# Computer Organization and Assembly Language CS / EE 320 Spring 2024

Shahid Masud

Lecture 7

# Topics

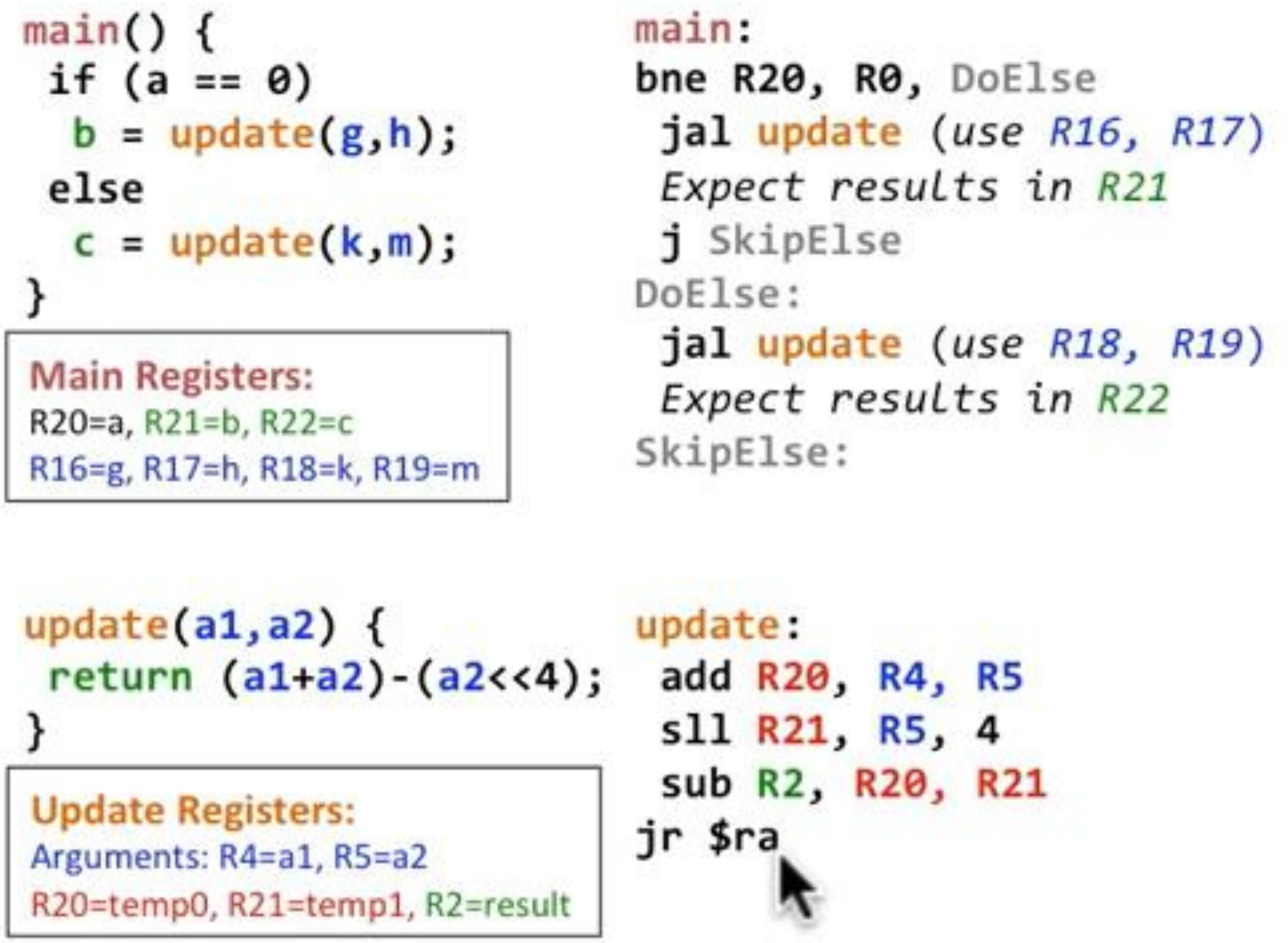
- Working of MIPS Assembly Instructions
    - Jump Instructions
    - Branch Instructions
    - Examples of Different Instruction Type
    - Addressing Modes
    - Interrupt Processing

Control Instruction

# Jump Instructions in MIPS ISA

- J jump to 26-bit Address
- JAL jump and link, return address stored in register ra
  - JAL is the only instruction that can access PC
- JR jump to register ra, to return after processing JAL
- JALR jump and link automatically to subroutine address and return address stored in ra

# Procedure Call Example



```
main() {
 if (a == 0)
  b = update(g,h);
 else
  c = update(k,m);
}
```

**Main Registers:**
R20=a, R21=b, R22=c
R16=g, R17=h, R18=k, R19=m

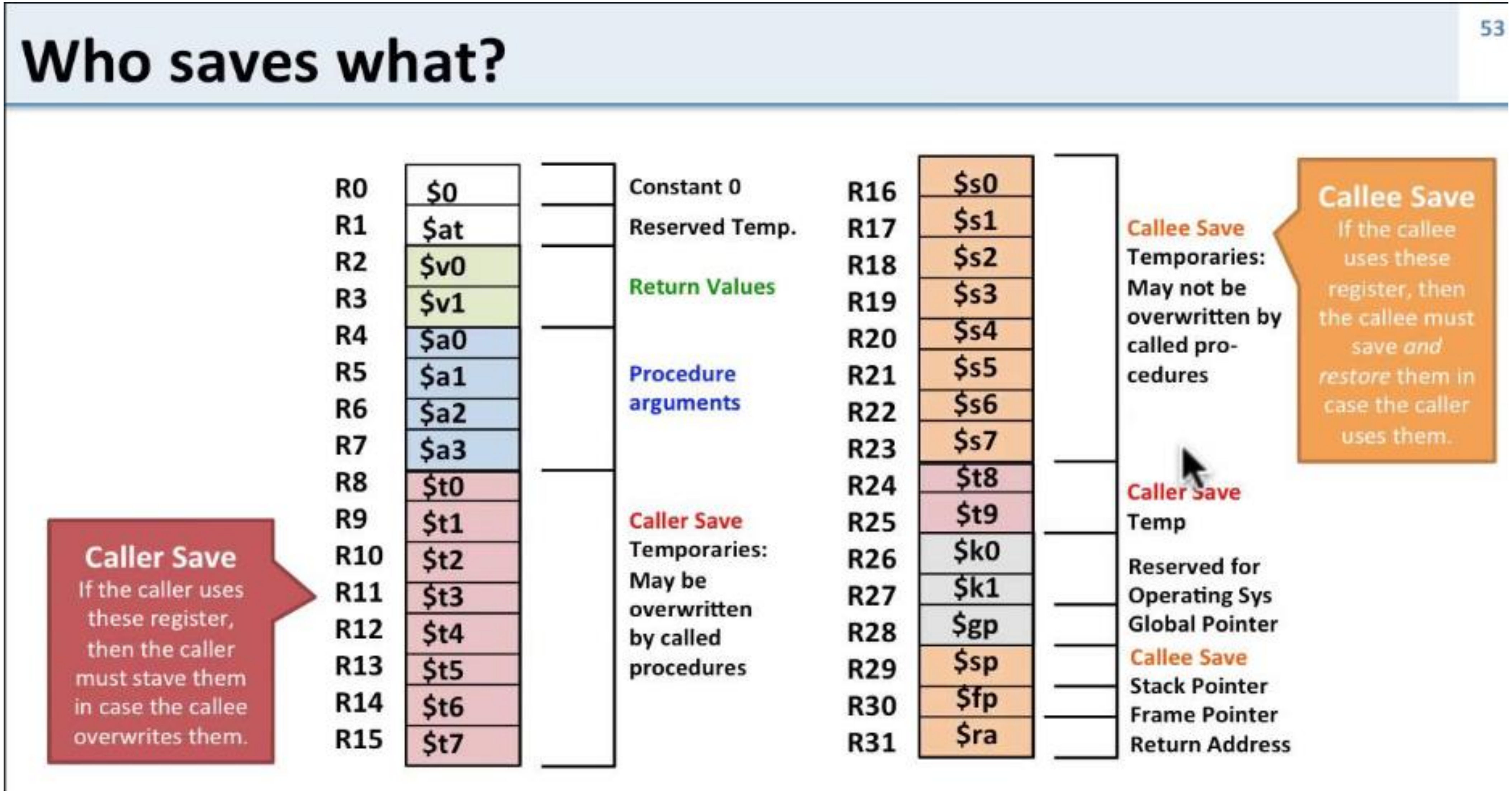
```
main:
bne R20, R0, DoElse
 jal update (use R16, R17)
 Expect results in R21
 j SkipElse
DoElse:
 jal update (use R18, R19)
 Expect results in R22
SkipElse:
```

```
update(a1,a2) {
 return (a1+a2)-(a2<<4);
}
```

**Update Registers:**
Arguments: R4=a1, R5=a2
R20=temp0, R21=temp1, R2=result

```
update:
 add R20, R4, R5
 sll R21, R5, 4
 sub R2, R20, R21
jr $ra
```

# Register Convention – Caller and Callee



53

## Who saves what?

| | | | |
|---|---|---|---|
| R0 | $0 | R16 | $s0 |
| R1 | $at | R17 | $s1 |
| R2 | $v0 | R18 | $s2 |
| R3 | $v1 | R19 | $s3 |
| R4 | $a0 | R20 | $s4 |
| R5 | $a1 | R21 | $s5 |
| R6 | $a2 | R22 | $s6 |
| R7 | $a3 | R23 | $s7 |
| R8 | $t0 | R24 | $t8 |
| R9 | $t1 | R25 | $t9 |
| R10 | $t2 | R26 | $k0 |
| R11 | $t3 | R27 | $k1 |
| R12 | $t4 | R28 | $gp |
| R13 | $t5 | R29 | $sp |
| R14 | $t6 | R30 | $fp |
| R15 | $t7 | R31 | $ra |

Constant 0

Reserved Temp.

