Instruction Set Architecture

An understanding of machine instructions is a important to an understanding of computer architecture. This is because the set of machine instructions reflects the architecture of the computer. This chapter presents the important features and attributes of machine instructions, such as formats, sizes, opcodes, and addressing modes.

Instruction set Architecture is a portion of the computer visible to machine language programmer / compiler writer. The type of the instructions, for a particular machine also depends on the execution environments. These are classified as follows.

Desktop: For these environments, the perfromance of programs is decided by integer and floating-point arithmetics, with little concern to power consumption, program size and applications. Main use is browsing, and limited computations. These programs use compiler generated code.

Servers: These environments are *data* or *file servers*, web applications, time sharing applications for large number of users.

Real-time and embedded systems : These environments are concerned with low cost and power, small code size, e.g., DSP (digital signal processing) and media processors, continuous streaming of data, fast execution of code (targeting the worst case performance). These code is generated manually optimized.

0.1 Classification of Instruction set Architectures

The instruction set for any processor depends on whether it is a *High performance Systems*, like RISC (reduced instruction set computing) architecture or *General Architectures*. Following is Classification of instruction set architectures:

1. *Stack Architecture*: All the operands are implicitly on top of stack.
2. *Accumulator Architecture*: One operand is specified explicitly, and other is implicitly in accumulator.
3. *General Purpose Register(GPR) architecture*: The operands are explicitly in registers or memory.
4. *Memory-Memory Architecture*: All operands are in memory.

In the GPR, the registers are faster, and more efficient to use by the compilers. For example, to compute the expression $(A * B) - (B * C) - (A * D)$, multiplications can be evaluated in any order by GPR but not by stack machine. For example, $B * C$ can be done before $A * B$ or other way.

The figure 1 shows the locations for different Instruction set architectures.

The stack machine operands are accessible from stack. Only the top most operand is directly accessible, as it is pointed by the stack-pointer (SP). The SP can be incremented to "push" the new operand, and can be decremented to retrieve, i.e., "POP" the the operand.

Example 0.1.1 *Computer $C = A + B$ using different architectures.*

The addition can be performed in different machines as follows:

```
\\ compute C = A + B
Stack:      Accumulator:  Register-memory
push A      Load A        //accesses memory as part of
push B      Add B         // instruction
Add         store C       Load R1, A
Pop C                     Add R3, R1, B
                          Store R3, C

Register-register:
//accesses memory through load/store instruction
```

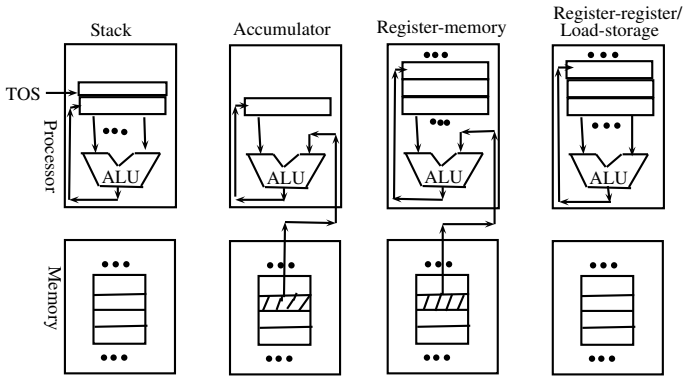



Figure 1: Instruction set architectures.

```
Load R1, A
Load R2, B
Add  R3, R1, R2
Store R3, C

Memory-memory architecture:
// For data movements

Load R1, addr1
Load R2, addr2
Load R3, count
Move R4, R1, R2 ; move data from addr1 onwards
; to addr2 onwards, decrementing count in R3,
; and incrementing R1, R2 each time until R3 == 0
```

□

The following examples demonstrates the case of stack machine to compute an expression.

Example 0.1.2 *Evaluate the expression* $(A * B) - (B * C) - (A * D)$.

```
Stack Machine:
PUSH A
PUSH B
MULT
PUSH B
PUSH C
```

```
MULT
SUB
PUSH A
PUSH D
MULT
SUB
POP T ; retrieve the result into loation T
```

□

Example 0.1.3 *Construct parse-tree for* $(A * B) - (B * C) - (A * D)$.

The figure 2 shows the parse-tree for the expression: $(A * B) - (B * C) - (A * D)$. When it is traversed in post-order we get the expression $AB * BC * -AD * -$. This post order expression is used for stack based computation. The Algorithm for computation using a post-order expression and a stack is given as algorithm 1. The *pop* operation stands for retrieving an operand from the top of stack. The pop1 stands for first popped value and pop2 stands for next popped value.

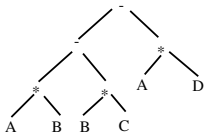


Figure 2: Tree for $(A * B) - (B * C) - (A * D)$.

□

The following examples demonstrates the case of Register-Memory Addressing to evaluate the expression $(A * B) - (B * C) - (A * D)$.

Example 0.1.4 *Evaluate the expression* $(A*B)-(B*C)-(A*D)$ *using register-memory instruction formats.*

```
MULT E, A, B; A-F registers
MULT F, B, C
SUB F, E, F
MULT E, A, D
SUB T, F, E; T is memory location
```

0.1.1 Issues in Instruction set Design

When variables are allocated to cpu registers, it would lead to reduction in memory traffic, hence would result to program speeds up. The code

Algorithm 1 Postfix-evaluate(post-fix expression, stack, stack-pointer

```
1: while True do
2:   read input
3:   if NUL then
4:     Return pop
5:   else
6:     if input is operand then
7:       push on to stack
8:     else
9:       if input is operator α then
10:        push(pop2 α pop1)
11:      else
12:        return error
13:      end if
14:    end if
15:  end if
16: end while
```

density reduces as registers are named with fewer bits. But not too many variables can be allocated to registers due to limited count of CPU registers.

A register can also be used as index to access a memory locations, that are contiguous. This is possible by incrementing the data address register after every memory fetch. We may need need such instructions, for example, to sum the continuous locations in memory.

In *R-M instruction* architecture, called register-memory instructions, one operand is in register and another in memory. Hence, in a instruction there are three fields, opcode, register number, and memory location. Here, there is flexibility of having large variables in memory, but such instructions would be little longer, consequently slows fetch, hence, slow program execution, as well its size is more.

In memory-memory instructions, called *M-M instructions* architectures, a instruction has two address operands. The length of instruction is longer, hence slow speed of program execution, but an instruction has flexibility to access any two arbitrary locations in the RAM.

To further improve the execution efficiency, frequently used instructions are coded with shorter length, while the less frequent instructions are coded longer. More are the memory addresses specified inside an instruction, more flexible and powerful it becomes. But slow in execution. On the other side, compact instructions, with too few operands represented inside an instruction makes the instruction compact, but one

requires to use number of instructions, even to perform a small operations.

A memory address my be directly specified as a constant in a instruction, or may be content of register, or may be at a memory whose address is part of an instruction (indirect address).

