

# CMP 1203

## LECTURE 9

# CPU Organization and Design

- Fetches instructions from memory
- Reads and writes data to and from memory
- Transfers data to and from I/O devices
- **3 major components**
- **Register set:**
- Depends on architecture
- General-purpose and special purpose registers
- **Arithmetic Logic Unit**
- Performs all arithmetic and logic operations
- **Control Unit**
- Fetches instructions from main memory, decodes and executes them

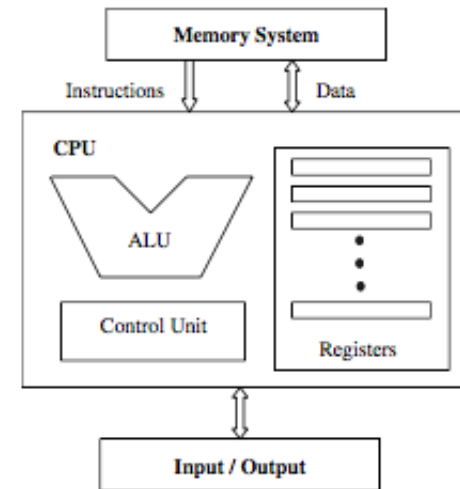


Figure 5.1 Central processing unit main components and interactions with the memory and I/O

# Typical Execution Cycle

1. Fetch next instruction to be executed (its address is got from Program Counter) from memory and stored in instruction register.
2. Decode instruction
3. Fetch operands from memory and store in CPU registers
4. Execute instruction
5. Transfer results from CPU registers to memory
  - Repeat as long as there are other instructions to be decoded
  - \*\*also routinely check for any pending interrupts and transfer to the interrupt handling routine
  - CPU actions during execution cycle are defined by micro-orders sent by CU
  - Micro orders are individual control signals sent over dedicated control lines

# Register Set

- Recall: fast memory locations within CPU that are used to create and store results of CPU operations and other calculations
- Vary from computer to computer: number of registers, types, length, usage etc
- ***Memory Access Registers***
- Memory writes and reads
- Memory Data register (MDR) and Memory Address Register (MAR)
- ***Instruction Fetching registers***
- Program counter contains address of next instruction to be fetched
- Fetched instruction loaded to IR for execution
- ***Condition Registers***
- Also called flags
- Used to maintain status information
- ***Special Purpose Address Registers***
- Index registers, Segment Pointers, Stack pointer

# Datapath

- CPU divided into data section and control section
- Control is control unit
- Data section is registers + ALU
- Internal data movements among registers and between registers and ALU done via local buses
- External data movements are between registers to memory and I/O via system bus
- Internal data movement can be via one-bus, two-bus or three-bus organizations

# One-bus Organization

- Single bus used to move data in/out
- Bus can handle one data movement within one clock cycle hence operations with more than one operand need multiple cycles to fetch all
- May need additional registers to buffer data for ALU
- Simple , cheapest BUT slow since limits amount of data transfer that can be done in same clock cycle