Return Values

Procedure arguments

Caller Save Temporaries: May be overwritten by called procedures

Callee Save Temporaries: May not be overwritten by called procedures

Caller Save Temp

Reserved for Operating Sys

Global Pointer

Callee Save Stack Pointer

Frame Pointer

Return Address

**Caller Save**
If the caller uses these register, then the caller must stave them in case the callee overwrites them.

**Callee Save**
If the callee uses these register, then the callee must save *and restore* them in case the caller uses them.

Ref: youtube channel of David Black-Schaffar

# MIPS Instruction Format and Registers

| Name | Fields | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All instr are 32 bits long |
| **R Format** | op | rs | rt | rd | shamt | funct | Arithmetic Instructions |
| **I Format** | Op | rs | rt | Address / Immediate **(16 bits)** | | | Transfer, Branch Instructions |
| **J Format** | Op | Target address **(26 bits)** | | | | | Jump Instructions |

- **MIPS has 3 Instruction Formats, viz a viz Registers**
  - **R:** operation **3 Registers** **0 immediate**
  - **I:** operation **2 Registers** **16 bit immediate**
  - **J:** jump **0 Registers** **26 bit immediate**

# Immediate Instruction Example

- **Addi R6, R0, 100**
- R0 is constant Zero
- Immediate Register has value 100 Decimal
  - Convert to Binary 16 bits
  - Sign Extend to full 32 bits
  - ALU adds sign extended (100) to Zero and stores result in register R6

# Addi Instruction with a Negative Immediate

R3 = 12

**addi R4, R3, -12**

R3 is a 32 bit register contains 12 =

0000 0000 0000 0000 0000 0000 **0000 1100**

16 bit Immediate value -12 =

– (0000 0000 0000 1100)

-12 in Two's Complement (with Sign Extension)

=

$$\overbrace{\text{1111 1111 1111 1111 1111 1111}}^{\text{sign extension}}\ \textbf{1111 0100}$$

Add Result → R4 = all Zeros

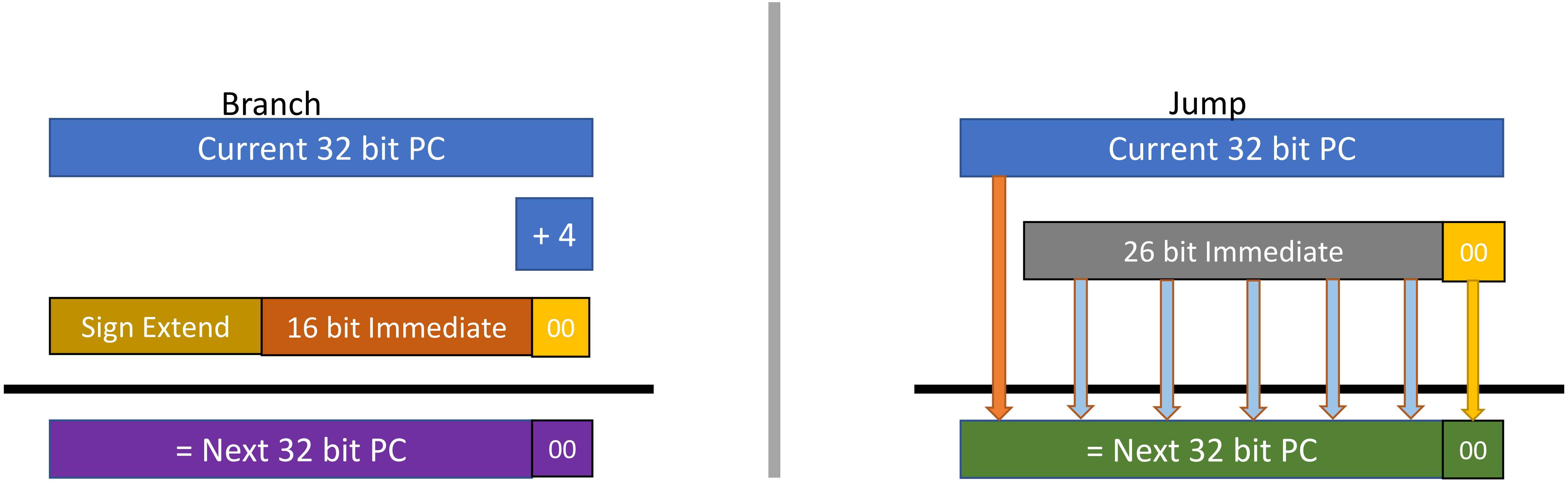
# Addresses in Branches and Jumps

- Branch Instructions
  - bne/beq I-Format 16 bit Immediate
  - j J-Format 26 bit Immediate

- But addresses are 32-bits, so:
  - Treat bne / beq as relative offset and add to current PC
  - Treat j as absolute value by replacing 26 bits of the PC

# Evaluating PC from Immediate Address

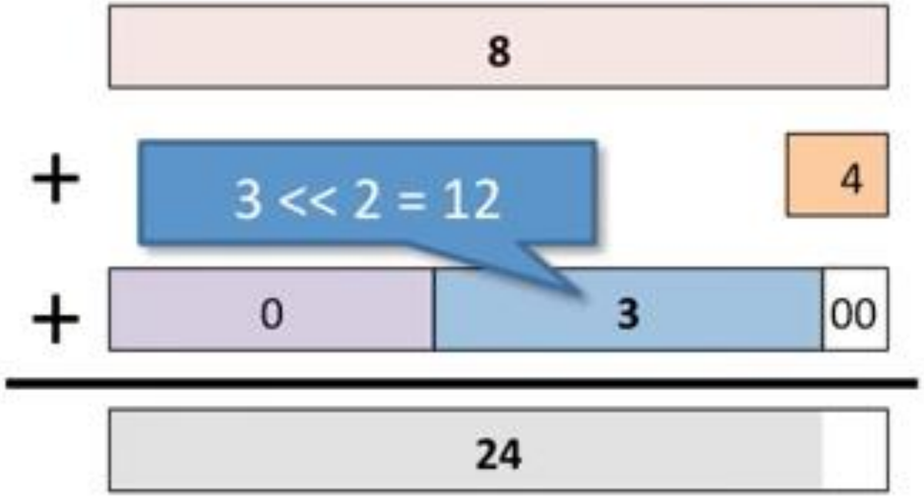
- The address provided by Jump Instruction can be ‘26 bits’
- The address provided by Branch (bne, beq) Instruction can be ‘16 bits’
- But all registers and memory addresses are 32 bits: So,



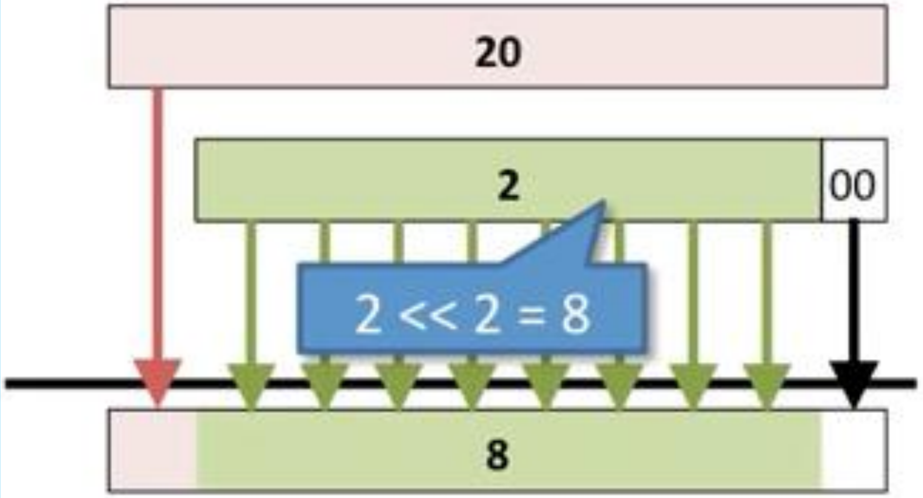
# Loops and Jump Instructions - Example

```
for (j=0; j<10; j++)
{
  b = b + j;
}
```

```
R5 = j;   R6 = b;
Addr        Instruction            Comment
0           addi R5, R0, 0         ; j ← 0 + 0
4           addi R1, R0, 10        ; R1 ← 0 + 10
8           beq  R5, R1, 3         ; if ( j == 10) goto 24
12          add  R6, R6, R5        ; b ← b + j
16          addi R5, R5, 1         ; j ← j + 1
20          j    2                 ; goto 8
24          ...                    ; done with loop
```



beq: PC=8+4+(3<<2)