If number of bytes, say 4-bytes, of a 32-bit words are stored in a memory some sequence of memory locations, there are two ways for doing it. A design may support of storing MSB at last location, then it is called Big Endian. In the Little Endian, the lower bytes are stored in last locations, and higher bytes in the begin location. For example, assume that 4 byes are 1A, 2B, 3C,4D in hexadecimal format, with 4D as LSB. The available memory locations are say, 10-13. When 1A stored at location number 10, and 4D at location 13, it is called *Little-endian*. With 4D at 10, and 1A at 13, it is *big-endian*.

0.1.2 Instruction format

Most instructions specify a register transfer operation of the form: an opcode followed with a set of n operands, e.g., $X_1 = f(X_1, X_2, \ldots, X_n)$, for example, ADD A, B, will some A and B and put the result in A. Figure 3 shows different instruction formats. The instruction I_0 has no oerand specified, it is implicit. In accumulator machines, the implicit location is accumulator, and in stack machines the implicit loccation is top of the stack. Example of instruction, in accumulator machines, the instruction may be "complement accumulator" (CMA in 8085 microprocessor), in a stack machine, the instruction may be "Add", which adds the two top most locations of stack and pushes the result back onto the stack.

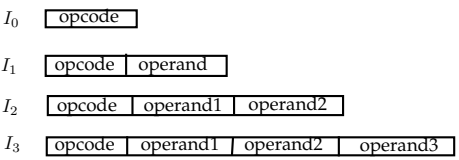


Figure 3: Instruction formats.

The instruction I_1 has one operand explicitly specified, the other may be accumulator or stack. For example, instruction "Add B" adds the register B into accumulator, or "Add #201" adds constant 201 into accumulator, or "Add 201" adds the contents of memory location 201 into the accumulator. In the casse of stack, "Push B" pushes (saves) the contents of register B onto the top of stack, and "POP B" retrives the top of stack

into the register B.

The instruction I_2 has two operands, both explicitly specified. For example, instruction "Add A B" adds the register B into register A, or "Add A, #201" adds constant 201 into register A, or "Add A 201" adds the contents of memory location 201 into the register A. The stack does not require two address instruction.

The instruction I_3 has three operands, all explicitly specifiedFor. For example, instruction "Add A B C" has the effect of computation $A \_ B+C$. The A, B, C can all be memory addresses, but only B and C can be constants.

How an instruction specifies the addresses for operands, it is called *Addressing Mode* of that instruction. As we are going to discuss in details in the subsequent sections, the address can be directly specified, it can be in a register or memory location, or operands are in register, or they are explicily given as literal in a instruction, accordingly, it is called in order, as direct, indirect, register, and immediate address instruction, respectively.

0.2 Intel 8085 Architecture

Intel 8085 is a genral purpose 8-bit microprocessor, with 64k bytes memory, 8/16 bit cpu registers, and 8-bit IO address to address total 256 IO devices. Following are the details:

- 8-bit microprocessor (word length = 8-bits)
- Stores 8-bit data (registers, accumulator, memory locations)
- Performs arithmetic, logic, and data movement operations using 8-bits
- Tests for conditions (if/then)
- Sequence the execution of instructions (conditional and unconditional: jumps, call, and return)
- Stores temporary data in RAM & register during runtime

0.2.1 Intel 8085 Bus structure

Following are the bus-level details for the 8085-microprocessor.

- 8-bit internal CPU bus

- Communicates with other units through *16-bit address bus, 8-bit data bus, and control bus*
- Address bus: $A_0 - A_{15}$, total addressable memory=2^{16} = 65536 (64k). Address locations 0 - 65535 (0000H - FFFFH).
- Databus $D_0 - D_7$ (Little Endian), multiplexed with lower 8 bits ($A_0 - A_7$) of address bus ($A_0 - A_{15}$).
- Control bus: Various signal lines (binary) carrying signals like Read/write, Enable, Ready, Flag bits, etc.

0.2.2 Intel 8085 Registers

Following are the register specifications of 8085 microprocessor.

- ACC + 6 general purpose registers (8-bit): A(111), B(000), C(001), D(010), E(011), H(100), L(101), which can be used to form 3 no. of 16-bit registers, BC(00), DE(01), HL(10), SP(11): two bits in 1st byte. The binary values in parentheses indicates the identifier of registers and register pairs.
- Accumulator + Flag register = PSW (processor status register) (status: Z, S, P, C, AC)
- *Flag bits:* To indicate the result of condition: C(carry), Z(zero), S(sign minus), P(sign plus), AC(auxiliary carry)
- Flag bits are used as Tests for conditions (if/then)
- *Program Counter (PC)*: Contains memory address of next instruction
- *Stack Pointer(SP)*: holds the return address for subroutine call, can save registers(PUSH, POP Instructions)

The binary number in parentheses indicate the code of the corresponding register, used as part of in instruction in place of operation code. The figure 4 shows the register-level architecture of 8085 microprocessor.

0.3 Intel 8085 assembly language programming

The assembly language is machine language for any processor; the only difference between the machine language in binary forms, and the assembly is that in assembly language instructions are represented in mnemonic

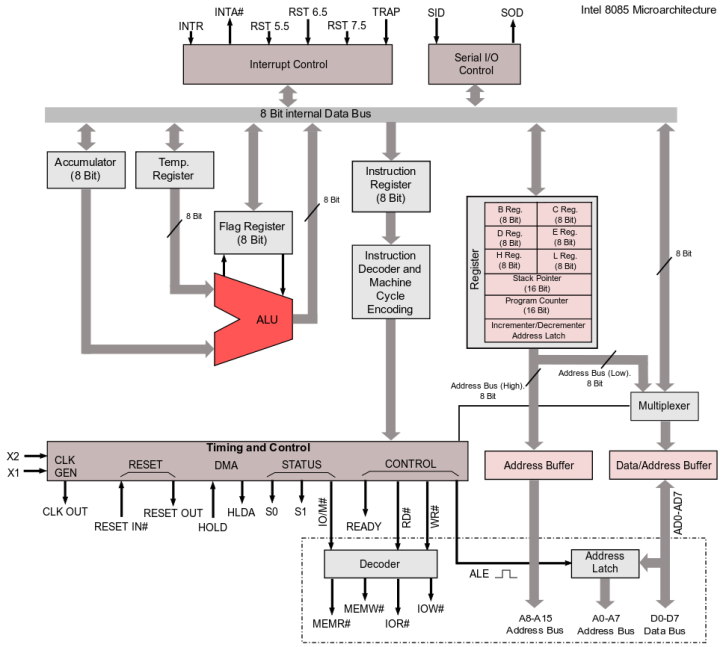


Figure 4: Microprocessor 8085 architecture (ref. Intel 8085 Handbook).

form, and operands and address are in decimal or hexadecimal or in octal format instead of binary strings. Each assembly instruction has one-to-one correspondence with the machine instruction of that processor.

