

Unit II: MACHINE INSTRUCTIONS and PROGRAMS

This unit deals with concepts as to how programs are executed in the computer from the machine instruction set viewpoint. All computers deal with numbers. They have instructions that perform operations on the data operands. Hence we start with basic Number systems.

NUMBER, ARITHMETIC OPERATIONS and CHARACTERS

Binary numbers (0, 1) are used in computers. Various number systems are used in computers. Numbers in binary are represented as vectors

$$B = b_{n-1} \dots b_1 b_0$$

Unsigned numbers are in range 0 to $2^n - 1$ and are represented by

$$V(B) = b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$



Negative Numbers: They can be represented in various ways given below.

- Sign-and-magnitude

Most significant bit determines sign, remaining unsigned bits represent magnitude

- 1's complement

Most significant bit determines sign. To change sign from unsigned to negative, invert all the bits (-3 is obtained by complementing each bit in vector 0011 to yield 1100).

- 2's complement

Most significant bit determines sign. To change sign from unsigned to negative, invert all the bits and add 1. This is equivalent to subtracting the positive number from 2^n .

The representations are as given in the table below

B				Values represented		
$b_3 b_2 b_1 b_0$				Sign and magnitude	1's complement	2's complement
0	1	1	1	+ 7	+ 7	+ 7
0	1	1	0	+ 6	+ 6	+ 6
0	1	0	1	+ 5	+ 5	+ 5
0	1	0	0	+ 4	+ 4	+ 4
0	0	1	1	+ 3	+ 3	+ 3
0	0	1	0	+ 2	+ 2	+ 2
0	0	0	1	+ 1	+ 1	+ 1
0	0	0	0	+ 0	+ 0	+ 0
1	0	0	0	- 0	- 7	- 8
1	0	0	1	- 1	- 6	- 7
1	0	1	0	- 2	- 5	- 6
1	0	1	1	- 3	- 4	- 5
1	1	0	0	- 4	- 3	- 4
1	1	0	1	- 5	- 2	- 3
1	1	1	0	- 6	- 1	- 2
1	1	1	1	- 7	- 0	- 1

Figure 1: Binary, signed integer representation

Addition & Subtraction of Signed Numbers: 3 systems of representing signed numbers

These systems differ only in the way they represent negative number

Sign and magnitude system – simplest representation – most awkward for addition and subtraction. 1's complement method is somewhat better. 2's complement is the most efficient method

Circle representation of Integer Mod N: This is a graphical technique to compute $(a+b) \bmod 16$. This can be also used for addition involving signed numbers. Both the cases are shown below

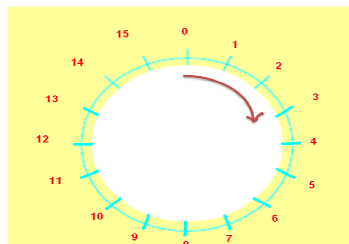


Figure 2: Circle representation of integer Mod 16

The operation $(7+4) \bmod 16$ yields the value 11. To perform this graphically using the above representation locate 7 on the circle and then move 4 units in the clock wise direction to arrive at the answer 11.

Next let us consider adding +7 to -3. The representation is as shown below

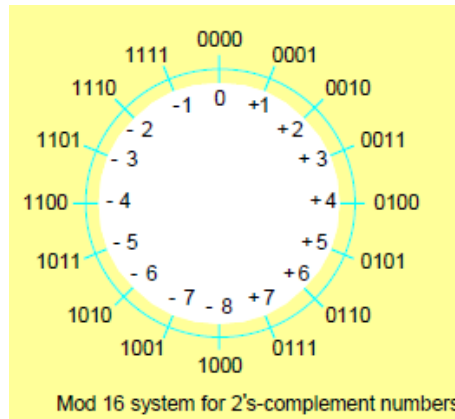


Figure 3: Mod 16 system for 2's complement numbers

2's complement representation for 7 is 0111 and -3 is 1101. Thus locate 0111 and then move 1101(13 steps) in clockwise direction to arrive at 0100 = +4.

