UCLA Samueli School of Engineering

# CS-M151B
# Computer Systems Architecture

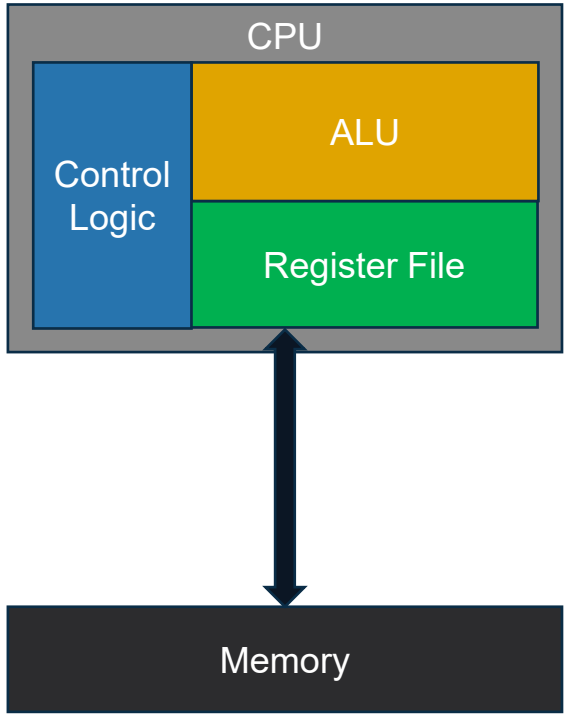
Blaise Tine
UCLA Computer Science

# Agenda

- Logistics
- ISA

# Computer Hardware Model

- Control Logic
  - Fetch and schedule Instructions
- Arithmetic Logic Unit
  - Execute instructions
- Register File
  - Array of registers
  - Fast, small storage
  - Inside the CPU
- Memory
  - Slow, large storage
  - Outside the CPU



# The Register File

- Array of registers
- Made of fast flip-flop devices
- Addressing an entry
  - Register index
- Operation
  - CPU read/write into array
  - Providing the index
- Given example
  - 4-entry 32-bit register file
  - r2 contains 0xF0F0F0F0

| ID | DATA |
|---|---|
| 0 | 0x00FF00FF |
| 1 | 0xAA008800 |
| 2 | 0xF0F0F0F0 |
| 3 | 0x00000000 |

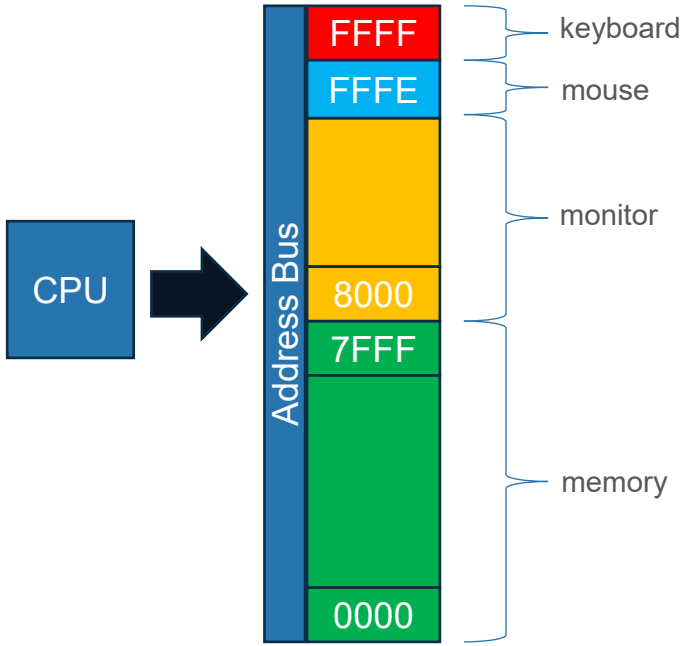
# The Main Memory

- Large array of words
- Made of slower DRAM devices
- Addressing an entry
  - word address
- Operation
  - CPU send read/write requests
  - via the system bus
- Given example
  - 256KB 32-bit memory device
  - Total entries = capacity / word size
  - 0x3 contains 0x00000001

| ADDRESS | DATA |
|---|---|
| 0 | 0x00FF00FF |
| 1 | 0xAA0088AA |
| 2 | 0xF0F0F0FF |
| 3 | 0x00000001 |
| | |
| …. | … |
| …. | … |
| 0xFFFF | 0x00000000 |

