Computer Organisation And Logic

Computer organisation
↳ structure and behavior of a computer system

Assembly Language
↳ ADD, MOV, SUB, CALL → alpha numeric symbols
↳ low level programming language
↳ depends on machine code instructions
↳ each assembly language has a one to one relationship with machine language

- assembly language is **not portable** as it is designed for a specific processor family

Machine Language	Assembly Language	High-Level Language
Collection of binary numbers	Symbolic form of machine language (I.e. symbolic names are used to represent operations, registers & memory locations	Combines algebraic expressions & symbols taken from English language (ex. C++, java, Pascal, FORTRAN, ...etc)

Ex.
10100001 00000000 00000000
00000101 00000100 00000000
10100011 00000000 00000000
	Ex.
	`MOV AX, A`
	`ADD AX, 4`
	`MOV A, AX`
		Ex.
		`A = A + 4`

operation, destination operand, source operand

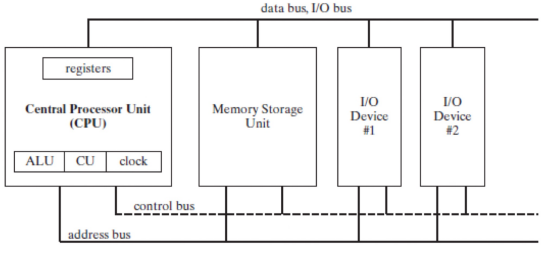
High level languages
↳ user friendly → closer to human language
↳ C++, Java, Python
↳ one to many relationship with machine and assembly

registers
↳ storage locations holding results of operations

Listing file
↳ programs source code
↳ line numbers
↳ numeric address of each instruction
↳ machine code bytes of each instruction
↳ symbol table

virtual machine: is a software program that emulates the functions of some other physical or virtual computer
↳ translation: completely translated then runs
↳ interpretation: interprets and executes instructions one by one

MICROCOMPUTER



Memory storage unit: instructions and data are held while program runs

Storage unit:

- receives data requests from CPU
- transfers data from RAM to CPU
- transfers from CPU to memory

↓

all processing occurs in CPU so programs in memory must be copied in CPU to be executed

CPU: calculations and logic operations are performed

↳ contains registers

↳ high frequency clock

↳ control unit

↳ arithmetic unit

faster than RAM as made from static RAM

cache: stores most recently used instructions and data

cache hit: processor finds data

cache miss: processor doesn't find data

Buses

bus: group of parallel wires that transfer data

↳ **data bus:** transfers instructions and data b/w CPU and memory

↳ **I/O bus:** transfers data b/w CPU and I/O devices

↳ **control bus:** uses binary signals to synchronize actions

↳ **address bus:** holds addres of instructions and data

Fetch execute cycle

1. CPU fetches instruction from instruction queue
2. CPU decodes instruction
3. If operands involved, CPU fetches operand
4. CPU executes instruction
5. CPU stores resul- in operand

Instruction Execution Cycle

Here are the steps to execute it:

1. First, the CPU **fetch the instruction** from the *instruction queue*
   - It then increments the instruction pointer
2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern
   - This bit pattern might reveal that the instruction has operands (input values)
3. If operands are involved, the CPU **fetches the operands** from registers and memory
   - Sometimes, this involves address calculations
4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step
   - It also updates a few status flags, such as Zero, Carry, and Overflow
5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its execution in the operand

Reading from Memory

As a rule, computers read memory much more slowly than they access internal registers.

Reading a single value from memory involves four separate steps:

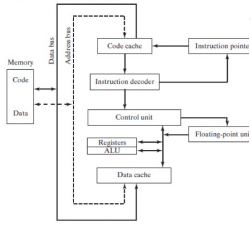
1. Place the address of the value you want to read on the address bus.
2. Assert (change the value of) the processor's RD *(read)* pin.
3. Wait few clock cycle for the memory chips to respond.
4. Copy the data from the data bus into the destination operand.

Each of these steps generally requires a single *clock cycle*,

Instruction Execution Cycle

- An *operand* is a value that is either an input or an output to an operation
- For example, the expression Z = X + Y has two input operands (X and Y) and a single output operand (Z)
- In order to read program instructions from memory, an address is placed on the address bus
- The memory controller places the requested code on the data bus
  - Making the code available inside the code cache

Figure 2–2 Simplified CPU block diagram.



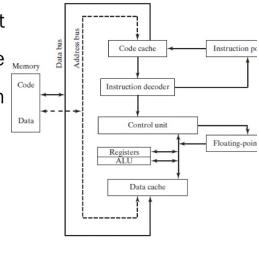
Cache (1 of 2)

- CPU designers figured out that computer memory creates a speed bottleneck
  - because most programs have to access variables
- To reduce the amount of time spent in reading and writing memory
  - the most recently used instructions and data are stored in high-speed memory called *cache*
- The idea is that a program is more likely want to access the same memory and instructions repeatedly
  - so cache keeps these values where they can be accessed quickly

Instruction Execution Cycle

- The instruction pointer's value determines which instruction will be executed next.
- The instruction is analyzed by the **instruction decoder**
  - Causing appropriate digital signals to be sent to the Control Unit
  - Control Unit coordinates with the ALU and floating-point unit.
- **Control bus** carries signals that use the system clock to coordinate the transfer of data between different CPU components.

Figure 2–2 Simplified CPU block diagram.

