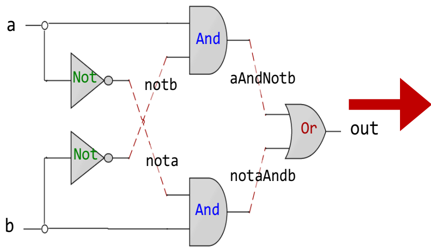
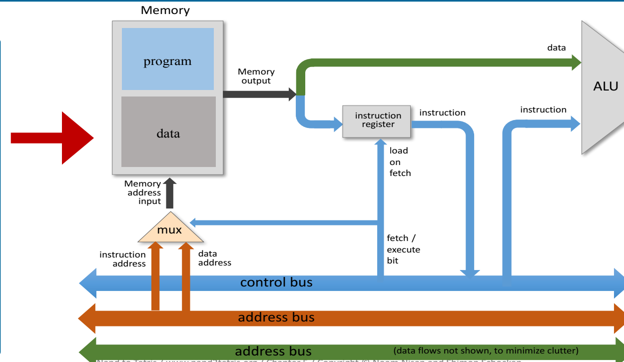
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

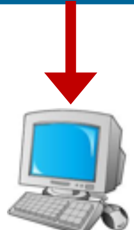
Lecture # 34

Memory Addressing Modes

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```



For resources visit my personal website:
https://www.arifbutt.me
and course bitbucket repository:
https://bitbucket.org/arifpucit/coal-repo

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Implied addressing mode
- Immediate addressing mode
- Register addressing mode
- Register Indirect addressing mode
- Auto Increment addressing mod
- Auto Decrement addressing mode
- Direct/Absolute addressing mode
- Memory Indirect addressing mode
- Displacement addressing modes
- x86-64 Base-Index-Scale-Displacement addressing mode

Addressing Modes

General format of an assembly instruction:

Opcode	Operand(s)

- In assembly language programming, the term addressing modes refers to the way in which the operand(s) of an instruction is/are specified, which can be immediate values, registers or memory locations
- An operand field provides the location, where the data to be processed is stored. Different architectures support different addressing modes
- x86 instructions vary in the number of operands, however, most instructions use two operands with the first operand as destination (Intel syntax). The following combinations are typically legal, but again this varies from instruction to instruction
  - Register to register
  - Register to memory
  - Memory to register
  - Immediate to register
  - Immediate to memory

Implied Addressing Mode

- In implied addressing mode, the operand is specified in the instruction opcode. Zero address instructions are designed with implied addressing mode

- **Examples:**
  - The ***CLA*** instruction means clear the Accumulator register
  - The ***CMA*** instruction means complement the Accumulator register
  - The ***CLE*** instruction means clear the overflow flag
  - The ***ION*** instruction means set the interrupt bit on
  - The ***IOF*** instruction means set the interrupt bit off
  - The stack based instruction ***ADD*** means, take two operands from top of process stack, add them, and place the result back on top of stack
- **Pros/Cons:**
  - The instruction size is very small

Immediate Addressing Mode

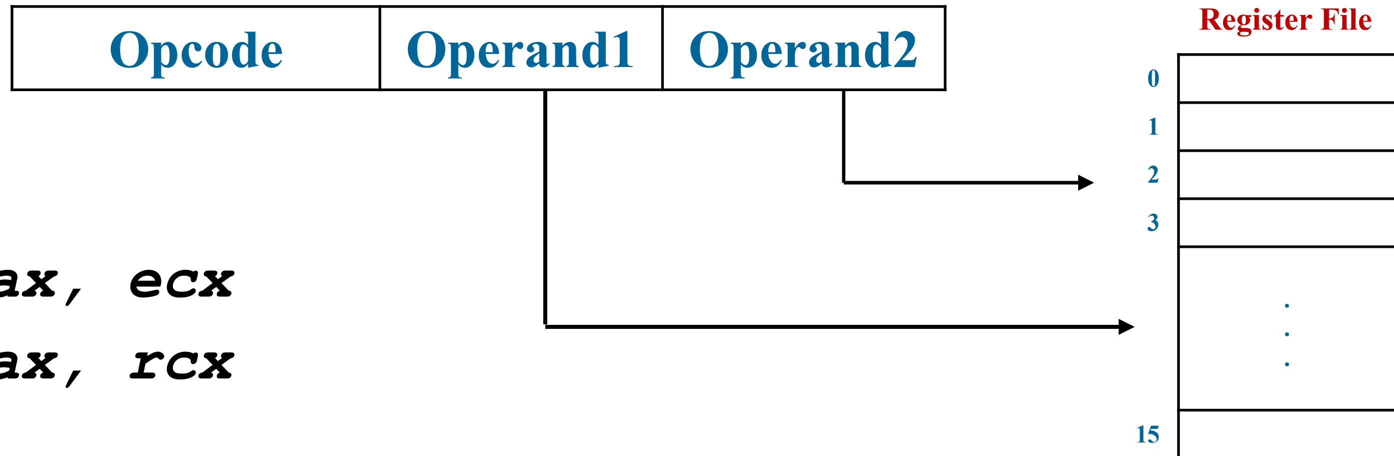
- In immediate addressing mode, the operand (8, 16, 32, 64 bits) is directly provided as a constant in the instruction itself