Example 0.3.1 *Program to add two integers.*

```
MVI A, 7BH
MVI B, 67H
ADD B
HLT
```

□

Example 0.3.2 *Program to multiply a given no. by number 4.*

```
MVI A, 30H
RRC
RRC
MOV B, A
HLT
```

□

Example 0.3.3 *Program to find greater of two numbers.*

```
     MVI B, 30H
     MVI C, 40H
     MOV A, B
     CMP C
     JZ eq
     JP gt
     JM lt
eq:  MVI D, 00H
     JMP stop
gt:  MVI D, 01H
     JMP stop
lt:  MVI D, 02H
stop: HLT
```

□

0.3.1 Intel 8085 Instruction set

The instruction set of any machine is designed based on two criteria: 1) it should be possible to code any algorithm using that instruction set, 2) all the operands should be efficiently accessible using the addressing modes available in the instruction set. The figure 5 shows the instruction set of 8085 processor.

Intel 8085 addressing modes

Immediate addressing: (MVI B, 25H)

Direct addressing: (LDA 1020H)

Register addressing: (MOV B, C)

Implied addressing: (CMA, RAR)

Register Indirect addressing: (MOV A, M; ADD M).

Register Indirect addressing: (LDAX B, LDAX D, STAX B, STAX D)

Example 0.3.4 *Sum five locations and store the result at subsequent location.*

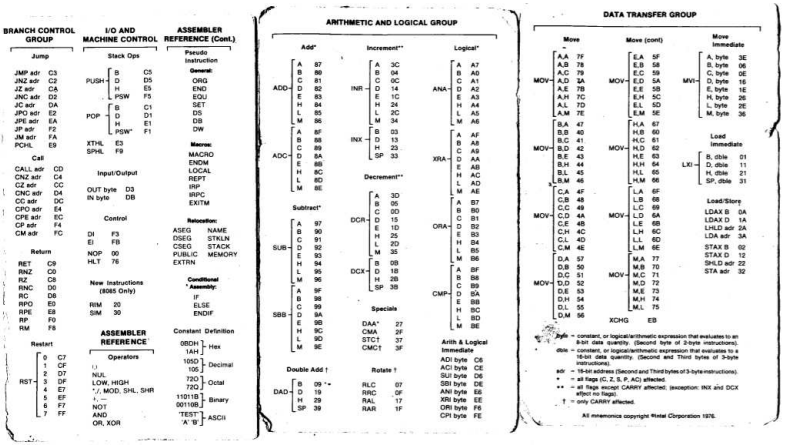


Figure 5: 8085-Instruction set (ref. Intel 8085 Handbook).

```
;Code to sum five locations and store the result at subsequent
;location:
      LXI H, 1010H ; memory pointer for start of data
      MVI C, 05H ; initialize counter value
      XRA A ; Exclusive OR A with itself
loop: ADC M ; add memory into Accumulator with carry
      INX H ; increment the memory pointer register pair HL
      DCR C ; decrement the counter
      JNZ loop ; if counter not zero then repeat the loop
      MOV M, A ; store the sum at subsequent location.
      HLT ; otherwise halt the processor
```

□

Example 0.3.5 *Copy paste 1000H to 2000H for 25H locations.*

```
;Code to copy paste 1000H to 2000H for 25H locations:
       LXI B, 1010H ; source pointer
       LXI D, 2000H ; destination pointer
       MVI H, 25H ; counter initialize
 loop: LDAX B
       STAX D
       INX B
       INX D
       DCR H
```

```
JNZ loop
HLT
```

□

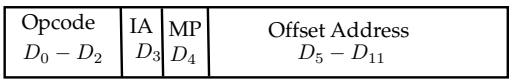
0.4 PDP-8 Architecture

PDP-8 was introduced in 1965, and by today's standards its architecture and capabilities are archaic (truly a dinosaur, except the PDP-8 was small). However, its simple design and low cost made the PDP-8 a very successful computer (50,000 units were produced). Due to its simple design, PDP-8 assembler is easy to understand, write, and use. Even PDP-8 machine coding can be done without too much difficulty!

The memory of the PDP-8 consists of 4096 (2^{12}) 12-bit words. Bits in a word are numbered left to right 0-11, with 0 bit being *most significant* bit. Memory is partitioned into 32 pages (5-bits address) of 128 words each (7-bit address). There is a 12-bit Accumulator with 1-bit *link* register (L) tht captures any carry out of the accumulator. PDP-8 instructions use the implicit link-accumulator pair.

A 12-bit multiply-quotient register (MQ) is used by multiplication and division operation in PDP-8 model. Special purpose registers include a 12-bit switch register (SR) to enter values from the console of PDP-8; 12-bit program counter (PC) that holds the address of the next instruction to be executed; a 3-bit *instruction register* (IR); 12-bit central processor memory address register (CPMA) to address data; a 12-bit memory buffer register (MB) to read or write data through it.

PDP-8 has 8 opcodes and 3 instruction formats, (fig. 6). Opcodes 0-5 are memory reference instructions (MRI); opcode 6 is family of 10 instructions; 7 is set of orthogonal micro-instructions that operate the link accumulator pair.



Bits 0 - 2: Operation code
Bit 3: Indirect addressing (0: direct, 1: indirect)
Bit 4: Memory page (0: zero page, 1: current page)
Bits 5 - 12: Offset address

Figure 6: PDP-8 Instruction format.

The MRI format has room for 7-bit offset address. To obtain 12-bit address, either five zeros are prefixed to the address (zero page addressing

is indicated by clearing bit D_4 of instruction to 0) or five leading bits of the address of instruction ($A_0 - A_4$) is prefixed to the offset. The table 1 shows the type of instructions in PDP-8 machine.

Table 1: PDP-8 Instruction set.

Machine code	Opcode	Remarks
0	AND	ACC $\leftarrow AC \wedge$(effective-address)
1	TAD	Twos complement add
2	ISZ	Increment and skip if zero (counter)
		effective-addr $\leftarrow$ effective-address + 1
3	DCA	Deposit and clear accumulator
4	JMS	Jump to the subroutine
		Effective-addr $\leftarrow$ PC
		PC $\leftarrow$ effective-addr + 1
5	JMP	Jump to effective address

A memory reference instruction has four fields: the 3-bit Opcode field, an Indirect bit (bit D_3), a Memory Page bit (bit D_4), and a 7-bit page offset field. If Memory Page bit D_4 is zero, the 7-bit page offset refers to page zero (zero-page addressing); if bit D_4 is 1, the page offset refers to the same page as the instruction (current page addressing, i.e., the page in which the instruction exists). Thus only two out of 32 pages of memory are directly addressable; indirection is needed to access the other 30.

Opcode 6 is a class of I/O instructions where bits 3 - 8 are the device field and bits 9 - 11 are the function field.

Opcode 7 is a class of 20 plus orthogonal *microinstructions*, divided into three groups depending on how bits 3 and 11 are set, each of the remaining bits 4 - 10 toggles a different microinstruction which can be combined to create compound instructions.

One of the opcode 7 groups contains instructions shown in table 2.

Table 2: PDP-8 Opcode group 7 Instructions.