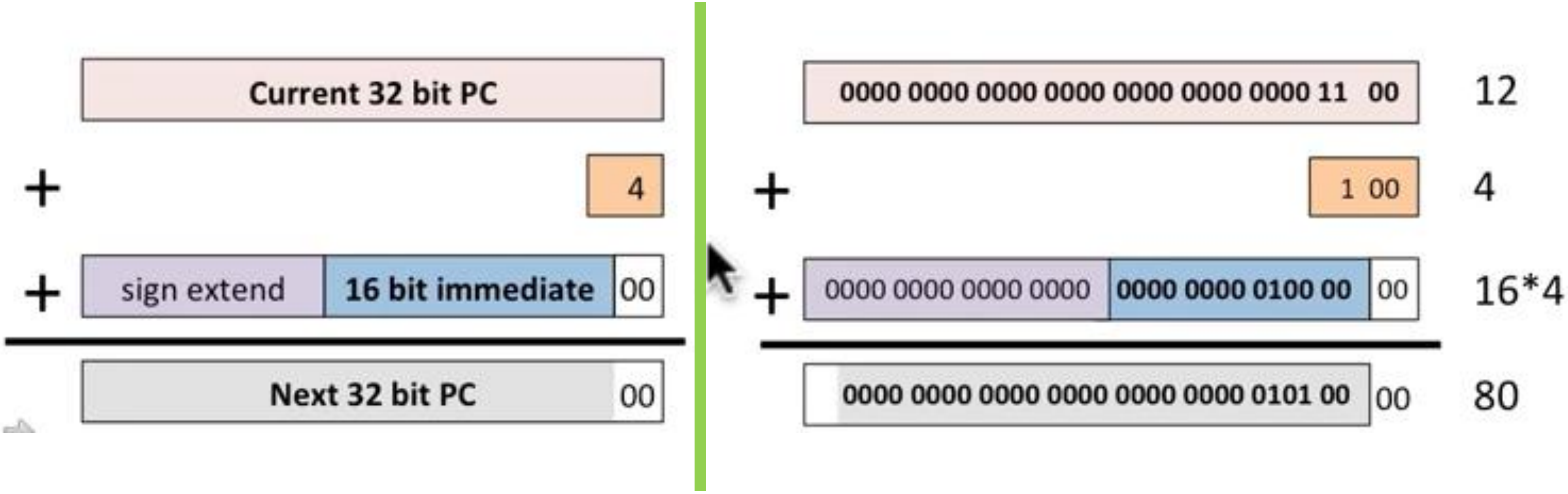


j: PC=[PC(31:28):2]<<2

# Example of Branch Destination

Instruction: **bne R0, R1, 16** # **Current Address is ‘12’**

Question: What is the destination address when Branch is Taken?



Current 32 bit PC

+ 4

+ sign extend | 16 bit immediate | 00

Next 32 bit PC | 00

0000 0000 0000 0000 0000 0000 0000 11 00 — 12

+ 1 00 — 4

+ 0000 0000 0000 0000 | 0000 0000 0100 00 | 00 — 16*4

0000 0000 0000 0000 0000 0000 0101 00 | 00 — 80

# Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

| op | rs | rt | constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

- **PC-relative addressing**
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

# Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

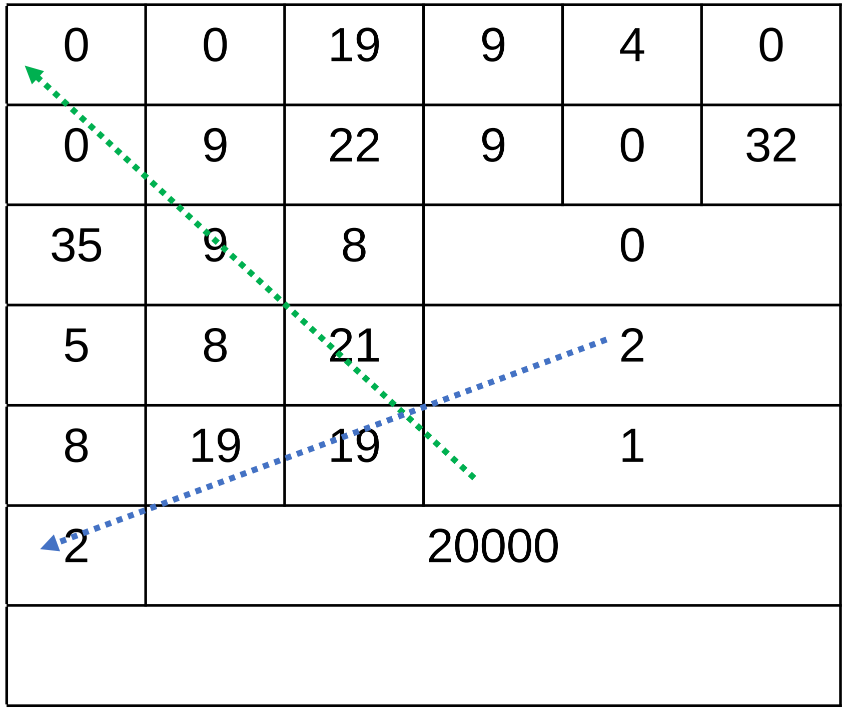
| op | **address** |
|---|---|
| 6 bits | 26 bits |

- (Pseudo) Direct jump addressing
  - Target address = $PC_{31\ldots28}$ **:** (address × 4)

# Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2          80000
      add  $t1, $t1, $s6        80004
      lw   $t0, 0($t1)          80008
      bne  $t0, $s5, Exit       80012
      addi $s3, $s3, 1          80016
      j    Loop                 80020
Exit: …                         80024
```



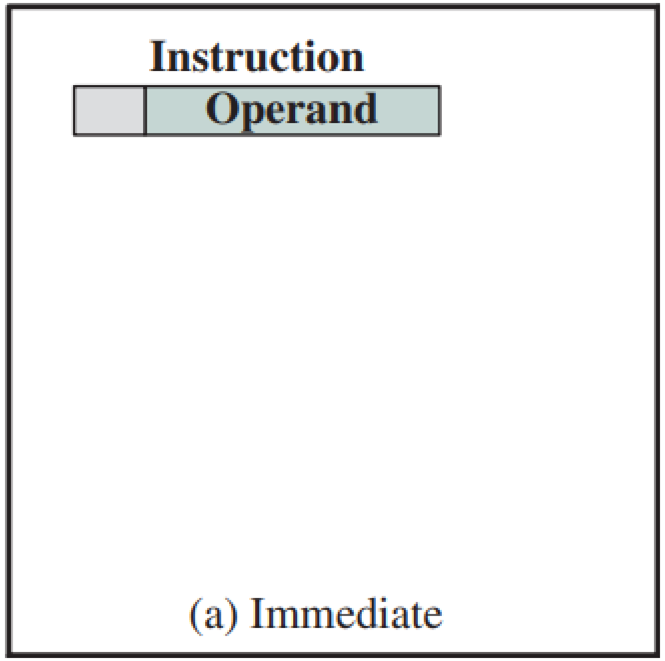
| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 19 | 9 | 4 | 0 |
| 0 | 9 | 22 | 9 | 0 | 32 |
| 35 | 9 | 8 | 0 | | |
| 5 | 8 | 21 | 2 | | |
| 8 | 19 | 19 | 1 | | |
| 2 | 20000 | | | | |
| | | | | | |

# Addressing Modes

# Addressing Modes in Assembly Language

- **Immediate**
- **Direct**
- **Indirect**
- **Register**
- **Register Indirect**
- **Displacement**
- **Stack**

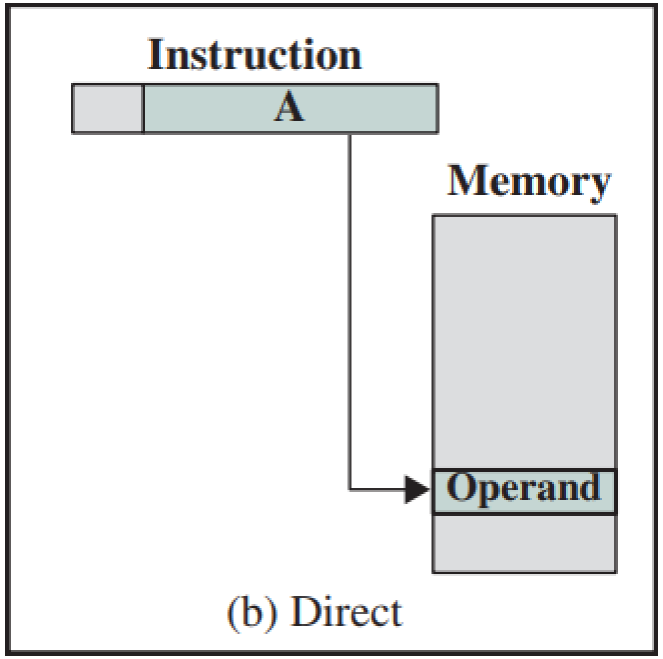
# Immediate Addressing



(a) Immediate

- The operand value is present in the instruction as Immediate
- Operand = A
- Mode is used to define Constants or Set Initial Values of Variables
- Mostly, Immediate value is stored in 2’s Complement form
- Advantage is that no other memory reference is required other than Instruction Fetch
- Disadvantage is the limited range of address and data that can be specified in Immediate field

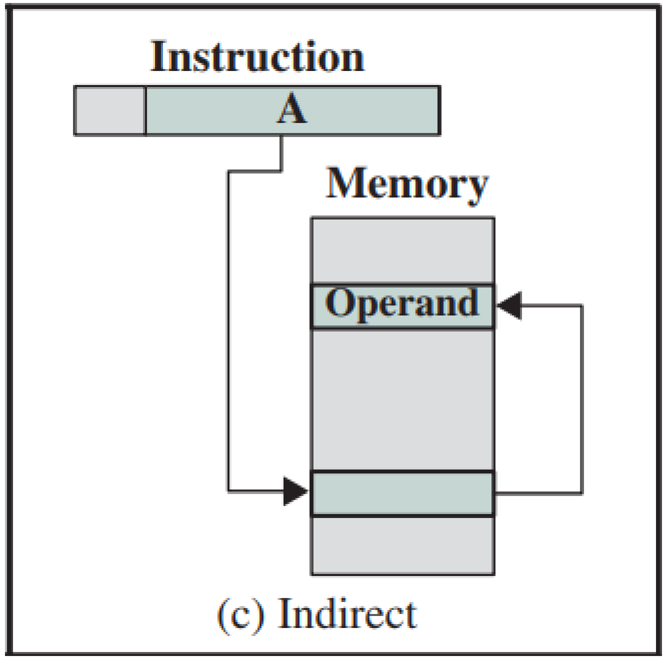
# Direct Addressing



(b) Direct

- Address field contains Effective Address of Operand
- It requires only one memory reference and no special calculations
- Disadvantage is limited address space can be specified