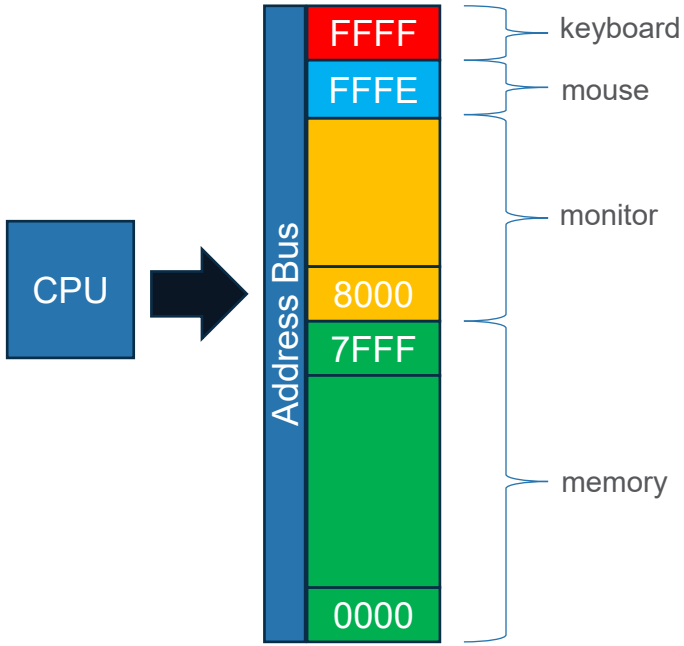
# A Memory Address Space

- A mapping contiguous range of address locations that a computer used to access main memory or other peripherals connected to the computer.
- The computer address bus carries the address to access a connected peripheral.



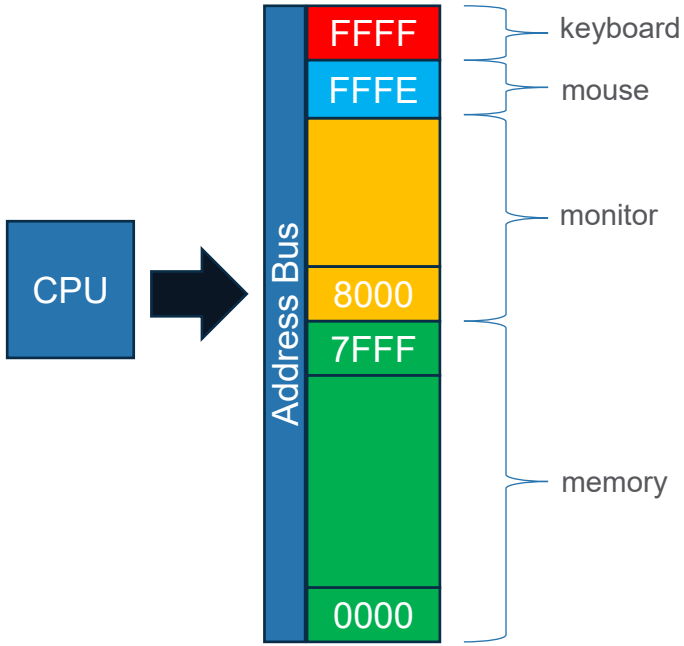
# A Memory Address Space

- Example2:
  - CPU receives load instruction:
  - e.g. LD r0, 0xFFFE
  - Load the value of 0xFFFE into r0
  - It will read the mouse input
  - Then save the value it into register r0