Mnemonic & Octal code	Comment
(CLA - 7200)	clear the accumulator
(CLL - 7100)	clear the link
(CMA - 7040)	complement the accumulator
(CML - 7020)	complement the link
(IAC - 7001)	increment the accumulator
(RAR - 7010)	rotate the link accumulator pair right
(RAL - 7004)	rotate the link accumulator pair left

The opcodes in table 2 can be freely combined, as shown in table 3.

Table 3: Combining PDP-8 group 7 instructions.

Mnemonic & Octal code	Comment
CLA CLL - 7300	clears both the accumulator and link
CMA IAC - 7041	negates the accumulator

Conditional branching is done by a second opcode 7 group. These include instructions shown in table 4.

Table 4: Conditional branching in PDP-8 group 7 instructions.

Mnemonic & Octal code	Comment
(SMA - 7500)	skip on minus accumulator
(SZA - 7440)	skip on zero accumulator
(SNL - 7420)	skip on non-zero link

Setting bit 8 for this group will reverse the sense of each yielding the instructions shown in table 5.

Table 5: Bit-D8 reverses the sense of condition.

Mnemonic & Octal code	Comment
(SPA - 7510)	skip on positive accumulator
(SNA - 7450)	skip on non-zero accumulator
(SZL - 7430)	skip on zero link
(HLT - 7402)	halt is a new in this group

A third group of opcode 7 microinstructions make use of the PDP-8's Extended Arithmetic Element, an option that implemented an extended set of instructions that included multiplication and division.

0.4.1 PDP-8 Addressing Modes

There are four addressing modes for the PDP-8. Zero page and Current page (direct) addressing have already been mentioned, modes that allow a seven-bit operand to be expanded to a twelve-bit memory address. Since *direct addressing* is restricted to two of the thirty-two pages of memory, a third mode, *indirect addressing* (bit $D_3 = 1$) allows access to all of memory. The fourth mode is auto-indexing. The eight memory locations $0010_8 - 0017_8$ on page Zero are special auto-index registers. Whenever one of these auto-index registers is addressed indirectly, its contents are first incremented then used as the effective address for the operand.

0.4.2 PDP-8 Machine coding

As a first exercise, let us write PDP-8 machine code programs. To keep the job simple, the set of machine codes is restricted to the four memory reference instructions.

Octal code	Mnemonic
1	TAD
2	ISZ
3	DCA
5	JMP

And the five opcode 7 "micro-instructions"

Mnemonic	Octal code
CLA CLL	7300
CMA IAC	7041
SMA	7500
SPA	7510
HLT	7402

You may write and execute a series of short programs covering both conditional branching and loops and ending in a routine to do multiplication (or division) by repeated addition (or subtraction). Only current page direct addressing is used, so the only tricky part of the assignment (apart from being restricted to a small subset of simple operations) is encoding the effective addresses for the memory reference instructions.

For example, the following program does the multiplication $P = M \times N$ by adding M to itself N times. The program is given in address/contents format (e.g. 0200/7300 means address 0200 contains the value 7300 - both values in octal) with attached comments. Observe that counting loops are implemented by setting the loop counter to a negative value (addresses 202, 203, and 207) then using the ISZ and JMP instructions (addresses 210, 211) to control the loop. The program also tests if N is zero (addresses 204, 205) halting the program if it is.

```
0200/7300  ;(CMA, CLL) clear AC and link
0201/3303  ;(DCA) store (zero out) P
0202/1302  ;(TAD) load N
0203/7041  ;(CMA, IAC) negate
0204/7500  ;(SMA) skip if negative (N != 0)
0205/5213  ;(JMP) otherwise jump to end
```

```
0206/3300  ;(DCA) store in CNT (loop counter)
0207/1301  ;(TAD) add M
0210/2300  ;(ISZ) inc CNT and skip if zero
0211/5207  ;(JMP) loop back
0212/3303  ;(DCA) deposit accumulator to P
0213/7402  ;halt
0214/5200  ;(JMP) go again
0300/0000  ;CNT loop counter
0301/0004  ;M Multiplicant
0302/0005  ;N Multiplier
0303/0000  ;P Product
```

0.4.3 Subroutines in PDP-8

The PDP-8 implements a simple subroutine call and return mechanism. Opcode 4, JMS (Jump to Subroutine) stores the program counter at the first word of the subroutine (the effective address of the instruction) then transfers control to the second word (the effective address + 1). A return is performed by an indirect jump through the first word of the subroutine (e.g. JMP I SUBR); no special return instruction is required. The simplicity of the PDP-8 subroutine call and return makes them easy to introduce while the simplicity of the PDP-8 instruction set necessitates subroutine use for all but the simplest of programming tasks.

0.4.4 PDP-8 I/O

The PDP-8 uses programmed I/O with wait loops. Opcode 6 instructions for I/O use bits 3 - 8 to identify the I/O device (by convention 03 is a keyboard and 04 is a printer) and bits 9-11 to encode the I/O function as an extended opcode.

I/O is done by transferring eight bit ASCII characters between the device buffer and the accumulator. Synchronization with the device is achieved by testing a one-bit status flag for the device (0 for busy, 1 for ready) using a "skip on flag set" instruction. For example, the following code (lines 3 - 5) is used to read a character.

```
1 *7400          ; page 30
2 XGetChar, 0    ; return address here
3 KSF            ; skip on kbd flag
4 JMP .-1        ; otherwise loop
5 KRB            ; read char to accumulator
6 JMP I XgetChar ; return
```


The KSF instruction (Keyboard Skip on Flag raised) checks if the keyboard device is ready. If not, it does not skip, and the following JMP instruction (the period is PAL for current address) loops control back to the KSF instruction; otherwise it skips to the KRB instruction (Keyboard Read Buffer) which transfer the character in the keyboard buffer to the right-most 8 bits of the accumulator and resets the keyboard flag. A similar wait loop is used to display a character.

0.4.5 Indirection

On the PDP-8, indirect addressing is needed to handle data structures like arrays and strings and to extend addressing beyond the two page range of zero and current page direct addressing. This is especially needed for calling off page subroutines like I/O routines.

Working with arrays requires run-time calculation of effective addresses. We introduce this notion by self-modifying code to access the elements of an array. (The PDP-8's ISZ instruction is easily used to modify the operand field of another instruction.) This leads to treating the address of an array as a variable and using indirect addressing to access array components.

The convention on the PDP-8 is to store the address of the string/array along with the string/array values and using the former to initialize a pointer variable for the address. For example the null-terminated string 'Hello World" (below) begins at address STR+1 while STR stores its own address.

```
STR, .           ; STR stores its own address
   'Hello World'; 0 ; string
PTR, 0
```

To display the string, the contents of STR is copied to PTR, PTR is incremented, then indirect addressing is used to load the accumulator with the character to be printed. An indirect call to the Type routine displays the character in the accumulator. The code loops until a null character is detected.

```
      tad STR       ; Put address of output
      dca PTR       ; string in pointer
Loop: isz PTR ; increment PTR
      tad i PTR     ; Get next character
      sna           ; Skip if not null
      jmp End       ; else end
      jms i Type    ; Call Type routine
```

```
jmp Loop        ; Loop!
end
```

0.5 PDP-11 Minicomputer

In the 16-bit DEC PDP-11 minicomputer family (1970), a single task addressed only 64 KB, or in later models (1973), 64 KB of instructions plus 64 KB of data.

