Lec 6

Assembly:

slide:3

Instruction Categories

MIPS instructions are grouped into several categories based on their purpose:

1. **Arithmetic**: Handles mathematical operations (e.g., addition, subtraction).
2. **Load/Store**: Moves data between memory and registers.
3. **Jump and Branch**: Controls program flow (e.g., loops, conditional jumps).
4. **Floating Point**: Handles operations on floating-point numbers, with the help of a **coprocessor** for efficiency.
5. **Memory Management**: Handles operations related to managing memory.
6. **Special**: Miscellaneous instructions for unique tasks.

Registers

MIPS uses 32 general-purpose registers, labeled:

- **R0 - R31**: General-purpose registers for data and computation.
- **PC (Program Counter)**: Holds the address of the next instruction to execute.
- **HI and LO**: Special registers used for specific instructions, such as multiplication and division.

Instruction Formats

MIPS instructions are always **32 bits wide**, and there are three main formats:

1. **R Format**: Used for arithmetic and logical instructions.
    - Fields:
        - OP (6 bits): Operation code.
        - rs (5 bits): Source register 1.
        - rt (5 bits): Source register 2.
        - rd (5 bits): Destination register.
        - sa (5 bits): Shift amount (for shift instructions).
        - funct (6 bits): Function code for specific operations.
2. **I Format**: Used for immediate values and memory operations.
    - Fields:
        - OP (6 bits): Operation code.
        - rs (5 bits): Source register.
        - rt (5 bits): Destination register.
        - immediate (16 bits): Immediate value or address offset.

3. **J Format**: Used for jump instructions.
    - Fields:
        - OP (6 bits): Operation code.
        - jump target (26 bits): Target address to jump to.

This slide explains the concept of **endianness**, which is how data is stored in memory. It uses an analogy to breaking an egg at either the sharp end or the rounded end. Here's the breakdown:

1. Endianness Concepts

- **Little-endian (Sharp-end):**
    - Stores the **least significant byte** (LSB) first (at the lowest memory address).
    - Commonly used in x86 processors.
- **Big-endian (Rounded-end):**
    - Stores the **most significant byte** (MSB) first (at the lowest memory address).
    - Used in some network protocols and older architectures.

2. Register Values

- Two registers ($s1 and $s2) hold 32-bit binary values:
    - $s1 = 1111 1111 0000 0000 1111 1111 0000 1111
    - $s2 = 0000 1111 1111 1111 0000 1111 1111 0000 These values represent how data is structured in the CPU registers.

3. Storing Values in Memory

The values in the registers are stored in memory using load instructions:

- **Instruction 1:** lw $s1 4($t0)
    - Load the value of $s1 into memory at an offset of 4 from the address in $t0.
    - $t0 = 0x4080 (memory address).
- **Instruction 2:** lw $s2 8($t0)
    - Load the value of $s2 into memory at an offset of 8 from the address in $t0.

Why Endianness Matters

- Depending on the system's endianness (little or big), the order of bytes in memory will differ.
- For example:
    - In **little-endian**, the LSB of $s1 is stored first.
    - In **big-endian**, the MSB of $s1 is stored first.

This affects how the values appear in memory when read or processed, which is crucial for cross-platform compatibility.

Here’s how the values in the registers would appear in memory based on **endianness**.

Given Data:

- $s1 = 1111 1111 0000 0000 1111 1111 0000 1111
  - In hexadecimal: FF 00 FF 0F
- $s2 = 0000 1111 1111 1111 0000 1111 1111 0000
  - In hexadecimal: 0F FF 0F F0

Memory Layout Explanation:

Assume $t0 = 0x4080, which is the starting memory address.

- $s1 is stored at 0x4084 (4 bytes offset from $t0).
- $s2 is stored at 0x4088 (8 bytes offset from $t0).

Little-endian (Sharp-end):

In little-endian, the **least significant byte (LSB)** is stored at the lowest memory address.

Memory Address	Stored Byte (Value of $s1 and $s2)
0x4084	0F (LSB of $s1)
0x4085	FF
0x4086	00
0x4087	FF (MSB of $s1)
0x4088	F0 (LSB of $s2)
0x4089	0F
0x408A	FF
0x408B	0F (MSB of $s2)

Big-endian (Rounded-end):

In big-endian, the **most significant byte (MSB)** is stored at the lowest memory address.