Some more examples of 2's complement add and subtract operations are as shown in figure below:

(a)	$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$	$\begin{pmatrix} +2 \\ +3 \end{pmatrix}$	(b)	$\begin{array}{r} 0100 \\ + 1010 \\ \hline 1110 \end{array}$	$\begin{pmatrix} +4 \\ -6 \end{pmatrix}$
(c)	$\begin{array}{r} 1011 \\ + 1110 \\ \hline 1001 \end{array}$	$\begin{pmatrix} -5 \\ -2 \end{pmatrix}$	(d)	$\begin{array}{r} 0111 \\ + 1101 \\ \hline 0100 \end{array}$	$\begin{pmatrix} +7 \\ -3 \end{pmatrix}$
(e)	$\begin{array}{r} 1101 \\ - 1001 \\ \hline 0100 \end{array}$	$\begin{pmatrix} -3 \\ -7 \end{pmatrix}$		$\begin{array}{r} 1101 \\ + 0111 \\ \hline 0100 \end{array}$	$\begin{pmatrix} +4 \end{pmatrix}$
(f)	$\begin{array}{r} 0010 \\ - 0100 \\ \hline 0110 \end{array}$	$\begin{pmatrix} +2 \\ +4 \end{pmatrix}$		$\begin{array}{r} 0010 \\ + 1100 \\ \hline 1110 \end{array}$	$\begin{pmatrix} -2 \end{pmatrix}$
(g)	$\begin{array}{r} 0110 \\ - 0011 \\ \hline 0111 \end{array}$	$\begin{pmatrix} +6 \\ +3 \end{pmatrix}$		$\begin{array}{r} 0110 \\ + 1101 \\ \hline 0011 \end{array}$	$\begin{pmatrix} +3 \end{pmatrix}$
(h)	$\begin{array}{r} 1001 \\ - 1011 \\ \hline 0110 \end{array}$	$\begin{pmatrix} -7 \\ -5 \end{pmatrix}$		$\begin{array}{r} 1001 \\ + 0101 \\ \hline 1110 \end{array}$	$\begin{pmatrix} -2 \end{pmatrix}$
(i)	$\begin{array}{r} 1001 \\ - 0001 \\ \hline 1000 \end{array}$	$\begin{pmatrix} -7 \\ +1 \end{pmatrix}$		$\begin{array}{r} 1001 \\ + 1111 \\ \hline 1000 \end{array}$	$\begin{pmatrix} -8 \end{pmatrix}$
(j)	$\begin{array}{r} 0010 \\ - 1101 \\ \hline 0101 \end{array}$	$\begin{pmatrix} +2 \\ -3 \end{pmatrix}$		$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$	$\begin{pmatrix} +5 \end{pmatrix}$

Figure 4: 2's complement add and subtract operations

Overflow in integer arithmetic: In 2's complement arithmetic addition of opposite sign numbers will never result in overflow . If the numbers are the same sign and the result is the opposite sign, overflow has occurred.

E.g. $0100 + 0111 = 1011$ (but 1011 is -5)

In case of unsigned numbers carry out signals that an overflow has occurred

Characters: Apart from numbers computers must be able to handle alphanumeric text information consisting of characters . Characters can be letters of alphabets, decimal digits , punctuation marks etc. Most widely used code was ASCII and now a days unicode is being used widely.

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(0010 1000
N	0100 1110	l	0110 1100)	0010 1001
				space	0010 0000

Figure 5: ASCII Codes

Position	Decimal	Name	Appearance
0x0901	2305	DEVANAGARI SIGN CANDRABINDU	ॠ
0x0902	2306	DEVANAGARI SIGN ANUSVARA	ॡ
0x0903	2307	DEVANAGARI SIGN VISARGA	ॢ
0x0905	2309	DEVANAGARI LETTER A	अ
0x0906	2310	DEVANAGARI LETTER AA	आ
0x0907	2311	DEVANAGARI LETTER I	इ
0x0908	2312	DEVANAGARI LETTER II	ई

