively connected to the 0V line, hence it is set to a logical zero. The purpose of the resistor is to prevent a run away current from flowing between the +V line and the 0V line when the transistor is in the “on” state.

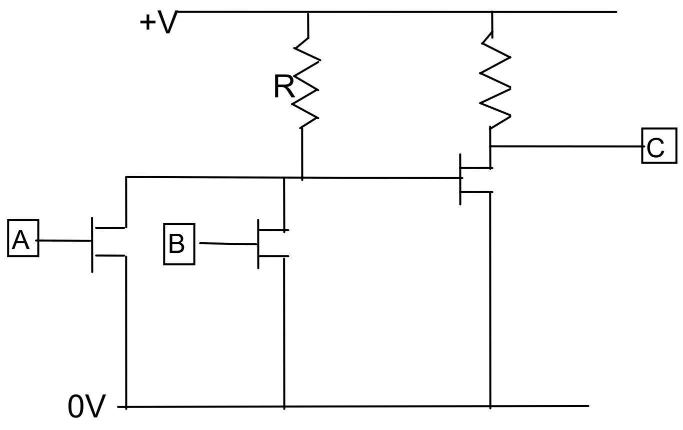
The above circuit is in fact a NOT gate. Put a logical 1 at A and you get a logical 0 at B. Put a logical 0 at A and you get a logical 1 at B.

Constructing an AND gate requires us to connect two transistors in series and then pass the result through a NOT gate. Consider the following circuit:



Both of the left hand transistors need to be switched on to get a logical 0 as input to the third transistor. i.e. to get a logical 1 at C both A and B need to be at logical 1.

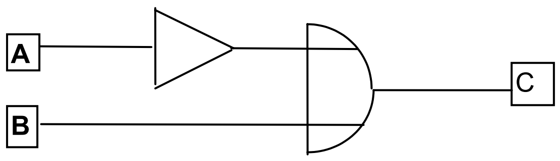
An OR gate can be constructed in a similar fashion, but both input transistors now need to be connected in parallel. Consider the following circuit:



Here, it is enough for either of the input transistors to be switched on for there to be a logical zero at the input of the third transistor. i.e. When either A **or** B is at a logical 1 then C will be at logical 1.

Exercises I

(1) Write out the truth table for the following logical circuit:

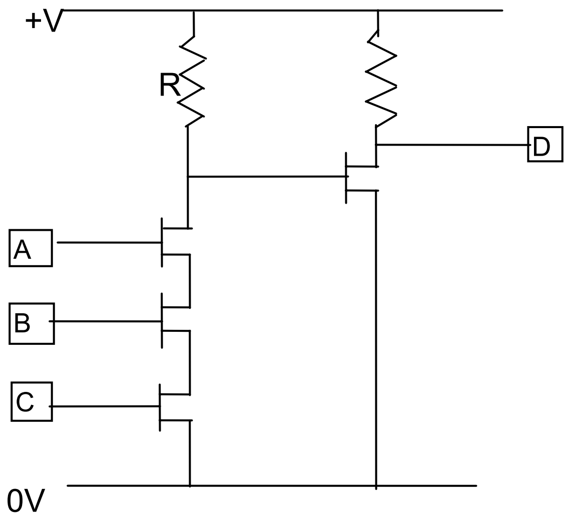


(2) A NOR gate has the following truth table:

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

What is the logical circuit needed to implement this?

(3) What logic does this circuit implement?



14 Memory

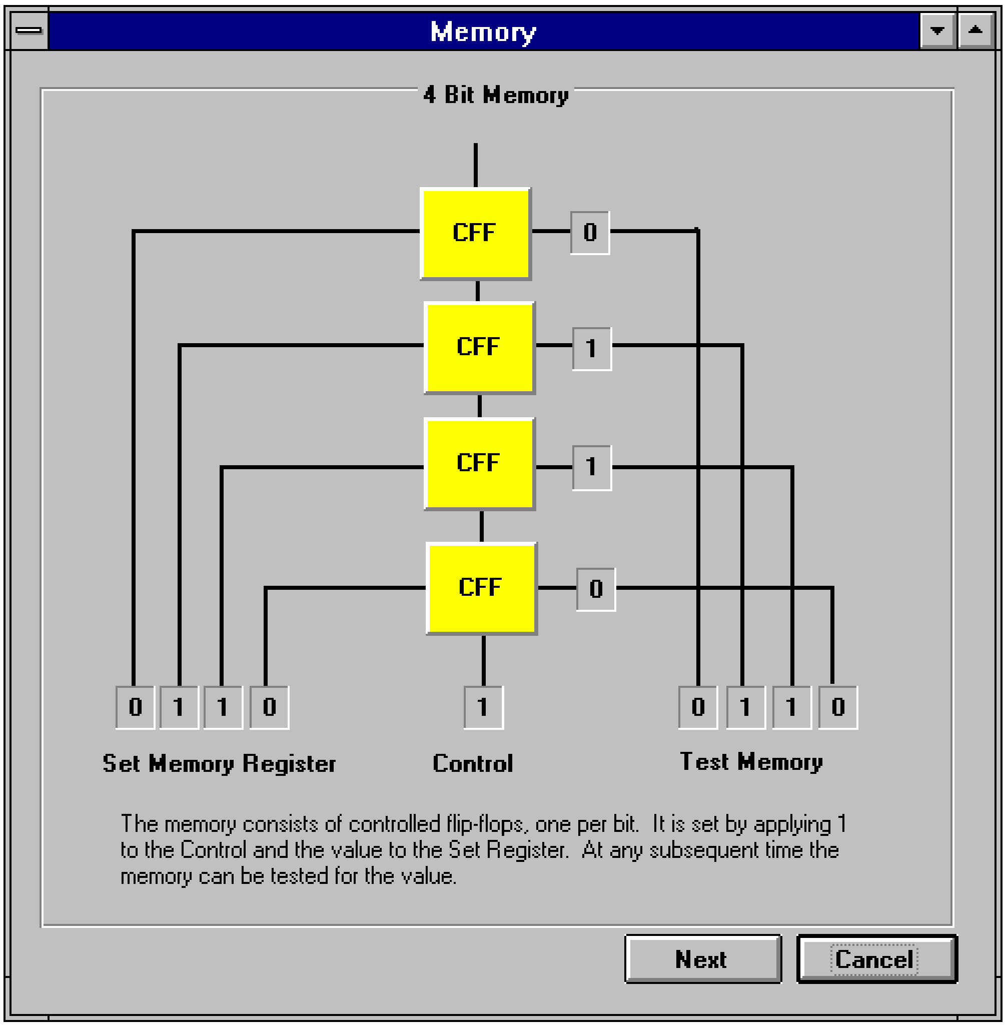
14.1 Introduction

TOM’s memory screens illustrate how values can be stored to and retrieved from memory. It is essential that the screens on logic gates have been fully understood as the basis for the construction of memory is the controlled flip-flop.

14.2 Setting and Storing Memory

TOM’s first memory screen shows how a single memory location can be constructed from an array of controlled flip-flops. Each controlled flip-flop is capable of storing a single bit of data. So for storing four bits we need four controlled flip-flops.

The screen shows a single 4 bit memory location. There are 4 inputs, one for each bit, and 4 outputs. There is also one control line which is connected to **all** the controlled flip-flops.



The memory consists of controlled flip-flops, one per bit. It is set by applying 1 to the Control and the value to the Set Register. At any subsequent time the memory can be tested for the value.

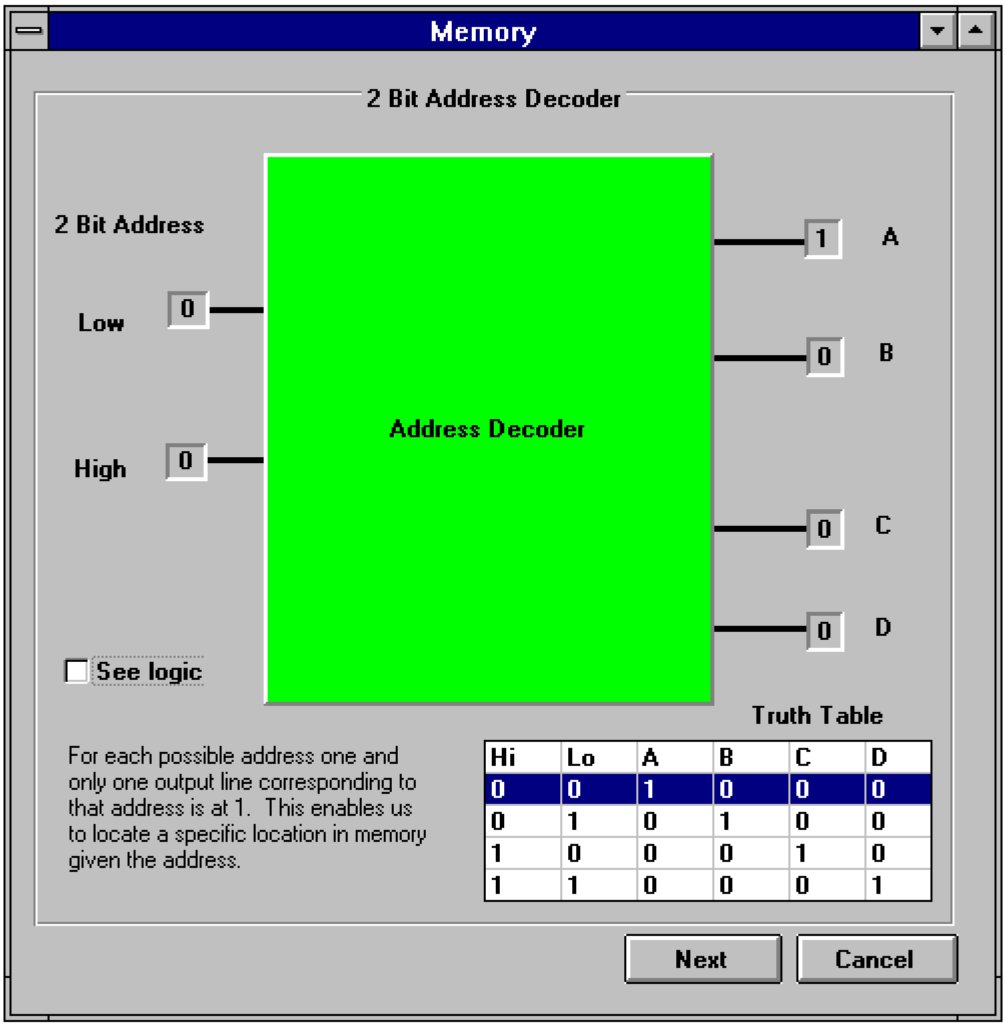
Each of the controlled flip-flops is only capable of changing state when the control line is at logical 1. Therefore to change the memory contents, you need to set the input memory registers on the left to your required value, and then switch the control to logical 1 and back to logical 0. At the point that the control line is at logical 1, the 4 bit value will be remembered by the memory and this value is available at the output labelled **Test Memory**.

14.3 Address Decoding

In order to get values out of memory we need to use an Address Decoder. What we need is to be able to identify a particular memory location given a particular address. The first part of this process is to be able to take an address and obtain an output that can be used to identify the value at that address.

The simplest situation is when we are dealing with a maximum of four addresses. We can arrange for a device to input a 2 bit address, and to have one output for each of the possible addresses. One and only one output will be at logical 1 for each of the possible inputs. These output lines can then be connected to each memory addressed and used for obtaining the value at that address. This is elaborated on in the final memory screen.

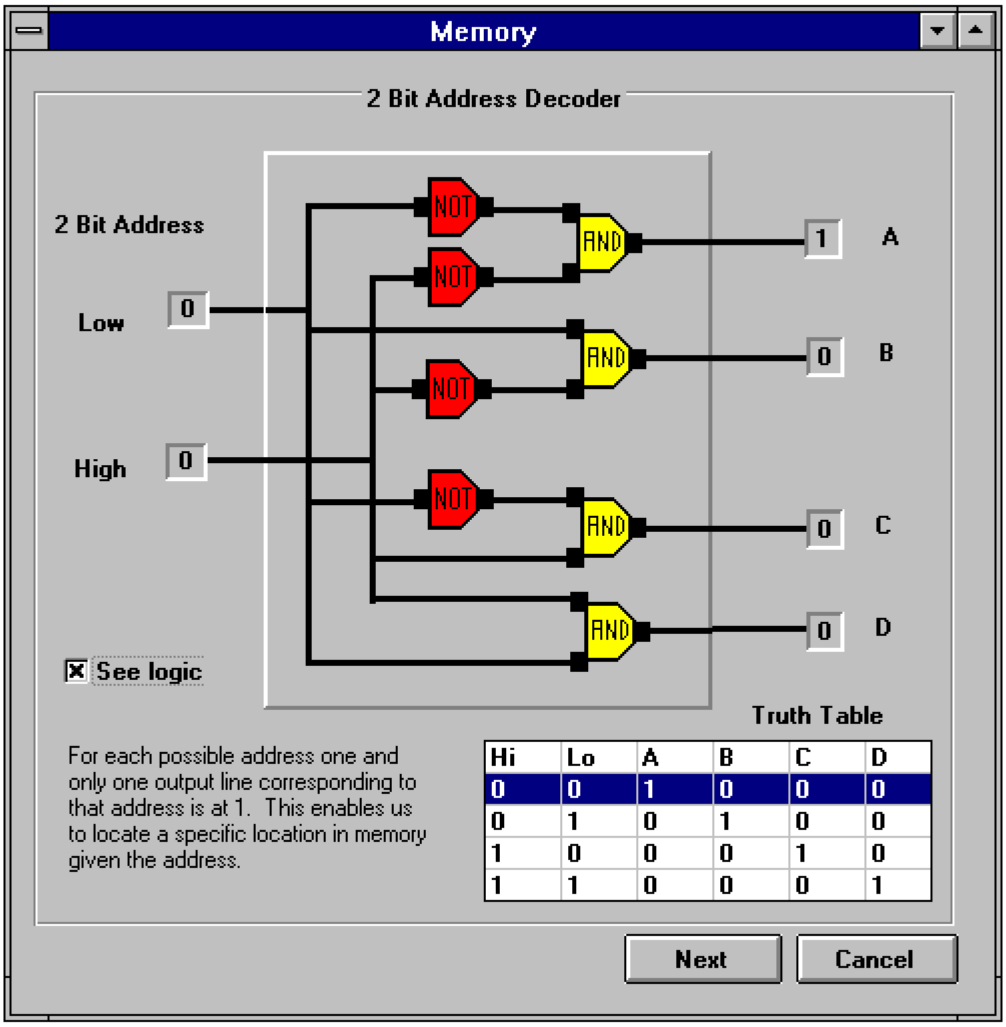
For the time being consider this device. Click on Low and High registers on the left to satisfy yourself of what it is doing.



For each possible address one and only one output line corresponding to that address is at 1. This enables us to locate a specific location in memory given the address.

Hi	Lo	A	B	C	D
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The underlying logic of how such an Address Decoder is constructed can be seen by checking the **See Logic** box on the left of the screen. This will then display:



For each possible address one and only one output line corresponding to that address is at 1. This enables us to locate a specific location in memory given the address.