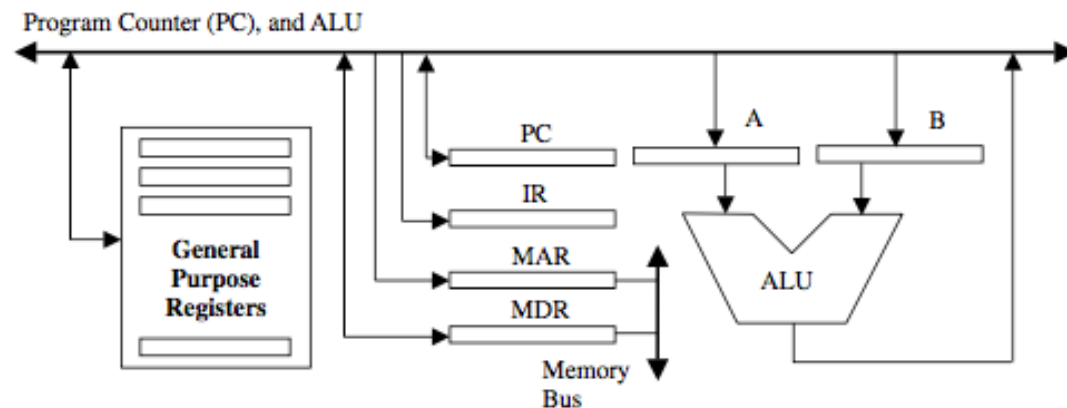
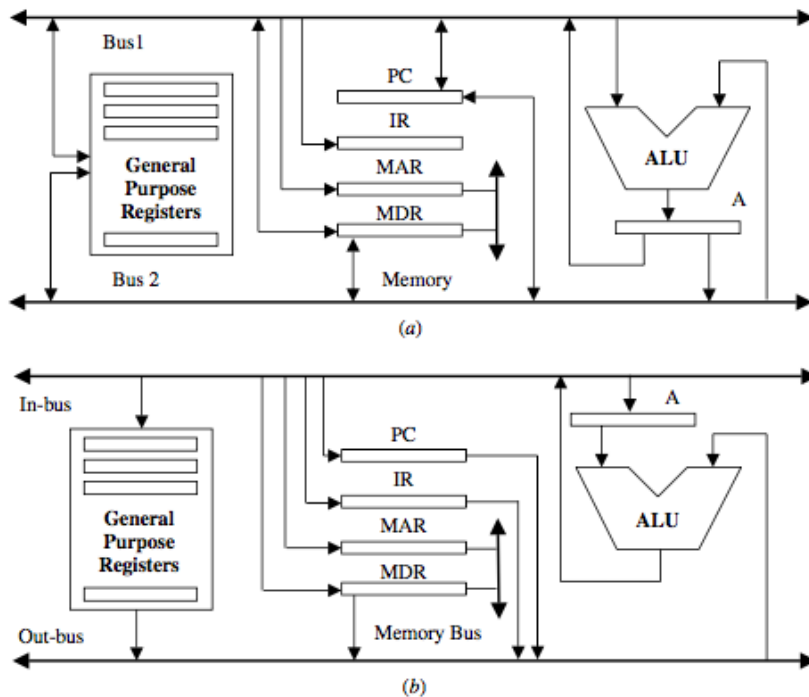


Figure 5.3 One-bus datapath

# Two-Bus Organization



**Figure 5.4** Two-bus organizations. (a) An Example of Two-Bus Datapath. (b) Another Example of Two-Bus Datapath with in-bus and out-bus

- Faster
- General purpose registers are connected to both buses
- Data can be transferred from two different registers to ALU at same time
- 2 –operand operation needs one clock cycle to fetch both
- Additional buffer register may be needed to hold ALU output when the two buses are carrying the operands

# Three-Bus Organization

- 2 buses can be used as source buses and third as destination
- Source bus move data out of registers (out-bus) and destination into registers (in-bus)
- More buses: more data that can be moved in single clock cycle BUT more complex hardware

