KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI

INSTITUTE OF DISTANCE LEARNING

(COMPUTER ENGINEERING 3)

COE 381: MICROPROCESSORS

[Credit: 3]

K. O. BOATENG

Publisher Information

For any information contact:

Dean
Institute of Distance Learning
New Library Building
Kwame Nkrumah University of Science and Technology
Kumasi, Ghana

Phone: +233-51-60013
+233-51-60023

Fax:
+233-51-60014

E-mail: cdce@fdlknust.edu.gh
kvcit@fdlknust.edu.gh
kvcitavu@yahoo.ca

Web: www.fdlknust.edu.gh

Guidelines for making use of learning support (virtual-classroom, etc.)

a. Go to www.kvcit.org
b. Sign in with your user name and password
c. Select your program and then the respective course
d. Read commends and contribute to the discussion or forum
e. For detail instruction see your e-classroom instruction

Course Author

DR. K. O. BOATENG

Course Introduction

Course overview

Main Objectives of Course

Course outline

Unit 1:	Basics of Microprocessors
Unit 2:	Memories
Unit 3:	Microprocessor Interfacing
Unit 4:	Intel Processor Basics
Tutorials:	Assembly Language

Reference/textbooks

Study Guide

week	reading (units/sessions)	FFFS/Assignments/Activity
1	1.1	
2	1.2	
3	2.1	
4	2.2	
5	3.1	
6	3.2	
7	4.1	
8	4.2	

Examinations

Table of Content

Information ii
Author iii
Introduction iv
of Content v
of Figures viii
of Tables x

1 1
OF MICROPROCESSORS 1
1-1: MICROPROCESSORS 1
-1.1 General-Purpose Computers and High Performance Systems 2
-1.2 Microcontrollers 3
-1.3 Embedded Computing 4
2-1: BASIC MICROPROCESSOR SYSTEM ARCHITECTURE 6
-1.1 The CPU 6
-1.2 Memory 10
-1.3 Input/Output 12
-1.4 The External Bus 12
3-1: THE BASIC COMPONENTS OF A MICROPROCESSOR 13
-1.1 The Registers 13
-1.2 The Internal Bus 15
-1.3 The Arithmetic Logic Unit 17
-1.4 The Control Unit 19
4-1 THE BASIC OPERATIONS OF A MICROPROCESSOR 19
-1.1 Instructions and Data 19
-1.2 The Control Logic 20
-1.3 The Internal Structure of a Simple Machine 21
-1.4 Microprocessor Instructions 22
-1.5 Microcycles 23

2 27
.... 27
1-2: MEMORY 28
-2.1 The stored program concept 28
-2.2 Addressing 29
-2.3 Address Decoding 30
-2.4 Commodity Memories 31
-2.5 Using Memory Chips 32
-2.6 Timing 33
-2.7 The Reality of Memory Decoding 34
-2.8 Filling the Memory Map 35
-2.9 Memory Map Details 36
-2.10 A separate I/O address space 37
-2.11 Endianness 37
-2.12 Memory Hierarchy 38
-2.13 Caches 39
-2.14 Harvard Architecture 40

-2.15 Read-Only Memory (ROM) 41
-2.16 Current Memory Technology 42
-2.17 Punch Cards, etc. 43
-2.18 ROM/PROM 44
-2.19 More Electrostatic Memories 46
-2.20 Magnetic Memories 47
-2.21 More Magnetic Memory 48
-2.22 Optical Memories 49
2-2: ADDRESS DECODING STRATEGIES 50
-2.1 Full Address Decoding 50
-2.2 Partial Address Decoding 53
-2.3 Designing Address Decoders 56
-2.3.1 Address Decoding with Random Logic 56
-2.3.2 Address Decoding with m-line-to-n-line Decoders 56
-2.3.3 Address Decoding with PROM 57
-2.3.4 Address Decoding with FPGA, PLA and PAL 61

3 66
INTERFACING 66
2-3: MICROPROCESSOR INTERFACING 71
-3.1 I/O addressing 71
-3.2 Compromises/extensions 72
-3.3 Types of bus-based I/O 72
-3.3.1 memory-mapped I/O and standard I/O 72
-3.3.2 Memory-mapped I/O vs. Standard I/O 73
-3.4 Microprocessor interfacing: interrupts 73
-3.4.1 Interrupt-driven I/O using fixed ISR location 73
-3.4.2 Interrupt-driven I/O using vectored interrupt 75
-3.4.3 Interrupt address table 78
-3.5 Direct memory access 78
-3.5.1 Peripheral to memory transfer without DMA, using vectored interrupt 79
-3.5.2 Peripheral to memory transfer with DMA 82
-3.6 Arbitration 85
-3.6.1 Priority arbiter 85
-3.6.2 Arbitration: Daisy-chain arbitration 86
-3.6.3 Network-oriented arbitration 87
-3.6.4 Intel 8237 DMA controller 87
-3.6.5 Intel 8259 programmable priority controller 88
-3.7 Multilevel bus architectures 89
-3.8 Advanced communication principles 90
-3.8.1 Parallel communication 90
-3.8.2 Serial communication 91
-3.8.3 Wireless communication 91
-3.9 Error detection and correction 91

4 93
PROCESSOR BASICS 93
1-4: MICROPROCESSOR FUNDAMENTALS 94
-4.1 The Major Components of a Computer 94
-4.1.1 Main Memory 95

-4.1.2 I/O Devices (Ports) 95
-4.1.3 Buses 95
-4.2 MPU Model 97
-4.2.1 Fetch-Decode-Execute 98
-4.3 Length of Instructions: 99
-4.4 Memory Map 99
-4.5 Memory Devices 100
-4.6 Memory Read and Write Control Signals 101
-4.7 Input and Output Cycles 103
-4.8 Memory Mapped I/O and Separate I/O Mapping. 103
-4.9 Address Decoding 104
-4.10 Data Buses and Tristate Logic 105
-4.11 Memory Access Timing 108
2-4: THE 8086 AND 8088 MICROPROCESSORS; INTERRUPTS AND DM 111
-4.1 The 8088 Microprocessor in Circuit 113
-4.2 8086/8088 Memory and I/O Cycles 116
-4.3 Interrupts 119
-4.4 Direct Memory Access (DMA) 122
-4.4.1 DMA Methods 122
.3.2.2 The DMA Interface with the Microprocessor 122
.3.2.3 The DMA Interface with the I/O Device 123
.3.2.4 DMA Control Signals for Data Transfer 123
-4.5 Microcomputer Performance Considerations 124
-4.5.1 Processor Clock Frequency 125
-4.5.2 Data Bus Width 125
-4.5.3 Address Bus Width 125

.... 127
LANGUAGE 128

List of Figures

1: Figure 1.1.1: Total Development and Production Cost vs. 2
2: Figure 1.1.2: Embedded and Non-embedded 5
3: Figure 1.2.1: Basic Microprocessor 6
4: Figure 1.2.2: Simple overview of CPU Internal 7
5: Figure 1.2.3: CPU Internal 9
6: Figure 1.2.4: Relationship between software languages, micro-operations and CPU 10
7: Figure 1.2.5: Input 12
8: Figure 1.3.1: 13
9: Figure 1.3.2: One Stage 15
10: Figure 1.3.3: One Stage Shift Left or 15
11: Figure 1.3.4 and 1.3.5: Bus Structure and 16
12: Figure 1.3.6: ALU 17
13: Figure 1.3.7: 74181 18
14: Figure 1.4.1: Simpilified Microprocessor Design 19
15: Figure 1.4.2: The Control Logic 20
16: Figure 1.4.3: The Simple 21
17: Figure 1.4.4: Memory Location and CPU 24
18: 30
19: 31
20: 2 Mbyte memory using 512Kbyte 33
21: 34
23: 35
24: 40
25: 45
26: 48
27: Figure 2.2.3 - Full Address Decoding 51
28: Figure 2.2.4 - Full Address Decoding Schematic Diagram Corresponding to Table 2.2.3. 52
29: Figure 2.2.5 - Partial Address Decoding of Two Memory 53
30: Figure 2.2.6 - Memory Map Corresponding to Figure 2.2. 54
31: Figure 2.2.7 - Partial Address Decoding Schematic Diagram Corresponding to Table 2.2.4. 55
32: Figure 2.2.8 - Pinout Diagram of 74LS138 56
33: Figure 2.2.9 - Applications of the three-line-to-eight-line 57
34: Figure 2.2.12 - PROM-based Address Decoder 58
35: Figure 2.2.13 - Structure of a PROM in Terms of Gate 62
36: Figure 2.2.14 - Structure of a PLA in Terms of Gate 63
37: Figure 2.2.15 - Structure of a PAL in Terms of Gate 64
38: 69
39: 70
40: 70
41: 71
42: 72
43: 74
44: 75
45: 75
46: 75

47: 76
48: 76
49: 77
50: 77
51: 78
52: 79
53: 79
54: 80
55: 80
56: 81
57: 81
58: 82
59: 82
60: 83
61: 83
62: 84
63: 84
64: 85
65: 85
66: 86
67: Daisy-chain aware 87
68: 87
69: 88
70: 89
71: 90
72: Fig.1. Simplified block diagram of a Microcomputer (MPU or CPU). 94
73: Fig. 2. Shows the Internal Block Diagram of a Microprocessor. 96
74: Fig.3 Typical Program Status Word or Status 97
75: a simple microprocessor 97
76: 98
77: Fig. 5. Fetch Decode Execute 98
78: 99
79: Fig. 6. Memory Map. 100
80: Fig. 7. Memory 101
81: 101
82: Fig. 8. Read Cycle. 102
83: Fig. 9. Write Cycle. 102
84: 102
85: 104
86: 105
87: 105
88: 107
89: 108
90: 109
91: 110
92: 110
93: 110
94: 110
95: 111
96: 112

97: 114
98: 115
99: 116
100: 116
101: 117
102: 118
103: 118
104: 120
105: 121
106: 123
107: 124
108: 129
109: 130
110: 132
111: 134
112: 136
113: 138
114: 141
115: 162
.... 176
.... 181
118: 182
119: 183

List of Tables

1: Table 1.1.1: Advantages of microprocessor design over random 1
2: Table 1.1.2: Representative applications of 4
3: Table 1.2.1: Common Memory 11
4: Table 1.3.1: Common micro- 14
5: Table 1.3.2: The micro-operation 14
6: Table 1.3.3: Some typical ALU 17
7:.. 33
8:.. 34
9: Table 2.2.1 - Devices Requiring Address 51
10: Table 2.2.2 - Device Memory 52
11: Table 2.2.3 - Address Decoding 52
12: Table 2.2.4 - Address Decoding Table for Partial Address 54
13: Table 2.2.5 - Improved Partial Address Decoding 55
14: Table 2.2.6 - Required Memory Space 58
15: Table 2.2.7 Proramming of Address Decoding 59
16:.. 88
17:.. 89
18:.. 131
19:.. 133
20:.. 133
21:.. 134
22:.. 135
23:.. 135
24:.. 137
25:.. 139
26:.. 139
27:.. 140
28:.. 140
29:.. 142
30:.. 147
31:.. 159
32:.. 161
33:.. 166
34:.. 167
35:.. 167
36:.. 169
37:.. 171
38:.. 173
39:.. 173
40:.. 174
41:.. 175
.... 176
43:.. 177
44:.. 178
45:.. 178
46:.. 179
47:.. 180

Unit 1

BASICS OF MICROPROCESSORS

Introduction

In a computing system the component known as the Central Processing Unit (CPU) is responsible for executing instructions which cause it to interact with the computing system's memory, input devices and output devices. Initially CPUs (or simply processors) were constructed from several silicon chips using random logic techniques. But as technology rapidly progressed Very Large Scale Integration (VLSI) techniques in the 1970s allowed a CPU to be fabricated on a single chip. Any single chip which includes as a part or implements fully a CPU is referred to as a microprocessor. Microprocessors are usually classified by their word size - the number of bits that can be stored in a single internal CPU storage unit.

Learning objectives

Unit content

Session 1-1: Microprocessors

1-1.1 General-Purpose Computers and High Performance Systems
1-1.2 Microcontrollers
1-1.3 Embedded Computing

Session 2-1: Basic Microprocessor System Architecture

2-1.1 The CPU
2-1.2 Memory

2-1.3 Input/Output
2-1.4 The External Bus

Session 3-1: The Basic Components of a Microprocessor
3-1.1 The Registers
3-1.2 The Internal Bus
3-1.3 The Arithmetic Logic Unit
3-1.4 The Control Unit

Session 4-1: The Basic Operations of a Microprocessor
4-1.1 Instructions and Data
4-1.2 The Control Logic
4-1.3 The Internal Structure of a Simple Machine
4-1.4 Microprocessor Instructions
4-1.5 Microcycles

SESSION 1-1: MICROPROCESSORS

The general use of a microprocessor in engineering applications is to replace hardwired logic. The main advantages offered by microprocessors are lower cost, fewer components and increased versatility. Existing systems can be modified at will by changing a sequence of instructions, often merely by exchanging a ROM (memory) device. However one should not be misled, as program development is time consuming and changes may not always be as simple as they sound.

Designing with a microprocessor involves exchanging logic arrays for software and there is an optimum balance between the two which provides the best performance in a given system at a reasonable cost. The system designer requires both logic design and software expertise. The table1 below summarizes the advantages of microprocessor design over random logic:

Table 1: Table 1.1.1: Advantages of microprocessor design over random logic

Area of comparison	Random logic	Microprocessor
Design:	Specific	General
Design Emphasis:	Hardware	Software
Package Count:	High	Low
Flexibility:	Low	High
Redesign/modification:	Difficult	Relatively easy (Families of Products)
Circuitry:	Redesign always	Alike
Speed:	Very fast	Moderate

However, there are exceptions to these rules namely that there are rare instances where the sophisticated optimizations inherent in a microprocessor design allows a software program to outperform the speed of an equivalent simple random logic design.

Both random logic and microprocessor designs have high development costs, which for random logic are in the hardware design and for microprocessors are mostly in the software design. Development using both methods requires one to have the necessary development tools to aid one in the hardware or complex software design process. However, once volume production of a design begins the far lower component count of the microprocessor design results in a marked effect on unit price as can be seen below:

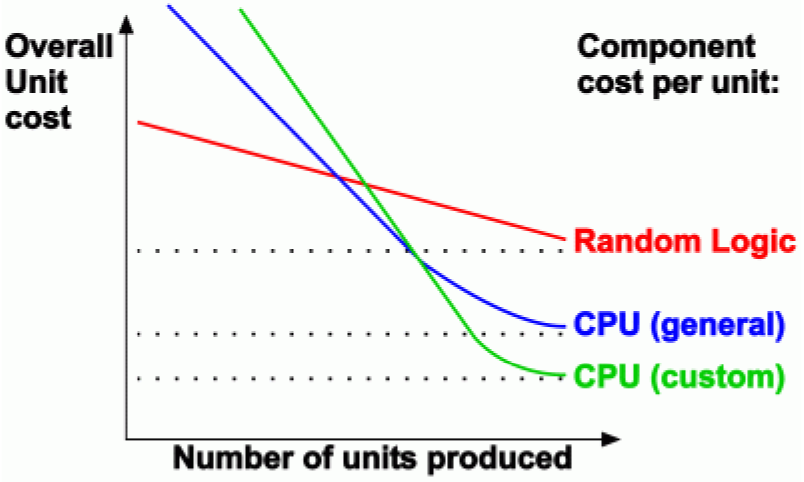


Figure 1: Figure 1.1.1: Total Development and Production Cost vs. Quantity

It should be noted that, in general, microprocessor systems can be divided into two groups:

1-1.1 General-Purpose Computers and High Performance Systems

As the name suggests general-purpose computers are designed to be as versatile as possible. They usually consist of a main Printed Circuit Board (PCB) referred to as a motherboard. A microprocessor can be found on this board along with its associated support chipset. In addition, expansion slots allow more PCBs to be connected to the motherboard. This in turn enables the computer to perform a huge variety of tasks simply by inserting boards with the desired capabilities. An example of such a system is a desktop Personal Computer (PC). Since both the hardware and the software for such systems are mass-produced the total system cost will be relatively low. The microprocessors in these machines may include additional logic which allows it to perform certain computational tasks more efficiently or which allows the support chipset to be simpler by moving some of the chipset's functionality onto the microprocessor chip.

High performance computational and data processing systems make use of general purpose computers whose microprocessors include such additional logic or which have had expansion boards added to aid in the task to be performed. Such additional logic can increase the system cost but the benefits gained outweigh this disadvantage since the main hardware design requirements in these systems are increased speed and computational capabilities. Such systems will normally be produced in relatively small quantities with the software development cost being enormous. Consequently, the system prices will be high.

The main factors in choosing a microprocessor for a general-purpose computer will be cost and speed whereas in a high-performance system the microprocessor's instruction set capabilities, cost, speed, memory range and available software development tools will all require careful consideration.

1-1.2 Microcontrollers

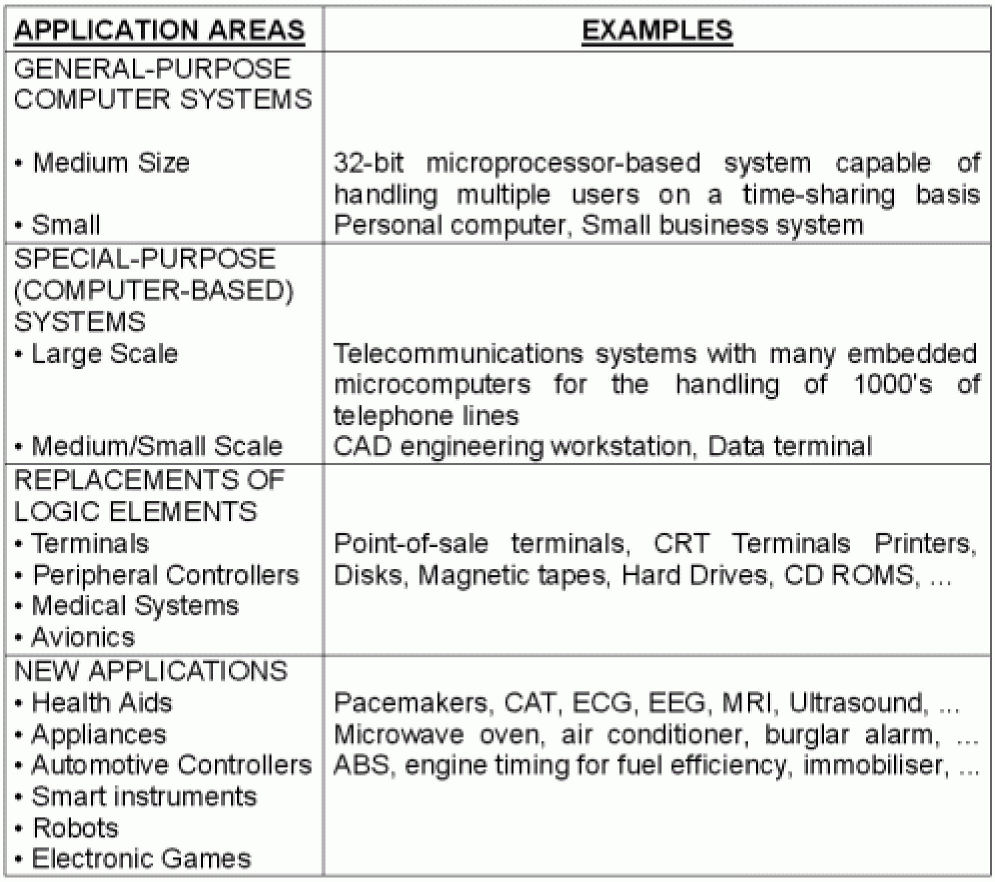
A microcontroller is a microprocessor-based system which replaces an equivalent random logic design. The microcontroller may be implemented as a set of chips on a PCB or may consist of just one single chip. If in the single-chip implementation the microcontroller software can be programmed into the chip after it has been placed on a PCB then the microcontroller is referred to as a Programmable In-circuit Controller (PIC).

Microcontrollers do not, in general, require excellent CPU performance or sophisticated software programs. Consequently, software costs may be moderate and the main cost factor will be the component count which must be kept as low as possible. A microcontroller may also be placed in a hazardous environment, may be mobile or may be one of many other microcontrollers in a given device. As a result, power consumption may also be an important consideration. While most microcontrollers are fixed and do not offer any means of expansion, there are instances where microcontrollers may be modified by software means while in service. In addition, some may be designed for some limited form of hardware expansion.

Hence, the main factors in choosing a microprocessor for a microcontroller will be the overall system chip count, available external interfacing features (I/O lines, ...), the number of on-chip support components (timers, DACs, ADCs, ...) and power consumption.

Some examples of applications in which microcomputers (general-purpose, high performance and microcontrollers) can be used are listed below:

Table 2: Table 1.1.2: Representative applications of microcomputers



APPLICATION AREAS	EXAMPLES
GENERAL-PURPOSE COMPUTER SYSTEMS	
• Medium Size	32-bit microprocessor-based system capable of handling multiple users on a time-sharing basis
• Small	Personal computer, Small business system
SPECIAL-PURPOSE (COMPUTER-BASED) SYSTEMS	
• Large Scale	Telecommunications systems with many embedded microcomputers for the handling of 1000's of telephone lines
• Medium/Small Scale	CAD engineering workstation, Data terminal
REPLACEMENTS OF LOGIC ELEMENTS	
• Terminals • Peripheral Controllers • Medical Systems • Avionics	Point-of-sale terminals, CRT Terminals Printers, Disks, Magnetic tapes, Hard Drives, CD ROMS, ...
NEW APPLICATIONS	
• Health Aids	Pacemakers, CAT, ECG, EEG, MRI, Ultrasound, ...
• Appliances	Microwave oven, air conditioner, burglar alarm, ...
• Automotive Controllers	ABS, engine timing for fuel efficiency, immobiliser, ...
• Smart instruments	
• Robots	
• Electronic Games	

1-1.3 Embedded Computing

In the broadest possible sense, an embedded computing system is any device which includes a computer but which is not intended to be a general-purpose computer. Hence, a desktop PC is not an embedded system but a desktop PC may be a component of an embedded computing system. Confused? Well, look at the diagram below:

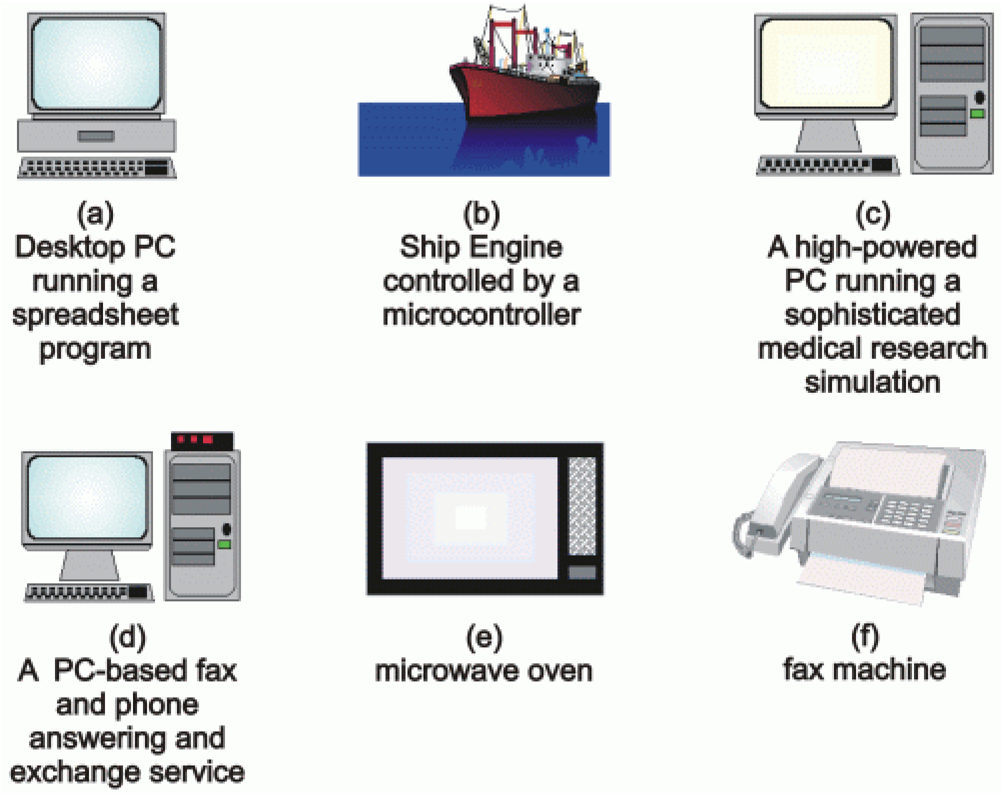

(a)
Desktop PC running a spreadsheet program
(b)
Ship Engine controlled by a microcontroller
(c)
A high-powered PC running a sophisticated medical research simulation
(d)
A PC-based fax and phone answering and exchange service
(e)
microwave oven
(f)
fax machine

Figure 2: Figure 1.1.2: Embedded and Non-embedded systems

Can you tell which of the systems above are embedded and which are not?
ANS: (b),(d),(e) and (f) are and (a) and (c) are not.

For engineers an Internet kiosk is not a very interesting embedded system: there is no component, including the coin payment unit, which is not "off-the-shelf" technology. Similarly, a simple digital thermometer is also not very interesting: there is no microcontroller which will not perform the task so one can just use the cheapest one available. In both these examples there is virtually nothing to do - one can almost do without an engineer! For this reason engineers are mainly interested in those embedded systems which present real engineering challenges to the designer.

Self-Assessment 1-1

1.

SESSION 2-1: BASIC MICROPROCESSOR SYSTEM ARCHITECTURE

The fundamental architecture of a microprocessor-based system is depicted below:

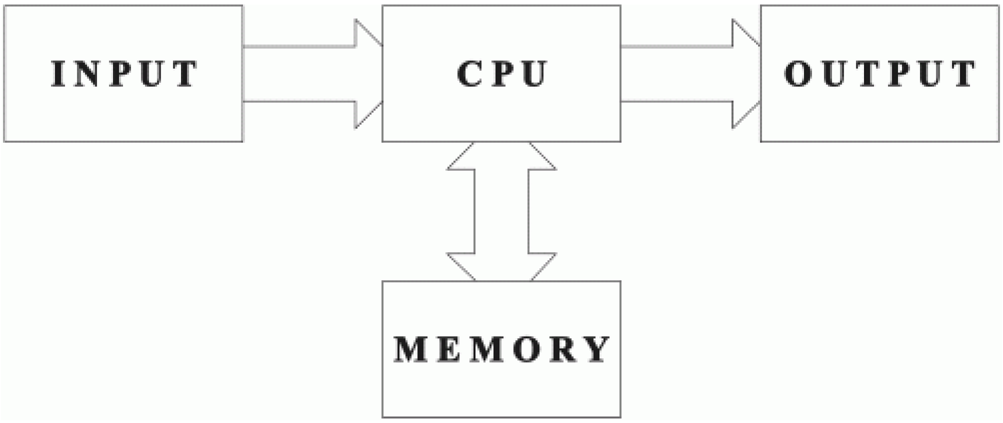


Figure 3: Figure 1.2.1: Basic Microprocessor System

This system is based on a simplified von Neumann architecture. In this architecture both the instructions which the CPU executes and the data it operates on are kept in a single memory store. An alternative to this is the Harvard architecture where data and instructions are kept in separate memory subsystems and operated on via separate physical data paths so that both data and instructions can be fetched or stored in a single memory operation. The von Neumann architecture is the most common while the Harvard architecture is employed in specialised microprocessors and other digital chip subsystems where it provides a performance advantage over the von Neumann design. Since the von Neumann architecture is the simplest, most common and most economical basic computer architecture it alone will be studied for the remainder of the notes.

2-1.1 The CPU

A simple overview of the internals of a typical CPU can be seen below:

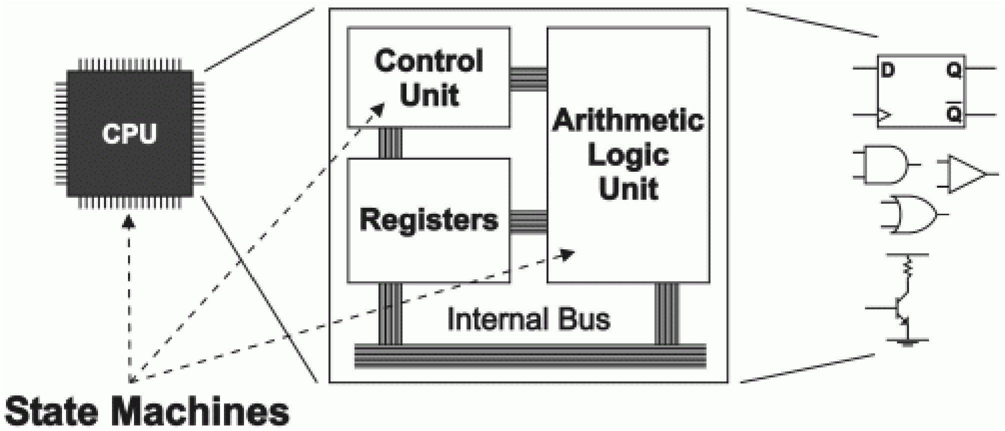


Figure 4: Figure 1.2.2: Simple overview of CPU Internal Structure

While the CPU is powered it performs the following instruction cycle continuously:

1. Fetch an instruction from memory
2. Decode the instruction
3. Perform the necessary steps to carry out the instruction (store/fetch data to/from external memory or internal CPU storage, perform an arithmetic operation on data held in the CPU, jump to a different instruction location based on the outcome of a previous instruction's logical result)
4. If the instruction carried out was not a jump to another instruction in memory then setup the CPU to fetch the next consecutive instruction in memory
5. Repeat all this starting again from step (1)

The CPU consists of three main sections:

1. *Registers*: high speed locations used to store important information during CPU operations.
2. *Arithmetic/Logic Unit*: performs operations on operands (stored in registers), for example the addition of two numbers, and stores the result (in a register).
3. *Control Unit*: generates all the control signals which control the flow of information within the CPU and the transfer of data between the CPU and memory and I/O devices.

The internal busses of the CPU allow data and control signals to be passed between these units.

Although a high-level view of the CPU has been presented so far, it is important to remember that any single component of the CPU, and the CPU in its entirety, is nothing more than a collection of millions of transistors making up a complex logic gate circuit. These circuits include both sequential and combinational logic blocks which make up the CPU - a single massive sequential logic design. Despite the size of the CPU it is well within the ability of a 3rd year electrical engineering student to build simple versions of every single block of a simple CPU directly using the techniques that they have learned: algorithmic state machine design, truth tables and so on. However, it is also important to remember that commercial CPUs are very complex devices built by teams of highly experienced engineers. The majority of such teams surpass embedded system design teams in terms of both the number of team members and available development funds.

Also, real CPUs may have several internal busses to allow the concurrent execution of instructions: if two consecutive instructions do not require access to the same units of the CPU in order to complete then they can be executed at the same time provided that there are separate busses linking the units in question. Note that this also requires that the CPU fetch the next instruction while the current one is executing or alternatively the CPU must fetch several instructions in one memory access. The parallel execution of instructions is made possible by a pipelined CPU design. Such a design effects the IR register, the control unit and the busses of the simple CPU we have discussed. This results in increased CPU performance. The complexity of such real CPUs and their operation is beyond the scope of these notes.

The CPU is responsible for the co-ordination of the information flow within a system. To achieve this goal the CPU fetches and executes instructions from memory. Similarly, the CPU’s control unit is responsible for co-ordinating the flow of information inside the CPU. In much the same way as one does not concentrate on the transistor-level logic when designing a sequential circuit with flip-flops one does not concentrate on the specific logic-level operations when designing or describing the operations of the control unit. One rather refers to collections of control unit logical operations as single micro-instructions. In order to describe these micro-instructions in more detail the structure of a simple real CPU is depicted below:

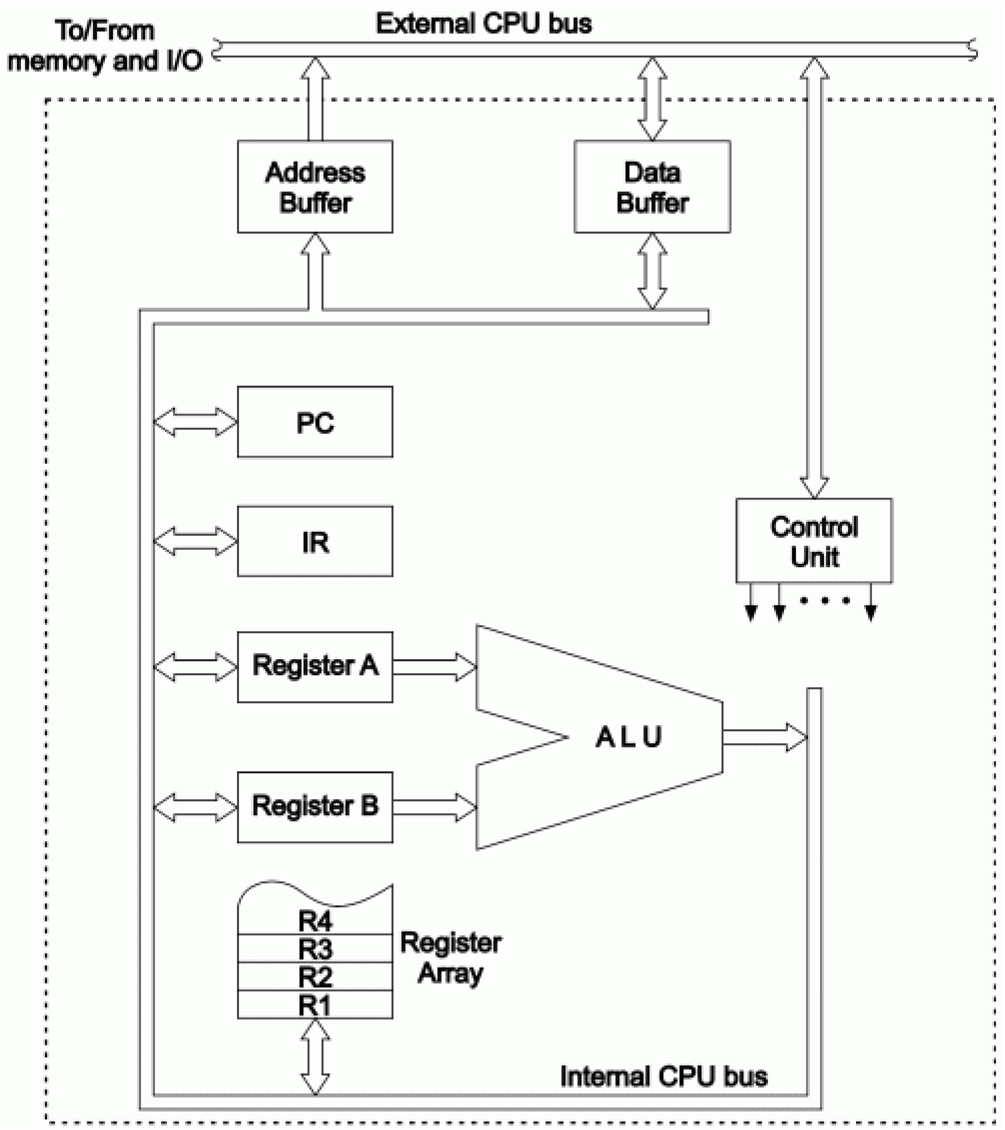


Figure 5: Figure 1.2.3: CPU Internal Organisation

As an example, in order to move the data contained in register R4 into the Address buffer register the control unit needs to assert the correct logic signals to make R4 place the data on the internal CPU bus and to assert the correct logic signals to make the Address buffer read this data off the bus. We would call this entire process of asserting the correct control signals a micro-operation of the CPU which is, of course, carried out by the control unit. The particular example micro-operation we have just described would be expressed as: **Address buffer ← R4.**

In order to carry out each step of a CPU instruction cycle the control unit must issue several micro-operations based on the fetched instruction. The set of instructions provided by the CPU to instruct it to perform certain operations is known as the instruction set. The instructions in this set are not able to refer to all registers of the CPU. Many registers are used as temporary storage locations when carrying out an instruction. In the above diagram for example, the IR holds the current instruction to be executed and cannot be referenced by any instruction in the CPU's instruction set yet it is used during each and every CPU instruction cycle.

Note that programmers do not program in micro-operations. Programmers use a high-level language, such as C++, to write programs. A program called a compiler compiles the high-level language into assembly language and then into machine code. An assembly language instruction will generally correspond exactly to a machine code instruction. The difference is that the assembly language instruction is in a human-readable form whereas the machine code instruction is simply a number which is understood by the control unit of the CPU. This number causes the control unit to issue a series of actions which we can describe using micro-operations. This relationship is depicted below:

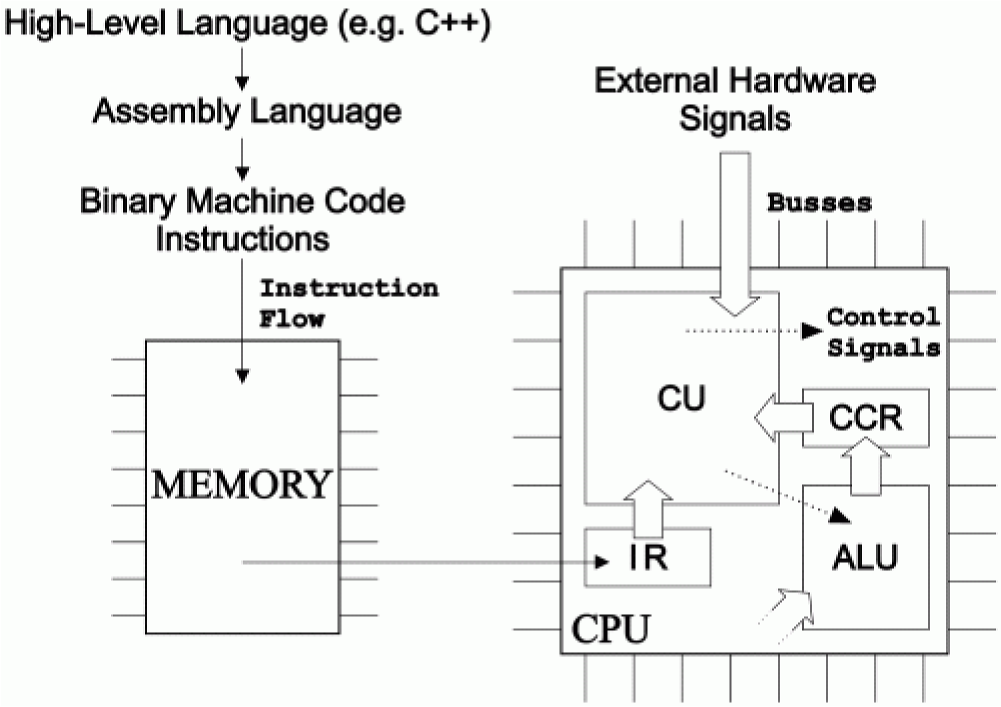

High-Level Language (e.g. C++)
Assembly Language
Binary Machine Code Instructions
Instruction Flow
MEMORY
External Hardware Signals
Busses
Control Signals
CU
CCR
IR
ALU
CPU

Figure 6: Figure 1.2.4: Relationship between software languages, micro-operations and CPU hardware

2-1.2 Memory

This is a (normally) very large array of storage cells, each capable of holding or remembering one bit (BInary digiT) of information. During the operation of the system, the CPU will constantly be retrieving binary information (reading) from the memory, or storing binary information (writing) for later use. It is very seldom that this is required to occur on a single-bit basis; normally a number of bits will be altered simultaneously, and the memory is consequently organised into words, each word consisting of a fixed number of bits. This number of bits is called the width of the memory, and varies from computer to computer. A width of 8 bits (a frequent sub-multiple of the word size) is known as a byte.

Since these words of memory will not normally be read or written in a strictly sequential or linear fashion, they are each given a unique address to distinguish them one from another. The CPU must indicate the address of the word that it wishes to read or write, and since this is naturally a binary number, the number of words in any memory array will be a power of 2 (e.g. 16-bit wide addresses = or 65 536 different combinations).

Memory sizes are usually expressed in units of (1024), which is denoted by the letter K, followed by the width in bits. Thus a memory array might be specified as 2K x 8, which means 2048 locations, each containing 8 bits (or one byte).

The CPU retrieves from memory:

1. The binary patterns that control its operation (the instructions; also referred to as code), and
2. The binary patterns that are used or affected during the course of such operation (data as opposed to code).

NOTE: Both code and data will occur in the same memory, and are not separated or distinguishable in any way. The only difference between them is one of context or interpretation.

A memory device may be read-only, which means that no information may be stored in any of the words (i.e. the information that is in the words already can never be changed) or it may be read/write, which means that the information in the words can be overwritten by the microprocessor. A third class of memory, write-once memory which may be altered only once, exists (for such things as audit trails) but is not widely used.

Memory devices fall into two basic classes:

1. *Volatile*: these devices will only retain their contents while power is applied. When power is first applied, their contents will be indeterminate. Such devices can never be read-only devices, and are normally called RAM (random access memory, an unfortunate but persistent acronym) or RWM (read/write memory, an unpronounceable acronym).

2. *Non-volatile*: A non-volatile device will retain its contents even if power is removed from the device and then re-applied, until it is overwritten (if this is possible). Such devices are used to contain the initial instructions that the CPU needs when power is first applied to the system. Although a non-volatile device is normally read-only and is consequently called a ROM (read-only memory), some non-volatile devices do exist which may be written to with varying degrees of ease and speed. Although technically these are also RAM devices, they are often called ROMs because the operation of writing to the words is relatively complex and slow compared to reading. They are instead identified by a letter or letters preceding the ROM (e.g. PROM = Programmable ROM, EPROM = Erasable Programmable ROM, EEROM = Electrically Erasable ROM etc.)

Common devices in each class are:

Table 3: Table 1.2.1: Common Memory Devices

Class	Type	Devices
Non-Volatile	Read Only	EPROM, PROM, ROM, Punched tape
	Read/Write	EEPROM, EAROM
Volatile	Read/Write	Dynamic RAM
	Static RAM	Magnetic Tape, Disk

In the microprocessor context, 'memory' is normally taken to mean high-speed devices from which data may be read or to which it may be written on a single word or byte basis at the speed of the microprocessor. Currently the only devices that fall into this class are semiconductor memories; tapes and disks are normally called peripheral or mass storage devices.

2-1.3 Input/Output

I/O devices are the means by which a computer system communicates with an external environment (the real world). These I/O devices must present to the computer a digital interface, known as an I/O port. These ports are similar to memory in that each port appears to the CPU as a set of storage cells, normally of the same width as the memory, and each port will have associated with it a unique address.

The difference is that the contents of a port are accessible to the external system as well as to the microprocessor system, so that the external system may alter the contents of a port (input port), and the operation of the CPU may subsequently change. Alternatively, the CPU may alter the port (output port) and the external system's operation may be affected as a result.

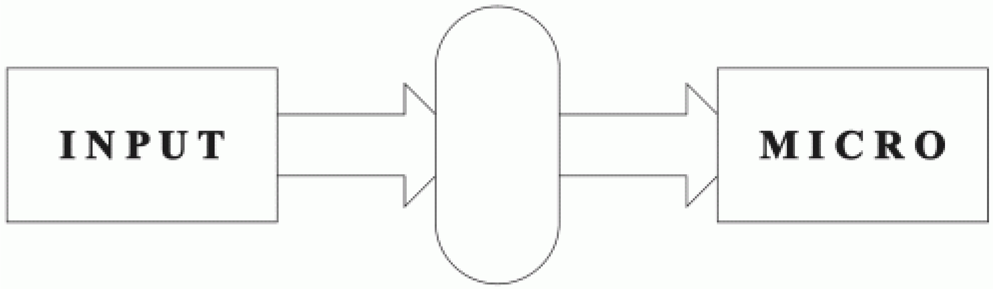


Figure 7: Figure 1.2.5: Input port

NOTE: I/O provides the only means by which the operation of a computer system may be altered, or by which it may alter another system's operation. Consequently any microprocessor system is useless without I/O.

2-1.4 The External Bus

The CPU communicates with memory and peripheral devices along a set of signal lines referred to as the bus. The CPU partitions the signal lines into address, data and control buses that connect the CPU with the other circuitry of the computer.
In general, a simple computer partitions the bus into memory address and data busses. In a Harvard architecture system there are two sets of memory and data bus pairs: one pair for the instruction memory bank and one pair for the data memory bank. Also, certain specialised processors may have separate busses for accessing memory and I/O devices.

Self-Assessment 2-1

1.

SESSION 3-1: THE BASIC COMPONENTS OF A MICROPROCESSOR

A microprocessor, in its simplest form, is a large clocked sequential circuit and as such will respond to a series of binary input information signals in a known and defined manner. By applying these signals in an orderly fashion, the designer may force the microprocessor to perform a specific operation; such binary signals are called instructions, and the sequence which makes the CPU operate usefully is called a program. The generation and application of these binary instructions is called machine-code programming.

A microprocessor is a very large and complex circuit, and cannot be easily discussed by referring to the individual digital elements (e.g. gates, flip-flops) from which it is composed. Its design can, however, be separated into a number of logically discrete functional units.

3-1.1 The Registers

All microprocessors have many registers, which are fundamental to the sequential nature of the device. Each register is made up of a number of flip-flops connected in parallel and simultaneously clocked. A part of a typical register is shown in in the figure below. Information is latched or stored in these registers; such information may be moved between the registers and the external memory devices, or it may be moved internally within the microprocessor from one register to another. While the information is in the registers within the CPU, it may be manipulated and processed in some way before it is transferred.

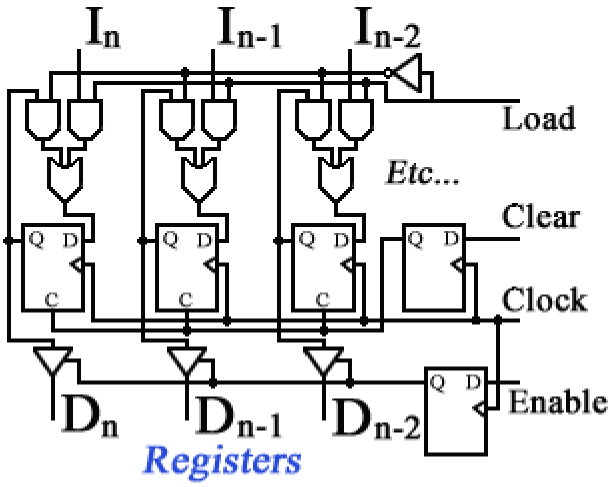


Figure 8: Figure 1.3.1: Registers

Usually the width of the registers will be related to the width of the memory by a factor of a power of two; for example, a microprocessor with a memory width of 8 bits will normally have registers of 8 and 16 bits width.

A register will be able to manipulate the data stored in it in some simple way, and will have some combinational logic associated with it to perform these functions. The operations that manipulate the register contents are called micro-operations; a single microprocessor instruction, as defined by a word in memory, will cause a defined sequence of micro-operations to occur within the CPU.

Some common micro-operations are given below. Normally, a register will not perform all of these operations, but only a small subset which is applicable to its purpose.

Table 4: Table 1.3.1: Common micro-instructions

Function	**Micro-operation**
Load register R	Ri ← Ii
Clear R (specialised load)	Ri ← 00
Complement R	Ri ← /Ri
Increment R (by one)	R ← R + 1
Decrement R	R ← R - 1
Shift R left	Ri+1 ← Ri ; R0 ← 0
Shift R right	Ri-1 ← Ri ; Rn ← 0
Serial load R (left)	Ri+1 ← Ri ; R0 ← I
Serial load R (right)	Ri-1 ← Ri ; Rn ← I

In these micro-operations, the following notation is assumed:

Table 5: Table 1.3.2: The micro-operation notation

Rn (Most significant bit)	Ri (A bit)	R0 (Least significant bit)

Some typical circuits which will perform these functions are given in Figures 1.3.2[9] and 3.3[10]. Note that the functions are shown in separate circuits, but in reality this is not normally the case; the desired functions for any register will be combined into one circuit.