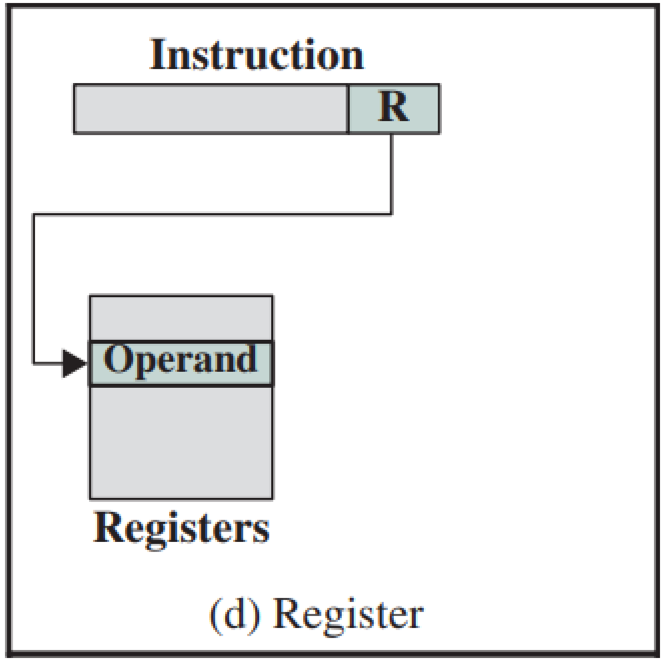
# Indirect Addressing



(c) Indirect

- The address field in Instruction refers to the address of a word in memory which in turn contains full-length address of the operand
- Advantage is that full range of memory is accessible
- Disadvantage is that two memory references are needed to fetch operand

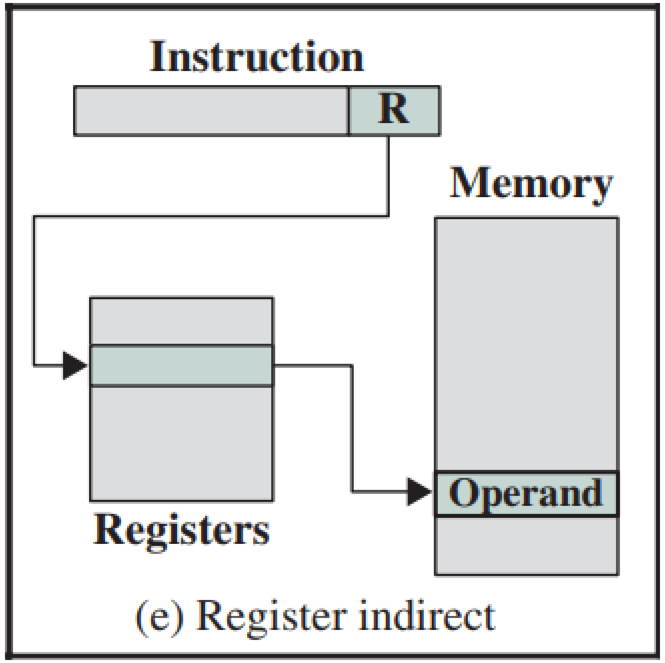
# Register based Addressing



(d) Register

- The address field in Instruction refers to a Register location
- Effective Address is stored in this register
- Advantage is that register access is fast and only one memory reference is needed to Fetch the operand
- Disadvantage is that number of registers is limited so only few registers can be used for this addressing mode

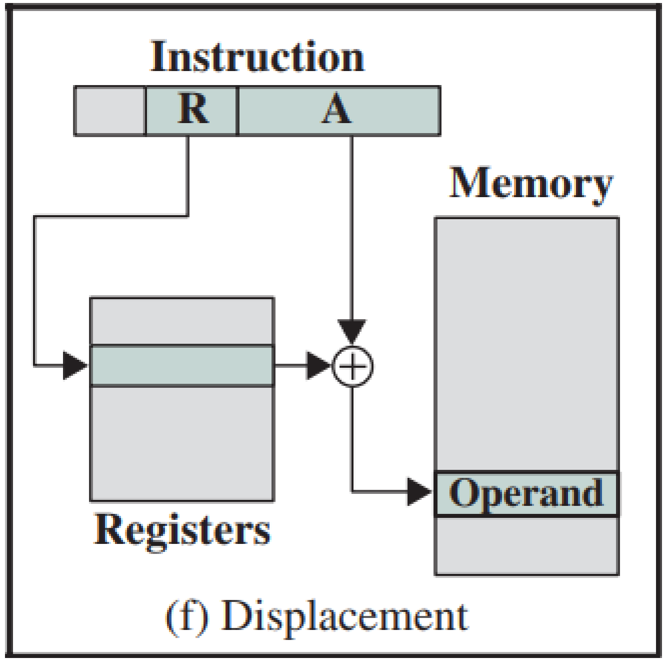
# Register Indirect



(e) Register indirect

- The address field refers to a Register that contains Memory Address of operand
- Limited number of registers is a constraint
- Only one memory reference is needed to Fetch the operand

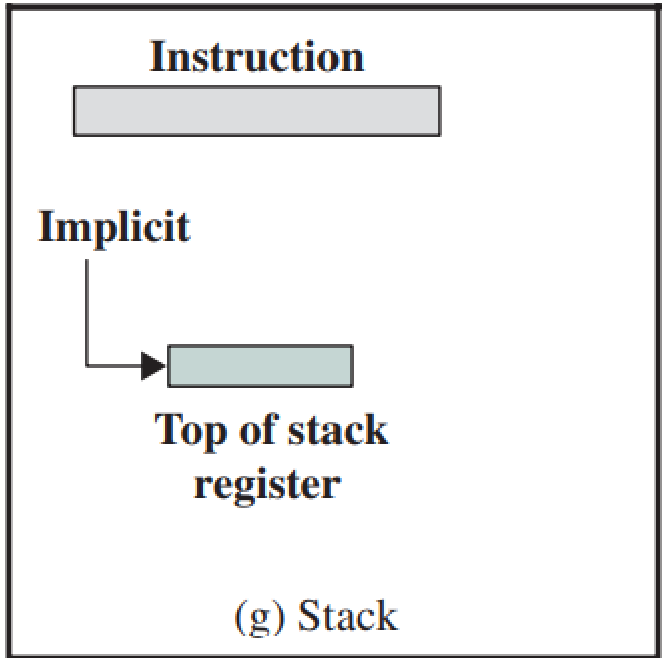
# Displacement



(f) Displacement

- Effective Address of operand is calculated by adding contents of ‘A’ Field in Instruction with contents of ‘R’ Register
- The operand is then fetched from the Effective Address
- Variants of this mode are:
  - PC Relative Addressing
  - Base Register Addressing (e.g. initial value in a String)
  - Indexing (storing in a sequence etc.)