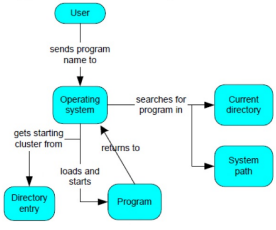


Cache (2 of 2)

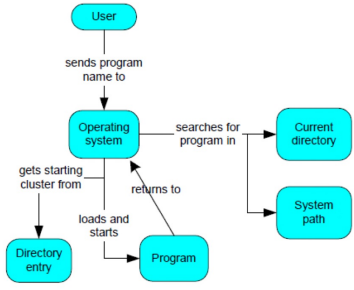
- When the CPU begins to execute a program, it loads the next thousand instructions (for example) into cache
  - The assumption is that these instructions will be needed in the near future.
- If it happens to be a loop in that block of code, the same instructions will be in cache
- When the processor is able to find its data in cache memory, we call that a *cache hit*
- On the other hand, if the CPU tries to find something in cache and it's not there, we call that a *cache miss*

Loading and Executing a Program (1 of 3)

- The operating system (OS) searches for the program's filename in the current disk directory
  - If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename
  - If the OS fails to find the program filename, it issues an error message



How a Program Runs



Loading and Executing a Program (2 of 3)

- If the program file is found, the OS retrieves basic information about the program's file from the disk directory
  - including the file size and
  - its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory
  - It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table*).
  - The OS also adjust the values of pointers within the program so they contain addresses of program data.

Loading and Executing a Program (3 of 3)

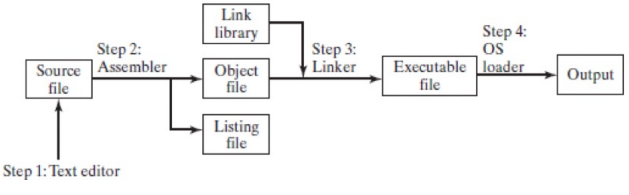
- The OS begins execution of the program's first machine instruction (its entry point).
- As soon as the program begins running, it is called a *process*.
- The OS assigns the process an identification number (*process ID*), which is used to keep track of it while running.
- It is the OS's job to track the execution of the process and to respond to requests for system resources.
  - Examples of resources are memory, disk files, and input-output devices.
- When the process ends, it is removed from memory.

What Are Assemblers and Linkers?

- **Assembler** is a utility program that converts source code programs from assembly language into an object file, a machine language translation of the program.
  - Optionally a Listing file is also produced.
  - We'll use MASM as our assembler.
- The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library.
  - The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*.
  - Microsoft 16-bit linker is LINK.EXE and 32-bit is Linker LINK32.EXE.
- **OS Loader:** A program that loads executable files into memory, and branches the CPU to the program's starting address, (may initialize some registers (e.g. IP)) and the program begins to execute.
- **Debugger** is a utility program, that lets you step through a program while it's running and examine registers and memory

FIGURE 3–7 Assemble-Link-Execute cycle.

MASM provides CodeView, TASM provides Turbo Debugger and msdev.exe for 32-bit Window console programs.



Real addres mode

Linear address = segment × 10 (hex) + offset

AIFD 04CO

↳ AIFD × 10 = AIFDO

↳ AIFDO
+ 004CO
A23CO

F+14=19

16 | 19
1 → 3
carry, remainder

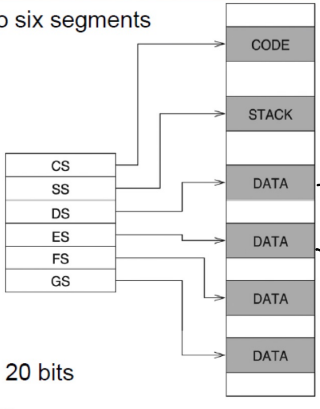
2. 'OA150
CD9D
16EEO

10+12=22

16 | 22
1 → 6

Real Address Mode (1)

- A program can access up to six segments at any time
  - Code segment
  - Stack segment
  - Data segment
  - Extra segments (up to 3)
- Each segment is 64 KB
- Logical address
  - Segment = 16 bits
  - Offset = 16 bits
- Linear (physical) address = 20 bits



→ holds base address of all executable instructions

→ holds base addres for stack

→ holds base address for variables. Stores data for the program

→ is an extra data segment

Slide Credit: Dr. Aiman El-Maleh & Dr. Kip Irvine

Registers

32-bit General-Purpose Registers

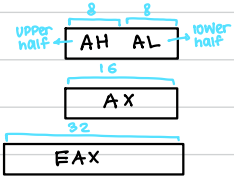
- EAX — accumulator (arithmetic operations)
- EBX — Base (holds address)
- ECX — counter for looping
- EDX — multiply/divide operations
- EBP — referencing access stack segment
- ESP — extended stack pointer register access stack segment
- ESI — extended source index point memory locations
- EDI — extended destination index point memory locations

EAX, EBX, ECX, EDX: data registers

EBP, ESP, ESI, EDI: address registers

↳ arithmetic and data mo

↳ can be addressed as



32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

remaining GPR

32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

16 bit Segment Registers

CS	ES
SS	FS
DS	GS

Index registers

↳ contain offset for data and instructions

↳ distance from base address of segment

Instruction Pointer register

↳ contains address of next instruction to be executed

EFLAGS Register

↳ binary bits controlling CPU operations

↳ flag set = 1

↳ flag clear/reset = 0

FLAGS

control

↳ how instructions are carried out

- **Direction flag (DF)** ↳ direction of block ↳ 1 = up ↳ 0 = down
- **Interrupt flag (IF)** ↳ determines if interrupts can occur
- **Trap flag (TF)** ↳ determines whether CPU is halted after every instruction ↳ used for debugging