Figure 6: Example of Uni Code

MEMORY LOCATIONS and ADDRESSES

Memory consists of storage cells. They store the bits 0 or 1. We can deal with them in n-bit groups called words (typically 8, 16, 32 or 64 bits). Usually refer to memory size in bytes e.g. we say we have 128MB memory and rarely use words as the unit. We use addresses to store or retrieve item of information. For some k, memory consists of 2^k unique addresses which range from 0 to $2^k - 1$. The possible addresses are the address space of the computer. E.g. 24-bit address has 2^{24} (16,777,216) locations. Information quantities: bit, byte, word where Byte=8 bits, word typically varies 16-64 bits. Most machines address memory in units of bytes. For a 32-bit machine, successive words are at address 0, 4, 8, 12 and so on.

Significant Bytes: Consider the hexadecimal (base 16) 32-bit number 34123F2A. This number is made up of four bytes 34, 12, 3F, 2A ($4 \times 8 = 32$ -bits). Bytes/bits with higher weighting are “more significant” i.e. the byte 34 is more significant than 2A. Bytes/bits with lower weighting are “less significant” i.e. 2A. Two ways byte addresses can be assigned across words

- More significant bytes first (big endian) - SUN/SPARC, IBM/RISC6000
- Less significant bytes first (little endian) - Intel Pentium Processors

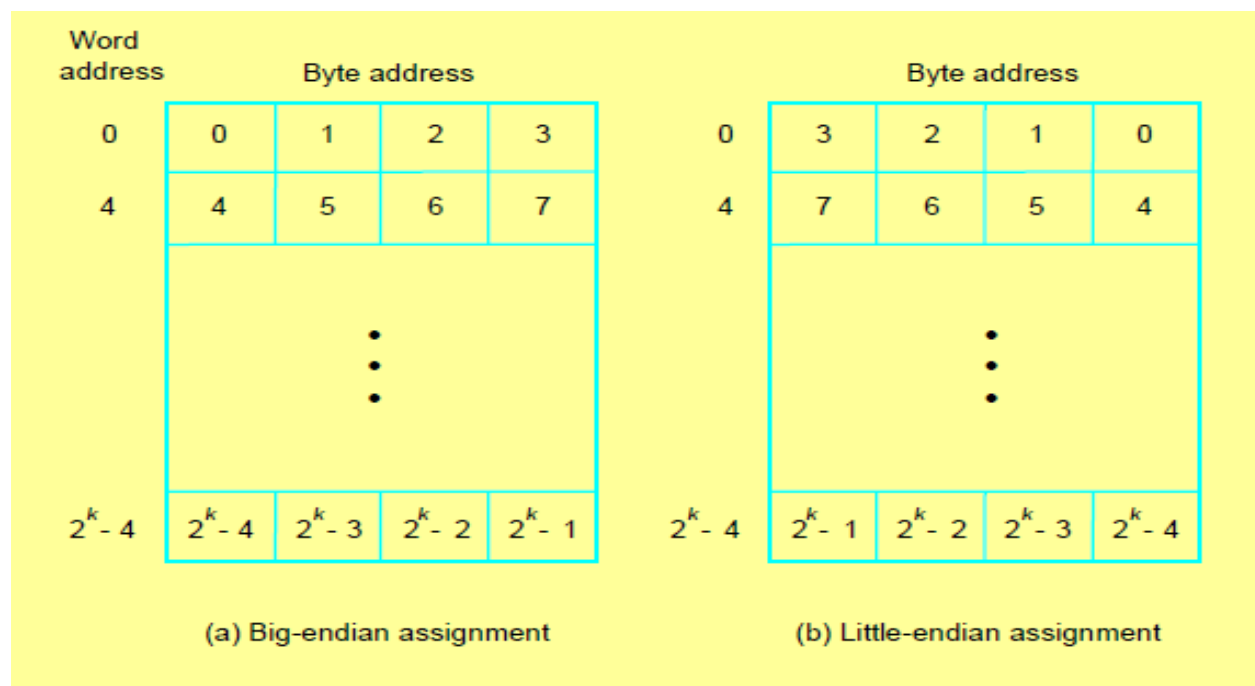


Figure 7: Byte and Word addressing

Big Endian and Little Endian: Consider a 32 bit integer (in hex): 0xabcdef12. It consists of 4 bytes: ab, cd, ef, and 12. Hence this integer will occupy 4 bytes in memory. Say we store it at memory address starting 1000. There are 24 different orderings possible to store these 4 bytes in 4 locations (1000 - 1003). 2 among these 24 possibilities are very popular. These are called as little endian and big endian.