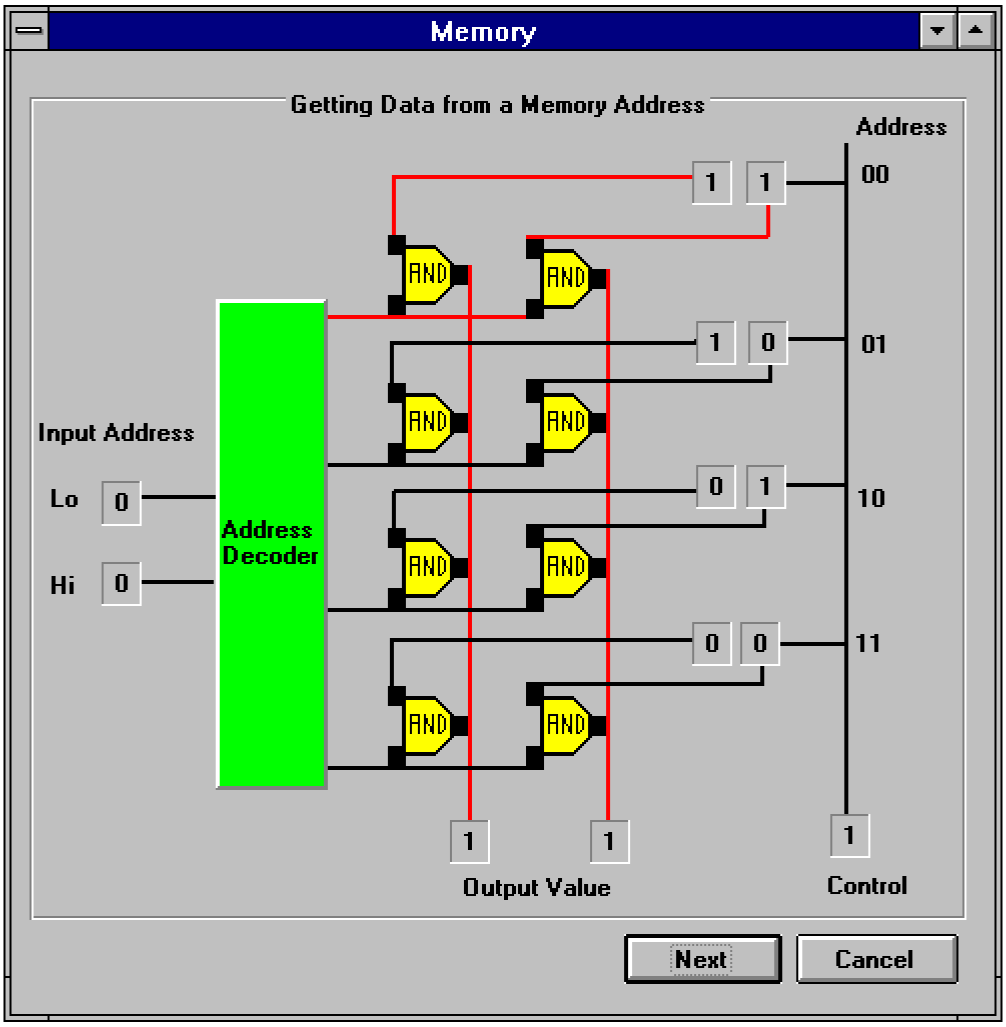
Hi	Lo	A	B	C	D
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Clearly a total of 4 memory locations is not going to be very useful. Fortunately we can connect two Address Decoders together to address eight memory locations. In fact if we can address 2^N+1 address locations with N address decoders. This point will be taken up again after the last of TOM’s memory screens has been discussed.

14.4 Fetching values from memory

The real purpose of memory is that, having put values into memory, we want to be able to say ‘what is the value at address X?’ (write only memory would have very little value!). i.e. we want to be able to fetch a data value from any address, or put another way, given an address we want the value at that address.

This can only be achieved by using an arrangement of AND gates. For the simple case where we have only 4 memory locations we can use a single 2 bit address decoder. TOM illustrates this case:



This illustrates that when the Address Decoder decodes address 0, its first output is at logical 1. This value is **anded** with the contents of the memory values at address 0 and so whatever is at these locations appears as the output value. **ALL THE OTHER LOCATIONS ARE IGNORED** because only the first of the address decoders outputs is at logical 1, all the others are at 0, and these are **ANDED** with the other locations (The result of **ANDING** 0 with anything is always 0).

Try putting your own values into the memory locations on the right of the screen by clicking on the memory cells, remember that these can only be changed when the control line is at logical 1.

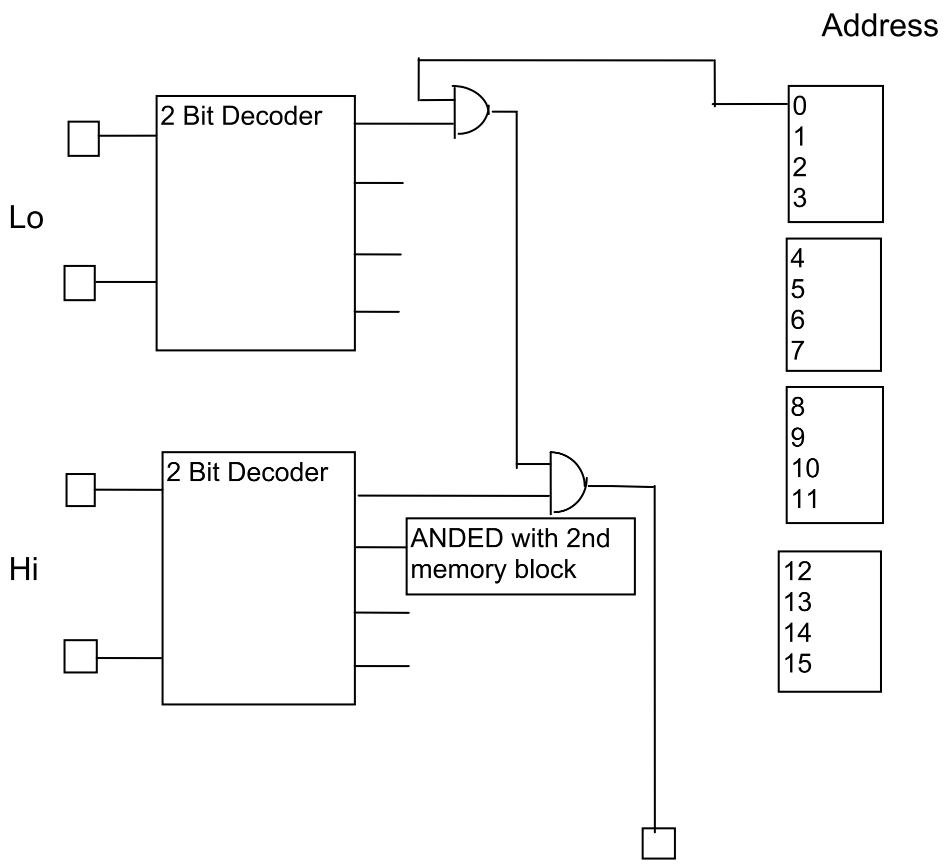
Now try changing the Input address to see how the values at the relevant memory location is logically arrived at in the Output.

14.5 Addressing larger memory ranges

As has been noted, a memory with only four distinct addresses is of very limited value. In order to address a larger range we need to connect more address decoders into the circuit.

In order to try and represent this on a diagram we need to simplify our representation somewhat. Instead of showing the entire set of gates for each bit of a memory location, I will simply show a single gate for the whole location at a particular address. This doesn't really lose us anything as we can quite easily replace the single gate with one for each bit if we needed to. The important point is that we are logically ANDing the contents of all bits at that address with some output of the Address Decoder.

Consider how we would address 16 locations:



We need to think of the address range as being split into 4 equal parts of 4 addresses each.

The two decoders are clearly capable of being given 16 different inputs, i.e. 0000 through to 1111 (0 to 15 in decimal).

The principal is to:

- AND together the first 4 addresses with the 0 output of the ‘Hi’ decoder’
- AND together the second 4 addresses with the 1 output of the ‘Hi’ decoder’
- AND together the third 4 addresses with the 2 output of the ‘Hi’ decoder’
- AND together the fourth 4 addresses with the 3 output of the ‘Hi’ decoder’

In this way each output of the Hi memory decoder is controlling the block of memory that it addresses. Whilst each output of the Lo memory decoder is controlling which of the addresses within that address range is being addressed.

Exercises J

(1) Draw the logic diagram for the two bit address decoder and mark on the diagram all the logical values for the case where the low input bit is 0 and the high input bit is 1.

(2) Complete the diagram given in the section ‘Addressing larger memory ranges’ for the first two memory blocks.

15 TOM's machine simulator

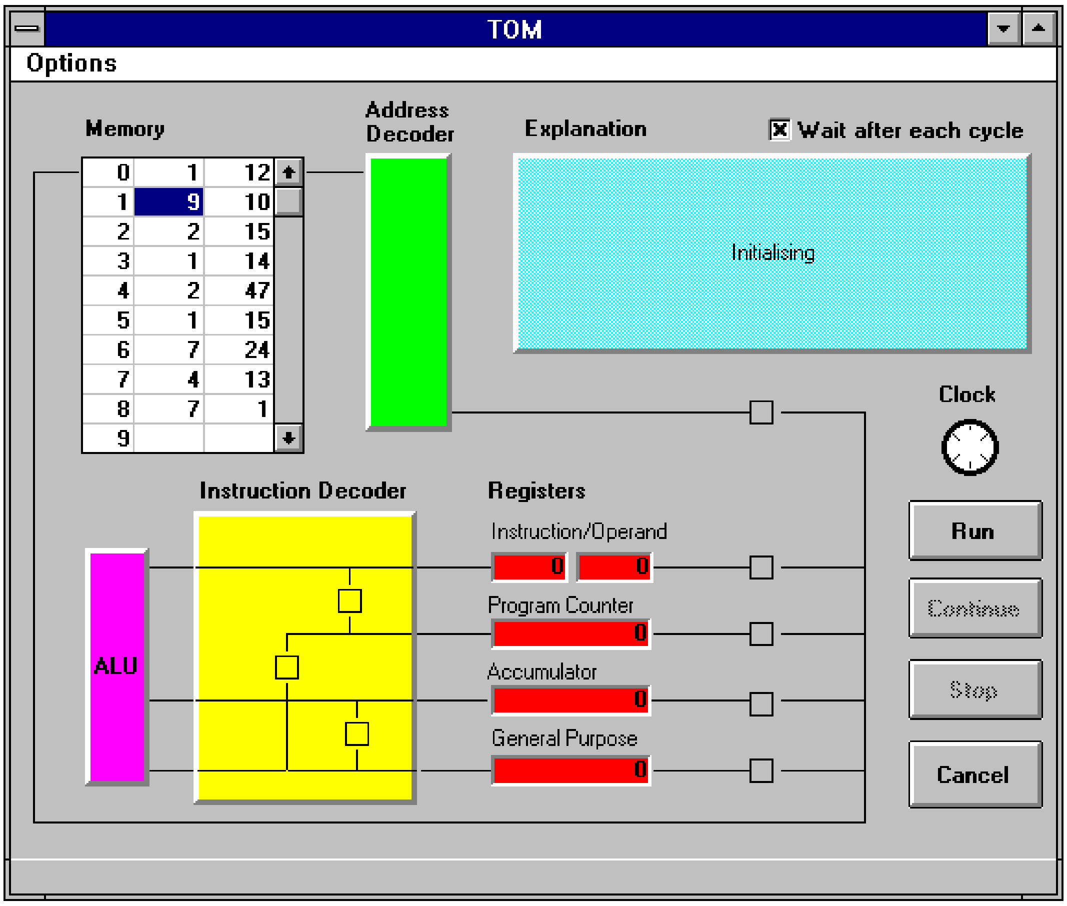
15.1 Introduction

TOM's machine simulator provides an accurate simulation of how a program is executed by a computer. The simulator will execute any program that TOM can normally execute. It gives a stage by stage demonstration of each part of the execution cycle, graphically displaying the various flows of data from one component to another.

It is also possible to guide the operation of the machine by use of the mouse allowing the user to become fully aware of the execution cycle.

15.2 The Machine Simulator's components

With a program already in TOM's memory (it's a good idea to choose a working program) and the program counter set to 0, choose **Machine** from the **Views** menu. You will see the following (although the contents of the memory will almost certainly be different):



0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

Initialising

Instruction/Operand: 0 0
Program Counter: 0
Accumulator: 0
General Purpose: 0

Status panel

At the bottom of the screen is a status panel, moving the mouse over the various components of TOM will cause descriptions of the components to appear on this panel. Try this now.

Memory

TOM’s memory is displayed at the top-left of the screen. You should be able to see that it has the same values at the same locations as TOM’s main screen. However, unlike TOM’s main screen you cannot interactively alter these values. If you need to change a memory value, you will need to go back to TOM’s main screen and edit your program in the usual manner.

Address decoder

To the right of the memory is the Address Decoder. This will identify the memory location which corresponds to an input address. The address will be displayed in the Address Decoder and when it is activated it will then highlight the location corresponding to this address in the Memory unit.

Explanation

The blue panel to the right of the Address Decoder contains text that explains the current operation that the machine is performing.

Arithmetic Logic Unit (ALU)

Near the bottom left of the screen is the Arithmetic Logic Unit, this is the part of the machine that performs arithmetic operations on the registers’ contents.

Instruction decoder

To the right of the Arithmetic Logic Unit is the Instruction Decoder. This performs operations on the registers which correspond to the instruction in the instruction register.

Registers

To the right of the Instruction Decoder is a set of registers. Two of these are the same as on TOM’s main screen, the Program Counter and the Accumulator. In addition is the Instruction/Operand pair which stores the current instruction being executed and its operand. There is also an internal General Purpose register, which, as its name implies, is used for a variety of purposes.

The Bus

The black line which connects all the components is the **Bus**. The purpose of the Bus is to allow various machine components to transfer values between them. The small black boxes represent gateways onto the bus. These are either open or closed. They are shown above in their original state of being closed. No information can flow through a gateway when it is closed. They appear with a black cross when they are open.

An example of a use of the bus is when a data value is transferred from the memory into one of the registers.

Control buttons

There are a set of buttons to the right of the registers which you can use to control the machine’s execution.

Clock

Above the control buttons is a small clock icon which indicates the part of the cycle that the machine is executing. The clock controls the opening and closing of the various gateways onto the bus. It is the timing of these openings of the Gateways which is so important to how the machine works.

15.3 Running the Machine Simulator

Program execution is started by clicking on **Run**.

15.4 TOM’s execution cycle

The simple explanation that is often given to describe a machine’s operation is that the machine enters a cycle of fetching instructions and executing them. Although this is accurate it is overly simplistic and needs to be broken down into rather more components for a complete understanding. TOM’s machine has precisely seven which are described in the following:

The first few cycles are involved with fetching the instruction and its operand from memory. To do this it needs to:

- Send the contents of the Program Counter to the Address Decoder.
- Copy the requested memory value into the Instruction and Operand registers.

The first part is needed in order to identify the correct instruction/operand pair in memory. Clearly it is the Program Counter which determines where in memory this pair are, this is what the Program Counter is for!

The next few cycles are used to determine what the Operand is referring to. We know that the operand always refers to a location in memory. Therefore the only logical thing to do is to fetch this value and store it in an internal register, TOM's machine uses the General Purpose Register for this. So in order to fetch what the operand is referring to the machine needs to:

- Send the contents of the Operand Register to the Address Decoder.
- Copy the requested memory value into the General Purpose Register

Now that the needed values have been fetched into the Machines registers it's time to execute the instruction. So the next cycle of the machine is simply:

- Execute the instruction.