status

- **carry flag (CF)** ↳ set when unsigned arithmetic result too large ↳ 1 = carry ↳ 0 = no carry
- **overflow (OF)** - Two positive operands are added and their sum is negative - Two negative operands are added and their sum is positive ↳ set when signed arithmetic result too large ↳ 1 = overflow 0 = no overflow
- **Sign flag (SF)** ↳ set when arithmetic result generates -ve value ↳ 1 = -ve ↳ 0 = +ve
- **zero flag (ZF)** ↳ set when arithmetic operation generates 0 ↳ 1 = zero ↳ 0 = not zero
- **Auxiliary carry** ↳ set when carry/borrow from bit 3 to 4 ↳ 1 = carry ↳ 0 = no carry
- **Parity** ↳ set if LSB contains even number of 1 bits

```
mov cx,1
sub cx,1                ; CX = 0, ZF = 1

mov ax,0FFFFh
inc ax                  ; AX = 0, ZF = 1
inc ax                  ; AX = 1, ZF = 0

mov cx,0
sub cx,1                ; CX = -1, SF = 1
add cx,2                ; CX = 1, SF = 0

mov al,0FFh
add al,1            ; AL = 00, CF = 1
```

```
mov al,1
sub al,2            ; AL = FFh, CF = 1

mov eax,4
sub eax,5           ; EAX = -1, SF = 1

mov al,127
add al,1                    ; OF = 1

mov al,-128
sub al,1                    ; OF = 1

mov al,-128 ; AL = 10000000b
neg al      ; AL = 10000000b, OF = 1
```

Zero Flag (ZF)

Whenever the destination operand equals Zero, the Zero flag is set.

```
mov cx,1
sub cx,1        ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax          ; AX = 0, ZF = 1
inc ax          ; AX = 1, ZF = 0
```

A flag is set when it equals 1.

A flag is clear when it equals 0.

Sign Flag (SF)

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
mov cx,0
sub cx,1            ; CX = -1, SF = 1
add cx,2            ; CX = 1, SF = 0
```

The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1            ; AL=11111111b, SF=1
add al,2            ; AL=00000001b, SF=0
```

Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al,+127
add al,1          ; OF = 1,   AL = ??

; Example 2
mov al,7Fh        ; OF = 1,   AL = 80h
add al,1
```

A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

```
What will be the values of OF flag?
   mov al,80h
   add al,92h        ; OF =

   mov al,-2
   add al,+127       ; OF =
```

31

Overflow/Carry Flags: Hardware Viewpoint

- How the **ADD** instruction modifies OF and CF:
  - CF = (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)
- How the **SUB** instruction modifies OF and CF:
  - NEG the source and ADD it to the destination
  - CF = INVERT (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)

Parity (PF) flag

- PF is set when LSB of the destination has an even number of 1 bits.

```
mov al, 10001100b
add al, 00000010b; AL=10001110, PF=1
sub al, 10000000b; AL=00001110, PF=0
```

Auxiliary Carry (AC) flag

- AC indicates a carry or borrow of bit 3 in the destination operand.
- It is primarily used in binary coded decimal (BCD) arithmetic.

```
mov al, oFh
add al, 1       ; AC = 1
```

```
mov ax,00FFh
add ax,1        ; AX= 0100h  SF= 0 ZF= 0 CF= 0
sub ax,1        ; AX= 00FFh  SF= 0 ZF= 0 CF= 0
add al,1        ; AL= 00h    SF= 0 ZF= 1 CF= 1
mov bh,6Ch
add bh,95h      ; BH= 01h    SF= 0 ZF= 0 CF= 1

mov al,2
sub al,3        ; AL= FFh    SF= 1 ZF= 0 CF= 1
```

Radix

13 decimal
13d decimal
1001b binary
A5h hexadecimal
12q octal
12o octal

Directives

commands that are recognised and acted upon by the compiler

```
myVar  DWORD 26          ; DWORD directive
mov    eax,myVar         ; MOV instruction
```

(DWORD: DIRECTIVE; mov: INSTRUCTION)

Defining Segments

- .data → identifies variables
- .code → identifies executable instructions
- .stack → identifies runtime stack

( )	1
+ , −	2
* , /	3
MOD	4
+ ,−	5

-5 ↖ (+ , −)
6-4 ↙ (+ ,−)

= directive

TITLE directive marks entire line as comment

```
TITLE Add and Subtract                (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc

.code → marks begining of code
main PROC → identifies begining of procedure

    mov   eax,10000h          ; EAX = 10000h
    add   eax,40000h          ; EAX = 50000h
    sub   eax,20000h          ; EAX = 30000h
    call  DumpRegs            ; display registers
                  ↳ displays value of registers
    exit
main ENDP  end directives
END main
```

also used for comments

copies necessary definitons

Real number constants

↳ [sign] integer . [integer] [exponent]
↳ atleast one digit and decimal point

2.
+3.0
-44.2E+05
26.E5

character constants

↳ 'A'
↳ "c"

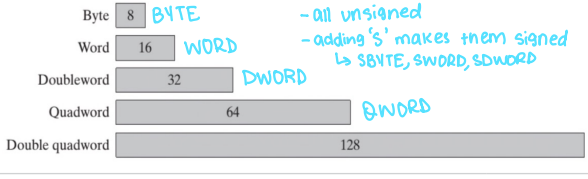
0 → 255 unsigned

-128 → 127 signed

String constants

↳ "hello"
↳ 'whats up'

```
value1 BYTE 'A'     ; character constant
value2 BYTE 0       ; smallest unsigned byte
value3 BYTE 255     ; largest unsigned byte
value4 SBYTE -128   ; smallest signed byte
value5 SBYTE +127   ; largest signed byte
value6 BYTE ?       ; uninitialized byte
```



- all unsigned
- adding 'S' makes them signed
  ↳ SBYTE, SWORD, SDWORD

copy small value to big