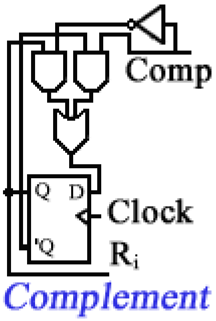


Figure 9: Figure 1.3.2: One Stage Complement

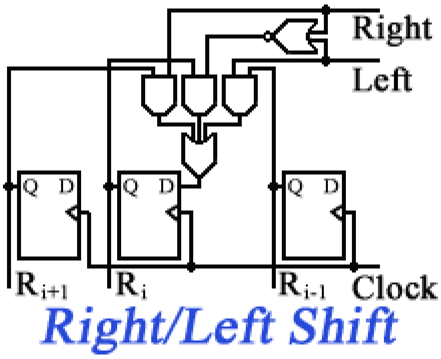


Figure 10: Figure 1.3.3: One Stage Shift Left or Right

3-1.2 The Internal Bus

The microprocessor will contain many registers, and will normally need to be able to transfer the contents of any register to any other register with ease. This is made possible by the use of a bus within the microprocessor as shown in Figure 1.3.4 [11] and 3.5 [12] below; all the like inputs and outputs of the registers are commoned together, so that any register's inputs may be driven by any register's outputs. Such an arrangement is possible only if the outputs of the registers can be disabled in some way to prevent multiple outputs from trying to drive a single line to different states. This may be achieved by using tri-state buffers on the outputs of the registers, as shown in the Figure 3.1 [?]; the outputs of the register are only enabled when required; otherwise they are placed in the high-impedance state. It should be noted that the enabling of a register is an exclusive function - only one enable line may be active at any stage.

NB: All control lines to the register are latched so that they take effect in synchronism with the microprocessor clock signal; thus the enable line for a register may be set to true (i.e. enable the outputs) at any stage, but the register outputs will not be enabled until the next edge of the clock signal. All the registers in the microprocessor, and thus all the information transfers and manipulation, are synchronised to one master clock signal.

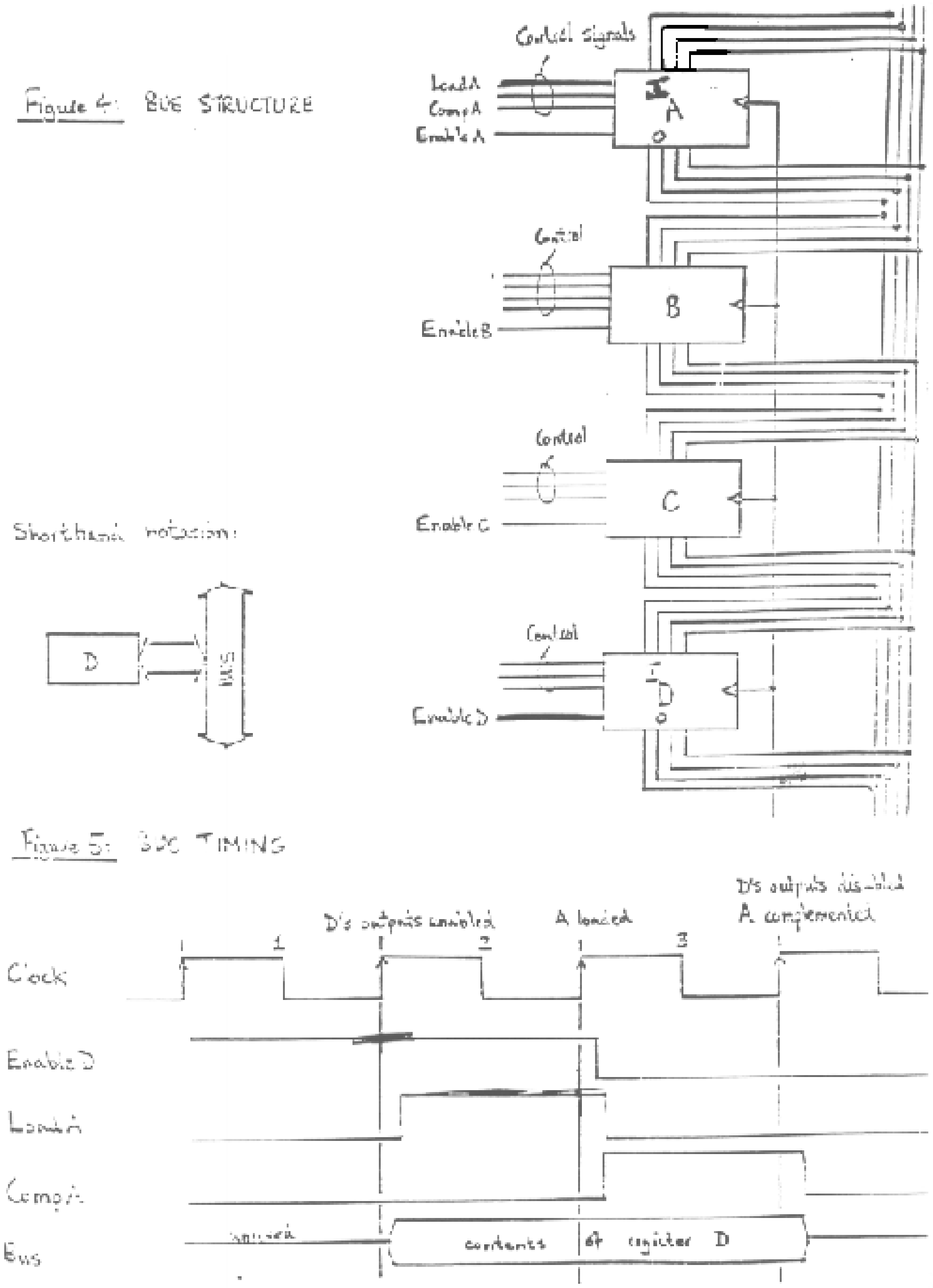


Figure 11: Figure 1.3.4 and 1.3.5: Bus Structure and Timing

As an example, based on the Timing Diagram and the Bus Structure in Figure 1.3.4 [11] and 1.3.5, let us transfer the contents of register D to register A, and then complement (invert) the contents of register A, the following micro-operations would be performed:

Clock cycle 1:	Enable D = 1	(Enable the outputs of D onto the bus)
Clock cycle 2:	Load A = 1	(Write the bus state into A)
Clock cycle 3:	Comp A = 1	(Complement the contents of A)
	Enable D = 0	(Disable the outputs of D)

These are shown in timing diagram form in Figure 1.3.6 [12]. Note that three clock cycles were used to achieve what was (presumably) one instruction. The first clock cycle was only used to enable the outputs of register D; if a number of instructions were being performed, this would probably be done during the last clock cycle of the previous instruction, i.e. the clock cycles of successive instructions would be overlapped to increase the speed of the processor.

3-1.3 The Arithmetic Logic Unit

The ALU is a complex combinational circuit that is used within the CPU to perform logic and arithmetic operations on the contents of registers. It is a dedicated unit and thus capable of performing much more complex operations than a register. It is shown in diagrammatic form in Figure 1.3.6[12] below, and the internal circuit of a sample ALU (the 74181 chip) is shown in Figure 1.3.7. [13]Note that the 74181 is only a four-bit ALU; most microprocessor ALUs are eight or sixteen bits wide.

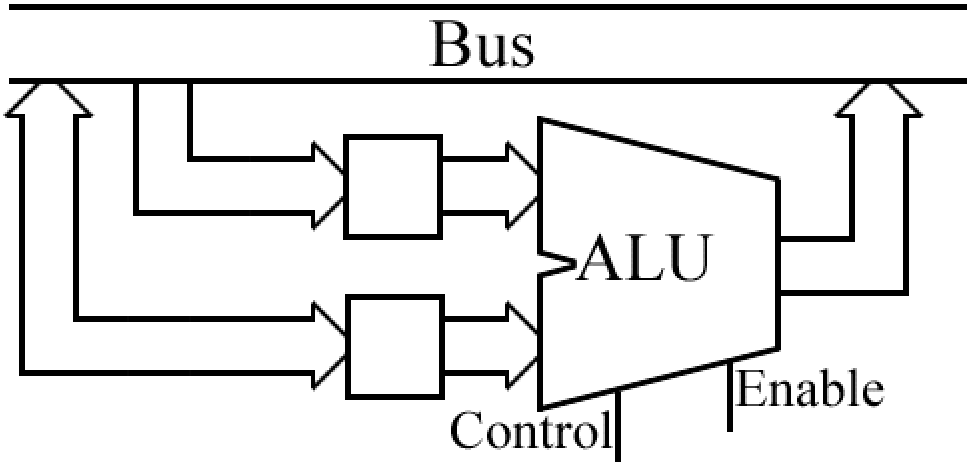


Figure 12: Figure 1.3.6: ALU Circuitry

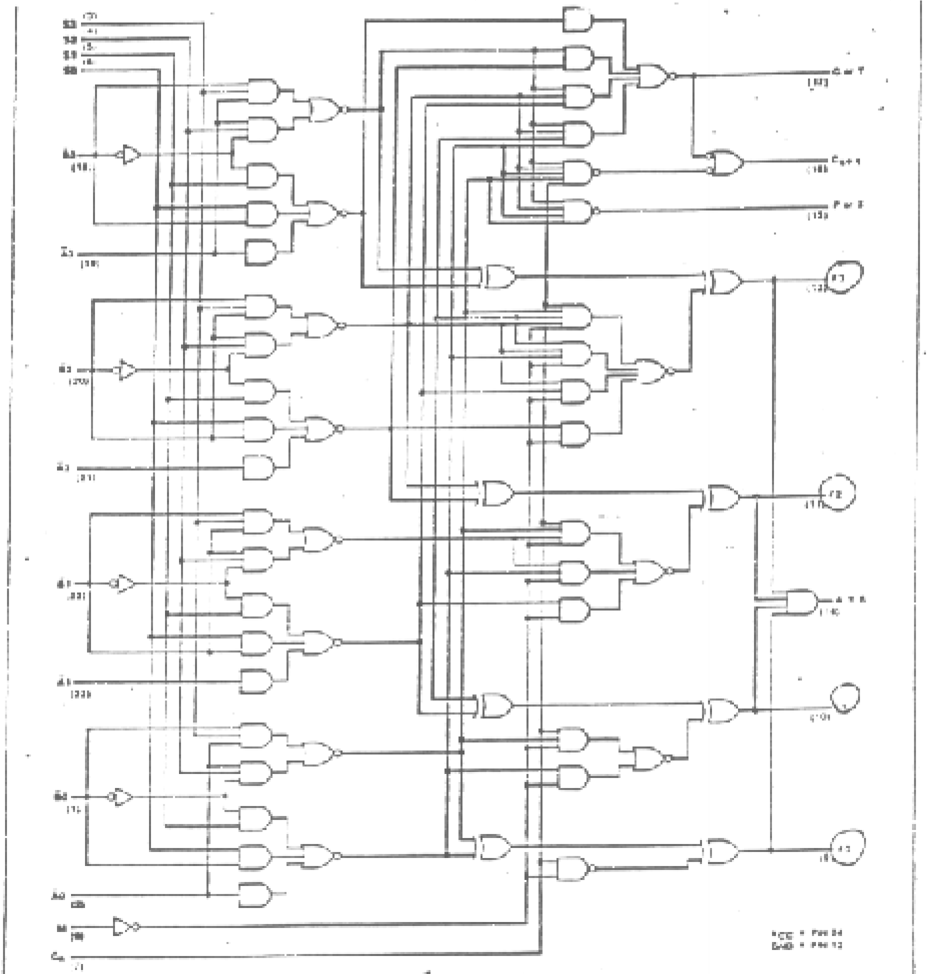
Some typical ALU operations might be:

Table 6: Table 1.3.3: Some typical ALU operations

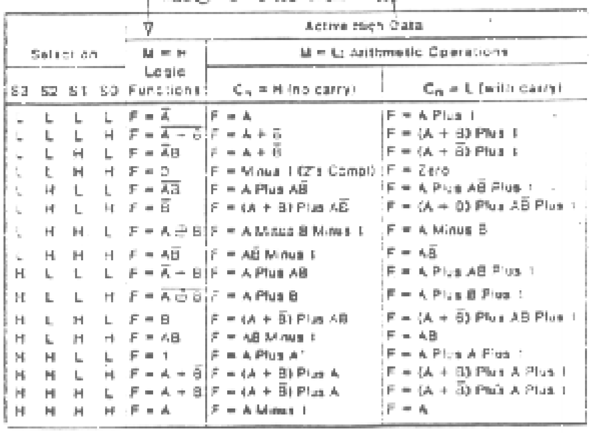
Function	**Micro-operation**
Addition	A plus B
Subtraction (2's complement)	A plus (B plus 1)
Logic AND	Ai . Bi
Logic OR	Ai + Bi
Logic XOR	Ai + Bi
Complement A	A← /A
Increment A	A plus 1

The outputs from the ALU are connected directly to the bus; thus they are buffered by tri-state buffers in the same way as a register's outputs.

Because it takes a number of clock cycles to set up the registers on the inputs of the ALU and to transfer the data from the ALU outputs into another register, it is slower than a direct register operation. For this reason some registers will have circuitry which performs the frequently required operations; the less frequently used operations will be done by the ALU.



mode control input



					Active High Data	
Selection				M = H	M = L; Arithmetic Operations	
				Logic		
S3	S2	S1	S0	Functions	C_n = H (no carry)	C_n = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A Plus 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) Plus 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ Plus 1
L	L	H	H	F = 0	F = Minus 1 (2's Compl)	F = Zero
L	H	L	L	$F = \overline{AB}$	$F = A$ Plus $A\overline{B}$	$F = A$ Plus $A\overline{B}$ Plus 1
L	H	L	H	$F = \overline{B}$	$F = (A + B)$ Plus $A\overline{B}$	$F = (A + B)$ Plus $A\overline{B}$ Plus 1
L	H	H	L	$F = A \oplus B$	F = A Minus B Minus 1	F = A Minus B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ Minus 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A Plus AB	F = A Plus AB Plus 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A Plus B	F = A Plus B Plus 1
H	L	H	L	F = B	$F = (A + \overline{B})$ Plus AB	$F = (A + \overline{B})$ Plus AB Plus 1
H	L	H	H	F = AB	F = AB Minus 1	F = AB
H	H	L	L	F = 1	F = A Plus A*	F = A Plus A Plus 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) Plus A	F = (A + B) Plus A Plus 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ Plus A	$F = (A + \overline{B})$ Plus A Plus 1
H	H	H	H	F = A	F = A Minus 1	F = A

*Each bit is shifted to the next more significant position.

M = L disables
the internal carry

Figure 6: 74181 ALU

Figure 13: Figure 1.3.7: 74181 ALU

3-1.4 The Control Unit

The control unit contains the hardware instruction logic. It decodes and monitors the execution of the instruction. It is to this combination of the control unit, the registers and the ALU interconnected by the internal bus that one calls the central processing unit.
The control unit acts as an arbiter since various portions of the system compete for the resources of the CPU. The activities of the CPU are synchronized by means of the system clock, which generates a regular pulse beat. All activities of the CPU are thus measured in clock cycles.

Self-Assessment 3-1

1.

SESSION 4-1 THE BASIC OPERATIONS OF A MICROPROCESSOR

4-1.1 Instructions and Data

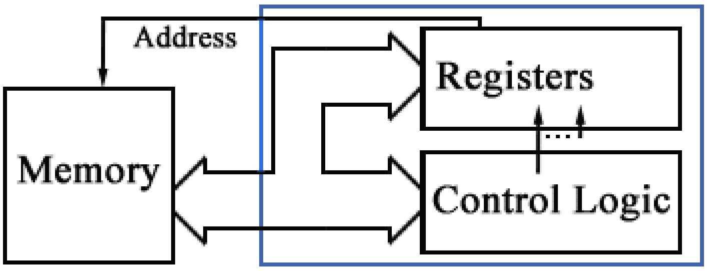


Figure 14: Figure 1.4.1: Simpilified Microprocessor Design Diagram

A simplified diagram of the microprocessor is given above. The binary information from the memory is applied to both the registers and the control circuitry, and is interpreted in different ways:

1. The binary pattern applied to the control logic is the instruction, and will cause the control logic to generate the necessary sequence of **micro-operations** that will accomplish the desired **macro-operation** indicated by the instruction.
2. The data information from the memory is transferred to and from the registers within the CPU, under control of the micro-operations generated by the control logic.

NOTE: Both instructions and data share a common path to and from the memory, although they are routed to different places within the CPU. This common data path is the **bus**'. Since both the instructions and the data are accessed from the memory in exactly the same way, there is thus no physical difference between them; the operation of the CPU is what determines whether a given word in memory contains an instruction or a data byte.

For an 8 bit CPU, the data bus width (which is the same as the memory width) is 8 bits, which implies that:

1. **Instructions** are 8 bits wide, and there are thus or 256 different combinations, or 256 possible instructions. Not all of these will be used; often only a smaller subset will be valid instructions. The remainder are called 'illegal instructions', and if the CPU attempts to treat them as instructions, the results will certainly be unpredictable and will often be disastrous.
2. **Data** which are transferred to and from the registers is in units of 8 bits. Thus if a register is 16 bits wide, its contents will require two bytes of memory to store.

4-1.2 The Control Logic

The control circuitry is in itself a small sequential circuit. Its function is to generate the control signals for the micro-operations to transfer and manipulate the data in the registers and ALU of the microprocessor. The inputs to the control circuitry are the bits of the instruction to be executed; the bits of each particular instruction will thus cause the control circuitry to cycle through a unique state sequence in order to generate the correct micro-operations to perform the instruction.

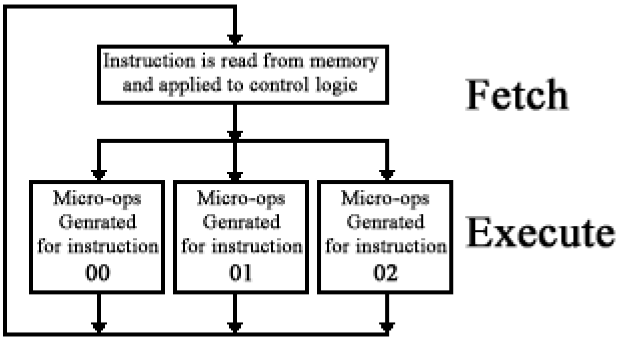

Instruction is read from memory and applied to control logic
Fetch
Micro-ops Genrated for instruction 00
Micro-ops Genrated for instruction 01
Micro-ops Genrated for instruction 02
Execute

Figure 15: Figure 1.4.2: The Control Logic Sequence

This sequence is shown in a very broad form here:
During the first part of the cycle, the instruction is read from memory and applied to the inputs of the control logic. Note that the micro-operations to achieve this are generated by the control logic itself; this first part of the cycle is always performed, and the control logic always generates the same micro-operations. It does not need to have any instruction applied to the inputs to do this. This first stage of the instruction processing is called the **fetch cycle**.
The second part of the instruction processing is called the **execute cycle**. The operations of the control logic are now dependent upon what instruction pattern is being applied to its inputs (i.e. on what instruction was fetched during the fetch cycle).

When the control logic has generated the required control signals to execute the instruction, it automatically performs another fetch cycle to retrieve the next instruction from memory and the cycle continues.

4-1.3 The Internal Structure of a Simple Machine

A diagram of the internal structure of a simple microprocessor is here:

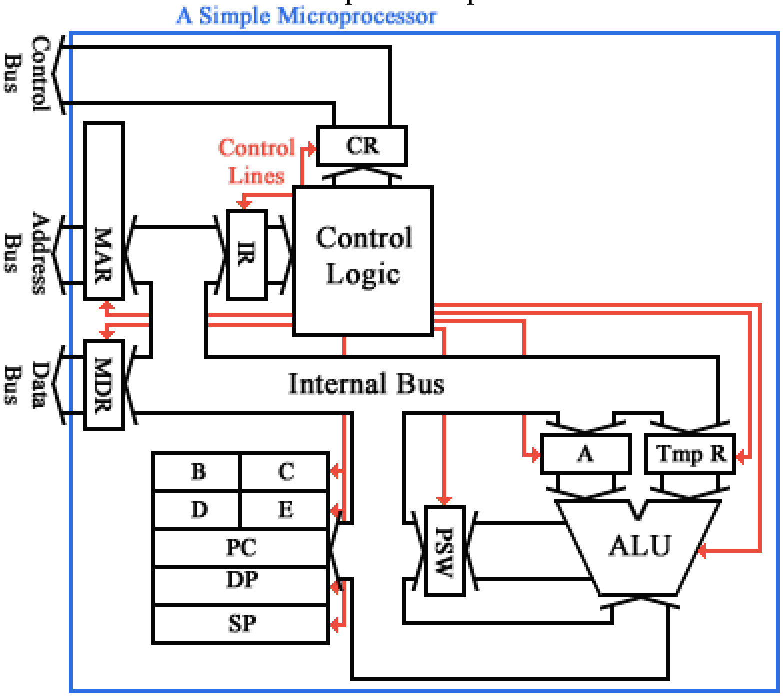


Figure 16: Figure 1.4.3: The Simple Machine

The registers are connected by a common bus and are consequently tri-state buffered. Note that the registers are not necessarily the same width, and consequently we assume that the bus is sufficiently wide to be able to transfer any register to any other register. Some of the registers have special functions:

1. **The Memory Address Register (MAR):** This register contains the address which is applied to the external memory array. It is the only way in which an address may be applied to the memory, which is otherwise isolated from the internal bus of the CPU. It is required because the address to the memory may come from a number of sources, and must be kept stable (unchanging) while the memory is read from or written to.
2. **The Memory Data Register (MDR):** This is the register into which words to and from the external memory array are transferred. It is used as a temporary storage location for these words only; the data in the MDR are never manipulated in any way, but only transferred to another register or to memory. It is a bi-directional register, since it may be loaded by both the CPU and the memory (unlike the MAR, which is never loaded from the memory).
3. **The Instruction Register (IR):** This is the register in which the instruction which is currently being executed is held. The IR drives the control logic directly; the opcode is placed in the IR during the fetch cycle, and is thereafter used by the control logic to generate the subsequent micro-operations to implement the instruction.
4. **The Program Counter (PC):** This is the register which contains the address in memory of the next instruction to be executed. It is also known as the Instruction Pointer (IP).

5. **The Accumulator (A):** the register on which most of the data manipulation (arithmetic, logic operations) take place. It drives one input of the ALU directly.
6. **The Control Register: (CR)** this is loaded only by the control logic, and is used to provide the necessary control signals to allow the memory and other I/O devices to be read to or written from. It allows the control logic to synchronise the data flow between the memory and the CPU.
7. **The Data Pointer (DP):** a register used to contain the address of any operand data that is required by the instruction. It is normally set up in the course of the opcode or operand fetch cycles, and is used during the execution cycle; its contents are loaded into the MAR during the instruction execution.
8. **The Stack Pointer (SP):** this is a dedicated register used to point to the top of the system stack.
9. **Processor Status Word (PSW):** is a word that stores the flags that indicate the status and error codes associated with the processor.
10. **The General Registers (B, C, D, E):** these registers are the general purpose working registers of the CPU. They are assumed to be the same width as the accumulator.

NOTE: In this simple machine, we assume that the PC has circuitry to automatically increment the register contents, and the stack pointer has circuitry to both increment and decrement its contents. The relevance of this will be seen later.

4-1.4 Microprocessor Instructions

Each instruction to the CPU will involve the processing or manipulation of data within the CPU in some way - transferring, adding, comparing etc. As such, each instruction must contain the following five pieces of information:

1. **The operation to be performed** (e.g. addition, data transfer, load, store etc.). This part of the instruction is referred to as the 'opcode'. A typical microprocessor will have a fairly limited set of primitive operations available - 16 or fewer.
2. **The source of the data** or where the data is to be found. If the data is in memory, then the address must be specified; if the data is elsewhere, its location must be given in some other way.
3. **The source of the second data item**, for those instructions that require it. (E.g. addition requires two data items; complementing requires only one).
4. **The destination of the resultant data** after the operation has been performed.
5. Where the **next instruction**, to be executed after this one is complete, is to be found.

Clearly this cannot be contained in a single (typically 8-bit) memory location, and consequently it is necessary to reduce the instruction length in some way. Each of the five pieces of information cannot be omitted, so the reduction is achieved by making assumptions for some of them.

1. Requirement (5) may be omitted by assuming that the next instruction which is to be executed follows immediately after the current instruction in memory - in the next sequential memory location.
2. Requirements (2) to (4) can be eliminated by making use of internal registers in the CPU to contain the data (or the address of the data) and making their use implicit in a particular instruction. This increases the number of possible

instructions, but by judicious choice it is possible to keep the total number of instructions below 256 and thus representable by a single 8-bit byte.

As an example, the 8085 instruction:
ADD C (instruction = 81H, or 10000001 binary)
Specifies that:

1. the operation is addition
2. the source of the first data byte is the accumulator (implied)
3. the source of the second data byte is register C (implied)
4. the destination is the accumulator (implied)

The complete instruction thus takes only 8 bits to specify and may thus be contained in a single memory location.

Even after these reductions have been achieved, it is still necessary for some instructions to have information specified in addition to the opcode. Such an instruction will typically involve a memory access, and thus is required to give the address of the memory location being accessed. In this case, it is necessary to use more than 8 bits to specify the instruction; such an instruction is called a **multibyte instruction**.
The first byte of a multibyte instruction is the opcode. The additional bytes which give the extra information form what is called the operand. The whole sequence of bytes is the instruction.

4-1.5 Microcycles

The execution of each instruction in the CPU is made up of a number of smaller operations, called micro-operations or **microcycles**. Each microcycle takes one clock cycle to perform, and represents one data transfer operation or one register manipulation operation. It is represented by the following notation:

Data transfer operation:	**destination ← source**
For example:	MAR ← PC
	IR ← MDR
Register Manipulation Operation:	**destination <- expression**
For example:	PC ← PC + 1
	SP ← SP - 1

If it is necessary to transfer data to or from memory, then this cannot be done by a single microcycle. The external memory device is regarded as a series of registers, like the internal registers of the CPU; however, before one of these registers (words) can be read or written, it is necessary to indicate which register is to be affected. This is done by first placing the word's address in the Memory Address Register. This then causes the appropriate memory word to be enabled and 'connected' to the Memory Data Register. The data may then be transferred between the MDR and the addressed word (the direction depending on whether a read or write is taking place). This data transfer must occur at least one clock cycle after the address of the word was written to the MAR, to give the memory time to stabilise. These microcycles are represented as follows:

MAR ← source register
MDR ← memory (MAR)

Each microcycle which involves a transfer of data between two of the internal registers of the CPU uses the internal bus; consequently only one such microcycle can occur during a single clock cycle. If a microcycle involves register manipulation, however, or uses the external memory or address buses, then it can occur at the same time as a register transfer.
As an example, take the instruction "Add the contents of the accumulator to the contents of memory location 2100, and leave the result of the addition in the accumulator".

Assuming that the opcode of this instruction is A2 (hex), one operand is required (the address of the memory location - in this case 2100). Assume that the operand is stored in the next two bytes after the opcode (the address is 16 bits long and the memory locations are only 8 bits, hence two bytes of memory must be used). Also assume that the least significant byte of the address (00) is stored at the lower address, i.e. the byte after the opcode.

The memory locations and the connection to the CPU will appear as follows:

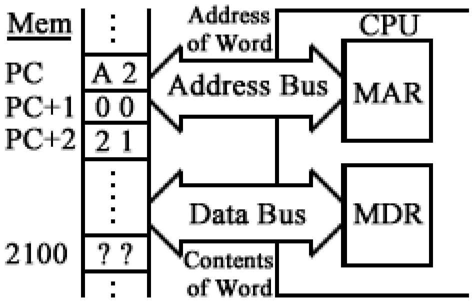


Figure 17: Figure 1.4.4: Memory Location and CPU Connection

The microcycles which would be involved on the simple machine illustrated in 4-1.3 would be:

Cycle	Micro-operations	Description
I	MAR ← PC	Opcode address
IIa	PC ← PC + 1	PC points to the 1st operand
IIb	MDR←memory(MAR)	Place opcode into MDR
III	IR ← MDR	Transfer opcode to IR
IV	MAR ← PC	Operand address into MAR
Va	PC ← PC + 1	PC points to 2nd operand
Vb	MDR←memory(MAR)	Get 1st operand byte
VI	DP(L) ← MDR	Put into LSByte of DP
VII	MAR ← PC	2nd operand byte address to MAR

VIIIa	PC ← PC + 1	Point to next opcode
VIIIb	MDR←memory(MAR)	Get 2nd operand byte
IX	DP(H) ← MDR	Put into MSbyte of DP
X	MAR ← DP	Address of operand data into MAR
XI	MDR←memory(MAR)	Get operand data byte
XII	tempR ← MDR	Put into tempR of ALU
XIII	A ← A plus tempR	Use ALU to add A and tempR

Note that four machine cycles were used to perform this instruction. The first three are fetch cycles, since bytes that are part of the instruction (opcode, operands) are fetched from memory so that the instruction can be subsequently executed. The PC is used to point to the next byte to be fetched, and is incremented by 1 automatically after every memory fetch.

1. The PC is incremented during the same microcycle as the data is transferred to the MDR from the memory. This is because the PC has logic built into it to increment the contents, and thus does not require the internal bus or ALU. Note that the incrementing of the PC must occur in the cycle following its transfer to the MAR - it cannot be incremented and transferred at the same time.
2. A total of 13 clock cycles are used. Each clock cycle indicates one micro-operation, or internal register transfer.
3. The first fetch (the opcode fetch cycle) is the same for all instructions. Thereafter any further fetch cycles are dictated by the opcode itself, which is now in the IR and influences the operation of the control logic.

Summarizing: The whole cycle is an **cycle**'. This is composed of two sections:

1. Fetch the instruction, and any necessary operands, using one or more fetch cycles. The PC is incremented automatically after every fetch cycle.
2. Execute the instruction.

Self-Assessment 4-1

1.

Learning Track Activities

Summary

Unit Summary

1.

Key terms in Unit

Review Question

Unit Assignment – 1

- **Discussion Question:**
- **Reading:**
- **webActivity:**

Unit 2
MEMORIES

Introduction

When a computer program is operating it needs to keep its data somewhere. There may be some registers (such as “Acc” in MU0) but these are not usually enough and a larger memory (or “**store**”) is required.

The computer program itself must also be kept somewhere. The earliest programmable devices were weaving looms. In 1804 Joseph Marie Jacquard invented the Jacquard Loom in Lyon. This used a sting of punched cards as a program; a binary hole/no hole system allowed complex patterns to be woven with the cards being advanced as the loom ran. Many later devices have also used this concept, notably the pianola or player piano (1895).

Learning objectives

Unit content

Session 1-2: Memory

1-2.1 The stored program concept
1-2.2 Addressing
1-2.3 Address Decoding
1-2.4 Commodity Memories
1-2.5 Using Memory Chips
1-2.6 Timing
1-2.7 The Reality of Memory Decoding
1-2.8 Filling the Memory Map
1-2.9 Memory Map Details
1-2.10 A separate I/O address space
1-2.11 Endianness
1-2.12 Memory Hierarchy

1-2.13 Caches
1-2.14 Harvard Architecture
1-2.15 Read-Only Memory (ROM)
1-2.16 Current Memory Technology
1-2.17 Punch Cards, etc.
1-2.18 ROM/PROM
1-2.19 More Electrostatic Memories
1-2.20 Magnetic Memories
1-2.21 More Magnetic Memory
1-2.22 Optical Memories

Session 2-2 Address Decoding Strategies

2-2.1 Full Address Decoding
2-2.2 Partial Address Decoding
2-2.3 Designing Address Decoders
2-2.3.1 Address Decoding with Random Logic
2-2.3.2 Address Decoding with m-line-to-n-line Decoders
2-2.3.3 Address Decoding with PROM
2-2.3.4 Address Decoding with FPGA, PLA and PAL

SESSION 1-2: MEMORY

1-2.1 The stored program concept

The concept of a ‘stored program’ is attributed to John von Neumann. Put simply it says: **“Instructions can be represented by numbers and stored in the same way as data.”**

Thus a bit pattern 01000101 might represent the number 4516 or the letter “E” as data but it could also be used to tell a processor to perform a multiplication. This has led to the so called “von Neumann” architecture which is followed by almost all modern computers where a single memory holds values which can be interpreted as data or as instructions by the processor.

Whilst it is rare that the same memory locations are used as instructions and data it does happen.

The most notable case is when a program is loaded and executed: the loader fetches words from an I/O device (e.g. disc) which it treats as data and puts into memory; the same values are interpreted as instructions when execution starts.

The address space of a computer such as our MU0 will normally contain **Random Access Memory** or **RAM**. RAM is memory where any location can be used at any time. MU0 has a 12-bit address bus and so can address up to 4Kwords of memory, each word being 16 bits wide. As this is a small memory by modern standards it is likely (now) that all the words would be implemented (although a few locations must be reserved for I/O or there would be no way to communicate with the computer).

Back in 1948 4Kwords (64 Kbits) would have seemed a very large memory which would require many memory devices to fill. By the end of the 20th century the largest RAM devices reached 256 Mbits so one device could provide for 4000 MU0s!

To come more up to date we shall use the ARM address model instead. ARM produces byte addresses and has a 32-bit address space, which allows the addressing of 232 separate bytes. However as instructions and most data are 32-bits (4 bytes) wide it is normal to read or write four bytes in parallel. We will therefore regard the ARM as having a 30-bit address space (the last 2 bits can specify one of the four bytes).

Thirty address bits allow the addressing of 230 separate words or 1 Gword. This is larger than contemporary devices (256 Mbits ==> 8 Mwords); it is therefore necessary to be able to map several memory devices into the address space. Not all these devices may be fitted in every system; whether the memory space is fully populated or not depends on the needs and the budget of the owner.

Definitions and usage

- RAM – Random Access Memory; by convention (& slightly incorrectly) used for memory which is readable and writeable. Most modern RAM 'forgets' when the power is turned off.
- ROM – Read Only Memory; usually a variation of RAM which cannot be written to; used to hold fixed programs. As it cannot be written to its contents must be permanent.

In addition there are forms of serial access memory (such as magnetic tape, networks).

1-2.2 Addressing

Within the CPU it is common for several things to happen in parallel; the memory only performs one operation at once.

This operation requires the answers to the questions:

- Do what? – Control (read or write)
- With what? – Data
- Where? – Address

Because only one operation is happening at a time the control signals and the data bus can be shared over the whole memory.

The address bus provides a code to specify which location is being used ("addressed").

Some definitions:

- Byte – now standardised as eight bits.
- Word – the 'natural' size of operands, which varies from processor to processor (16 bits in MU0, 32 bits in ARM). Usually the width of the data bus.
- Nibble – four bits or half a byte (sometimes "nybble")

- Width – the number of bits in a bus, register or other RTL block.
- Address range – the number of elements which can be addressed.
- Type – what the data represents. This is really a **software** concept in that the hardware (usually) does not care whether a word is to be *interpreted* as an instruction, an integer, a 'float', an address (pointer) etc. This may, however, influence the **size** of the transfer (byte, word, etc.).

The figure shows part of a memory; four **words** of four bits each are depicted (although the decoders imply that another four words are omitted). The bits in each word are stacked vertically; note that the write enables and the read enables (to the tristate buffers) are common across each word. The words can be made as wide as required in this way.

The width of the memory is normally the same as the width of the CPU's datapath, but it may not always be so; for example some high-performance processors use wider memory so that they can fetch two (four, …) instructions simultaneously.

1-2.3 Address Decoding

An address is coded as a binary number to minimise the number of bits/wires required.
The memory requires a word select as a "1-of-N" code.
The conversion is performed by an **address decoder**.

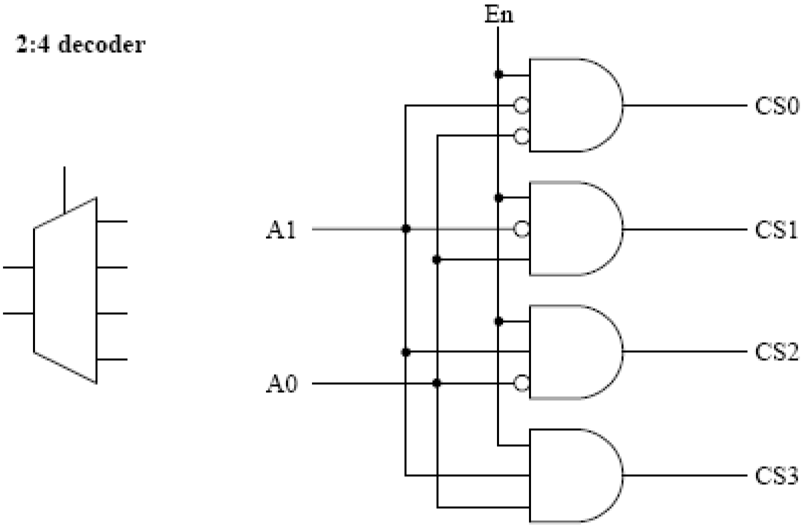


Figure 18:

It is often both inconvenient and impractical to decode the entire address bus in a single decoder. Instead a hierarchical approach is used:

Here the first decoder is used to enable one of the next set of decoders to give a 6-to-64 decoder (not all of which is shown). This can be extended further if required. In practice the decoders need to be very large, but the last stage of decoding (which could be decoding around 20 address lines!) is built into the memory device. The designer only needs to produce the equivalent of the first level of the address decoder which selects which memory device is active. This is described in more detail later.

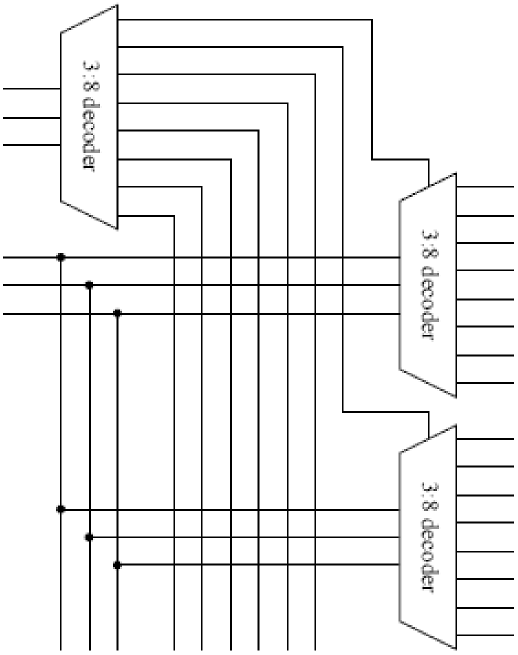


Figure 19:

1-2.4 Commodity Memories

All von Neumann computers need memory. Sometimes their needs are small – an embedded controller operating a central heating system probably needs only a few byte of RAM – but others need many megabytes. Even a heating controller may need a kilobyte or so of program memory.

Small memories (a few Kbytes) are often constructed on the same chip as the processor, I/O etc. Large memories will need one or more separate, dedicated devices.

Even using modern semiconductors to make memory there are several options:

- D-type flip-flops
  - Convenient for synchronous logic (e.g. FSMs)
  - Very large area per bit
- Transparent latches
  - Okay for logic but not as convenient
  - Smaller than D-types, but still large
- SRAM
  - Small area per bit
  - Need (shareable) interface logic
  - Simple to use
- DRAM
  - Very small area per bit
  - Need considerable interface logic
  - Many awkward timing constraints

The reason several types of memory exist is that the cost trade-offs vary according to the system requirement. For example DRAM is the most area efficient but it is slower than

SRAM and requires more support logic and can be more expensive for memories below a certain size.

SRAM is Static RAM. It has a simple interface, reasonable storage density and is fast to access. It consumes little power when not in use. Typical application would be the cache memory on a PC motherboard (fast) or the memory on a mobile ’phone (low power).

DRAM is Dynamic RAM. It has a complex interface because of its timing characteristics. It also ‘forgets’ over time (times in milliseconds) so needs to be constantly read and rewritten, consuming power even when not being actively used. It is slower than DRAM although average speed can be improved if ‘bursts’ of consecutive locations are read or written. Its big advantage is its that it gives very dense storage. A typical application would be the ‘main’ memory of a PC (large, cost-effective).

In practice both the SRAM and DRAM need other circuits (such as amplifiers) to interface them to computational circuits. However the overhead is small because a few amplifiers can be shared by many thousands of bits of store.

A bit of ROM will be roughly the same size as a bit of DRAM or SRAM, depending on the technology employed.

When building a system the **cost** is related to the number of silicon chips and their size. Thus if D-type flip-flops were used the memory which could be implemented at a given price would be much smaller (i.e. fewer bits). Cost is extremely important in system design!

Multi-port memory

The memories described here are single port memories: they have a single data bus which is used *either* for reading or writing during any cycle. It is also possible to build multiport memories which allow two (or more) operations during the same cycle – typically on different locations!

SRAM and DRAM sizes are limited by the wiring density. Adding an extra port means adding an extra (decoded) address wire and an extra data wire. As this gives less than half the bit densities it is not normally worthwhile and multiport memories are uncommon, though not unheard of.

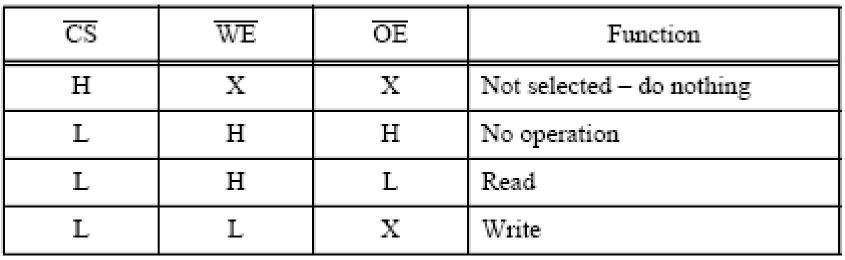
1-2.5 Using Memory Chips

The memory device shown is a 628512. This is a 4Mbit SRAM chip (memory sizes are normally quoted in **bits**) organised as 512 Kwords of 8 bits each.

It therefore requires nineteen address lines and eight data lines; together with its power supplies {Vdd, Vss} and three control signals these occupy all the pins on a 32 pin DIL (Dual, In-Line) package.

The following table defines the memory chip’s behaviour.

Table 7:



$\overline{CS}$	$\overline{WE}$	$\overline{OE}$	Function
H	X	X	Not selected – do nothing
L	H	H	No operation
L	H	L	Read
L	L	X	Write

Points to note:

- All the **control** signals are **active low**
- If the chip is not selected (CS = H), nothing happens
- Write enable overrides read operations
- The **data** bus is **bidirectional** (*either* read *or* write – saves pins)

The CS signal is used to indicate that this particular device is in use. If a larger memory is required an external decoder can be fitted so that only one memory chip is enabled at once. In this way several memory chips can be wired with all their pins in parallel except for CS.

Observation
Because it uses the same pins for read and write operations it does not actually matter what order the address and data pins are wired in. The user does not care what the address of any particular location is as long as its address does not vary.

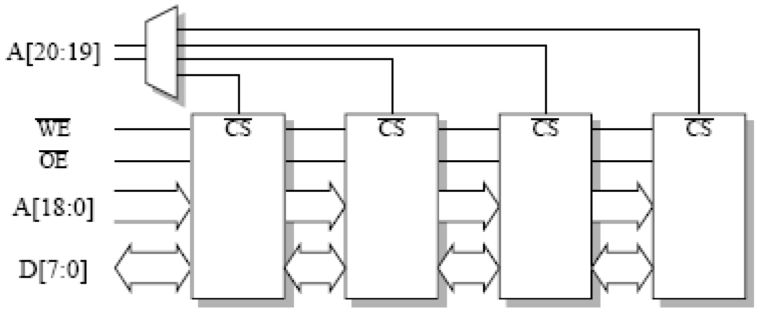


Figure 20: 2 Mbyte memory using 512Kbyte chips

1-2.6 Timing

Memory holds state. It can be written to. When writing it is important that the correct data is written to the correct location; it is also important to ensure that no other memory locations are corrupted.

In the MU0 model described earlier the memory was controlled by *read* and *write* control signals and it was assumed that the processor clock would control state changes. A real memory device often has no clock input!

A simple SRAM has the same timing characteristics as a **transparent latch**. If the chip is selected (CS=L) and write enabled (WE=L) then the data inputs will be copied into the addressed location. It is important that the address is **stable** during the write operation; if it is not, other locations may also be affected.

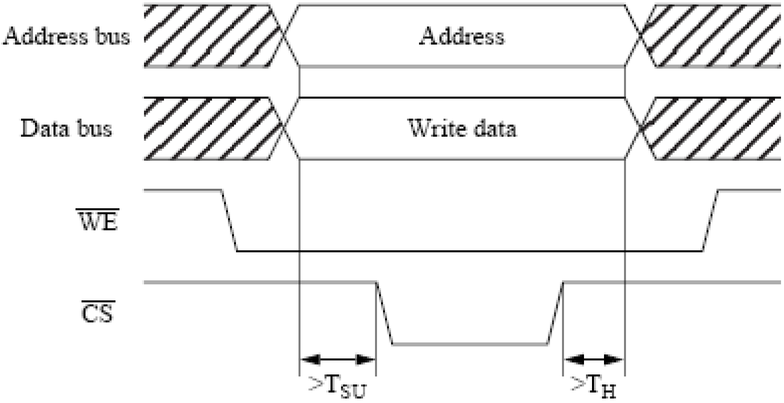


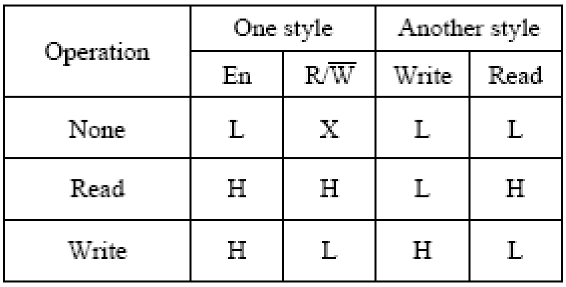
Figure 21:

There are **set-up** and **hold** time requirements for the address and data values around the write cycle. (The set-up time is normally greater to allow for the address to be decoded.)

The actual **write strobe** is a logical AND of the write enable and chip select signals; both must be active for data to be written. The timing diagram shown above is therefore only one possible approach to strobing the memory. Another approach could use WE as the **timing signal**.

Different processors (& different implementations) encode timing differently. That's okay, as long as timing is included somewhere.

Table 8:



Operation	One style		Another style	
	En	$R/\overline{W}$	Write	Read
None	L	X	L	L
Read	H	H	L	H
Write	H	L	H	L

Note that this is not *essential* for read operations, because they do not change the state of the memory; it does no harm though.

1-2.7 The Reality of Memory Decoding

In the foregoing it is assumed that each address corresponds to one memory 'location'.
In a 'real' memory system this is often not the case. For example an ARM processor can address memory in 32-bit words or 8-bit bytes (or 16-bit "halfwords) and the memory system must be able to support all access sizes.

Addresses are decoded to the minimum addressable size (in this case bytes). Addressing a word requires fewer address bits.

Thus the least significant bit used by the address decoder is A[2]; A[1] and A[0] act as **byte selects**, which will be ignored when performing word-wide operations. Of course the bus must also carry signals to specify the transfer size.

Byte accesses
Notice that when the processor reads word 00000000 it receives data on all its data lines (D[31:0]). When the processor reads byte 00000000 it receives data only on one quarter of the data bus (D[7:0]); furthermore if the processor reads byte 00000001 it uses a *different* subset of the data bus (D[15:8]). The processor resolves this internally by shifting the byte to the required place (an ARM always moves the byte to the eight least significant bits when loading a register).

The same is true when writing quantities of less than a full word – the data must be copied onto the appropriate data lines.

When reading a byte it is possible to 'cheat' by reading an entire word from memory and ignoring the bits that are unwanted. This works because reading memory does not affect its contents. However when writing it is essential that only the byte(s) to be modified receive a WE signal or other bytes in the same word will be corrupted. This would be a Bad Thing.