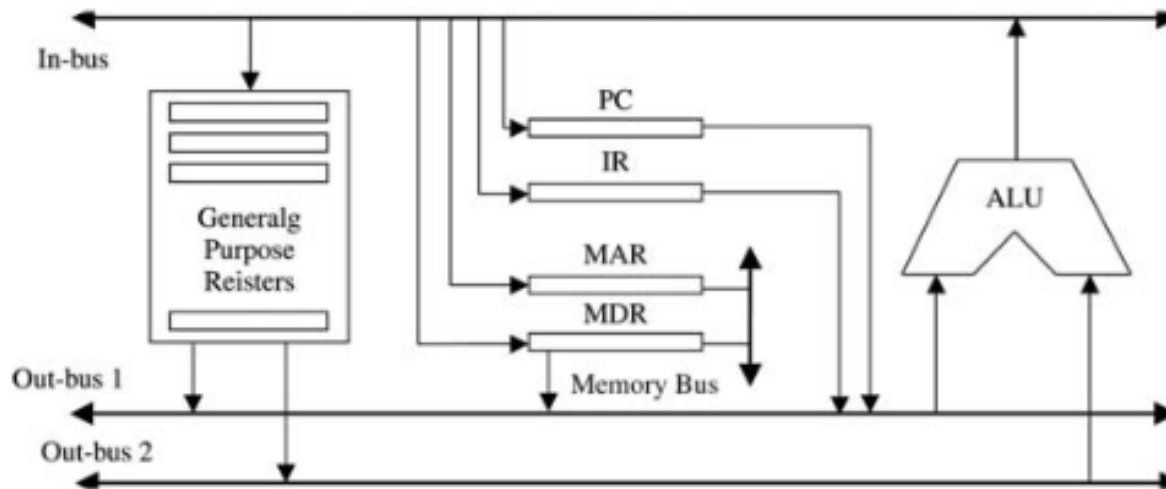


Figure 5.5 Three-bus datapath



# ALU

- All elements of computer system serve to bring data into ALU for processing and then take results back out
- Core of the computer
- Operands for arithmetic and logic operations are presented to ALU in registers and results stored in other registers
- ALU may also set flags as a result of an operation
- Review integer and floating point representation and arithmetic

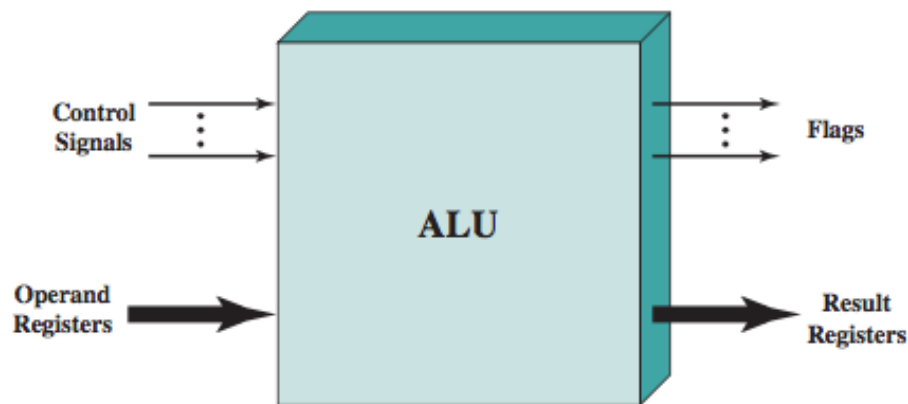


Figure 10.1 ALU Inputs and Outputs

# Instruction Set Architecture

- Why do we care about machine instructions yet we program in high level languages? : Need to understand computer architecture to write more effective programs
- **Instruction formats**
- Machine instructions consist of opcodes and 0 or more operands
- Opcode: specify operations to be executed
- Operand: specify register or memory locations of data
- Different architectures differ in number of bits allowed per instruction, number of operands allowed per instruction and types of instructions and data they can operate on

# Instruction Set Architecture

- Instruction sets are differentiated by:
  1. Operand storage
    - Data can be stored in stacks or registers
  2. Number of operands per instruction
    - 0, 1,2,3 are common
  3. Location of operand
    - Instructions can be register-to-register, register-to-memory, memory-to-memory
  4. Type of operations and whether operation can access memory or not
  5. Type and size of operands ( addresses, numbers or characters ...)