Opcode	Operand(s)

- **Examples:**
  - ***MOV al, 0x3A***
  - ***MOV eax, 123***
- **Pros/Cons:**
  - Fast, since no memory reference is required to access the data
  - Range of constant value depends on the total number of bits of the operand field in the instruction. For example, for 8 bit operand, you can have an unsigned constant ranging from 0 to 255, or a signed constant ranging from -128 to 127

Register Addressing Mode

- In register addressing mode, the operand is present in the register, and the register number is encoded within the instruction itself

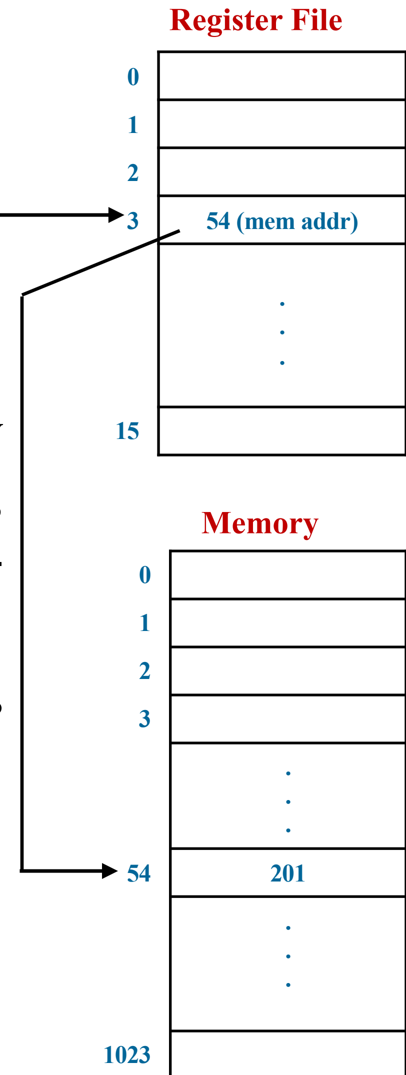


- **Examples:**
  - ***MOV eax, ecx***
  - ***ADD rax, rcx***
- **Pros/Cons:**
  - Less number of bits required to encode the registers, as compared to memory address. For example if there are a total of 16 registers, only 4 bits are required to encode one register
  - Speed is better as registers can be accessed far quickly than memory

Register-Indirect Addressing Mode

- In register-indirect addressing mode, the register contains memory address of operand rather than the operand itself

- **Example:**
  - ***MOV rax, qword[r3]***
- **Pros/Cons:**
  - For 1 GiB memory, if we want to encode the memory address directly inside the instruction, we need 30 bits, thus increasing the size of the instruction. Register-indirect addressing mode reduces the instruction size
  - Disadvantage is a bit of computation is involved as compared to register addressing mode



Auto-Increment Addressing Mode

- In auto-increment addressing mode the effective address of the operand is the contents of a register specified in the instruction. **After** accessing the operand, the contents of this register are automatically incremented to point to the **next** operand in memory
- Useful for stepping through arrays in a loop, where register r2 contains the start of array
- One register reference, one memory reference and one ALU operation is required to access the data

- **Example:**
  - ***Add r1, [r2]+***

```
r1 = r1 + M[r2]
r2 = r2 + d  // where d is the size of an element
```

Auto-Decrement Addressing Mode

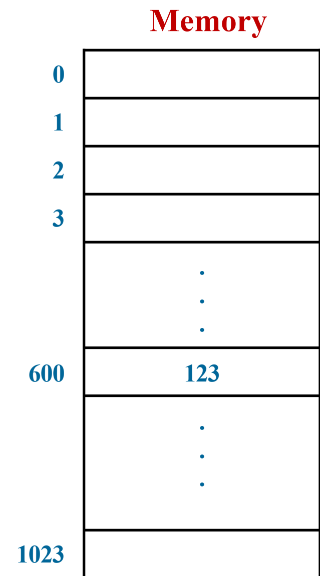
- In auto-decrement addressing mode the effective address of the operand is the contents of a register specified in the instruction. **Before** accessing the operand, the contents of this register are automatically decremented and are then used as the effective address of the operand
- One register reference, one memory reference and one ALU operation is required to access the data
- **Example:**
  - ***Add r1, -[r2]***

r2 = r2 - d // where d is the size of an element

r1 = r1 + M[r2]