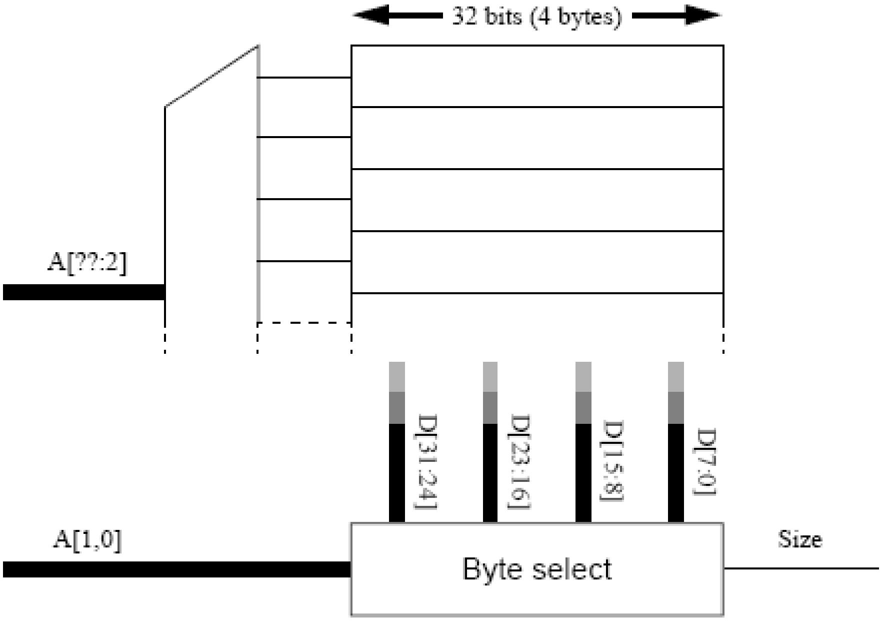


Figure 22:

1-2.8 Filling the Memory Map

The ARM processor

- has a 32-bit word length.
- produces a 32-bit *byte* address.
- can perform read and write operations with 32-, 16- and 8-bit data.

The normal design for the memory system would therefore be a space of 230 words (byte addressing, remember) of 32-bits each. Let's see how this could be **populated**, using the RAM chips described above.

The RAMs are 8 bits wide; therefore four devices are required to make a 32-bit word. This then gives 512 Kwords of memory. We can then repeat this arrangement another 2048 (=211) times to fill the address space, using the appropriate decoder circuits.

Of course the 8192 RAM chips required will be expensive, will occupy a large volume and use a lot of power (thus generating unwanted heat) and it is unlikely that we really need 4 gigabytes of memory!

The usual alternative with a large address space is to make it sparsely populated; this saves on memory chips and also simplifies the decoder circuits. Let's say we need only 1 Mword of RAM, as in the figure opposite.

(In this figure the ability to perform 16-bit 'halfword' transfers has been omitted.)

- Here the memory is 32-bits wide which requires four, 8-bit wide chips per 'bank'.
- Two banks of memory provide 1Mword.
- The two least significant address lines (A[1:0]) are used as a byte select.

These are ignored if a word transfer is performed.

- The next nineteen address lines (i.e. A[20:2] are connected to all the RAM chips. Note that the signal A[2] will be wired to the pin A0 on the RAMs (and so forth) because the RAM address is the word address, not the byte address. A[1:0] are used as a byte address within the word.
- A[21] is used here to select between the two banks of RAM. The last stage of the decoder is shown as explicit gates which drive the individual chip selects. (NAND gates provide for the fact that CS is active low.)
- The chip selects are the only signals which are distributed on a 'one per-chip' basis; other signals can be broadcast across many/all devices. This simplifies the wiring on the PCB1.
- A[29:22] are ignored!
- The RAM *region* select signal is produced from the most significant address bits; here RAM is selected when A[31:30] = 01.This means RAM occupies addresses 40000000-7FFFFFFF inclusive. The lowest region is reserved for ROM because the ARM starts executing at address 00000000 when the power is switched on.

1-2.9 Memory Map Details

The memory map described (which is just one possible example) shows many of the basic properties found in real systems.

- The memory is coarsely divided into areas with different functions.
    - m Areas may contain different types of memory or different technologies that run at different speeds. For example the I/O area may be designed to cycle more slowly (i.e. more clock cycles) than the RAM.
    - Some integrated CPU devices may provide such decoders 'onboard'.

- Some area are left ‘blank’.
    - The previously described decoder does not use one of the area selects.
    - Writing to such areas has no effect.
    - Reading from such areas could return *any* value (i.e. it is **undefined**)
- Some physical devices can appear at several different addresses.
    - This is due to ignoring some address lines when decoding.
    - Fewer levels of decoding reduce cost and increases speed.
    - This is known as **aliasing**.
- The I/O space is unlikely to be full.
    - There will be both undecoded and aliased locations.
    - In most cases peripheral devices will be byte-wide so not all bits in the word will be defined. When reading peripherals it is important to **mask** out the undefined bits.
    - Peripheral devices are sometimes unlike memory in that reading an address will not return the same value that was written to that address.
    - Input ports, by definition, are **volatile** in that their value can change without processor intervention.

1-2.10 A separate I/O address space

The memory map shown here includes space for ROM, RAM and I/O peripherals. I/O access patterns are somewhat different from memory accesses in that they are much rarer and often come individually (as opposed, for example, to instruction fetches which run in long sequences).

Some processor families, a notable example being the x86 architecture, provide a completely separate address space which is intended for I/O. If this is used it leaves a ‘cleaner’ address space just for ‘true’ memory. The programmer can get at this space by using different instructions (e.g. “IN” and “OUT” replace “LOAD” and “STORE”) which are usually provide only limited addressing modes and, possibly, a smaller address range.

The hardware view typically uses the same bus (with an added address line *IO/mem*). The hardware may also slow down bus cycles automatically in the expectation that peripheral devices are slower than memory.

Note that the system designer is not compelled to use these spaces in this way!

1-2.11 Endianness

Generically “endianness” refers to the way sub-elements are numbered within an element, for example the way that bytes are numbered in a word. By convention the bytes-in-a-word definition tends to dominate, thus a “big-endian” processor will typically still number its bits in a little-endian fashion (see slide).

This can get pretty confusing. If it’s any consolation the numbering schemes used to be worse!

Little endian addressing
Pick a word address; say 00001000, in a 32-bit byte-addressable address space. Let’s store a word (say, 12345678) at this address.

- Address 1000 contains byte 78
- Address 1001 contains byte 56
- Address 1002 contains byte 34
- Address 1003 contains byte 12

i.e. the least significant byte is at the lowest address. This has the effect that, if displayed as bytes, a memory dump would look like:

```
        00001000 78 56 34 12
```

i.e. the bytes appear reversed (because higher addresses appear further to the right).

If a byte load was performed on the same address the result would be: 00000078

Big endian addressing
Using the same word address (00001000) for the same word (12345678).

- Address 1000 contains byte 12
- Address 1001 contains byte 34
- Address 1002 contains byte 56
- Address 1003 contains byte 78

i.e. the least significant byte is at the lowest address.

This has the effect that, if displayed as bytes, a memory dump would look like:

```
        00001000 12 34 56 78
```

If a byte load was performed on the same address the result would be: 00000012

Choice of endianness
Some processors are designed to be little endian (x86, ARM, …), others to be big endian (68k, MIPS, …). There is no particular rationale behind this. Most modern workstation processors allow their endianness to be programmed at the memory interface.

1-2.12 Memory Hierarchy

Bottom line:
for a given price

- big memory = slow memory
- small memory = fast memory

If a programme has to run from ‘main’ memory it will only run at the speed at which its instructions can be read – maybe 10x slower than the processor can go. However in reality typical programmes show a great deal of **locality**, i.e. they spend maybe 90% of their time using perhaps only 10% of the code.

If the critical 10% of the code is placed in a small, fast memory then the performance of the overall programme can be significantly increased without the expense of filling the address space with fast memory.

This is exploited extensively in high performance systems. Depending on the implementation it may be known as **caching** or **virtual memory**1; the principle is the same in each case.

A typical PC will probably have four levels in its memory hierarchy:

- An on-chip cache, integrated onto the processor chip (**SRAM**)
- A much larger secondary cache on the motherboard (**SRAM**) (sometimes erroneously referred to as "*the* cache")
- The 'main' memory – usually many megabytes of some cost-effective **DRAM**
- The virtual memory space which is kept on a hard disc (**magnetic**)

There may be more levels than this though!

Although the principle of locality is used at each level of the hierarchy the process of choosing the "**working set**" (the elements to store) is often implemented differently: it is sometimes done by hardware and sometimes by software and may be static or dynamic.

In the future it is likely that the technology will evolve but it is unlikely that memory hierarchies will disappear.

The Register Bank

Unlike MU0 modern processors usually have a significant number (ARM has 16, MIPS has 32, …) of registers forming a register bank (sometimes called a register 'file'). These registers are used for operands for and results from the current set of calculations.

Although they are not addressed in the same way as memory, the registers can be regarded as the topmost level of the memory hierarchy. The management of what is stored in the registers is done 'manually' by the compiler or the programmer directly and is – of course – specified explicitly in the object code.

1-2.13 Caches

Two observations:

- Large memories (at an economical price) tend to be slower than small ones.
- A program spends 90% of its time using 10% of the available address space1.

No one has said that the memory has to be homogeneous; it is quite possible to have memories of different speeds at different addresses. If you can organise things so that the 10% of the address space which is frequently used is in fast memory then you can get startling improvements at relatively small cost.

In some circumstances this is possible. In **embedded controllers** the software is **fixed** and the programmer can **profile** and arrange the code to exploit different memory speeds.

In **general purpose** machines (e.g. PCs) the code is **dynamic** (a posh way of saying you run lots of different programs) and those programs are designed to run on different machine configurations. Profiling is not a great help here.

A cache memory adapts itself to prevailing conditions by allowing the addresses it occupies to change as the program runs. It relies on:

- **Spacial Locality** – guessing that if an address is used others nearby are likely to be wanted.
- **Temporal Locality** – guessing that if an address has been used it is likely to be used again in the near future.

Two illustrations will illustrate why this often works:

- Instructions are (usually) fetched from successive addresses and loops repeat sections of code many times.
- Many data are held on a stack which uses a fairly small area of RAM repeatedly.

Sufficient to say this works very well. In 'typical' code a cache will probably satisfy ~99% of memory transactions.

Cache Hierarchies

Caches work so well that it is now common practice to have a cache of the cache. This introduces several **levels** of cache or a cache hierarchy.

The first level (or "**L1**") cache will be integrated with the processor silicon ("**onchip**").

There will be a second level of the cache ("**L2**"); this may be on the PCB, on the CPU chip or somewhere in between such as the integrated processor module.

Further cache levels are also possible; "L3" is increasingly common in high-performance systems.

1-2.14 Harvard Architecture

The term "Harvard architecture" is normally used for stored program computers which separate instruction and data buses. This separation may apply to the entire memory architecture (as shown on the slide) or may be limited to the cache architecture (below).

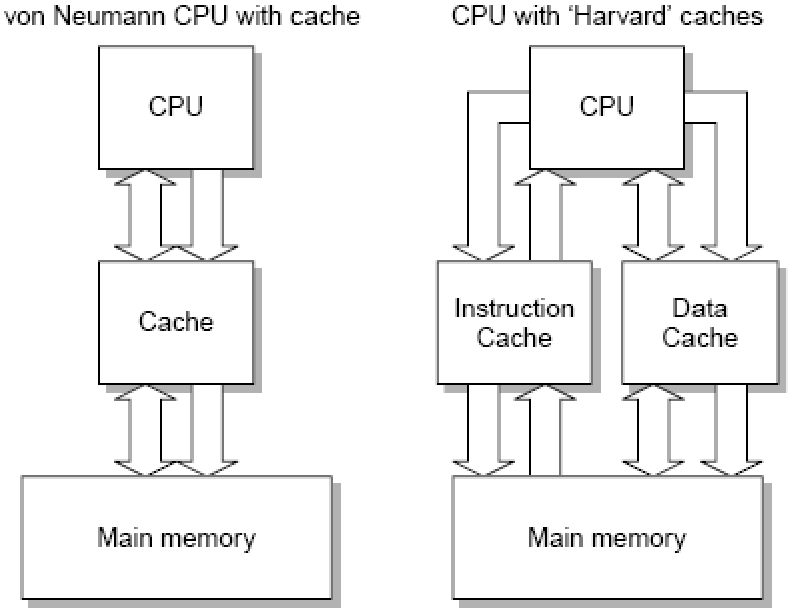


Figure 23:

You might like to identify and label the buses here. Where should the I/O be in each?

The Harvard architecture logically separates the fetching of instructions from data reads and writes (e.g. 'load' and 'store'). However its real **purpose** is to **increase** memory **bandwidth**.

Bandwidth is the quantity of data (number of bits) which can be transferred in a given time.

In a von Neumann architecture instruction fetches and data references share the same bus and so compete for resources. In a Harvard architecture there is no competition so instruction fetches and data reads/writes can take place in **parallel**; this means that the overall processing speed is increased.

The disadvantages of Harvard architecture are:

- the available memory is pre-divided into code and data areas; in a von Neumann machine the memory can be allocated differently according to the needs of a particular program
- it is hard/impossible for the code to modify itself (not often a problem, but can make loading programs difficult!)
- more wiring (pins, etc.)

Note: with a Harvard architecture the main memory may be completely divided in two. The parts need not have the same width or address range. For example a processor could have 32-bit wide data memory and 24-bit wide instruction memory.

Many **DSP**s (Digital Signal Processors) have more 'unusual' Harvard architectures.

1-2.15 Read-Only Memory (ROM)

ROMs are usually random-access memory devices. They use a similar IC technology to RAMs, with lower cost/bit than RAM.

They are:

- **read-only** which means their contents cannot be corrupted by 'accidents' such as bugs or crashes.
- **non-volatile** (i.e. they retain their information when power is removed).

Uses

- 'Bootstrap' programs.
- 'Fixed' operating system and application code.
- Logic functions (e.g. microcode, finite state machines).

Types of ROM

- **Mask programmed** ROMs are programmed during chip manufacture.
  - Cheap for large quantities.
  - Used in ASIC1 applications
- **PROM**s are '**Programmable**' after manufacture, using programming equipment.
  - Each individual IC is separately programmed (a manual operation).
  - Contents cannot be changed after programming.
- **EPROM**s are **Erasable** and **Programmable** (usually by exposure to strong ultra-violet light).

  - A technology in decline.
- **EEPROM**s are **Electrically Erasable**.
  - Currently one of the most popular ROM technologies.
  - Many can be altered ‘in-circuit’, i.e. without removal from the PCB.
  - They differ from RAM in that they require considerable time to alter a location (writes take >100x the read time).
  - Many devices also require ‘bulk’ erasure so that all or a large portion of the chip is ‘blanked’ before new values can be written.
  - Widely used for non-volatile store in consumer applications such as telephones, TV remote controls, digital cameras et al.
  - “**Flash Memory**” falls into this category.

The reprogrammable devices suffer a small but cumulative amount of damage each time they are erased/reprogrammed and are only guaranteed for a limited number (say, 100 000) of write operations.

Other Memories

So far we have treated “memory” as simply the **directly addressable** memory space (which is the usual interpretation of the term). However there are a number of other storage devices in use in a modern stored program computer.

One other form of store is the processor registers. In RISC processors it is usually clear that these form a separate, addressable ‘memory’ space: e.g. in an ARM “R7” means the “**R**egister store with address **7**” (not to be confused with the memory location with address 7).

Perhaps more obvious are magnetic storage devices such as **discs**. The primary function of a disc store is act as a **filing system**. In a filing system each file is a separate, addressable entity where the ‘address’ is the name of the file. File handling is beyond the scope of the processor hardware and is performed by specialist software, usually as part of an **operating system**. Files may be stored on local discs (i.e. on the machine which is using them) on elsewhere (e.g. on a networked **fileserver**); this should be transparent to the user.

Memory used as file storage has the following characteristics:

- Addressed in variable size elements (“files”)
- Addresses (“filenames”) variable length
- Address decoding done by software (“filing system”)

It is possible – with some extra hardware support – to make disc store to ‘stand in’ for areas of the address space not populated with semiconductor memory. This is a **virtual memory** scheme and will be described more fully in later courses.

Another type of addressable store uses addresses of the form: “http://www.cs.man.ac.uk/”

1-2.16 Current Memory Technology

SRAM

- fast
- truly random access
- relatively expensive per bit

DRAM

- significantly slower than a fast processor
- faster if addressed in ‘bursts’ of addresses
- medium cost per bit

Magnetic storage

- very slow (compared to processor speeds)
- variable in their access times (think of the mechanics involved)
- read/writeable only in blocks
- cheap per bit

Optical storage

- very slow (compared to processor speeds)
- variable in their access times (think of the mechanics involved)
- primarily (but not exclusively) read only
- extremely cheap per bit
- cheap to make as ROMs (think CDs, DVDs, …)

1-2.17 Punch Cards, etc.

Early references in weaving

- 1725: M. Bouchon used a pierced band of paper pressed against horizontal wires.
- 1728: M. Falcon suggested a chain of cards and a square prism instead of the paper.
- 1745: Jacques de Vancanson automated the process using pierced paper over a moving, pierced cylinder.
- 1790: Joseph-Marie Jacquard developed the mechanism which still bears his name.

Uses in computing

Analytical Engine (1837)
Certainly the earliest ‘use’ of punched cards was in Charles Babbage’s (1792-1871) design of the Analytical Engine (a mechanical digital computer). This was never built but was, in most respects, modern computer architecture with a processor (called the “mill”), memory and I/O. The analytical engine had, in fact, three types of card decks with “operation cards” (instructions), “cards of the variables” (addresses), and “cards of numbers” (immediates).

Hollerith machine (1884)
Not truly ‘computing’ as much as a counting (& accounting) machine the Hollerith machine revolutionised record keeping. With information on punched cards a machine could be ‘programmed’ to count all cards with certain sets of punched holes. This was first used for applications such as the US census in 1890; Hollerith cards were used extensively in early electronic computers and for other systems – they were familiar, everyday objects from the 1950s to the 1970s – and in use for some applications (such as voting) into the late 20th century.

In 1928 the standard card size increased from 45 to 80 columns (960 bits). In computing this was adopted as a line of text/program and was used as the width of a Visual Display Unit (VDU). This survives as the ‘standard’ page width.

Paper tape
Paper tape uses the same principle as punch card to store data. It has the advantage in density and is faster to feed through a reader. It is also not as easy to get muddled (or 'hacked') as a deck of cards because it cannot be shuffled.

Conversely the ability to edit programs by adding, deleting and substituting punched cards could be very useful.

Editing paper tape is difficult. One of these difficulties has left a trace in the ASCII character set where the character 7F (DEL or 'delete') is separated from the other 'control' characters; this code is used because it was represented by all the holes punched out (ASCII is a 7-bit code) and so could be used to overwrite mistakes.

Similarly the character 00 (NUL) is used as a 'no operation' in order to allow an indefinite length of unpunched "leader" on the reel of tape.

'Millipede'
A possible new technology, using a microscopic punched card; hunt out your own references.

1-2.18 ROM/PROM

Some ROMs retain data as physical wire connections. In mask programmed ROMs these wires are fixed at manufacture. In the (currently defunct) fuse PROM technology the wires were fuses which – if not required – were overloaded and 'blown' during programming.

Older technologies used matrices of diodes on PCBs for a similar effect.

Delay lines
A delay line is a device which exploits the 'time of flight' of bits in transit. It would be possible, for instance, to do this optically but sound – which travels more slowly – gets more bits into a short space.

Delay lines are **dynamic** store in that data must be read, regenerated and rewritten continuously. Clearly random access is not possible as data can only be read or written as the required 'location' circulates through the electronics. Access to a given 'memory' is, of course, strictly bit-serial.

Many early electronic computers – e.g. ENIAC (1946), EDSAC (1949) – used **mercury delay lines** (or "tanks") as their main store. A typical 5¢ delay could hold about 1 Kbit. It was folded up for convenience (rather like a bassoon). Mercury delay lines were originally developed in the 1940s for radar applications.

Mercury is a good acoustic conductor but is rather expensive (and heavy). A more convenient system was sought. The solution was the **magnetostrictive delay** line.

Magnetostriction is a stress induced in a magnetostrictive material (such as nickel) when it us subject to a magnetic field. (A magnetic field is, of course, generated by a flowing electric current.) This was translated into torsional (twisting) waves on a long rod. The process is reversible, so the bit stream can be detected again at the far end and neoprene buffers damp out any excess energy.

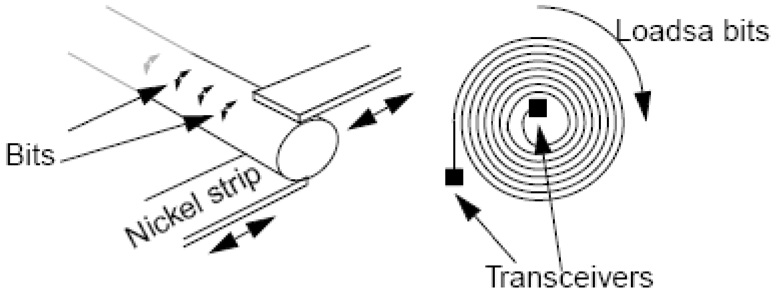


Figure 24:

As the system runs at high frequency (~1Mbit/s) the 'rod' could really be quite a light wire which could be loosely coiled onto a circuit board. Single lines of up to 100¢ were made which could store up to 10 Kbits.

Electrostatic Memories

Williams Tube (1948)

The Williams Tube (more correctly the Williams-Kilburn Tube) was an early all-electrical storage device developed in Manchester. Its basis is a Cathode Ray Tube (CRT) similar to those used in televisions and computer monitors. Bits were stored as charge patterns on the phosphor screen. In effect some electrical charge was or was not planted at each point on the screen using an electron beam. The bits were read back by displacing these charges with another electron beam which caused a discharge into the screen; the discharge was picked up by a wire mesh across the screen's front.

The first Williams tubes could store 2Kbits – perhaps twice the contents of a mercury 'tank'. They offered the added advantage that the data could be viewed by the operator (although a second, parallel tube was needed because the actual store was enclosed).

Reading the data is destructive, therefore it was necessary to regenerate the charge and **refresh** the display. In any, case as charge tended to leak away, regular refreshing was necessary; the store was therefore '**dynamic**'. This was the store technology employed in the Manchester 'Baby', a computer which was really built as a memory test unit.

Dynamic RAM (DRAM) (1970s-present)

Instead of a glass and phosphor screen it is possible to store charge in a large array of capacitors. However making such an array was very expensive until it could be done on a single silicon chip. This is the principle behind DRAM.

The capacitors are accessed via a matrix of wires and switches which allow individual capacitors to be charged or discharged. Opening these switches (which are really transistors) isolates the cells. Closing the switches again allows the charge to escape, which can be sensed and amplified as a read operation.

Read operations s are destructive and therefore any data which are read must be rewritten afterwards. Also the capacitors are not perfect so charge gradually leaks away, therefore periodic refreshing is required – hence the name *dynamic* RAM.

Each bit store comprises one capacitor and one switch (transistor) and these can be made very small. It is therefore possible to fit many megabits on a single chip. This is why DRAM has remained the cost-effective choice for large addressable memories for several decades.

DRAM is customised and marketed in a number of guises such as EDO-RAM, SDRAM, Rambus etc.; all these use the same basic technology.

The Decatron
Not an electrostatic memory – indeed related to very little else – the decatron was a neon discharge tube with 10 anodes; the discharge could be jumped from one to another where it would remain following the ionisation path. This was a *decimal* memory cell which also acted as a display. Now a historical curiosity.

1-2.19 More Electrostatic Memories

EPROM (1970s-1990s)
If the charge leakage can be (effectively) eliminated and a DRAM read can be made non-destructive then the store would be even more useful. This is the principle behind EPROM (Erasable Programmable Read Only Memory) which is non-volatile – i.e. it retains its data indefinitely (even when the power is off).

This is done by adding a ‘floating gates’ to the memory transistors; these are ‘islands’ where charge can be stored which are insulated by (relatively) thick glass (SiO2). Charge was driven through the glass by (relatively) high voltages after which it stayed in a stable state (discharge times >10 years).

To erase the chip the charge was drained by a short (~10 minute) exposure to powerful ultra-violet light which lent enough energy to the electrons so they could escape. EPROM devices therefore required a quartz1 window in their package so they could be erased.

Programming required a special programmer and the device had to be removed from the circuit; it was therefore important that an EPROM was socketed on the PCB. The socket, expensive windowed package and programming procedure make EPROMs relatively unattractive *if* there is an alternative. (Sometimes a saving was made by using OTP or “One Time Programmable” EPROMs – the same devices but without the windows.)

EEPROM (1990s-present)
An EEPROM (Electrically Erasable Programmable Read Only Memory) uses EPROM technology but erasure may be done electrically. The devices may now be programmed ‘in situ’.

Sadly eliminating the charge leakage adds so much ‘insulation’ that the cell becomes difficult (slow) to write to. In addition, in many cases (e.g. “Flash” memory) erasure is still a ‘bulk’ erasure rather than the ability to modify single bits. Thus EEPROM is a complementary rather than a replacement technology for DRAM.

FRAM (1990s-?)
Ferroelectric Random Access Memory is another variant of DRAM which uses an electrically polarisable material to store the ‘charge’. This has the advantage that it ‘sets’ in the position it

is moved to making the memory contents non-volatile and eliminating the need for refresh. Unlike EEPROM this is truly a RAM technology though.

Although the bit is stored as an electrical rather than a magnetic polarisation this is very similar to magnetic core store in operation (see below).

Multimedia cards, “Memory Sticks”, etc.
During the 2000s **solid-state** electrostatic memories have become large enough to use as portable storage media. At the time of writing (2004) modules up to 1GB are available.

1-2.20 Magnetic Memories

Core
The memory element in core store was a small torus1 (“core”) of ferrite. This could be magnetised in either direction. This can be set (written) by passing a current through a wire threading the core. To read the device the core was probed and – if it switched – a characteristic pulse was returned. (The read was destructive, so the data has to be written back.)

Because it requires a current over a certain threshold to switch the polarisation of a core it was possible to produce dense, 2D arrays. These use two ‘address’ wires running at right angles; the current in each was kept below the switching threshold but where they crossed the *sum* of the resultant magnetic fields was great enough to affect just this one bit.

The legacy of core memory still exists in some terminology: a computer’s main memory is still sometimes called “the core”, and “core dump” for an output of a memory image is still in common usage.

Core store was followed by “plated wire” as a miniaturisation step. Magnetic core technology was in use in specialist applications (such as space shuttles) in the 1980s because it is both non-volatile and radiation resistant (“rad-hard”).

Bubble Memory (1970s+)
Now a historical curiosity ‘bubble memory’ was once thought to be *the* technology for light, portable equipment. Functionally it is the precursor of EEPROM, but works in an entirely different way.

Bits are stored in a thin film of a medium such as gadolinium gallium garnet which is magnetisable only along a single axis (across the film). A magnetic field (from a flowing electric current) can be used to generate or destroy magnetically polarised ‘bubbles’ in the film which represent the two states of a bit.

These bubbles are non-volatile. Perhaps unfortunately – if only for the name – bubble memory devices proved more expensive than other technologies.

Moving Magnetic Media
Rather than providing wires to each memory element the memory density can be increased – and the cost decreased – by providing a thin magnetic coating on a substrate material and moving this to the read/write element.

Drums
Drums were the earliest magnetic stores and often acted as directly **addressable** memory where each CPU generated address corresponds to a particular place on the drum's surface. (This is in contrast to the modern use of – for example – discs, which form **secondary** storage and is managed by a layer of software such as a filing system).

Drums were used as both primary and secondary store on many early machines; however they proved bulkier and less convenient than discs and were gradually superseded as secondary store. Core memory proved significantly better as main store.

If you want to know more about drums – and the sort of programmers who used them – look up "The Story of Mel".

Magnetic Tape
Magnetic tape uses the same storage technology as disc but the magnetic medium is carried on a flexible plastic tape rather than a plastic or metal disc. The tape is dragged past the read/write head(s) by capstan.

The heyday of tape storage was the 1950s & 1960s where science fiction films always showed computers as banks of spinning tape drives. In fact the engineering required for a **tape transport** to allow heavy reels of tape to start, stop and reverse rapidly is quite complex.

However modern systems have relegated tape to **archival** storage (such as backups) where large volumes of data are **streamed** onto tape in handy-sized cartridges. Here the slow, serial access is not a significant problem and the thin tape wound onto a spool packs a lot of bits into a small volume.

1-2.21 More Magnetic Memory

Magnetic Discs
Discs in one form or another should be familiar and need little further description. They come in several forms but can loosely be classified into hard discs ("Winchesters1") which use a metal substrate and flexible ("floppy") discs which use plastic.

Hard disc drives often contain several **platters** on a single spindle, with **surfaces** on each side of a platter. The heads are linked to the same mechanical structure ('arm'). A set of tracks at the same radius is referred to as a **cylinder**.

Figure 25:

Hard discs can store data more densely than floppies because the heads can approach more closely, more reliably; the discs can also rotate faster without distortion. Hard disc heads literally fly over the surface on a thin layer of entrained air. They are enclosed to prevent dust particles disrupting their operation.

The price/bit of disc storage is declining rapidly as the density increases. Future disc technologies may become more exotic. For example to store a bit in a very small area the magnetic material needs to be quite ‘stiff’; it may then need zapping with a laser to warm and ‘soften’ it each time it is written. Research is ongoing …

Unlike semiconductor RAM the access time of a disc memory depends on its mechanical configuration and will vary depending on circumstances.

1-2.22 Optical Memories

CD-ROM uses the presence/absence of pits in a foil disc to represent bits. The disc is read with a laser and optical sensor but the transport is otherwise largely similar to a magnetic disc.

A CD-ROM holds up to 650 Mbytes of data. **DVD** is simply an extension of CD technologies, with smaller (denser) bits. The only significant development is that there are two planes of bits on (each side of) the disc which are separated by ‘focus pulling’.

A DVD can contain 4.7 Gbytes of data. A HD DVD has a capacity ~15GB per layer per side.

Increasing the laser frequency (e.g. “Blu-ray”) promises ~25GB per layer (c. 2005).

Other, similar optical storage formats are possible. Particularly attractive are those which dispense with the spinning disc and tracking head (and hence the large motors with their associated power consumption). Instead of moving the medium the laser can be scanned instead, using a smaller, lighter mechanism. The medium can also be made smaller (e.g. credit card sized). Such **optical memory cards** are under investigation and development.

Being physically smaller than a CD-ROM an optical memory cards holds 2.8 Mbytes.

Holographic storage
In theory the storage of data in some sort of transparent ‘crystal’ could be very space efficient, not least because it offers 3D storage.

Such storage was hinted at in, for example, the film “2001: A Space Odyssey” (1968) as the basis of the “HAL 9000” computer. Sadly this prediction proved somewhat optimistic. Another potential of holographic memory is the ease of construction of **associative** or ‘Content Addressable’ Memory (**CAM**). This is used in (for example) parts of cache memories but optical CAL may be useful for more elaborate tasks such as pattern recognition. This is beyond the scope of this course but this is an active and ongoing field of research. Searching the WWW will give more up-to-date details than can be included here.

Self-Assessment 1-2

1. What are the advantages of fetching several instructions in a single cycle?
2. What are the disadvantages?

SESSION 2-2: ADDRESS DECODING STRATEGIES

This material deals with address "mapping". In the simplest ideal case, one would have a microprocessor interfacing to a 16 megabyte memory chip, which has 8 data lines and 24 address lines. However, in practice, this is hardly the case. The "memory space" might have to include a number of memory devices, peripherals (in memory-mapped I/O systems) etc. Each of these devices must then be mapped onto the address space. For example, we might want addresses $000000 to $00FFFF to represent a 64 kilobyte RAM device, addresses $010000 to $010FFF to map to a 4 kilobyte ROM device and so on.

Thus, we require some hardware logic between the microprocessor address lines and the memory / peripheral devices to perform this address mapping and the selection of the correct device. Continuing the above example, we require logic such that if the address lines contain the address $010ABB, then the 64 kb RAM device must be disabled and the 4kb ROM device must be enabled, with the address $ABB at its address inputs. In other words, we must map the processor address range $010000 - $010FFF to the ROM device address range $000 - $FFF. There are two basic methods of performing this address mapping / decoding which are discussed in this material - Full Address Decoding and Partial Address Decoding.

2-2.1 Full Address Decoding

Full address decoding is when each addressable location within a memory component responds only to a single, unique address on the address bus. Thus, all 23 address lines must be used to determine each physical memory location for a 16-bit word - either to specify a given device or to specify an address within it.

In our example, full address decoding can be used to distinguish between the memory blocks M1 and M2. As expected, 11 address lines are to be used to access the actual memory locations within M1 or M2. Further, the remaining lines (A12 to A23) must be used to determine which device is to be accessed. One of the many possible solutions is:

1. M1 is chip selected whenever A12 is zero, and A13 to A23 are all zero.
2. M2 is chip selected whenever A12 is one, and A13 to A23 are all zero.

Thus, we are in effect mapping M1 to all addresses from $000000 to $000FFE, and similarly mapping M2 to all addresses from $001000 to $001FFE. (Note that the last digit is hex E and not F because we are treating the microprocessor as a word (16-bit) addressing machine and thus must think of addresses as even numbers only. A00 comes into play when a byte is to be addressed.) Figure 2.2.3 shows the resulting memory map graphically and also shows the schematic diagram of the address decoder for the above example. Note that a * (star) after the CS_M1 and CS_M2 signals means the NOT (!) operation which indicates that a signal is active low. (If you look at the diagram carefully, you will notice that the entire array of NOT gates and the AND-NOT (NAND) gate that follows may simply be replaced by an OR gate.

The representation used in the diagram has been used to make the "theory" behind the schematic diagram easier to understand. When it comes to the actual hardware implementation, the engineer is of course free, and encouraged, to use the simple OR gate solution.)

Figure 26: Figure 2.2.3 - Full Address Decoding Example

As a more advanced example (and thus more realistic) consider that we are required to provide full address decoding circuitry for five peripheral devices as shown in Table 2.2.1.

Table 9: Table 2.2.1 - Devices Requiring Address Decodin

Device Description	**Device Name**	**Amount of Memory to Address**
ROM chip	ROM1	2k words
RAM chip	RAM	2k words
ROM chip	ROM2	8k words
Peripheral 1	PERI1	2 words
Peripheral 2	PERI2	2 words

We then design that the devices will be mapped to the microprocessor memory space as shown in Table 2.2.2.

Table 10: Table 2.2.2 - Device Memory Mapping

Device Name	Start Address	End Address
ROM1	$000000	$000FFE
RAM	$001000	$001FFE
ROM2	$002000	$007FFE
PERI1	$008000	$008002
PERI2	$008004	$008006

The above relationships are usually represented in an address decoding table, as shown in Table 2.2.3. Note that in order to get from the values in Table 2.2.2 to those in Table 2.2.3 one should note that there is an address line A00 involved. Further, an 'x' in the table indicates that the address line in question points to a location within the device.

Table 11: Table 2.2.3 - Address Decoding Table

Finally Figure 2.2.4 shows the schematic diagram corresponding to the address decoding structure described by Table 2.2.3.

Figure 27: Figure 2.2.4 - Full Address Decoding Schematic Diagram Corresponding to Table 2.2.3.

2-2.2 Partial Address Decoding

Partial address decoding is so called because all the address lines available for address decoding do not take part in the decoding process. Partial address decoding is the simplest and most inexpensive form of address decoding to implement. Figure 2.2.5 shows how the two 2k word memory blocks of Figure 2.2.2 may be connected so that they are never accessed simultaneously. Simply, if A23 is 0, then M1 is selected and if A23 is 1, then M2 is selected.

Clearly, this is **much** simpler than the full address decoding example. However, a price has been paid. Since the selection depends only on the state of A23, obviously either M1 or M2 **must** always be selected. Thus, even though the microprocessor is able to access up to 8M words, this decoding arrangement has limited us to only 4k words of addressable space - that is, the entire address space has been dedicated to the two 2k word memory devices. This is shown more clearly in Figure 2.2.6.

Figure 28: Figure 2.2.5 - Partial Address Decoding of Two Memory Devices

Figure 29: Figure 2.2.6 - Memory Map Corresponding to Figure 2.2.5

As may be seen, the memory space of M1 appears 2048 times in the lower half of the memory map, and the space of M2 appears 2048 times in the upper half of the map. Nowdays, this method is only used in small systems, where low cost or small size is of paramount importance. Otherwise, this method prevents one from fully using the processor's available memory space and causes difficulties when the memory system is expanded at a later date. Also, due to all the repetitions, if a spurious memory access is made due to an error in the program, the memory location that responded to the access may be corrupted, thus causing malfunctioning of the system.

Consider again the more advanced example of the previous section - however this time partial address decoding will be used. The address decoding table is shown in Table 2.2.4.

Table 12: Table 2.2.4 - Address Decoding Table for Partial Address Decoding

The 'x' characters represent address lines that are required by the device in question. The blank areas represent address lines that are not used in the decoding. Further, lines A21 to A23 are used here for the actual address decoding. This results in the simple address decoding implementation as shown in Figure 2.2.7.

Now, it may be seen that now ROM1 is mapped to the $000000 to $000FFE address range. Similarly, PERI1 is mapped to the $C00000 to $C00002 range. However, the above implementation would never be used in practice since the memory space cannot be expanded. An improved partial address decoding scheme is shown in Table 2.2.5. In this way, when A23 is zero, one of the five devices in question may be accessed. Thus, the space from $800000 to $FFFFFE is never accessed, and is thus available for future expansion.

Table 13: Table 2.2.5 - Improved Partial Address Decoding Scheme

Figure 30: Figure 2.2.7 - Partial Address Decoding Schematic Diagram Corresponding to Table 2.2.4.

Block Address Decoding

As a compromise, block address decoding consists of both full and partial address decoding schemes. The memory space is divided into a number of equally sized blocks, and the addresses within each block are uniquely addressable. Thus, one could use address lines A20 to A23 to divide the 8M word address space into sixteen 512k word blocks. Each device would then be assigned a block completely to itself. Thus, expansion from our 5 devices to, say, 10 devices would in fact require no hardware modifications at all.

A good example of this being performed in software (rather than hardware) is the way that MS-DOS divides RAM into 64kbyte blocks. Hence normally when programming, for example in C or Pascal, the maximum number of contiguous elements in one array may not exceed 64kb, assuming each element is 1 byte long. This restriction, along with other mis-designs, makes MS-DOS one of the worst designed operating systems of all time. The lesson to be learnt is that future expansion and overall flexibility must be some of the topmost priorities when hardware systems are designed.

2-2.3 Designing Address Decoders

In general, there are four ways of implementing the address decoding schemes described in sections 2-2.1 and 2-2.2. Each has its advantages and disadvantages, and the choice of method often depends on the type of system being designed, the scale of its production and whether or not it needs to be expandable.

2-2.3.1 Address Decoding with Random Logic

Random logic is a term describing a system made up of Small Scale Integration (SSI) logic such as AND, OR, NAND, NOR and NOT gates. Address decoding entirely with random logic is seldom found, mainly due to its high cost in terms of the number of chips required. Also, it is always custom made and thus lacks flexibility. The only advantage is speed - the fastest logic circuitry may be chosen, and the design customized so as to minimize propagation delays. The low cost of SSI devices is not really an advantage since it is offset by the increased costs of design, board area and testing. The previous sections all show address decoding implementations which use random logic.

2-2.3.2 Address Decoding with m-line-to-n-line Decoders

The complexity and problem of address decoding is greatly reduced by using data decoders that decode an m-bit binary code into one of n outputs, where n = 2m. Numerous types and variations of these decoders exist, however only the 74LS138 three-line-to-eight-line decoder will be discussed here - the pinout diagram being shown in Figure 2.2.8.

Figure 31: Figure 2.2.8 - Pinout Diagram of 74LS138 Decoder

Note that this decoder's inputs and outputs, with the exception of the E3 (enable 3) pin, are active low. The truth table for this decoder is simple - if the decoder is not enabled, all outputs are high. If the decoder is enabled (!E1 low, !E2 low and E3 high), then the binary value at the C,B,A inputs is decoded to the correct Yx pin. For example, if C,B,A is 0,1,1, then all Y output pins are high except for Y3. Figure 2.2.9 shows five theoretical examples of how the decoder may be used.

Figure 32: Figure 2.2.9 - Applications of the three-line-to-eight-line Decoder

2-2.3.3 Address Decoding with PROM

Since address decoding simply involves the generation of different chip select outputs from a number of address inputs, any technique that allows for the implementation of Boolean functions may in fact be used. In the previous section, m-line-to-n-line decoders were used to do this. Another suitable device is a PROM. Figures 2.2.12a and b show the logical arrangement of a PROM and example pinouts of a 64 word by 8 bit PROM. The PROM has m address inputs and p data outputs. Whenever the PROM is enabled, the m-bit address at its inputs selects one of 2m possible p-bit words and applies it to the p outputs. Thus, a PROM is simply a look-up table, and instead of decoding an address by solving Boolean equations, or by dividing memory into blocks using an m-line-to-n-line decoder, it is possible to program the PROM with the turth table directly relating addresses to device-select outputs.

Note that when the PROM is disabled, its outputs float and must consequently be pulled up high in order to force the active low chip select lines to their inactive high. Alternatively, the PROM must always be enabled. There is however one snag when a PROM is used for address decoding. Suppose we wish to have a maximum of 16 devices (i.e. p = 16), and we wish to decode block sizes of 1k words. Thus, to decode 8M words into blocks of 1k words, 13 address lines (i.e. m = 13) are required (i.e. 8M * 1024 / 213 = 1k words). Thus, a PROM with a capacity of 128k-bits is required (i.e. p*2m), a relatively large PROM is required for the decoding process which makes the design and testing procedures complex. As a solution, it is often convenient to have some random logic performing basic address decoding and then have PROM's to do the more fine decoding.

Now for an example. Table 2.2.6 shows the memory space allocation that must be implemented. Note that unused memory space has purposefully been left free between the ROM and RAM, and the RAM and PERI's, in order that future expansion is not only possible but also simple.

Table 14: Table 2.2.6 - Required Memory Space Allocation

Further, Figure 2.2.12 shows the resulting implementation of the address decoder, and Table 2.2.7 shows the programming of the decoder PROM. The student is left to examine the example carefully.

Figure 33: Figure 2.2.12 - PROM-based Address Decoder Implementation

In summary, the greatest advantage of using a PROM as an address decoder is that it is able to select memory blocks of differing size (e.g. ROM1 and RAM1 are of different size) and that it is extremely versatile. The disadvantage, as mentioned previously, is that it is an excessively large lookup table - the larger the PROM, the more expensive it is and difficult to test.

Table 15: Table 2.2.7 Proramming of Address Decoding PROM

Address range of CPU	System Address Lines					System Device Enables							
	A15	A14	A13	A12	A11	PROM1	PROM2	PROM3	RAM1	PERIs			
	PROM Address Input					PROM Data Output							
	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
000000- 0007FF	0	0	0	0	0	0	1	1	1	1	1	1	1
000800- 000FFF	0	0	0	0	1	0	1	1	1	1	1	1	1
001000- 0017FF	0	0	0	1	0	1	0	1	1	1	1	1	1
001800- 001FFF	0	0	0	1	1	1	0	1	1	1	1	1	1
002000- 0027FF	0	0	1	0	0	1	1	0	1	1	1	1	1
002800- 002FFF	0	0	1	0	1	1	1	0	1	1	1	1	1
003000- 0037FF	0	0	1	1	0	1	1	1	1	1	1	1	1
003800- 003FFF	0	0	1	1	1	1	1	1	1	1	1	1	1
004000- 0047FF	0	1	0	0	0	1	1	1	1	1	1	1	1
004800- 004FFF	0	1	0	0	1	1	1	1	1	1	1	1	1
005000- 0057FF	0	1	0	1	0	1	1	1	1	1	1	1	1
005800- 005FFF	0	1	0	1	1	1	1	1	1	1	1	1	1
006000- 0067FF	0	1	1	0	0	1	1	1	1	1	1	1	1
006800- 006FFF	0	1	1	0	1	1	1	1	1	1	1	1	1
007000- 0077FF	0	1	1	1	0	1	1	1	1	1	1	1	1
007800-	0	1	1	1	1	1	1	1	1	1	1	1	1

007FFF													
008000- 0087FF	1	0	0	0	0	1	1	1	1	1	1	1	1
008800- 008FFF	1	0	0	0	1	1	1	1	1	1	1	1	1
009000- 0097FF	1	0	0	1	0	1	1	1	1	1	1	1	1
009800- 009FFF	1	0	0	1	1	1	1	1	1	1	1	1	1
00A000- 00A7FF	1	0	1	0	0	1	1	1	1	1	1	1	1
00A800- 00AFFF	1	0	1	0	1	1	1	1	1	1	1	1	1
00B000- 00B7FF	1	0	1	1	0	1	1	1	1	1	1	1	1
00B800- 00BFFF	1	0	1	1	1	1	1	1	1	1	1	1	1
00C000- 00C7FF	1	1	0	0	0	1	1	1	0	1	1	1	1
00C800- 00CFFF	1	1	0	0	1	1	1	1	1	1	1	1	1
00D000- 00D7FF	1	1	0	1	0	1	1	1	1	1	1	1	1
00D800- 00DFFF	1	1	0	1	1	1	1	1	1	1	1	1	1
00E000- 00E7FF	1	1	1	0	0	1	1	1	1	0	1	1	1
00E800- 00EFFF	1	1	1	0	1	1	1	1	1	1	1	1	1
00F000- 00F7FF	1	1	1	1	0	1	1	1	1	1	1	1	1
00F800- 00FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1

Note on Address Decoding and its Impact on Timing

Due to the address decoding hardware, there must consequently be a delay between the time when the address is first valid and the time at which the appropriate chip-select is asserted. The delay of course depends on the speed of the decoding hardware and the number of levels or stages in the decoder. For exmaple, a multi-stage decoder is likely to incur a long delay, and some action must be taken to rectify this - wait states must be added or faster components must be used. The next section on FPGA's, PLA's and PAL's introduces some ways that make it easier to design both fast and complex address decoders.

2-2.3.4 Address Decoding with FPGA, PLA and PAL

These devices are general purpose programmable logic elements which give the designer both the speed of random logic and the flexibility of the PROM.

FPGA's

The first of these and the most advanced is the Field Programmable Logic Array, and due to the extreme complexity of its internal circuitry, it will only be given a short overview in this course. Currently, the most advanced FPGA types are the XilinX 3000 and 4000 series chips - with clock speeds of up to 150 MHz and some with over 128 configurable input or output pins. The design is made simpler by a large array of supporting software. One of these packages is XABEL (a derivative of ABEL, with which the student should be familiar).

In order to design an address decoder, the Boolean functions which represent the address decoding logic are programmed using the high level description language of XABEL. Once the description is compiled (and automatically optimized), it is imported into OrCAD where one specifies which physical device pins correspond to which Boolean input and output signals from the XABEL program. The OrCAD design is then compiled with the XilinX Design Manager (XDM) software. The XDM software finally provides binary code which is then downloaded via the serial port of a PC to the XilinX device in question.

The main advantage of using these XilinX FPGA's is that they are easily reprogrammable. If the address decoding scheme has to be redesigned due to expansion or upgrading, the XABEL program is edited and the whole compilation procedure is followed once again and down loaded to the device. Further, the XilinX devices have enough internal logic elements (especially the 4000 series) to allow for almost any imaginable address decoding scheme of any complexity. The only disadvantage is the relatively high cost of the devices. However, this cost is offset by greatly reduced board area, reduced testing complexity, and smaller chip count. An excellent example of the use of FPGA's is on any of the newer 486DX2 and 486DX4 PC motherboards where one single FPGA performs all memory and I/O address decoding, as well as numerous other functions.

PLA

The Programmable Logic Array (PLA) is a close neighbour of the PROM, but with one extremely important feature / improvement. This is best seen in a graphical form. Figure 2.2.13 shows the structure of a PROM, and Figure 2.2.14 the structure of the PLA, both in terms of gate arrays.

Figure 34: Figure 2.2.13 - Structure of a PROM in Terms of Gate Arrays

As may be seen, the difference is that with the PROM, the AND gate array is fixed, while in the PLA the AND gate array is fully programmable. Thus, in a PROM every possible product term must have its own storage location, whether or not that product term represents a don't care condition. A PLA is a great improvement since now instead of having 2m AND gates, each with the entire product terms, a vastly reduced number of AND gates exists whose inputs may be variables, their complements or don't care states. The simplest way to program these devices is through the use of ABEL by specifying the Boolean functions to be implemented. However, since this was dealt with in detail in the third year digital systems course, the topic will not be discussed here.