- Registers: R0:R5, SP=R6, PC=R7,(all 16 bits), Status Flags: I, V, N, Z, C
- Address: 16-bits (64K), (32-k words)
- additional instructions, like MUL, DIV, WAIT, RESET, and many more powerful instructions
- later versions supported Virtual memory.
- 8085 instruction set is subset of PDP11 (DEC machine)
- Addressing modes: register, auto-increment, auto-decrement, index, indirect, immediate, absolute, relative

The figure 7 shows the register-level architecture of PDP-11 minicomputer.

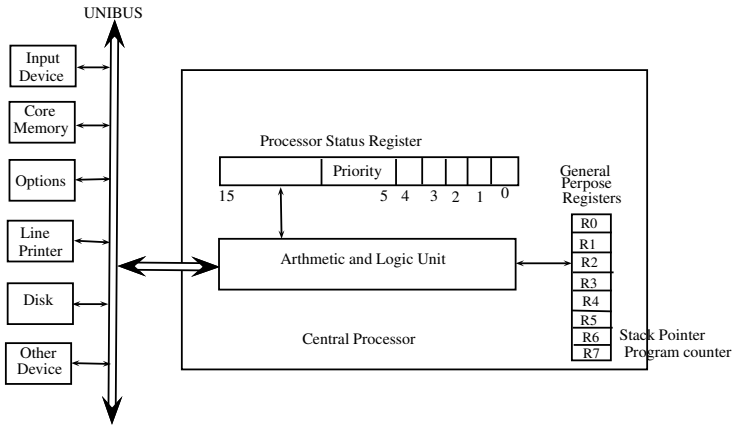


Figure 7: PDP-11 architecture.

0.6 IBM 360/370 System

The IBM System/360, first introduced in 1964, was the real beginning of modern computer architecture. Although computers in the System/360 "family" provided a different level of performance for a different price, all ran identical software. The System/360 originated the distinction between computer architecture-the abstract structure of a computer that a machine-language programmer needs to know to write programs-and the hardware implementation of that structure. Micro-programming was the primary technological innovation behind the marketing concept. Micro-programming relied on a small control memory and was an elegant way of building the processor control unit for a large instruction set.

Each word of control memory is called a *microinstruction*, and the contents are essentially an *interpreter*, programmed in microinstructions. Minicomputer manufacturers tend to follow the lead of mainframe manufacturers, especially when the mainframe manufacturer is IBM, and so microprogramming caught on quickly.

With the continuing growth of semiconductor memory, a much richer and more complicated instruction set could be implemented. The architecture research community argued for richer instruction sets. Let us review some of the arguments they advanced at that time:

1. Richer instruction sets would simplify compilers. As the story was told, compilers were very hard to build, and compilers for machines with registers were the hardest of all. Compilers for architectures with execution models based either on *stacks* or *memory-to-memory* operations were much simpler and more reliable, then register-based architecture.

2. Richer instruction sets would alleviate the software crisis. At a time when software costs were rising as fast as hardware costs were dropping, it seemed appropriate to move as much function to the hard- ware as possible. The idea was to create machine instructions that resembled programming language statements, so as to close the "semantic gap" between programming languages and machine languages.

In the 1970s led to certain design principles that guided computer architecture:

1. The memory technology used for microprograms was growing rapidly, so large microprograms would add little or nothing to the cost of

the machine. Since microinstructions were much faster than normal machine instructions, moving software functions to microcode made for faster computers and more reliable functions.

2. Since execution speed was inversely proportional to program size, architectural techniques that led to smaller programs also led to faster computers.

3. Registers were old fashioned and made it was hard to build compilers; stacks or memory-to-memory architectures were superior execution models.

The IBM 360/370 machine has 16 General Purpose Register:$R_0 - R_{15}$, and memory address of 20-bits.

Register	Add R3, R4
Immediate	Add R4, #3
Displacement	Add R4, 100(R1)
Register Indirect	Add R4, (R1)
Indexed	Add R3, (R1+R2); R1 base, R2 Index
Direct/Absolute	Add R1, (1001); [R1] ← [R1]+m[1001]
Memory Indirect	Add R1, @(R3); [R1]←[R1]+M[M[R3]]
Auto Increment	Add R1, (R2)+; [R1]←[R1]+M[R2]
	[R2]← [R2]+d; d size of elem.
Auto decrement	Add R1, -(R2);[R2]← [R2]-d; R[1]←[R1]+M[R2]

Add R3, R4: Adds the contents of register R4 into R3. The addressing mode is called *register* or *register direct* addressing mode.

Add R4, #3: Adds the immediate operand #3 into the register R4.

Add R4, 100(R1): The data address is 100 plus contents of register R4. If number of consecutive data values exist at location 100 onwards, the R1 is initialized with 0. Incrementing R1 allows to access subsequent locations using the same instruction.

Add R4, (R1): This is same case as earlier, with *displacement* value 0.

Add R3, (R1+R2): The R1 as *base* means it is relative position program start with respect to zero location. When run, if the program is loaded at 1000 address, the base address is 1000. The index register R2 is initialized to zero at start.

Add R1, (1001): In absolute/direct addressing, the given value is used as address of the data.

Add R1, @(R3): The address of data is available at the location pointed by R3.

Add R1, (R2)+: The R2 is incremented every time after the instruction is executed to access the next data location, pointed by R2.

0.7 Important Features of Architectures

Most computers have an architecture designed in the early 1960's. They have remained substantially unchanged for a decade in the name of compatibility in spite of their obstacles to generating efficient code from high level languages. The architecture had two explicit goals: 1. minimizing program size, and 2. providing a target language to which compilation is straightforward. We choose to minimize program size rather than maximize execution speed for several reasons.

First, execution speed depends not only on the raw clock rate, but also on the characteristics of the underlying microinstruction set. Given a high level language benchmark program and any two proposed instruction sets, it is possible to determine unambiguously which object program is smaller, but *not which is faster!* By hypothesizing a faster clock or better micro-architecture either machine can be sped up. In other words, minimizing size is a more clearly defined goal than maximizing speed.

Second, program size and speed are highly intertwined. All other factors being equal, a shorter program will execute faster than a longer one, since fewer bits need be processed. If the memory bandwidth is N words/sec and the mean instruction size is L words, the maximum instruction execution rate will be N/L instructions/sec. The smaller is L, the faster the machine can be.

Furthermore, on a machine with virtual memory, reducing program size reduces the number of page faults, which, in turn, reduces the time required to process the page faults, thereby speeding up execution.

Third, on large computers with sophisticated multiprogramming systems, a decrease in program size means an increase in the degree of multiprogramming, hence a higher CPU utilization, as well as less swapping.

Fourth, the small amount of memory available on minicomputers is often a serious limitation. Making the program fit into the memory may take precedence over all other considerations.