Some instructions are capable of altering values in memory, for example the Store operation. So it is essential that the execution cycle includes a write to memory, even though this will not be used by many instructions, without it we could never get our programs to change the contents of memory (all the cycles to this point have simply been reading memory). Therefore the next phase is to write the contents of the General Register to memory:

- Send the contents of the General Purpose Register to Memory.

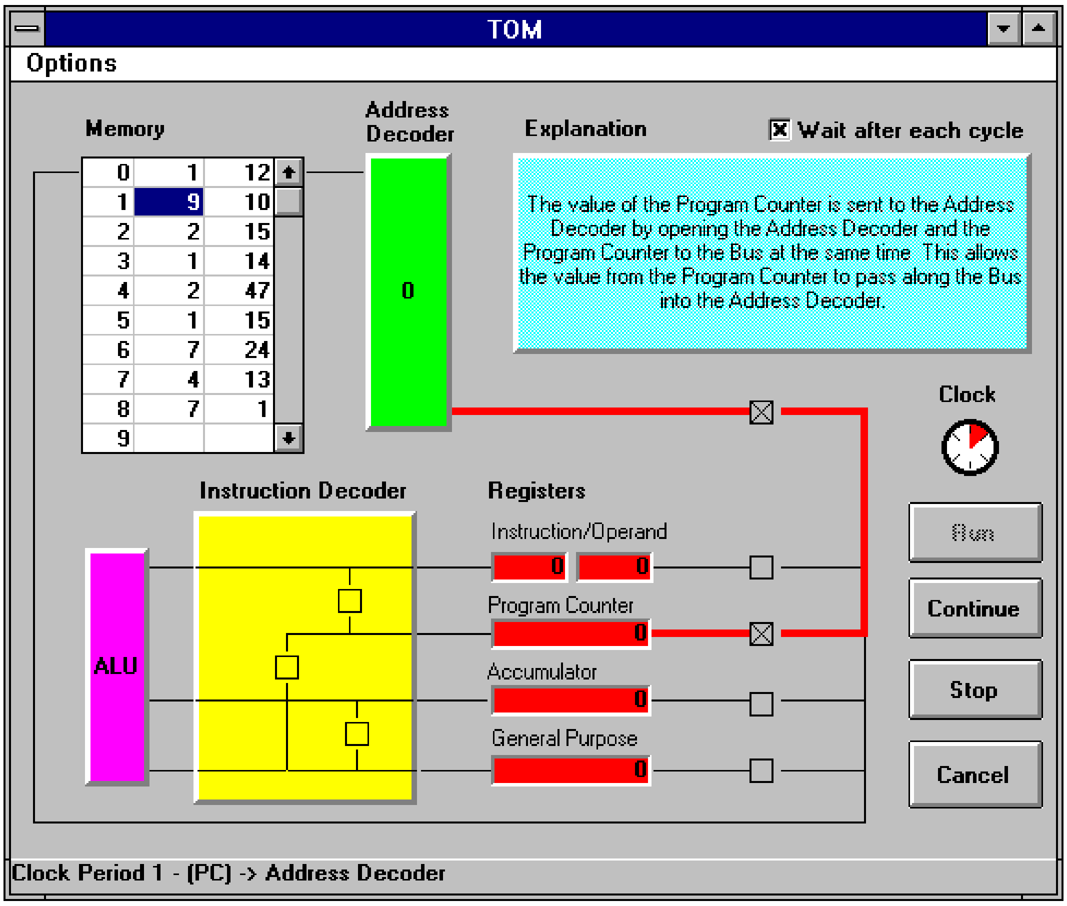
Finally we need to advance the Program Counter so that the next instruction can be fetched and executed. **BUT**, this cycle is missed out if the instruction executed was any one of the JUMP instructions

- Increment the Program Counter.

Now the whole process can start again, but of course this next time it will be fetching and executing the next instruction.

Watching TOM's machine perform these instructions should make the whole process much clearer.

Machine cycle clock Period 1



0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

The value of the Program Counter is sent to the Address Decoder by opening the Address Decoder and the Program Counter to the Bus at the same time This allows the value from the Program Counter to pass along the Bus into the Address Decoder.

Clock Period 1 - (PC) -> Address Decoder

The value of the Program Counter is sent to the Address Decoder by opening the Address Decoder and the Program Counter to the Bus at the same time This allows the value from the Program Counter to pass along the Bus into the Address Decoder.

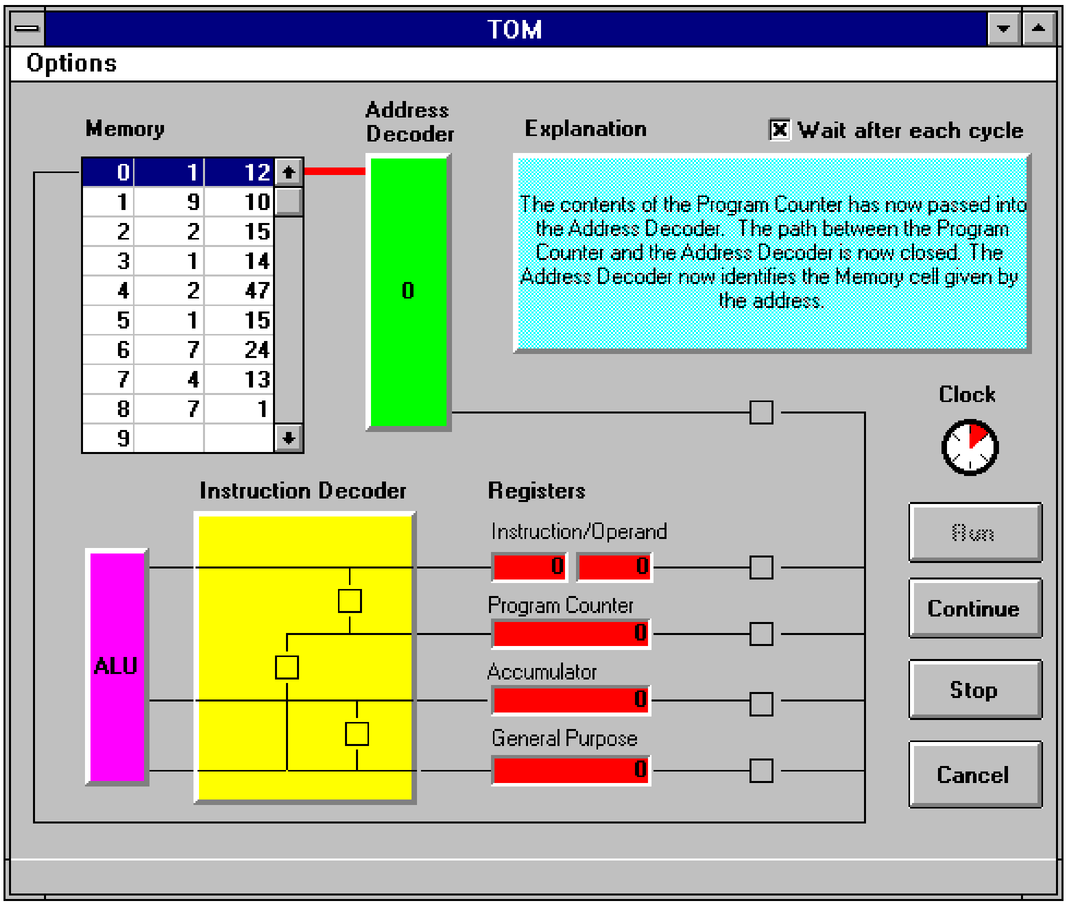
Note the red line which indicates the path that the data takes from the Program Counter to the Address Decoder. Also note that the two necessary gateways onto the bus are open to allow this to happen.

The clock is indicating that the first part of the Machine Cycle is being executed.

The status panel gives a short description of the current operation, note that (PC) means the contents of the Program Counter.

The current value of the Program Counter is displayed in the Address Decoder.

Pressing **Continue** displays the final part of Period 1:



0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

The contents of the Program Counter has now passed into the Address Decoder. The path between the Program Counter and the Address Decoder is now closed. The Address Decoder now identifies the Memory cell given by the address.

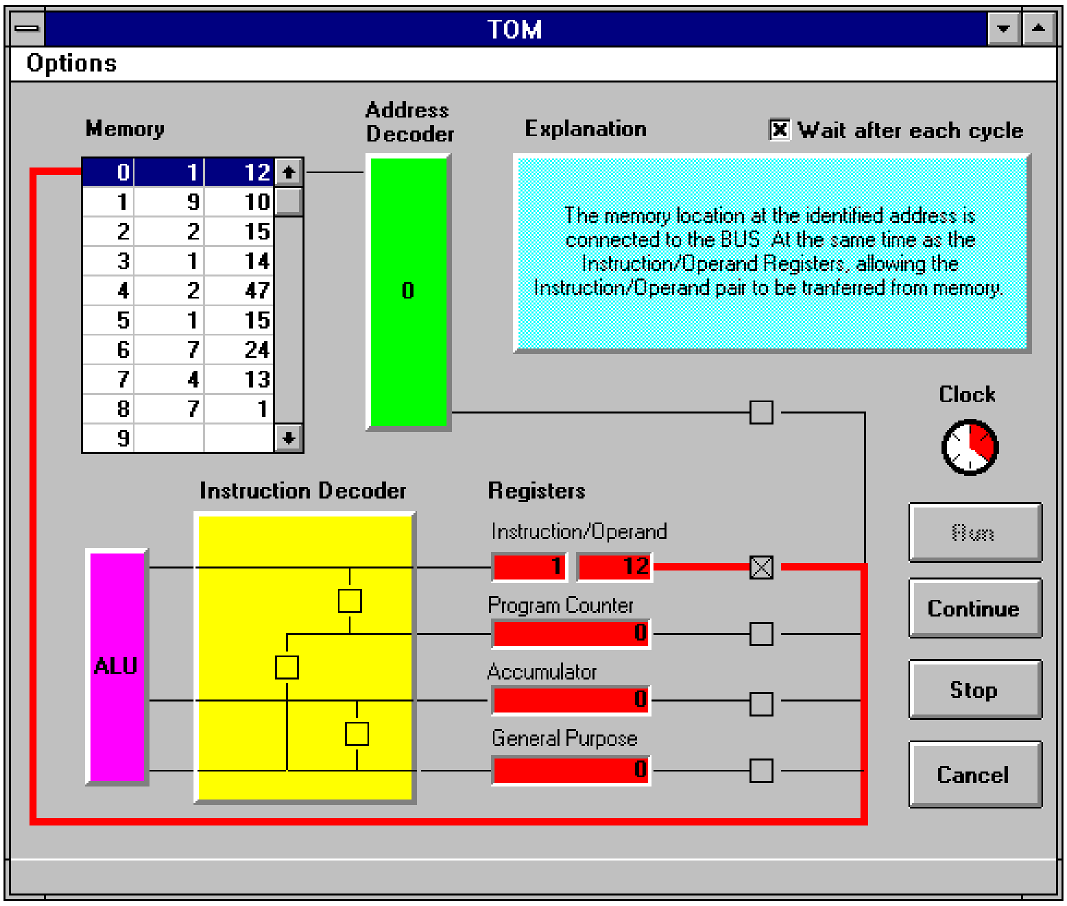
Instruction/Operand: 0 0
Program Counter: 0
Accumulator: 0
General Purpose: 0

The path between the Program Counter and the Address Decoder is now closed. The Address Decoder is activated and identifies the Memory cell given by the address.

Note how the relevant location in memory is now highlighted and appears at the top of the memory unit.

Machine cycle clock Period 2

The instruction and operand are now read from memory into the instruction/operand registers.



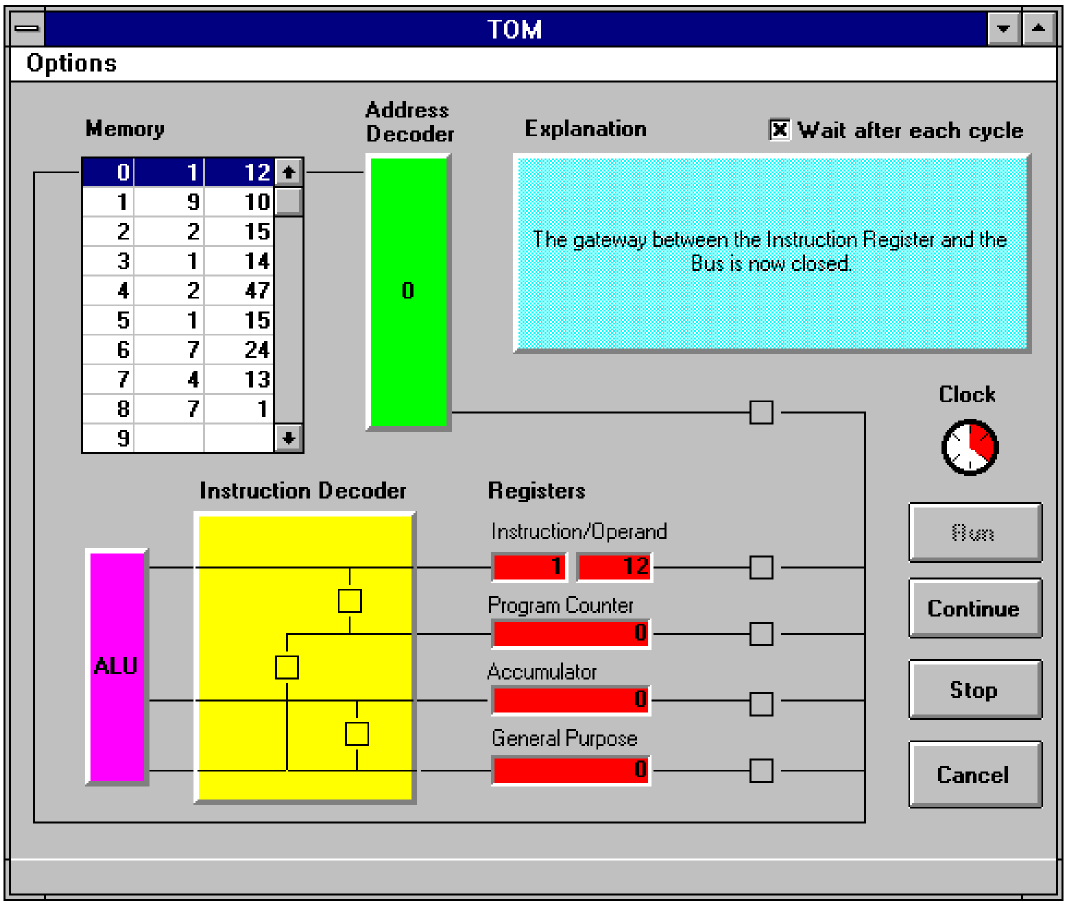
The memory location at the identified address is connected to the BUS. At the same time as the Instruction/Operand Registers, allowing the Instruction/Operand pair to be tranferred from memory.

0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

Address Decoder: 0

Instruction/Operand: 1 12
Program Counter: 0
Accumulator: 0
General Purpose: 0

The memory location at the identified address is connected to the bus. At the same time as the Instruction/Operand Registers, allowing the Instruction/Operand pair to be transferred from memory into the registers.