Figure 35: Figure 2.2.14 - Structure of a PLA in Terms of Gate Arrays

PAL

The Programmable Array Logic (PAL) is the same as the more advanced PLA except that the array of the OR gates is fixed (not programmable). The gate array structure of a typical PAL is shown in Figure 2.2.15. Comparison of this figure to Figure 2.2.14 shows the difference clearly. Since PAL's have also been dealt with in the past, they will not be discussed any further.

Finally, the need for high speed and versatile decoders has led to the development of several special purpose programmable address decoders, such as the 18N8. The student is referred to the appropriate manufacturer documentation.

Figure 36: Figure 2.2.15 - Structure of a PAL in Terms of Gate Arrays

Self-Assessment 2-2

1.

Learning Track Activities

Summary

Unit Summary

1.

Key terms in Unit

Review Question

Unit Assignment – 2

- **Discussion Question:**
- **Reading:**
- **webActivity:**

Unit 3

MICROPROCESSOR INTERFACING

Introduction

This unit deals with microprocessor interfacing with foci on I/O addressing (port and bus-based I/O: memory mapped I/O and Standard I/O), interrupts, direct memory access and bus arbitration; and communications terminology with a special emphasis on communications protocols: basic terminology, advanced communication principles (Serial / Parallel / Wireless communication, Error detection and correction), serial protocols, parallel protocols, wireless protocols.

Learning objectives

Unit content

Session 1-3: Embedded system

- **1-3.1 A simple bus**
- **1-3.2 Timing Diagrams**
- **1-3.3 Basic protocol concepts**
- **1-3.4 control methods**
- **1-3.5 A strobe/handshake compromise**

Session 2-3: Microprocessor interfacing

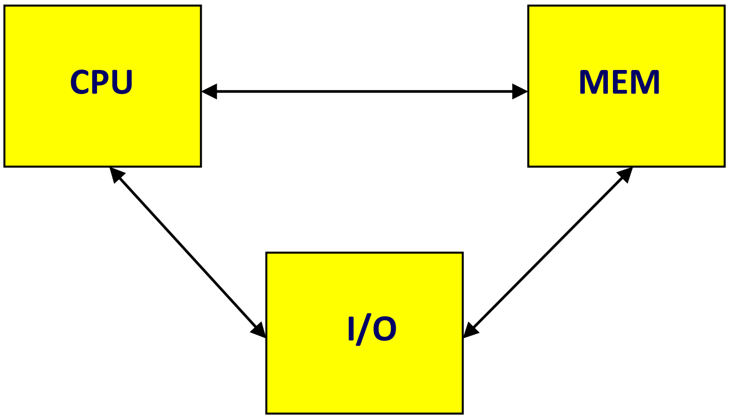
- **2-3.1 I/O addressing**
- **2-3.2 Compromises/extensions**
- **2-3.3 Types of bus-based I/O**
- **2-3.4 Microprocessor interfacing: interrupts**
- **2-3.5 Direct memory access**
- **2-3.6 Arbitration**
- **2-3.7 Multilevel bus architectures**

2-3.8 Advanced communication principles
2-3.9 Error detection and correction

SESSION 1-3: EMBEDDED SYSTEM

Embedded system has three functionality aspects:

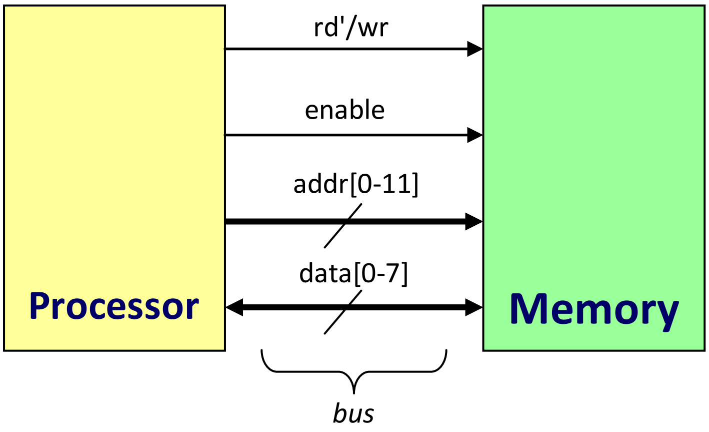
- Processing
  - Transformation of data
  - Implemented using processors
- Storage
  - Retention of data
  - Implemented using memory
- Communication
  - Transfer of data between processors and memories
  - Implemented using buses
  - Called *interfacing*



1-3.1 A simple bus

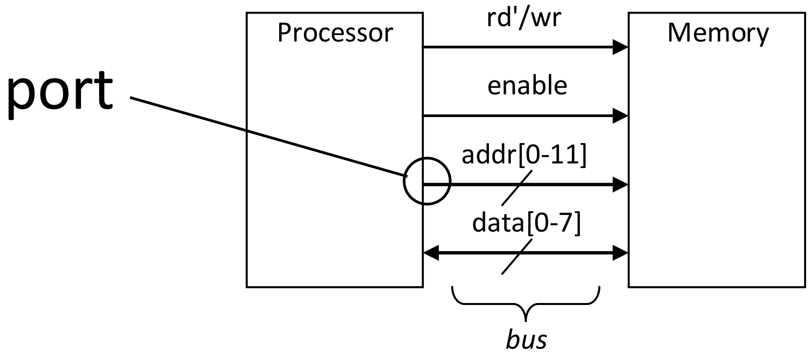
Wires are conducting channels interconnecting system devices and components. They may be uni-directional or bi-directional. A system diagram represents a wire with a line. One line may also be used to represent multiple wires.

Bus is a set of wires with a single function e.g. address bus, data bus. It may also be an entire collection of wires with various functionalities, say, address, data and control, with associated protocol, which is a set of rules for communication over the wires



bus structure

Ports

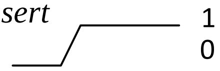


A port is a conducting device on periphery of a design entity. For example, a port connects bus to processor or memory. Ports are often referred to as *pins.* Actual pins are located on the periphery of an IC package that plugs into a socket on a printed-circuit board.

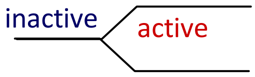
Sometimes metallic balls are used instead of pins. Within some single ICs of today, metal "pads" connect processors and on-chip memories. A "pad" may be a single wire or set of wires with single function. E.g., 12-wire address port

1-3.2 Timing Diagrams

Most common method for describing a communication protocol is by the use of timing diagrams. On a timing diagram, time proceeds to the right on x-axis. A control signal may by low or high at some intervals. A signal may be active low (e.g., go', /go, or go_L). The term *assert* is used to indicate that the signal is made active and *deassert* means deactivated. Asserting go' means set go=0.



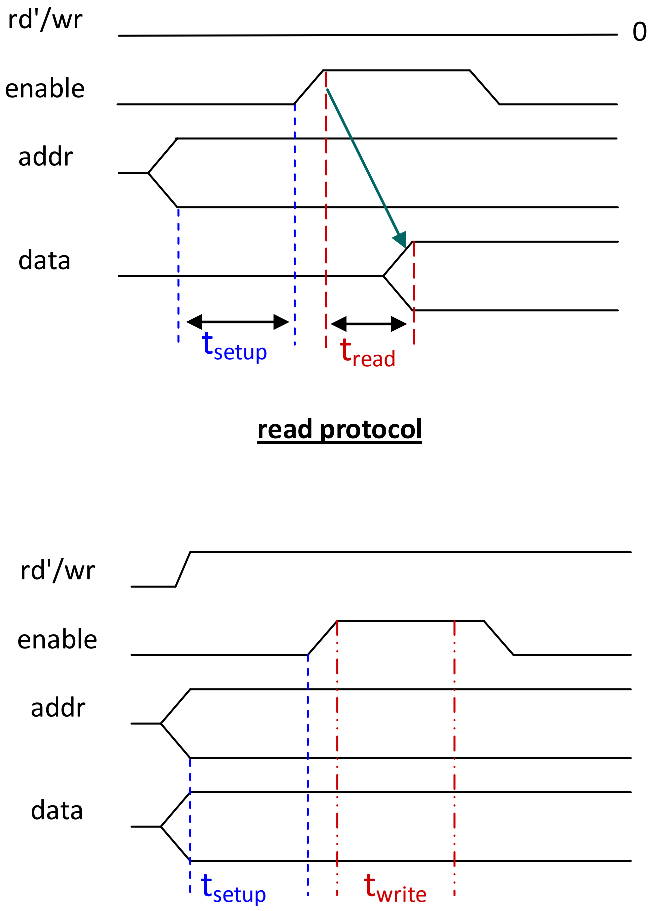
Data signal may be valid or not



A bus protocol may have sub-protocols. A sub-protocol is also called a bus cycle, e.g., read and write. Each bus cycle may take several clock cycles.

Read example

- *rd'/wr* set low,
- address placed on *addr* for at least   (setup time) before *enable* asserted,
- *enable* triggers memory to place data on *data* wires by time



<u>**read protocol**</u>

<u>**write protocol**</u>

Figure 37:

Write example

- *rd'/wr* set high,
- address placed on *addr* for at least (setup time) before *enable* asserted,
- data placed on *data* for at least (setup time) before *enable* asserted,
- *enable* allows data to flow on *data* wires into memory within time

1-3.3 Basic protocol concepts

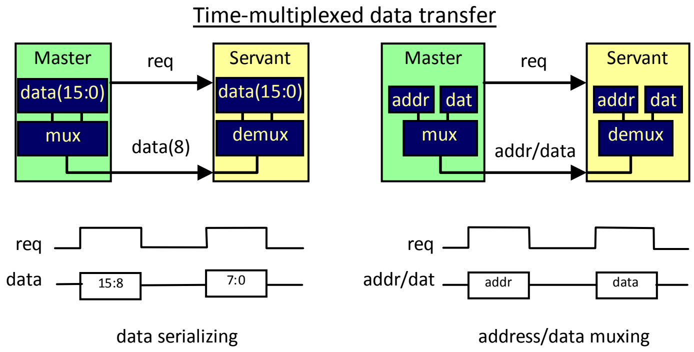
Actor: design entity (e.g. processor or memory) involved in the data transfer: master (processor) initiates, servant (slave – usually a peripheral or memory) responds

Direction: sender, receiver. Either master or slave can be the sender or receiver

Addresses: special kind of data that specifies a location in memory, a peripheral, or a register within a peripheral, where data is to be written or read.

Time (division) multiplexing (very commonly used in today's voice / data communications): Sharing a single set of wires for multiple pieces of data to save amount of wires at expense of time.

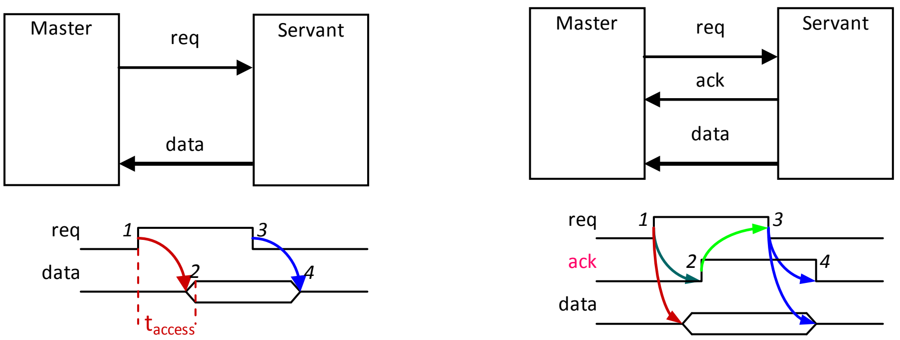
Send 16 bit data over 8 bit



Send both data and address through a

Figure 38:

1-3.4 control methods



Master wants to receive data

1. Master asserts *req (uest)* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

Strobe protocol

(faster if response time of servant is known)

Master wants to receive data

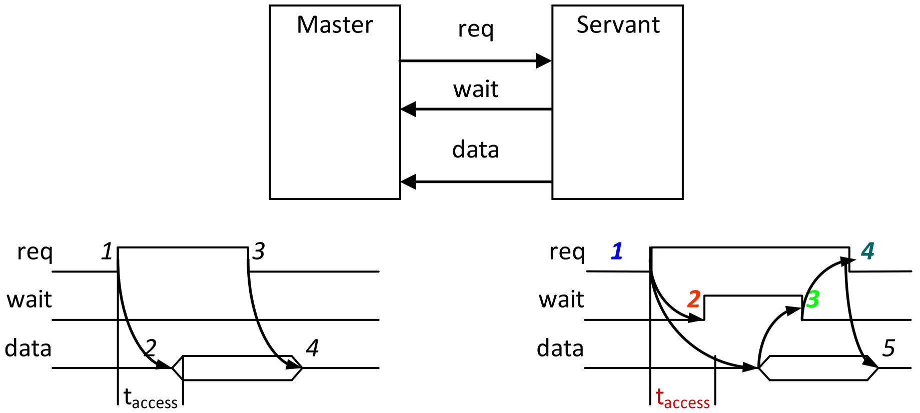
1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack*** to indicate data is ready and valid
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

(better if there are multiple servants with different response times – ack line required)

Figure 39:

1-3.5 A strobe/handshake compromise



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}** (wait line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

Fast-response case

1. Master asserts *req* to receive data
2. Servant can't put data within **t_{access}, asserts *wait***
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

Slow-response case

Figure 40:

Self-Assessment 1-3

1.

SESSION 2-3: MICROPROCESSOR INTERFACING

2-3.1 I/O addressing

A microprocessor communicates with other devices using some of its pins (non-control pins). For an I/O device, communication may be port-based or bus-based.

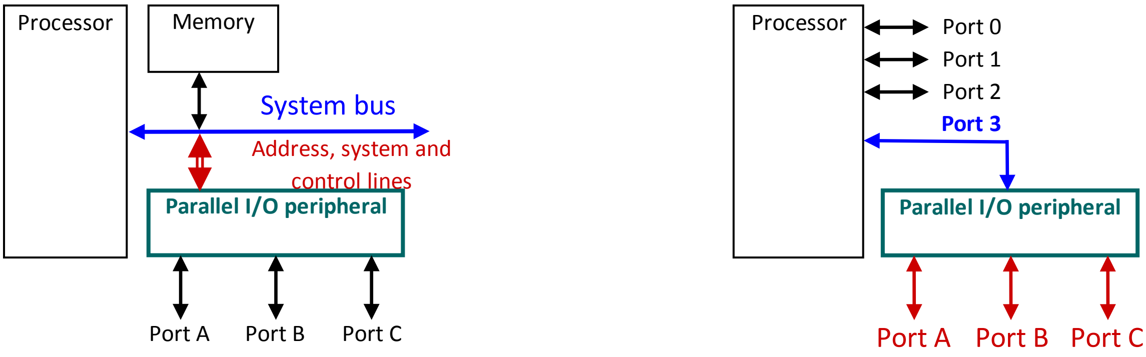
In the case of port-based I/O (parallel I/O) the processor has one or more N-bit ports, connected to dedicated registers. The processor software reads and writes a port just like a register e.g., P0 = 0xFF; v = P1; where P0 and P1 are 8-bit ports.

For bus-based I/O the processor has address, data and control ports that form a single bus. Communication protocol is built into the processor and a single instruction carries out the read (or write) sub-protocol on the bus

2-3.2 Compromises/extensions

Parallel I/O peripheral: When processor only supports bus-based I/O but parallel I/O needed, each port on peripheral is connected to a register within the peripheral that is read/written by the processor.

Extended parallel I/O: When processor supports port-based I/O but more ports needed, one or more processor ports interface with parallel I/O peripheral, thus extending total number of ports available for I/O e.g., extending 4 ports to 6 ports in figure



Adding parallel I/O to a bus-based I/O processor

Extended parallel I/O

Figure 41:

2-3.3 Types of bus-based I/O

2-3.3.1 memory-mapped I/O and standard I/O

Processor talks to both memory and peripherals using same bus. There are two ways to do these.

The first termed memory-mapped I/O. Here, peripheral registers occupy addresses in same address space as memory e.g., bus has 16-bit address with lower 32K addresses corresponding to memory while the upper 32k addresses corresponds to peripherals.
The second is standard I/O (or I/O-mapped I/O). In this case, additional pin (*M/IO*) on bus indicates whether a given access is memory or peripheral access, e.g., bus has 16-bit address all 64K addresses correspond to memory when *M/IO* set to 0 and all 64K addresses correspond to peripherals when *M/IO* set to 1

2-3.3.2 Memory-mapped I/O vs. Standard I/O

Memory-mapped I/O has an advantage, in that, there is no requirement for special instructions for I/O access. Assembly instructions involving memory, like MOV and ADD, work with peripherals as well. Standard I/O requires special instructions to move data from peripheral registers.

The advantages of standard I/O are in the following facts. 1. There is no loss of memory addresses to peripherals. 2. Simpler address decoding logic in peripherals is possible (resulting in smaller and/or faster comparators).

2-3.4 Microprocessor interfacing: interrupts

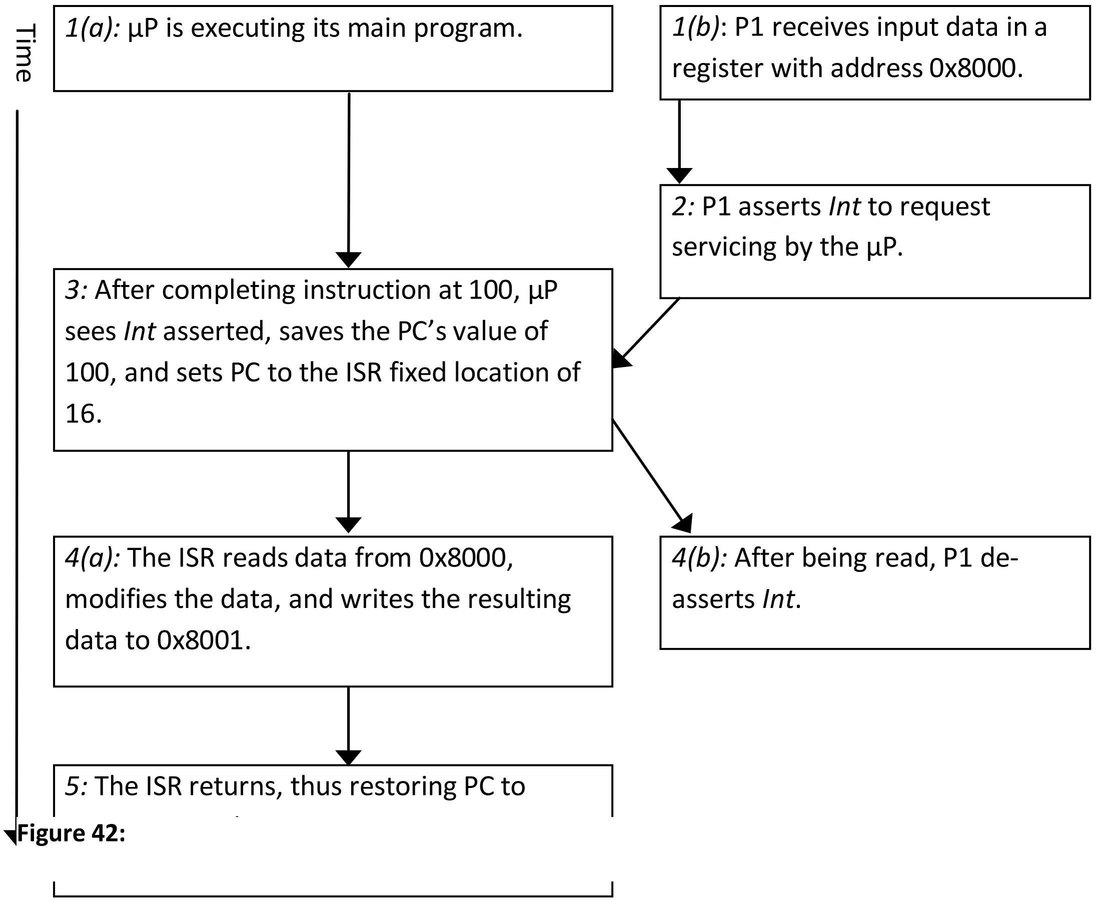
Suppose a peripheral intermittently receives data, which must be serviced by the processor. The processor can *poll* the peripheral regularly to see if data has arrived. This method is wasteful if it is performed too frequently and may miss service request if not performed frequently enough! Alternatively, the peripheral can *interrupt* the processor when it has data. The interrupt method requires an extra pin or pins, say *Int* If *Int* is 1, the processor suspends current program, jumps to an Interrupt Service Routine (ISR). An I/O that facilitates this method is known as an interrupt-driven I/O. Essentially, “polling” of the interrupt pin is built-into the hardware, so no extra time is consumed!

What is the address (interrupt address vector) of the ISR? Fixed interrupt has the address built into microprocessor and cannot be changed. Either the ISR is stored at address or a jump-to-actual-ISR is stored at the address. Several interrupt pins are available to service interrupts from multiple peripherals.

Vectored interrupt is the case where the peripheral must provide the address of associated ISR. This is common when microprocessor has multiple peripherals connected by a system bus. Only one interrupt pin is necessary, which can be asserted by any peripheral. A compromise is the provision of an interrupt address table.

2-3.4.1 Interrupt-driven I/O using fixed ISR location

Read data received by Peripheral 1 (a sensor), transform and write it to Peripheral 2 (a display). µP normally runs its main program which is located at address 100 at the time interrupt from P1 is received.



1(a): µP is executing its main program.

1(b): P1 receives input data in a register with address 0x8000.

2: P1 asserts *Int* to request servicing by the µP.

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.

4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

4(b): After being read, P1 de-asserts *Int*.

5: The ISR returns, thus restoring PC to

Figure 42:

1(a): µP is executing its main program
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts *Int* to request servicing by the microprocessor
3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.
4(a): The system bus gets activated, the ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
4(b): After being read, P1 deasserts *Int*.
5: The ISR returns, thus restoring PC to 100+1=101, where µP resumes execution.

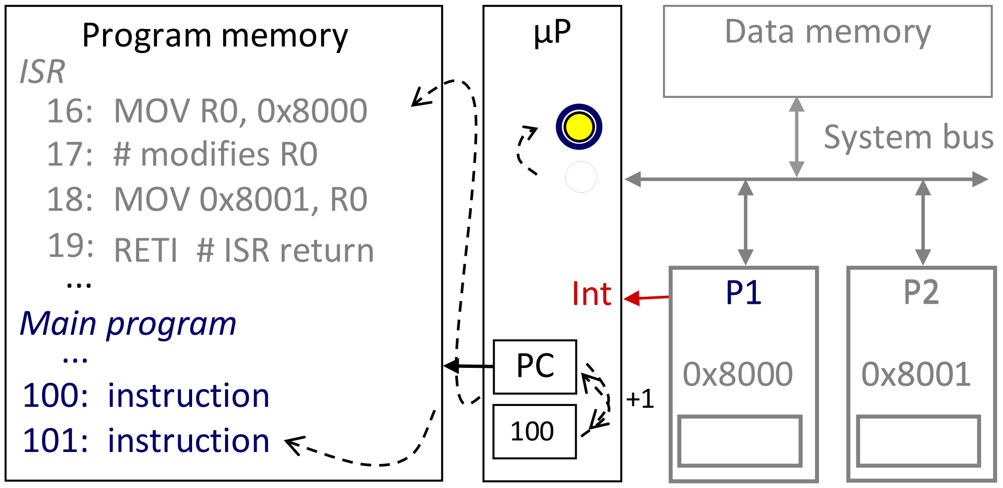
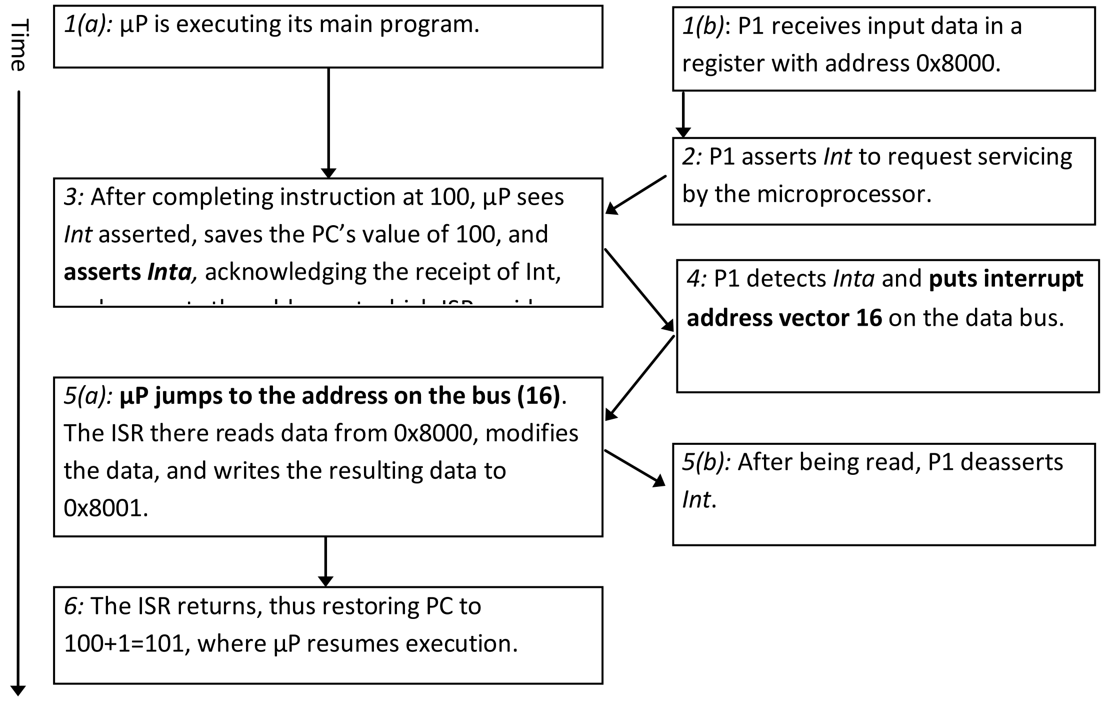


Figure 43:

2-3.4.2 Interrupt-driven I/O using vectored interrupt

Read data received by Peripheral 1 (a sensor), transform and write it to Peripheral 2 (a display). µP normally runs its main program which is located at address 100 at the time interrupt from P1 is received.



1(a): µP is executing its main program.

1(b): P1 receives input data in a register with address 0x8000.

2: P1 asserts *Int* to request servicing by the microprocessor.

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and **asserts *Inta*,** acknowledging the receipt of Int,

4: P1 detects *Inta* and **puts interrupt address vector 16** on the data bus.

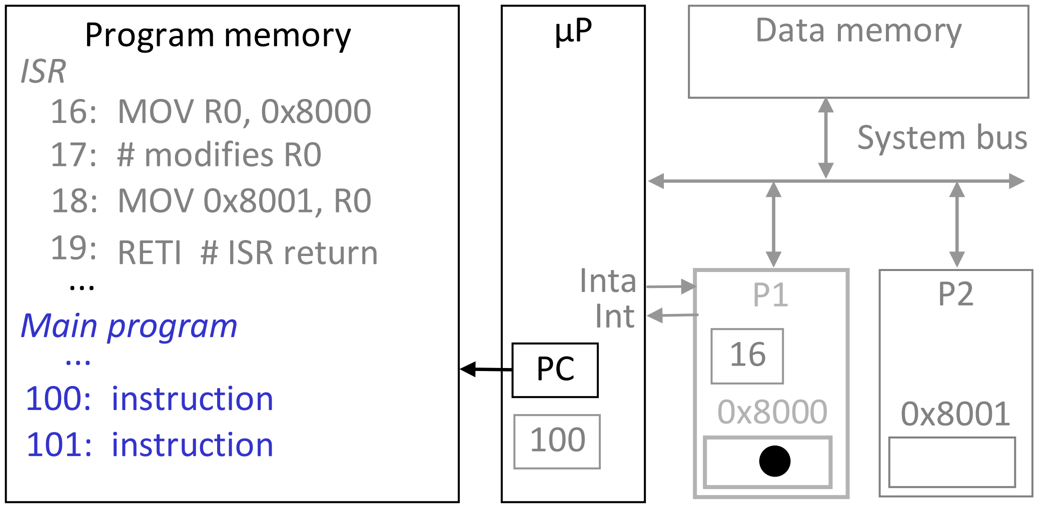
5(a): **µP jumps to the address on the bus (16)**. The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

5(b): After being read, P1 deasserts *Int*.

6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes execution.

Figure 44:

1(a): P is executing its main program
1(b): P1 receives input data in a register with address 0x8000.



```
ISR
   16:  MOV R0, 0x8000
   17:  # modifies R0
   18:  MOV 0x8001, R0
   19:  RETI  # ISR return
   ...
Main program
   ...
 100:  instruction
 101:  instruction
```

Figure 45:

2: P1 asserts *Int* to request servicing by the microprocessor

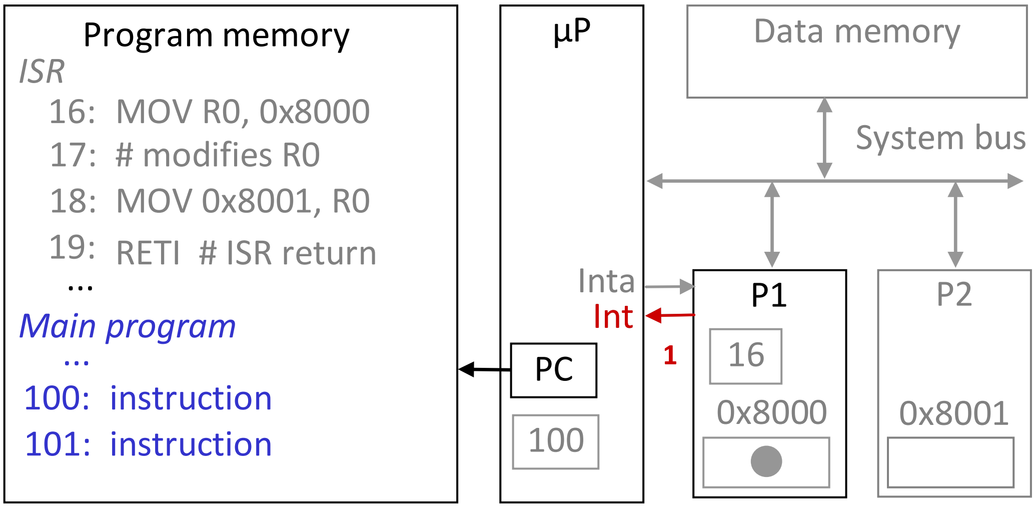


Figure 46:

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC’s value of 100, and **asserts *Inta***

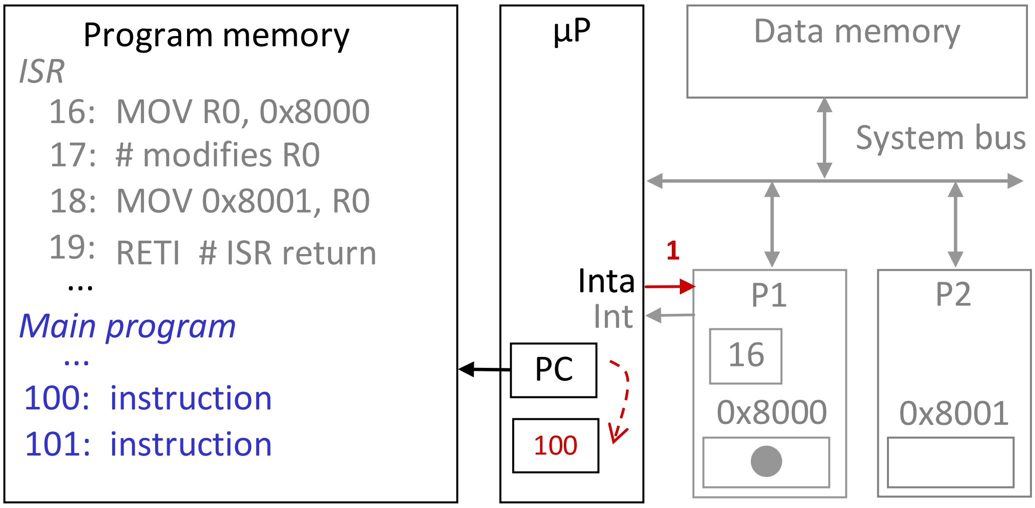


Figure 47:

4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus

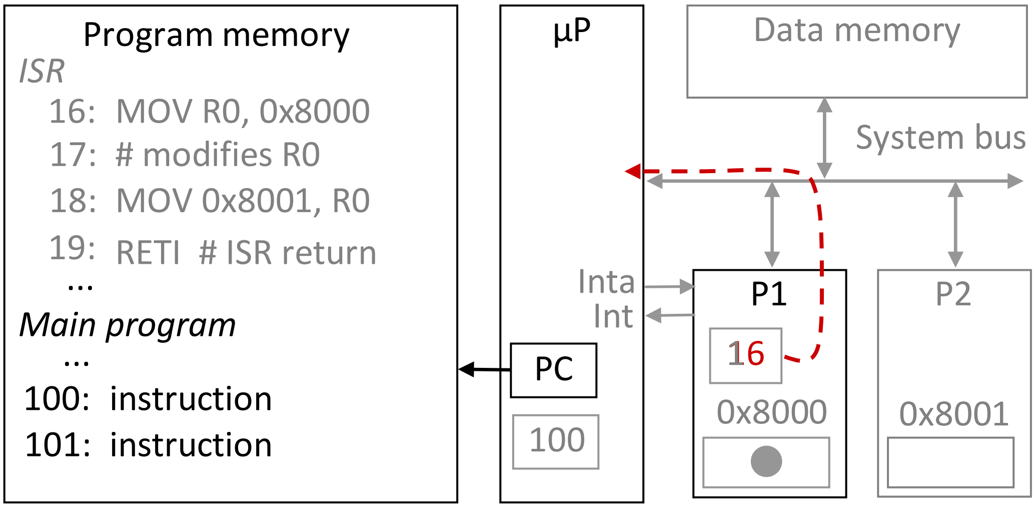


Figure 48:

5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
5(b): After being read, P1 deasserts *Int*.

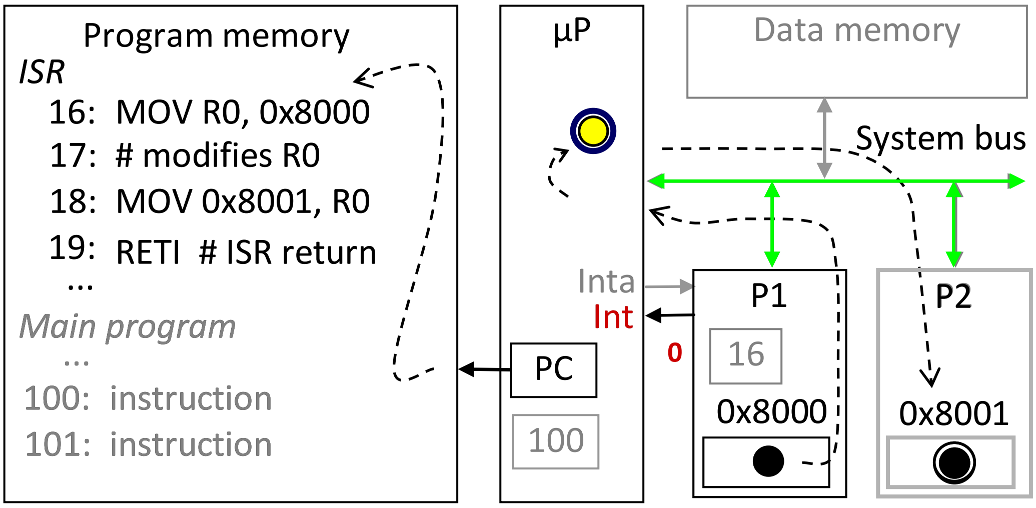


Figure 49:

6: The ISR returns, thus restoring the PC to 100+1=101, where the µP resumes

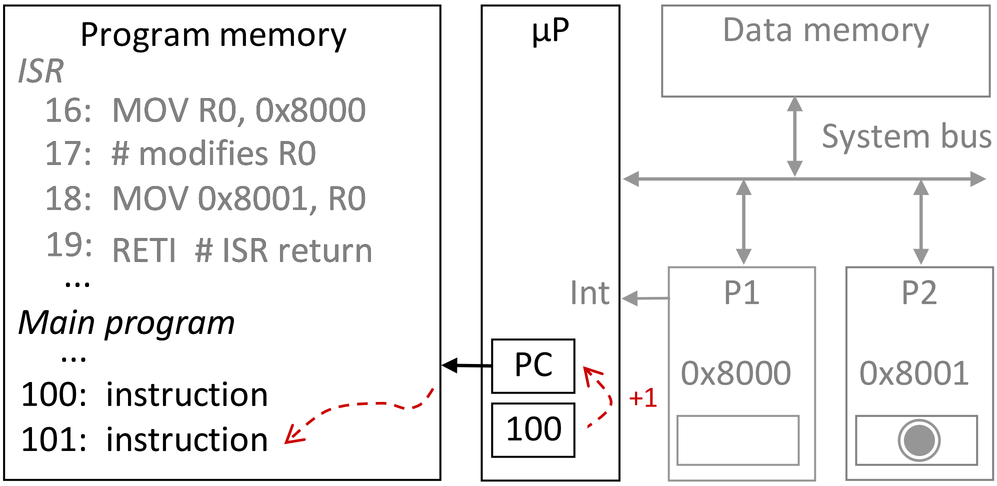


Figure 50:

2-3.4.3 Interrupt address table

Interrupt address table is a compromise between fixed and vectored interrupts. It only requires one interrupt pin and a table in memory holds ISR addresses (maybe 256 words). A peripheral doesn't provide ISR address, but rather index into table, this way fewer bits are sent by the peripheral. Also, ISR location can be moved without changing peripheral.

An interrupt may be maskable or non-maskable. A maskable interrupt provides a means by which a programmer can set a bit (or bits) that causes processor to ignore the interrupt. This is important in critical situations such as when the processor is in the middle of executing the instructions of time-critical code (time counting). A non-maskable interrupt has a separate interrupt pin that cannot be masked. This interrupt is typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory.

2-3.5 Direct memory access

Buffering refers to temporary storage of data in memory before processing. Data accumulated in peripherals are commonly buffered. The microprocessor could handle buffering with an ISR. Doing this for operations such as storing and restoring of microprocessor state (register contents, such as PC) are inefficient—while doing this regular program must wait.

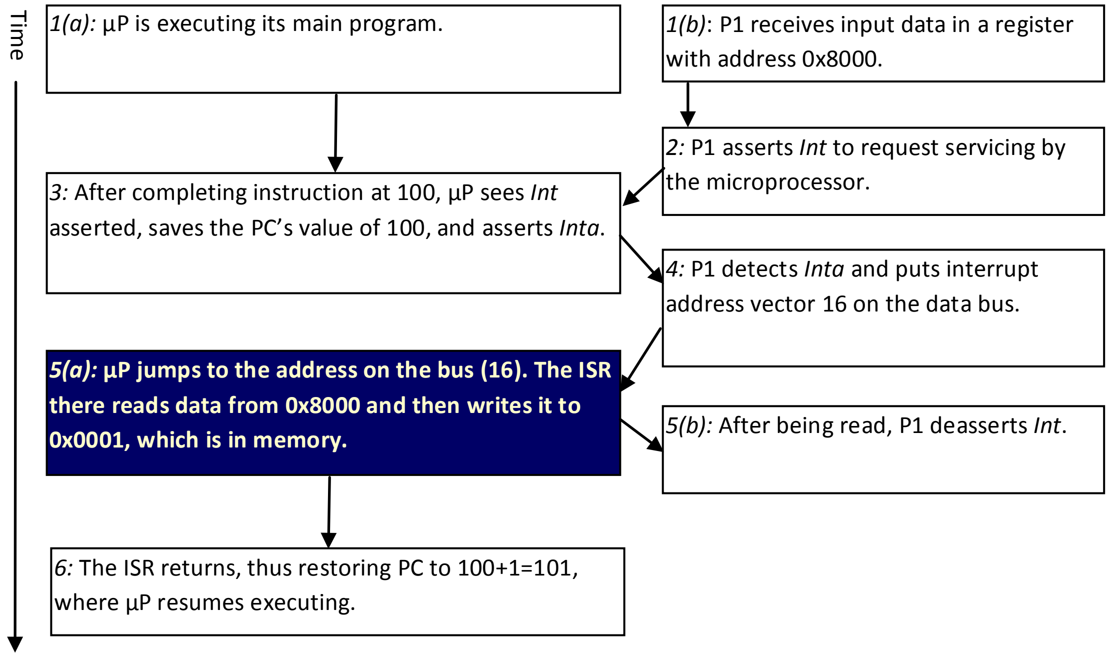
A DMA controller is more efficient. A separate single-purpose processor is given the responsibility for **moving data between peripherals and memory.**

- Peripheral makes a request to DMA to write data to the memory
- DMA requests bus control from the µP
- Microprocessor merely relinquishes control of system bus to DMA controller (rather than jumping into an ISR) avoiding all the overhead associated with ISR

Microprocessor can meanwhile execute its regular program. No inefficient storing and restoring of processor state due to ISR call is necessary. This means regular program need not wait unless it requires the system bus. In the case of Harvard architecture the processor can fetch and execute instructions as long as they do not access data memory, if they do the processor stalls

2-3.5.1 Peripheral to memory transfer without DMA, using vectored interrupt

Writing peripheral data into a memory works just like an ISR. ISRs are costly, not worth going all that trouble just to write data to the memory!



1(a): µP is executing its main program.

1(b): P1 receives input data in a register with address 0x8000.

2: P1 asserts *Int* to request servicing by the microprocessor.

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*.

4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.

***5(a):* µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.**

5(b): After being read, P1 deasserts *Int*.

6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Figure 51:

1(a): µP is executing its main program
1(b): P1 receives input data in a register with address 0x8000.

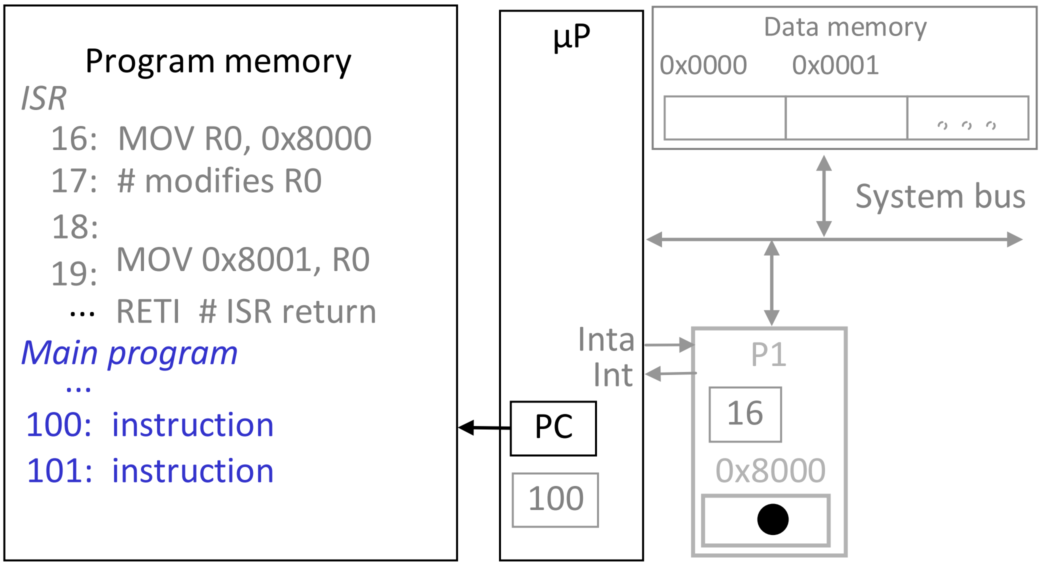


Figure 52:

2: P1 asserts *Int* to request servicing by the microprocessor

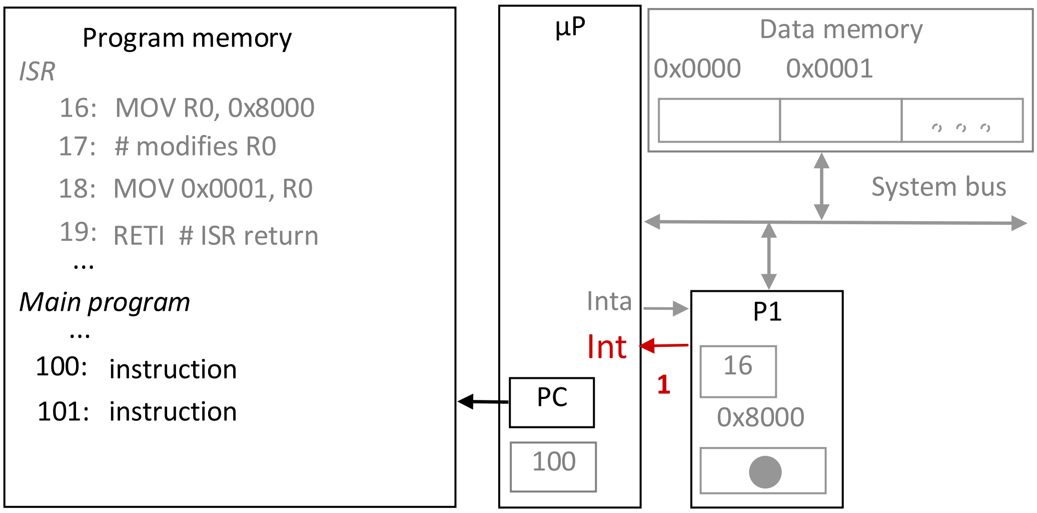


Figure 53:

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*.

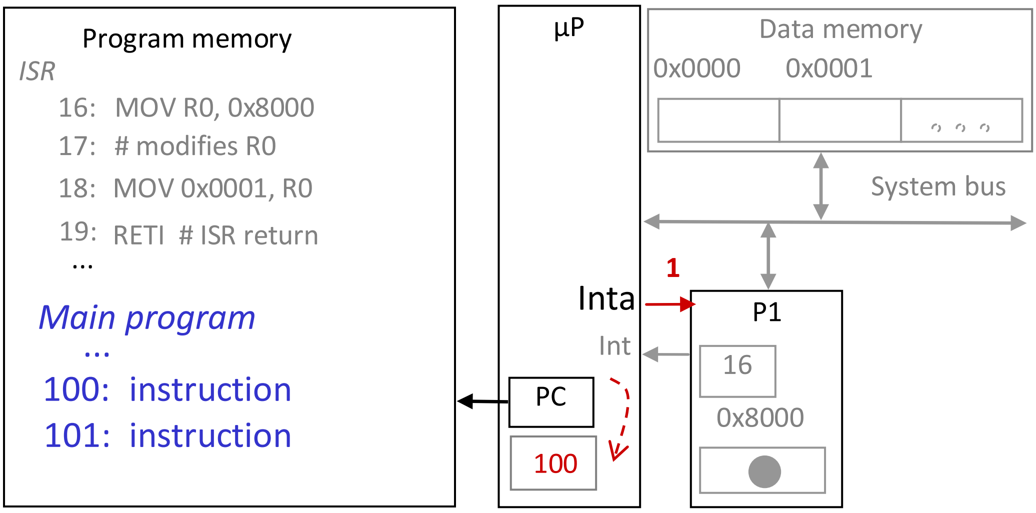


Figure 54:

4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.