# Designing an Instruction Set

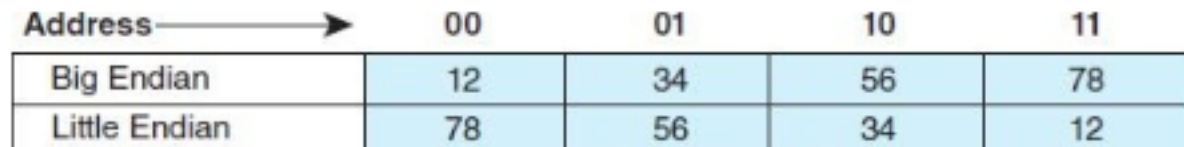
- ISAs are measured by:
- **Amount of space a program needs**
- **Complexity**
- How much decoding is needed to execute the instruction ?
- How complex are the tasks the instruction can execute?
- **Instruction length**
- **Total number of instructions**

# Design Considerations

- Short instructions occupy less space in memory and can be fetched quickly BUT limit size and number of operands and instructions
- Fixed length instructions are easy to decode but waste space
- Memory organization affects instruction format
- Addressing modes
- Number of bits in operand field: instruction can be fixed length but bits in operand can vary :” expanding opcode”
- What order should bytes be stored on byte-addressable computer? Little vs. big endian
- How many registers are required? Register organization?
- How should operands be stored on CPU?

# Little Vs. Big Endian

- Endian means a computer architecture's "byte order" i.e. how does a computer store bytes for a multiple byte data element
- Little endian: store Least significant byte at lower address
- Big endian: store most significant byte at lower address
- Most UNIX machines are big endian, PCs little endian, RISC are big endian
- E.g. storage of hex number 12345678



Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

**FIGURE 5.1** The Hex Value 12345678 Stored in Both Big and Little Endian Formats

# Little Vs. Big Endian

- Homework: Advantages and disadvantages of little and big endian.
- Any program which writes data must know how the machine orders its bytes
- Considered in software design
- Big endian: adobe photoshop, JPEG, MacPaint,
- Little endian: GIF, Paint brush, RTF
- Both: Microsoft WAV, AVI files, TIF files etc

# CPU Internal storage: Stacks Vs. Registers

- After determining byte ordering in memory, need to choose how CPU stores data
- **1. Stack architecture**
- Uses stack to execute instructions
- Operands are found on top of the stack
- Good code density but stack can't be accessed randomly so hard to make code efficient
- **2. Accumulator architecture**
- One operand in the accumulator
- Low complexity and can make instructions short
- **3. General-purpose register architecture**
- Uses general purpose registers
- Most common
- Registers faster than memory, easy for compilers to use, effective and efficient
- However, results in longer instructions because all operands must be named
- Hence long fetch and decode times



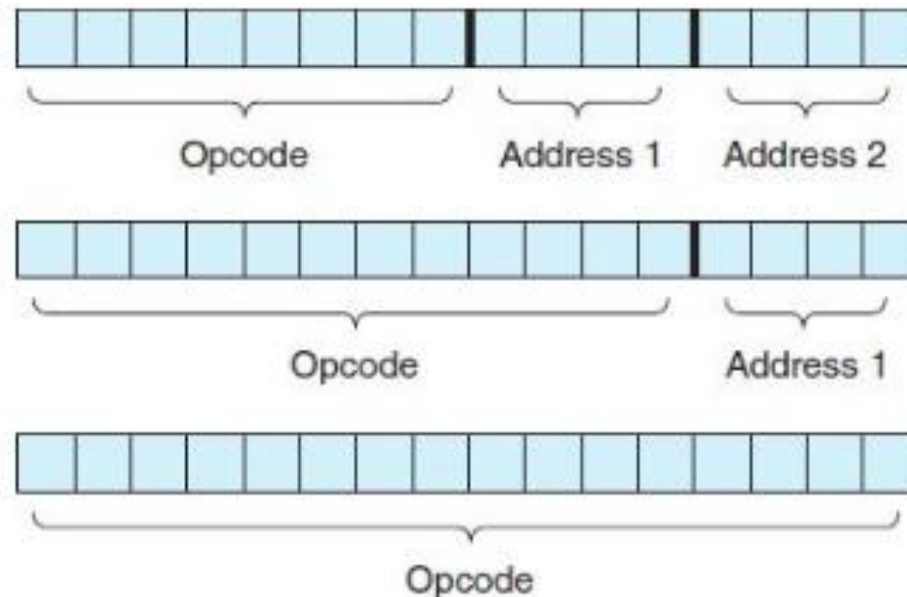
# Number of Operands and Instruction Length