- On little Endian system, memory will be like:

Address	Value
1000	12
1001	ef
1002	cd
1003	ab

- On Big Endian system, memory will be like:

Address	Value
1000	ab
1001	cd
1002	ef
1003	12

INSTRUCTIONS and INSTRUCTION SEQUENCING

A computer must have instruction capable of performing the following operations. They are:

- Data transfer between memory and processor register.
- Arithmetic and logical operations on data.
- Program sequencing and control.
- I/O transfer.

Register Transfer Notation: The possible locations that may be involved during data transfer are

Memory Location

Processor register

Registers in I/O sub-system.

Location	Hardware Binary Address	Expression	Description
Memory	LOC, PLACE, A, VAR2	$R1 \leftarrow [LOC]$	The contents of memory location LOC are transferred to the processor register R1.
Processor	R0, R1, R2...Rn	$[R3] \leftarrow [R1] + [R2]$	Add the contents of register R1 & R2 and the result of the operation is stored into register R3.

Assembly Language Notation:

Assembly Language Format	Description
Move LOC,R1	Transfers the contents of memory location LOC to the processor register R1.
Add R1,R2,R3	Add the contents of register R1 & R2 and stores their sum into register R3.

Basic Instruction Type:

Instruction Type	Syntax	Expression	Description
Three Address	Operation Source1,Source2,Destination	Add A,B,C	$C \leftarrow [A] + [B]$
Two Address	Operation Source,Destination	Add A,B	$B \leftarrow [A] + [B]$
One Address	Operation Operand	Add B	Content of B added with the content of the accumulator

INSTRUCTION EXECUTION and STRAIGHT LINE SEQUENCING

Instruction Execution: There are 2 phases for executing an instruction. They are,

- Instruction Fetch
- Instruction Execution

Instruction Fetch:

The instruction is fetched from the memory location whose address is in PC. This is then placed in IR.

Instruction Execution:

Instruction in IR is examined and decoded to determine which operation is to be performed.

Program execution Steps:

To begin executing a program, the address of first instruction must be placed in PC.

The processor control circuits use the information in the PC to fetch & execute instructions one at a time in the order of increasing order.

This is called Straight line sequencing. During the execution of each instruction, the PC is incremented by 4 to point to the address of next instruction.

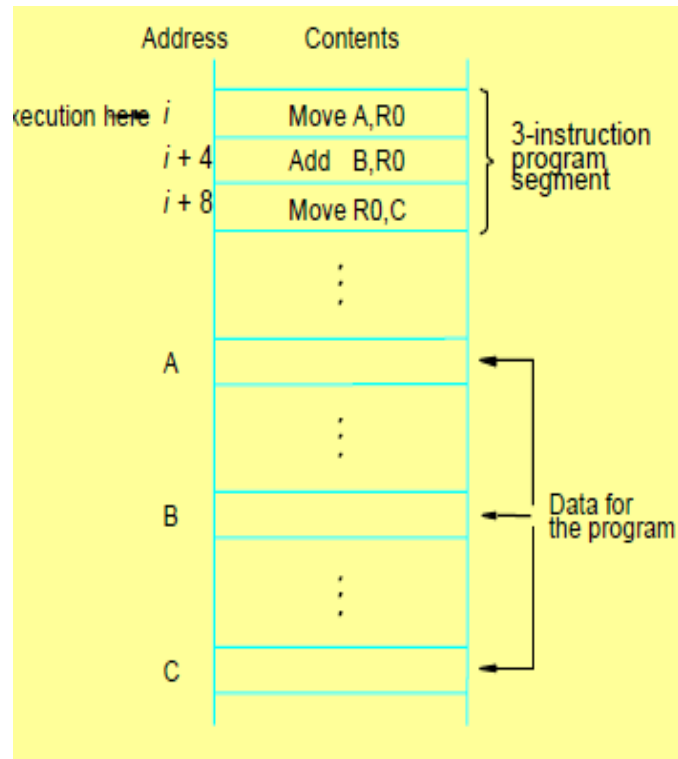


Figure 8: A program for $C \leftarrow [A] + [B]$

Branching: The Address of the memory locations containing the n numbers are symbolically given as NUM1, NUM2, ..., NUM n .