# Stack based Addressing



(g) Stack

- Stack is a linear array of locations, also called ‘LIFO’ Last In First Out queue
- A Stack Pointer Register always contains address of the top of the Stack
- Implied Addressing uses Stack Pointer
- Some Displacement could also be added to the Stack Pointer

# Addressing Modes - Summary

Abbreviations:

A = Contents of Address field inside an Instruction

R = Contents of an Address field inside an Instruction that refers to a Register

(X) = contents contained in memory location X or register X

EA = Effective Address

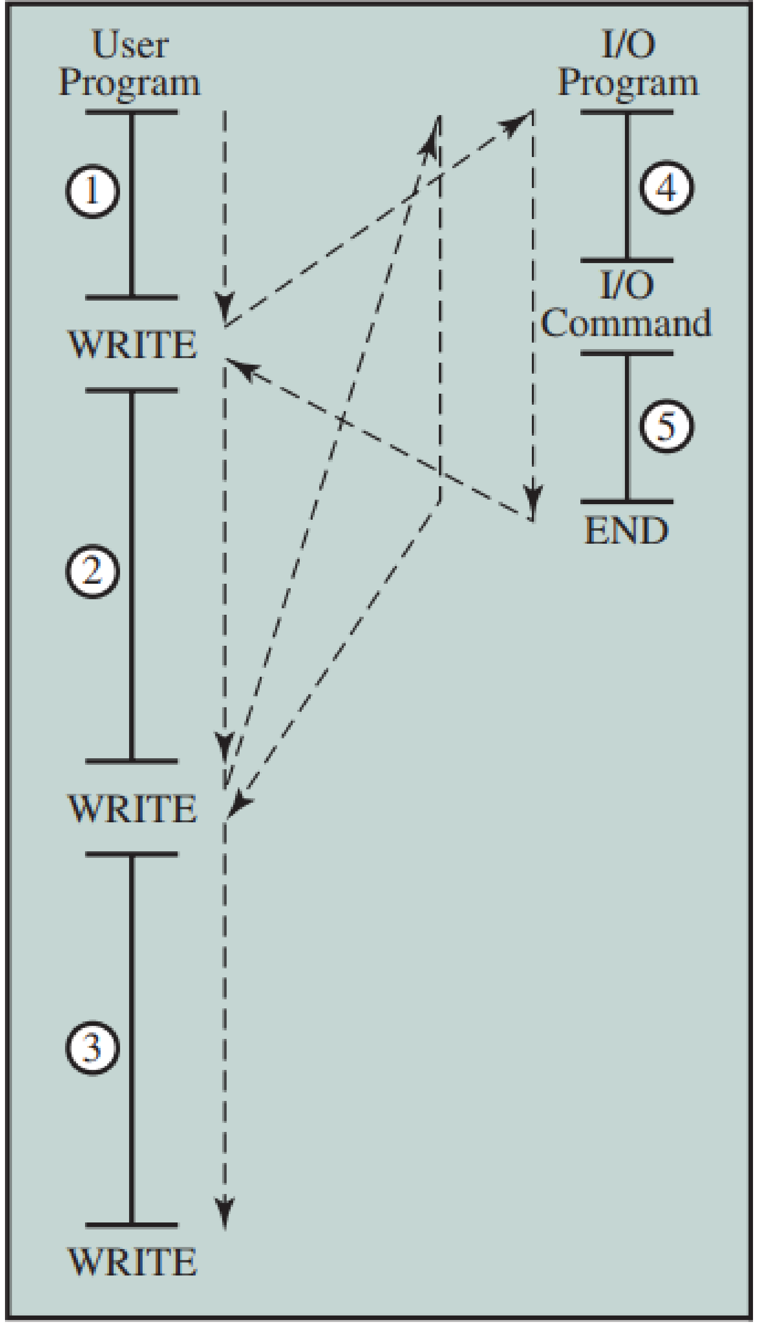
| Mode | Method | Advantage | Disadvantage |
|---|---|---|---|
| Immediate | Operand = A | No memory Reference | Limited Operand magnitude |
| Direct | EA = A | Simplicity | Limited address space |
| Indirect | EA = (A) | Large Address space | Multiple memory references |
| Register | EA = R | No memory reference | Limited Address space |
| Register Indirect | EA = (R) | Large Address space | Extra memory reference |
| Displacement | EA = A + (R) | Flexibility | Complexity |
| Stack | EA = top of stack, stack pointer | No memory reference | Limited usage and capability |

# Interrupts Processing

# Interrupts

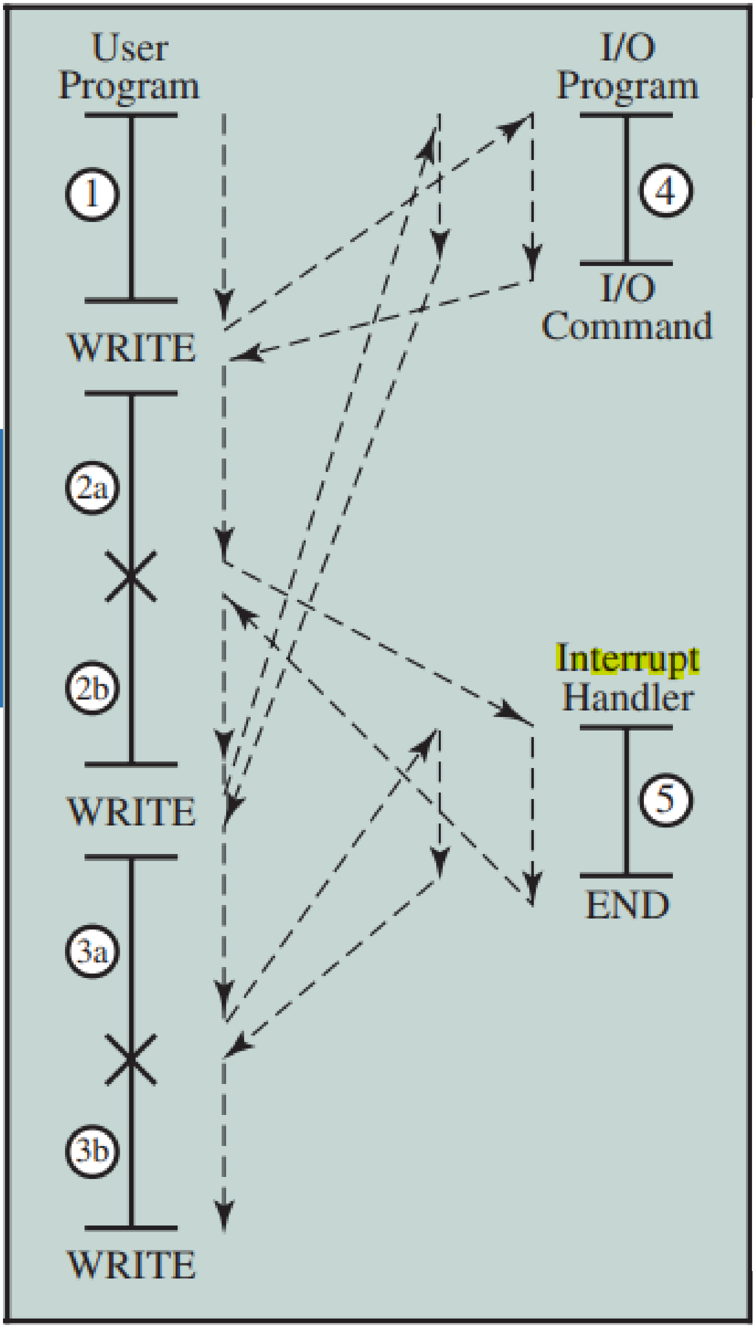
- Mechanism by which other modules (e.g. I/O) may interrupt normal sequence of processing
- Program Exception
  - e.g. overflow, division by zero
- Timer
  - Generated by internal processor timer