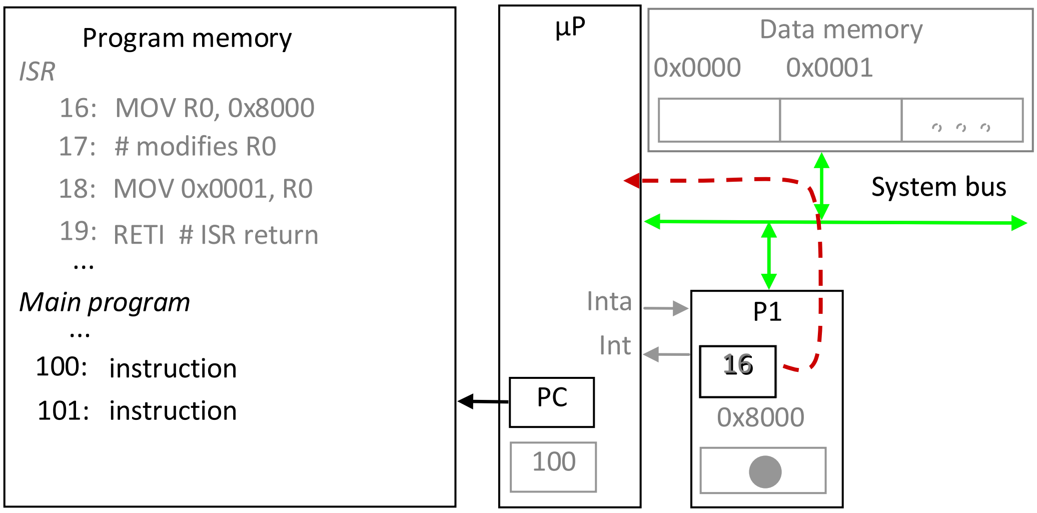


Figure 55:

5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.
5(b): After being read, P1 de-asserts *Int*.

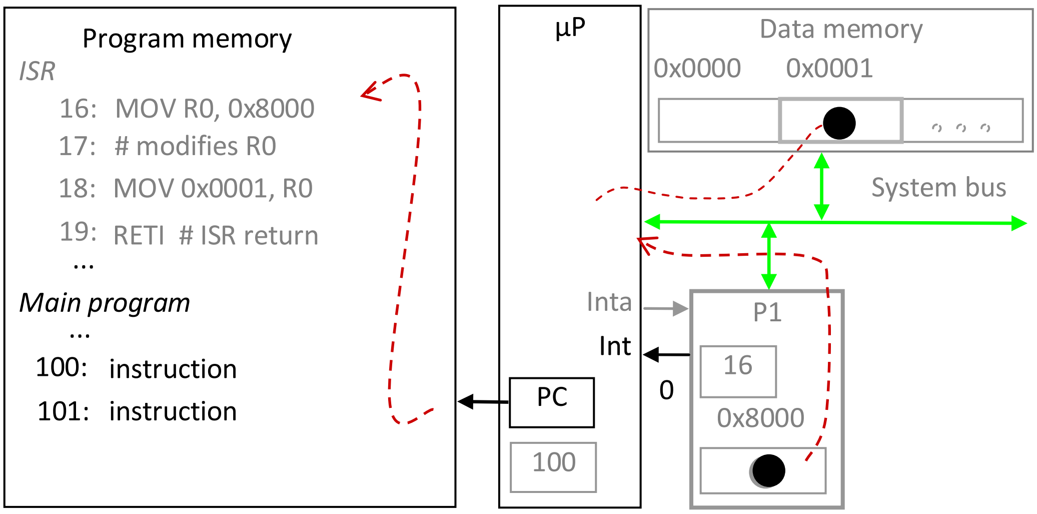


Figure 56:

6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

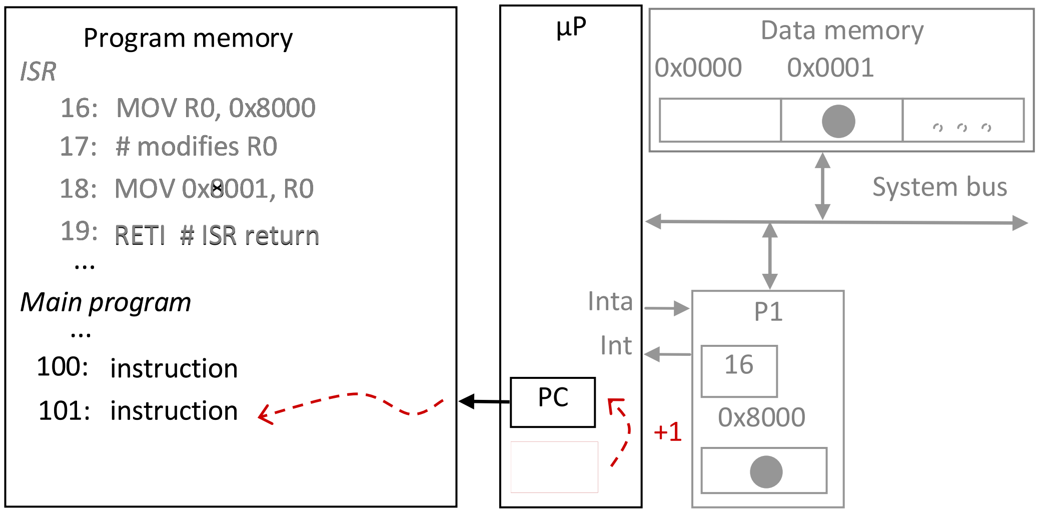


Figure 57:

2-3.5.2 Peripheral to memory transfer with DMA

Using DMA requires the µP to have two additional pins: *Dreq* used by the DMA controller to request control of the bus, and *Dack* used by the µP to acknowledge that bus control has been granted. The DMA also has two such pins *ack and rec* for a similar handshake with the peripheral.

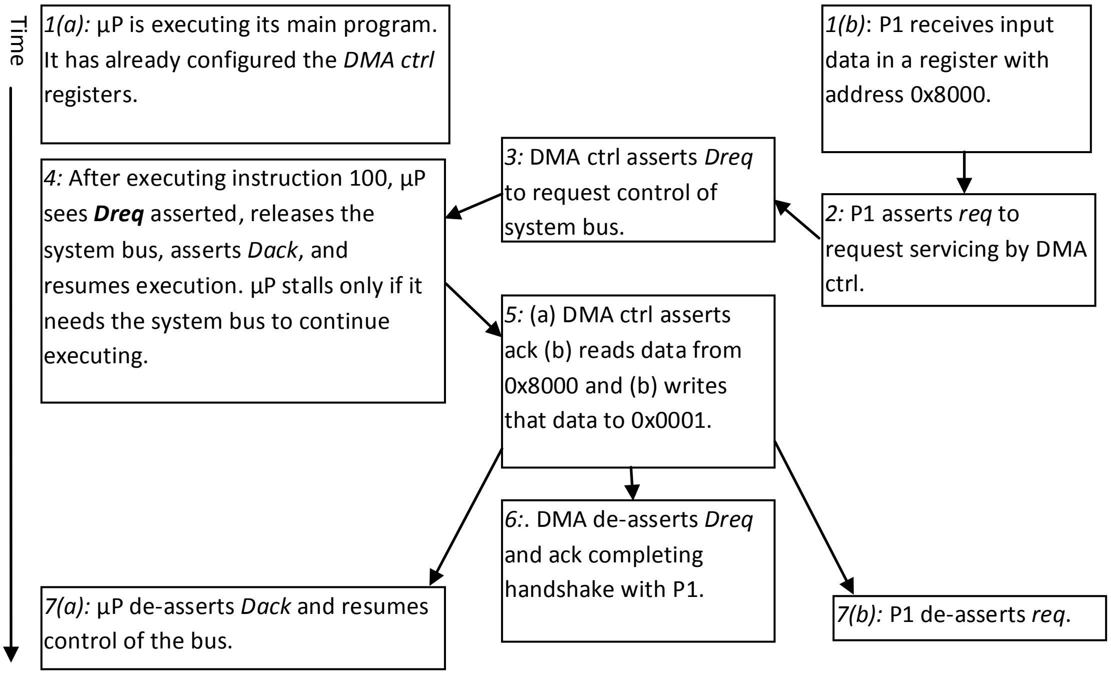

Time
1(a): µP is executing its main program. It has already configured the DMA ctrl registers.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts req to request servicing by DMA ctrl.
3: DMA ctrl asserts Dreq to request control of system bus.
4: After executing instruction 100, µP sees Dreq asserted, releases the system bus, asserts Dack, and resumes execution. µP stalls only if it needs the system bus to continue executing.
5: (a) DMA ctrl asserts ack (b) reads data from 0x8000 and (b) writes that data to 0x0001.
6:. DMA de-asserts Dreq and ack completing handshake with P1.
7(a): µP de-asserts Dack and resumes control of the bus.
7(b): P1 de-asserts req.

Figure 58:

1(a): µP is executing its main program. It has already configured the DMA ctrl registers
1(b): P1 receives input data in a register with address 0x8000.

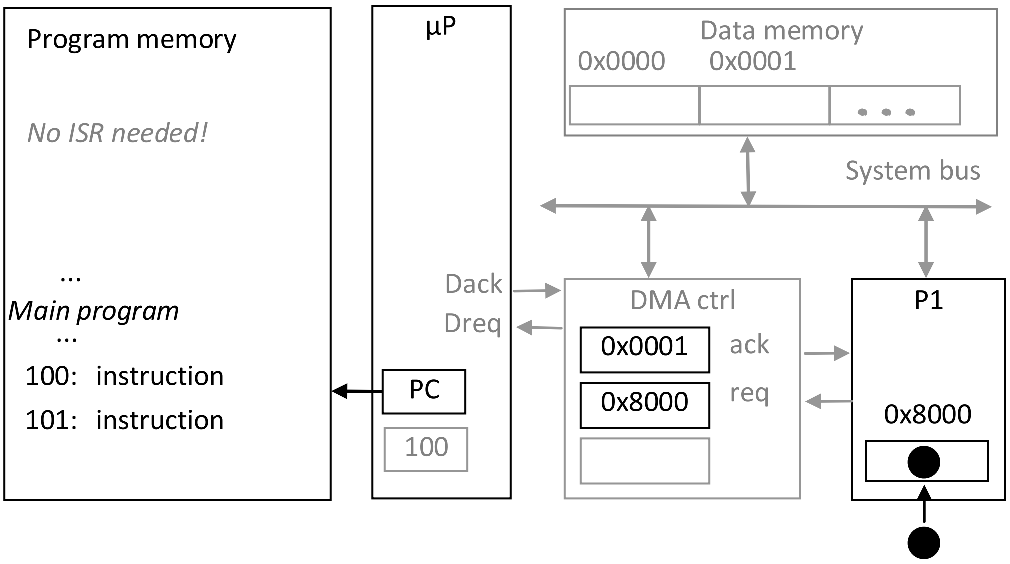


Figure 59:

2: P1 asserts *req* to request servicing by DMA ctrl.
3: DMA ctrl asserts *Dreq* to request control of system bus

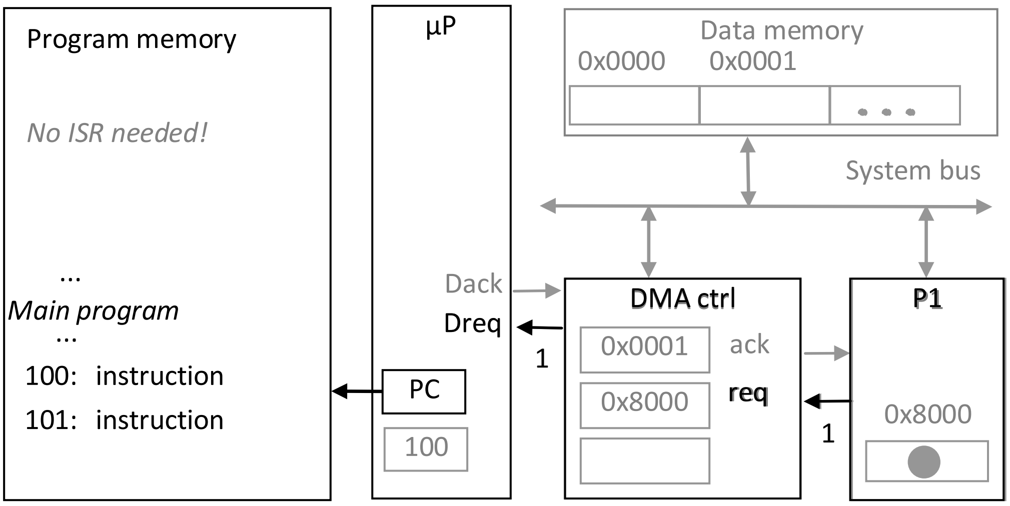


Figure 60:

4: After executing instruction 100, µP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, µP stalls only if it needs the system bus to continue executing.

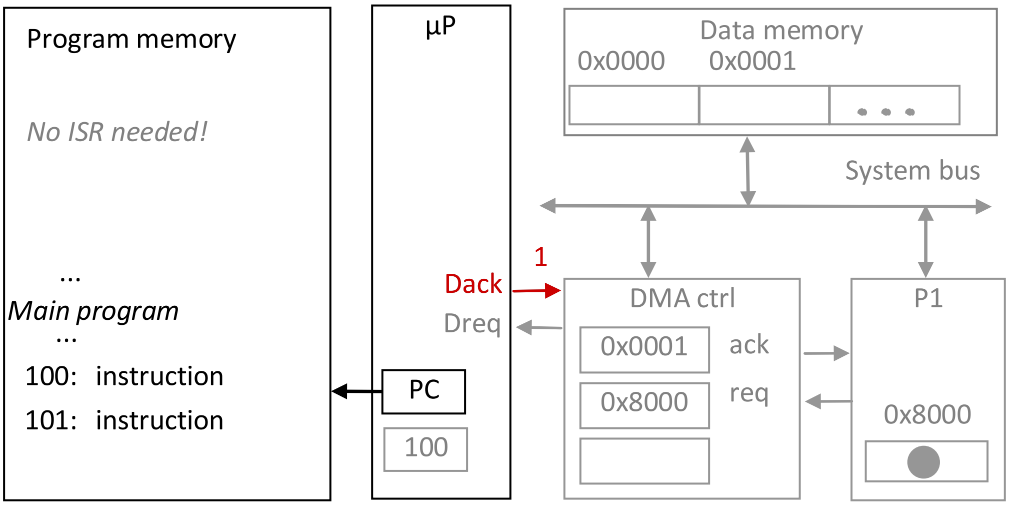


Figure 61:

5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001. (Meanwhile, processor still executing if not stalled!)

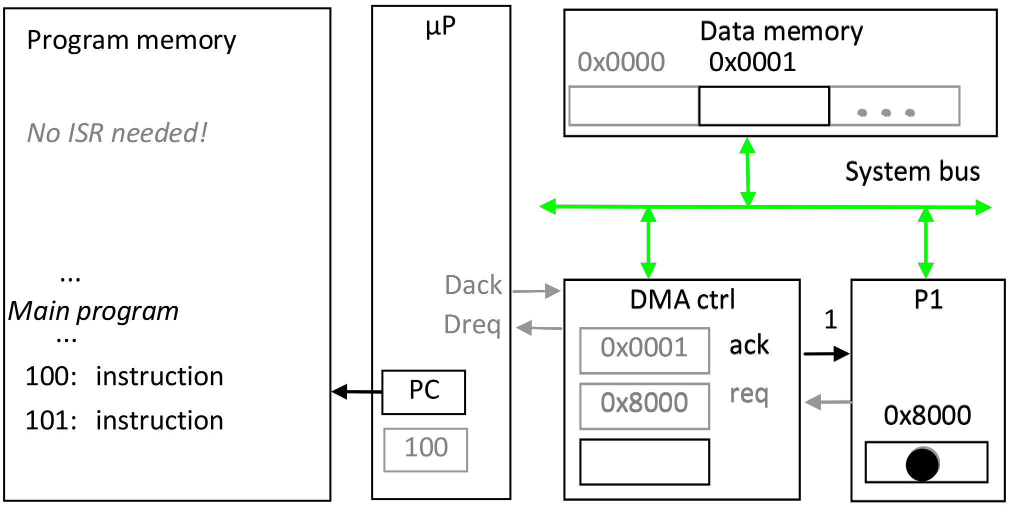


Figure 62:

6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.

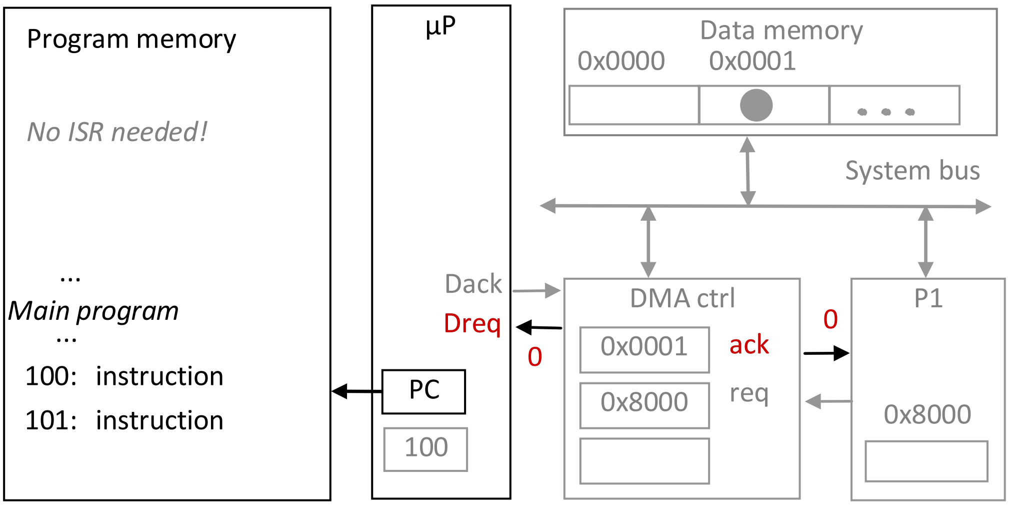


Figure 63:

2-3.6.1 Priority arbiter

Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller) simultaneously, which one gets serviced first? A priority arbiter provides a solution. A single-purpose processor is the arbiter. Peripherals make requests to the arbiter and the arbiter makes requests to the resource on behalf of the requesting peripheral with the highest priority. The arbiter is connected to system bus for the purpose of configuration only.

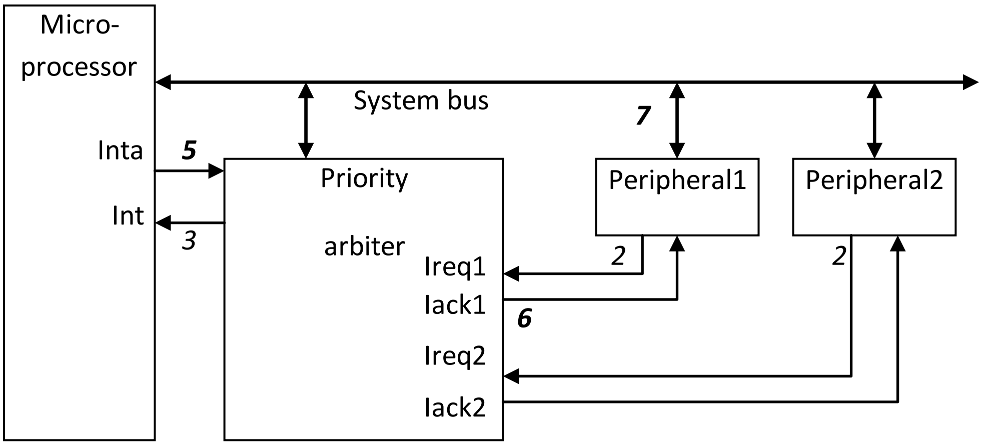


Figure 64:

1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted and so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts *Inta*.
6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus

8. Microprocessor jumps to the address of ISR read from data bus and ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

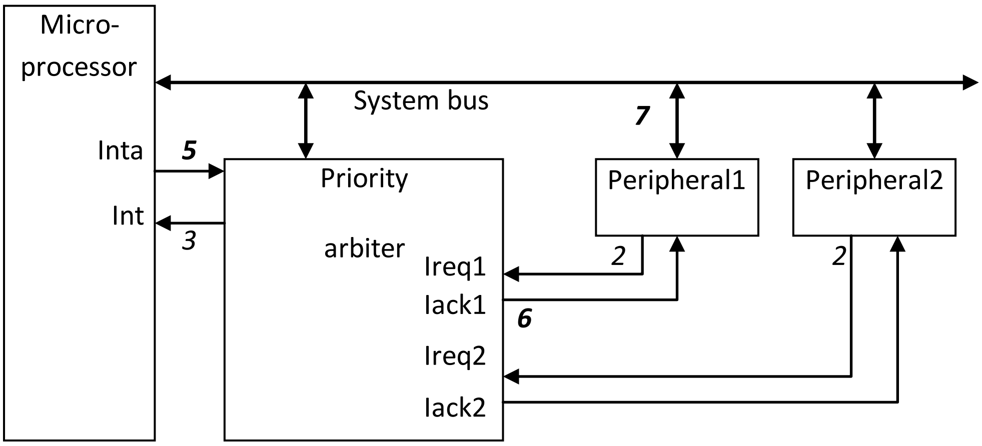


Figure 65:

There are two types of priority. Fixed priority and rotating priority. In the case of fixed priority, each peripheral has unique rank and the highest rank is chosen first in the event of simultaneous multiple requests. This type is preferred when a clear difference in rank between peripherals exists.

The second type is the rotating priority (or round-robin). Here priority changes based on history of servicing. This affords better distribution of servicing, especially among peripherals with similar priority demands.

2-3.6.2 Arbitration: Daisy-chain arbitration

In this approach arbitration is done by peripherals themselves. The arbitration functionality is either built into the peripheral or is provided by added external logic. *req* input and *ack* output are added to each peripheral in this arrangement. Peripherals are connected to each other in a daisy-chain manner. In this arrangement one peripheral is connected to the resource and all other peripherals are connected "upstream". A peripheral's *req* flows "downstream" to the resource and the resource's *ack* flows "upstream" to requesting peripheral. The closest peripheral has highest priority.

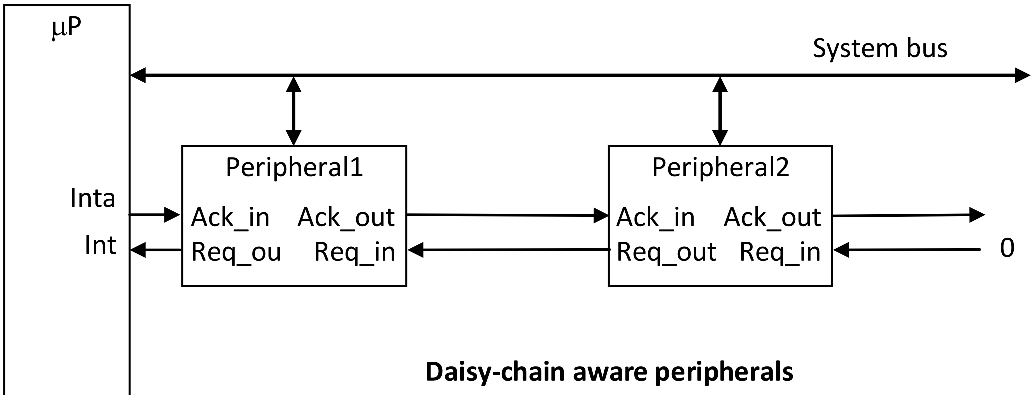


Figure 66: Daisy-chain aware peripherals

Pros and cons of daisy-chain arbitration are as follows.

- Easy to add/remove peripheral - no system redesign needed
- Does not support rotating priority
- One broken peripheral can cause loss of access to other peripherals

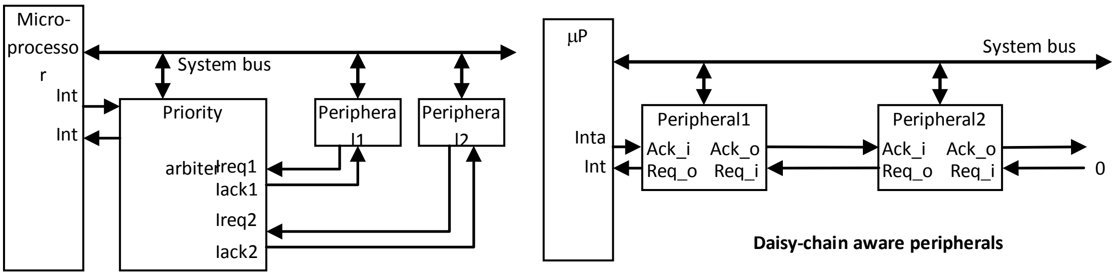


Figure 67:

2-3.6.3 Network-oriented arbitration

When multiple microprocessors share a bus (sometimes called a network).

- Arbitration typically built into bus protocol
- Separate processors may try to write simultaneously causing collisions
  - Data must be resent
  - Don’t want to start sending again at same time
    - statistical methods can be used to reduce chances

This arbitration is typically used for connecting multiple distant chips. The trend is to use this approach to connect multiple on-chip processors.

2-3.6.4 Intel 8237 DMA controller

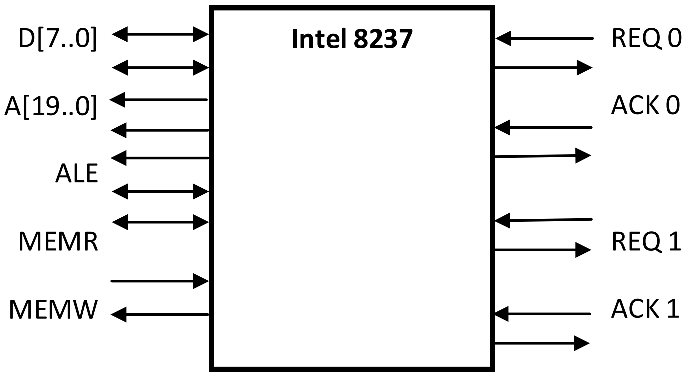


Figure 68:

Table 16:

[redacted]

2-3.6.5 Intel 8259 programmable priority controller

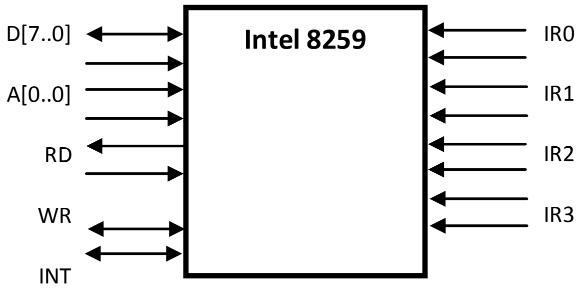


Figure 69:

Table 17:

[redacted]

2-3.7 Multilevel bus architectures

This suited to situations when it is desirable not to use one bus for all communication. This architecture requires high-speed, processor-specific bus interface. This implies excess gates, power consumption, and cost. This makes system less portable and in addition having too many peripherals leads to a slow-down of bus.

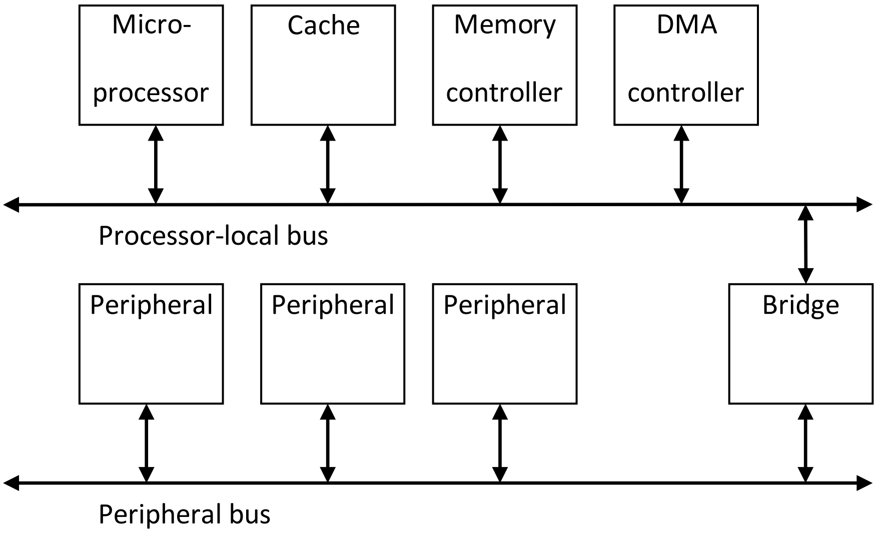


Figure 70:

- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc.
- Peripheral bus
  - Lower speed, narrower, less frequent communication
  - Typically industry standard bus (ISA, PCI) for portability
- Bridge
  - Single-purpose processor converts communication between busses

2-3.8 Advanced communication principles

Layering is the term used to describe the breaking complexity of communication protocol into pieces that are easier to design and understand. The pieces are called layers of the protocol. Lower levels (layers) provide services to higher levels and might work with bits while higher level might work with packets of data. Physical layer is the lowest level in hierarchy and provides the medium to carry data from one actor (device or node) to another.

- Parallel communication:
  - Physical layer capable of transporting multiple bits of data
- Serial communication
  - Physical layer transports one bit of data at a time
- Wireless communication
  - No physical connection needed for transport at physical layer

2-3.8.1 Parallel communication

Multiple data, control, and possibly power wires are involved. A word is sent one bit per wire. High data throughput is achievable but often used over short distances. Parallel communication is typically used when connecting devices on same IC or same circuit board. The bus must be kept short because long parallel wires result in high capacitance values, which requires more time to charge/discharge. Also, as length increases data misalignment between wires, cost and bulkiness increase.

2-3.8.2 Serial communication

Single data wire, possibly there may also be control and power wires. Words are transmitted one bit at a time. Serial communication has higher data throughput with long distances, since having less average capacitance gives rise to more bits per unit time. This means of communication is cheaper and less bulky.
However, serial communication requires more complex interfacing logic and communication protocol for the following reasons.

- Sender needs to decompose word into bits
- Receiver needs to recompose bits into word
- Control signals often sent on same wire as data, increasing protocol complexity

2-3.8.3 Wireless communication

Infrared (IR)
Electronic wave frequencies just below visible light spectrum are used. Diode emits infrared light to generate signal. Infrared transistor detects signal. This transistor conducts when exposed to infrared light. It is cheap to build but needs line of sight to operate. Also operation is in a limited range.

Radio frequency (RF)
Electromagnetic wave frequencies in radio spectrum are used. Analog circuitry and antenna are needed on both sides of transmission. Line of sight is not needed and the range is determined by transmitter power.

2-3.9 Error detection and correction

Error handling is often part of bus protocol. Error detection refers to the ability of receiver to detect errors during transmission, and error correction to the ability of receiver and transmitter to cooperate to correct errors. Typically this is done by acknowledgement/retransmission protocol.

- Bit error: single bit is inverted
- Burst of bit error: consecutive bits received incorrectly
- Parity: extra bit sent with word used for error detection
    - Odd parity: data word plus parity bit contains odd number of 1's
    - Even parity: data word plus parity bit contains even number of 1's
    - Always detects single bit errors, but not all burst bit errors
- Checksum: extra word sent with data packet of multiple words
    - e.g., extra word contains XOR sum of all data words in packet.

Self-Assessment 2-3

1.

Learning Track Activities

Summary

Unit Summary

1.

Key terms in Unit

Review Question

Unit Assignment – 3

- **Discussion Question:**

- **Reading:**
- **webActivity:**

Unit 4
INTEL PROCESSOR BASICS

Introduction

Learning objectives

Unit content

Session 1-4: Microprocessor Fundamentals
1-4.1 The Major Components of a Computer
1-4.2 Main Memory
1-4.3 I/O Devices
1-4.4 Buses

1-4.5 MPU Model
1-4.6 Fetch-Decode-Execute
1-4.7 Length of Instructions:
1-4.8 Memory Map
1-4.9 Memory Devices
1-4.10 Memory Read and Write Control Signals
1-4.11 Input and Output Cycles
1-4.12 Memory Mapped I/O and Separate I/O Mapping.
1-4.13 Address Decoding
1-4.14 Data Buses and Tristate Logic
1-4.15 Memory Access Timing

Session 2-4: The 8086 and 8088 Microprocessors; Interrupts and DMA

2-4.1 The 8088 Microprocessor in Circuit
2-4.2 8086/8088 Memory and I/O Cycles
2-4.3 Interrupts
2-4.4 Direct Memory Access (DMA)
2-4.5 Microcomputer Performance Considerations

SESSION 1-4: MICROPROCESSOR FUNDAMENTALS

1-4.1 The Major Components of a Computer

Fig.1 shows a simplified block diagram of a Microcomputer.

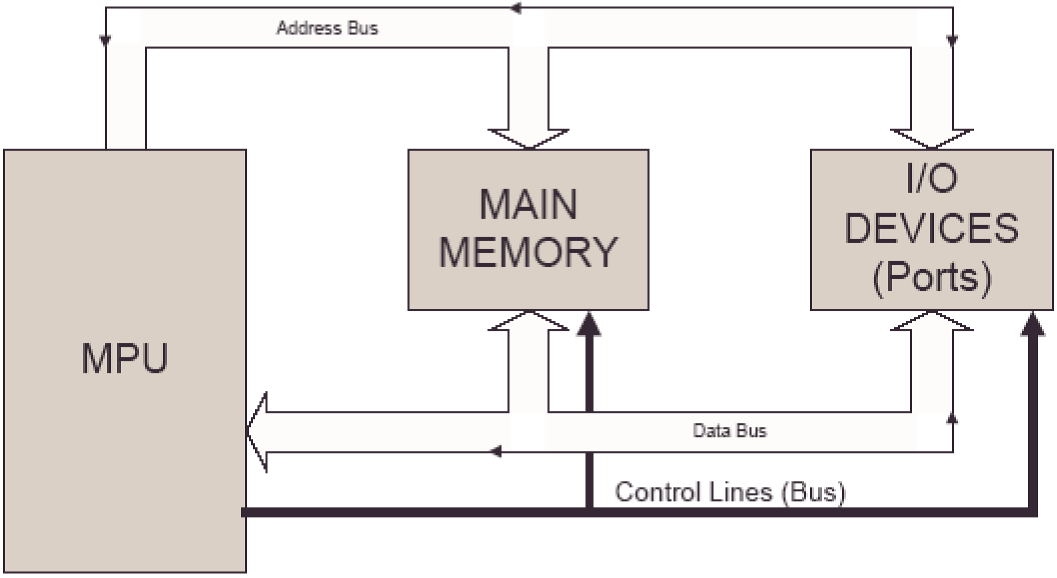


Figure 71: Fig.1. Simplified block diagram of a Microcomputer (MPU or CPU).

The Microprocessor Unit (MPU or CPU) is the heart of the computer. It controls execution of the instructions.

1-4.1.1 Main Memory

Used to store programs and data for immediate access.

1-4.1.2 I/O Devices (Ports)

Input and Output devices area range of devices that may be connected (interfaced) to the computer.

E.g. Input Devices – keyboard, mouse Output Devices – printer, monitor.
Some devices have both input and output functions and are classified as input/output devices – e.g. disk drives, serial ports etc.

1-4.1.3 Buses

A **bus** is a set of connections along which parallel binary information can be transmitted from one computer component to another. There are the address bus, the data bus and the control bus.

To access a memory location a unique binary-address is put on the **address bus** to identify a memory location and the data is then read from (or written to) the memory. The data is moved along the bidirectional **data bus**. Control lines assert signals to control the timing of various data transfer operations. A collection of such control lines is sometimes referred to as the **control bus**.

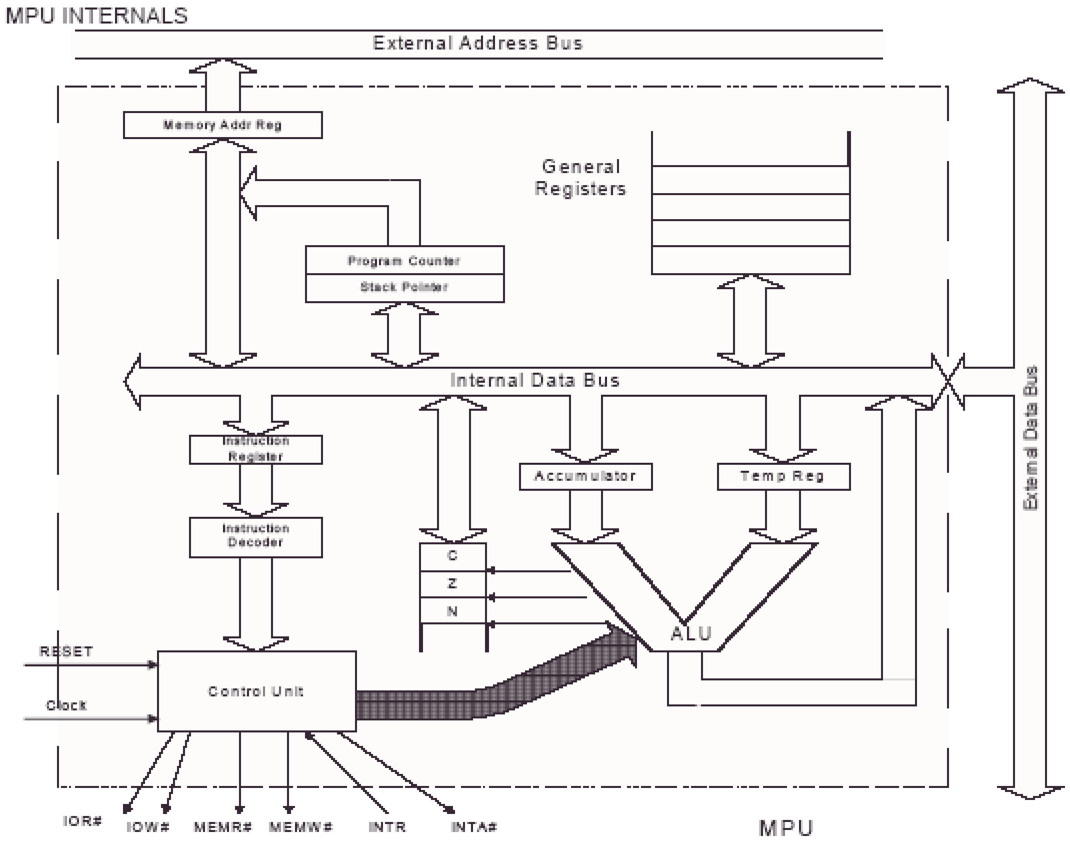


Figure 72: Fig. 2. Shows the Internal Block Diagram of a Microprocessor.

The MPU implements two conceptually different functions: the arithmetic/logic unit (ALU) and the control unit (CU). The ALU contains the logic to perform data manipulation (e.g. addition, shift, complement etc). The control unit co-ordinates the machine's activities so that the sequential order of instructions are assured. (A single machine instruction is a sequence of ones and zeros which is decoded within the CPU to define a unique operation).

General Registers The CPU contains a small set of internal registers as temporary storage for data. The register set can be seen as a small memory. The CU ensures that the data from the correct internal registers is presented to the ALU and that the data from the CPU is placed back in the correct register. Some registers have specific functions and this is particularly true of Intel Architecture processors. It is common to use one register to hold the output of ALU operations and this register is referred to as the accumulator.

The **status** or **flags register** contains individual bits to store condition information e.g. flag if the last arithmetic result was positive or negative. A typical status register is given below:

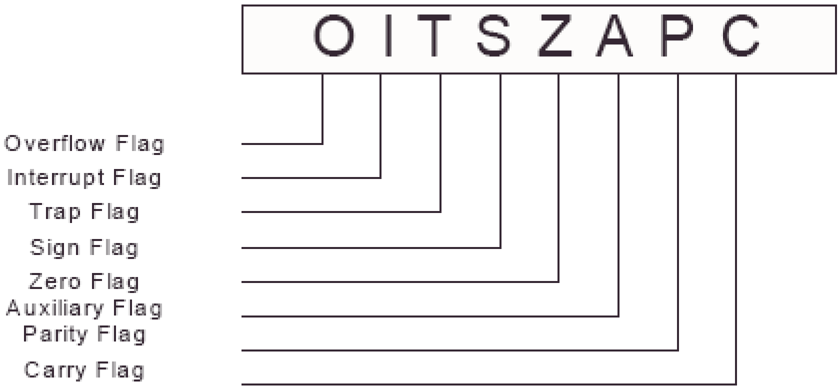


Figure 73: Fig.3 Typical Program Status Word or Status Register

The function of each flag will be discussed later.

The Program Counter register The Program Counter register is used to point to the next instruction to be executed. In the 8086 processor the term Instruction Pointer is used in place of Program Counter. We will use the terms interchangeably.

The Stack Pointer Register The stack is part of the main memory where program information can be conveniently stored by a simple push operation and restored by a simple pop operation. The stack area in main memory is defined under program control and the stack pointer register keeps track of the next memory location available on the stack. The stack is operated as a FILO (first in last out) type of buffer.

1-4.2 MPU Model

Figure 4 shows a simple microprocessor model.

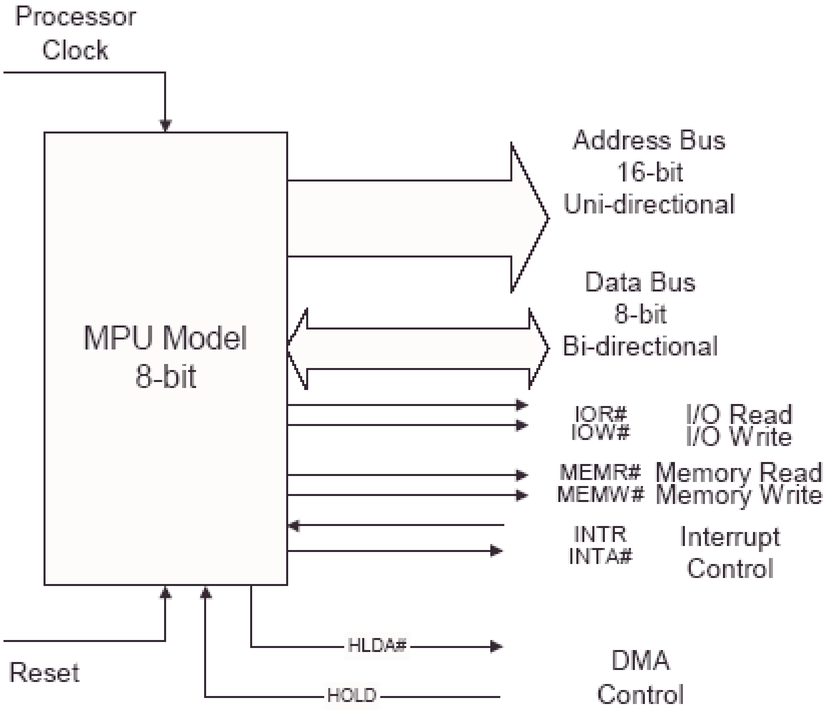


Figure 74: a simple microprocessor model

This simple model of a small microprocessor system with an 8-bit accumulator, a 16-bit address bus and various control signals is used to help describe the various signals. This model is actually quite similar to an Intel 8-bit processor, which in many cases forms the basis for even the latest generation of Intel-based systems.

1-4.2.1 Fetch-Decode-Execute

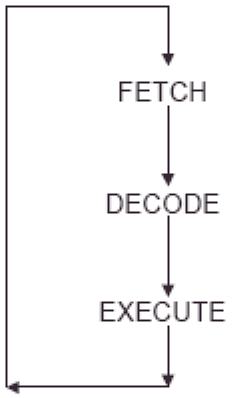
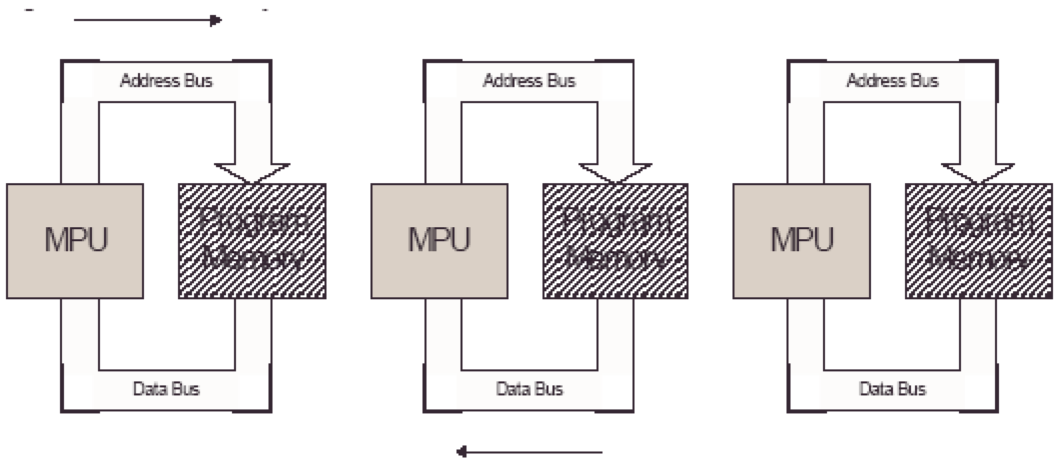


Figure 75:

A microprocessor continually performs a FETCH-DECODE-EXECUTE (or FETCH-DECODE-EXECUTE-WRITEBACK) cycle.

The processor fetches an instruction from memory by putting out the instruction pointer (IP) address out on the address bus and then reading in the instruction op-code into its instruction register via the data bus. The IP is incremented and the instruction is decoded and executed. Figure 5 shows how this cycle works.



Address (IP) is sent to memory via the address bus

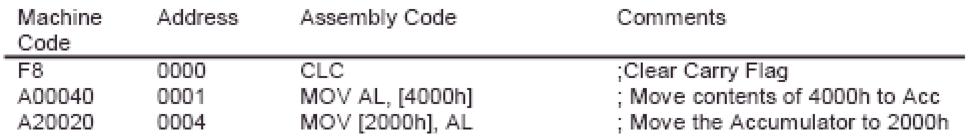
Instructiom Op Code is read into the IR register via the data bus for decoding

Instruction is executed

Figure 76: Fig. 5. Fetch Decode Execute cycle

Instruction Cycle Examples

The following code fragment starts execution at address 0000h:



Machine Code	Address	Assembly Code	Comments
F8	0000	CLC	;Clear Carry Flag
A00040	0001	MOV AL, [4000h]	; Move contents of 4000h to Acc
A20020	0004	MOV [2000h], AL	; Move the Accumulator to 2000h

Figure 77:

Execution Sequence: IP is the Program Counter in this example
Send out IP Contents (0000) and Fetch instruction byte (F8)
Increment IP
Execute Instruction – Clear Carry Flag
Send out IP Contents and Fetch Instruction byte (A0)
Increment IP
Send out IP Contents and Fetch Instruction byte (00)
Increment IP
Send out IP Contents and Fetch Instruction byte (40)
Increment IP
Execute Instruction (MOV AL, [4000h]) – this involves a memory read cycle
Send out IP Contents and Fetch Instruction byte (A2)
Increment IP
Send out IP Contents and Fetch Instruction byte (00)
Increment IP
Send out IP Contents and Fetch Instruction byte (20)
Increment IP
Execute Instruction (MOV [4000h], AL) – this involves a memory write cycle

1-4.3 Length of Instructions:

Single Byte Instruction:

CLC: Clear the Carry Flag. This is a single byte OP CODE. It has no OPERAND. The Full instruction can be read in one cycle.

Multiple Byte Instruction:

MOV AL, [4000h]: Move the contents of memory location 4000h into the Accumulator. This is a multiple byte Instruction. A2 is the instruction OP CODE and 4000 is the instruction OPERAND.

The data in our example microprocessor is only 8 bits wide so that it takes three memory read (FETCH) cycles to read a three byte instruction.

1-4.4 Memory Map

Our Model CPU is capable of addressing 64K of memory locations. This is determined by the Simple Microprocessor Model, because the Address bus is 16 bits wide. This means that it can uniquely address any of (65536) memory locations. Each memory location will be 1 byte in size. Again we know this because the data bus that we showed was 8-bits wide.

Fig.6. below shows the memory map available to our CPU. We can look at this as an empty memory map because it does not show any details of where the memory devices are located but it does show the size, or space, available for memory.