```
.data
count WORD 1
```
count: variable; WORD: 16 bit; 1: value

```
.code
mov ecx,0
mov cx,count
```
mov ecx,0 → equate to 0 → for signed equate to FFFFFFFFh

cx: 16 bit

MOVZX

↳ used with unsigned integers
↳ extends value to 16 or 32 bits

```
.data
val BYTE 10001111b
```
8 bit

```
.code
movzx ax,val
```
AX = 0000000010001111b

16 bit

MOVSX

↳ used with signed integers
↳ extends to 16 or 32 bits

```
.data
val BYTE 10001111b
```
16 unsigned

```
.code
movsx ax,val
```
AX = 1111111110001111b

16 bit

add : addition
sub : subtraction
inc : increment
dec : decrement
neg : reverses the sign by 2's complement
PTR : overide declared size of an operand
TYPE : returns size in bytes
LENGTHOF : counts no. of elements in an array
SIZEOF : returns LENGTHOF x TYPE

indirect operators/direct-offset

```
.data
arrB BYTE 10h,20h,30h
.code
mov esi,OFFSET arrB
```
41; address (pointer); OFFSET arrB →

arrB	address	esi
10	41	41
20	42	42
30	43	43

```
mov al,[esi]
```
41; AL = 10h

```
inc esi
```
42

```
mov al,[esi]
```
42; AL = 20h

Scale factors in Indexted Operands

variable [reg32 * TYPE variable]
arrB [esi * TYPE arrayB]

will increment according to type
e.g BYTE inc 1
WORD inc 2

DUP operator

↳ allocates storage for multiple data items

- BYTE 20 DUP(0) all 20 bytes equal to 0

BYTE 20 DUP(?) all 20 bytes uninitialized

HLL compilers translate mam expressions into binary

MOV INSTRUCTION

- Operands of same size
- Both can't be memory operators } same for ADD, SUB

```
.data
var1 BYTE 10h
.code
mov al,var1
mov al,[var1]
```

alternate format

```
.data
    count BYTE 100
    wVal  WORD 2
    aVal  WORD 3
.code
    mov bl,count
    mov ax,wVal
    mov count,al
    mov al,wVal     ; error
    mov ax,count    ; error
    mov eax,count   ; error
```

mov aVal, wVal ;error

XCHG OPERATOR

- one operands needs to be a register

size same

```
.data
    var1 WORD 1000h
    var2 WORD 2000h

.code
    xchg ax,bx              ; exchange 16-bit regs
    xchg ah,al              ; exchange 8-bit regs
    xchg var1,bx            ; exchange mem, reg
    xchg eax,ebx            ; exchange 32-bit regs

    xchg var1,var2          ; error: two memory operands
```

INC/DEC OPERATOR

- works only on memory or register

```
.data
    myWord  WORD 1000h
    myDword DWORD 10000000h
.code
    inc myWord              ; 1001h
    dec myWord              ; 1000h
    inc myDword             ; 10000001h

    mov ax,00FFh
    inc ax                  ; AX = 0100h
    mov ax,00FFh
    inc al                  ; AX = 0000h

  inc [esi]   ;  error:  operand  must  have  size
```

Examples that use multiple initializers:

```
list1 BYTE 10,20,30,40
list2 BYTE 10,20,30,40
      BYTE 50,60,70,80
      BYTE 81,82,83,84
list3 BYTE ?,32,41h,00100010b
list4 BYTE 0Ah,20h,'A',22h
```

PTR OPERATOR

- can be used to override size o f an operand

```
.data
        myDouble DWORD 12345678h
.code
        mov ax,myDouble          ;error - why?
        mov ax,WORD PTR myDouble ; loads 5678h
```

TYPE operator

- returns the size, in bytes

```
.data
      var1 BYTE ?
      var2 WORD ?
      var3 DWORD ?
      var4 QWORD ?
```

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

LENGTHOF operator

- counts the number of elements in an array

```
.data
      byte1 BYTE 10,20,30
      array1 WORD 30 DUP(?),0,0
      array2 WORD 5 DUP(3 DUP(?))
      array3 DWORD 1,2,3,4
      digitStr BYTE "12345678",0
.code
      mov ecx, LENGTHOF array1
```

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

SIZEOF operator

- returns LENGTHOF * TYPE

```
.data
    byte1  BYTE 10,20,30                ; 3
    array1 WORD 30 DUP(?),0,0           ; 64
    array2 WORD 5 DUP(3 DUP(?))         ; 30
    array3 DWORD 1,2,3,4                ; 16
    digitStr BYTE "12345678",0          ; 9
.code
      mov ecx, SIZEOF array1
```

INDIRECT. operand
•holds the address of a variable

```
.data
    val1 BYTE 10h,20h,30h
.code
    mov esi,OFFSET val1
    mov al,[esi]                ; dereference ESI (AL = 10h)

    inc esi
    mov al,[esi]                ; AL = 20h

    inc esi
    mov al,[esi]                ; AL = 30h
```

- Because the assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

INDEXTED operand
•adds a constant to a register to generate an effective address.

```
.data
    arrayW WORD 1000h,2000h,3000h

.code
    mov esi,0
    mov ax,[arrayW + esi]           ; AX = 1000h

    mov ax, arrayW[esi]             ; alternate format

    add esi,2
    add ax,[arrayW + esi]
```

JMP instruction
•causes an unconditional transfer to a destination

```
top:    INC AX
        MOV BX, AX
        jmp top
```

LOOP instruction
•loop destination must be within -128 to +127 bytes of the current location counter.
The execution of the LOOP instruction involves two steps:

1. First, it subtracts 1 from ECX.
2. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by *destination*. Otherwise, no jump takes place, and control passes to the instruction following the loop.