[✓ Wait after each cycle]

Memory

0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

Address Decoder: 0

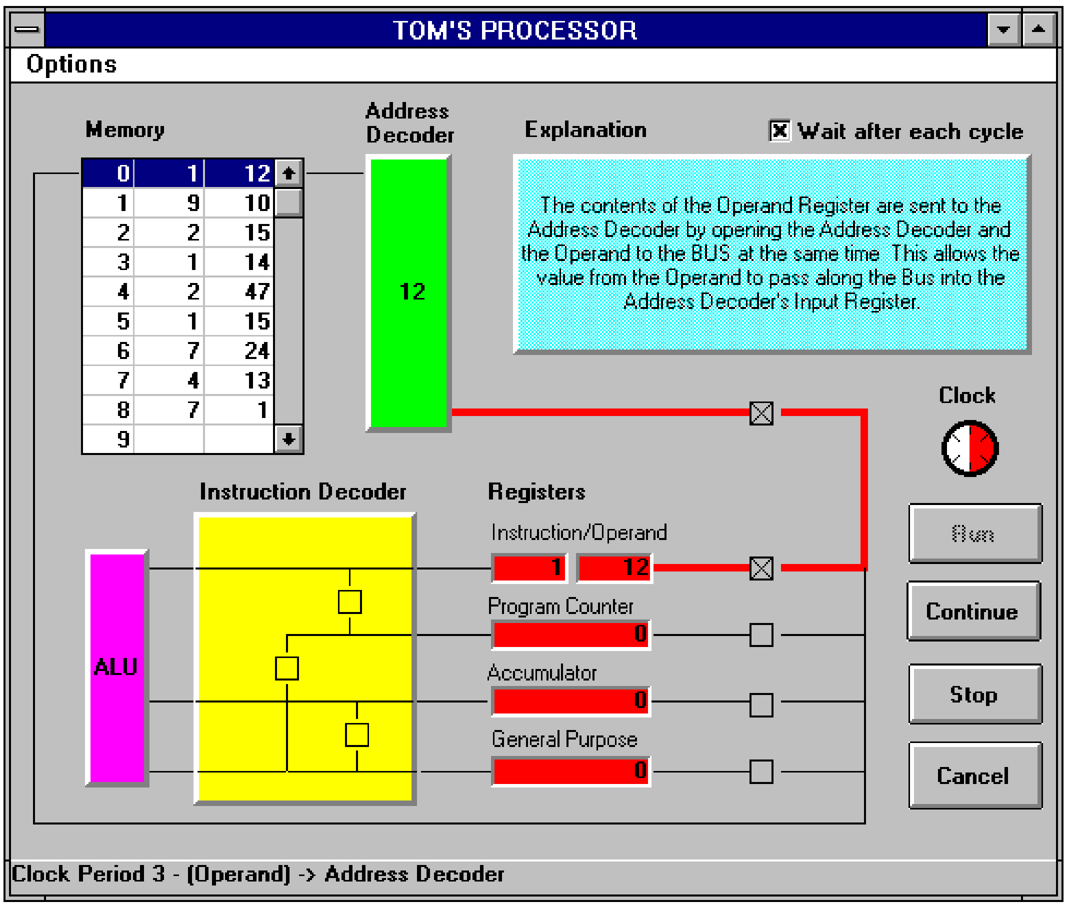
Explanation: The gateway between the Instruction Register and the Bus is now closed.

Registers		
Instruction/Operand	1	12
Program Counter	0	
Accumulator	0	
General Purpose	0	

The gateway between the Instruction Register and the Bus is now closed.

Machine cycle clock Period 3

Now we need to fetch the operand value, i.e. the value that the operand is referring to, from memory. This involves sending the contents of the Operand register to the address decoder. This is to enable the value at that address to be read:



[✓ Wait after each cycle]

0	1	12
1	9	10
2	2	15
3	1	14
4	2	47
5	1	15
6	7	24
7	4	13
8	7	1
9		

The contents of the Operand Register are sent to the Address Decoder by opening the Address Decoder and the Operand to the BUS at the same time. This allows the value from the Operand to pass along the Bus into the Address Decoder's Input Register.

Instruction/Operand: 1 12
Program Counter: 0
Accumulator: 0
General Purpose: 0

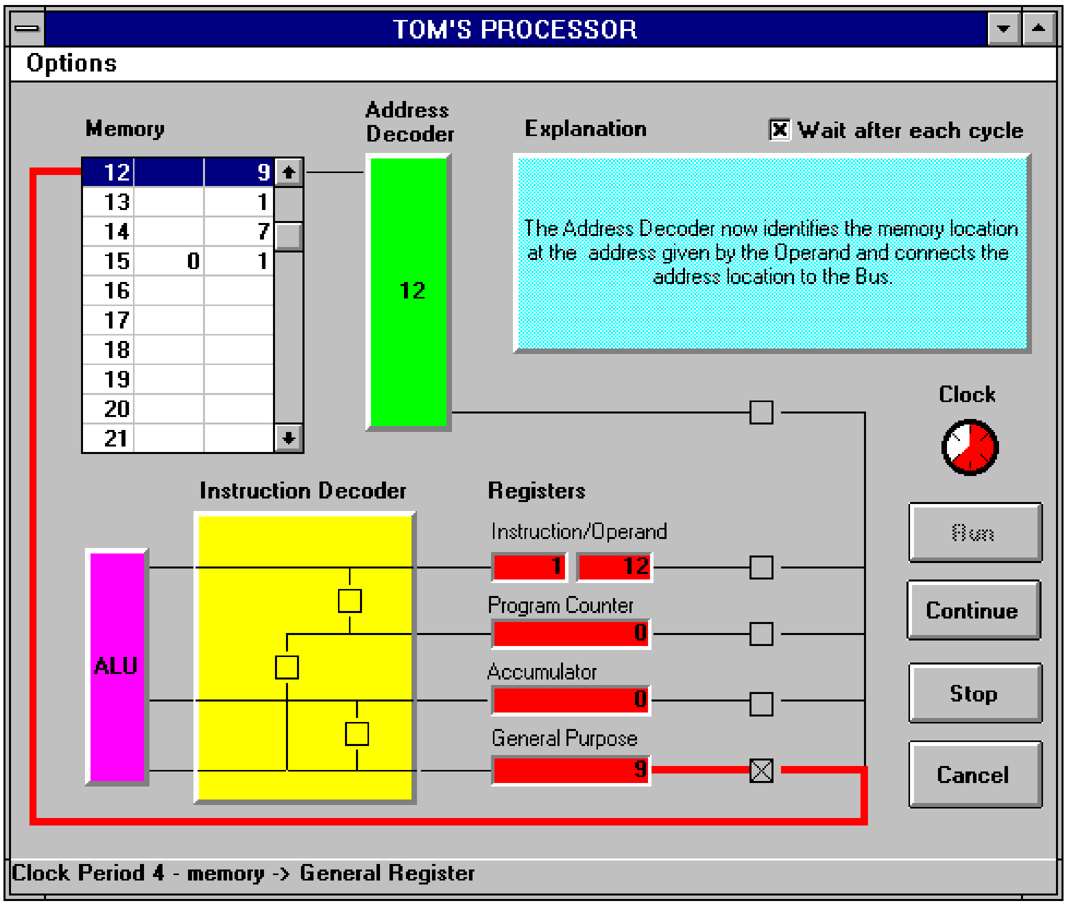
Clock Period 3 - (Operand) -> Address Decoder

The Operand register and the Address Decoder are both opened to the BUS allowing the value from the Operand Decoder to be read into the Address Decoder.

This causes the memory location at address 12 to be identified for the subsequent transfer.

Machine cycle clock Period 4

The contents of the memory location just identified (i.e. the operand value) is now read into the General Purpose register.



12		9
13		1
14		7
15	0	1
16		
17		
18		
19		
20		
21		

The Address Decoder now identifies the memory location at the address given by the Operand and connects the address location to the Bus.

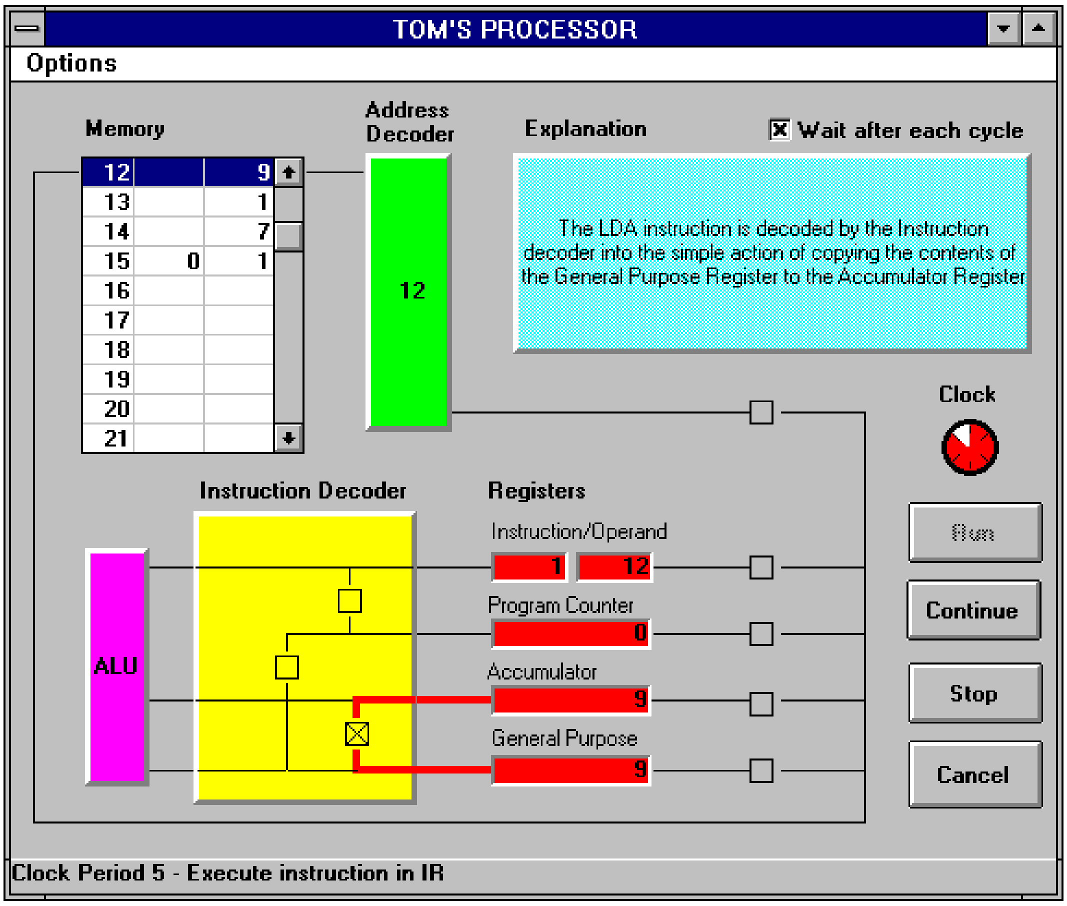
Instruction/Operand: 1 12
Program Counter: 0
Accumulator: 0
General Purpose: 9

Clock Period 4 - memory -> General Register

The General Purpose register is opened to the BUS allowing the memory value from location 12 to be transferred into the General Purpose register.

Machine cycle clock Period 5

Now the instruction in the instruction register can be executed:



12		9
13		1
14		7
15	0	1
16		
17		
18		
19		
20		
21		

Address Decoder: 12

The LDA instruction is decoded by the Instruction decoder into the simple action of copying the contents of the General Purpose Register to the Accumulator Register

[✓ Wait after each cycle]

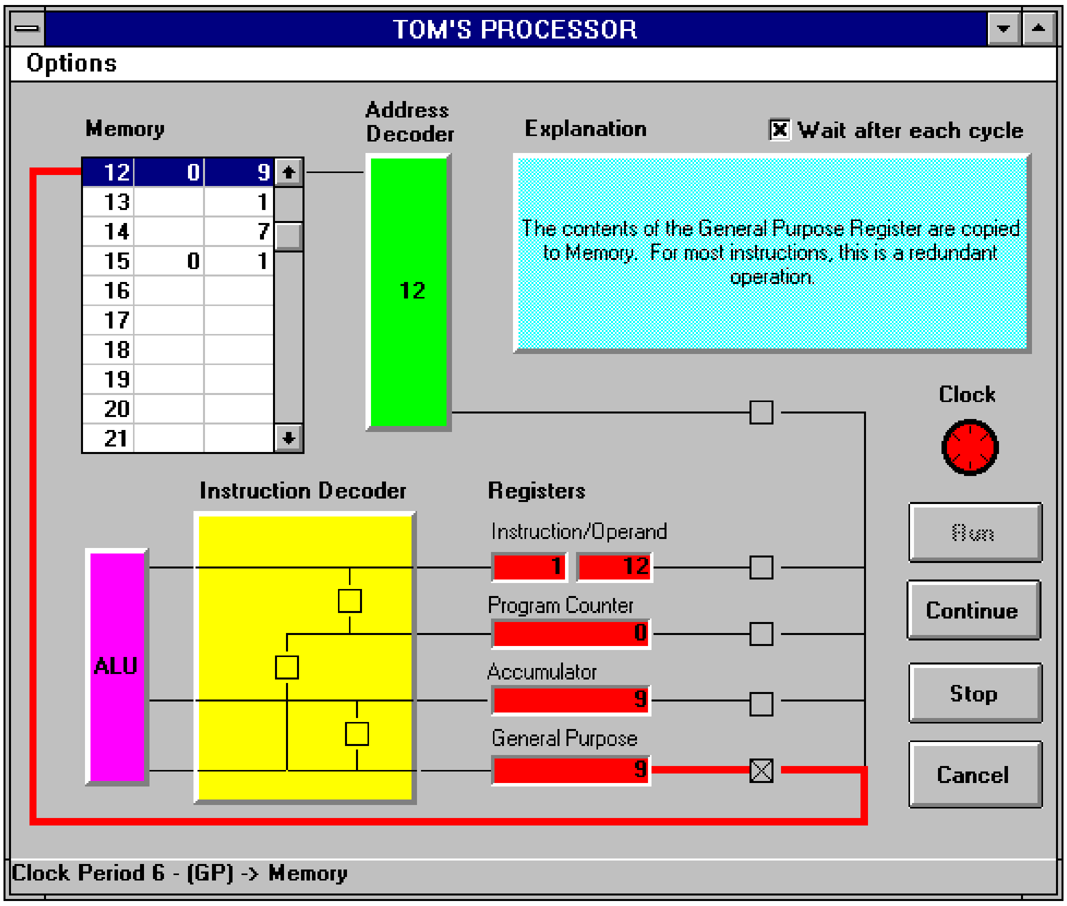
Instruction/Operand: 1 12
Program Counter: 0
Accumulator: 9
General Purpose: 9

Clock Period 5 - Execute instruction in IR

The instruction being executed here is the Load Accumulator instruction, which simply copies the contents of the General Purpose register to the Accumulator. This is one of the simplest operations that the Instruction Decoder has to do, in this case it is only necessary to transfer the value from one register to another by opening an internal gateway as illustrated within the Instruction Decoder in the above screen shot.

Machine cycle clock Period 6

The contents of the General Purpose register are sent to memory. This is a redundant operation for this instruction, but it is an essential part of the entire machine cycle, as the processor must be capable of writing to memory as some of the instructions require (i.e. STORE ACCUMULATOR).



