GHIJ

UNIT NO:
D75P 34R

UNIT TITLE:
Computer Architecture 1

Session 2004 - 2005

Outcome 3

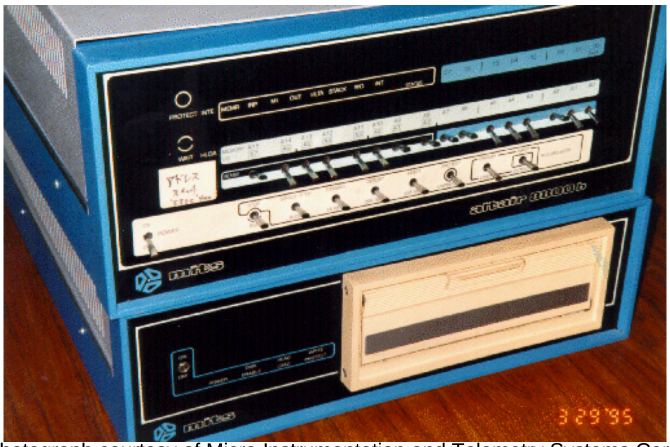
Demonstrate an understanding of the principles of Central Processor Unit (CPU) operation

1234567
Awarded for excellence

Computing (TN3)
Engineering, Computing and Business Studies

Week 16. A Short History of Software.

We already saw that computers can only understand one real language - binary values. In the bad old days before programming languages were invented, this involved writing entire programs in binary. Very early model computers involved practically re-wiring the insides every time a bit pattern had to be changed, and the term "software engineer" meant exactly that - only a small step removed from an electrician with a toolbox! The first "home" computer to hit the shelves in the late 70's, the Altair 8800, also had to be programmed in binary - users input data through a series of switches, and read off the output from a bank of lightbulbs.



Photograph courtesy of Micro Instrumentation and Telemetry Systems Corp.

The coding of programs in binary, or "machine language", was tedious and error-prone and is almost never done nowadays. A better solution was the introduction of "Assembly Language" in the 1940's. This took a series of symbols written by the programmer and assembled them into the final machine language to be used by the processor. There was usually a one-to-one relationship between the assembly language statements and an instruction to the processor. For example,

```
add r0, r2, r3
```

meant "get the contents of registers 2 and 3, add them together and put the result in register 0".

This is a lot more human-friendly than

```
0010101111001010101000001111000011110110011000001111101011001010101000
101011110101010101010101100110100011111100000110011101
```

although the downside is, of course, that the machine has to work harder to understand what it is supposed to do.

Improvements to this meant that you could have labels on code segments, thus making it possible to call procedures. Further improvements meant you could have *directives*, and therefore define data structures like strings and arrays.

Eventually Assembly Language developed to the point that one instruction at assembly level could correspond to many machine-code commands, finally removing the one-to-one relationship between the assembly language statement and processor instruction.

A Timeline of Software Development…

- Prior to 1960, computers are programmed in binary code!

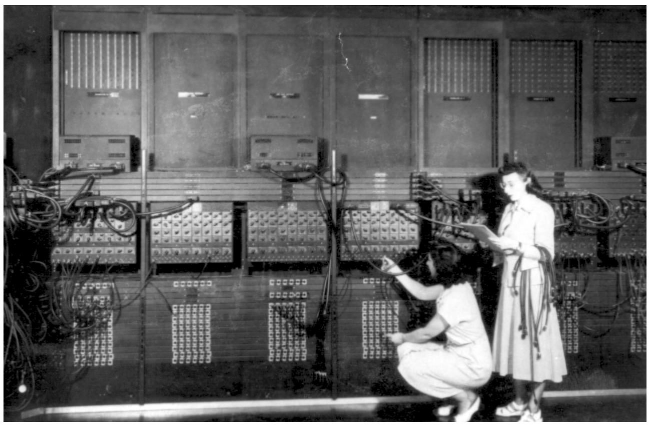


Photo courtesy US Army. The ENIAC had a system of plugs, similar to an old-fashioned telephone switchboard, for connecting the relevant circuits to form the bit patterns.