```
        mov ax,0
        mov ecx,5

L1:     inc ax
        loop L1
        mov bx,ax
```

DIRECT OFFSET OPERANDS

```
.data
   arrayB BYTE 10h,20h,30h,40h

.code
   mov al,arrayB+1     ; AL = 20h
   mov al,[arrayB+1]   ; alternative notation
```

INDEX SCALING

```
.data
arrayB BYTE  0,1,2,3,4,5
arrayW WORD  0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5
.code
mov esi,4
mov al,arrayB[esi*TYPE arrayB]    ; 04
mov bx,arrayW[esi*TYPE arrayW]    ; 0004
mov edx,arrayD[esi*TYPE arrayD]   ; 00000004
```

NESTED LOOPS

```
.data
      count DWORD ?
.code
      mov ecx,100         ; set outer loop count
L1:
      mov count,ecx       ; save outer loop count
      mov ecx,20          ; set inner loop count
L2:
      .
      loop L2             ; repeat the inner loop
      mov ecx,count       ; restore outer loop count
      loop L1             ; repeat the outer loop
```

procedures / functions

Example 04: (Addition of Two Numbers)

```
INCLUDE Irvine32.inc
.data
var1 DWORD 5
var2 DWORD 6
.code
main PROC
call AddTwo
call writeint
call crlf
exit
main ENDP
AddTwo PROC
mov eax,var1
mov ebx,var2
add eax,var2
ret
AddTwo ENDP
END main
```

function name

return

NOT instruction

NOT *destination*

- Example:

```
mov al, 11110000b
not al
```

```
NOT   00111011
      11000100   inverted
```

NOT

X	¬X
F	T
T	F

AND instruction

AND *destination, source*

- Example:

```
mov al, 00111011b
and al, 00001111b
```

```
  00111011
AND 00001111
cleared  0000 1011  unchanged
      bit extraction
```

AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

49

OR instruction

OR *destination, source*

- Example:

```
mov dl, 00111011b
or  dl, 00001111b
```

```
  00111011
OR 00001111
unchanged  0011 1111  set
```

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

50

XOR instruction

XOR *destination, source*

- Example:

```
mov dl, 00111011b
xor dl, 00001111b
```

```
  00111011
XOR 00001111
unchanged  0011 0100  inverted
```

XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR is a useful way to invert the bits in an operand and data encryption

51

Convert to uppercase

```
mov al,'a'          ; AL = 01100001b
and al,11011111b    ; AL = 01000001b
```

TEST instruction

- Performs a nondestructive **AND** operation between each pair of matching bits in two operands
- No operands are modified, but the flags are affected.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al,00000011b
jnz  ValueFound
```

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al,00000011b
jz   ValueNotFound
```

Compare instructions

unsigned	ZF	CF
destination<source	0	1
destination>source	0	0
destination=source	1	0

signed	flags
destination<source	SF != OF
destination>source	SF == OF
destination=source	ZF=1

Setting and clearing individual flags

```
and al, 0          ; set Zero
or  al, 1          ; clear Zero
or  al, 80h        ; set Sign
and al, 7Fh        ; clear Sign
stc                ; set Carry
clc                ; clear Carry

mov al, 7Fh
inc al             ; set Overflow

or eax, 0          ; clear Overflow
```

61

1. **Clrscr**
   Clears the console window and locates the cursor at the above left corner.

2. **Crlf**
   Writes the end of line sequence to the console window.

3. **WriteBin**
   Writes an unsigned 32-bit integer to the console window in ASCII binary format.

4. **WriteChar** al
   Writes a single character to the console window.

5. **WriteDec** eax
   Writes an unsigned 32-bit integer to the console window in decimal format.

6. **WriteHex**
   Writes a 32-bit integer to the console window in hexadecimal format.

7. **WriteInt**
   Writes a signed 32-bit integer to the console window in decimal format.

8. **WriteString (EDX= OFFSET String)** edx
   Write a null-terminated string to the console window.

9. **ReadChar**
   Waits for single character to be typed at the keyboard and returns that character.

10. **ReadDec**
    Reads an unsigned 32-bit integer from the keyboard.

11. **ReadHex**
    Reads a 32-bit hexadecimal integers from the keyboard, terminated by the enter key.

12. **ReadInt** eax
    Reads a signed 32-bit integer from the keyboard, terminated by the enter key.

13. **ReadString (EDX=OFFSET, ECX=SIZEOF)** edx
    Reads a string from the keyboard, terminated by the enter key.

14. **Delay (EAX)**
    Pauses the program execution for a specified interval (in milliseconds).

15. **Randomize** call Random range
    Seeds the random number generator with a unique value.

16. **DumpRegs**
    Displays the EAX, EBX, ECX, EDX, ESI, EDI, ESP, EIP and EFLAG registers.

17. **DumpMem (ESI=Starting OFFSET, ECX=LengthOf, EBX=Type)**
    Writes the block of memory to the console window in hexadecimal.

18. **getDateTime**
    Gets the current date and time from system

19. **GetMaxXY (DX=col, AX=row)**
    Gets the number of columns and rows in the console window buffer.

20. **GetTextColor (Background= Upper AL, Foreground= Lower AL)**
    Returns the active foreground and background text colors in the console window.

21. **Gotoxy (DH=row , DL=col)**
    Locates the cursor at a specific row and column in the console window. By default X coordinate range is 0-79,and Y coordinate range is 0-24.

22. **MsgBox (EDX=OFFSET String, EBX= OFFSET Title)**
    Displays a pop-up message box.

23. **MsgBoxAsk (EDX=OFFSET String, EBX= OFFSET Title)**
    Displays a yes/no question in a pop-up message box.
    (EAX=6 YES, EAX=7 NO)

24. **SetTextColor (EAX= Foreground + (Background*16))**
    Sets the foreground and background colors of all subsequent text output to the console.

25. **WaitMsg**
    Display a message and wait for the Enter key to be pressed.

black = 0	red = 4	gray = 8	lightRed = 12
blue = 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

Some conditional jump instructions treat operands of the CMP(compare) instruction as signed numbers.

Mnemonic	Description
JE	Jump if equal
JG/JNLE	Jump if greater/Jump if not less than or equal
JL/JNGE	Jump if less/Jump if not geater
JGE/JNL	Jump if greater or equal/Jump if less
JLE/JNG	Jump if less or equal/Jump if not greater
JNE	Jump if not equal

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

MUL examples

1MUL → signed

multiplied by eax/ax/al

unsigned

100h * 2000h, using 16-bit operands:

```
.data
val1 WORD 2000h
val2 WORD 100h
.code
mov ax,val1
mul val2  ; DX:AX=00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

DIV examples

1DIV → signed

dividend eax/ax/ah

unsigned

Divide 8003h by 100h, using 16-bit operands:

```
mov dx,0          ; clear dividend, high
mov ax,8003h      ; dividend, low
mov cx,100h       ; divisor
div cx            ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

```
mov edx,0         ; clear dividend, high
mov eax,8003h     ; dividend, low
mov ecx,100h      ; divisor
div ecx           ; EAX=00000080h,EDX=3
```

121

CBW, CWD, CDQ instructions

- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
  - CBW (convert byte to word) extends AL into AH
  - CWD (convert word to doubleword) extends AX into DX
  - CDQ (convert doubleword to quadword) extends EAX into EDX
- For example:

```
mov eax,0FFFFFF9Bh      ; -101 (32 bits)
cdq          ; EDX:EAX = FFFFFFFFFFFFFF9Bh
             ; -101 (64 bits)
```

```
var3 = (var1 * -var2) / (var3 - ebx)

mov eax,var1
mov edx,var2
neg edx
mul edx          ; left side: edx:eax
mov ecx,var3
sub ecx,ebx
div ecx          ; eax = quotient
mov var3,eax
```

NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.data
valB BYTE -1
valW WORD +32767
.code
   mov al,valB       ; AL = -1
   neg al            ; AL = +1
   neg valW          ; valW = -32767
```

WHILE Loops

```
while( eax < ebx)
   eax = eax + 1;
```

```
_while:
    cmp eax,ebx     ; check loop condition
    jae _endwhile   ; false? exit loop
    inc eax         ; body of loop
    jmp _while      ; repeat the loop
_endwhile:
```


LOOPNZ ?
LOOPNE

Jumps based on unsigned comparisons

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

> ≧ < ≦

Table-driven selection

Step 1: create a table containing lookup values and procedure offsets:

```
.data
CaseTable BYTE 'A'          ; lookup value
          DWORD Process_A   ; address of procedure
          EntrySize = ($ - CaseTable)
          BYTE 'B'
          DWORD Process_B
          BYTE 'C'
          DWORD Process_C
          BYTE 'D'
          DWORD Process_D

NumberOfEntries = ($ - CaseTable) / EntrySize
```

Step 2: Use a loop to search the table. When a match is found, we call the procedure offset stored in the current table entry:

```
   mov ebx,OFFSET CaseTable ; point EBX to the table
   mov ecx,NumberOfEntries  ; loop counter

L1:cmp al,[ebx]             ; match found?
   jne L2                   ; no: continue
   call NEAR PTR [ebx + 1]  ; yes: call the procedure
   jmp L3                   ; and exit the loop
L2:add ebx,EntrySize        ; point to next entry
   loop L1                  ; repeat until ECX = 0

L3:
```

required for procedure pointers

SHL instruction

t left, add 0 in end

Shifting left 1 bit multiplies a number by 2

```
mov dl,5
shl dl,1
```

Before: 00000101 = 5
After: 00001010 = 10

Shifting left *n* bits multiplies the operand by 2^n

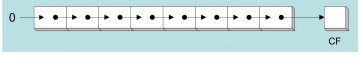
For example, $5 * 2^2 = 20$

```
mov dl,5
shl dl,2              ; DL = 20
```

LOGICAL : adds 0 in place

SHR instruction

Shift right, add 0 at beg

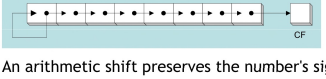


Shifting right n bits divides the operand by 2^n

```
mov dl,80
shr dl,1              ; DL = 40
shr dl,2              ; DL = 10
```

SAL and SAR instructions

- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.