Address Bus: 16 bits, = 64K locations.
Data Bus: 8bits (1byte)

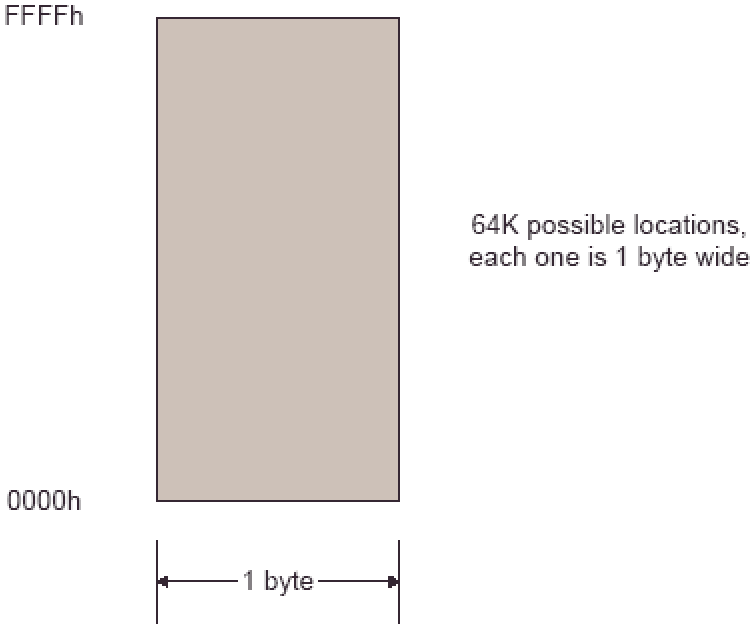


Figure 78: Fig. 6. Memory Map.

In a real microcomputer various memory and I/O devices will be located at different address ranges. We use address decoding to divide up the Memory Map into ranges in which different and unique devices can be addressed or accessed.

1-4.5 Memory Devices

Fig. 7 shows a simple ROM and a simple RAM device. The devices shown are 8K x 8 devices. Each device has 8K storage locations, each of which is one byte wide. The 13-bits of the address bus can uniquely decode each storage location (byte) within the device (= 8192 or 8K). Each device is selected by a Chip Enable signal, which allows a unique device to be selected at a given time. It’s up to the hardware decoder to generate a unique chip enable for each memory device. The output of each device is selected by an output enable signal, when data is being read from the device. Since the RAM can also be written to, it has a Write Enable signal that allows data to be written into the RAM. The control signals are active low.

Note that in Intel Architecture devices the Write Enable and Output Enable signals can often be connected directly to the MEMR# and MEMW# signals of the microprocessor.

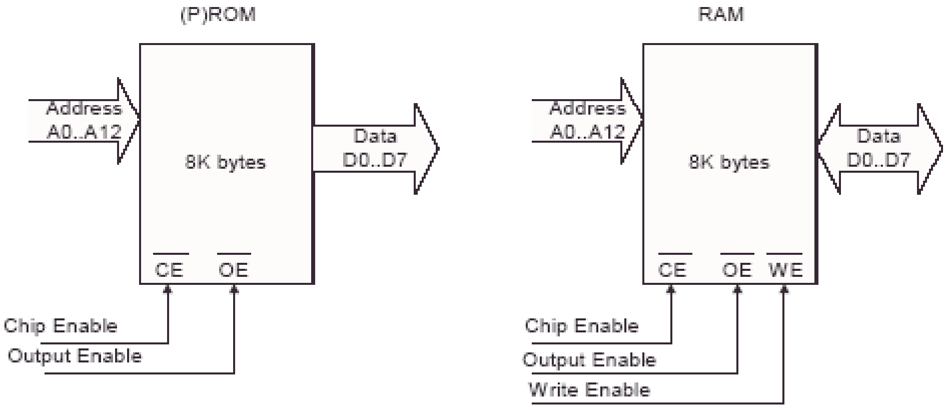


Figure 79: Fig. 7. Memory Devices

1-4.6 Memory Read and Write Control Signals

Hardware control lines driven by the CPU are used to control access to memory. These signals specify if the operation is a read or a write operation and they control the exact time for reading or writing the memory.

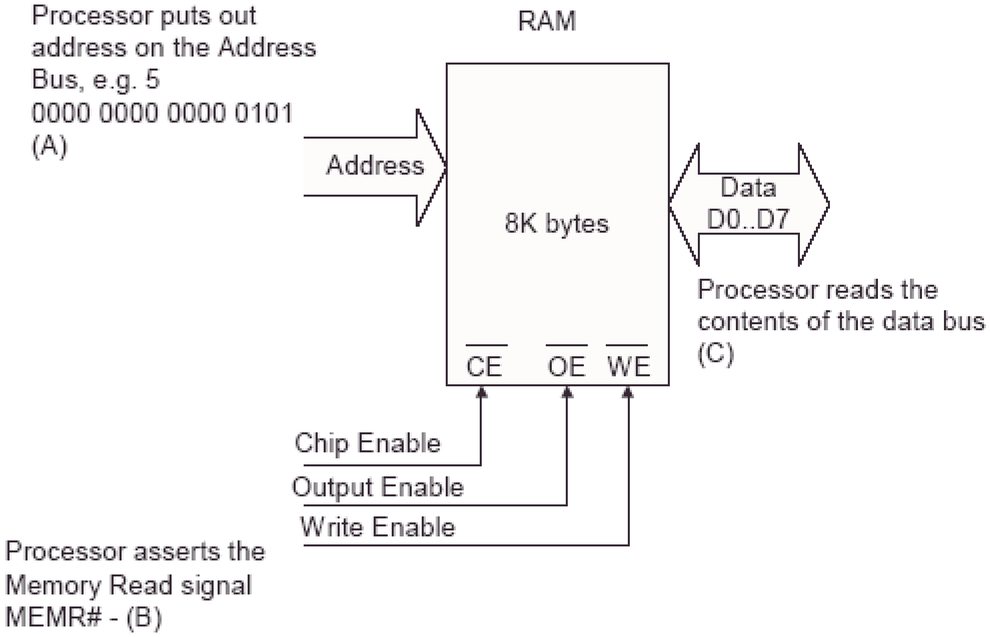

Processor puts out address on the Address Bus, e.g. 5
0000 0000 0000 0101
(A)
RAM
Address
8K bytes
Data
D0..D7
Processor reads the contents of the data bus
(C)
CE
OE
WE
Chip Enable
Output Enable
Write Enable
Processor asserts the Memory Read signal MEMR# - (B)

Figure 80:

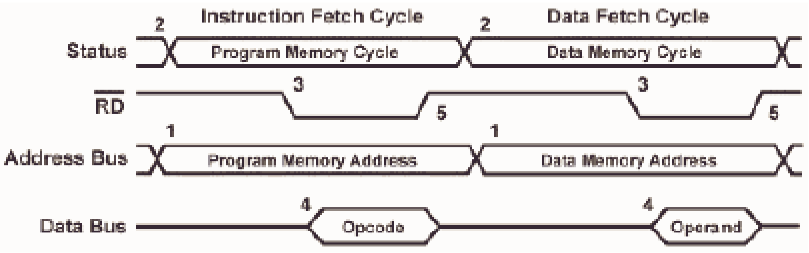


Figure 81: Fig. 8. Read Cycle.

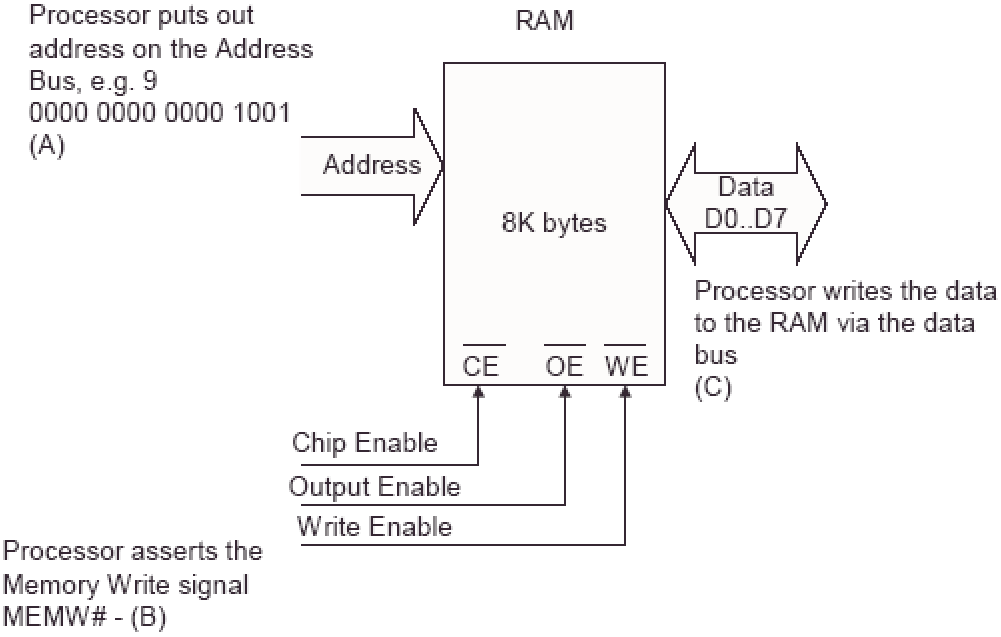

Processor puts out address on the Address Bus, e.g. 9
0000 0000 0000 1001
(A)
RAM
Address
8K bytes
Data
D0..D7
Processor writes the data to the RAM via the data bus
(C)
CE
OE
WE
Chip Enable
Output Enable
Write Enable
Processor asserts the Memory Write signal MEMW# - (B)

Figure 82: Fig. 9. Write Cycle.

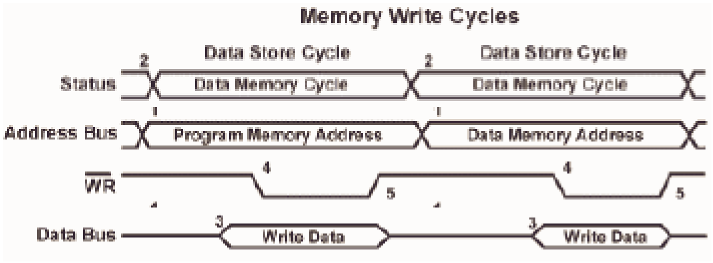


Figure 83:

In addition to the RD# and WR# signals the Chip Enable signal (CE#) must also be asserted. If CE# is not asserted the memory device will remain inactive – nothing will be written on a write cycle and the data bus will not be driven in a read cycle. Most often some hardware decode logic is used to cause the CE# signal to be asserted. This is usually based on the upper address lines. Sometimes to reduce the external hardware requirements this decode logic is actually incorporated on the processor itself. In other cases – to improve the protection schemes in an operating system environment some other signals are used to prevent certain devices from being accessed by user applications.

1-4.7 Input and Output Cycles

The Intel Architecture processors feature an I/O address space, which allows I/O devices to be decoded separately from Memory devices (code and data). In this case the IOR# and IOW# signals are used to define the I/O cycle. Note that Input and Output devices are often very different from the memory devices we looked at earlier.

I/O Instructions

If the microprocessor in question has IOR# and IOW# signals then it will need to have separate I/O instructions for inputting and outputting data. In small model CPUs the I/O instructions may use a shorter address operand – e.g. 8 bits as opposed to 16 or more bits for memory accesses, e.g.

```
MOV AL, (400Fh)     ; the instruction provides a 16-bit address
IN AL, 2Ch          ; the instruction provides an 8 bit
address.
```

(Note that you can address 64K of I/O on Z80 and all x86 processors using indirect addressing, with a general register being used as a pointer.)

Typical I/O instructions are:

```
IN AL, 80h    ; read the data from input device at 80h into the
accumulator OUT 4Ch, AL    ; output the accumulator contents to the I/O
device at 4Ch
```

1-4.8 Memory Mapped I/O and Separate I/O Mapping.

In the examples we looked at above the microprocessor had an I/O address space which was separate from the memory address space. This configuration is sometimes called Ported I/O.

Some Microprocessors do not support separate I/O mapping and therefore do not have the IOR# or IOW# signals nor do they have separate I/O instructions. These processors decode the I/O devices within the single memory map space.

Motorola processors (6800, 68000, PowerPC) do not support separate I/O mapping, while Intel processors (8085, 80x86, Pentium) do. Even if separate I/O mapping is supported the system designer has the option of decoding I/O devices within the memory address space.

Advantages of the Memory Mapped I/O.

✓ I/O locations are treated as memory locations so there’s no need for separate I/O instructions – the overall size of the instruction set can be reduced.

✓ Memory manipulation instructions allow functions such as increment, shift etc to be performed directly on the I/O device without the need for reeading data into the accumulator first.

✓ The hardware IOR# or IOW# signals or their equivalents are not required on the microprocessor thereby reducing the chip’s pin count.

Advantages of the Separate I/O Mapping

- ✓ All locations in the memory space are available for real memory devices – you don’t have to cut a chunk out for I/O devices.
- ✓ Smaller address range allows shorter instructions to be used for I/O. Shorter instructions execute faster and take up less code memory.
- ✓ Less hardware is required to decode the I/O address range because the address range is smaller.
- ✓ It’s easier to distinguish memory instructions from I/O from an assembly language point of view,

1-4.9 Address Decoding

Consider the following microcomputer system:

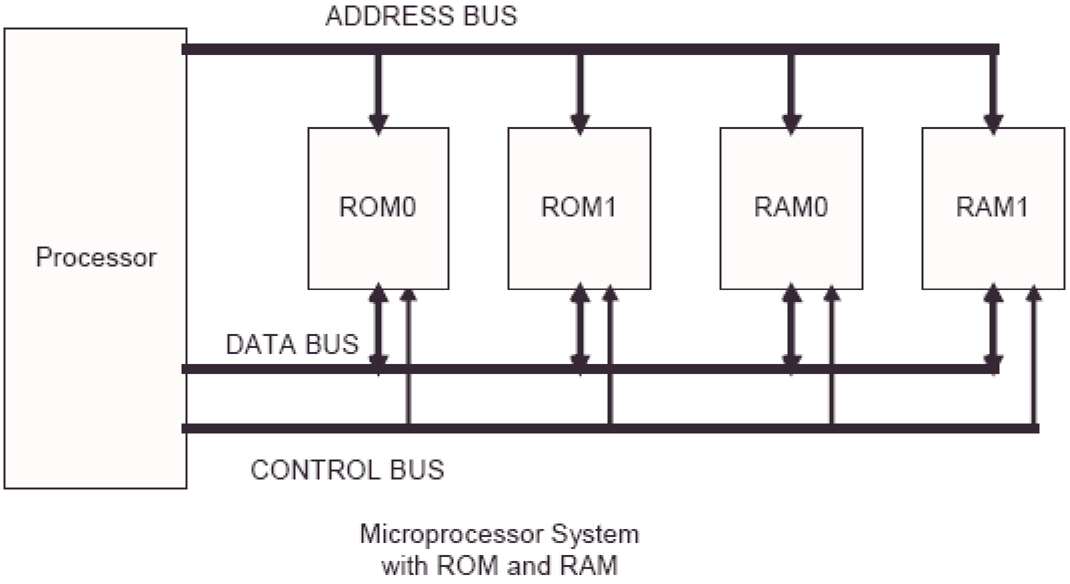


Figure 84:

For simplicity, we’ll assume that all the memory devices are the same size and that the same address, data and control signals go to each device. The question that presents itself is how does the system decide upon which memory device to access at any one time?

The obvious answer is that we need to use some external hardware to ensure that only one memory device is accessed in each read or write cycle. If this is not the case we may get contention on the data bus and the data read back will be unpredictable.

The simplest technique is to use some decoding logic that causes the Chip Enable or Chip Select of only one device to be active in any one cycle. The inputs to the decoding logic will usually be the upper address lines of the microprocessor. For example suppose that we want to implement the following system:

All memory devices are 16K x 8 in size (they each get address lines A0..A13).
We want to decode the following system:

ROM0	0h – 3FFFh

ROM1 4000h – 7FFFh
RAM0 8000h – BFFFh
RAM1 C000h – FFFFh

Graphically we can represent the memory map as follows:

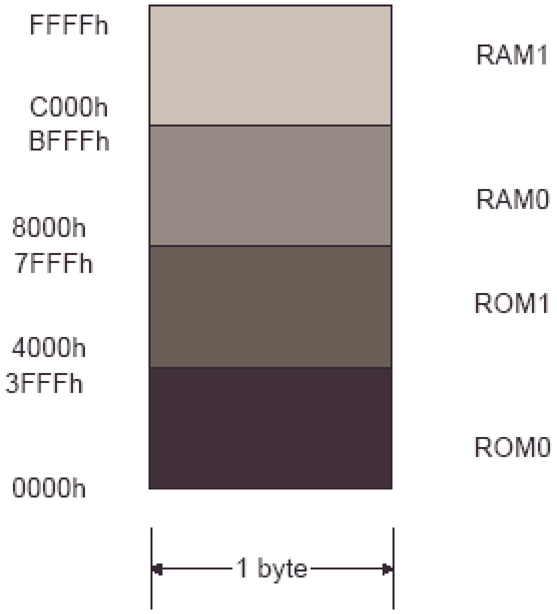


Figure 85:

1-4.10 Data Buses and Tristate Logic

In a computer system, the data signals of many memory devices (the data bus) may be connected together – and also to the CPU. Tristate logic allows these devices to be connected together without causing contention or clash on the data bus. The figure below shows a non-inverting tristate buffer, with an active low enable.

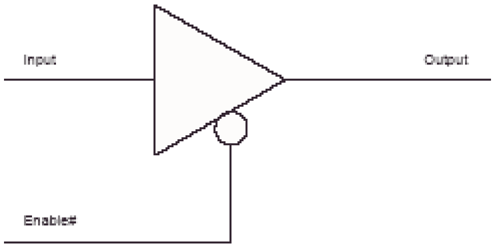


Figure 86:

Tristate buffer devices (the simplest example) have an input, an output and a second input, called the enable signal, which may be active high or low depending on the device in question. When the enable signal is active (low in the example above) the output follows the state of the input signal (after a short delay tpd, called the propagation delay of the device).

When the enable is inactive (high in our case) the output of the buffer is effectively disconnected from the input. This is often referred to as being in the high impedance state, the high-Z state or in tristate. A typical tristate buffer is a 74HC244, and a typical bi-directional tristate buffer or transceiver is a 74HC245. Bi-directional buffers are often used in data buses to increase the drive characteristics of the memory devices. Note that almost all CPUs, RAMs and ROMs have tristate buffers on their data bus pins. When the output enable or the chip enable of the RAM, for example, is inactive, the data bus of the RAM is in the high-impedance state and is effectively disconnected from the bus. Therefore another memory device or buffer will be free to drive the bus undisturbed.

Processors may also have tristatable address and control buses to allow other devices to take control of these buses, for example during a DMA operation (see below).

Discrete Tristate buffers are used to buffer various signals on a bus. Tristate buffers and transceivers can usually drive greater loads than memory devices or the CPU.

When all the signals on a bus are in the high impedance state, the bus is not driven at all. In this case there is a danger that the floating signals on the bus might oscillate thereby causing the whole circuit to malfunction. For this reason, it is common to add pull up resistors to a data bus to ensure that the signals on the bus remain at a defined, high, logic level when no device is driving the bus. Modern buffers (e.g. 74LVTH245) often incorporate special bus hold circuitry to avoid the need to pull up resistors. These devices hold the bus at the last logic level when all bus devices are in tristate.

Bus Performance

When designing a high performance bus (e.g. a data bus) you have to take both static and dynamic behaviour into account. Static behavior means that the bus drivers and buffers have to be able to drive signals to their logic high and logic low states. For example, a 74HC245 is specified to have 6mA-drive capability, while a 74AC245 can drive 24mA. The 74AC245 can drive more loads statically than a 74HC245. Note that each device on the bus represents a small resistive load to the driving device. Thus to drive a logic low the driver must be able to sink a certain amount of current, while to drive a logic high the driver must be able to source a certain current. Thus the guaranteed minimum output high voltage, VOH, for a 74AC245 at is given as 4.86V at IOH = -24mA. For driving a backplane, for example, in a PC, you need to make sure that the backplane driver device can source and sink enough current to sustain valid logic high and low levels all along the bus.

Examining the dynamic behaviour of a bus means that we have to make sure that the outputs of a buffer settle at a logic high or low level in time for the CPU or memory to read or write the data correctly. The data sheets for buffers and transceivers specify a propagation time – the time taken for the data at the output of the buffer to reach the logic level at the input after the input has changed state. For a 74HC245, the tpd (max) might be given as 19ns, while the figure for a 74AC245 is given as 7.0ns. Note that the propagation delays vary with temperature.

BUT BE CAREFUL: In CMOS systems, the capacitive loading of a bus can dramatically effect the performance of the bus. Thus the data sheet for the 74HC245 specifies that the tpd is 19ns when the device has to drive a 50pF load, but the tpd is 23ns when it has to drive a 150pF load.

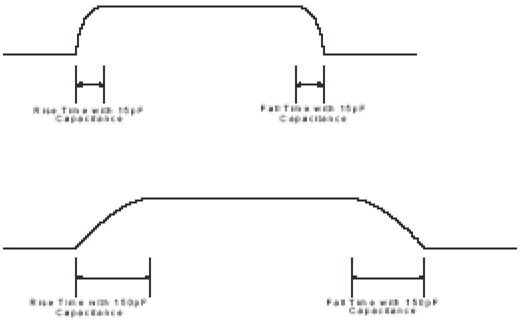

Rise Time with 15pF Capacitance
Fall Time with 15pF Capacitance
Rise Time with 150pF Capacitance
Fall Time with 150pF Capacitance

Figure 87:

In the diagram above, you can see that, in practice, the digital signals in the computer are not idealized square waves, but are affected by the resistance, capacitance and inductance of the circuit in which they are placed. The deviation from ideal signal pulse shapes is a limiting factor on the speed of the address and data bus.

The other parameters that count in buffers and transceivers (and memories as well) are the output enable and output disable times, , , , and . These are the times taken for the output to change from the high impedance state to driving a valid logic level after the Enable signals has been asserted (low in these cases). At 50pF loading, a 74HC245 specifies an output enable time of 32ns at 6V VCC, 50pF loading and . Under equivalent conditions a 74AC245 device specifies an output enable time of 9ns.

Capacitive loading arises because each device on a bus represents a capacitive load to the device that's trying to drive the bus. This figure could be up to 15pF. Often, just as important is the fact that PCB traces contribute to the capacitive loading of a bus. A long PCB track might look like a capacitor to the driving device.

Two other important related factors are: noise/transmission line effects and voltage swing. A long PCB trace looks like a transmission line to a driving buffer device and the signal may be reflected back to the source from the destination. These reflections may cause the effective time delay to be very much greater than the figure quoted in the data sheet for the buffer device.

In addition, as the driving buffer changes from a logic high to a logic low (or vice versa), it must source or sink a lot of current in a very short time. This causes very high transient currents to flow through the device and through the power and ground tracks to the device for a short time. These high currents can appear as noise around the circuit. Recall that V=IR, so if the track resistance is not very small indeed, you'll see the voltage at the device VCC drop below its specified minimum value, or you might see the voltage at the "ground" pin of the device rise well above 0V. To reduce resistance in the power and ground tracks modern PCBs as in a fast PC are always multilayer. Of course this helps fit more chips on the board too.

To reduce noise caused by high voltage swing modern transceivers sometimes do not drive the output high all the way to the VCC level. Instead they only drive the output to the voltage level that can be recognized by the input they're driving. Intel processors use Gunning Transistor Logic (GTL or GTLP) to interface between the Pentium processor and devices near it. GTLP has a voltage swing of less than 1V. It also has special circuitry to reduce the rate at which the output voltage changes state and this also helps reduce noise.

1-4.11 Memory Access Timing

Most memory devices are *random access* devices and the time it takes to access a memory location is independent of the location's position or address with in the memory. When an address is presented to the memory device, it takes a finite time for the internal logic in the memory to access the required memory location or reading or writing operations.

For example, taking a simple ROM memory we can measure the time taken for the output to become valid after an address (AND Chip Select or Chip Enable) has been properly presented to the memory device. The delay measured is the Access Time taken from the Address Change. This figure is the one quoted when you specify a memory device: e.g. a 70ns PROM or a 12ns SRAM.

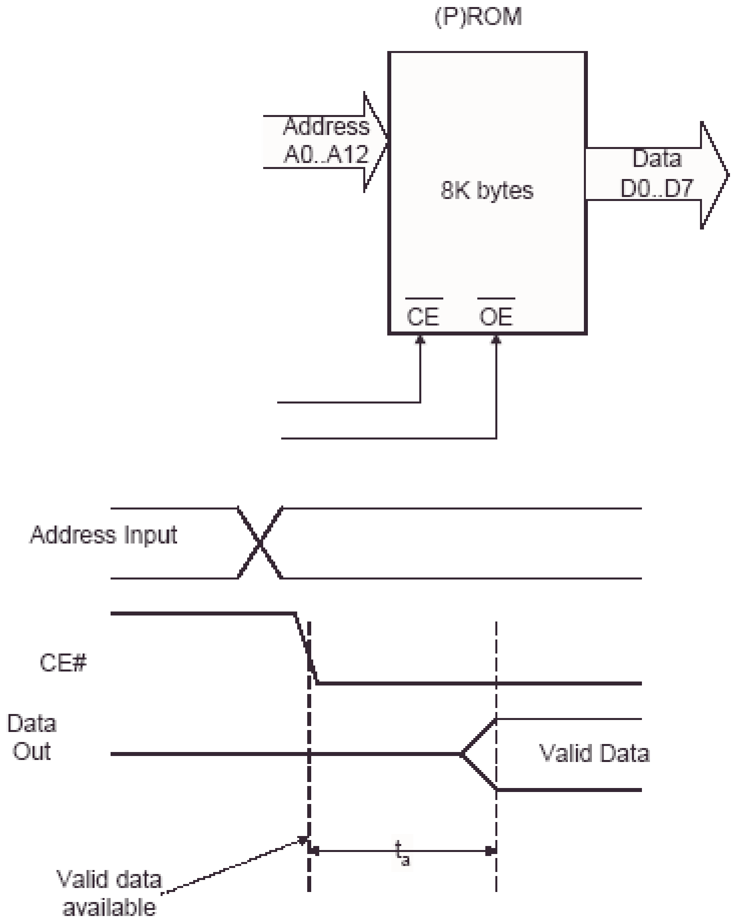


Figure 88:

(Note that we're ignoring the effect of the OE# signal for the moment.) The figure shown above gave us the time that it takes to get data out of the ROM. Now in any given microprocessor based system we also need to look at the microprocessor's read cycle and we need to see if this matches up to the read cycle from the memory device. If the microprocessor's memory cycle is very short, then the data from the memory device may not be properly available at the point when the processor reads from the data bus. In fact this is the central aspect of memory sub-system design. Let's look at the read cycle a little more closely.

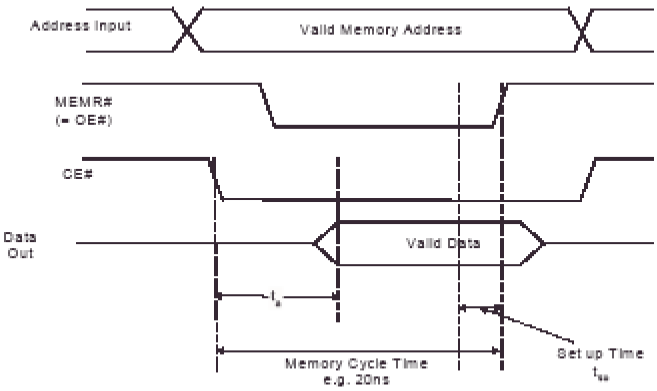


Figure 89:

We're considering the case where MEMR# is connected directly to the ROM's OE# signal. The microprocessor reads the data on the rising edge of the MEMR# signal. In order for the microprocessor to read the memory correctly the data must be valid before the rising edge of the MEMR# signal. In fact the manufacturer will also specify that the data be valid for a setup time before the MEMR# comes high. It is also necessary to meet this set up time in order for the microprocessor to read the data correctly. Therefore the access time of the memory device must be small enough to be less than the memory cycle time less the read set up time specified by the microprocessor manufacturer. The manufacturer will also specify a hold time after the MEMR# signal goes high – the memory design must keep the data on the bus valid for a time equal to or greater than the time specified in the microprocessor data sheet.

Due to technology constraints, costs, etc it is often necessary to use memory devices which have an access time that is greater then the memory cycle time. In this case it is necessary to introduce 'wait states' which stretch out the processor memory cycle so that the rising edge of MEMR# will not be asserted until the data from the memory has become valid. A memory controller circuit may be used to introduce wait states, which will normally last for one or more basic processor clock cycles. The memory controller may be part of the processor support chip set or may be incorporated into the microprocessor itself.

Notes on Memory and I/O Access Times.
There are other key memory device timing parameters for ROMs and RAMs but 'ta' is the most widely quoted figure-of-merit for classifying memory devices. We will go into this in more detail later.

Wait states are also used in memory write cycles.

The same types of access time issues arise in accessing I/O ports (in fact they're even more common for I/O), so wait states are often introduced during I/O access cycles too.
Note that 'ta' and other device timing parameters vary with temperature so it is important to provide some timing margins when implementing a design.

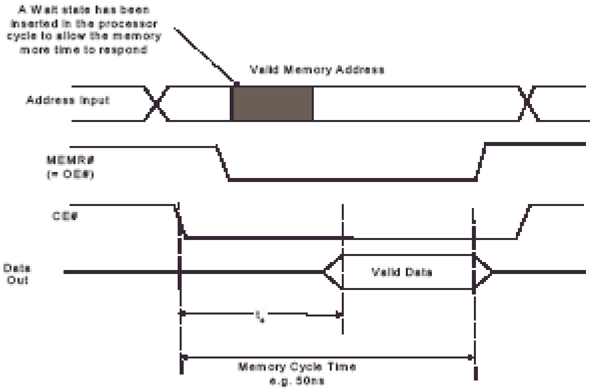


Figure 90:

Pre-fetching from memory:
We looked at the Fetch-Decode-Execute cycle earlier. We just have seen that Fetching data from memory is a comparatively slow process, and it turns out to be a big factor in slowing down the performance of a microprocessor. Consider the following time line for a microprocessor following this cycle:

Fetch1	Decode1	Execute1	Fetch2	Decode2	Execute2	Fetch3	Decode3	Execute3

Figure 91:

In practice, the decode-execute stages don't need to use the external bus, so we can get a speed-up if we can arrange the CPU so that the Fetch cycle goes in parallel with the other cycles:

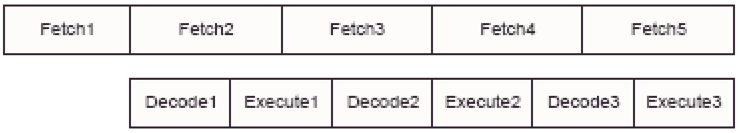


Figure 92:

Since the decode, execute stages don't need the bus, the bus interface can be used to fetch the next instruction while the processor control unit is decoding and executing the current one. To do this it is necessary to modify the CPU structure slightly, to add a buffer or queue where the prefetched instructions can be stored before they are executed. It looks like this:

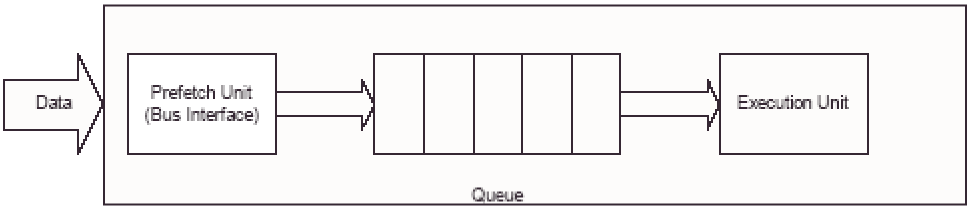


Figure 93:

There is a price to be paid: the prefetcher fetches data sequentially, so if the program encounters a branch or a jump, the Queue has to be emptied (flushed) and refilled from the jump destination address. However, this price is worth paying. Later we'll see how more powerful processors generalize this technique into an instruction pipeline, and we'll also look at how a Pentium attempts to deal with the problems caused by jumps and branches.

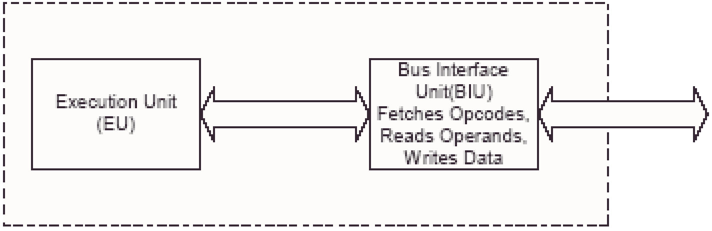
Self-Assessment 1-4

1) Using simple gates (AND, OR, INVERTER) design a memory decoder that addresses each device according to the memory map shown above.
2) Sketch Input and Output processor cycles using the memory read and write cycles as a guide.

SESSION 2-4: THE 8086 AND 8088 MICROPROCESSORS; INTERRUPTS AND DM

The first IBM PC was based on the i8088 processor. This device is fully software compatible with the i8086, but it has an 8-bit external data bus (16-bit internally) and has a 4 byte instruction prefetch queue. The 8086 and 8088 can really be considered together for our purposes. The 8086/8088 microprocessor consists of two internal units: the execution unit (EU), which executes the instructions, and the bus interface unit, which fetches instructions, reads operands and writes results. This is sketched below (from Floyd's book). The BIU allows some overlapping between instruction fetching and execution.

The following figure shows the internal organisation of the 8086/8088:


Execution Unit (EU)
Bus Interface Unit(BIU) Fetches Opcodes, Reads Operands, Writes Data

8086/8088 MPU

8086/8088 Architecture.

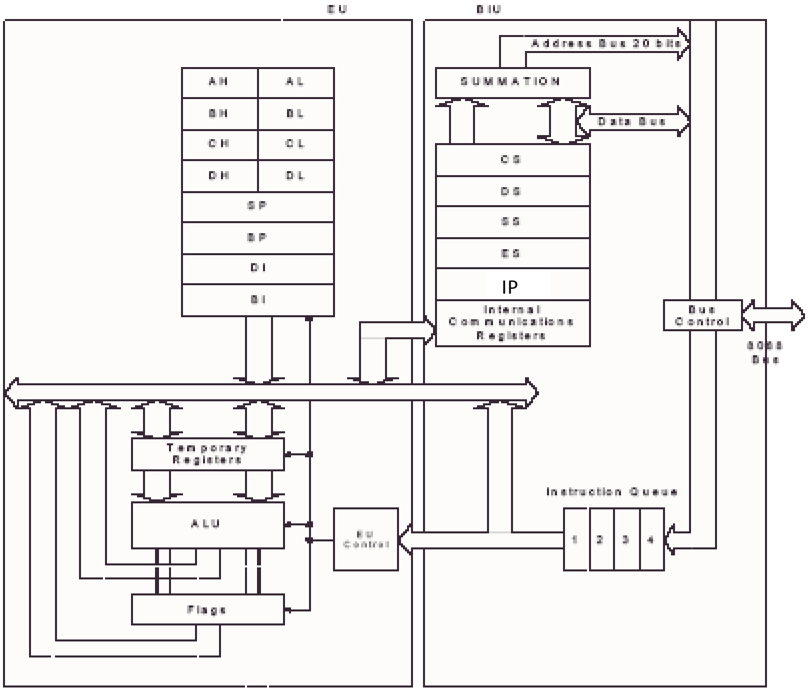


Figure 94:

The Bus Interface Unit (BIU) consists of the following:
Instruction Queue: this allows the next instructions or data to be fetched from memory while the processor is executing the current instruction. The memory interface is usually much slower than the processor execution time, so this decouples the memory cycle time from the execution time.

Segment Registers: The Code Segment (CS), Data Segment (DS), Stack Segment (SS) and Extra Segment (ES) registers are 16-bit registers, used with the 16-bit Base registers to generate the 20-bit address required to allow the 8086/8088 to address 1Mb of memory. They are changed under program control to point to different segments as a program executes. The Segmented architecture was used in the 8086 to keep compatibility with earlier processors such as the 8085. It is one of the most significant elements of the Intel Architecture.

The Instruction Pointer (IP) and Address Summation: The IP contains the Offset Address of the next instruction, which is the distance in bytes from the base address given by the current Code Segment (CS) register. The figure shows how this is done.

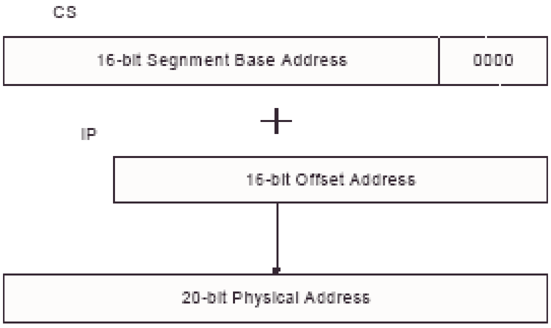
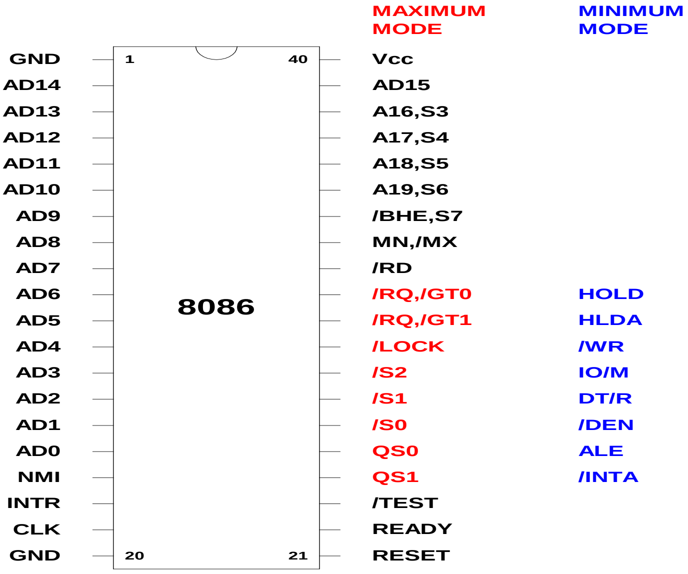


Figure 95:

The contents of the CS are shifted left by four. Bit 15 moves to the Bit 19 position. The lowest four bits are filled with zeros. The resulting value is added to the Instruction Pointer contents to make up a 20-bit physical address. The CS makes up a segment base address and the IP is looked as an offset into this segment. This segmented model also applies to all the other general registers and segment registers in the 8086 device. For example, the SS and SP are combined in the same way to address the stack area in physical memory.

2-4.1 The 8088 Microprocessor in Circuit

The 8086 and 8088 microprocessors require some additional support circuit to operate in a microcomputer system. For example, to fit in a 40-pin package, these processors multiplex the address and data buses on the same pins. So some demultiplexing logic is needed to build up separate address and data buses that can interface with devices like RAMs and ROMs. In fact, in Maximum Mode the 8086 or 8088 needs to be supported by at least the following devices: 8288 Bus Controller, 8284A Clock Generator as well as 74HC373 and 74HC245 devices. The block diagram is as follows:



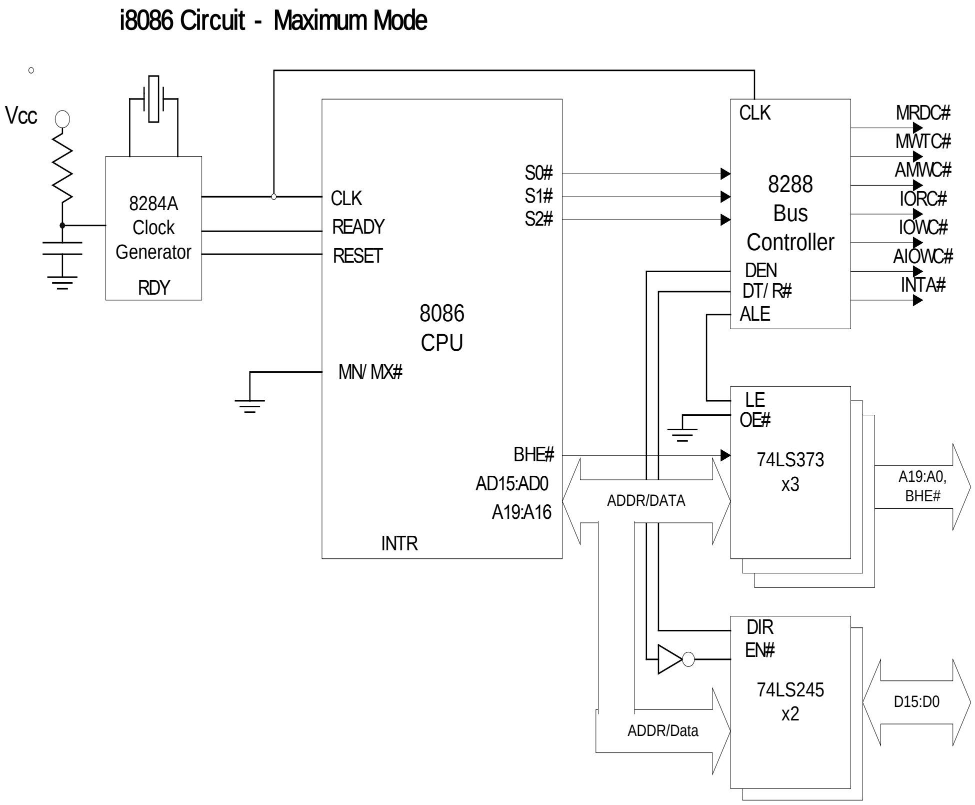
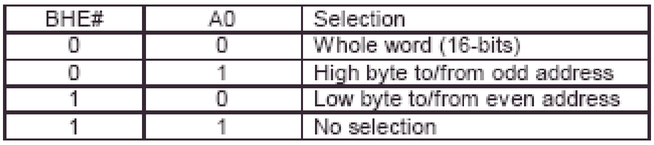


Figure 96:

Note that in Maximum mode, the 8288 uses a set of status signals (S0, S1, S2) to rebuild the normal bus control signals of the microprocessor (MRDC#, MWTC#, IORC#, IOWC# etc). We'll just look at the function of a few 8086/8288 pins briefly:

RESET: The processor reset input pin, clears the flags register, segment registers etc., sets the effective program address to FFFF0h (CS=FFFFh, IP=0000). BHE# (8086 only): Bus High Enable (BHE#) is used with the A0 signal to help interface the 16-bit data bus of the 8086 to byte-wide memory. The table shows how BHE# and A0 are used together:



BHE#	A0	Selection
0	0	Whole word (16-bits)
0	1	High byte to/from odd address
1	0	Low byte to/from even address
1	1	No selection

This is how memory is accessed using these signals:

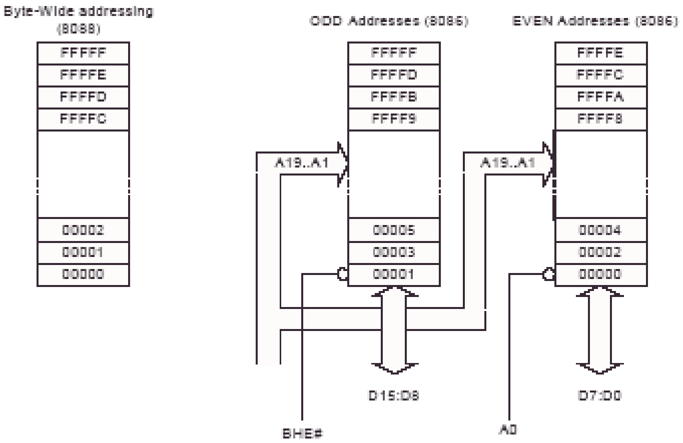


Figure 97:

This scheme applies even when16-bit memories are used. It allows the 8086 to access byte data. Similar schemes allow 32-bit processors like the 80386 to access byte data

ALE (Address Latch Enable): On both the 8086 and 8088 processors the address and data buses are multiplexed. This means that the same pins are used to carry both address and data information – at different times during the read or write cycle. At the start of the cycle the address/data bus carries the address signals, while at the end of the cycle the pins are used for the data bus. The ALE signal is used to allow external logic to LATCH the addresses while the AD lines carry address data and hold those addresses so that they can be applied to the other devices in the system. The address latches used are 74HC373 or equivalent parts.

Unlike a flip flop, the 74HC373 is a transparent latch. When its Latch Enable (LE) signal is high the 74HC373 outputs follow its inputs. When the LE signal goes low, the 74HC373 outputs hold the data that the inputs had prior to when the LE signal went low. The ALE signal is connected directly to the LE input of the 74HC373. The diagram below shows this:

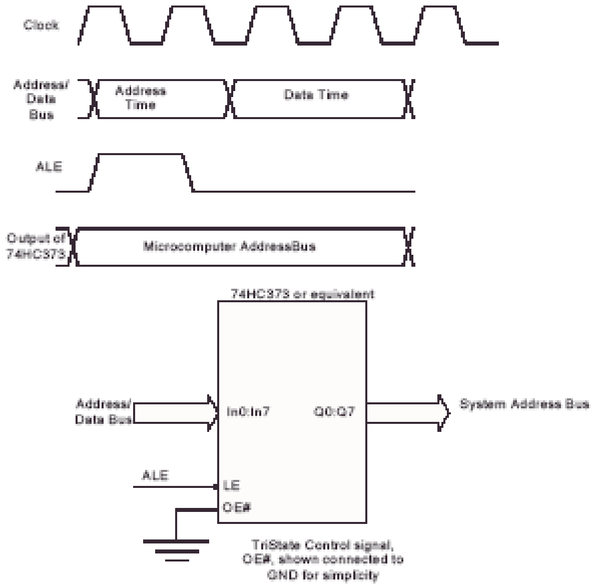


Figure 98:

The 8288 Bus Controller generate the DEN# and DT/R# signals that control the Enable and Direction Control inputs, respectively, of a 74HC245 bi-directional transceiver, which is placed between the processor’s address/data bus and the data bus of the microcomputer.

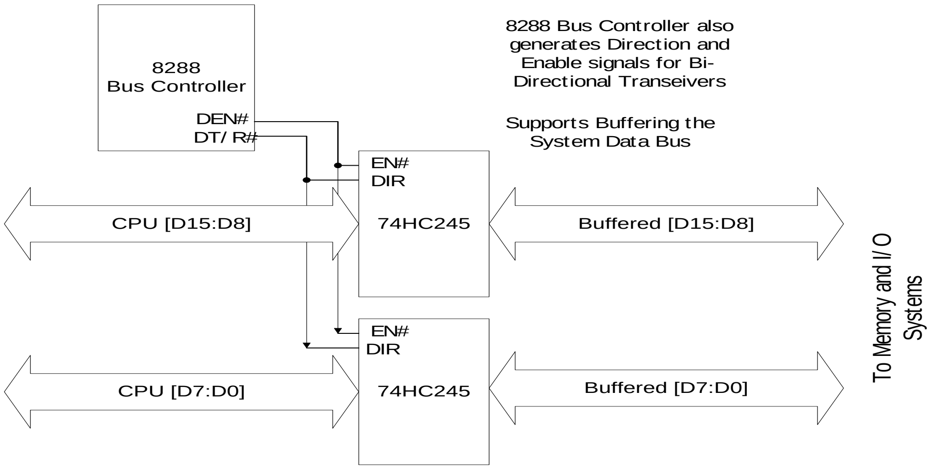

8288
Bus Controller
DEN#
DT/ R#
8288 Bus Controller also generates Direction and Enable signals for Bi-Directional Transeivers
Supports Buffering the System Data Bus
EN#
DIR
74HC245
CPU [D15:D8]
Buffered [D15:D8]
EN#
DIR
74HC245
CPU [D7:D0]
Buffered [D7:D0]
To Memory and I/O Systems

Figure 99:

Can you see why multiplexed address/data buses had to be abandoned in high performance Intel processors?

2-4.2 8086/8088 Memory and I/O Cycles

The following figures show an 8086/8088 read, write and I/O cycles. Recall that microprocessors are clocked synchronous state machines, that is, they perform some action when they receive a clock edge. That’s why we show the clock signal in the following diagrams: the processor bus unit takes some action when the clock signal occurs.

Bus Read Cycle (Memory or I/O)

- The 4 processor clock cycles are called T states. Four cycles is the shortest time that the processor can use for carrying out a read or an input cycle.
- At the beginning of T1, the processor outputs S2, S1, S0, A16/S3…A19/S6, AD0..AD15 and BHE#/S7.
- The 8288 bus controller transitions the ALE signal from low to high, thereby allowing the address to pass through the transparent latches (74HC373). The address, along with the BHE# signal is latched when ALE goes low, providing the latched address A0..A19.
- During T2 the processor removes the address and data. S3..S6 status is output on the upper 4 address/status lines of the processor.
- The AD0..AD15 signals are floated as inputs, waiting for data to be read.
- Data bus transceivers (74HC245) are enabled towards the microprocessor (the READ direction) by the DT/R# and DEN signals.
- The MRDC# (ie MEMR#) or IORC# (IOR#) signal is asserted.
- The signals are maintained during T3. At the end of T3 the microprocessor samples the input data.
- During T4 the memory and I/O control lines are de-asserted.