Separate Add instruction is used to add each number to the contents of register R0.

After all the numbers have been added, the result is placed in memory location SUM.

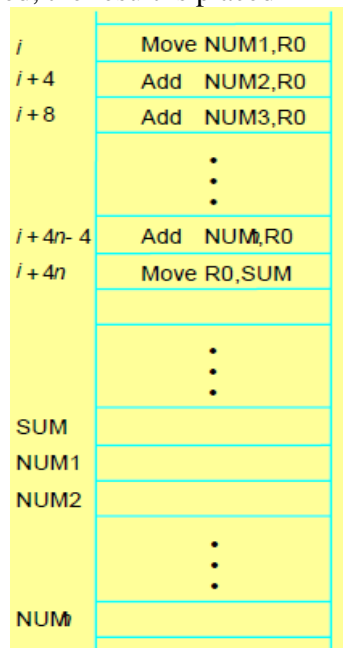


Figure 9: A straight line program for adding n numbers

Using loop to add 'n' numbers:

- Number of entries in the list „n is stored in memory location M. Register R1 is used as a counter to determine the number of times the loop is executed.
- Content location M are loaded into register R1 at the beginning of the program.
- It starts at location Loop and ends at the instruction, Branch>0. During each pass, the address of the next list entry is determined and the entry is fetched and added to R0.
- Decrement R1; It reduces the contents of R1 by 1 each time through the loop.
- Branch >0 Loop; A conditional branch instruction causes a branch only if a specified condition is satisfied.

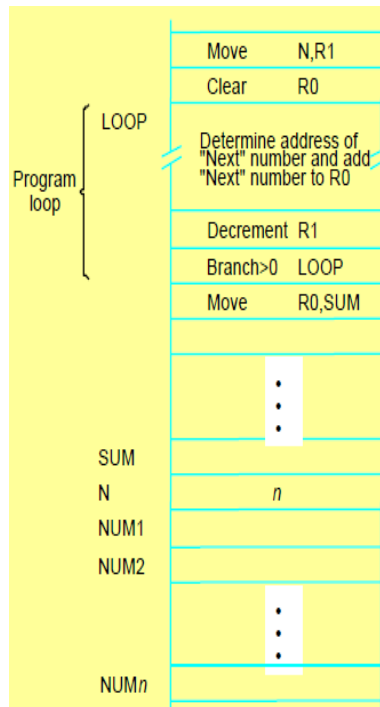


Figure 10:Using a loop to add n numbers

Conditional Codes:In order to do conditional branches and other instructions, operations implicitly set flags. Four commonly used (1-bit) flags

- N (negative) 1 if result –ve else 0
- Z (zero) 1 if result 0 else 0
- V (overflow) 1 if arithmetic overflow occurs else 0
- C (carry) 1 if carry out occurs –ve else 0

ADDRESSING MODE

The different ways in which the location of an operand is specified in an instruction is called as Addressing mode.

Generic Addressing Modes:

- Immediate mode
- Register mode
- Absolute mode
- Indirect mode
- Index mode
- Base with index
- Base with index and offset
- Relative mode
- Auto-increment mode
- Auto-decrement mode

Implementation of Variables and Constants:

Variables:

The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are

- Register Mode
- Absolute Mode

Register Mode:

The operand is the contents of the processor register. The name (address) of the register is given in the instruction.

Absolute Mode (Direct Mode):

The operand is in a memory location. The address of this location is given explicitly in the instruction.