- Several computer manufacturers and mathematicians from the Pentagon develop Common Business Oriented Language - COBOL (1960)
- The first computer language for writing Artificial Intelligence Programs - LISP - is invented, also in 1960.
- ALGOL is the first high-level language with a readable, structured, and systematically defined syntax and designed for mathematical algorithms - again in 1960.
- The ASCII code is formalised - 1963.
- BASIC is invented at Dartsmouth College for use as a teaching tool (1964).
- Seymour Papert develops LOGO, a teaching language for children (1967).
- Prof. Niklaus Wirth at ETH Zürich defines Pascal in 1970. It is named after 18th Century French mathematician Blaise Pascal, who designed a revolutionary mechanical counting machine based on place values of numbers.
- Bill Gates and Paul Allen write a new version of BASIC to run on the MITS Altair 8800 (1974). This new version later becomes known as Microsoft Basic - and the rest, as they say, is history!
- The US Department of Defence initiates an international competition to design a language primarily for embedded systems. The winning design is chosen in 1979 and called Ada, in honour of Ada Augusta Byron, the Countess of Lovelace and daughter of Lord Byron, who is generally accepted as the worlds' first computer programmer.
- MS-DOS is introduced in 1981.
- The Smalltalk language introduces the graphical user interface (GUI) as we know it today (1983).

- Microsoft launches Visual Basic in 1991.

- Java is launched by Sun Microsystems in 1995. It is designed to be robust, easily learnable and cross-platform. Microsoft responds by introducing C# (pronounced C-sharp) in 2000.

So why do people still program in assembler?

- It's the best way to grasp how a processor works at lower levels.
- Special functions may have to be written in assembly, particularly if there are timing or memory problems (and you need to know exactly how many machine cycles, or how much memory, the instruction will use).
- It provides a very fast and direct link between the CPU and any peripherals used. A common use of assembly code is for writing device drivers.
- It provides a high level of control over the hardware. Graphics-based programs such as games and screensavers will often have large embedded sections in assembly.
- There are a large number of programs around which were originally written in assembly, and these need to be maintained and updated (and these can be found still in use by some HUGE multinational enterprises!)
- In order to convert a high-level program into an executable, you need a *compiler*, and at some point the compiler has to be written in assembly.
- Hardware manufacturers who design and build processors need to be able to understand assembly language instruction sets.

The assembly language instruction set defines the interface between the hardware and software, and so underpins every single function of that system.

An example RISC program using a MIPS instruction set.

```
##
## hello.a prints out the ubiquitous "hello world!"
##
## a0 - points to the string of characters
##
###################################################
#                 text segment                    #
###################################################

     .text
     .globl  __start

__start:             # program execution starts here
     la $a0, str     # put address of the string into a0
     li $v0, 4       # sets up to make it print to screen
     syscall         # system call to make it print

     li $v0, 10      #
     syscall         # byeeeeeee......

###################################################
#           data segment                          #
###################################################

     .data

str: .asciiz  "hello world!\n"

##
## end of file
```

Note to the above - Information in the data segment does not contain instructions that are executed. This is data that is used during the execution of the program - like variable contents in Pascal, for example. The hash sign # is the start of a comment line and everything immediately afterwards is ignored by the compiler. Because assembly language is further removed from English than, say, Pascal, it is important to include lots of comments to make it clear to the human reader, what the instruction does.

Assembly Language and instruction Sets.

We already saw that different processor models used different **instruction sets**. The instruction set is the list of "commands" that the processor can understand and respond to; this is also very useful for programmers, as they can use these instructions to program the computer without having to load in all the data in pure binary. The computer can then interpret and run the instructions via an **assembler,** another piece of software that does the actual translation.

Programming at this level is known as **assembly or low-level programming,** and this produces the fastest and most powerful type of program, because it is nearer to the native language of the computer than a high-level program written in, say, Pascal or C++. Most games programs will have at least some sections hand-coded at assembler level; another major use is for writing device drivers.