Memory Address	Stored Byte (Value of $s1 and $s2)
0x4084	FF (MSB of $s1)
0x4085	00
0x4086	FF
0x4087	0F (LSB of $s1)
0x4088	0F (MSB of $s2)
0x4089	FF
0x408A	0F
0x408B	F0 (LSB of $s2)

Visual Comparison

Little-endian:

Address: 0x4084 0x4085 0x4086 0x4087

Data: 0F FF 00 FF (for $s1)

Address: 0x4088 0x4089 0x408A 0x408B

Data: F0 0F FF 0F (for $s2)

Big-endian:

Address: 0x4084 0x4085 0x4086 0x4087

Data: FF 00 FF 0F (for $s1)

Address: 0x4088 0x4089 0x408A 0x408B

Data: 0F FF 0F F0 (for $s2)

Key Takeaways:

- **Little-endian**: Bytes are reversed in memory (LSB first).
- **Big-endian**: Bytes are stored in the same order as in the register (MSB first).

(slide pg: 14,15)

This process describes how a processor calculates the target address for a branch instruction when a branch is taken. Here's a simpler breakdown:

1. **Update the Program Counter (PC)**: The processor starts by updating the Program Counter (PC) to point to the next instruction. It does this by adding 4 to the current PC value, resulting in PC+4PC + 4PC+4.
2. **Get the Offset from the Branch Instruction**: The branch instruction includes a 16-bit offset that indicates how far to jump if the branch condition is true. This offset is the "distance" from the current PC to the target address.
3. **Convert the Offset to 32-Bits**:
   - To make this 16-bit offset usable, it's first extended to an 18-bit number by adding two zeros to the end, making it a bit longer.
   - Then, it's *sign-extended* to 32 bits, meaning that if the original offset was negative, the extension will keep it negative, and if positive, it stays positive.
4. **Calculate the Target Address**: The modified 32-bit offset is then added to PC+4PC + 4PC+4 to find the target address.
5. **Update the PC if the Branch Condition is True**: If the branch condition (like checking if a comparison is true) is satisfied, this new target address is loaded into the PC. Otherwise, the PC just continues to the next instruction.

This slide is explaining how branch instructions like bne (branch if not equal) are assembled in machine code. Here's a breakdown of each part to help you understand:

1. **Assembly Code**

- The assembly code on the slide shows:
- bne $s0, $s1, Label
- add $s3, $s0, $s1
- Label:
- The bne $s0, $s1, Label instruction checks if $s0 and $s1 are not equal. If they aren't equal, it branches (jumps) to the address marked by Label.
- If $s0 and $s1 are equal, it moves to the next instruction (add $s3, $s0, $s1).

2. **Machine Format of bne**

- This format represents how the bne instruction is stored in memory in binary.
- The layout includes:
  - **op** (operation code): specifies that this is a bne operation.

- **rs** and **rt**: registers involved in the comparison (in this case, $s0 and $s1).
- **16-bit offset**: a 16-bit number that represents the "distance" from the current instruction to Label.

3. **16-bit Offset Calculation**

- The offset is used to calculate where to branch (jump) if the condition is met.
- The offset is provided as a 16-bit value but needs to be converted to a 32-bit address.

4. **Steps for Branch Calculation** (Explained in the “Remember” section):

- **PC Update**: After fetching the bne instruction, the Program Counter (PC) is updated to point to the next instruction, calculated as PC + 4.
- **Offset Conversion**:
  - The 16-bit offset in the bne instruction is first shifted left by 2 (adding two zeros) to make it an 18-bit number.
  - This is done because instructions are word-aligned (4 bytes), so we need to multiply the offset by 4 to get the correct byte address.
  - **Sign-extension**: The offset is sign-extended to 32 bits, which preserves its positive or negative sign, making it usable as a branch target address.

5. **Final Calculation**

- The 32-bit offset is then added to PC + 4 to get the final branch target address.
- If the condition ($s0 != $s1) is true, the PC is updated to this new address, and execution jumps to Label.
- If the condition is false, execution proceeds to the next instruction (add $s3, $s0, $s1).