Fifth, on mini and micro computer systems, the cost of memory is much larger than the CPU cost. Reducing memory requirements has a much greater effect on total system cost than reducing execution time.

On the other side, the fact that few compilers for new generation computers could produce code that even came close to what a skilled assembly language programmer can generate, argues strongly for redesigning machine architectures so that compilers can do their job better. It is for this reason that we consider a *stack machine*, since generating efficient reverse Polish is simpler than generating efficient code for a register oriented machine. We assume the presence of a cache to eliminate the need for memory cycles when referencing the stack.

0.8 RISC and CISC Computing

Considering that R is clock rate, N as number of machine instructions in a given program, and S is basic steps to execute a machine instruction. Obviously, in a serial processing system, maximum one basic operation can be completed in one clock cycle. The time T required by this program shall be

$$T = \frac{N \times S}{R} \; secs. \tag{1}$$

There are two approaches for designing the instructions. In one, each instruction can be simple to execute, it will be faster (i.e., reducing S), but overall number of instructions N in the program shall be large in number. Thus, $N \times S$ may turn out to be not very different. This architecture is called RISC (Reduce Instruction Set Computers).

On the other hand if individual instruction is complex performing number of operations, that it, increasing S. This may reduce the total number of instructions N, in a program. Hence, the execution time, may not be very different. This architecture is called CISC (Complex Instruction Set Computers).

This is provided that, in the above two cases, the the clock-rate remains the same.

0.8.1 Origin of RISC

Several people, including those who had been working on microprogramming tools, began to rethink the architectural design principles of the 1970s. In trying to close the *semantic gap* (between high level language and assembly language), these principles had actually introduced a "performance gap." The attempt to bridge this gap with WCSs (writable control Stores) was unsuccessful, although the motivation for WCS-that

instructions should be no faster than microinstructions and that programmers should write simple operations that map directly onto microinstructions-was still valid. Furthermore, since caches had allowed "main" memory accesses at the same speed as control memory accesses, microprogramming no longer enjoyed a ten-to-one speed advantage.

A new computer design philosophy evolved: *Optimizing compilers* could be used to compile "normal" programming languages down to instructions that were as unencumbered (i.e., not slowed down) as microinstructions in a large virtual address space, and to make the instruction cycle time as fast as the technology would allow. These machines would have fewer instructions-a *reduced set*- and the remaining instructions would be simple and would generally execute in one cycle - *reduced instructions* - hence the name *reduced instruction set computers* (RISCs). RISCs inaugurated a new set of architectural design principles:

1. Functions should be kept simple unless there is a very good reason to do otherwise. A new operation that increases cycle time by 10 percent must reduce the number of cycles by at least 10 percent to be worth considering.

2. Microinstructions should not be faster than simple instructions. Since cache is built from the same technology as writable control store, a simple instruction should be executed at the same speed as a microinstruction.

3. Microcode was not found to be a magic! Moving software into microcode does not make it better, it just makes it harder to change. To paraphrase the Turing Machine argument, anything that can be done in a microcoded machine can be done in assembly language in a simple machine. The same hardware primitives assumed by the microinstructions must be available in assembly language. The run-time library of a RISC has all the characteristics of a function in microcode, except that it is easier to change.

4. Simple decoding and pipelined execution are more important than program size. Imagine a model in which the total work per instruction is broken into pieces, and different pieces for each instruction execute in parallel. At the peak rate a new instruction is started every cycle. This *Assembly-line* approach performs at the rate determined by the length of individual pieces rather than by the total length of all pieces. This kind of model gave rise to instruction formats that are simple to decode and to pipeline them.

5. Compiler technology should be used to simplify instructions rather than to generate complex instructions. RISC compilers try to remove as much work as possible at compile time so that simple instructions can be used. For example, RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used.

   Traditional compilers, on the other hand, tries to discover the ideal addressing mode and the shortest instruction format to add the operands in memory.

   In general, in the designers of RISC compilers prefer a register-to-register model of execution so that compilers can keep operands that will be reused in registers, rather than repeating a memory access on a calculation. They therefore use LOADS and STORES to access memory so that operands are not implicitly discarded after being fetched, as in the memory-to-memory architecture.

0.8.2 Common RISC Traits

The RISC machines had a great deal in common, irrespective of the project for which they were created.

1. *Operations are register-to-register, with only LOAD and STORE accessing memory.* Allowing compilers to reuse operands requires registers. When only LOAD and STORE instructions access memory, the instruction set, the processor, and the handling of page faults in a virtual memory environment are greatly simplified. Cycle time is shortened as well.

2. *The operations and addressing modes are reduced.* Operations between registers complete in one cycle, permitting a simpler, hardwired control for each RISC, instead of microcode. Multiple-cycle instructions such as floating-point arithmetic are either executed in software or in a special-purpose co-processor. (Without a co-processor, RISCs have mediocre floating-point performance.) Only two simple addressing modes, indexed and PC-relative, are provided. More complicated addressing modes can be synthesized from the simple ones.

3. Instruction formats are simple and do not cross word boundaries. This restriction allows RISCs to remove instruction decoding time from the critical execution path. The RISC register operands are always in the same place in the 32-bit word, so register access can

take place simultaneously with opcode decoding. This removes the instruction decoding stage from the pipe-lined execution, making it more effective by shortening the pipeline.

4. Single-sized instructions also simplify virtual memory, since they cannot be broken into parts that might wind up on different pages.

5. RISC branches avoid pipeline penalties. A branch instruction in a pipelined computer will normally delay the pipeline until the instruction at the branch address is fetched. Several pipelined machines have elaborate techniques for fetching the appropriate instruction after the branch, but these techniques are too complicated for RISCs.

   The generic RISC solution, commonly used in microinstruction sets, is to redefine jumps so that they do not take effect until after the following instruction; this is called the delayed branch. The delayed branch allows RISCs to always fetch the next instruction during the execution of the current instruction. The machine-language code is suitably arranged so that the desired results are obtained.

   Because RISCs are designed to be programmed in high-level languages, the programmer is not required to consider this issue; the "burden" is carried by the programmers of the compiler, the optimizer, and the debugger. The delayed branch also removes the branch bubbles normally associated with pipelined execution.

RISC optimizing compilers are able to successfully rearrange instructions to use the cycle after the delayed branch more than 90 percent of the time. It has been found that more than 20 percent of all instructions are executed in the delay after the branch.

Virtually all machines with variable-length instructions use a buffer to supply instructions to the CPU. These units blindly fill the prefetch buffer no matter what instruction is fetched and thus load the buffer with instructions after a branch, despite the fact that these instructions will eventually be discarded. Since studies show that one in four VAX instructions changes the program counter, such variable-length instruction machines really fetch about 20 percent more instruction words from memory than the architecture metrics would suggest. RISCs, on the other hand, nearly always execute something useful because the instruction is fetched after the branch.

0.9 RISC Variations

Each RISC machine provides its own particular variations on the common theme. This makes for some interesting differences.

0.9.1 Compiler Technology versus Register Windows

Both IBM and Stanford pushed the state of the art in compiler technology to maximize the use of registers. In Berkeley, the first step was to have enough registers to keep all the local scalar variables and all the parameters of the current procedure in registers.