# A Memory Address Space

- Example2:
  - CPU receives store instruction:
  - e.g. ST r1, 0xF
  - Store the value of r1 into 0xF
  - It will write r1 into memory
  - At the memory location 0xF

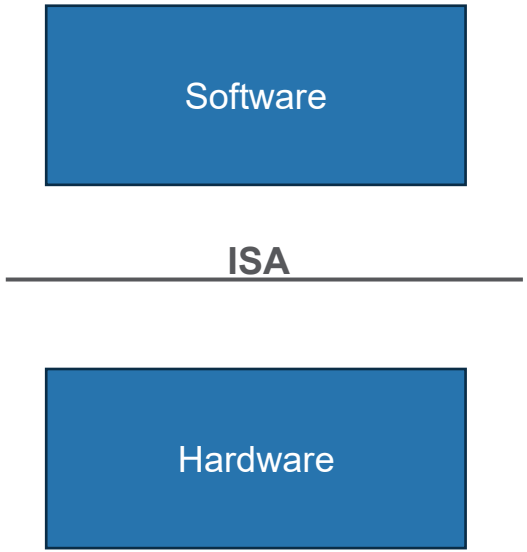


# Computer Execution Model

- Basic Instruction Format
  - Opcode
    - Type of operation
    - e.g. ADD, MUL
  - Operands
    - Source data to use
    - register identifier (e.g. r0)
    - Or memory address (e.g. 0x8000)

# What is included in the ISA?

- Instruction formats
- Opcodes description
- Data types
- Registers
- Addressing modes
- Memory alignment
- Endianness
- Control/status registers
- Others
  - Extensions



# ISA: Instruction Formats

- Enable efficient encoding and decoding
  - Encoding classification
  - Fixed fields
- Examples
  - RISC-V R-Type
    - [op7 | rd | func3 | rs1 | rs2 | funct7]
    - ADD x1, x2, x3
  - RISC-V I-Type
    - [op7 | rd | func3 | rs1 | imm12]
    - ADDI x1, x2, 10
  - What makes this efficient to decode on hardware?

# Whiteboard: Parallel Decode

# ISA: Data Type

- Data types
  - Types of data that CPU can directly process
  - Size information as well (32-bit or 64-bit)
- Common Data Types
  - Integers
  - Floating-points

# ISA: Data Type

- Extended data types
  - Half-precision floats (RISC-V Zfh)
  - Strings (x86)
  - Complex numbers (x86)
  - Vectors (Intel AVX, RISC-V Vector)
  - Matrices (Intel AMX)

# ISA: Registers

- Defines operand storage types and sizes
  - 32-bit registers
  - 64-bit registers
  - Floating-point registers
  - Double-precision registers
  - Vectors registers
  - Matrices registers

# ISA: Registers

- **Managing Architectural states**
  - **General purpose registers:** hold user data
  - **Program counter:** hold next instruction address (Branches)
    - e.g R15 is explicit in ARM
  - **Status/flag registers:** hold branches or exceptions status
    - rounding mode, division by zero
    - e.g. PSW in x86, CPSR in ARM, CSR in RISC-V

# ISA: Addressing Modes

- Addressing modes:

  Schemes to encode source operands

  1. Immediate addressing
  2. Register addressing
  3. Direct addressing
  4. Indirect addressing
  5. Displacement addressing
  6. Indexed addressing
  7. Relative addressing
  8. Stack addressing

# ISA: Addressing Modes

- **Immediate addressing**
- The operand value is directly embedded within the instruction
- **Example:**
  - ADD r1, r1, #5
- **Pros:**
  - Fast to decode
- **Cons:**
  - Limited range due to instruction size
  - Cannot change operand value at runtime

# ISA: Addressing Modes

- **Register addressing**
- The operand is located in a register
- **Example:**
  - ADD r1, r1, r2
- **Pros:**
  - Fast to decode
- **Cons:**
  - Need register file access
  - Register dependencies stall due to data hazards