12	0	9
13		1
14		7
15	0	1
16		
17		
18		
19		
20		
21		

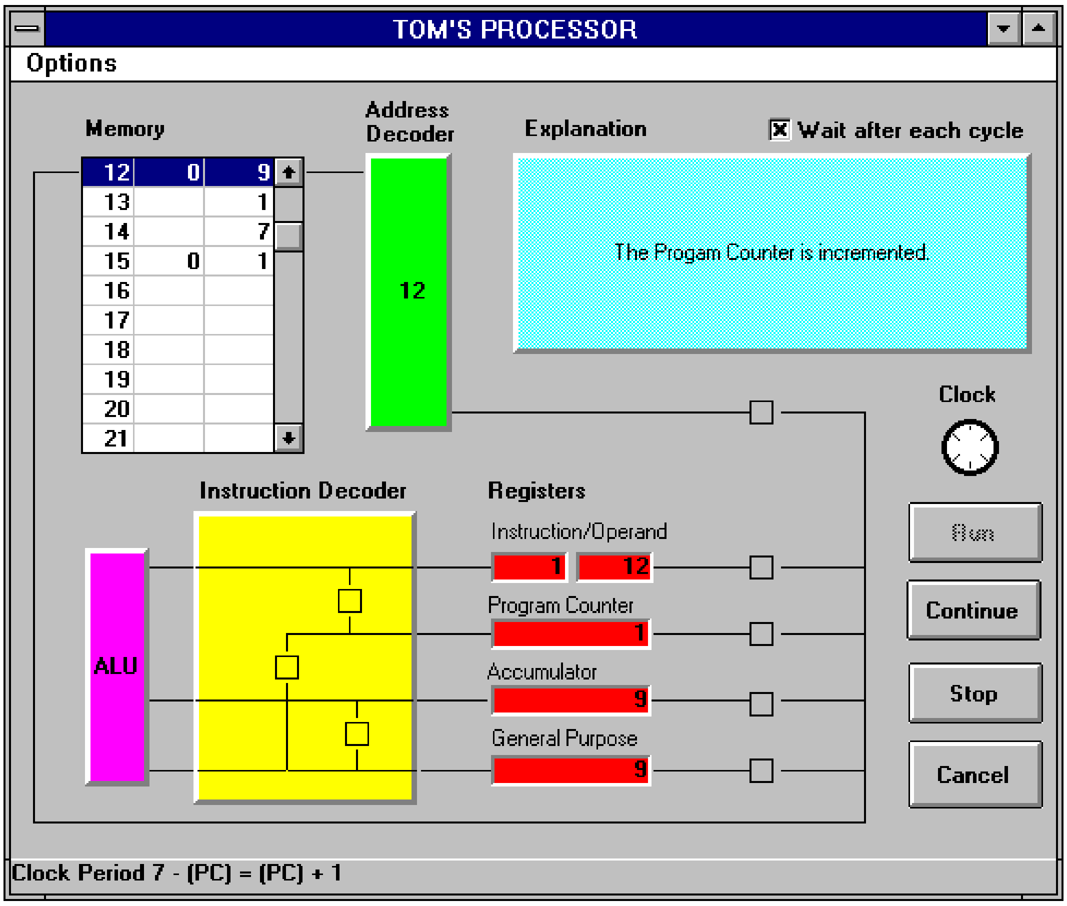
The contents of the General Purpose Register are copied to Memory. For most instructions, this is a redundant operation.

Instruction/Operand: 1 12
Program Counter: 0
Accumulator: 9
General Purpose: 9

Clock Period 6 - (GP) -> Memory

Machine cycle clock Period 7

Finally, the Program Counter is incremented. The entire cycle can now be repeated.



[✓ Wait after each cycle]

12	0	9
13		1
14		7
15	0	1
16		
17		
18		
19		
20		
21		

Address Decoder: 12

The Progam Counter is incremented.

Register	Value
Instruction/Operand	1 · 12
Program Counter	1
Accumulator	9
General Purpose	9

Clock Period 7 - (PC) = (PC) + 1

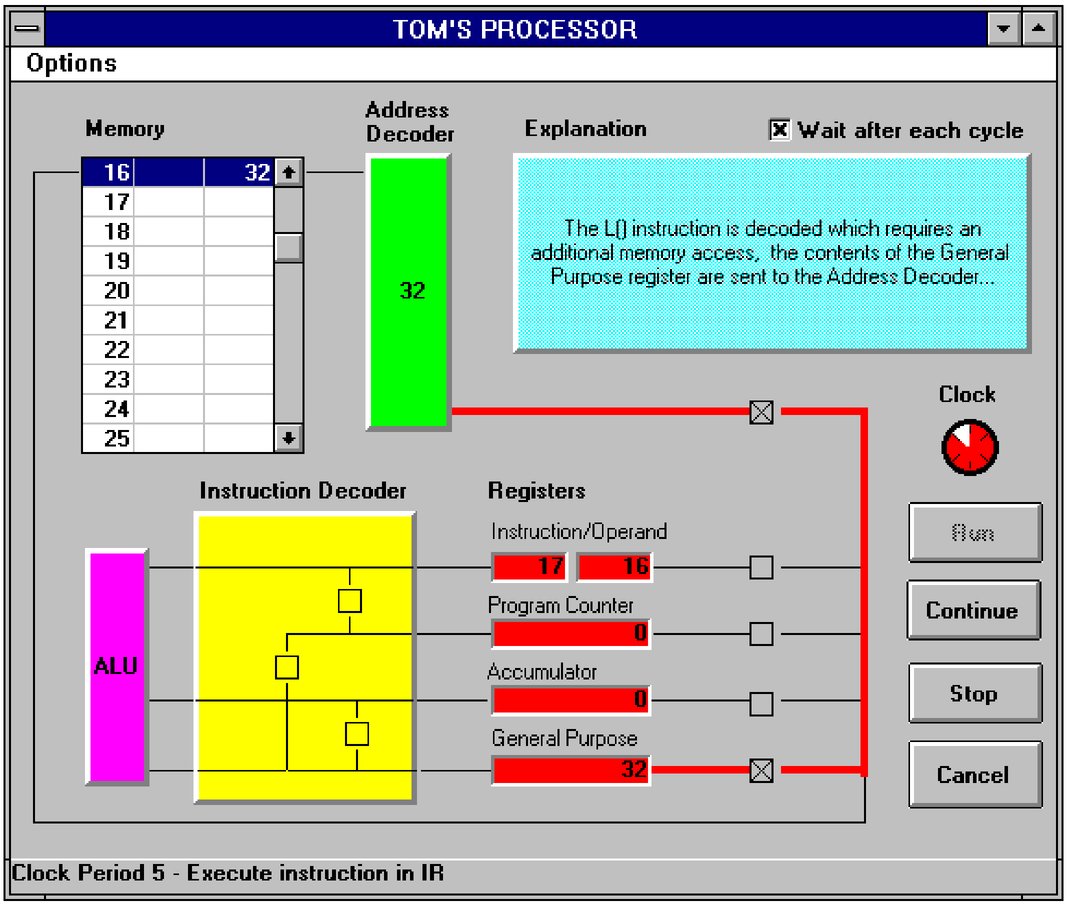
15.5 Indirect instructions

The indirect load and store instructions, L() and S(), cause the processor to access memory a further time within the machine cycle.

Note that when a simple direct instruction such as LDA is executed, the operand, which is a memory address, has already been used to fetch the value it refers to into the General Purpose register. But when an indirect instruction is executed such as L(), the value in the General Purpose Register is the address of the required value. Consider the following short program:

```
 0|L()|  16|
 1|OUT|    |
 2|HLT|    |
16|   |  32|
32|   |  99|
```

This simply copies the value from location 32 into the accumulator and outputs it, but it uses the indirect load instruction, L(), to fetch the value from the address stored in location 16. TOM’s machine operates as normal until it gets to decoding the load indirect instruction. At this point it realises that it needs to dereference the contents of the General Purpose Register. That is, it needs to find the value from memory that the General Purpose Register is addressing. It does this by sending the contents of the General Purpose register to the address decoder:



16	32
17	
18	
19	
20	
21	
22	
23	
24	
25	

Address Decoder: 32

The L() instruction is decoded which requires an additional memory access, the contents of the General Purpose register are sent to the Address Decoder...

Instruction/Operand: 17 16

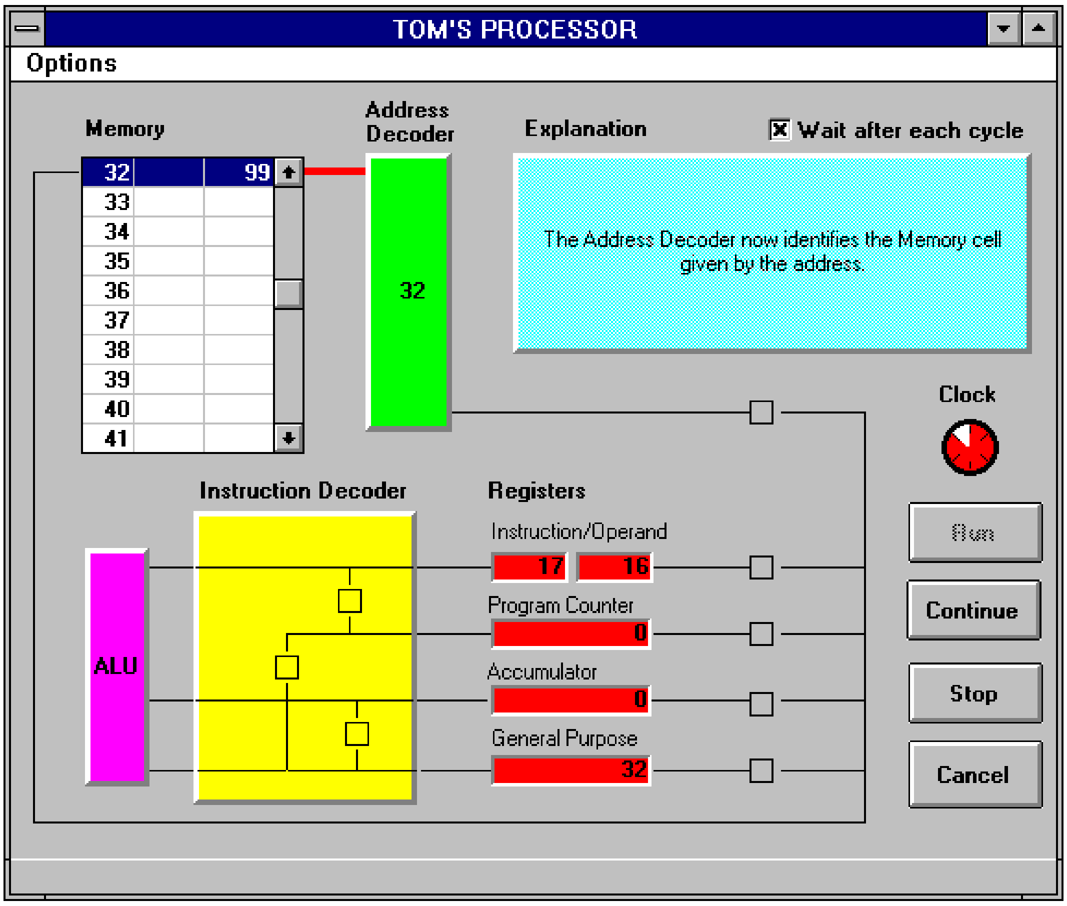
Program Counter: 0

Accumulator: 0

General Purpose: 32

Clock Period 5 - Execute instruction in IR

The memory value at this address is then identified:



The Address Decoder now identifies the Memory cell given by the address.

32	99
33	
34	
35	
36	
37	
38	
39	
40	
41	

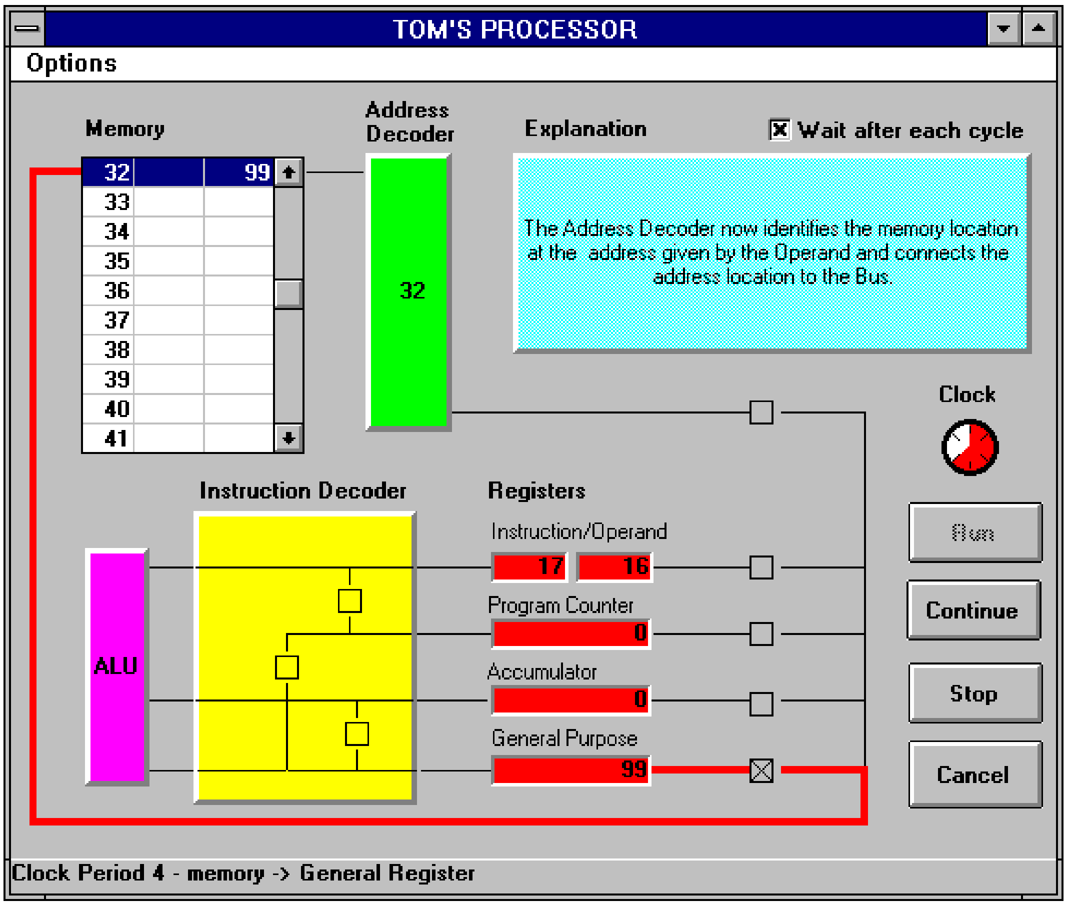
Instruction/Operand 17 16

Program Counter 0

Accumulator 0

General Purpose 32

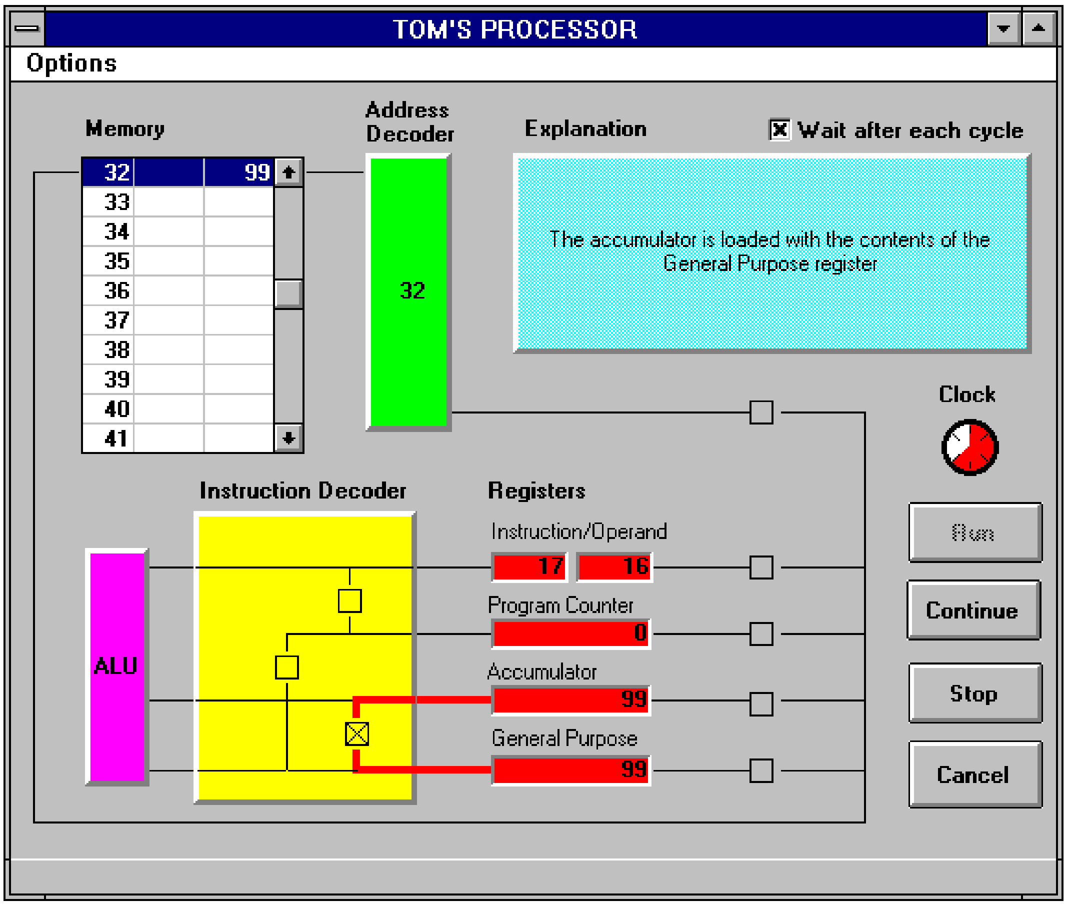
The value from the identified location is then read into the General Purpose Register:



The Address Decoder now identifies the memory location at the address given by the Operand and connects the address location to the Bus.