  - Used in pre-emptive multi-tasking
- I/O
  - from I/O controller
- Hardware failure
  - e.g. memory parity error

# Normal Program Flow Control

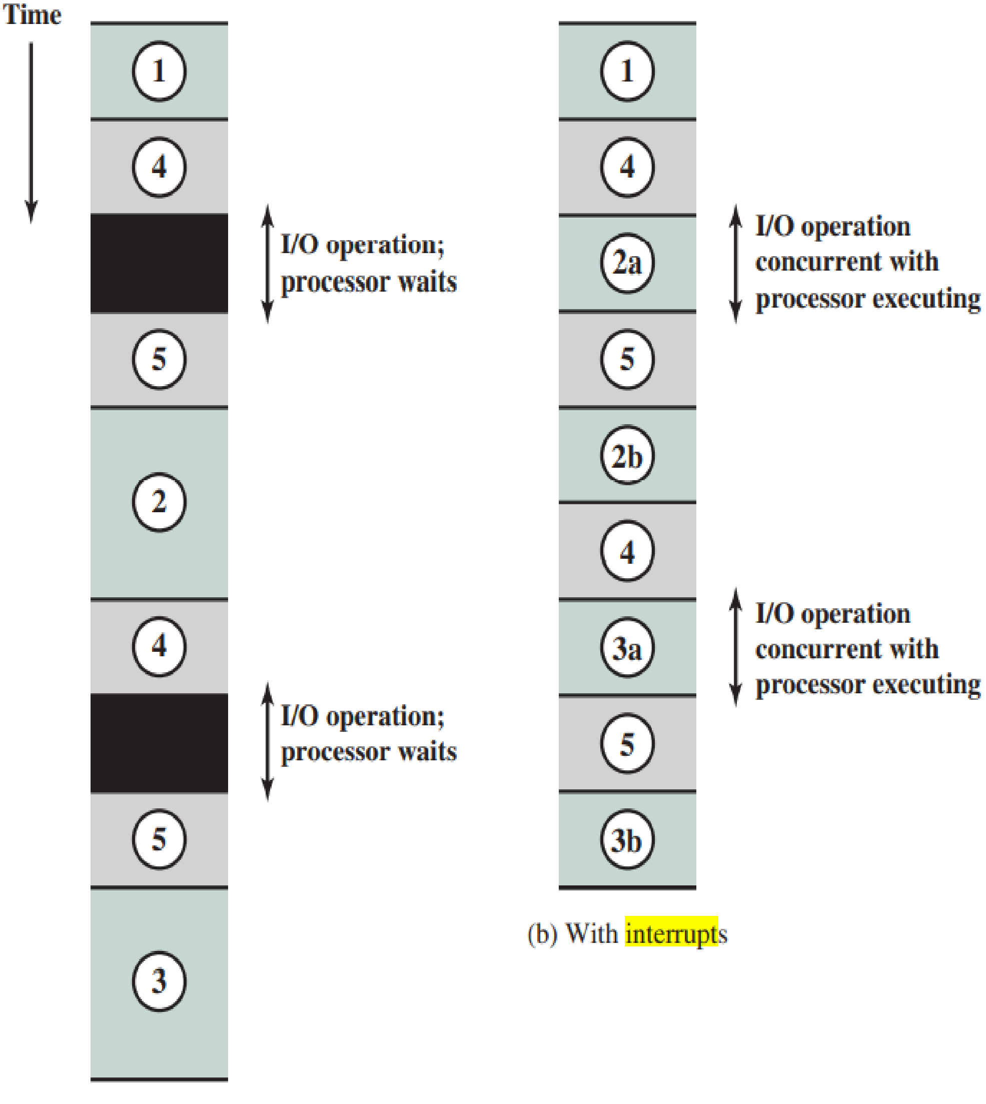


(a) No interrupts

# Program Timing: Short I/O Wait



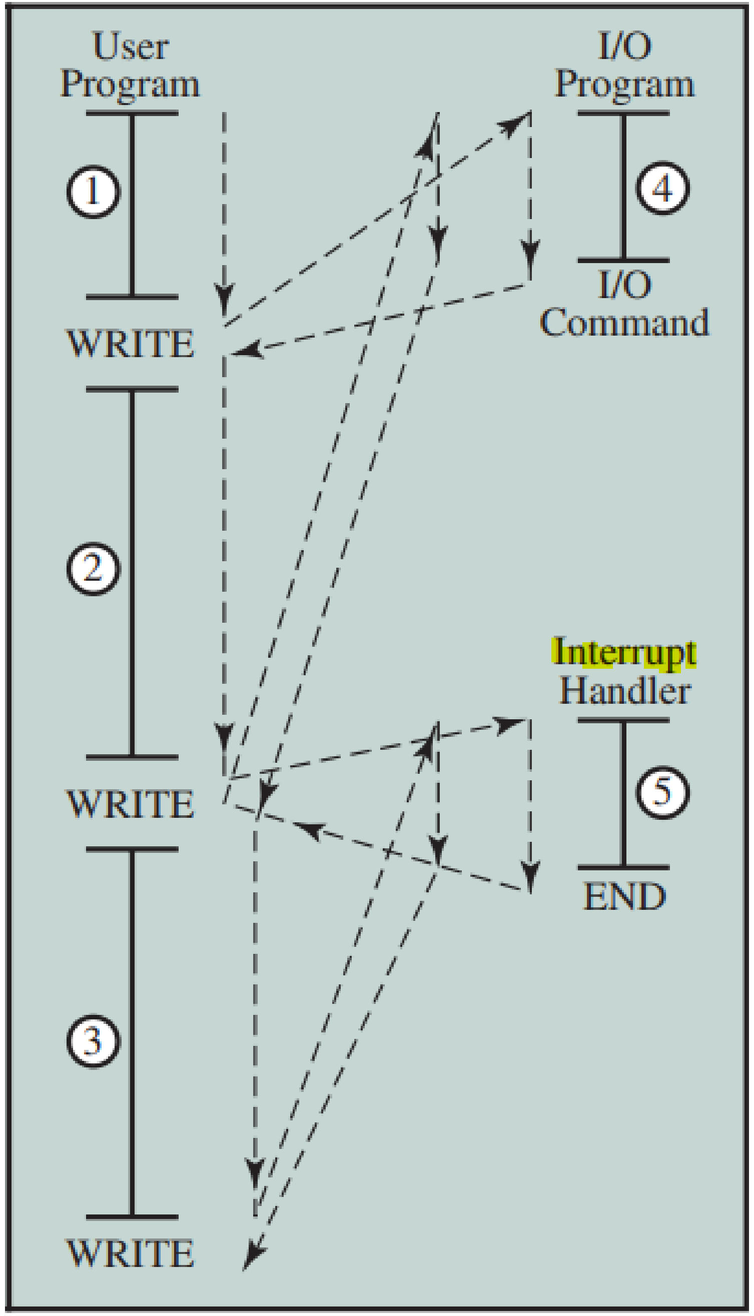
(b) Interrupts; short I/O wait


Time
1
4
I/O operation; processor waits
5
2
4
I/O operation; processor waits
5
3
1
4
2a
I/O operation concurrent with processor executing
5
2b
4
3a
I/O operation concurrent with processor executing
5
3b

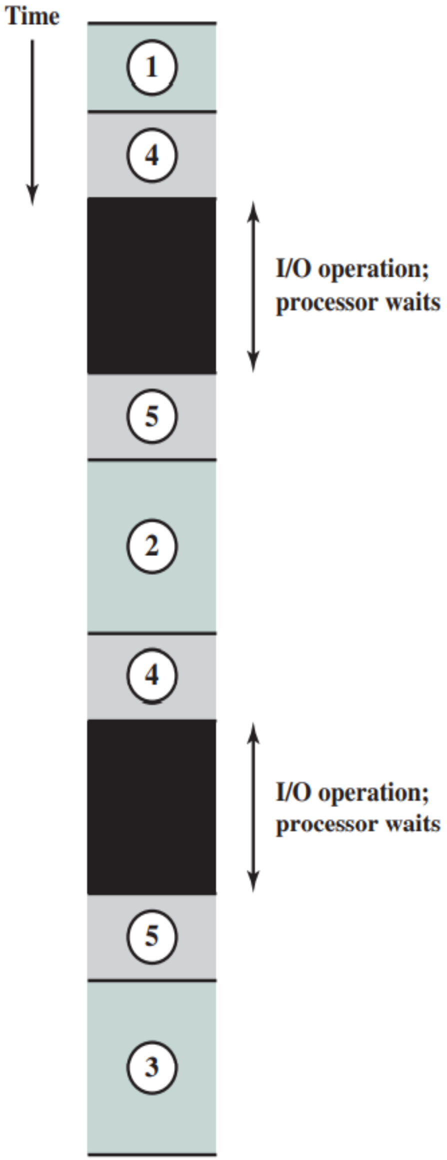
(b) With interrupts

(a) Without interrupts

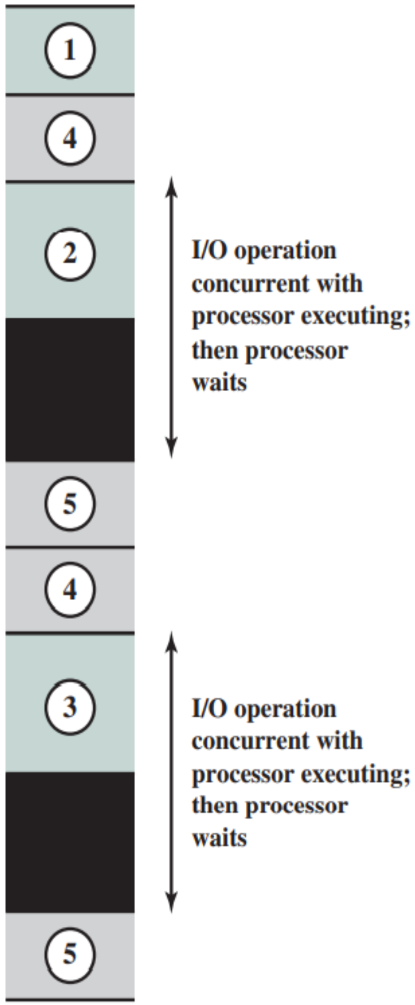
# Program Timing: Long I/O Wait



(c) Interrupts; long I/O wait

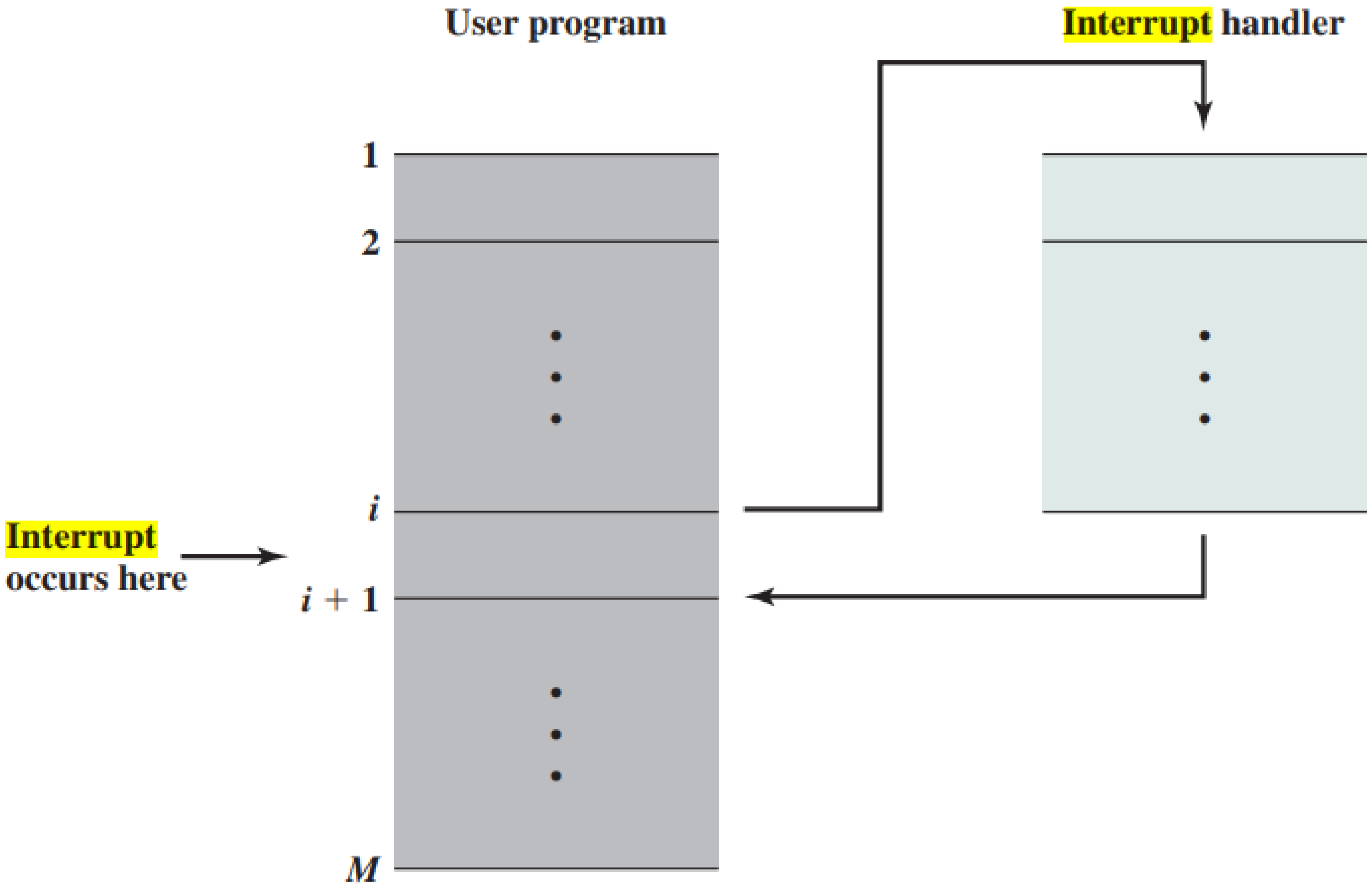

Time
1
4
I/O operation; processor waits
5
2
4
I/O operation; processor waits
5
3

(a) Without interrupts


1
4
2
I/O operation concurrent with processor executing; then processor waits
5
4
3
I/O operation concurrent with processor executing; then processor waits
5

(b) With interrupts

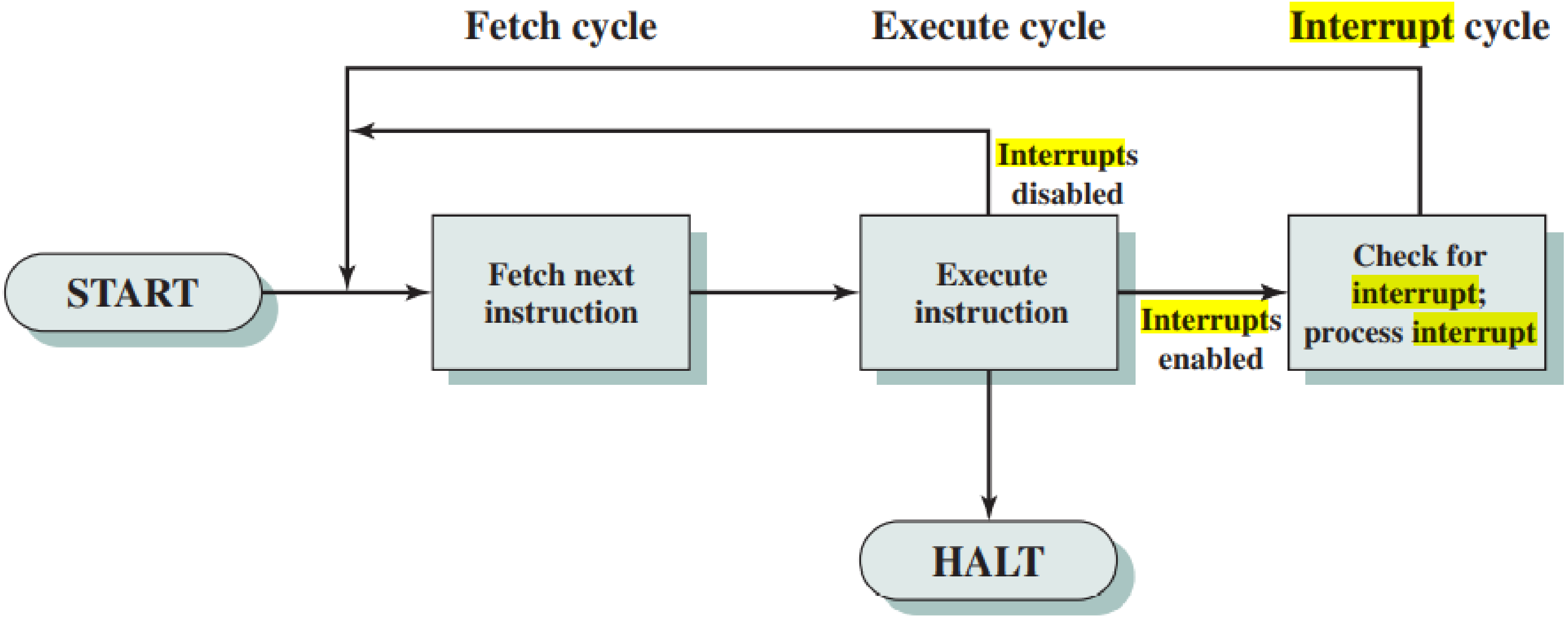
# Transfer of Control via Interrupt



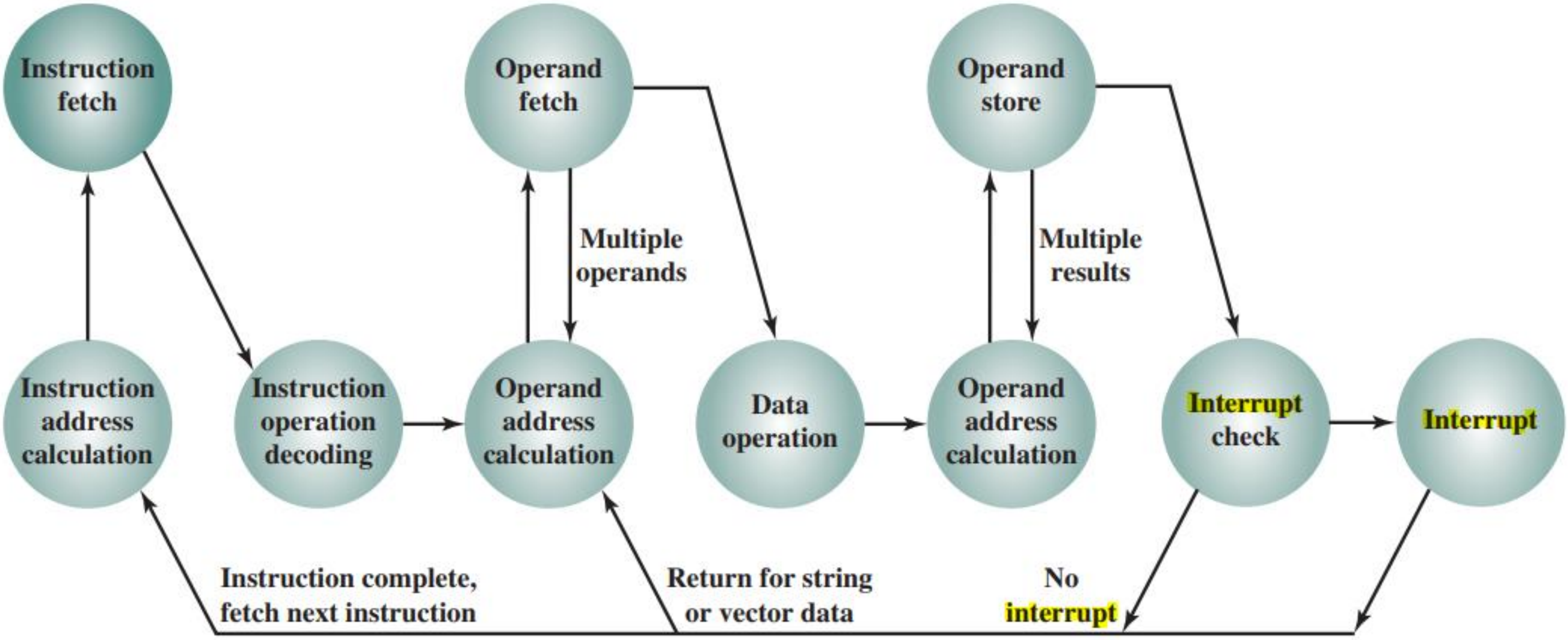
# Interrupt Cycle

- Added to instruction cycle
- Processor checks for interrupt
  - Indicated by an interrupt signal
- If no interrupt, fetch next instruction
- If interrupt pending:
  - Suspend execution of current program
  - Save context
  - Set PC to start address of interrupt handler routine