The various addressing modes and their assembler syntax and functions are as shown in figure below:

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R i	EA = R i
Absolute (Direct)	LOC	EA = LOC
Indirect	(R i) (LOC)	EA = [R i] EA = [LOC]
Index	X(R i)	EA = [R i] + X
Base with index	(R i ,R j)	EA = [R i] + [R j]
Base with index and offset	X(R i ,R j)	EA = [R i] + [R j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R i)+	EA = [R i] ; Increment R i
Autodecrement	– (R i)	Decrement R i ; EA = [R i]

EA = effective address
Value = a signed number

Figure 11:Generic addressing modes

Immediate Addressing Mode: The operand is specified in the instruction itself .

Move 200immediate, R0

Move #200, R0

Direct Addressing Mode: Operand resides in Memory and its address is given explicitly in the address field of an instruction.

Move P, R0

Move R0, S

Add Q, R0

Register Addressing Mode: name of the register (address code of a specific general purpose register) appears in the address field of an instruction i.e.

Move B, R1

Indexing and Arrays:

Index Mode:

The effective address of an operand is generated by adding a constant value to the contents of a register. The constant value uses either special purpose or general purpose register.

X (RI)

where **X** – denotes the constant value contained in the instruction

Ri – name of the register involved.

The Effective Address of the operand $EA = X + [Ri]$

The index register R1 contains the address of a new location and the value of X defines an offset (also called a displacement).

To find operand first go to Reg R1 (using address)-read the content from R1 i.e. 1000

Add the content 1000 with offset 20 to get the result. Here the constant X refers to the new address and the contents of index register that defines the offset to the operand.

The sum of two values is given explicitly in the instruction and the other is stored in register.

Add 20(R1), R2 (or) $EA \Rightarrow 1000 + 20 = 1020$

Relative Addressing: It is same as index mode. The difference is, instead of general purpose register, here we can use program counter (PC).

Relative Mode:

The Effective Address is determined by the Index mode using the PC in place of the general purpose register.

This mode can be used to access the data operand. But it's most common use is to specify the target address in branch instruction. Eg. Branch>0 Loop

It causes the program execution to go to the branch target location. It is identified by the name loop if the branch condition is satisfied.

Additional Modes:

There are two additional modes. They are

- Auto-increment mode
- Auto-decrement mode

Auto-increment mode: The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

Auto-decrement mode: The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

Assembly Language

We generally use symbolic names to write a program. A complete set of such symbolic names and rules for their use constitute a programming language, is referred to as assembly language.

LOAD	- To load operand from memory
STORE	- To store operand to memory
MOVE	- To transfer data from one location to another location/Register

Assembler Directives: Directives are the assembler commands to the assembler concerning the program being assembled. These commands are neither translated into machine opcode nor assigned any memory location in the object program.

S EQU 150

EQU directs the assembler that the symbolic name S must be replaced with memory location address 150,

ORIGIN 201

Instruct assembler to initiate data block at main memory locations starting from 201

N DATAWORD 40

Inform the assembler that value of N i.e. data value 40 is to be placed in the memory location 201.

ORIGIN 100

States that assembler directive must load machine instructions of the object program in the main memory starting from location 100.

END START

End of the program and the label of where program starts

N1 RESERVE 400

Reserve memory block of 400 bytes

Assembler: Has to know

- How to interpret machine language (directives, instructions, addressing modes etc)
- Where to place the instructions in memory
- Where to place the data in memory ; Scans through source program, keeps track of all names and corresponding numerical values in symbol table e.g. what all the labels mean
- Calculate branch addresses; Forward branch problem – how can it work out forward addresses?

Two Pass Assembler:

- First pass
 - Work out all the addresses of labels
- Second pass
 - Generate machine code, substituting values for the labels

BASIC INPUT/OUTPUT OPERATIONS

I/O is the means by which data are transferred between the processor and the outside world. Devices operate at different speeds to the processor so handshaking is required.