Clock Period 4 - memory -> General Register

And finally, the value from the General Purpose Register is loaded into the accumulator:



The accumulator is loaded with the contents of the General Purpose register

Notice that during the execution of this extra memory fetch that the Clock is not indicating the true part of the cycle. This is because the indirect instructions are taking control of the machine cycle for a short period in order to insert the extra needed memory access.

This is the normal way for indirect instructions to execute, the normal execution cycle is extended to allow for the extra memory access required. You will see very similar behaviour for the Store Indirect instruction, S().

15.6 Subroutine calls

Using the subroutine instructions, JSB and RTN, also cause a change to the normal execution cycle. The extra store implied by a JSB instruction, in order to remember the return address, and the incrementing of the Stack Pointer requires additional machine cycles.

Normally the Stack Pointer register does not appear on TOM's Processor screen, but if you invoke the machine which uses the subroutine instructions then the Stack Pointer register appears automatically.

Consider the following short program that loads the accumulator with a number and then jumps to a subroutine to output that number, and then returns. On returning a new number is loaded into the accumulator and the program halts.

0	LDA	16
1	JSB	32
2	LDA	17

```
 3|HLT|   |
```

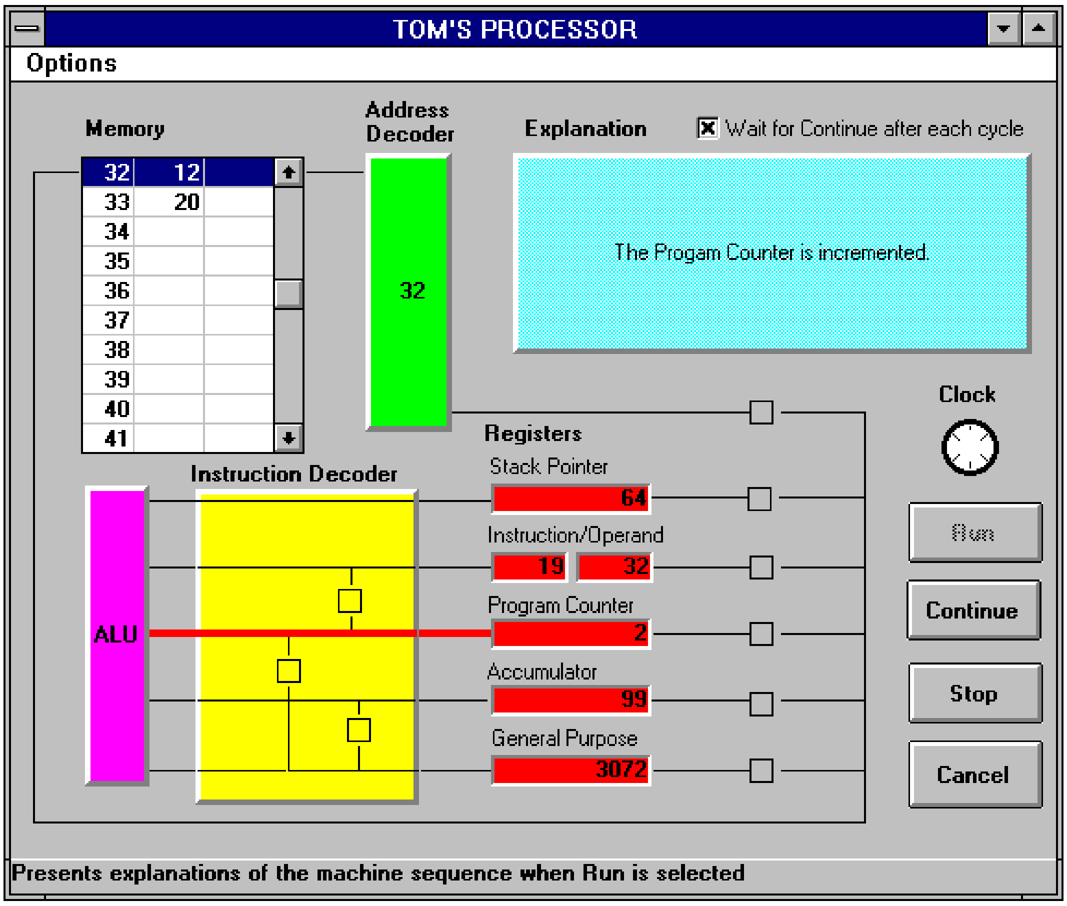
Then we have a couple of data values at addresses 16 and 17:

```
16|   | 99|
17|   | 66|
```

And finally we have the subroutine which consists of just two instructions:

```
32|OUT|   |
33|RTN|  |
```

Execution continues as normal until we get the Jump to Subroutine instruction. This starts executing quite normally, i.e. its operand is fetched, but it then immediately starts doing its own thing. Firstly it needs to increment the program counter because this will be the return address:



[✓ Wait for Continue after each cycle]

Memory	
32	12
33	20
34	
35	
36	
37	
38	
39	
40	
41	

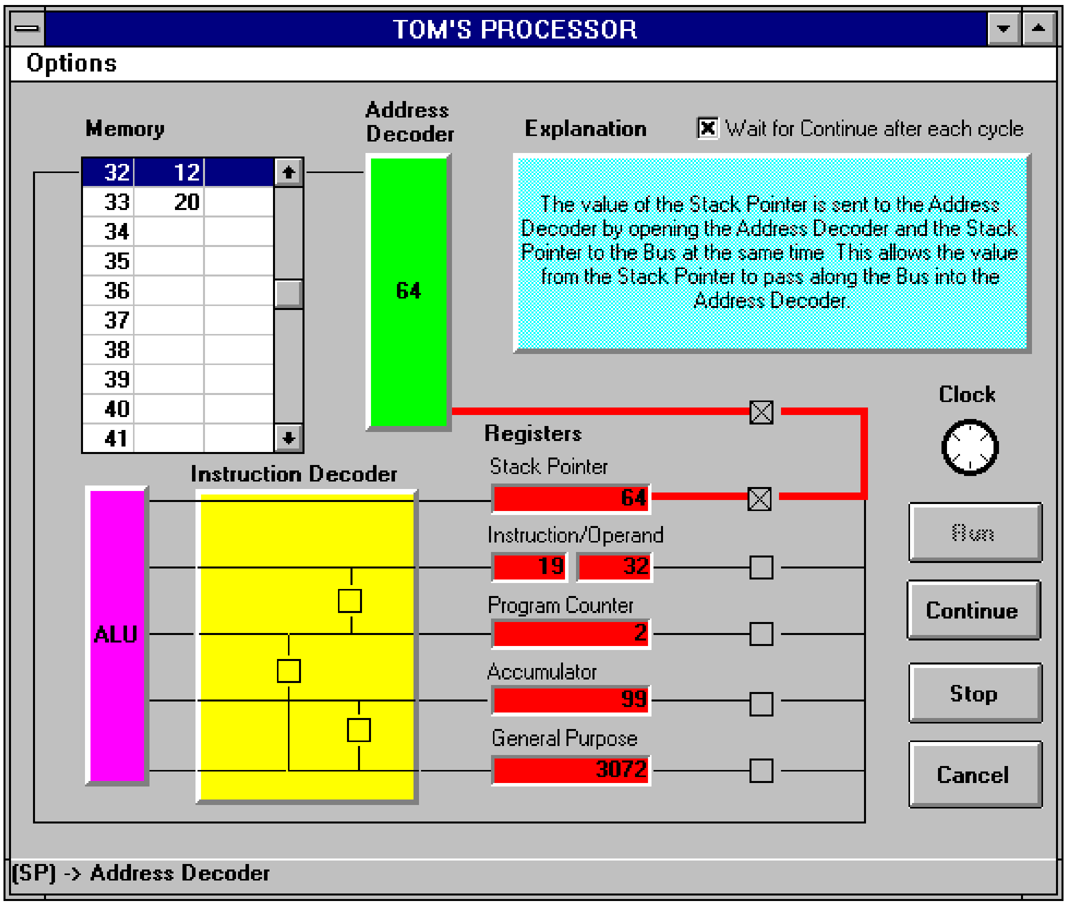
Address Decoder: 32

The Progam Counter is incremented.

Stack Pointer: 64
Instruction/Operand: 19 32
Program Counter: 2
Accumulator: 99
General Purpose: 3072

Presents explanations of the machine sequence when Run is selected

Now it needs to identify the location in memory pointed at by the Stack Pointer, because this is where the return address is going to be stored:



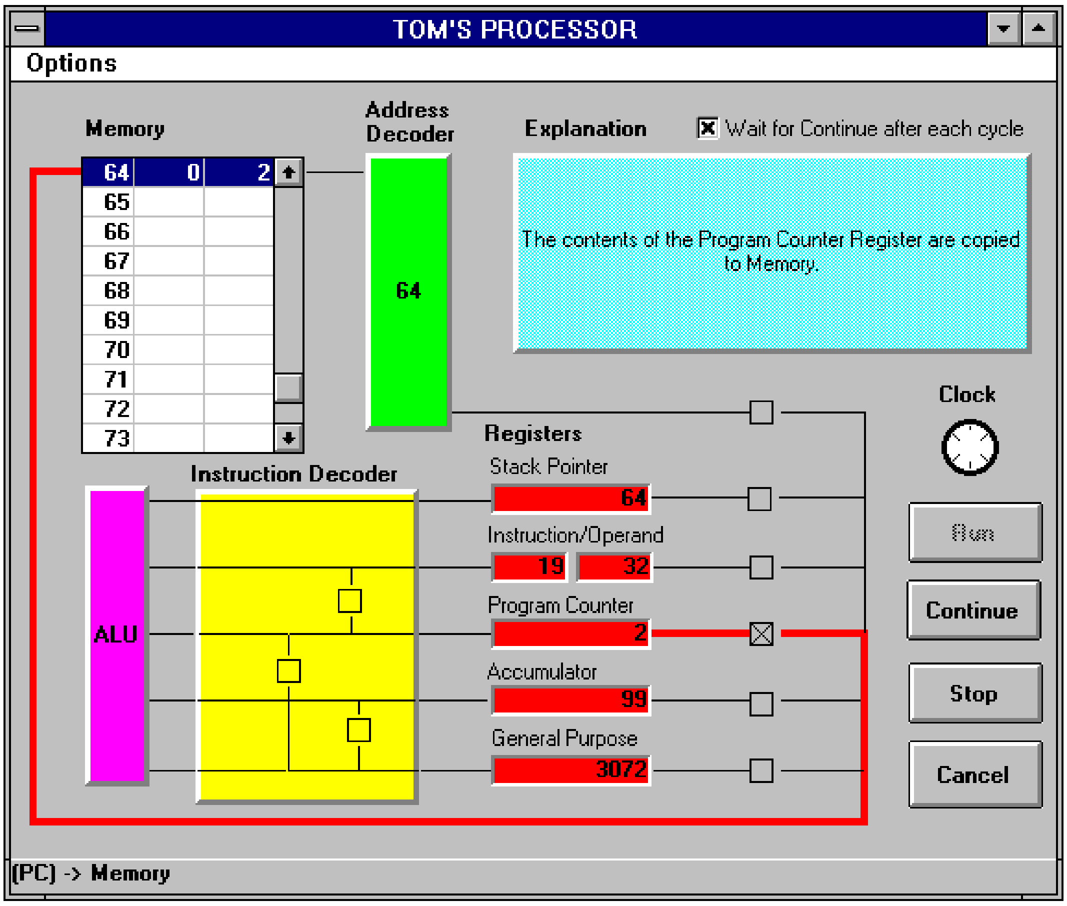
32	12
33	20
34	
35	
36	
37	
38	
39	
40	
41	

Stack Pointer: 64
Instruction/Operand: 19 32
Program Counter: 2
Accumulator: 99
General Purpose: 3072

The value of the Stack Pointer is sent to the Address Decoder by opening the Address Decoder and the Stack Pointer to the Bus at the same time. This allows the value from the Stack Pointer to pass along the Bus into the Address Decoder.