The list of operations that most processors can respond to will vary, but there are nine general groups of instructions that all processors can handle.

1. **Load** and **Move** instructions. These move data into specific registers.
2. **Arithmetic** operators, such as ADD, SUB, MUL and DIV.
3. **Store** instructions for moving data from registers back into memory.
4. **Logical operators** i.e. the Logic Gates AND, OR, XOR etc.
5. **Shift** and **Rotate** operations, which move bits left or right in a register.
6. **Compare** and **test** operations, used for comparing values for equality.
7. **Jump** or **branch** operators, which transfer control to another point in the program. These can be unconditional, or only fired in the event of some tested condition being true or false.
8. **Push** and **pop** operations, used for storing data on the stack, or taking it back off.
9. **Call** operations, which trigger subroutines or procedures as required.

A line of assembly source code will consist of the following -

mnemonic	operands	labels	comments	whitespace
The instruction itself	Data being operated on, or information needed by the instruction in order to be carried out	Destinations for jumps	Information ignored by the computer but of great value to human readers!	Spaces to make the program more readable to the human eye

Example of assembly code. The user has input a character (which is taken into the AL register by default) which we are checking to see is an upper case letter. If not, an error will be displayed. If it is, we will convert it to a lower case letter.

```
mov         bl, al                  ;save char in al to bl
cmp         bl,'A'                  ;see if it's uppercase A
jl          invalid                 ;if it is < A jump to
                                    ;invalid routine
cmp         bl, 'Z'                 ;compare it to a Z
jg          invalid                 ;if it is > Z jump to
                                    invalid routine
                                    ;if we have reached here
                                    ;it must be a valid
                                    letter
add         bl, 32d                 ;converts it to lower
                                    case
```

; and so on… further down there will be the "invalid" subroutine

```
invalid:
mov         dx, offset errormessage  ;points to a previously
                                     written ;error message
mov         ah, 9                    ;sets up a software
                                     interrupt
int         21h                      ;display message on a
                                     DOS ;screen
```

Compare this example, which was written to run on an Intel-based processor, to the example on page 5, which was written for a StrongArm, as used in the Nokia below.

One method of making assembly programming easier was to create complex instructions that would perform common tasks, rather than have the programmer write several small instructions every time a specific task was required. This had the dual benefit of both making the program smaller and easier to write; however it made the program run much more slowly, because the processor had more decoding to do before it could figure out what was actually required. While a relatively simple instruction maybe took place over 2 or 3 clock cycles, some of the more complicated ones could take 20, 30 or more clock cycles to execute.

This Nokia Communicator is built around a StrongArm RISC chip, which was in turn developed from the same source as the old Acorn Archimedes!

Meanwhile, ongoing technological development led to the introduction of **high level** languages such as Fortran, C, Pascal and later C++, Java etc. These languages look more like English than any machine code, but of course are much further away from the native language of the machine. One high-level instruction, therefore, may have to be translated into may low level -ones; in order to do the translation, the high-level code must be run through a **compiler.**

By the 1980s most programs were being written in high-level code, and many of the embedded instructions in CPUs were not in fact being used at all! This excess baggage was actually reducing the processing power available to the end user, so the next logical step was to design a chip with a much smaller instruction set. This in turn reduced the number of transistors and gates required, and therefore the size and cost of the processor itself; or alternatively, the extra space could be used for increasing the size of the cache. Because of the reduced number of transistors, they were also less prone to overheating!

Week 17. Different Instruction Sets.