- Computer architecture described by maximum number of operands, or addresses contained in each instruction
- Fixed length instructions: waste space but fast and good performance if use instruction level pipelining
- Variable length: more complex to decode but save storage space
- Typically use 0 – 3 operands
- E.g.  $Z = (X * Y) + (W * U)$

3 operands	2 operands	One operand
Mult R1, X, Y	Load R1, X	Load X
Mult R2, W, U	Mult R1, Y	Mult Y
Add Z, R2, R1	Load R2, W	Store Temp
	Mult R2, U	Load W
	Add R1, R2	Mult U
	Store Z, R1	Add Temp
		Store Z

# Expanding Opcodes

- Varying length of opcodes
- Allow short opcodes but have a way to provide longer ones when required
- E.g 16-bit instructions



Three Possibilities for a 16-Bit Instruction Format

# Types of Instructions

- **Data Movement**
- Most common
- Data moved from memory to registers, registers to registers and registers to memory
- MOVE, LOAD, STORE, PUSH, POP, EXCHANGE etc
- **Arithmetic Operations**
- Use integers and floating point numbers
- ADD, SUBTRACT, MULTIPLY, DIVIDE, INCREMENT, DECREMENT, NEGATE etc
- **Boolean Logic Instructions**
- Perform Boolean operations similar to arithmetic
- Commonly used to control I/O devices
- AND, NOT, OR, XOR, TEST, COMPARE

# Types of Instructions

- **Bit Manipulation Instructions**
- Used for setting and resetting individual bits or groups of bits within a given data word
- Include both arithmetic and logical SHIFT and ROTATE instructions
- **Input/Output Instructions**
- Vary among architectures
- Input (read) or output (write) instructions
- **Instructions for Transfer of control**
- Used to alter normal sequence of program execution
- Branches, skips, returns, program termination etc

# Types of Instructions

- **Special Purpose Instructions**
- Flag control, word/byte conversions, cache management, registers access etc
- And any other which don't fit in previous categories
- Complete instruction set is necessary for any architecture
- Designers need to ensure no duplication of instructions: each instruction must perform a unique function

# Address Modes

- Allow designer to specify where instruction operands are located
- Can specify a constant, register or a location in memory
- **Immediate addressing**
- Value to be referenced immediately follows the opcode in the instruction i.e. data is part of the instruction e.g. Load 008 loads 8 into accumulator
- Fast but limited flexibility because data to be manipulated is fixed
- **Direct addressing**
- Value to be referenced is got by specifying its memory address in the instruction e.g. Load 008 loads data in memory address 8 into AC
- Fast because data is still quickly accessible since address is specified in instruction
- More flexible because data value can vary

# Address Modes

- **Register addressing**
- Register instead of memory is used to specify the operand
- Similar to direct addressing
- **Indirect addressing**
- Bits in address field specify a memory address which is used as a pointer
- The effective address of the operand is found by going to this memory address
- E.g. Load 008 says that operand is the one whose address is stored in memory address 0X008.
- E.g. if 0x2A0 is stored in 0x008. Then the real address we need is 0x2A0
- Value found at 0x2A0 is loaded into AC

# Address Modes

- **Register indirect addressing**
- Same as previous only uses register instead of memory address to point to data
- **Homework: Based addressing, indexed addressing, stack addressing**
- How does computer know which addressing mode is supposed to be used?
- Information is specified in instruction in some way
- Multiple versions of same instruction, each version corresponds to a particular mode