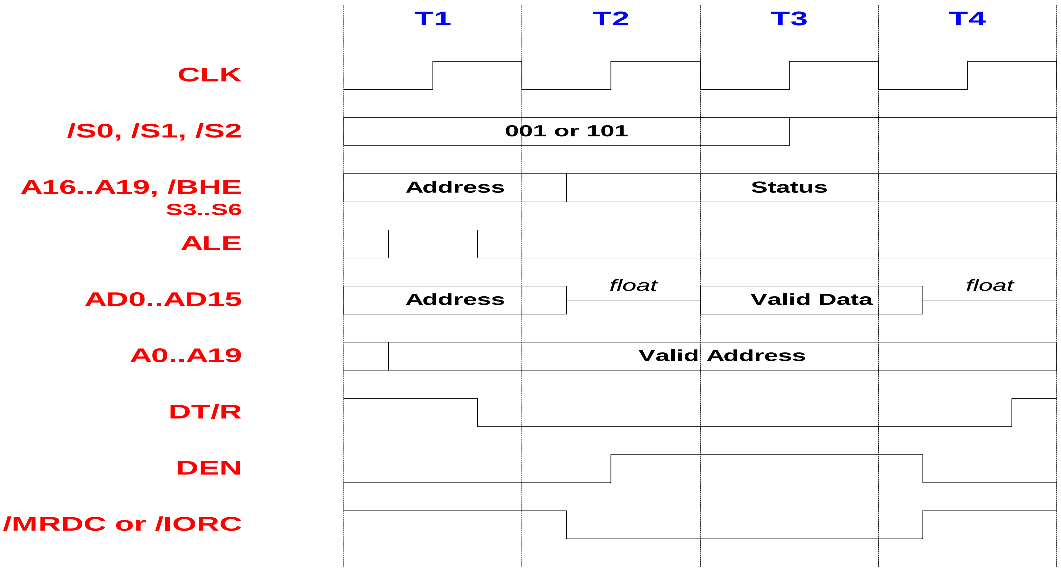


Figure 100:

Bus Write Cycle (Memory or I/O)

- The 4 processor clock cycles are called T states. Four cycles is the shortest time that the processor can use for carrying out a write or an output cycle.
- At the beginning of T1, the processor outputs S2, S1, S0, A16/S3…A19/S6, AD0..AD15 and BHE#/S7.
- The 8288 bus controller transitions the ALE signal from low to high, thereby allowing the address to pass through the transparent latches (74HC373). The address, along with the BHE# signal is latched when ALE goes low, providing the latched address A0..A19.
- During T2 the processor removes the address and data. S3..S6 status is output on the upper 4 address/status lines of the processor.
- Output data is driven out on the AD0..AD15 lines.
- Data bus transceivers (74HC245) are enabled away from the microprocessor (the WRITE direction) by the DT/R# and DEN signals.

- The MWRC# (ie MEMW#) or IOWC# (IOW#) signal is asserted at the beginning of T3.
- The signals are maintained during T3.
- During T4 the memory and I/O control lines are de-asserted. In simple Intel Architecture systems, the data is usually written to the memory or output device at the rising edge of the MWRC# or IOWC# signal.

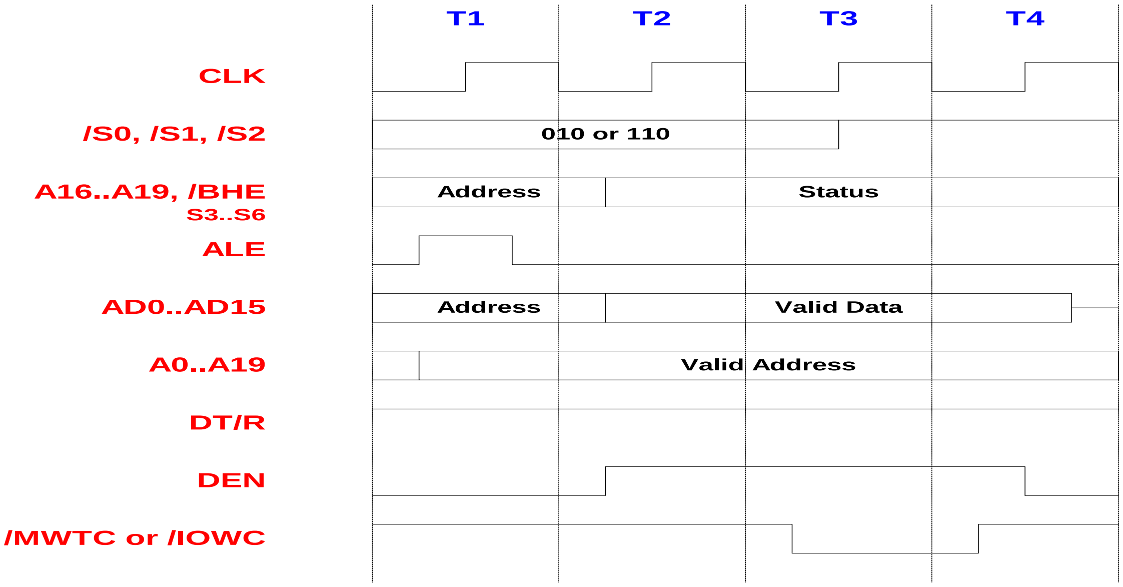


Figure 101:

Wait States

Wait States are used to help interface to slow memory or I/O devices. The READY input signal on the 8086 is used to insert wait states into the processor bus cycle, so that the processor stretches out its read or write cycle, to accommodate the slow device.

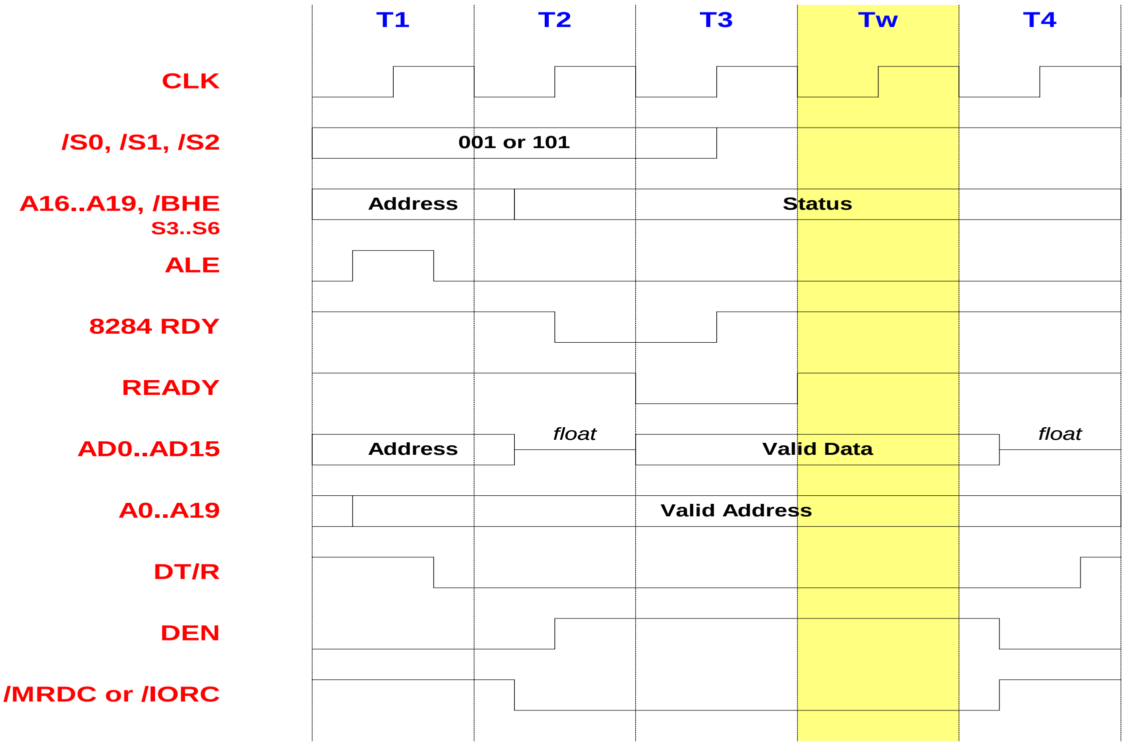


Figure 102:

Generating Wait States

The normal memory or I/O cycle on an 8086 is 4 clocks long – T1 to T4. Wait states, called Tw can be inserted in the bus cycle as follows:

The 8086 READY line is sampled at the rising edge of T3. If READY is low, a WAIT state is inserted. During the WAIT state the READY is sampled again at the next rising edge of the clock, and another WAIT is inserted if READY is still low. A number of further WAIT states can be inserted in this way. The memory or I/O device can initiate WAIT state generation by bringing a RDY signal low. To synchronise the 8086 READY signal and to ensure that the 8086 timing requirements are met the memory device's RDY signal is normally connected to the 8284's RDY input. The memory device needs to bring RDY low prior to the rising edge of the 8086's T2 clock. The 8284 drives the 8086 READY signal low at the falling edge of T2. When the 8086 samples READY at the rising of T3 it finds that it is low, and it inserts a WAIT state for the next clock state. The memory device has to bring RDY high early in T3 so that the 8284 can bring READY high before the rising edge of T3 if another WAIT state is to be avoided. The figure below shows a read or input cycle using WAIT states.

2-4.3 Interrupts

There are a number of cases in which we need to be able to halt the normal flow of instructions that a microprocessor is executing. Exceptions may be due to software or hardware events. We will discuss hardware interrupts for the moment. We use hardware interrupts so that an external event can halt the currently executing set of instructions in order to cause the microprocessor to perform some specific actions. We say that the interrupt occurs asynchronously as the interrupt can be asserted at any time. An interrupt may be asserted by an external device to indicate to the CPU that it is ready to transfer data. For example, serial ports often use this technique. The interrupt also allows certain events to have a higher priority than the currently running task – e.g. the serial port may want to indicate that its I/O buffer has been filled up and needs to be emptied so that new data is not missed.

Suppose an I/O device generates a signal called an interrupt request. In the small CPU model we looked at earlier the INTR control line indicated an interrupt request. When the processor accepts the interrupt request it suspends the operation of its current program, *services* the interrupt and then returns to the program it was running. The processor does not have to waste time interrogating the status of the I/O device. Now suppose that the currently executing program is very important (has a high priority). In this case we may not want it to be interrupted. It is possible to prevent interrupts from being accepted by the processor by clearing the *interrupt enable* flag in the status register. In x86 processors, the CLI instruction clears the interrupt enable flag while STI sets it. Note that most I/O devices also have local interrupt enable flags that allow their interrupts to be blocked out while interrupts from other devices can still be accepted.

Maskable and Non-Maskable Interrupts

An interrupt that can be disabled by means of a flag is called a *maskable interrupt* in that the programmer can decide to ignore the interrupt by applying a mask.

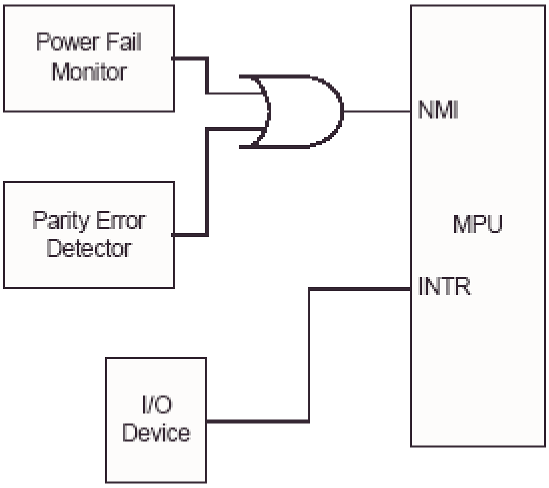


Figure 103:

However, most microprocessors also have a separate interrupt line – the Non-Maskable Interrupt (NMI), which cannot be masked out. This is the top priority interrupt and is usually used to interrupt program execution in response to an important, usually critical, external event like a power fail signal or a parity error signal. This is illustrated below:
On receipt of an interrupt the CPU automatically carries out the following sequence of operations:

- ✓ Complete the execution of the current instruction
- ✓ Push the contents of the status (flags) register on to the stack
- ✓ Disable further interrupts by clearing the interrupt enable flag in the status register
- ✓ Push the contents of the Instruction Pointer onto the stack. This now points to next instruction in the program – i.e. the one that would have been executed if the interrupt had not occurred.
- ✓ Trap to the start address of the Interrupt Service Routine (ISR). Different processor architectures do this in many different ways but in a very simple model we'll consider that this happens by loading a fixed new address, which points to the start of the ISR into the IP

The interrupt service routine (ISR) or interrupt handler is a user-written program, which is executed when the interrupt has been accepted. The programmer must include the following operations into the ISR:

- ✓ Save onto the stack the contents of any registers used by the ISR, so as to preserve data (entry sequence)
- ✓ Execute the 'body' of the ISR. This is the activity required by the interrupting device. Exit Sequence: Pop the registers from the stack in the same order they were pushed so as to restore the original contents
- ✓ (Optional*) Enable interrupts again by resetting the interrupt flag, thereby allowing further interrupts to be serviced*
- ✓ Issue a Return From Interrupt (IRET) instruction.

On returning from an interrupt the processor automatically carries out the following operations:

- ✓ Pop the status register from the stack
- ✓ Pop the Instruction Pointer from the stack
- ✓ Now because the IP contains the next instruction address of the interrupted program the

program resumes execution from the point where it was interrupted.

Note that in many cases it is not necessary to re-enable interrupts explicitly as popping the status register restores the original state (enabled) of the I flag. In some processors (e.g. Z80) this was not the case so that it was necessary to reset the flag explicitly.
Figure X1 shows the sequence of events for a simple single source interrupt.

This is a simplified view of processor interrupts. We will discuss x86 interrupts in greater detail later on.

Note 1. The normally running program is called the *'main program'* to distinguish it from the ISR. Another term you may come across is *'background task'*.

Note 2. Although the main program has been interrupted it behaves logically as if the interrupt had never occurred, because its full status has been preserved. All the internal processor registers have been restored to the values they had before the interrupt occurred. So logically the program is not changed, except that its temporal behaviour has been affected by the interrupt – some time has been lost. Note also that it's up to the programmer to ensure that the Return sequence of the ISR restores everything to its previous state.

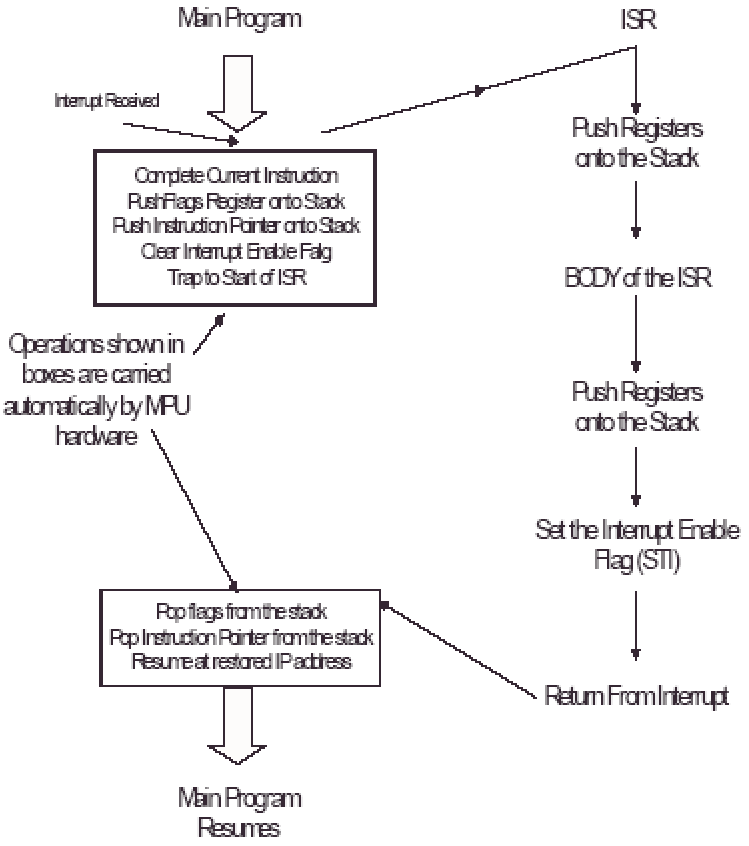

Main Program
ISR
Interrupt Received
Complete Current Instruction
Push Flags Register onto Stack
Push Instruction Pointer onto Stack
Clear Interrupt Enable Falg
Trap to Start of ISR
Push Registers onto the Stack
BODY of the ISR
Operations shown in boxes are carried automatically by MPU hardware
Push Registers onto the Stack
Set the Interrupt Enable Flag (STI)
Pop flags from the stack
Pop Instruction Pointer from the stack
Resume at restored IP address
Return From Interrupt
Main Program Resumes

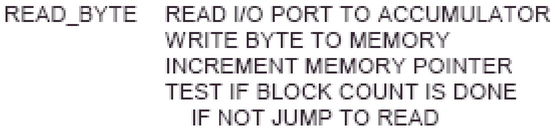
Figure 104:

Programming with interrupts enabled can be very tricky and requires considerable attention to the Real-Time aspects of the system being designed.

2-4.4 Direct Memory Access (DMA)

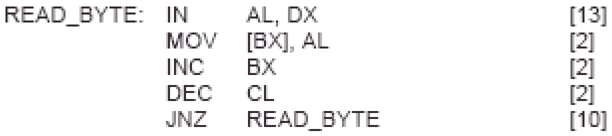
Direct Memory Access (DMA) techniques are designed to improve system performance by allowing external devices to transfer data directly to or from memory under hardware control. This differs from the interrupt method which is a software method and therefore is slower. The data rates required by disk drives, network cards or graphic devices are greater than can be achieved under normal program control.

Consider the transfer of a block of data from I/O to memory under program control. For each byte transferred the following operations are required: I/O to memory block transfer example:



```
READ_BYTE   READ I/O PORT TO ACCUMULATOR
            WRITE BYTE TO MEMORY
            INCREMENT MEMORY POINTER
            TEST IF BLOCK COUNT IS DONE
              IF NOT JUMP TO READ
```

On an i80386 microprocessor the assembly language to implement code might be as follows:



```
READ_BYTE:  IN    AL, DX             [13]
            MOV   [BX], AL           [2]
            INC   BX                 [2]
            DEC   CL                 [2]
            JNZ   READ_BYTE          [10]
```

The processor takes a number of clock cycles to execute each instruction. In this case the byte transfer takes 29 clock cycles. At a 20MHz clock (say), it will take 1.45us to complete each transfer, which represents a data rate of 1/1.45us = 670Kbytes/sec.
[f = 20MHz; t = 1/f = 50ns; 29 x 50ns = 1450ns = 1.45us]
This example assumed no memory WAITs and no delays in the I/O channel. In reality the performance would be worse than shown. The example shows that a 20MHz processor can only transfer bytes at a rate of less than or equal to 670Kbytes. High

Performance I/O devices require better performance than this.
DMA has the potential to allow data transfer at memory cycle rates. For a RAM cycle time of 100ns (time to read or write one byte), the DMA byte transfer could be up to 10MB/sec (i.e. 1/(100)ns). DMA has the potential to support data rates many times greater than program controlled I/O.

2-4.4.1 DMA Methods

DMA Transfer is controlled by a separate hardware device called a DMA Controller. The DMA Controller takes over the MPU buses. During a DMA transfer operation the MPU relinquishes control of its memory and allows the DMA Controller to control the data transfer operations. DMA Controllers can be programmed to operate in SINGLE BYE and BLOCK TRANSFER Modes.

4.3.2.2 The DMA Interface with the Microprocessor

The DMA Controller HOLD signal initiates the take-over of control from the CPU. When it receives this signal, the CPU suspends operation and effectively disconnects itself from the address, data and control buses, putting these lines into a high impedance state. The microprocessor issues an acknowledge signal – HLDA – to the DMA when it has given up control. Note that for DMA transfers, unlike the case of interrupt processing, the processor does not have to save its CPU registers when it gives up control of its buses. Therefore the response time to a DMA request is much faster than the response to an interrupt request.

4.3.2.3 The DMA Interface with the I/O Device

Two signals are used – DMA REQUEST (DREQ) and DMA ACKNOWLEDGE (DACK). When the I/O device has data ready to read or write from memory it issues a DERQ signal to the DMA Controller. The DMA controller responds by placing the microprocessor into a high impedance state (using HOLD, HLDA). The DMA controller then issues the DACK signal to the I/O device and at the same it takes over bus control by placing an address out on the address bus and using the control lines to carry out an I/O – memory transfer.

4.3.2.4 DMA Control Signals for Data Transfer

The DMA Controller takes control of the MEMR#, MEMW#, IOR# and IOW# signals.
To READ from MEMORY to I/O: MEMR# and IOW# active
To WRITE to MEMORY from I/O: MEMW# and IOR# active.
A system with a DMA controller is shown below.
The DMA does not bother with the data bus: it simply transfers data and is not interested in its content.

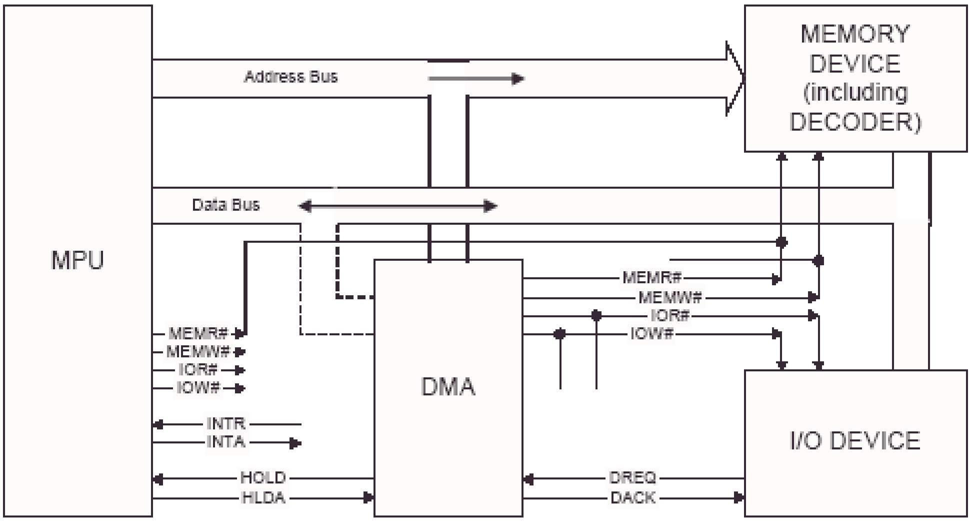


Figure 105:

Can you see why the data bus goes to the DMA Controller?

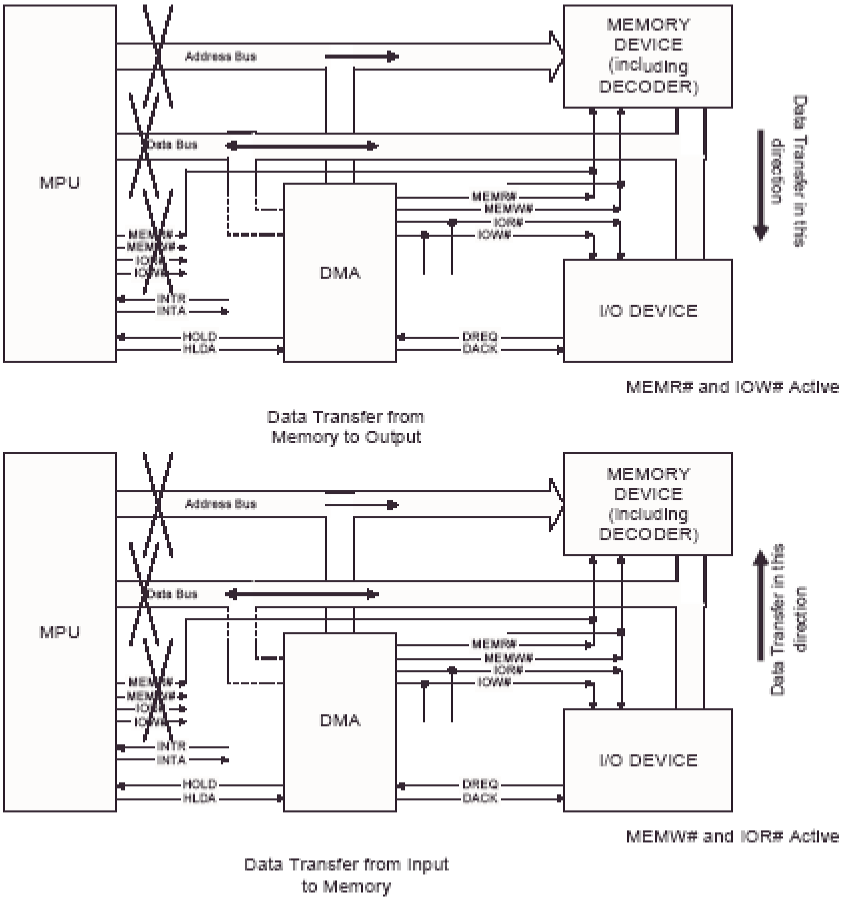
DMA Transfer and Timing

Before a DMA Transfer can occur the DMA controller needs to be initialised so that it ‘knows’ exactly what to do when a DMA request is made. The initialisation procedure does the following:

- ✓ Define read from or write to memory
- ✓ Define number of bytes to be transferred
- ✓ Define memory address for start of transfer
- ✓ Enable the DMA

The sequence of operations for a DMA transfer is summarised below:

1. The I/O device initiates a transfer by asserting the DREQ line, thereby requesting the MPU to give up control of its buses.
2. The DMA Controller issues a HOLD (Request) signal to the MPU.
3. The MPU responds by issuing a HLDA (Hold Acknowledge) signal back to the DMA Controller. The MPU issues the HLDA signal *after* it has relinquished its buses (i.e. put its bus drivers into high impedance or tri-state).
4. The DMA Controller sends a DACK (DMA Acknowledge) back to the requesting device.
5. The data transfer takes place and the relevant MEMR#, MEMW#, IOR# or IOW# signals are asserted according to the transfer in question.



MEMR# and IOW# Active

Data Transfer from Memory to Output

MEMW# and IOR# Active

Data Transfer from Input to Memory

Figure 106:

2-4.5 Microcomputer Performance Considerations

The simple 8-bit MPU that we used for illustrations represents a typical low-cost, low-performance microprocessor. Although 8-bit processors running at 50MHz are available, mostly they are clocked at frequencies less than about 25MHz. Let's look at some features of our model:
Processor Clock Frequency: 4 MHz ALU Register width (precision): 8 bits Data Bus width: 8 bits Address Bus width: 16 bits
Each of these characteristics limits the processor performance in some way.

2-4.5.1 Processor Clock Frequency

State of the Art PCs are clocked at over 1GHz (Jan 2001). By increasing the clock frequency we can increase the frequency at which the CPU can execute instructions. This is an obvious way of increasing performance. However, memory and I/O devices typically cannot ‘keep up’ with very high frequency CPUs, so the access times of the memory and I/O devices can become a performance bottleneck unless some extra measures are taken.

ALU Register Width The ALU operates at a precision of N bits, where N is typically 4, 8, 16 32, 48, 56 or 64 bits. We say that a processor is an N-bit processor, where N represents the precision of the ALU. Usually the internal registers will also be at least N bits long. For example a 32-bit processor can add two 32-bit numbers by performing 2 memory reads, and a simple addition operation. By contrast, an 8-bit processor would have to perform 4 reads for each operand and would also have to apply multiple-precision (slower) arithmetic to perform the calculation.

2-4.5.2 Data Bus Width

If we had a 32-bit processor but were confined to an 8-bit bus we would run into a considerable performance bottleneck. Internally the CPU could perform 32-bit arithmetic but it would waste a lot of time reading and writing memory in byte-wide fashion – incurring 4 memory cycles to read or write 32 bits of data. The wider the data bus the faster we can transfer data (more bits per cycle). Since the memory and I/O device access times are finite, the more bits transferred per cycle the better.

2-4.5.3 Address Bus Width

Increasing the address bus width does not provide a ‘speed’ increase but it does provide the ability to increase the range of memory that the CPU can directly address. This is important because modern high performance PCs use big programs, which would not fit in a smaller address space. Although there are various methods for overcoming this limitation, all these methods take time, which impacts on the overall system performance.

16-bit address bus can address 64K locations
20-bit address bus can address 1M locations
24-bit address bus can address 16M locations
32-bit address bus can address 4G locations
A wider address bus supports a greater address space.

Self-Assessment 2-4

1. The CS contains A820, while the IP contains CE24. What is the resulting physical address?
2. The CS contains B500, while the IP contains 0024. What is the resulting physical address?

Learning Track Activities

Summary

Unit Summary

1.

Key terms in Unit

Review Question

Which mapping system is preferable in your opinion? What is the most important factor that affects your choice?

Unit Assignment – 4

1) At the start of the following sequence the Stack Pointer has the value C000h. The following code is executed

```
PUSH AL       ; Push 8 bit accumulator data
PUSH PSW      ; Push 8 bit flags register
```

What is the value of the SP at this point?

The following instructions are executed without any further stack activity in the meantime

```
POP PSW       ; Restore 8 bit flags register
POP AL        ; Restore 8 bit accumulator data
```

2) What is the value of the SP at this point? Note how the POP order is the reverse of the PUSH order.

TUTORIAL

ASSEMBLY LANGUAGE

Introduction

This tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. Of course if you have knowledge of some high level programming language (java, basic, c/c++, Pascal...) that may help you a lot. But even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.

Learning objectives

Tutorial Content

Session 1-T: 8086 Assembly Language

Session 2-T: Memory Access

Session 3-T: Variables

Session 4-T: Interrupts

Session 5-T: Library of common functions - emu8086.inc

Session 6-T: Arithmetic and Logic Instructions

Session 7-T: Program flow control

Session 8-T: Procedures

Session 9-T: The Stack

Session 10-T: Macros

Session 11-T: Controlling External Devices

SESSION 1-T: 8086 ASSEMBLY LANGUAGE

It is assumed that you have some knowledge about number representation (hex/bin).

1-T.1 What Is Assembly Language?

Assembly language is a low level programming language. You need to get some knowledge about computer structure in order to understand anything. The simple computer model as i see it:

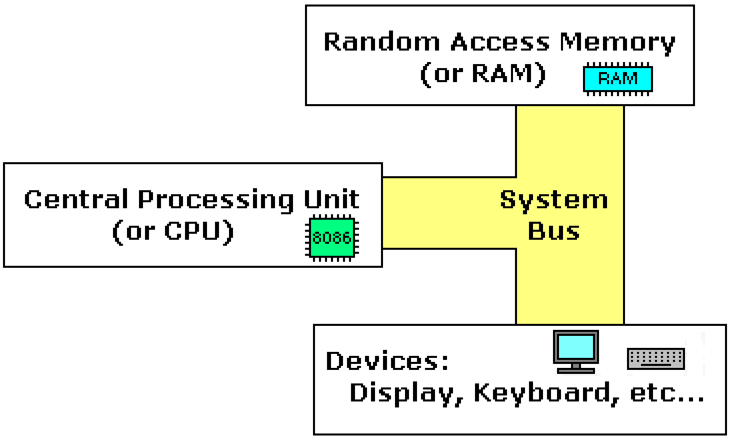


Figure 107:

the **system bus** (shown in yellow) connects the various components of a computer. The **CPU** is the heart of the computer, most of computations occur inside the **CPU**. **RAM** is a place to where the programs are loaded in order to be executed.

1-T.2 Inside The CPU

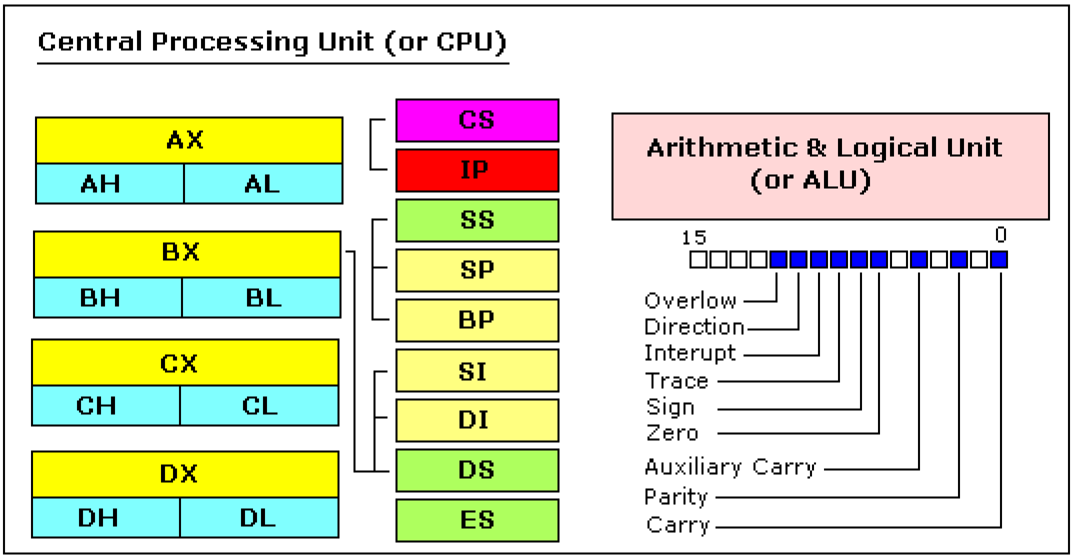


Figure 108:

1-T.2.1 General purpose registers

8086 CPU has 8 general purpose registers, each register has its own name:

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, it's something like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.

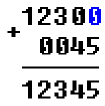
Four general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for low part.
Because registers are located inside the cpu, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

1-T.2.2 Segment registers

- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.

- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory. Segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values. CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h * 10h + 45h = 12345h):



```
 12300
+ 0045
 -----
 12345
```

the address formed with 2 registers is called an **effective address**.

By default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address! Also, although **BX** can form an effective address, **BH** and **BL** cannot.

1-T.2.3 Special purpose registers

- **IP** - the instruction pointer.
- **flags register** - determines the current state of the microprocessor.

IP register always works together with **CS** segment register and it points to currently executing instruction. **Flags register** is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. Generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

SESSION 2-T: MEMORY ACCESS

To access memory we can use these four registers: **BX, SI, DI, BP**. combining these registers inside **[ ]** symbols, we can get different memory locations. These combinations are supported (addressing modes):

Table 18:

[BX + SI]	[SI]	[BX + SI + d8]
[BX + DI]	[DI]	[BX + DI + d8]
[BP + SI]	d16 (variable offset only)	[BP + SI + d8]
[BP + DI]	[BX]	[BP + DI + d8]

[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

d8 - stays for 8 bit signed immediate displacement (for example: 22, 55h, -1, etc...)
d16 - stays for 16 bit signed immediate displacement (for example: 300, 5517h, -259, etc...).
Displacement can be an immediate value or offset of a variable, or even both. If there are several values, assembler evaluates all values and calculates a single immediate value.
Displacement can be inside or outside of the **[ ]** symbols, assembler generates the same machine code for both ways. Displacement is a **signed** value, so it can be positive or negative.

Generally the compiler takes care of the difference between **d8** and **d16**, and generates the required machine code.
For example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.
The following addressing mode—**[BX + SI] + 25**—is calculated by processor to this physical address: **100 * 16 + 30 + 70 + 25=1725**.

By default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used.
There is an easy way to remember all those possible combinations using this chart:

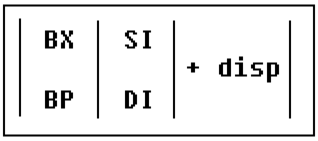


Figure 109:

You can form all valid combinations by taking only one item from each column or skipping the column by not taking anything from it. As you see **BX** and **BP** never go together. **SI** and **DI** also don't go together. Here are examples of a valid addressing mode: **[BX+5]**, **[BX+SI]**, **[DI+BX-4]**

The value in segment register (CS, DS, SS, ES) is called a **segment**, and the value in purpose register (BX, SI, DI, BP) is called an **offset**.

When DS contains value **1234h** and SI contains the value **7890h** it can be also recorded as **1234:7890**. The physical address will be 1234h * 10h + 7890h = 19BD0h.
If zero is added to a decimal number it is multiplied by 10, however **10h = 16**, so if zero is added to a hexadecimal value, it is multiplied by 16, for example:
7h = 7
70h = 112

In order to tell the compiler about data type, these prefixes should be used:
byte ptr - for byte.
word ptr - for word (two bytes).

For example:
byte ptr [BX]   ; byte access.

or
word ptr [BX]   ; word access.
Assembler supports shorter prefixes as well:
b. - for **byte ptr**
w. - for **word ptr**

In certain cases the assembler can calculate the data type automatically.

2-T.1 MOV Instruction

- copies the **second operand** (source) to the **first operand** (destination).
- the source operand can be an immediate value, general-purpose register or memory location.
- the destination register can be a general-purpose register, or memory location.
- both operands must be the same size, which can be a byte or a word.

Table 19:

These types of operands are supported:
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
Memory: [BX], [BX+SI+7], variable, etc...

Immediate: 5, -24, 3Fh, 10001101b, etc...

Table 20:

For segment registers only these types of **MOV** are supported:
MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
SREG: DS, ES, SS, and only as second operand: CS.

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

Memory: [BX], [BX+SI+7], variable, etc...

The **MOV** instruction cannot be used to set the value of the **CS** and **IP** registers.

Table 21:

Here is a short program that demonstrates the use of **MOV** instruction:

```
ORG 100h          ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h    ; set AX to hexadecimal value of B800h.
MOV DS, AX        ; copy value of AX to DS.
MOV CL, 'A'       ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh      ; set BX to 15Eh.
MOV [BX], CX      ; copy contents of CX to memory at B800:015E
RET               ; returns to operating system.
```

You can **copy & paste** the above program to the code editor, and press [**Compile and Emulate**] button (or press **F5** key on your keyboard).

The emulator window should open with this program loaded, click [**Single Step**] button and watch the register values.

How to do **copy & paste**:

1. select the above text using mouse, click before the text and drag it down until everything is selected.
2. press **Ctrl + C** combination to copy.
3. go to the source editor and press **Ctrl + V** combination to paste.

As you may guess, "**;**" is used for comments, anything after "**;**" symbol is ignored by compiler.

You should see something like that when program finishes:

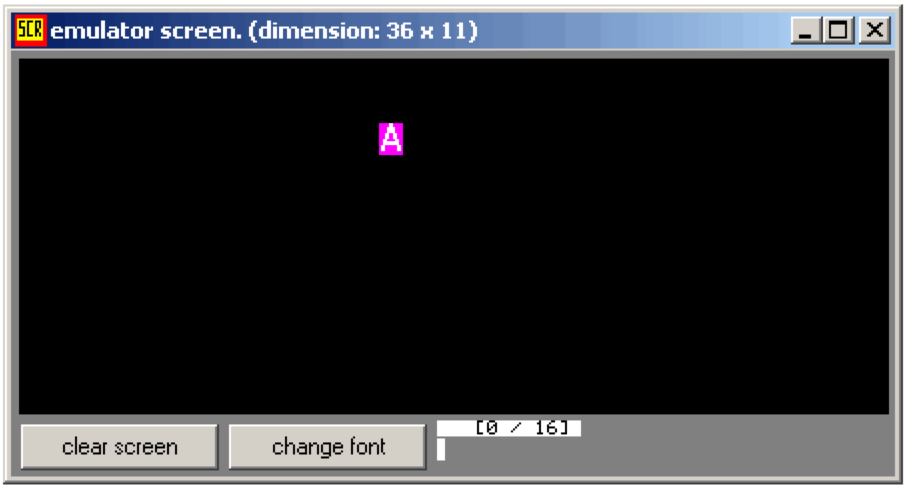


Figure 110:

Actually the above program writes directly to video memory, so you may see that **MOV** is a very powerful instruction.

SESSION 3-T: VARIABLES

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.
Our compiler supports two types of variables: **BYTE** and **WORD**.

Table 22:

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

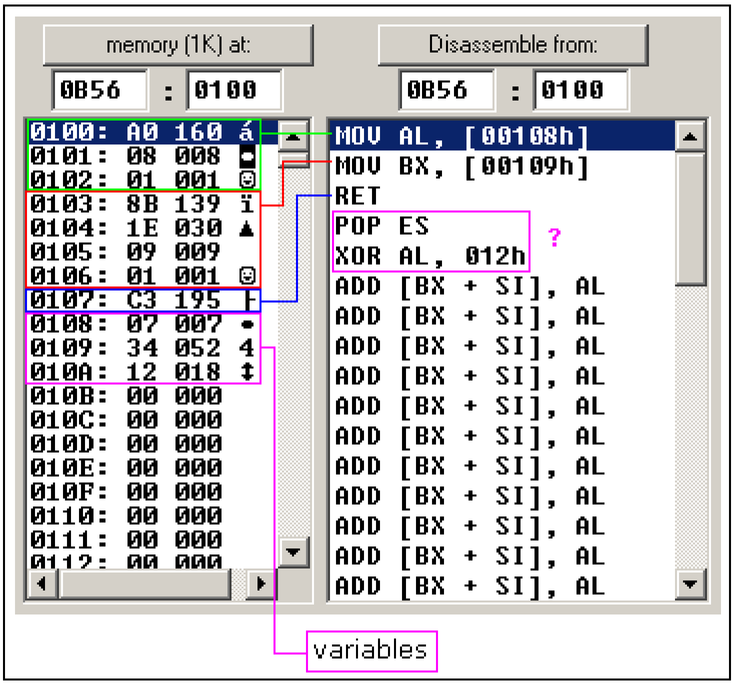
value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

As you probably know from *part 2* of this tutorial, **MOV** instruction is used to copy values from source to destination.
Let's see another example with **MOV** instruction:

Table 23:

```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

Copy the above code to the source editor, and press **F5** key to compile it and load in the emulator. You should get something like:



```
0100: A0 160 á
0101: 08 008 ◘
0102: 01 001 ☺
0103: 8B 139 ï
0104: 1E 030 ▲
0105: 09 009
0106: 01 001 ☺
0107: C3 195 ├
0108: 07 007 •
0109: 34 052 4
010A: 12 018 ↕
010B: 00 000
010C: 00 000
010D: 00 000
010E: 00 000
010F: 00 000
0110: 00 000
0111: 00 000
0112: 00 000
```

```
MOV AL, [00108h]
MOV BX, [00109h]
RET
POP ES
XOR AL, 012h
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
```

Figure 111:

As you see this looks a lot like our example, except that variables are replaced with actual memory locations. When compiler makes machine code, it automatically replaces all variable names with their **offsets**. By default segment is loaded in **DS** register (when **COM** files is loaded the value of **DS** register is set to the same value as **CS** register - code segment).
In memory list first row is an **offset**, second row is a **hexadecimal value**, third row is **decimal value**, and last row is an **ASCII** character value.

Compiler is not case sensitive, so "**VAR1**" and "**var1**" refer to the same variable.
The offset of **VAR1** is **0108h**, and full address is **0B56:0108**.

The offset of **var2** is **0109h**, and full address is **0B56:0109**, this variable is a **WORD** so it occupies **2 BYTES**. It is assumed that low byte is stored at lower address, so **34h** is located before **12h**.

You can see that there are some other instructions after the **RET** instruction, this happens because disassembler has no idea about where the data starts, it just processes the values in memory and it understands them as valid 8086 instructions (we will learn them later).

You can even write the same program using **DB** directive only:

Table 24:

```
ORG 100h

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h
```

Copy the above code to the source editor, and press **F5** key to compile and load it in the emulator. You should get the same disassembled code, and the same functionality!
As you may guess, the compiler just converts the program source to the set of bytes, this set is called **machine code**, processor understands the **machine code** and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.

Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.

Though this is true for **COM** files only, **EXE** files are loaded at offset of **0000**, and generally use special segment for variables. Maybe we'll talk more about **EXE** files later.

3-T.1 Arrays

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b is an exact copy of the a array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:

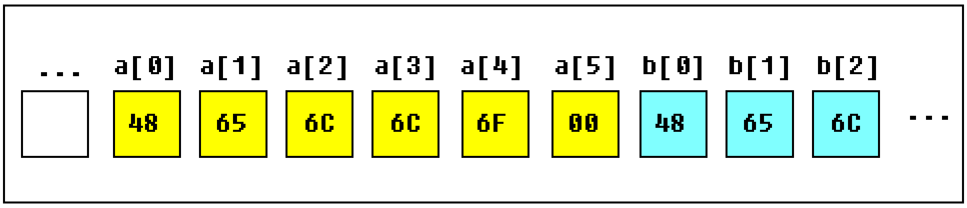


Figure 112:

You can access the value of any element in array using square brackets, for example:
MOV AL, a[3]

You can also use any of the memory index registers BX, SI, DI, BP, for example:
MOV SI, 3
MOV AL, a[SI]

If you need to declare a large array you can use DUP operator.
The syntax for DUP:

number DUP (value(s))
number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:
c DB 5 DUP(9)
is an alternative way of declaring:
c DB 9, 9, 9, 9, 9

one more example:
d DB 5 DUP(1, 2)
is an alternative way of declaring:
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings.

3-T.2 Getting The Address Of A Variable

There is **LEA** (Load Effective Address) instruction and alternative **OFFSET** operator. Both **OFFSET** and **LEA** can be used to get the offset address of the variable.
LEA is more powerful because it also allows you to get the address of an indexed variable. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.
Reminder:
In order to tell the compiler about data type,
these prefixes should be used:

BYTE PTR - for byte.
WORD PTR - for word (two bytes).

For example:
BYTE PTR [BX]   ; byte access.
    or
WORD PTR [BX]   ; word access.
assembler supports shorter prefixes as well:

b. - for **BYTE PTR**
w. - for **WORD PTR**

in certain cases the assembler can calculate the data type automatically.
Here is first example:

Table 25:

```
ORG 100h

MOV    AL, VAR1              ; check value of VAR1 by moving it to AL.

LEA    BX, VAR1              ; get address of VAR1 in BX.

MOV    BYTE PTR [BX], 44h    ; modify the contents of VAR1.