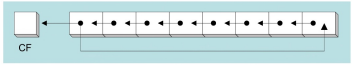


- An arithmetic shift preserves the number's sign.

```
mov dl,-80
sar dl,1              ; DL = -40
sar dl,2              ; DL = -10
```

ROL instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost




```
mov al,11110000b
rol al,1              ; AL = 11100001b

mov dl,3Fh
rol dl,4              ; DL = F3h
```

102

ROR instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost


```
mov al,11110000b
ror al,1              ; AL = 01111000b

mov dl,3Fh
ror dl,4              ; DL = F3h
```

103

RCR instruction

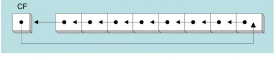
- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag


```
stc                   ; CF = 1
mov ah,10h            ; CF,AH = 00010000 1
rcr ah,1              ; CF,AH = 10001000 0
```

105

RCL instruction

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag



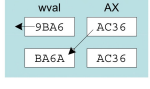

```
clc                   ; CF = 0
mov bl,88h            ; CF,BL = 0 10001000b
rcl bl,1              ; CF,BL = 1 00010000b
rcl bl,1              ; CF,BL = 0 00100001b
```

104

SHLD example

Shift **wval** 4 bits to the left and replace its lowest 4 bits with the high 4 bits of **AX**:

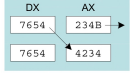
```
.data
wval WORD 9BA6h
.code
mov  ax,0AC36h
shld wval,ax,4
```



SHRD example

Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov  ax,234Bh
mov  dx,7654h
shrd ax,dx,4
```

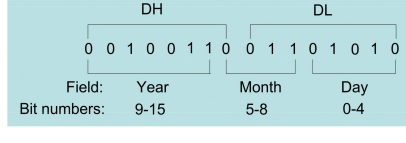


Isolating a bit string

```
mov al,dl         ; make a copy of DL
and al,00011111b  ; clear bits 5-7
mov day,al        ; save in day variable
```

```
mov ax,dx         ; make a copy of DX
shr ax,5          ; shift right 5 bits
and al,00001111b  ; clear bits 4-7
mov month,al      ; save in month variable
```

```
mov al,dh         ; make a copy of DX
shr al,1          ; shift right 1 bit
mov ah,0          ; clear AH to 0
add ax,1980       ; year is relative to 1980
mov year,ax       ; save in year
```



```
INCLUDE Irvine32.inc
.data
	array DWORD 60,4,17,45,7
	inde DWORD 0
	len DWORD 0
	sto DWORD ?
	esiadd DWORD ?
.code
swaps PROTO, a1:PTR DWORD,sizess:DWORD,index:DWORD
main PROC

	mov ecx,lengthof array
	mov len,ecx
	mov esi,offset array


l1:
	mov sto,ecx
	mov esi,offset array
	mov ecx,len
	mov eax,0
	max:

		cmp [esi],eax
		jge great
		jl quit

		great:
		mov eax,[esi]
		mov edx,ecx
		jmp quit

		quit:
		add esi,4
	loop max
	mov eax,len
	sub len,edx
	mov edx,len
	mov len,eax
	INVOKE swaps, offset array,len,edx
	dec len

	call display

	mov ecx,sto
loop l1



	exit
main ENDP
```

```
display PROC uses esi ecx ebx,
	mov esi , OFFSET array
	mov ecx, LENGTHOF array
	mov ebx, TYPE array
	call dumpmem
	ret 5
display ENDP

swaps PROC, a1:PTR DWORD ,sizess:DWORD,index:DWORD

	pushad
	mov edx, sizess
	dec edx
	mov edi,index
	mov esi,a1
	mov eax,[esi+edi*4]
	mov ebx,[esi+edx*4]
	xchg eax,ebx
	mov [esi+edi*4],eax
	mov [esi+edx*4],ebx
	popad
	ret
swaps ENDP

END main
```

copies from on string to another

MOVSB, MOVSW, MOVSD

copy from esi

pase to edi use rep

esi length in ecx

esi, edi automatically inc/dec

Direction Flag

DF = 0 , inc edi, esi

DF = 1 , dec edi, esi

cld → mov forward

std → mov backward

compare two strings

CMPSB, CMPSW, CMPSD

compare esi and edi

Scan for some char

SCASB, SCASW, SCASD

compare al/ax/eax to edi use Repne

↓ char ↓ string

Store ax into edi

STOSB, STOSW, STOSD

mov [edi], ax

use Rep

unless using loop

no need to inc

as done automatically

opposites

```
mov esi, offset array
mov ecx, length of array
mov ebx, TYPE array
call dumpmem
```

OR

```
push TYPE array
push length of array
push offset array
call dumpmem
```

LODSB, LODSW, LODSD

mov al/ax/eax, [esi]

```
Str_length PROTO,
  pString:PTR BYTE     ; pointer to string
```

returns length in eax

Example:

```
Str_copy PROTO,
  source:PTR BYTE, ; pointer to string
  target:PTR BYTE  ; pointer to string
```

```
Str_trim PROTO,
  pString:PTR BYTE,    ; points to string
  char:BYTE            ; char to remove
```