Keyboard/display Example: The keyboard and display are coordinated via software

- Register (on device) assigned to the keyboard hardware
 - DATAIN contains ASCII of last typed character
 - SIN is the status control flag, normally 0. When a character typed, becomes 1.After the processor reads DATAIN, it is automatically set back to 0
- Register (on device) assigned to the display hardware
 - DATAOUT receives a character code
 - SOUT is the status control flag. It is 1 when ready to receive a character, set to 0 when the character is being transferred
- These registers form the respective device interface

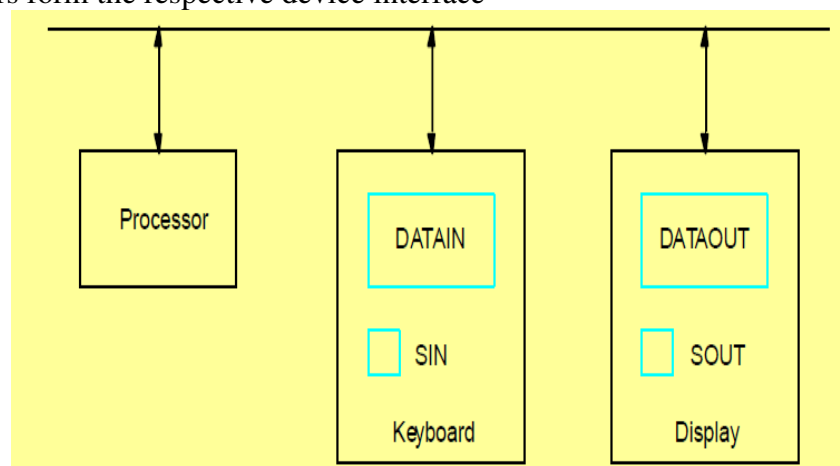


Figure 12: Bus Connection for processor, keyboard and display

Memory mapped IO and IO mapped IO:

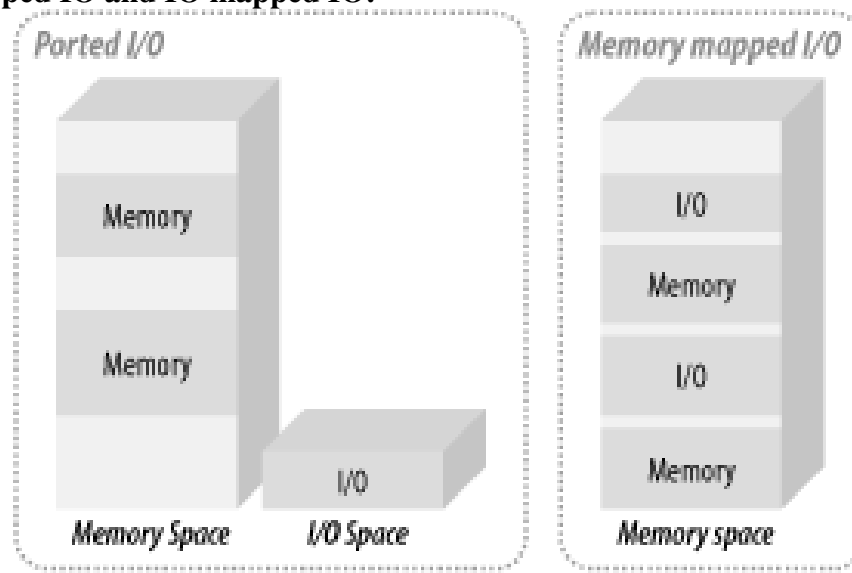


Figure 13: Memory mapped IO and IO mapped IO

I/O Driver program (Programmed IO):

READWAIT Branch to READWAIT if SIN=0

INPUT from DATAIN to R1

WRITEWAIT Branch to WRITEWAIT if SOUT=0

Output from R1 to DATAOUT

Memory Mapped IO:

- On many machines, registers such as DATAIN, DATAOUT are memory-mapped
 - Read and write specific memory locations to communicate with device
 - Move Byte DATAIN, R1
 - Move Byte R1, DATAOUT
- SIN and SOUT might be bits in a device status register e.g. bit 3