MOV    AL, VAR1              ; check value of VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Here is another example, that uses **OFFSET** instead of **LEA**:

Table 26:

```
ORG 100h

MOV    AL, VAR1              ; check value of VAR1 by moving it to AL.

MOV    BX, OFFSET VAR1       ; get
```

```
address of VAR1 in BX.

MOV     BYTE PTR [BX], 44h     ; modify
the contents of VAR1.

MOV    AL, VAR1             ; check value of
VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

These lines:
LEA BX, VAR1
MOV BX, OFFSET VAR1
are even compiled into the same machine code: MOV BX, num
num is a 16 bit value of the variable offset.
Please note that only these registers can be used inside square brackets (as memory pointers): **BX, SI, DI, BP**!

3-T.3 Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:
name EQU < any expression >

For example:

Table 27:

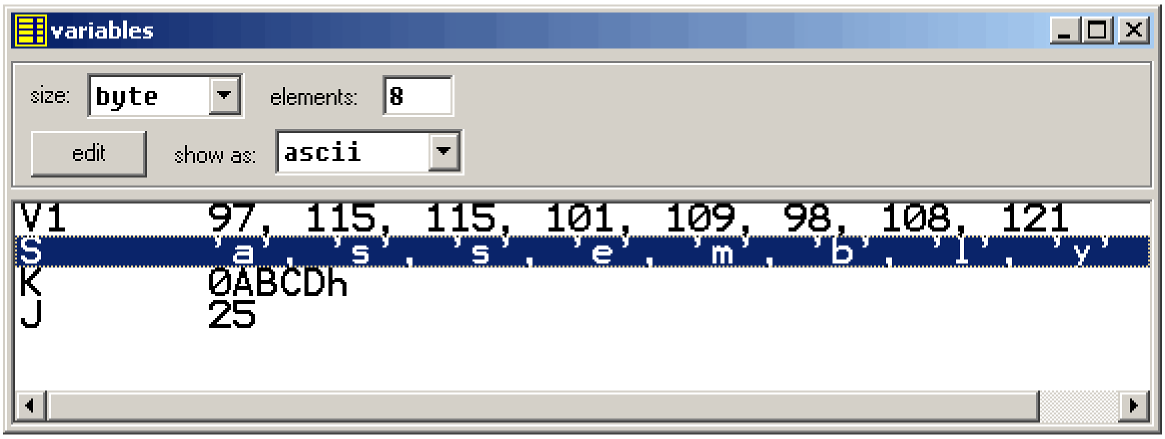
```
k EQU 5

MOV AX, k
```

The above example is functionally identical to code:

Table 28:

```
MOV AX, 5
```

You can view variables while your program executes by selecting "**Variables**" from the "**View**" menu of emulator.



size: byte
elements: 8
show as: ascii

```
V1          97, 115, 115, 101, 109, 98, 108, 121
S           'a', 's', 's', 'e', 'm', 'b', 'l', 'y'
K           0ABCDh
J           25
```

Figure 113:

To view arrays you should click on a variable and set **Elements** property to array size. In assembly language there are not strict data types, so any variable can be presented as an array.

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (there are 256 symbols, some symbols are invisible).

You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.

It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. String can be entered this way:
'hello world', 0
(this string is zero terminated).

Arrays may be entered this way:

1, 2, 3, 4, 5
(the array can be array of bytes or words, it depends whether **BYTE** or **WORD** is selected for edited variable).

Expressions are automatically converted, for example:
when this expression is entered:

5 + 2
it will be converted to **7** etc...

SESSION 4-T: INTERRUPTS

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware. We call such functions **software interrupts**.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:
INT value

Where **value** can be a number between 0 to 255 (or 0 to 0FFh), generally we will use hexadecimal numbers. You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions. To specify a sub-function **AH** register should be set before calling interrupt.
Each interrupt may have up to 256 sub-functions (so we get 256 * 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.
The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This function displays a character on the screen, advancing the cursor and scrolling the screen as necessary.

Table 29:

```
ORG    100h          ; instruct compiler to
make simple single segment .com file.

; The sub-function that we are using
; does not modify the AH register on
; return, so we may set it only once.

MOV    AH, 0Eh    ; select sub-function.

; INT 10h / 0Eh sub-function
; receives an ASCII code of the
; character that will be printed
; in AL register.

MOV    AL, 'H'    ; ASCII code: 72
INT    10h        ; print it!
```

```
MOV    AL, 'e'    ; ASCII code: 101
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'o'    ; ASCII code: 111
INT    10h        ; print it!

MOV    AL, '!'    ; ASCII code: 33
INT    10h        ; print it!

RET               ; returns to operating system.
```

Copy & paste the above program to the source code editor, and press [**Compile and Emulate**] button. Run it!

The list of all interrupts that are currently supported by the emulator.
These interrupts should be compatible with IBM PC and all generations of x86, original Intel 8086 and AMD compatible microprocessors; however Windows XP may overwrite some of the original interrupts.

4-T.1 The Short List Of Supported Interrupts With Descriptions:

INT 10h / **AH = 0** - set video mode.
input:
AL = desired video mode.

these video modes are supported:

00h - text mode. 40x25. 16 colors. 8 pages.

03h - text mode. 80x25. 16 colors. 8 pages.

13h - graphical mode. 40x25. 256 colors. 320x200 pixels. 1 page.
example:

```
mov al, 13h
mov ah, 0
int 10h
```

INT 10h / **AH = 01h** - set text-mode cursor shape.

input:
CH = cursor start line (bits 0-4) and options (bits 5-7).
CL = bottom cursor line (bits 0-4).

when bit 5 of CH is set to **0**, the cursor is visible. when bit 5 is **1**, the cursor is not visible.

```
; hide blinking text cursor:
            mov ch, 32
            mov ah, 1
            int 10h

; show standard blinking text cursor:
            mov ch, 6
            mov cl, 7
            mov ah, 1
            int 10h

; show box-shaped blinking text cursor:
            mov ch, 0
            mov cl, 7
            mov ah, 1
            int 10h

;           note: some bioses required CL to be >=7,
;           otherwise wrong cursor shapes are displayed.
```

INT 10h / **AH = 2** - set cursor position.
input:
DH = row.
DL = column.
BH = page number (0..7).
example:

```
            mov dh, 10
            mov dl, 20
            mov bh, 0
            mov ah, 2
            int 10h
```

INT 10h / **AH = 03h** - get cursor position and size.
input:
BH = page number.
return:
DH = row.
DL = column.
CH = cursor start line.
CL = cursor bottom line.

INT 10h / **AH = 05h** - select active video page.

input:
AL = new page number (0..7).
the activated page is displayed.

INT 10h / **AH = 06h** - scroll up window.
INT 10h / **AH = 07h** - scroll down window.
input:
AL = number of lines by which to scroll (00h = clear entire window).
BH = used to write blank lines at bottom of window.
CH, CL = row, column of window's upper left corner.
DH, DL = row, column of window's lower right corner.

INT 10h / **AH = 08h** - read character and at cursor position.
input:
BH = page number.
return:
AH = .
AL = character.

INT 10h / **AH = 09h** - write character and at cursor position.
input:
AL = character to display.
BH = page number.
BL = .
CX = number of times to write character.

INT 10h / **AH = 0Ah** - write character only at cursor position.
input:
AL = character to display.
BH = page number.
CX = number of times to write character.

INT 10h / **AH = 0Ch** - change color for a single pixel.
input:
AL = pixel color
CX = column.
DX = row.

example:

```
mov al, 13h
mov ah, 0
int 10h     ; set graphics video mode.
mov al, 1100b
mov cx, 10
mov dx, 20
mov ah, 0ch
```

```
int 10h     ; set pixel.
```

INT 10h / AH = 0Dh - get color of a single pixel.
input:
CX = column.
DX = row.
output:
AL = pixel color

INT 10h / AH = 0Eh - teletype output.
input:
AL = character to write.
this functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary. the printing is always done to current active page.

example:

```
mov al, 'a'
mov ah, 0eh
int 10h

; note: on specific systems this
; function may not be supported in graphics mode.
```

INT 10h / AH = 13h - write string.
input:
AL = write mode:
   bit 0: update cursor after writing;
   bit 1: string contains   .
BH = page number.
BL =   if string contains only characters (bit 1 of AL is zero).
CX = number of characters in string (attributes are not counted).
DL,DH = column, row at which to start writing.
ES:BP points to string to be printed.
example:

```
mov al, 1
mov bh, 0
mov bl, 0011_1011b
mov cx, msg1end - offset msg1 ; calculate message size.
mov dl, 10
mov dh, 7
push cs
pop es
mov bp, offset msg1
mov ah, 13h
int 10h
```

```
        jmp msg1end
        msg1 db " hello, world! "
        msg1end:
```

INT 10h / AX = 1003h - toggle intensity/blinking
.
input:
BL = write mode:
  0: enable intensive colors.
  1: enable blinking (not supported by the emulator and windows command prompt).
BH = 0 (to avoid problems on some adapters).
example:

```
mov ax, 1003h
mov bx, 0
int 10h
```

Bit color table:
character attribute is 8 bit value, low 4 bits set fore color, high 4 bits set background color.
Note: the emulator and windows command line prompt do not support background blinking, however to make colors look the same in dos and in full screen mode it is required to turn off the background blinking.

Table 30:

HEX	BIN	COLOR
0	0000	black
1	0001	blue
2	0010	green
3	0011	cyan
4	0100	red
5	0101	magenta
6	0110	brown
7	0111	light gray
8	1000	dark gray
9	1001	light blue
A	1010	light green
B	1011	light cyan
C	1100	light red
D	1101	light magenta
E	1110	yellow
F	1111	white

note:

```
; use this code for compatibility with dos/cmd prompt full screen mode:
mov     ax, 1003h
mov     bx, 0   ; disable blinking.
```

```
int     10h
```

INT 11h - get BIOS equipment list.
return:
AX = BIOS equipment list word, actually this call returns the contents of the word at 0040h:0010h.

Currently this function can be used to determine the number of installed number of floppy disk drives.

Bit fields for BIOS-detected installed hardware:

bit(s)	Description
15-14	Number of parallel devices.
13	Reserved.
12	Game port installed.
11-9	Number of serial devices.
8	Reserved.

7-6	Number of floppy disk drives (minus 1):
	00 single floppy disk;
	01 two floppy disks;
	10 three floppy disks;
	11 four floppy disks.

5-4	Initial video mode: 00 EGA,VGA,PGA, or other with on-board video BIOS; 01 40x25 CGA color. 10 80x25 CGA color (emulator default). 11 80x25 mono text.
3	Reserved.
2	PS/2 mouse is installed.
1	Math coprocessor installed.
0	Set when booted from floppy.

INT 12h - get memory size.
return:
AX = kilobytes of contiguous memory starting at absolute address 00000h, this call returns the contents of the word at 0040h:0013h.

4-T.2 Floppy Drives Are Emulated Using *FLOPPY_0(..3)* Files.

INT 13h / **AH = 00h** - reset disk system.

INT 13h / **AH = 02h** - read disk sectors into memory.
INT 13h / **AH = 03h** - write disk sectors.
input:
AL = number of sectors to read/write (must be nonzero)
CH = cylinder number (0..79).

CL = sector number (1..18).
DH = head number (0..1).
DL = drive number (0..3 , for the emulator it depends on quantity of FLOPPY_ files).
ES:BX points to data buffer.
return:
CF set on error.
CF clear if successful.
AH = status (0 - if successful).
AL = number of sectors transferred.
Note: each sector has **512** bytes.

INT 15h / **AH = 86h** - BIOS wait function.
input:
CX:DX = interval in microseconds
return:
CF clear if successful (wait interval elapsed),
CF set on error or when wait function is already in progress.

Note:
The resolution of the wait period is 977 microseconds on many systems (1 million microseconds - 1 second).
Windows XP does not support this interrupt (always sets CF=1).

INT 16h / **AH = 00h** - get keystroke from keyboard (no echo).
return:
AH = BIOS scan code.
AL = ASCII character.
(if a keystroke is present, it is removed from the keyboard buffer).

INT 16h / **AH = 01h** - check for keystroke in the keyboard buffer.
return:
ZF = 1 if keystroke is not available.
ZF = 0 if keystroke available.
AH = BIOS scan code.
AL = ASCII character.
(if a keystroke is present, it is not removed from the keyboard buffer).

INT 19h - system reboot.
Usually, the BIOS will try to read sector 1, head 0, track 0 from drive **A:** to 0000h:7C00h.
The emulator just stops the execution, to boot from floppy drive select from the menu:
'virtual drive' -> 'boot from floppy'

INT 1Ah / **AH = 00h** - get system time.
return:
CX:DX = number of clock ticks since midnight.
AL = midnight counter, advanced each time midnight passes.

notes:
there are approximately **18.20648** clock ticks per second,
and **1800B0h** per 24 hours.
AL is not set by the emulator.

INT 20h - exit to operating system.

4-T.3 The Short List Of Emulated MS-DOS Interrupts -- INT 21h

---

DOS file system is emulated in **C:\emu8086\vdrive\x** (x is a drive letter)

If no drive letter is specified and current directory is not set, then **C:\emu8086\MyBuild\** path is used by default. **FLOPPY_0,1,2,3** files are emulated independently from DOS file system.

For the emulator physical drive **A:** is this file **c:\emu8086\FLOPPY_0** (for BIOS interrupts: **INT 13h** and boot).

For DOS interrupts (**INT 21h**) drive **A:** is emulated in this subdirectory: **C:\emu8086\vdrive\a\**

Note: DOS file system limits the file and directory names to 8 characters, extension is limited to 3 characters;
example of a valid file name: **myfile.txt** (file name = 6 chars, extension - 3 chars). extension is written after the dot, no other dots are allowed.

INT 21h / **AH=1** - read character from standard input, with echo, result is stored in **AL**.
If there is no character in the keyboard buffer, the function waits until any key is pressed.

example:

```
mov ah, 1
int 21h
```

INT 21h / **AH=2** - write character to standard output.
entry: **DL** = character to write, after execution **AL = DL**.

example:

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h / **AH=5** - output character to printer.
entry: **DL** = character to print, after execution **AL = DL**.

example:

```
mov ah, 5
mov dl, 'a'
int 21h
```

INT 21h / **AH=6** - direct console input or output.

parameters for output: **DL** = 0..254 (ascii code)
parameters for input: **DL** = 255

for output returns: AL = DL
for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
AL = character read; buffer is cleared.

example:

```
mov ah, 6
mov dl, 'a'
int 21h      ; output character.

mov ah, 6
mov dl, 255
int 21h      ; get character from keyboard buffer (if any) or set ZF=1.
```

INT 21h / **AH=7** - character input without echo to AL.
if there is no character in the keyboard buffer, the function waits until any key is pressed.

example:

```
mov ah, 7
int 21h
```

INT 21h / **AH=9** - output of a string at **DS:DX**. String must be terminated by '**$**'.

example:

```
org 100h
mov dx, offset msg
mov ah, 9
int 21h
ret
msg db "hello world $"
```

INT 21h / **AH=0Ah** - input of a string to **DS:DX**, fist byte is buffer size, second byte is number of chars actually read. This function does **not** add '$' in the end of string. To print using **INT 21h** / **AH=9** you must set dollar character at the end of it and start printing from address **DS:DX + 2**.

example:

```
org 100h
mov dx, offset buffer
mov ah, 0ah
int 21h
jmp print
buffer db 10,?, 10 dup(' ')
print:
xor bx, bx
mov bl, buffer[1]
mov buffer[bx+2], '$'
mov dx, offset buffer + 2
mov ah, 9
int 21h
ret
```

The function does not allow to enter more characters than the specified buffer size.
See also **int21.asm** in c:\emu8086\examples

INT 21h / **AH=0Bh** - get input status;
returns: **AL = 00h** if no character available, **AL = 0FFh** if character is available.

INT 21h / **AH=0Ch** - flush keyboard buffer and read standard input.
entry: **AL** = number of input function to execute after flushing buffer (can be 01h,06h,07h,08h, or 0Ah - for other values the buffer is flushed but no input is attempted); other registers as appropriate for the selected input function.

INT 21h / **AH= 0Eh** - select default drive.

Entry: **DL** = new default drive (0=A:, 1=B:, etc)

Return: **AL** = number of potentially valid drive letters

Notes: the return value is the highest drive present.

INT 21h / **AH= 19h** - get current default drive.

Return: AL = drive (0=A:, 1=B:, etc)

INT 21h / **AH=25h** - set interrupt vector;
input: **AL** = interrupt number. **DS:DX** -> new interrupt handler.

INT 21h / **AH=2Ah** - get system date;
return: **CX** = year (1980-2099). **DH** = month. **DL** = day. **AL** = day of week (00h=Sunday)

INT 21h / **AH=2Ch** - get system time;
return: **CH** = hour. **CL** = minute. **DH** = second. **DL** = 1/100 seconds.

INT 21h / **AH=35h** - get interrupt vector;
entry: **AL** = interrupt number;
return: **ES:BX** -> current interrupt handler.

INT 21h / **AH= 39h** - make directory.
entry: **DS:DX** -> ASCIZ pathname; zero terminated string, for

example:

```
org 100h
mov dx, offset filepath
mov ah, 39h
int 21h

ret

filepath DB "C:\mydir", 0     ; path to be created.
end
```

the above code creates **c:\emu8086\vdrive\C\mydir** directory if run by the emulator.

Return: **CF** clear if successful **AX** destroyed. **CF** set on error **AX** = error code.
Note: all directories in the given path must exist except the last one.

INT 21h / **AH= 3Ah** - remove directory.

Entry: **DS:DX** -> ASCIZ pathname of directory to be removed.

Return:

CF is clear if successful, **AX** destroyed **CF** is set on error **AX** = error code.

Notes: directory must be empty (there should be no files inside of it).

INT 21h / **AH= 3Bh** - set current directory.

Entry: **DS:DX** -> ASCIZ pathname to become current directory (max 64 bytes).

Return:

Carry Flag is clear if successful, **AX** destroyed.
Carry Flag is set on error **AX** = error code.
Notes: even if new directory name includes a drive letter, the default drive is not changed, only the current directory on that drive.

INT 21h / **AH= 3Ch** - create or truncate file.

entry:

CX = file attributes:

```
mov cx, 0      ;  normal - no attributes.
mov cx, 1      ;  read-only.
mov cx, 2      ;  hidden.
mov cx, 4      ;  system
mov cx, 7      ;  hidden, system and read-only!
mov cx, 16      ;  archive
```

DS:DX -> ASCIZ filename.

returns:

CF clear if successful, **AX** = file handle.
CF set on error **AX** = error code.

Note: if specified file exists it is deleted without a warning.

example:

```
        org 100h
        mov ah, 3ch
        mov cx, 0
        mov dx, offset filename
        mov ah, 3ch
        int 21h
        jc err
        mov handle, ax
        jmp k
        filename db "myfile.txt", 0
        handle dw ?
        err:
        ; ....
        k:
        ret
```

INT 21h / **AH= 3Dh** - open existing file.
Entry:

AL = access and sharing modes:

```
mov al, 0   ; read
mov al, 1   ; write
mov al, 2   ; read/write
```

DS:DX -> ASCIZ filename.

Return:

CF clear if successful, **AX** = file handle.

CF set on error **AX** = error code.

note 1: file pointer is set to start of file.
note 2: file must exist.

example:

```
org 100h
mov al, 2
mov dx, offset filename
mov ah, 3dh
int 21h
jc err
mov handle, ax
jmp k
filename db "myfile.txt", 0
handle dw ?
err:
; ....
k:
ret
```

INT 21h / **AH= 3Eh** - close file.

Entry: **BX** = file handle

Return:

CF clear if successful, **AX** destroyed.
CF set on error, **AX** = error code (06h).

INT 21h / **AH= 3Fh** - read from file.

Entry:

BX = file handle.
CX = number of bytes to read.
DS:DX -> buffer for data.

Return:

CF is clear if successful - **AX** = number of bytes actually read; 0 if at EOF (end of file) before call.
CF is set on error **AX** = error code.

Note: data is read beginning at current file position, and the file position is updated after a successful read the returned **AX** may be smaller than the request in **CX** if a partial read occurred.

INT 21h / **AH= 40h** - write to file.
entry:
BX = file handle.
CX = number of bytes to write.
DS:DX -> data to write.
return:
CF clear if successful; **AX** = number of bytes actually written.
CF set on error; **AX** = error code.
Note: if **CX** is zero, no data is written, and the file is truncated or extended to the current position data is written beginning at the current file position, and the file position is updated after a successful write the usual cause for **AX** < **CX** on return is a full disk.

INT 21h / **AH= 41h** - delete file (unlink).
Entry:
DS:DX -> ASCIZ filename (no wildcards, but see notes).
return:
CF clear if successful, **AX** destroyed. **AL** is the drive of deleted file (undocumented).
CF set on error **AX** = error code.

Note: DOS does not erase the file's data; it merely becomes inaccessible because the FAT chain for the file is cleared deleting a file which is currently open may lead to filesystem corruption.

INT 21h / **AH= 42h** - SEEK - set current file position.
Entry:
AL = origin of move: **0** - start of file. **1** - current file position. **2** - end of file.
BX = file handle.
CX:DX = offset from origin of new file position.
Return:
CF clear if successful, **DX:AX** = new file position in bytes from start of file.
CF set on error, AX = error code.
Notes:

For origins **1** and **2**, the pointer may be positioned before the start of the file; no error is returned in that case, but subsequent attempts to read or write the file will produce errors. If the new position is beyond the current end of file, the file will be extended by the next write (see =40h)
example:

```
org 100h
mov ah, 3ch
mov cx, 0
mov dx, offset filename
mov ah, 3ch
int 21h  ; create file...
mov handle, ax

mov bx, handle
mov dx, offset data
mov cx, data_size
mov ah, 40h
int 21h ; write to file...

mov al, 0
mov bx, handle
mov cx, 0
mov dx, 7
mov ah, 42h
int 21h ; seek...

mov bx, handle
mov dx, offset buffer
mov cx, 4
mov ah, 3fh
int 21h ; read from file...

mov bx, handle
mov ah, 3eh
int 21h ; close file...
ret

filename db "myfile.txt", 0
handle dw ?
data db " hello files! "
data_size=$-offset data
buffer db 4 dup(' ')
```

INT 21h / **AH= 47h** - get current directory.

Entry:

DL = drive number (00h = default, 01h = A:, etc)
DS:SI -> 64-byte buffer for ASCIZ pathname.

Return:

Carry is clear if successful
Carry is set on error, **AX** = error code (0Fh)

Notes:

the returned path does not include a drive and the initial backslash.

INT 21h / **AH=4Ch** - return control to the operating system (stop program).

INT 21h / **AH= 56h** - rename file / move file.

Entry:

DS:DX -> ASCIZ filename of existing file.

ES:DI -> ASCIZ new filename.

Return:

CF clear if successful.
CF set on error, **AX** = error code.

Note: allows move between directories on same logical drive only; open files should not be renamed!

4-T.4 Mouse Driver Interrupts -- INT 33h

INT 33h / **AX=0000** - mouse ininialization. any previous mouse pointer is hidden.

returns:
if successful: **AX**=0FFFFh and **BX**=number of mouse buttons.
if failed: **AX**=0
example:

```
mov ax, 0
int 33h
```

see also: mouse.asm in examples.

INT 33h / **AX=0001** - show mouse pointer.

example:

```
mov ax, 1
int 33h
```

INT 33h / **AX=0002** - hide visible mouse pointer.

example:

```
mov ax, 2
int 33h
```

INT 33h / **AX=0003** - get mouse position and status of its buttons.

returns:
if left button is down: **BX**=1
if right button is down: **BX**=2
if both buttons are down: **BX**=3
CX = x
DX = y
example:

```
mov ax, 3
int 33h
```

```
; note: in graphical 320x200 mode the value of CX is doubled.
; see mouse2.asm in examples.
```

SESSION 5-T: LIBRARY OF COMMON FUNCTIONS - emu8086.inc

To make programming easier there are some common functions that can be included in your program. To make your program use functions defined in other file you should use the **INCLUDE** directive followed by a file name. Compiler automatically searches for the file in the same folder where the source file is located, and if it cannot find the file there - it searches in **Inc** folder.

Currently you may not be able to fully understand the contents of the **emu8086.inc** (located in **Inc** folder), but it's OK, since you only need to understand what it can do.

To use any of the functions in **emu8086.inc** you should have the following line in the beginning of your source file:

include 'emu8086.inc'

---

emu8086.inc defines the following **macros**:

- **PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
- **GOTOXY col, row** - macro with 2 parameters, sets cursor position.
- **PRINT string** - macro with 1 parameter, prints out a string.
- **PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

To use any of the above macros simply type its name somewhere in your code, and if required parameters, for example:

Table 31:

```
include emu8086.inc

ORG    100h

PRINT 'Hello World!'

GOTOXY 10, 5

PUTC 65          ; 65 - is an ASCII code for 'A'
PUTC 'B'
```

```
RET             ; return to operating system.
END              ; directive to stop the
compiler.
```

When compiler process your source code it searches the **emu8086.inc** file for declarations of the macros and replaces the macro names with real code. Generally macros are relatively small parts of code, frequent use of a macro may make your executable too big (procedures are better for size optimization).

---

emu8086.inc also defines the following **procedures**:

- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.
- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:

  ```
  CALL PTHIS
  db 'Hello World!', 0
  ```

  To use it declare: **DEFINE_PTHIS** before **END** directive.
- **GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare: **DEFINE_GET_STRING** before **END** directive.
- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.
- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.
- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before the **END** directive), and then use **CALL** instruction followed by a procedure name.
For example:

Table 32:

```
include 'emu8086.inc'
ORG    100h
LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.

MOV    AX, CX         ; copy the number to
AX.

; print the following string:
CALL   pthis
DB  13, 10, 'You have entered: ', 0

CALL   print_num      ; print number in AX.

RET                   ; return to operating system.

msg1   DB  'Enter the number: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required
for print_num.
DEFINE_PTHIS

END                   ; directive to stop the
compiler.
```

First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred to procedure. This is quite useful, since even if you call the same procedure 100 times in your code you will still have relatively small executable size. Seems complicated, isn't it? That's ok, with the time you will learn more, currently it's required that you understand the basic principle.

SESSION 6-T: ARITHMETIC AND LOGIC INSTRUCTIONS

Most Arithmetic and Logic Instructions affect the processor status register (or **Flags**)

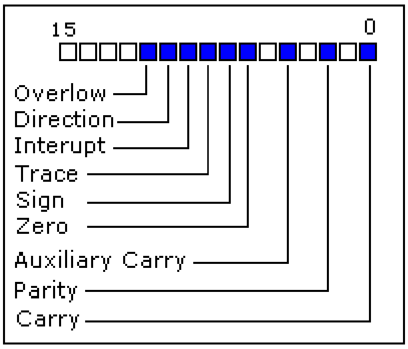


Figure 114:

As you may see there are 16 bits in this register, each bit is called a **flag** and can take a value of **1** or **0**.

- **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow this flag is set to **0**.
- **Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
- **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually this flag take the value of the most significant bit.
- **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).
- **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bits in result, and to **0** when there is odd number of one bits. Even if result is a word only 8 low bits are analyzed!
- **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
- **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.
- **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.

There are 3 groups of instructions.

6-T.1 First Group: ADD, SUB, CMP, AND, TEST, OR, XOR

These types of operands are supported:
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...
After operation between operands, result is always stored in first operand. **CMP** and **TEST** instructions affect flags only and do not store a result (these instructions are used to make decisions during program execution).

These instructions affect these flags only:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **ADD** - add second operand to first.
- **SUB** - Subtract second operand to first.
- **CMP** - Subtract second operand from first **for flags only**.
- **AND** - Logical AND between all bits of two operands. These rules apply:
  1 AND 1 = 1
  1 AND 0 = 0
  0 AND 1 = 0
  0 AND 0 = 0
  As you see we get **1** only when both bits are **1**.
- **TEST** - The same as **AND** but **for flags only**.
- **OR** - Logical OR between all bits of two operands. These rules apply:
  1 OR 1 = 1
  1 OR 0 = 1
  0 OR 1 = 1
  0 OR 0 = 0
  As you see we get **1** every time when at least one of the bits is **1**.
- **XOR** - Logical XOR (exclusive OR) between all bits of two operands. These rules apply:
  1 XOR 1 = 0
  1 XOR 0 = 1
  0 XOR 1 = 1
  0 XOR 0 = 0
  As you see we get **1** every time when bits are different from each other.

6-T.2 Second Group: MUL, IMUL, DIV, IDIV

These types of operands are supported:
REG
memory

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

MUL and **IMUL** instructions affect these flags only:
CF, **OF**
When result is over operand size these flags are set to **1**, when result fits in operand size these flags are set to **0**.

For **DIV** and **IDIV** flags are undefined.

- **MUL** - Unsigned multiply:
  when operand is a **byte**:
  AX = AL * operand.
  when operand is a **word**:
  (DX AX) = AX * operand.
- **IMUL** - Signed multiply:
  when operand is a **byte**:
  AX = AL * operand.
  when operand is a **word**:
  (DX AX) = AX * operand.
- **DIV** - Unsigned divide:
  when operand is a **byte**:
  AL = AX / operand
  AH = remainder (modulus). .
  when operand is a **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .
- **IDIV** - Signed divide:
  when operand is a **byte**:
  AL = AX / operand
  AH = remainder (modulus). .
  when operand is a **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .

6-T.3 Third Group: INC, DEC, NOT, NEG

These types of operands are supported:
REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

INC, **DEC** instructions affect these flags only:
ZF, **SF**, **OF**, **PF**, **AF**.

NOT instruction does not affect any flags!

NEG instruction affects these flags only:

CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** - Reverse each bit of operand.
- **NEG** - Make operand negative (two's complement). Actually it reverses each bit of operand and then adds 1 to it. For example 5 will become -5, and -2 will become 2.

SESSION 7-T: PROGRAM FLOW CONTROL

Controlling the program flow is a very important thing; this is where your program can make decisions according to certain conditions.

7-T.1 Unconditional Jumps

The basic instruction that transfers control to another point in the program is **JMP**.
The basic syntax of **JMP** instruction:

```
JMP label
```

To declare a *label* in your program, just type its name and add "**:**" to the end, label can be any character combination but it cannot start with a number, for example here are 3 legal label definitions:

```
label1:
label2:
a:
```

Label can be declared on a separate line or before any other instruction, for example:

```
x1:
MOV AX, 1

x2: MOV AX, 2
```

here's an example of **JMP** instruction:

```
org    100h
mov    ax, 5          ; set ax to 5.
mov    bx, 2          ; set bx to 2.

jmp    calc           ; go to 'calc'.

back:  jmp stop       ; go to 'stop'.

calc:
add    ax, bx         ; add bx to ax.
jmp    back           ; go 'back'.

stop:
ret                   ; return to operating system.
```

Of course there is an easier way to calculate the some of two numbers, but it's still a good example of **JMP** instruction.
As you can see from this example **JMP** is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

7-T.2 Short Conditional Jumps

Unlike **JMP** instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

7-T.2.1 Jump instructions that test single flag

Table 33:

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

As you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**, **JC** is assembled the same as **JB** etc...

Different names are used to make programs easier to understand, to code and most importantly to remember. very offset dissembler has no clue what the original instruction was look like that's why it uses the most common name.

If you emulate this code you will see that all instructions are assembled into **JNB**, the operational code (opcode) for this instruction is **73h** this instruction has fixed length of two bytes, the second byte is number of bytes to add to the **IP** register if the condition is true. because the instruction has only 1 byte to keep the offset it is limited to pass control to -128 bytes back or 127 bytes forward, this value is always signed.

- jnc a
- jnb a
- jae a
-
- mov ax, 4
- a: mov ax, 5
- ret

7-T.2.2 Jump instructions for signed numbers

Table 34:

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ

JG , JNLE	Jump if Greater (>).
	Jump if Not Less or Equal (not <=).
		ZF = 0
		and
		SF = OF
			JNG, JLE

JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

- <> - sign means not equal.

7-T.2.3 Jump instructions for unsigned numbers

Table 35:

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ

JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ

JA , JNBE	Jump if Above (>).
	Jump if Not Below or Equal (not <=).
		CF = 0
		and
		ZF = 0
			JNA, JBE

JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB

JBE , JNA	Jump if Below or Equal (<=).
	Jump if Not Above (not >).
		CF = 1
		or
		ZF = 1
			JNBE, JA

7-T.3 Compare

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).

- The logic is very simple, for example:
  it's required to compare 5 and 2,
  5 - 2 = 3
  the result is not zero (Zero Flag is set to 0).
  Another example:
  it's required to compare 7 and 7,
  7 - 7 = 0
  the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

- Here's an example of **CMP** instruction and conditional jump:

```
include "emu8086.inc"

org    100h

mov    al, 25     ; set al to 25.
mov    bl, 10     ; set bl to 10.

cmp    al, bl     ; compare al - bl.

je     equal      ; jump if al = bl (zf = 1).

```

- putc 'n' ; if it gets here, then al <> bl,
- jmp stop ; so print 'n', and jump to stop.
-
- equal: ; if gets here,
- putc 'y' ; then al = bl, so print 'y'.
-
- stop:
-
- ret ; gets here no matter what.

Try the above example with different numbers for **AL** and **BL**, open flags by clicking on flags button, use single step and see what happens. you can use **F5** hotkey to recompile and reload the program into the emulator.

7-T.4 Loops

Table 36:

instruction	operation and jump condition	Opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ
LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ
JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

Loops are basically the same jumps, it is possible to code loops without using the loop instruction, by just using conditional jumps and compare, and this is just what loop does. All loop instructions use **CX** register to count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF, however with some agility it is possible to put one loop into another, and another into another two, and three and etc... and receive a nice value of 65535 * 65535 * 65535 ....till infinity.... or the end of ram or stack memory. it is possible store original value of cx register using **push cx** instruction and return it to original when the internal loop ends with **pop cx**,

for example:

- org 100h

```

mov bx, 0  ; total step counter.
mov cx, 5
k1: add bx, 1
    mov al, '1'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5
      k2: add bx, 1
      mov al, '2'
      mov ah, 0eh
      int 10h
      push cx
         mov cx, 5
         k3: add bx, 1
         mov al, '3'
         mov ah, 0eh
         int 10h
         loop k3    ; internal in internal loop.
      pop  cx
      loop  k2      ; internal loop.
    pop cx
loop k1             ; external loop.

ret

```

bx counts total number of steps, by default emulator shows values in hexadecimal, you can double click the register to see the value in all available bases.

Just like all other conditional jumps loops have an opposite companion that can help to create workarounds, when the address of desired location is too far assemble automatically assembles reverse and long jump instruction, making total of 5 bytes instead of just 2, it can be seen in disassembler as well.

For more detailed description and examples refer to **8086 instruction set**

- All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes).
- We can easily avoid this limitation using a cute trick:
  - Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
  - Use **JMP** instruction to jump to desired location.
  - Define *label_x:* just after the **JMP** instruction.

  label_x: - can be any valid label name, but there must not be two or more labels with the same name.

```
here's an example:
include "emu8086.inc"
org    100h

mov    al, 5
mov    bl, 5

cmp    al, bl     ; compare al - bl.

; je equal        ; there is only 1 byte

jne    not_equal  ; jump if al <> bl (zf = 0).
jmp    equal
not_equal:

add    bl, al
sub    al, 10
xor    al, bl

jmp skip_data
db 256 dup(0)     ; 256 bytes
skip_data:

putc   'n'        ; if it gets here, then al <> bl,
jmp    stop       ; so print 'n', and jump to stop.

equal:            ; if gets here,
putc   'y'        ; then al = bl, so print 'y'.

stop:
ret
```

Note: the latest version of the integrated 8086 assembler automatically creates a workaround by replacing the conditional jump with the opposite, and adding big unconditional jump. To check if you have the latest version of emu8086 click **help-> check for an update** from the menu.

Another, yet rarely used method is providing an immediate value instead of label. When immediate value starts with $ relative jump is performed, otherwise compiler calculates instruction that jumps directly to given offset. For example:

Table 37:

```
org    100h

; unconditional jump forward:
; skip over next 3 bytes + itself
; the machine code of short jmp instruction
takes 2 bytes.
jmp $3+2
a db 3    ; 1 byte.
b db 4    ; 1 byte.
c db 4    ; 1 byte.

; conditional jump back 5 bytes:
mov bl,9
dec bl      ; 2 bytes.
cmp bl, 0   ; 3 bytes.
jne $-5     ; jump 5 bytes back

ret
```

SESSION 8-T: PROCEDURES

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC

      ; here goes the code
      ; of the procedure ...

RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Here is an example:

Table 38:

```
ORG    100h

CALL   m1

MOV    AX, 2

RET                   ; return to operating system.

m1     PROC
MOV    BX, 5
RET                   ; return to caller.
m1     ENDP

END
```

The above example calls procedure **m1**, does **MOV BX, 5**, and returns to the next instruction after **CALL**: **MOV AX, 2**.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in **AL** and **BL** registers, multiplies these parameters and returns the result in **AX** register:

Table 39:

```
ORG    100h

MOV    AL, 1
MOV    BL, 2

CALL   m2
CALL   m2
CALL   m2
CALL   m2

RET                   ; return to operating system.

m2     PROC
MUL    BL                ; AX = AL * BL.
RET                   ; return to caller.
m2     ENDP

END
```

In the above example value of **AL** register is update every time the procedure is called, **BL** register stays unchanged, so this algorithm calculates **2** in power of **4**, so final result in **AX** register is **16** (or 10h).

Here goes another example,
that uses a procedure to print a *Hello World!* message:

Table 40:

```
ORG    100h

LEA    SI, msg        ; load address of msg to SI.

CALL   print_me

RET                   ; return to operating system.

;
==========================================================
; this procedure prints a string, the string should be null
; terminated (have zero in the end),
; the string address should be in SI register:
print_me     PROC

next_char:
    CMP  b.[SI], 0    ; check for zero to stop
    JE   stop         ;

    MOV  AL, [SI]     ; next get ASCII char.

    MOV  AH, 0Eh      ; teletype function number.
    INT  10h          ; using interrupt to print a char in AL.

    ADD  SI, 1        ; advance index of string array.

    JMP  next_char    ; go back, and type another char.

stop:
RET                   ; return to caller.
print_me     ENDP
;
==========================================================

msg    DB  'Hello World!', 0   ; null terminated string.

END
```

"**b.**" - prefix before [SI] means that we need to compare bytes, not words. When you need to compare words add "**w.**" prefix instead. When one of the compared operands is a register it's not required because compiler knows the size of each register.

SESSION 9-T: THE STACK

Stack is an area of memory for keeping temporary data. Stack is used by **CALL** instruction to keep return address for procedure, **RET** instruction gets this value from the stack and returns to that offset. Quite the same thing happens when **INT** instruction calls an interrupt, it stores in stack flag register, code segment and offset. **IRET** instruction is used to return from interrupt call.

We can also use the stack to keep any other data,
there are two instructions that work with the stack:

PUSH - stores 16 bit value in the stack.

POP - gets 16 bit value from the stack.

Table 41:

Syntax for **PUSH** instruction: PUSH REG PUSH SREG PUSH memory PUSH immediate **REG**: AX, BX, CX, DX, DI, SI, BP, SP. **SREG**: DS, ES, SS, CS. **memory**: [BX], [BX+SI+7], 16 bit variable, etc... **immediate**: 5, -24, 3Fh, 10001101b, etc...
Syntax for **POP** instruction: POP REG POP SREG POP memory **REG**: AX, BX, CX, DX, DI, SI, BP, SP. **SREG**: DS, ES, SS, (except CS). **memory**: [BX], [BX+SI+7], 16 bit variable, etc...

Notes:

- **PUSH** and **POP** work with 16 bit values only!
- Note: **PUSH immediate** works only on 80186 CPU and later!

The stack uses **LIFO** (Last In First Out) algorithm,
this means that if we push these values one by one into the stack:
1, 2, 3, 4, 5
the first value that we will get on pop will be **5**, then **4**, **3**, **2**, and only then **1**.

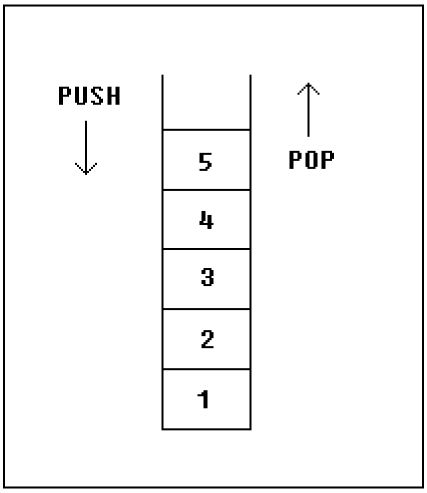


Figure 115

It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).

PUSH and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using **PUSH**).
- Use the register for any purpose.
- Restore the original value of the register from stack (using **POP**).

Here is an example:

Table 42

```
ORG    100h

MOV    AX, 1234h
PUSH   AX          ; store value of AX in
stack.
```

```
MOV    AX, 5678h   ; modify the AX value.

POP    AX          ; restore the original value
of AX.

RET

END
```

Another use of the stack is for exchanging the values,
here is an example:

Table 43:

```
ORG    100h

MOV    AX, 1212h   ; store 1212h in AX.
MOV    BX, 3434h   ; store 3434h in BX


PUSH   AX          ; store value of AX in
stack.
PUSH   BX          ; store value of BX in
stack.

POP    AX          ; set AX to original value
of BX.
POP    BX          ; set BX to original value of
AX.

RET

END
```

The exchange happens because stack uses **LIFO** (Last In First Out) algorithm, so when we push **1212h** and then **3434h**, on pop we will first get **3434h** and only after it **1212h**.

The stack memory area is set by **SS** (Stack Segment) register, and **SP** (Stack Pointer) register. Generally operating system sets values of these registers on program start.

"**PUSH** ***source***" instruction does the following:

- Subtract **2** from **SP** register.
- Write the value of ***source*** to the address **SS:SP**.

"**POP *destination***" instruction does the following:

- Write the value at the address **SS:SP** to ***destination***.
- Add **2** to **SP** register.

The current address pointed by **SS:SP** is called **the top of the stack**.

For **COM** files stack segment is generally the code segment, and stack pointer is set to value of **0FFFEh**. At the address **SS:0FFFEh** stored a return address for **RET** instruction that is executed in the end of the program.

You can visually see the stack operation by clicking on [**Stack**] button on emulator window. The top of the stack is marked with "<" sign.

SESSION 10-T: MACROS

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it. **.inc** is a good example of how macros can be used, this file contains several macros to make coding easier for you.

Table 44:

```
Macro definition:

name    MACRO  [parameters,...]

             <instructions>

ENDM
```

Unlike procedures, macros should be defined above the code that uses it, for example:

Table 45:

```
MyMacro    MACRO  p1, p2, p3

    MOV AX, p1
    MOV BX, p2
    MOV CX, p3

ENDM
```

```
ORG 100h
MyMacro 1, 2, 3
MyMacro 4, 5, DX
RET
```

The above code is expanded into:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Table 46:

Some important facts about **macros** and **procedures**:

- When you want to use a procedure you should use **CALL** instruction, for example:
  CALL MyProc
- When you want to use a macro, you can just type its name. For example:
  MyMacro
- Procedure is located at some specific address in memory, and if you use the same procedure 100 times, the CPU will transfer control to this part of the memory. The control will be returned back to the program by **RET** instruction. The **stack** is used to keep the return address. The **CALL** instruction takes about 3 bytes, so the size of the output executable file grows very insignificantly, no matter how many time the procedure is used.
- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.
- You should use **stack** or any general purpose registers to pass parameters to procedure.
- To pass parameters to macro, you can just type them after the macro name. For example:
  MyMacro 1, 2, 3
- To mark the end of the macro **ENDM** directive is enough.
- To mark the end of the procedure, you should type the name of the procedure before the **ENDP** directive.

Macros are expanded directly in code, therefore if there are labels inside the macro definition you may get "Duplicate declaration" error when macro is used for twice or more. To avoid such problem, use **LOCAL** directive followed by names of variables, labels or procedure names. For example:

Table 47:

```
MyMacro2    MACRO
            LOCAL label1, label2

            CMP  AX, 2
            JE label1
            CMP  AX, 3
            JE label2
            label1:
                         INC  AX
            label2:
                         ADD  AX, 2
ENDM


ORG 100h

MyMacro2

MyMacro2

RET
```

If you plan to use your macros in several programs, it may be a good idea to place all macros in a separate file. Place that file in **Inc** folder and use **INCLUDE *file-name*** directive to use macros. See **of common functions - emu8086.inc** for an example of such file.

SESSION 11-T: CONTROLLING EXTERNAL DEVICES

There are 7 devices attached to the emulator: traffic lights, stepper-motor, LED display, thermometer, printer, robot and simple test device. You can view devices when you click "**Virtual Devices**" menu of the emulator.

For technical information see **/O ports** section of emu8086 reference.

In general, it is possible to use any x86 family CPU to control all kind of devices, the difference maybe in base I/O port number, this can be altered using some tricky electronic equipment. Usually the "**.bin**" file is written into the Read Only Memory (ROM) chip, the system reads program from that chip, loads it in RAM module and runs the program. This principle is used for many modern devices such as micro-wave ovens and etc...

11-T.1 Traffic Lights

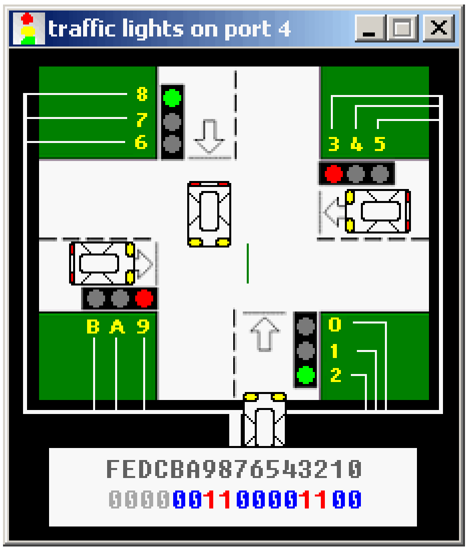


Figure 116

Usually to control the traffic lights an array (table) of values is used. In certain periods of time the value is read from the array and sent to a port. For example:

```
; controlling external device with 8086 microprocessor.
; realistic test for c:\emu8086\devices\Traffic_Lights.exe

#start=Traffic_Lights.exe#

name "traffic"


mov ax, all_red
out 4, ax


mov si, offset situation


next:
mov ax, [si]
out 4, ax

; wait 5 seconds (5 million microseconds)
mov     cx, 4Ch    ;    004C4B40h = 5,000,000
mov     dx, 4B40h
mov     ah, 86h
int     15h
```

```
add si, 2 ; next situation
cmp si, sit_end
jb  next
mov si, offset situation
jmp next

;                    FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $

all_red         equ     0000_0010_0100_1001b
```

11-T.2 Stepper-Motor

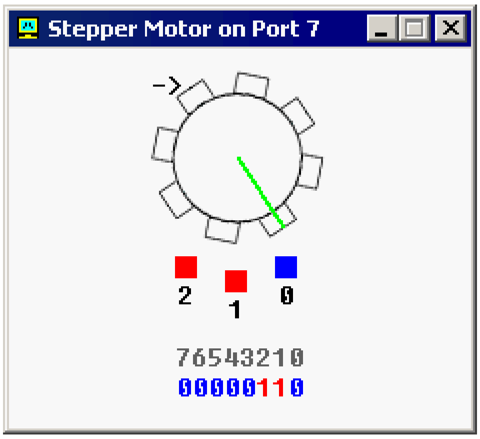


Figure 117:

The motor can be half stepped by turning on pair of magnets, followed by a single and so on.

The motor can be full stepped by turning on pair of magnets, followed by another pair of magnets and in the end followed by a single magnet and so on. The best way to make full step is to make two half steps.

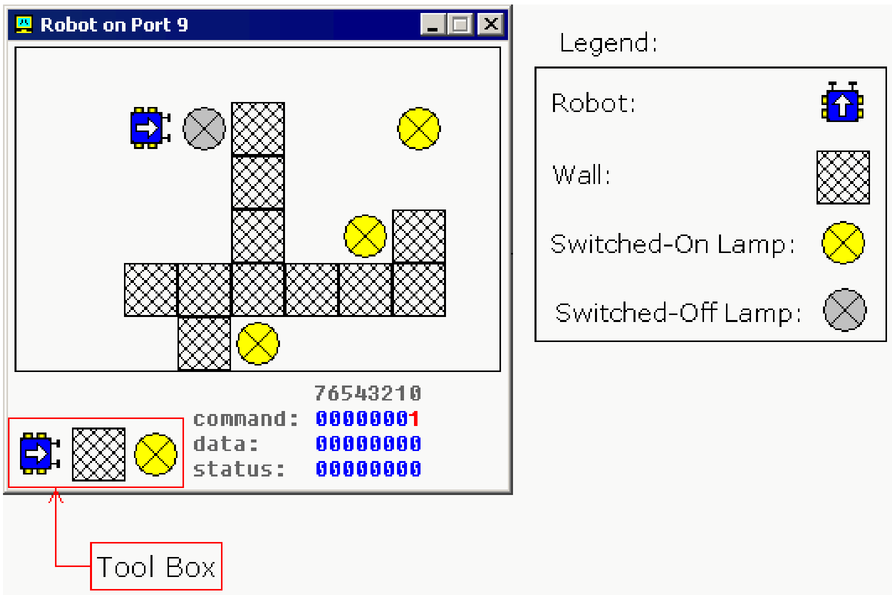
Half step is equal to **11.25** degrees.
Full step is equal to **22.5** degrees.

The motor can be turned both clock-wise and counter-clock-wise.

open **stepper_motor.asm** from c:\emu8086\examples

See also **/O ports** section of emu8086 reference.

11-T.3 Robot



```
          76543210
command:  00000001
data:     00000000
status:   00000000
```

Figure 118:

Complete list of robot instruction set is given in /O section of emu8086 reference.

To control the robot a complex algorithm should be used to achieve maximum efficiency. The simplest, yet very inefficient, is random moving algorithm, open robot.asm from c:\emu8086\examples

It is also possible to use a data table (just like for Traffic Lights), this can be good if robot always works in the same surroundings.