Direct/Absolute Addressing Mode

- In direct/absolute addressing mode, the actual memory address is given inside the instruction usually indicated by a variable name



- **Examples:**
  - ***`MOV al, byte[600]`***
- **Pros/Cons:**
  - Effective address is there inside the instruction itself, therefore, only one memory reference is required to access the operand
  - Consider a fixed length instruction of 16 bits only, `add reg, [addr]`, with 5 bits for opcode, 4 bits for register. Then we are left with only 9 bits to encode the memory address. So only 512 memory locations can be accessed

Memory In-Direct Addressing Mode

- In in-direct addressing mode, the address inside the instruction is the memory address, which further contain the address where the actual operand resides (used to implement pointers and passing parameters)

Opcode	Memory Address

- **Pros/Cons:**
  - Large memory address space can be accessed using in-direct addressing mode
  - Slow, since two memory accesses are required to access the operand

Displacement Addressing Mode

- In displacement addressing mode, the operand is at some memory location, whose effective address is the sum of some register contents and a constant within the instruction itself. Generally, it is of three categories:
  - Relative addressing mode
  - Indexed addressing mode
  - Base addressing mode

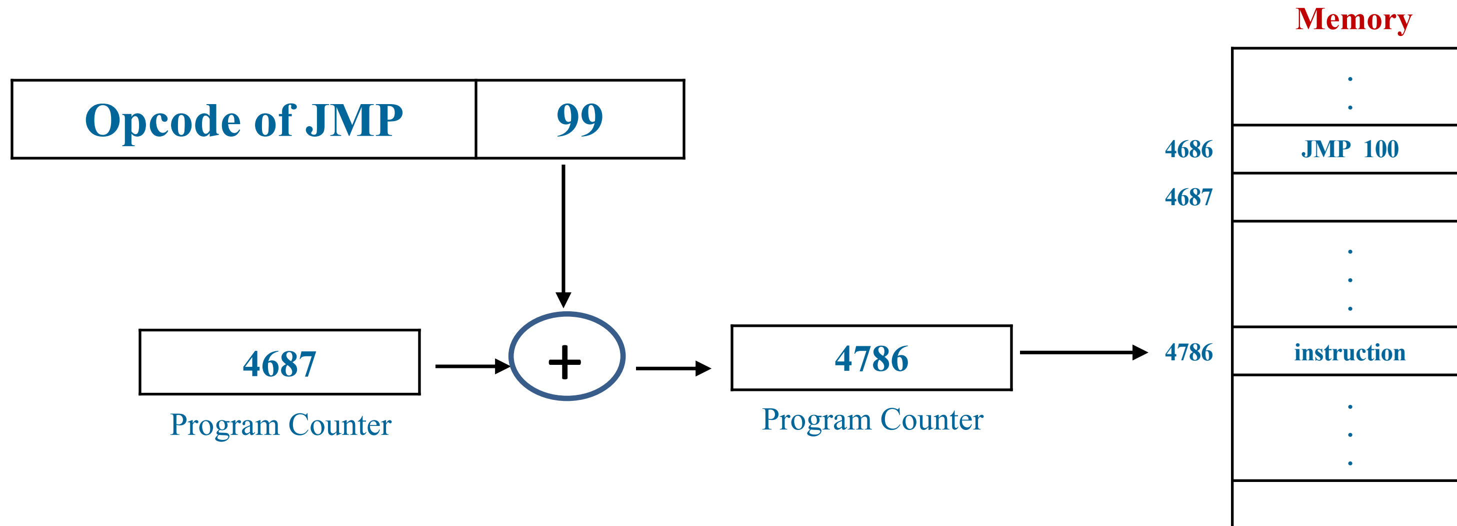
Opcode	R	A

Register File
0
1
2
3 value
15

Memory
0
1
2
3
532 operand
1023

Relative Addressing Mode

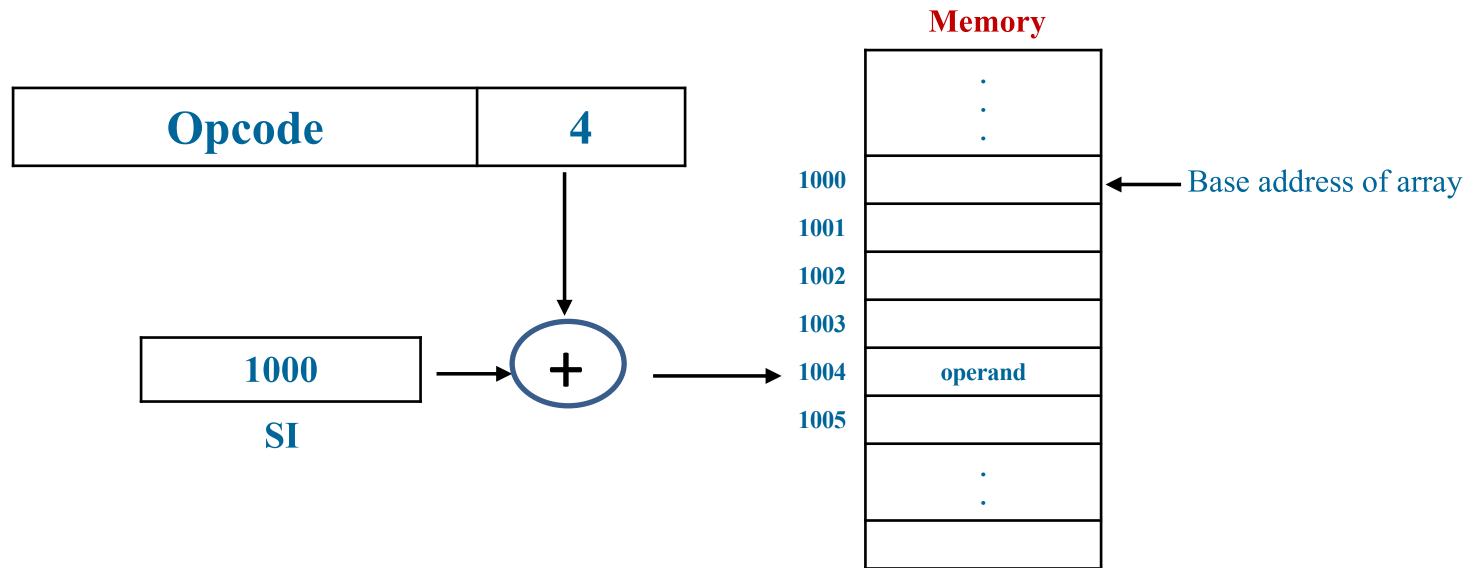
- This is also called PC Relative addressing mode, and is used to implement intra segment transfer of control (branching)



Remember, PC always contains the address of the next instruction, so the offset or displacement should be one less

Indexed Addressing Mode