# ISA: Addressing Modes

- **Direct memory addressing**
- The memory operand is directly embedded within the instruction
- **Example:**
  - LD r1, #80000000
- **Pros:**
  - Fast to decode
- **Cons:**
  - Limited range due to instruction size
  - Static address

# ISA: Addressing Modes

- **Indirect memory addressing**
- The instruction specifies a register that contains the address of the operand
- **Example:**
  - LD r0, (r1)
- **Pros:**
  - Flexible dynamic address
  - Flexible address range
- **Cons:**
  - Need register file access
  - Register dependencies stall due to data hazards

# ISA: Addressing Modes

- **Displacement addressing**
- This combines an address held in a register with an offset value from the instruction to calculate the effective address
- **Example:**
  - LD r0, #5(r1)
- **Pros:**
  - Flexibility for offset-based objects (e.g. structs)
  - Compact S/W translation => less instructions
- **Cons:**
  - Need register file access
  - Register dependencies stall due to data hazards
  - Limited offset range due to instruction size
  - Hardware complexity: extra addition

# ISA: Addressing Modes

- **Indexed addressing**
- Similar to displacement addressing but typically used with arrays. A base address is combined with an index from another register.
- **Example:**
  - LD r0, r1(r2)
- **Pros:**
  - Flexibility for array-based indexing (e.g. widgets[i])
  - Compact S/W translation => less instructions
- **Cons:**
  - Need register file access
  - Register dependencies stall due to data hazards (2x)
  - Hardware complexity: extra addition

# Whiteboard: Struct/Array addressing

**Example:**

A point p is located at address 0x400 in memory

You want to access element p->y.

Use direct, indirect, displacement, indexed addressing

```
struct Point {
    int x; // Offset 0
    int y; // Offset 4
};

struct Point *p;
int val = p->y;
```

# Attendance

- Please Fill the form with today’s keyword to register your attendance.

# ISA: Addressing Modes

- **Relative addressing**
- Often used in branch instructions, where the operand is a memory address relative to the Program Counter (PC)
- **Example:**
  - JMP 4 => Jump to memory at PC + 4
- **Pros:**
  - Flexible branching: relocatable program
  - Compact S/W translation => less instructions
- **Cons:**
  - Limited range due to instruction size limit
  - Constraint to PC base address

# ISA: Addressing Modes

- **Stack addressing**
- Operands are implicitly located on the stack.
- **Example:**
  - PUSH r0
  - POP r1
- **Pros:**
  - Simplify function calls: call stack
  - Automatic memory management: implicit allocation/deallocation
- **Cons:**
  - Restricted access pattern: top of the stack
  - Limited applications: call stack

# ISA: Encoding large Immediate

- **Variable-length instructions**
  - Use multiple fetch operations
  - e.g. loading 0xaaff into r0 using 16-bit instructions (8-bit immediate)
    - Instruction word0 = IMX r0, 0xff
    - Instruction word1 = 0xaa

# ISA: Encoding large Immediate (2)

- **Pseudo instructions**
  - Higher-level simplified assembly commands
  - e.g. loading 0xaaff into r0 using 16-bit instructions (8-bit immediate)
  - **LA 0xaaff**
    - MOV r1, 0xaa
    - SHL r1, 8  // shift left by 8 bits
    - OR r1, 0xff
    - LD r0, (r1)

# ISA: Encoding large Immediate (3)

- **Fixed offset**
  - Assume a fixed offset to construct the full value
  - For instance, using the upper half of the range
  - e.g. loading 0xaaff into r0 using 16-bit instructions (8-bit immediate)
    - LUI r0, 0xaa  // load 0xaa into upper half (0xaa00)
    - OR r0, r0, 0xff

# ISA: Memory alignment

- Memory can be described as a contiguous array of bytes
- Restrict memory accesses to addresses that are multiples of some fixed value (usually the size of the data being accessed).
- **e.g.** 4-byte alignment: can only read words at addresses 0, 4, 8, 12, 16, …