READWAIT Branch to READWAIT if SIN=0

INPUT from DATAIN to R1

READWAIT Test bit #3, INSTATUS

Branch=0 READWAIT

Move Byte DATAIN, R1

WRITEWAIT Branch to WRITEWAIT if SOUT=0
 Output from R1 to DATAOUT

WRITEWAIT Test bit #3, OUTSTATUS
 Branch=0 WRITEWAIT
 Move Byte R1, DATAOUT

Program to read a line of character and display it :

	Move	#LOC,R0	Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered in the keyboard buffer DATAIN.
	Branch=0	READ	
	MoveByte	DATAIN,(R0)	Transfer the character from DATAIN into the memory (this clears SIN to 0)
ECHO	TestBit #3,	OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(R0)+	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

STACKS and QUEUES

List of data elements (usually bytes or words). Elements can only be removed at one end of the list. Last-in-first-out. Can be implemented in several ways, one way is

- First element placed in BOTTOM
- Grows in direction of decreasing memory address
- Assume 32-bit data

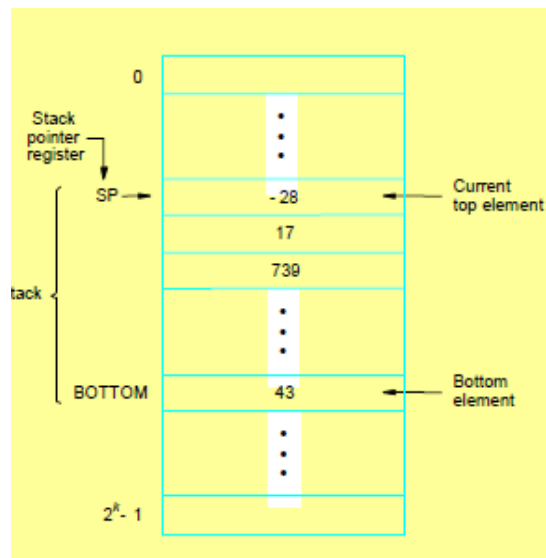


Figure 14: A stack of words in the memory

Stack Implementation:

Subtract #4, SP

Move NEWITEM,(SP) ; push

Move (SP), ITEM; pop

Add #4, SP

With auto increment and auto decrement

Move NEWITEM,-(SP); push

Move (SP) +, ITEM; pop

Queue:

- First-in-first-out
- Unlike a stack, need to keep track of both the front and end for removal and insertion respectively
- Need two pointers to keep track of both ends
- Assuming it moves through memory in direction of higher addresses, as it is used, it walks through memory towards higher addresses.

Circular buffers:

- Avoid this problem by limiting to a fixed region in memory
- Start at BEGINNING and entries appended until it reaches END after which it wraps back around to BEGINNING
- Need to deal with cases when it is completely full and completely empty

SUBROUTINES

- Often need to perform subtask on different data we use subtask called a subroutine
- Rather than include the same sequence of instructions everywhere it is needed, call a subroutine instead
 - One copy of subroutine stored in memory
 - Subroutine call causes a branch to the subroutine
 - At the end of the subroutine, a return instruction is executed
 - Program resumes execution at the instruction immediately following the subroutine call

Parameter Passing:

Subroutine call

- e.g. SUM = listadd (N, NUM);
- N is a variable in memory and NUM is an address pointing to the start of the NUM list
- How do we send the parameters N, NUM to the subroutine?
- How do we receive the return value SUM?

One way is putting the parameters in registers; second way is Passing Parameters on stack and third way is passing by value and reference.

CISC & RISC

- Multiple length instructions are difficult to implement with high clock rate
- Complex instruction set computers (CISC) have complex instruction encodings like this (e.g. IA-32)
- Reduced instruction set computers (RISC) only allow simple 32-bit formats, few addressing modes and all data to be manipulated must be in registers e.g. Add (R3),R2 is not allowed, instead use Move (R3),R1 followed by Add R1,R2 (e.g. ARM)

RISC machines often are 3-address machines as the addressing mode field is either not necessary or simplified e.g. Add R1, R2, R3

- CISC machines usually require less instructions but have a lower clock rate, RISC require more instructions but have a higher clock rate