```
Str_ucase PROTO,
  pString:PTR BYTE     ; pointer to string
```

EQU
assigns name to given value

ESP

ESP

100
96 PUSH EAX
94 PUSH EBX
no change XCHG EAX, ECX
96 POP ECX
94 PUSH EAX
96 POP EBX

- Push esp decrements
- POP esp increments

enter → Push ebp
mov ebp,esp
:
leave → mov esp,ebp
Pop ebp

enter 8,0 → Push ebp
mov ebp,esp
sub esp,8
reserve space for local variables

pushad
↳ Pushes all registers
popad → pops all registers

Uses esi,ecx → Push esi
Push ecx
:
→ POP ecx
POP esi

ret n same thing add ESP,12
EIP = [ESP]
ESP: ESP+4+n
cleans up stack

replace with
mov esi, offset string
lea esi, string

Procedure PROC
local temp: DWORD, swap: BYTE
↳ push ebp
mov ebp,esp
add, 0FFFFFFF8h ; add -8 to ESP
...
mov esp, ebp
POP ebp
ret

destination ↑ source

MOV BL, AL

D → linked ← REG

as reg has source → AL → source → choose self → always a register

0 → 000 → check table

R/M → linked ← MOD

other operator ← BL → 011 → check table

11 → as both registers

OPCODE → MOV → 100010 → will be given

W → 0 → as 8 bit operation

OPCODE	D	W	MOD	REG	R/M
100010	0	0	11	000	011

Convert to hexa

MOV BL, AL => 88C3H

destination ↑ source

MOV CX, DX

OPCODE	D	W	MOD	REG	R/M
100010	1	1	11	001	010
given	REG is destination	16 bit	both registers	CX → take any as both registers	DX

into hexa

89CAH

MOV 100010

ADD 000000

SUB 001010

REG = source D=0

REG = destination D=1

memory — reg

ADD [BX],[DI] + 4567H, DX

000000 — H 45 L 67 — 16 bit

OPCODE	D	W	MOD	REG	R/M	displacement LOW	displacement H
000000	0	1	10	010	001	67H	45H
	source	16 bit	memory 16	DX			

into hexa

0191 6745H

W=1 16 bit

W=0 8 bit

ADD AX, [BP][SI] + 45 H — 8 bit

OPCODE	D	W	MOD	REG	R/M	displacement LOW	displacement H
000000	1	1	01	000	010	45H	
	destination	16 bit AX	8 bit memory	AX	memory		

into hexa

0342 45H

BYTE 1 — BYTE 2

Opcode (6 bit)	D (direction)	W (size of memory)	MOD	REG	R/M (one second)	LB displacement	HB displacement
100010	0	1	10	010	110	34H	12H

mov mem — between a register & mem decide one ! — IF REG = SOURCE D= IF OPP = DESTINATION

(a)

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W=0	W=1	R/M	MOD=00	MOD=01	MOD=10
000	AL	AX	000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16
001	CL	CX	001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16
010	DL	DX	010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16
011	BL	BX	011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16
100	AH	SP	100	(SI)	(SI)+D8	(SI)+D16
101	CH	BP	101	(DI)	(DI)+D8	(DI)+D16
110	DH	SI	110	DIRECT ADDRESS	(BP)+D8	(BP)+D16
111	BH	DI	111	(BX)	(BX)+D8	(BX)+D16

111 BH DI

CISC

- emphasis on hardware
- multiple clock cycle
- less pipeline
- memory to memory (load and store)
- variable format instruction
- any instruction may refer memory

RISC

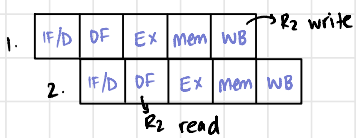
clock cycle (fetching, decoding, executing)

- emphasis on software
- single clock cycle
- highly pipelined
- register to register
- fixed format instruction
- only load/store refer memory

Pipeline Hazards

- Data Hazard
  - write after read
  - write after write
  - read after write

1. $R_2 \leftarrow R_2 + R_3$
2. $R_5 \leftarrow R_2 + R_4$



hence reads wrong value of R_2

Pipelining

- process of arrangement of hardware elements of CPUs to increase performance
- execute more than one instruction simultaneously
- instructions overlap in execution

- uses RISC to pipeline in 5 stages
- space time diagram

solution

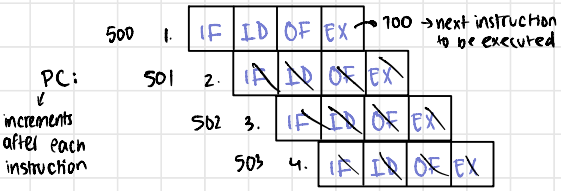
- add nop (non operation instructions)
- instruction scheduling

$R_2 \leftarrow R_2 + R_3$
nop
nop
nop
$R_5 \leftarrow R_2 + R_4$

- add stalls / bubbles
- data forwarding

↳ Control Hazard

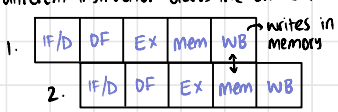
branch conditions



but we going linewise so all these will be flushed as 700 th instruction need to be executed

↳ Structural Hazard

different instruction acess the same thing



to different instructions accesing memory at same time

C B W	convert byte to word
C W D	word to dword
C D Q	dword to quadword

MIPS

Types of instructions

- **Data operations** R-type
  - Arithmetic (add, subtract, …)
  - Logical (and, or, not, xor, …)
- **Data transfer** I-type
  - Load (memory → register)
  - Store (register → memory)
- **Sequencing**
  - Branch (conditional, e.g., <, >, ==) I-type
  - Jump (unconditional, e.g., goto)

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 – R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R1 * R2
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 143
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

(Complete table is printed in the book for reference.)

R-Format for Register-Register ALU Instructions

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

- **op: (opcode)** identifies the ALU instruction type;
- **rs:** first source operand
- **rt:** second source operand
- **rd:** destination register
- **shamt:** shift amount
- **funct:** identifies the different type of ALU instructions

I-Format for Immediate ALU Instructions

op	rs	rt	immediate / offset / address
6 bit	5 bit	5 bit	16 bit

- **op (opcode):** identifies immediate instruction type;
- **rs:** source register;
- **rt:** destination register;
- **immediate:** contains the value of the immediate operand (in the range -2^{15} $+2^{15}-1$).

J-Format for Unconditional Jumps

op	address
6 bit	26 bit

- **op (opcode):** identifies the jump instruction type;
- **address:** contains 26-bit of 32-bit absolute word address of jump destination:

4 bit	26 bit	2 bit
PC+4 [31-28]	address [25-0]	00

ishma hafeez notes