The Assembly Code instruction set, or list of the commands that the processor can understand, will vary depending on the particular type and model of processor involved. The usual Complex instruction set may have anything up to 300 commands; a Reduced instruction set will typically have no more than about 100. Both types consist of *mnemonics*, short words which sound like their actions - for example `LOD` loads data from a specified address, `ADD` adds two operands and so on. For learning purposes, we will use the following simplified instruction set, which is not based on any particular processor but contains typical commands.

```
LDA Load Accumulator
MVA Move contents of specified register to accumulator
STA Store content of accumulator
CLA Clear contents of accumulator
ADD Add contents of data bus to accumulator
INC increments the value in the register by 1
SUB Subtract contents of data bus from accumulator
IN get input from device nn
OUT output data to device nn
HLT stop program execution
BRLT Branch to nn if contents of accumulator is less than 0
BRLE Branch to nn if accumulator is = or < 0
BRGT Branch to nn if accumulator is > 0
BRGE Branch to nn if accumulator > = 0
BRNE Branch to nn if accumulator != 0
BREQ Branch to nn if accumulator = 0 exactly
AND Ands the data in the specified location with whatever is in the
accumulator
OR ors the bits on the data bus with whatever is in the accumulator
NOT reverses all the bit values
CPAX Copy accumulator contents to the X register
CPXA Copy X register to the accumulator
```

A sample program, to get two numbers from the user and add them, would look like this:-

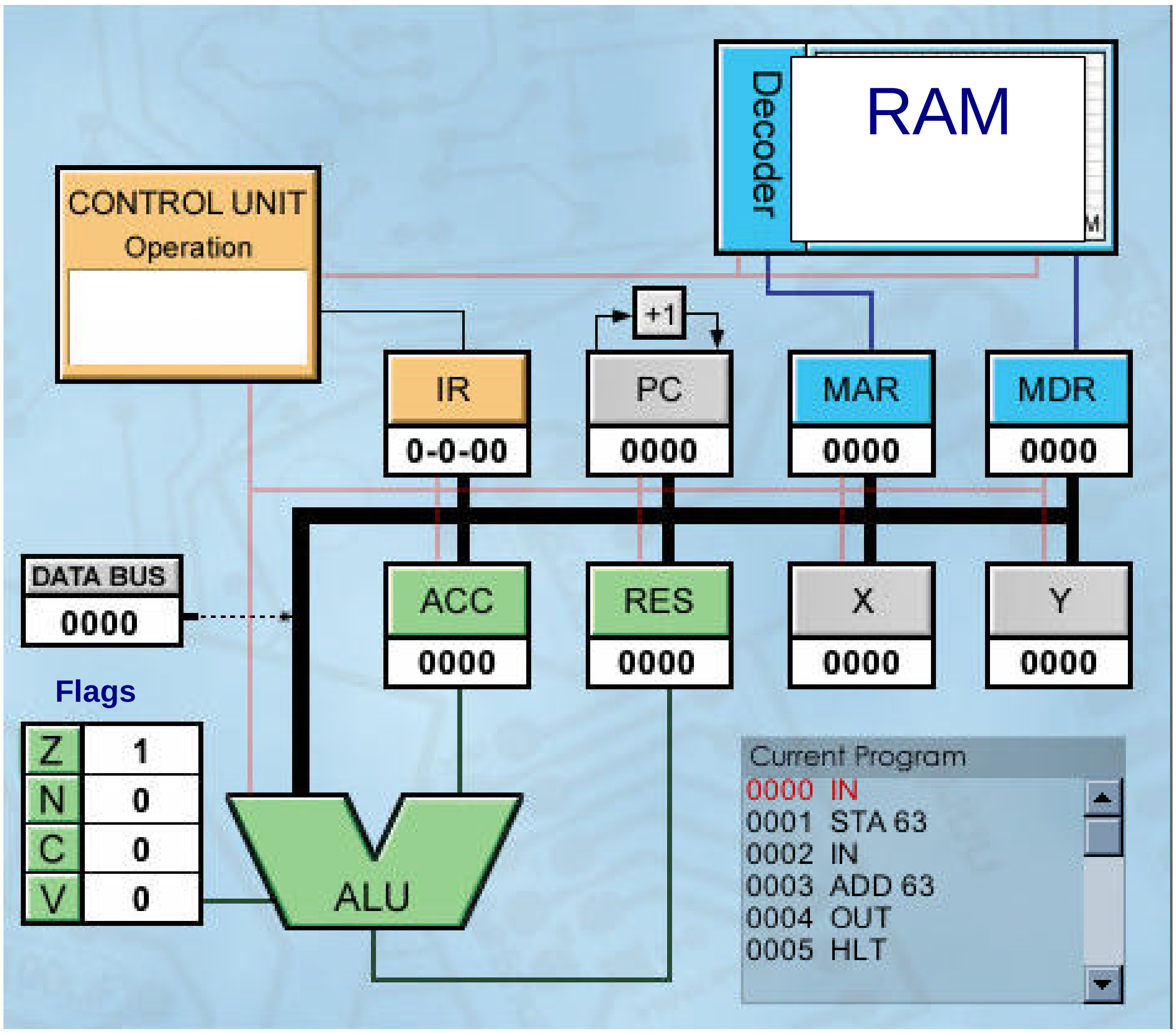
```
0000 IN    # get input from user and put it in the accumulator
0001 STA 63 # store the contents of the accumulator in RAM
           # address 63
0002 IN    # same as line 0000
0003 ADD 63     # retrieve the contents of cell 63 and add it to the
           # value in the accumulator
0004 OUT   # send the accumulator contents to an output device
0005 HLT   # stop the program
```