Attention was directed to these variables because of two very opportune properties: Most procedures only have a few variables (approximately a half-dozen), and these are heavily used (responsible for one-half to two-thirds of all dynamically executed references to operands). Normally, it slows procedure calls when there are a great many registers. The solution was to have many sets, or windows, of registers, so that registers would not have to be saved on every procedure call and restored on every return.

A procedure call automatically switches the processor to use a fresh set of registers. Such buffering can only work if programs naturally behave in a way that matches the buffer. Caches work because programs do not normally wander randomly about the address space. Similarly, through experimentation, a locality of procedure nesting was found; programs rarely execute a long uninterrupted sequence of calls followed by a long uninterrupted sequence of returns.

To further improve performance, the Berkeley RISC machines have a unique way of speeding up procedure calls. Rather than copy the parameters from one window to another on each call, windows are overlapped so that some registers are simultaneously part of two windows. By putting parameters into the overlapping registers, operands are passed automatically.

0.9.2 Delayed Loads and Multiple Memory and Register Ports

Since it takes one cycle to calculate the address and one cycle to access memory, the straightforward way to implement LOADS and STORES is to take two cycles. This is what was done in the Berkeley RISC architecture. To reduce the costs of memory accesses, both the 801 and the MIPS provide "one-cycle" LOADS by following the style of the delayed branch. The first step is to have two ports to memory, one for instructions and

one for data, plus a second write port to the registers. Since the address must still be calculated in the first cycle and the operand must still be fetched during the second cycle, the data are not available until the third cycle.

Therefore, the instruction executed following the one-cycle LOAD must not rely on the operand coming from memory. The 801 and the MIPS solve this problem with a delayed load, which is analogous to the delayed branch described above. The two compilers are able to put an independent instruction in the extra slot about 90 percent of the time. Since the Berkeley RISC executes many fewer LOADS, it was decided to bypass the extra expense of an extra memory port and an extra register write port. Once again, depending on goals and implementation technology, either approach can be justified.

The table 6 shows the three different RISC architectures. None of these machines use microprogramming; all three use 32-bit instructions and follow the register-to-register execution model. The number of instructions in each of these machines is significantly lower than for those in most CISC.

Table 6: Primary Characteristics of Three RISC Machines

S.No.	Machine	IBM 801	RISC I	MIPS
1.	Year	1980	1982	1983
2.	Number of instructions	120	39	55
3.	Control memory size	0	0	0
4.	Instruction sizes (bits)	32	32	32
5.	Technology	ECLMSI	NMOS VLSI	NMOS VLSI
6.	Execution model	reg-reg	reg-reg	reg-reg

0.9.3 Pipelines

All RISCs use pipelined execution, but the length of the pipeline and the approach to removing pipeline bubbles vary. Since the peak pipelined execution rate is determined by the longest piece of the pipeline, the trick is to find a balance between the four parts of a RISC instruction's execution:

1. instruction fetch,

2. register read,
3. arithmetic/logic operation, and
4. register write.

Exercises

1. Assume an instruction set that uses a fixed 16-bit instruction length. Operand specifiers are 6 bits in length. There are k two-operand instructions and l zero operand instructions. What is the maximum number of one-operand instructions that can be supported?

2. A given processor has 32 registers, uses 16-bit immediate and has 142 instructions. In a given program,

   (i) 20 % of the instructions take 1 input register and have 1 output register.,

   (ii) 30 % have 2 input registers and 1 output register,

   (iii) 25 % have 1 input register, 1 output register and take an immediate input as well, and

   (iv) remaining 25 % have one immediate input and 1 output register.

   (a) For each of the 4 types of instructions, how many bits are required? Assume that it requires that all instructions be a multiple of 8 bits in length.

   Ans. Instruction's opcode length for 142 different instructions = $\lceil \lg_2(142) \rceil$ = 8.For 20%: to designate a registers it requires 5 bits, and next register 5 bits. So length = 8+5+5 = 18. Hence, 24 bits. Similarly it can be calculated for others.

   calculate the average length of program taking instructions of each type as 200, 300, 250, 250 in numbers. Say it is x.

   (b) How much less memory does the program take up if variable-length instruction set encoding is used as opposed to fixed-length encoding?

   Ans. We use two bits for selection of instruction type, out of the above 4 types.

   First 25% there are total 0.25*142 = 36 instructions. 6 bits are are used for 36 instructions, plus two bits for instruction type. That makes 8 bit for opcode. There is One input register

(5 bits), one output register (5 bits), plus 16 bit immediate operands, that makes 6 + 2 + 5 + 5 + 16 = 34 bits. So, let this be 40 bits (multiple of 8 bits).

On similar lines we can compute the variable length operand instructions.

Calculate the length of program taking instructions of each type as 200, 300, 250, 250 instructions. Say it is y.

The percent saving of space = (x -y)*100/y.

3. Compare the memory efficiency of the following instruction set architectures:

   - *Accumulator:* All operations occur between a single register and a memory location. There are two accumulators of which one is selected by the opcode;
   - *Memory-memory:* All instruction addresses reference only memory locations
   - *Stack:* All operations occur on top of the stack. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references.
   - *Load-store:* All operations occur in registers, and register-to register instructions have three register names per instruction.

   To measure memory efficiency, following are assumptions about these 4 instruction sets:

   - All instructions are an integral number of 8-bit in length;
   - The opcode is always 8 bits;
   - Memory accesses use direct address
   - The variables A, B, C, and D are initially in memory

   Invent your own assembly language mnemonics and for each of the above architecture and write the best equivalent assembly language code for the following high level language code:
   A = B + C;
   B = A + C;
   D = A - B;

Following can be a program for *accumulator* type instructions: Let the accumulators are identified as A and B. $ADDA\ x$, $ADDB\ x$ adds the memory contents x to the accumulator A or B. Similarly there are subtract instructions. There are $LOADA\ x$, $STORA\ x$ and $LOADB\ x$, $STORB\ x$ instructions to load and store in x from and to x, with respect to the accumulators A and B. The symbols a, b, c, d are memory addresses.

```
LOADA b
ADDA  c
STORA a
LOADB a
ADDB c
STORB b
LOADA a
SUBA b
```

Note that there are no register to register instructions.

The memory-memory and and load-store are extensions of accumulator based instructions.

The stack machine instruction set is as follows: Operations like add, sub are done with implicit operands, which are always at top of the stack. The result of operation is pushed back on the stack. The other operands are accessible as memory locations. $LOAD\ x$ and $STOR\ x$ pushes a value from memory address x on top of stack, and pops and stores this at location x. In addition, there are instructions $SWAP$, $DROP$, DUP to swap top mots stack operands, deletes the top of stack value, and duplicates the top of stack value. Following is the program.

```
LOAD b
LOAD c
ADD
STOR a
LOAD c
ADD
STOR b
LOAD c
LOAD a
SUB
STOR D
```


4. Assume the given code sequence is from a small, embedded computer application, such as a microwave oven controller that uses 16-bit memory addresses and data operands.