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FF | 7 | 0 | EE | EF | CC | 1B | A0 |

# ISA: Memory alignment (2)

- **Motivation:**
  - When data is aligned, it is easier to access it in a single memory operation
  - Affect performance and hardware cost

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FF | 7 | 0 | EE | EF | CC | 1B | A0 |

# ISA: Memory alignment (3)

- **Example:**
  - Assume a 32-bit CPU wants to support unaligned memory
  - But its fetch unit can only load 4 bytes of word at the time.
  - How will CPU execute LD r0, #1?
  - Answer: send two fetch requests:
    - Read w0:  EFCC1BA0
    - Read w1:  FF0700EE
    - Combine w0 & w1: EEEFCC1B

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FF | 7 | 0 | EE | EF | CC | 1B | A0 |

# ISA: Endianness

Endianness refers to the order in which bytes are arranged and accessed in computer memory

- **Big-Endian**: the most significant byte (the "big end") of a word is stored at the smallest memory address and the least significant byte at the highest.
  - e.g. word0 = 0x12345678
- **Pros**: human readable, network-friendly format

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 78 | 56 | 34 | 12 |

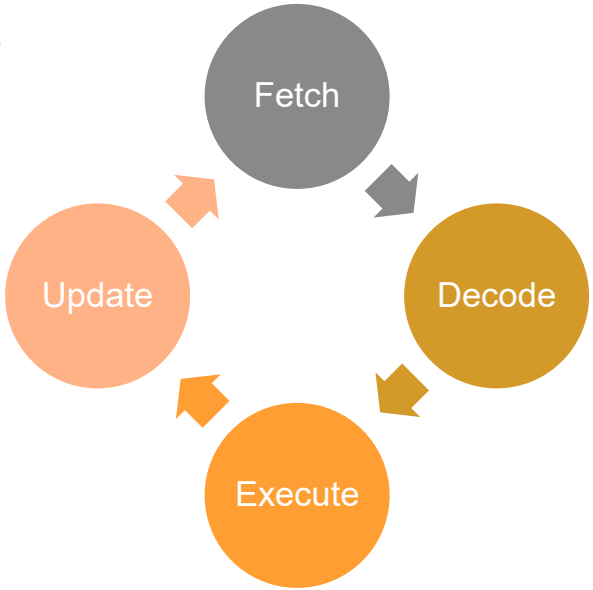
# ISA: Endianness

- **Little-Endian**: the least significant byte (the "little end") of a word is stored at the smallest memory address and the most significant byte at the highest.
  - e.g. word0 = 0x12345678
- **Pros**: arithmetic-friendly format for computation

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 12 | 34 | 56 | 78 |

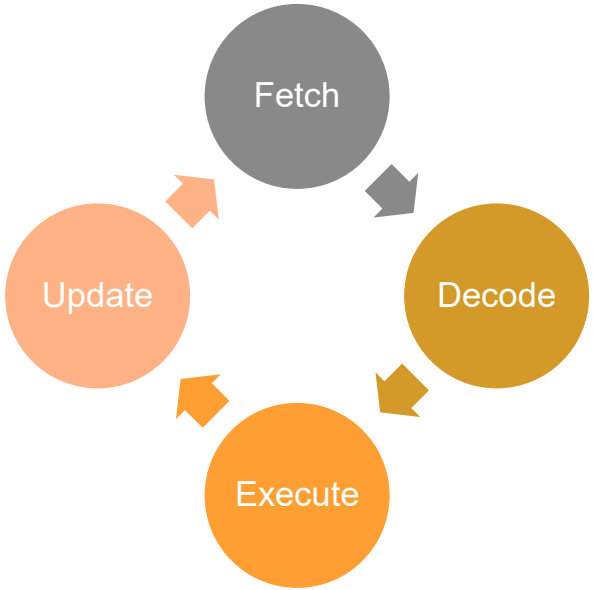
# How do you design an ISA?