These instructions would be loaded into the relevant RAM locations as follows :-

0000 50-00	0001 20-63	0002 50-00	0003 30-63	0004 60-00
0005 70-00	0006	0007	0008	0009
000A	000B	000C	000D	000E
000F	And so on….			0063

Note that the mnemonic commands have been replaced by numbers, called *opcodes*. Each command will have an associated opcode and these will be described as we encounter them. Now we will see how the program is executed.

The Fetch – Decode - Execute Cycle.



```
Current Program
0000  IN
0001  STA 63
0002  IN
0003  ADD 63
0004  OUT
0005  HLT
```

FETCH.

1. The contents of the Program Counter (PC) are copied to the Memory Address Register (MAR). This sets up the RAM location (in this case, #0000) ready for the next instruction.
2. The PC is incremented by 1, ready for the next instruction. (PC = 0001, MAR = 0000).
3. The Control Unit (CU) issues a READ request. The memory location specified in the MAR (#0000) is located in the RAM. The contents of that "cell" are copied to the Memory Data Register (MDR). For example, if the contents of #0000 were 5000, then the contents of the MDR would also become 5000

EXECUTE.

The instruction in the IR (in this case 5-0-00) is decoded and executed. (Opcode 50 means "Get input from a user device"). The first part is the Opcode and the lower 8 bits give the address portion.

The CU extracts the instruction position and decodes the operation. The rest of the execute phase depends on what the instruction was.

Example:- Opcode 50 means "get input from the user and store the value in the Accumulator". The Accumulator is a register used by the ALU to perform Maths and Logic.

The Fetch Phase.

The Fetch part of each instruction never varies - it is exactly the same for each line of code.
1. The contents of the Program Counter are copied to the MAR.
2. The Program Counter increments itself by 1.
3. The Control Unit issues a READ request, and the RAM location read will be the one that the MAR is pointing to. A signal is sent along the Address Bus to the relevant cell.
4. The contents of that address are dumped on the Data Bus and brought back to the MDR.
5. The contents are then copied to the IR, ready to be decoded - i.e. the CU reads the instruction and figures out what to do with it!

The Decode Phase. (Note: in some books this is shown as being a part of the Execute Phase rather than a separate step. Either method is acceptable.)

1. The instruction in the IR is decoded and executed. All instructions are composed of two parts - the *opcode* and the *operand*, the opcode being the instruction itself and the operand(s) being any data required by the opcode in order to work. (These could be locations, reference pointers, actual data values, another part of the program, anything!).

If we were to look at the instruction 50-00, for example, this would be interpreted as "get input from a user device and put it in the accumulator". Typically an opcode is an 8- or 16- bit binary value, sitting in the top half of the word. The operand values are stored in the lower end. In our example, the opcode 50 refers to "get input" and the 00 gives the location (in this case the accumulator).