(SP) -> Address Decoder

After this address has been identified by the address decoder the value of the Program Counter is written to memory:



64	0	2
65		
66		
67		
68		
69		
70		
71		
72		
73		

Address Decoder: 64

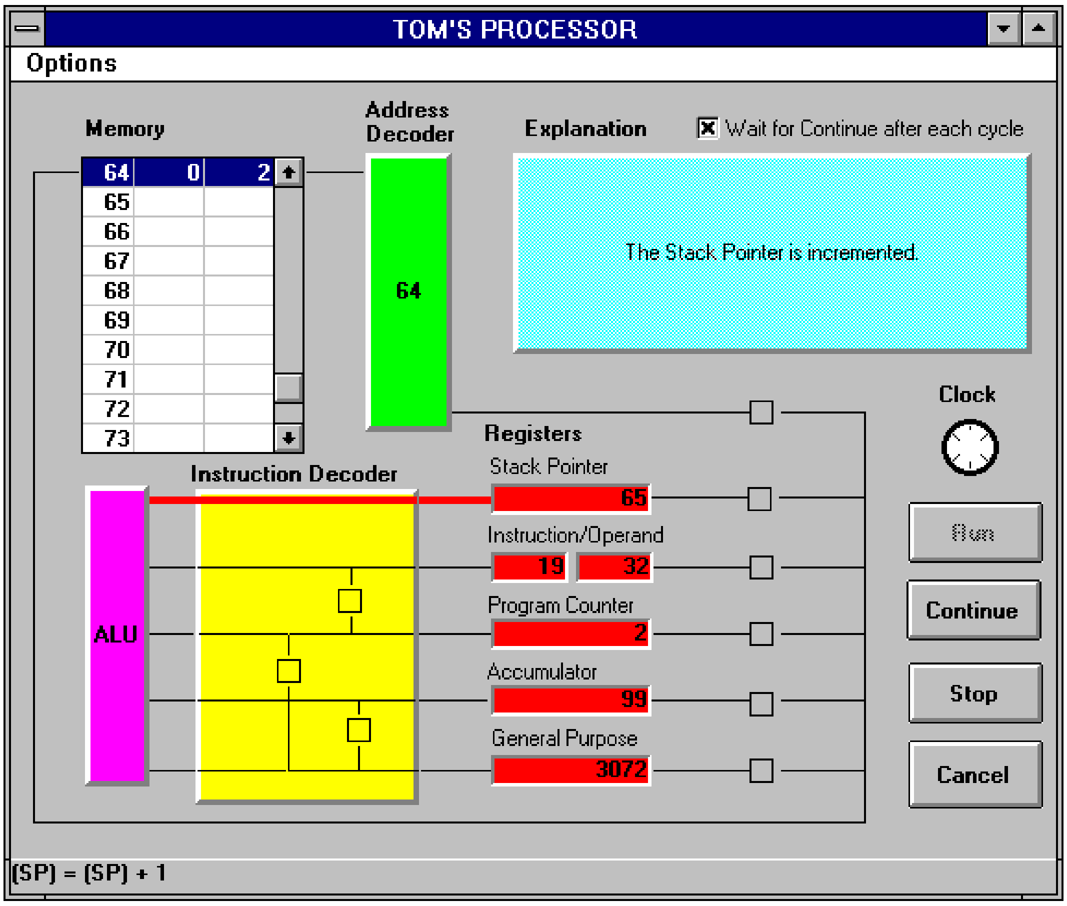
[✓ Wait for Continue after each cycle]

The contents of the Program Counter Register are copied to Memory.

Register	Value
Stack Pointer	64
Instruction/Operand	19 32
Program Counter	2
Accumulator	99
General Purpose	3072

(PC) -> Memory

The Stack Pointer is then incremented:



64	0	2
65		
66		
67		
68		
69		
70		
71		
72		
73		

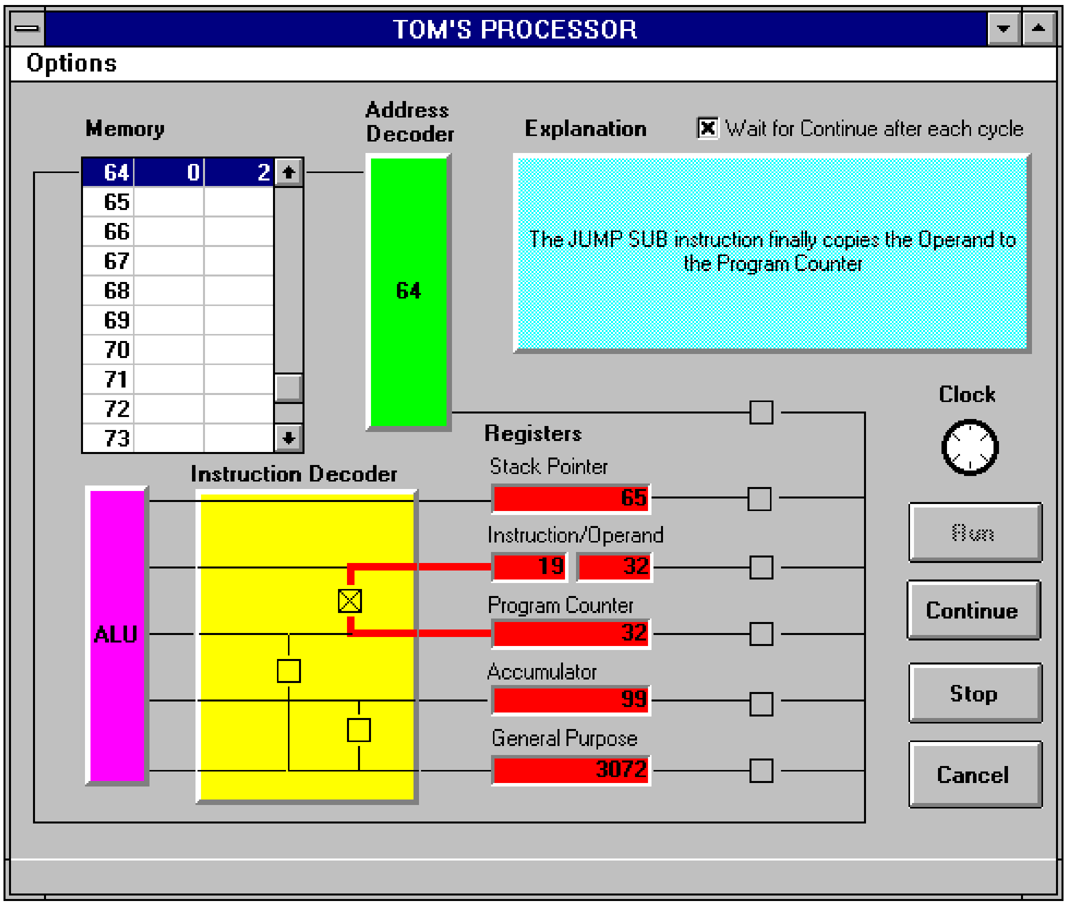
Address Decoder: 64

The Stack Pointer is incremented.

Stack Pointer: 65
Instruction/Operand: 19 32
Program Counter: 2
Accumulator: 99
General Purpose: 3072

(SP) = (SP) + 1

Finally the jump is made to the subroutine by copying the value of the Operand to the Program Counter:



The JUMP SUB instruction finally copies the Operand to the Program Counter

Similar operations which involve the Stack Pointer register are used when the RTN instruction is executed.

15.7 Manually controlling the machine cycle

TOM’s processor can be controlled manually by opening gateways and double clicking on individual components. The following describes the actions that can be taken to control TOM’s processor:

- Open and close gateways. Each of the visible gateways connecting components of the processor can be clicked on to toggle them between open and closed.

- Read a value into Address Decoder. Clicking the Address Decoder causes it to read a value from the BUS. The value available on the BUS is determined by what Gateways are open. This will automatically Identify a memory location.

- Read a value into a register. Clicking on a register will read a value from the BUS into the register. Except, clicking on the Program Counter will increment its value by 1.

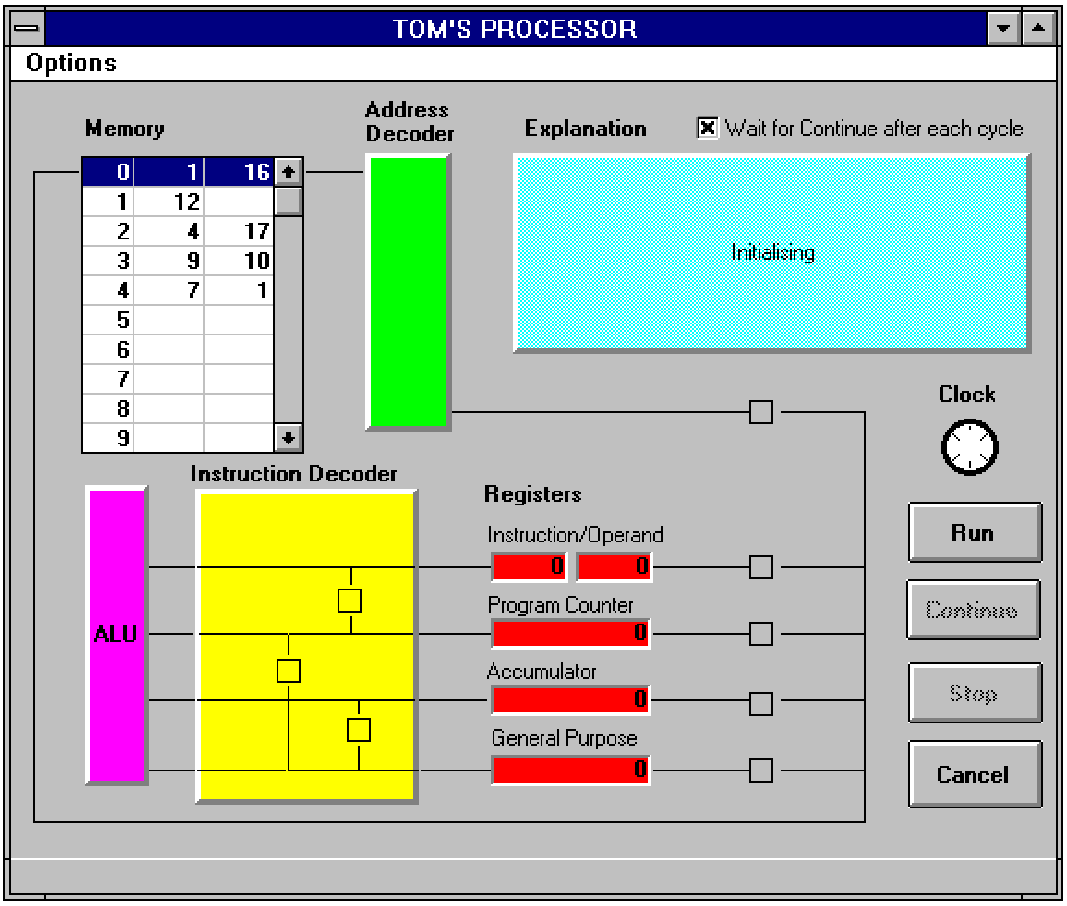
Consider the following short program that implements a simple loop running from 9 through to 0:

```
0│LDA│ 16│
1│OUT│   │
2│AC-│ 17│
3│JM0│ 10│
```

```
 4 |JMP |   1|
10 |HLT |    |
16 |    |   9|
17 |    |   1|
```

To execute this manually:

Display the TOM Processor screen, you will be looking at:



0	1	16
1	12	
2	4	17
3	9	10
4	7	1
5		
6		
7		
8		
9		

Instruction/Operand 0 0
Program Counter 0
Accumulator 0
General Purpose 0

Now:

- Click on the check box that connects the Program Counter to the Bus.
- Click on the check box that connects the Address Decoder to the Bus.
- Click on the Address Decoder.

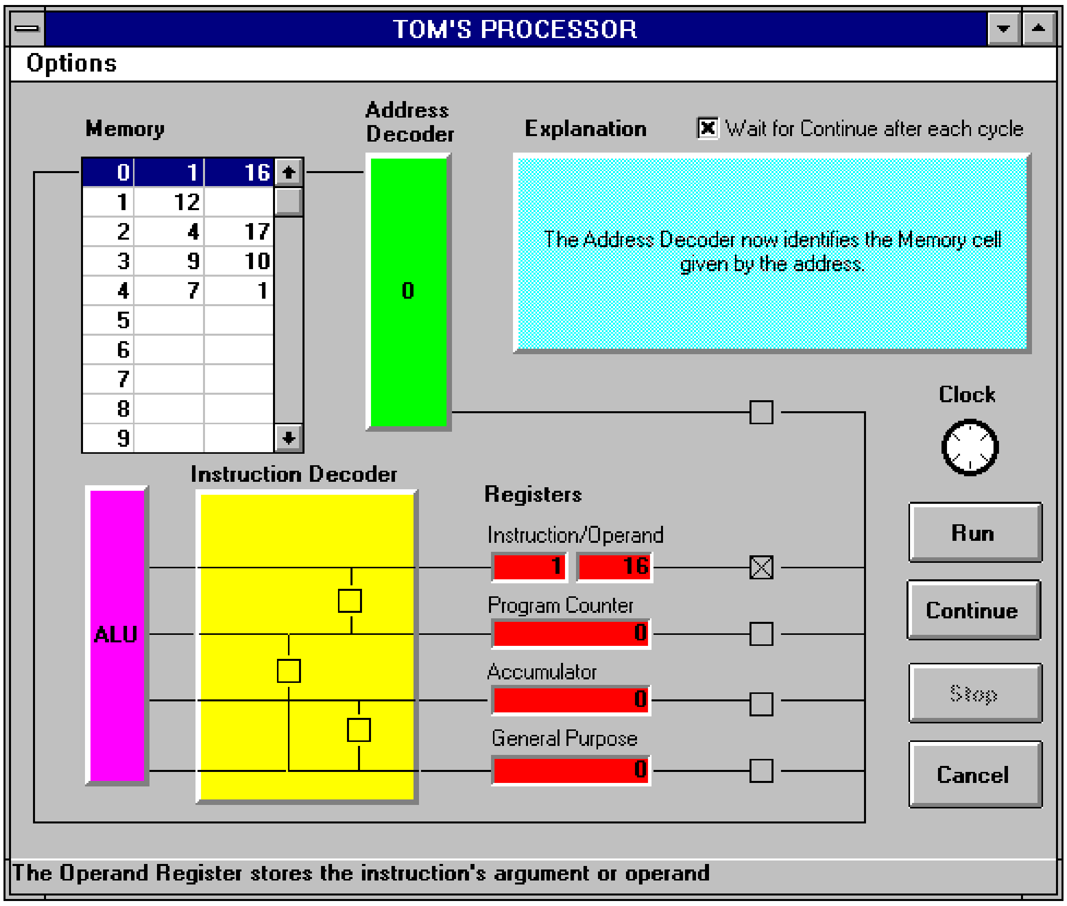
This will identify the memory location corresponding to the Program Counter. The above display already shows this.

Now close the gateways you have just opened. open the Instruction/Operand Registers to the BUS and copy the instruction/operand from memory:

- Click on the check box that connects the Program Counter to the Bus.
- Click on the check box that connects the Address Decoder to the Bus.

- Click on the check box that connects the Instruction/Operand registers to the Bus.
- Click on one of the Instruction/Operand pair of registers.

You should now be looking at:



[✓ Wait for Continue after each cycle]

0	1	16
1	12	
2	4	17
3	9	10
4	7	1
5		
6		
7		
8		
9		

Address Decoder: 0

The Address Decoder now identifies the Memory cell given by the address.