- Goal?
  - Execute the program as fast as possible
- How?
  - Make all the steps execute fast
  - How do you design the ISA to achieve that?



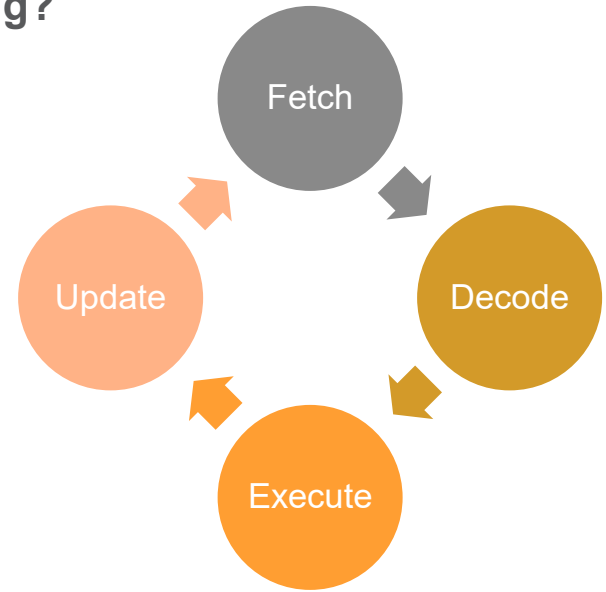
# How do you design an ISA?

- How does the ISA affect instruction Fetch?
  - Instruction size
  - Instruction alignment



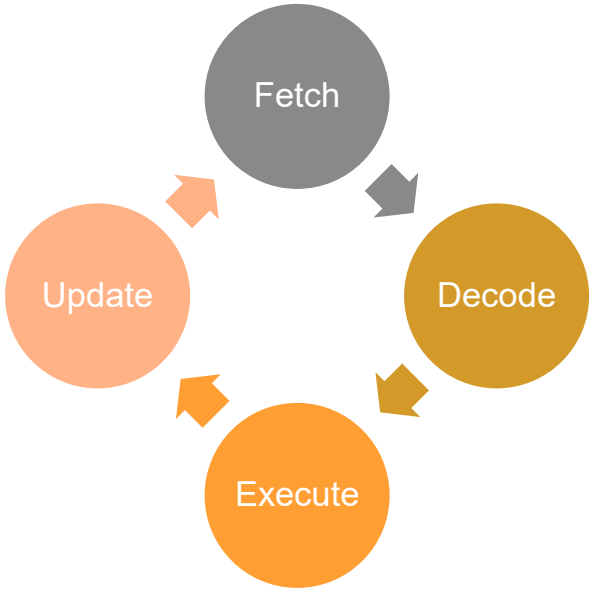
# How do you design an ISA?

- How does the ISA affect instruction Decoding?
  - Instruction formats
  - Addressing modes



# How do you design an ISA?

- How does the ISA affect Execution?
  - Opcode description
  - Registers
  - Data types
  - Addressing modes
  - Memory alignment
  - Endianness



# Example: 6502 8-bit CPU

- Usage: poplar in the 80's: Apple 1, 2, Nintendo NES, Atari 2600
  - Variable-length instruction: 1-3 bytes
    - 1 byte opcode
    - 1-2 bytes operands
  - 5 Registers: A (accumulator), X (index), Y (index), SP (stack), SR (status)

Accumulator-based architecture

- A reserved register (A) used for intermediate operations.
- e.g. LDA 0x10: load immediate 0x10 into accumulator.
  - ADC 0x3: Add immediate 3 to accumulator
- **Pros/Cons of this design?**

# Example: 6502 8-bit CPU

- Accumulator benefits?
  - Programmability
    - Simplify register allocation
    - Manual assembly development
  - Efficiency
    - Fast decode: simple instruction set
    - Fast execute: single register operand

Reference: https://en.wikibooks.org/wiki/6502_Assembly

# Q&A