The Execute Phase.

This will, of course, vary depending on what the instruction was! Let's look at the STA 63 instruction, as this involves writing data back to the RAM.

1. The instruction STA 63 translates to 20-63, so the CU reads the 20 as "get data from the accumulator and store it in some specified RAM location", and that location will, of course, be 63.
2. The address to be written to (i.e. 63) gets transferred to the MAR.
3. The data in the accumulator is transferred to the MDR. Remember that EVERYTHING entering and leaving the CPU can only come or go through this "gate".
4. The data value is transferred up the Data Bus and written back to the RAM.

Now we are ready for the next instruction!

Exercises.

Study the attached sample programs and complete the grids provided, showing the values in each register at any given point.

Contents of RAM for Program 1	
20	LDA 24
21	ADD 25
22	STA 26
23	HLT
24	10
25	5
26	100
27	0
28	50
29	2
2A	4

Contents of RAM for Program 2	
20	LDA 2C
21	NEG (B1-00)
22	INC
23	BREQ 2A (85-2A)
24	STA 2C
25	CLA (C1-00)
26	ADD 2D
27	STA 2E
28	LDA 2C
29	BRA 22 (80-22)
2A	HLT
2B	2
2C	3
2D	4
2E	

An example CISC program using an Intel-8086 instruction set.

```
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;file : dot.asm
;written by Chrissie Nyssen 18/05/01
;draws a red dot in middle of screen.  This is how many graphics
;and games programs begin life!
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.model small

.stack 100h

.code

start:
   mov ax,13                ;mode is 13h
   int 10h                  ;call bios video service

   mov ah,0ch               ;function 0ch
   mov al,4                 ;colour 4 is red
   mov cx,160               ;x position is 160
   mov dx,100               ;y position is 100
   int 10h                  ;BIOS interrupt

   inc dx                   ;plot pixel downwards
   int 10h                  ;BIOS interrupt
   inc cx                   ;plot pixel to right
   int 10h                  ;BIOS interrupt
   dec dx                   ;plot pixel up
   int 10h                  ;call BIOS service

   xor ax,ax                ;function 00h of 16h gets a key
   int 16h                  ;call BIOS keyboard interrupt
   mov ax,3                 ;mode = 3
   int 10h                  ;BIOS video interrupt

   mov ax, 4c00h            ;exit to DOS
   int 21h                  ;stops system from crashing

end start
```

The above program only does one thing - it displays a red dot in the middle of the screen. Now think about the code involved in something like *Tomb Raider…*

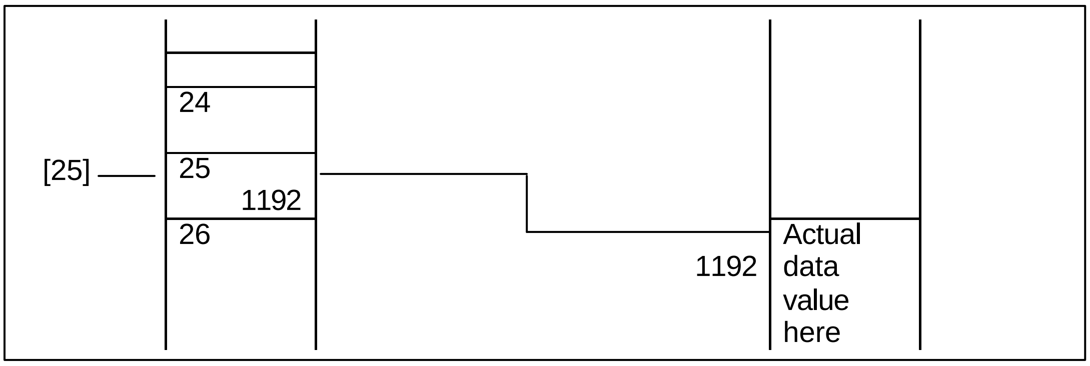
In December 1967, a computer at the University of London was used to simulate the Cheltenham Gold Cup. The current form of the 19 runners and riders was fed in, a random element added and the race was on!