   A = B + C;
   B = A + C;
   D = A - B;

   If a load-store architecture is used, and assume that it has 16 general-purpose registers. Answer the following questions:

   (a) How many instruction bytes are fetched?

   (b) How many bytes of data are transferred from/to memory?

   (c) Which architecture is the most efficient as measures in code size?

   (d) Which architecture is most efficient as measured by total memory traffic (code + data)

5. Specify the register contents and the flag status as the following 8085 instructions are executed:

```
XRA A
MVI B, FFH
INR B
DCR A
ADD B
SUI 86H
ANA C
RST1
```

6. A system is designed to monitor the temperature of a furnace. Temperature readings are recorded in 16 bits and stored in memory locations starting at 7060H. The high-order byte is stored first and the low-order byte is stored in the next consecutive memory location. However, the low-order byte of all the temperature readings is constant. Write 8085 assembly language program (ALP) to transfer the high-order readings to consecutive memory locations starting at 7080H and discard the low-order bytes.Temperature Readings (H): 6745, 8745, 1F45, 3045, 8045, 7F45.

7. First set of data is stored from memory locations starting from 6155H to 6165H. Second set of data is stored from 6255H to 6265H. Write 8085 ALP to interchange the contents of memory locations

with each other.

(6155H) < . . . > (6255H)

(6165H) < . . . > (6265H)

8. Download Intel 8085 simulator (gnusim8085), using command:
   $ sudo apt-get install gnusim8085 <enter>,
   and run the various small simulation programs on this< perticulalry those discussed in the class.

9. List the assembly language program generated by a compiler from the following Fortran program. Assume integer values of one byte size.

```
SUM = 0
SUM = SUM + A + B
DIF = DIF - C
SUM = SUM + DIF
```

10. List the assembly language program generated by the compiler for the following Fortran IF statement:

```
IF(A - B) 10, 20, 30
```

    The program branches to statement 10 if A-B <0; to statement 20 if A - B = 0; and to statement 30 if A - B > 0.

11. Write a program to multiply two unsigned numbers.

12. The ALU, the bus and all the registers in the data path are of identical size. All operations including incrementation of the PC and the GPRs are to be carried out in the ALU. Two clock cycles are needed for memory read operation : one for loading address in the MAR and the next one for loading data from the memory bus into the MDR. (see figure 8).

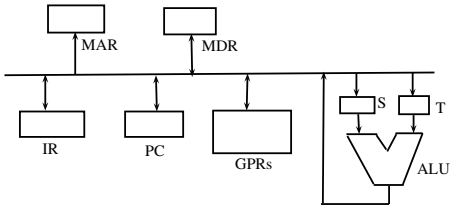


Figure 8: ALU and registers.

(a) The instruction "add R0, R1" has the register transfer interpretation $R0 \leftarrow R0 + R1$. The minimum number of clock cycles needed for execution cycle of this instruction is:
(a) 2 (b) 3 (c) 4 (d) 5

(b) The instruction "call Rn, sub" is a two word instruction. Assuming that PC is incremented during the fetch cycle of the first word of the instruction, its register transfer interpretation is

$$Rn <= PC + 1;$$
$$PC <= M[PC];$$

The minimum number of CPU clock cycles needed during the execution cycle of this instruction is:
(a) 2 (b) 3 (c) 4 (d) 5

13. Which of the following addressing modes permits relocation without without any change of whatsoever in the code?

(a) Indirect addressing (b) Indexed addressing
(c) Base register addressing (d) PC relative addressing

14. A CPU has 24-bit instructions. A program starts at address 300 (in decimal). Which one of the following is a legal program counter (all values in decimal)?
(A) 400 (B) 500 (C) 600 (D) 700

15. Suppose a stack representation supports, in addition to PUSH and POP, an operation REVERSE, which reverses the order of the elements on the stack.

(a) To implement a queue using the above stack implementation, show how to implement ENQUEUE using a single operation and DEQUEUE using a sequence of three operations.

(b) The following postfix expression, containing single digit operands and arithmetic operators + and *, is evaluated using a stack.

$$52*34*52**+$$

show the contents of stack:

i after evaluation of 5 2 * 3 4 +

ii after evaluation of 5 2 * 3 4 + 5 2

iii at the end of evaluation.

16. Given that following is an 8085 program sequence:

```
        LHLD 5000
        MVI  B, 5
GET:    IN   20
        MOV  M, A
        INX  H
        DCR  B
        JNZ  GET
```

(a) Identify the function performed by this program.

(b) List the role of registers used and the address referred to by it.

17. Consider the following program segment. Here R1, R2 and R3 are the general purpose registers. (for next three questions):

```
      Instruction          Operation       Instruction size
                                           (no.of words)
      MOV R1, (3000)       R1 <- m[3000]          2
LOOP: MOV R2, (R3)         R2 <- M[R3]            1
      ADD R2, R1           R2 <-R1 + R2           1
      MOV (R3), R2         M[R3] <- R2            1
      INC R3               R3 <- R3 + 1           1
      DEC R1               R1 <- R1 - 1           1
      BNZ LOOP             Branch on not zero     2
      HLT                  Stop                   1
```

Assume that the content of memory location 3000 is 10 and the content of the register R3 is 2000. The content of each of the memory locations from 2000 to 2010 is 100. The program is loaded from the memory location 1000. All the numbers are in decimal.

(a) Assume that the memory is word addressable. The number of memory references for accessing the data in executing the program completely is:

(A) 10 (B) 11 (C) 20 (D) 21

(b) Assume that the memory is word addressable. After the execution of this program, the content of memory location 2010 is:

(A) 100 (B) 101 (C) 102 (D) 110

(c) Assume that the memory is byte addressable and the word size is 32 bits. If an interrupt occurs during the execution of the instruction "INC R3", what return address will be pushed on to the stack?

(A) 1005 (B) 1020 (C) 1024 (D) 1040

18. The contents of A register after execution of following 8085 microprocessor program is:

```
MVI A, 55H
MVI C, 25H
ADD C
DAA
```

(a) 7AH (b) 80H (c) 50H (d) 22H

Bibliography

[1] John P. Hayes, "Computer Architecture and Organization", 2nd Edition, McGraw-Hill, 1988.

[2] William Stalling, "Computer Organization and Architecture", 8th Edition, Pearson, 2010.

[3] M. Morris Mano, "Computer System Architecture", 3rd Edition, Pearson Education, 2006.

[4] Carl Hamacher, Zvono Vranesic, and Safwat Zaky, "Computer Organization", , 5th edition, McGrawhill Education, 2011. (chapter 7)

[5] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-823-computer-system-architecture-fall-2005/lecture-notes/

[6] Andrew S. Tanenbaum, "Implications of Structured Programming for Machine Architecture", *Communications of the ACM*, March 1978, Volume 21, Number 3, pp. 237-246.

[7] David A. Patterson, "Reduced Instruction Set Computers", *Communications of the ACM*, January 1985, Volume 28, Number 1, pp. 8-21.