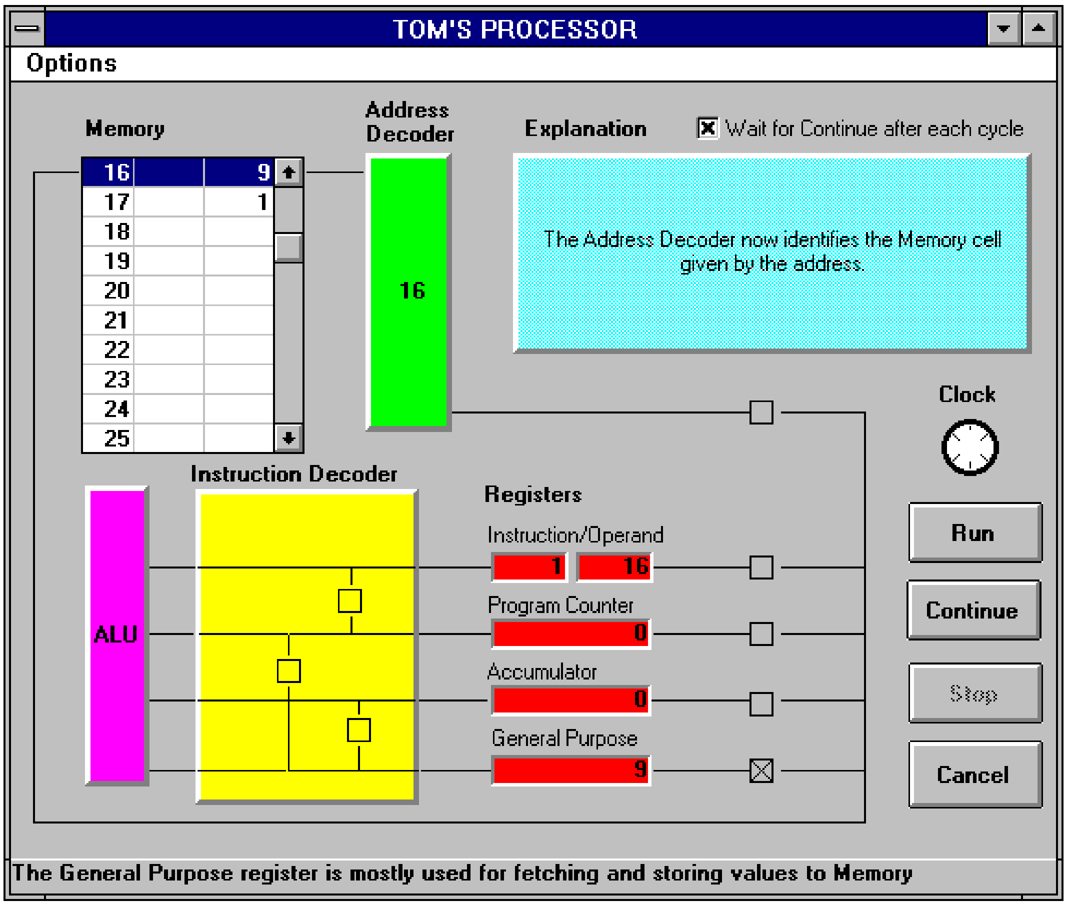
Instruction/Operand: 1 16
Program Counter: 0
Accumulator: 0
General Purpose: 0

The Operand Register stores the instruction's argument or operand

You should be able to see the instruction/operand pair loaded into the instruction/operand registers. Now you need to fetch the operand:

- Open the operand to the Address Decoder by clicking on the check box connecting the Address Decoder.
- Click on the Address Decoder, this will identify the memory location at address 16.
- Close the gateways connecting the Address Decoder to the Operand.
- Open the General Purpose Register to the Bus by clicking on the check box to the right of the General Purpose register.
- Click on the General Purpose register.

Now you should be looking at:



16	9
17	1
18	
19	
20	
21	
22	
23	
24	
25	

[✓] Wait for Continue after each cycle

The Address Decoder now identifies the Memory cell given by the address.

Instruction/Operand: 1 16
Program Counter: 0
Accumulator: 0
General Purpose: 9

The General Purpose register is mostly used for fetching and storing values to Memory

Now it's time to execute the instruction:

Click on the Instruction Decoder, you will see the LDA instruction executed and the value from the General Purpose register will be copied into the accumulator.

This completes the execution of the first instruction. As you can see it is rather long winded, but if you drive the machine manually for a few cycles it will be almost impossible not to understand these fundamental operations of the processor.

In order to execute the next instruction we need to increment the program counter:

- Click on the Program Counter.

Now we can repeat the previous cycle by fetching the instruction, then the operand, then executing the instruction and so on.

15.8 Conclusion

This chapter should have taught you how a machine operates at a very low level. You should have a clear impression of just how mechanical a process it is, involving no intelligence but only a blind following of simple actions over and over again. If there is to be any intelligence in the entire process it has to reside in the programs we write and absolutely not in the machines that execute them!

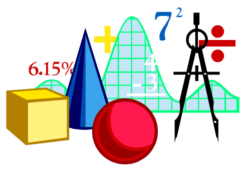
Exercises K

(1) Write a small program using the indirect store operation, S(). Run TOM's Processor screen and note down the extra operation required by the machine to achieve this instruction's purpose.

(2) Write a small program that implements a simple loop as in the text. With TOM's Processor screen displayed, manually cause this program to execute by clicking on the necessary gateways to open and close them and clicking on the various components.

(3) Run the subroutine example given in the chapter and write down the actions caused by the return from subroutine, RTN, instruction.

(4) What normal cycle is missing from the JSB and RTN instructions? Why do you think that this cycle is missed out in these cases?



Appendix A
Complete instruction set

0 HALT

Causes execution of the TOM program to cease. No operand.

1 LOAD ACCUMULATOR

Place a copy of the contents of the memory location given as the operand into the accumulator. The contents of the memory location are unchanged.

2 STORE ACCUMULATOR

Place a copy of the contents of the accumulator into the memory location given as the operand. The contents of the accumulator are unchanged.

3 ADD TO ACCUMULATOR

Place into the accumulator the arithmetic sum of the contents of the memory location given as the operand and the current contents of the accumulator. The contents of the memory location are unchanged.

4 SUBTRACT FROM ACCUMULATOR

Place into the accumulator the result of subtracting the contents of the memory location given as the operand from the current contents of the accumulator. The contents of the memory location are unchanged.

5 MULTIPLY ACCUMULATOR

Place into the accumulator the product of the contents of the memory location given as the operand and the current contents of the accumulator. The contents of the memory location are unchanged.

6 DIVIDE ACCUMULATOR

Divide the current contents of the accumulator by the contents of the memory location given as the operand and place the resulting quotient into the accumulator. The remainder is lost and the original contents of the store location are not affected.

7 JUMP UNCONDITIONALLY

Control is transferred to the memory location whose address is given as the operand. Jump instructions alter the sequence in which a program is executed.

8 JUMP IF ACCUMULATOR NEGATIVE

If the contents of the accumulator is a negative number then control is transferred to the memory location whose address is given as the operand, otherwise program execution continues at the next sequential instruction.

9 JUMP IF ACCUMULATOR IS ZERO

If the contents of the accumulator is zero then control is transferred to the store location whose address given as the operand, otherwise program execution continues at the next sequential instruction.

10 JUMP INDIRECT

Jump unconditionally to the address stored at the address given by the operand. Only the least significant word is considered to form the address.

11 INPUT A NUMBER TO ACCUMULATOR

A dialogue box is displayed into which you can type in a number of up to six digits. If this is a legal number it is placed into the accumulator when you click the OK button. Clicking the Cancel button will cause program execution to terminate. No operand.

12 OUTPUT A NUMBER FROM THE ACCUMULATOR

The contents of the accumulator is output as a number to the output area of TOM's screen. No operand.

13 SET INTERRUPT MASK TO OFF

Setting the interrupt mask to off (0) prevents an input device interrupting TOM's processor. i.e. an input device can raise an interrupt, but the processor will ignore it for as long as the interrupt mask is set to zero.

14 SET INTERRUPT MASK TO ON

Setting the interrupt mask to on (1) allows an input device to interrupt TOM's processor. i.e. an input device can raise an interrupt, and the processor will then process the interrupt whose start address is stored in location 47.

15 READ INTERRUPT INPUT

Reads the value from the interrupting device into the accumulator. For the keyboard input the input value will be the ASCII code for the character pressed.

16 SET RETURN FROM INTERRUPT

Allows execution to proceed from the point where the interrupt was raised. The Program Counter and Accumulator are restored from their saved values and the Interrupt Register is set to 0

17 LOAD INDIRECT

Loads a value into the accumulator from an address stored in the address given as the argument.

18 STORE INDIRECT

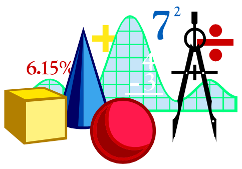
Stores the accumulator's value to the memory location given by the address stored in the address given as the argument.

19 JUMP SUBROUTINE

Puts the program counter + 1 (i.e. the address to return to) into the memory location into the memory location given by the stack pointer, Increments the stack pointer and jumps to the address given as the operand.

20 RETURN FROM SUBROUTINE

Decrements the stack pointer and puts the contents of the memory location given by the stack pointer into the Program Counter.



Appendix B

ASCII Collating Sequence

Many of the ASCII codes are historical, in that they were intended for controlling devices such as magnetic tape and teletypes. They can, for the most part be safely ignored.

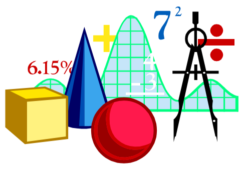
Decimal	Hex	Binary	Char
0	00	0000 0000	NUL
1	01	0000 0001	SOH
2	02	0000 0010	STX
3	03	0000 0011	ETX
4	04	0000 0100	EOT
5	05	0000 0101	ENQ
6	06	0000 0110	ACK
7	07	0000 0111	BEL
8	08	0000 1000	BS
9	09	0000 1001	HT
10	0A	0000 1010	LF
11	0B	0000 1011	VT
12	0C	0000 1100	FF
13	0D	0000 1101	CR
14	0E	0000 1110	SO
15	0F	0000 1111	SI

16	10	0001 0000	DLE
17	11	0001 0001	DC1
18	12	0001 0010	DC2
19	13	0001 0011	DC3
20	14	0001 0100	DC4
21	15	0001 0101	NAK
22	16	0001 0110	SYN
23	17	0001 0111	ETB
24	18	0001 1000	CAN
25	19	0001 1001	EM
26	1A	0001 1010	SUB
27	1B	0001 1011	ESC
28	1C	0001 1100	FS
29	1D	0001 1101	GS
30	1E	0001 1110	RS
31	1F	0001 1111	US
32	20	0010 0000	space
33	21	0010 0001	!
34	22	0010 0010	“
35	23	0010 0011	#
36	24	0010 0100	$
37	25	0010 0101	%
38	26	0010 0110	&
39	27	0010 0111	‘
40	28	0010 1000	(
41	29	0010 1001	)
42	2A	0010 1010	*
43	2B	0010 1011	+
44	2C	0010 1100	,

45	2D	0010 1101	-
46	2E	0010 1110	.
47	2F	0010 1111	/
48	30	0011 0000	0
49	31	0011 0001	1
50	32	0011 0010	2
51	33	0011 0011	3
52	34	0011 0100	4
53	35	0011 0101	5
54	36	0011 0110	6
55	37	0011 0111	7
56	38	0011 1000	8
57	39	0011 1001	9
58	3A	0011 1010	:
59	3B	0011 1011	;
60	3C	0011 1100	<
61	3D	0011 1101	=
62	3E	0011 1110	>
63	3F	0011 1111	?
64	40	0100 0000	@
65	41	0100 0001	A
66	42	0100 0010	B
67	43	0100 0011	C
68	44	0100 0100	D
69	45	0100 0101	E
70	46	0100 0110	F
71	47	0100 0111	G
72	48	0100 1000	H
73	49	0100 1001	I

74	4A	0100 1010	J
75	4B	0100 1011	K
76	4C	0100 1100	L
77	4D	0100 1101	M
78	4E	0100 1110	N
79	4F	0100 1111	O
80	50	0101 0000	P
81	51	0101 0001	Q
82	52	0101 0010	R
83	53	0101 0011	S
84	54	0101 0100	T
85	55	0101 0101	U
86	56	0101 0110	V
87	57	0101 0111	W
88	58	0101 1000	X
89	59	0101 1001	Y
90	5A	0101 1010	Z
91	5B	0101 1011	[
92	5C	0101 1100	\
93	5D	0101 1101	]
94	5E	0101 1110	^
95	5F	0101 1111	_
96	60	0110 0000	`
97	61	0110 0001	a
98	62	0110 0010	b
99	63	0110 0011	c
100	64	0110 0100	d
101	65	0110 0101	e
102	66	0110 0110	f

103	67	0110 0111	g
104	68	0110 1000	h
105	69	0110 1001	i
106	6A	0110 1010	j
107	6B	0110 1011	k
108	6C	0110 1100	l
109	6D	0110 1101	m
110	6E	0110 1110	n
111	6F	0110 1111	o
112	70	0111 0000	p
113	71	0111 0001	q
114	72	0111 0010	r
115	73	0111 0011	s
116	74	0111 0100	t
117	75	0111 0101	u
118	76	0111 0110	v
119	77	0111 0111	w
120	78	0111 1000	x
121	79	0111 1001	y
122	7A	0111 1010	z
123	7B	0111 1011	{
124	7C	0111 1100	|
125	7D	0111 1101	}
126	7E	0111 1110	~
127	7F	0111 1111	DEL



Appendix C
Glossary

Accumulator

An internal register used for various purposes such as intermediate results of arithmetic statements.

Address

A unique identifier for a memory location.

Address decoder

A logical device which given an address outputs a logical 1 on a particular output line that indicates the given address.

Arithmetic logic unit

The logical device that can perform arithmetic on the contents of the machine's internal registers.

Arithmetic overflow

Occurs when an attempt is made to store or calculate a value that cannot be stored in the register. Indicates overflow from the highest bit.

ASCII

ASCII stands for American Standard Code for Information Interchange and is a simple collating sequence that maps numbers to characters

Assembler

A program that can convert the mnemonic form of a program to binary machine instructions.

Binary

Number system with base 2.

Bit

Smallest amount of storage possible, enough to store one of two states, a 1 or a 0.

Bus

The bus connects the various components, such as the registers and the memory , of the computer together so that information can be passed between them.

Binary encoded decimal

Method of mapping each decimal digit to a four bit binary number. Wasteful on memory but very simple.

Byte

Eight contiguous bits.

Clock

Controls the sequence of opening and closing the gateways onto the bus. This allows information to pass between the computer's components at specific points in the execution cycle. The clock is a vital component of TOM's machine.

Controlled flip-flop

A flip-flip with a control line which determines when the flip-flop should listen and be capable of changing state.

Decimal

The number system based on powers of ten,

Flip-flop

An electronic device that can be set in one of two states.

Full adder

A logical device that implements addition on two input bits and a carry in bit, resulting in a single output bit and a carry out bit. Can be implemented by using two half adders.

Gateway

A logical device that can connect and disconnect the components of TOM's machine to the bus.

Half adder

A logical device that implements addition on two single bits with a single result bit and a carry out bit.

Hexadecimal

Number system based on 16. This is a very useful number system because it is so closely allied to the binary system, the number 16 being a power of 2.

Instruction decoder

Interprets an instruction held in the instruction register into a specific action on the registers . The instruction decoder is a component of TOM's machine.

Instruction register

Holds the current instruction consisting of the operator and operand.

Integer

A whole number, either positive or negative.

Interrupt

An external device, such as a keyboard, raises an interrupt when it requires attention from the processor in order to have its input processed.

Interrupt mask

A bit that is set to indicate that interrupts should be taken notice of.

Machine code

The internal numeric (actually binary) form of a program. The form that can be directly executed by the processor.

Mnemonic

A short character sequence which stands for an instruction, such as JMP standing for the jump instruction.

Opcode

A numeric code given to a particular operator.

Operand

The argument to an operator, i.e. it is what the operator is to operate on.

Pointer

An address of a memory location.

Program counter

An internal register that stores the address of the next instruction to be executed.

Register

A processors internal memory consists of a set of registers. Used for storing intermediate values, program counter, and various other uses.

Stack

An area of memory used for storing values, usually so they are retrieved in the reverse order than they are stored.

Stack pointer

An index (pointer) to the next free location on the stack.

Status register

An internal register that stores the status values for various computer functions, such as arithmetic overflow and interrupts.

Subroutine

A distinct part of a program that can be called in order to execute a specific function and will then return control to whatever called it.

Transistor

A subminiature electronic device which can be used as a very fast switch.

Index