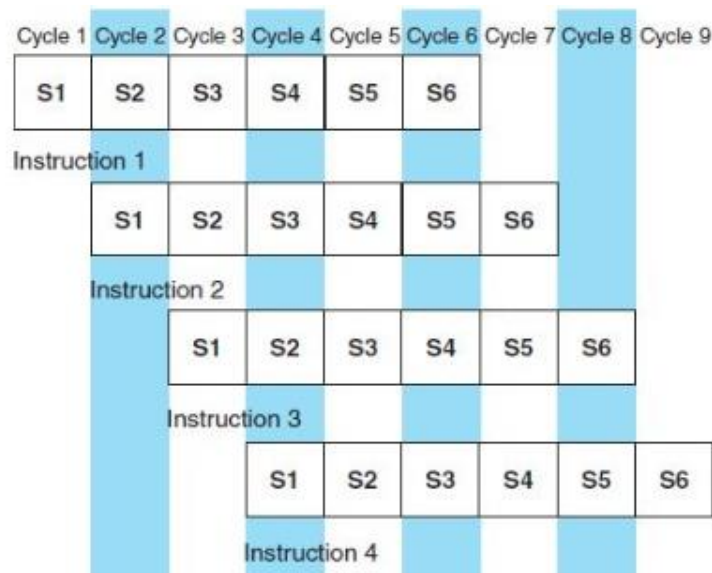


# Instruction Pipelining

- Recall fetch-decode-execute-cycle
- Each step in cycle controlled by clock pulse
- Some CPUs break these steps further into smaller steps which can be performed in parallel
- Aim is to speed up execution
- This is called ***pipelining***
- Stages need to be balanced in time or else faster stages will have significant wait time for slower ones to complete

# Instruction Pipelining

- Events for different instructions occur in parallel
- 1-fetch, 2- decode, 3-calculate, 4-operand fetch, 5-execute, 6-store
- Assuming k stage pipeline, with clock cycle time =  $t_p$ , the theoretical speedup from pipelining is



$$\text{Speedup} = \frac{k \times t_p}{t_p} = k$$

**FIGURE 5.5** Four Instructions Going through a 6-Stage Pipeline

# Instruction Pipelining

- Not all instructions require all stages e.g. instruction with no operands doesn't need operand fetch
- To simplify pipelining hardware and timing, all instructions go through all stages
- There could be different conditions that result in conflicts along pipeline
- These can delay completion of one instruction per clock cycle
- Resource conflicts
- Data dependencies
- Conditional branch statements

# Instruction Pipelining

- **Resource conflicts**
- Also called structural hazards
- E.g. if different instructions need memory access e.g. one is storing a value to memory another instruction is being fetched from memory
- Usually force fetch to wait
- Can also provide separate pathways from memory: one for data one for instructions
- **Data dependencies:** when result of one instruction not yet available is required as operand for next instruction e.g.  $X = Y + 3$ ;  $Z = 2 * X$

Time Period →	1	2	3	4	5
$X = Y + 3$	fetch instruction	decode	fetch Y	execute & store X	
$Z = 2 * X$		fetch instruction	decode	fetch X	

# Instruction Pipelining

- Can include hardware to detect which instructions need operands that are still further up the pipeline
- Hardware can insert a delay to allow for time to pass
- Can also detect conflicts and route data to special paths
- Allow compiler to reorder instructions and resolve conflict
- **Homework: how computers deal with branching issues in pipeline?**

# Instruction Level Parallelism

- Uses techniques to allow overlapping of instructions i.e. allow more than one instruction within a single program to execute concurrently
- Two types:
- One decomposes instruction into stages and overlaps these stages = pipelining
- Second allows individual instructions to overlap i.e. multiple instructions can be executed at the same time by the processor itself

# Information

- 2 more lectures
- CAT 2 16<sup>th</sup> April
- Begin studying for your exam.
- A LOT of material
- Do ALL homework problems from Lecture 1

# References/Hand out

- Null and Lobur, Chapter 5
- Stallings, Chapters 12-14
- El-Barr and El-Rewini, chapter 5 and 9