  - Process interrupt
  - Restore context and continue interrupted program

# Instruction Cycle with Interrupts (Simplified)



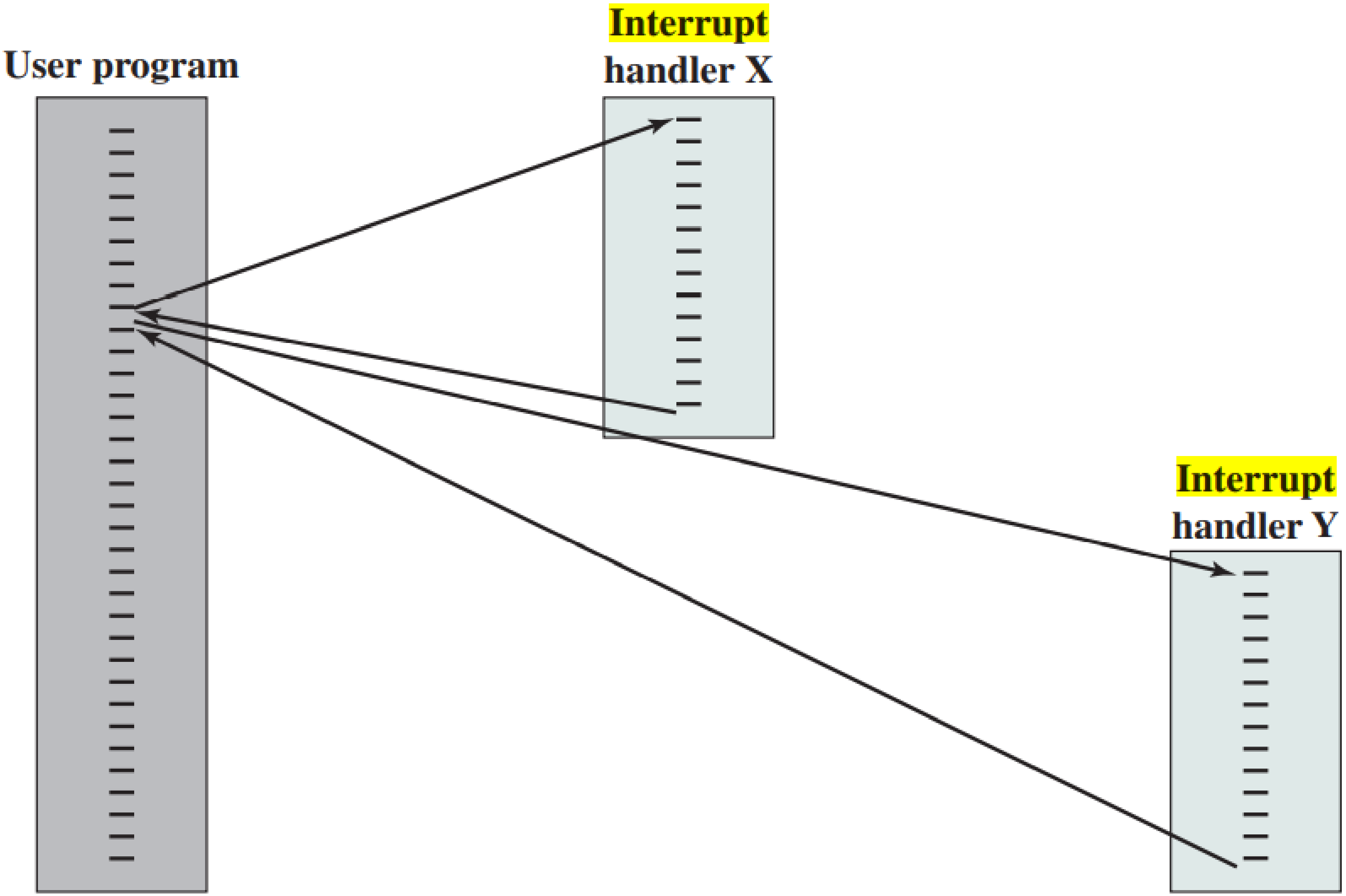
# Instruction Cycle (with Interrupts) - State Diagram


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt

# Multiple Interrupts

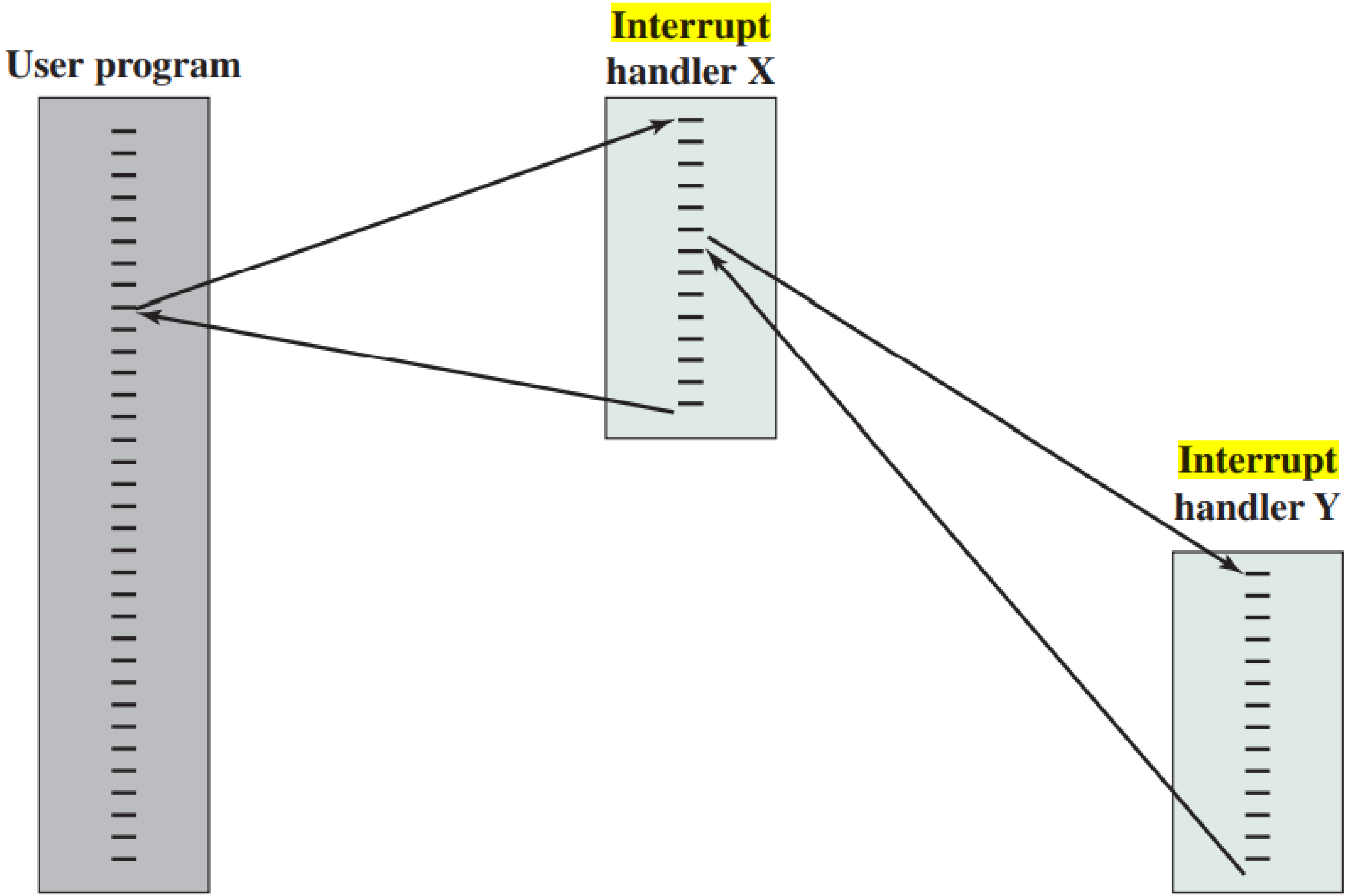
- Disable interrupts
  - Processor will ignore further interrupts whilst processing one interrupt
  - Interrupts remain pending and are checked after first interrupt has been processed
  - Interrupts handled in sequence as they occur
- Define priorities
  - Low priority interrupts can be interrupted by higher priority interrupts
  - When higher priority interrupt has been processed, processor returns to previous interrupt

# Multiple Interrupts - Sequential



(a) Sequential interrupt processing

# Multiple Interrupts - Nested



(b) Nested interrupt processing

# Interrupt Processing - Summary

- Important Concepts:
  - Polled Approach vs Interrupt Approach
  - Exceptions and Interrupt Handling through Interrupt Servicing
  - Dealing with Multiple Interrupts through Nesting, Priority and Masking
  - Instruction Cycle including Interrupt Processing
  - Interrupt Service Routine ISR
  - Concept of Stack to store CPU Status, stack pointer **$sp**, **LIFO**

# Readings

- P&H Text Book, Chapter 2 and Appendix A
- William Stallings books to read about Addressing Modes and Interrupts