In this picture the legendary Peter O'Sullivan does a live radio commentary as the computer prints out the latest positions. Bookmakers toted odds and took bets, just as if it were a live horse race - which was not possible at that time due to Foot and Mouth restrictions.

Indirect Addressing.

Up until now we have been assuming *direct addressing* mode, i.e. any locations we have specified contains the actual operand value. However, most processors also support Register-indirect, or just indirect, addressing. This is where the specified location contains the *address* of the operand value (the "effective address"). In other words, the first location acts as a pointer to where the value is actually held.

Indirect addressing is especially useful for circumstances where addresses of data structures might not be known until run-time; the program can still be set up with these "pointers", and the actual value can be loaded in later. Indirect addressing modes are usually expressed using square brackets - e.g. LOD RX [25] indirectly loads general purpose register X from location 25, which does not contain the actual data value but points to where it can be found.



As part of your Assessment question, you will be required to complete a grid showing the values in various registers as a particular program fragment runs. The following exercises are examples of assessment - level questions.

Assessment Example 1 - taken from the current SQA Exemplar © SQA 2001.

1. Complete the table to show all of the processor steps in fetching and executing the program instruction at location 3. You may use assembler mnemonics where appropriate. The table should be completed in base 10 numbers.

Note that the ADD instruction acts implicitly on the accumulator, and this version uses absolute addressing.

Location	Contents
7	22_{10}
6	
5	
4	
3	ADD 7_{10}
2	
1	
0	

Step	PC	MAR	MDR	IR	ACC
1	*3*	-	-	-	*5*
2					
3					
4					
5					
6					
7					
8					
9 10					
11					
12					
13					
14					
15					
16					

2. Complete the table to show all of the processor steps in fetching and executing the program instruction at location 4. Note that this is an indirect load into the general purpose register R1.

Location	Contents
14	254
13	
12	
11	
10	
9	
8	
7	14
6	
5	
4	LD R1 [7]
3	
2	
1	
0	

Step	PC	MAR	MDR	IR	ACC	R1
1	4				4	156
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14 15						

Example assessment 2. Taken from assessment in previous year.

Study the attached Assembly code program, RAM contents (partial diagram) and grid. Use the grid to complete the sequence of events, showing the values in each of the registers as lines 0000 to 0004 are executed. (You do not have to go any further than this.)

Note that line 0000 asks for input from the user. You may use any number you like (in hexdecimal format) provided that you show it being entered in the appropriate registers and memory locations.

Program.

```
0000 IN          # gets input from user & stores it in accumulator
0001 STA 15      # moves accumulator contents to RAM location 15
0002 LOD X [25]  # indirectly loads X from specified location
0003 CPXA        # copies contents of X to the accumulator
0004 ADD 15      # adds contents of 15 to accumulator
0005 OUT         # outputs the result
0006 HLT         # stop here!
```

RAM contents.

0000 10-00	0001 20-15	0002 30-[25]	0003 40–00	0004 50-15
0005 60-00	0006 70-00	0007	0008	0009
000A 7	000B	000C	000D	000E
000F	0010	0011	0012	0013
0014	0015	0016	0017	0018
0019	001A	001B	001C	001D
001E	001F	0020	0021	0022
0023	0024	0025 000A	0026	0027

Note - X is a general purpose register.

FETCH-DECODE-EXECUTE Grid.

Instruction	PC	MAR	MDR	IR	ACC	X
0000						
Fetch	0	0000	10-00			
Decode	1			10-00		
Execute					*	
0001						
Fetch	1	0001	20-15			
Decode						
Execute						
0002						
Fetch						
Decode						
Execute						
0003						
Fetch						
Decode						
Execute						
0004						
Fetch						
Decode						
Execute						

*Start here with the hexadecimal value you are inputting.