- In indexed addressing mode, the effective address of the operand is generated by adding a constant value to the contents of a register
- Used to access or implement arrays efficiently
- **Example:**
  - ***MOV eax, [si + 4]***



X86-64 Displacement Addressing Mode

- In x86-64, there are four components used to compute the 64-bit effective address:

  [baseAddr + (indexReg * scaleValue) + displacement]

  - **baseAddr** is a register or a variable name
  - **indexReg** must be a register
  - **scaleValue** is a 2-bit constant factor that is either 1, 2, 4, or 8
  - **displacement** is an immediate value/offset, normally limited to 32 bits
- **Examples:**

```
; baseAddr only
MOV rax, qword[rbx]
; baseAddr with displacement
MOV rax, qword[rbx + 128]
; indexReg with displacement
MOV rax, qword[rdi*2 + 128]
; baseAddr with indexReg
MOV rax, qword[rbx+rsi*4]
; baseAddr with indexReg and displacement
MOV eax, dword[rbx+rsi*4+ 50]
```

Examples: X86-64 Displacement Addressing Mode

[baseAddr + (indexReg * scaleValue) + displacement]

```
arr dd 0x65, 0x2a, 0x1c, 0x46
```

;First element of the array can be accessed as

```
mov eax, dword [arr]
```

;Another way to access the first element of the array is

```
mov rbx, arr
mov eax, dword [rbx]
```

;To access the third element of the array

```
mov eax, dword [arr+8]
```

;Another way to access the third element of the array is

```
mov rbx, arr
mov rsi, 8
mov eax, dword [rbx+rsi]
```

;Another way to access the third element of the array is

```
mov rsi, 2
mov eax, dword [arr+rsi*4]
```

Address	Value	
1015	0x00	
1014	0x00	
1013	0x00	
1012	0x4c	arr[3]
1011	0x00	
1010	0x00	
1009	0x00	
1008	0x1c	arr[2]
1007	0x00	
1006	0x00	
1005	0x00	
1004	0x2a	arr[1]
1003	0x00	
1002	0x00	
1001	0x00	
arr → 1000	0x65	arr[0]

Things To Do

Coming to office hours does NOT mean you are academically week!