Lecture 2 Notes

Parallel Processing CS471

1 Instruction Set Architectures

The instruction set architecture (ISA) is the 'model' that defines how software programs can interact with the processor.

1.1 Reduced Instruction Set Computer (RISC)

RISC architectures are a class of instruction sets that use a small, highly optimized set of instructions, each of which can be executed in a single clock cycle. RISC architectures are characterized by:

- All instructions are typically the same size.
- All operations on data apply to data in registers and typically change the entire register.
- The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, i.e., all instructions operate on the registers except for the load/store instructions used to access memory.

RISC Implementation of `C = A + B`

The characteristics of a RISC architecture are evident when we look at how it would deal with a simple addition task like `C = A + B`:

```
LW R1, A
LW R2, B
ADD R3, R1, R2
SW R3, C
```

Due to the register-register architecture of RISC, we first have to load (`LW`) our operands into registers, perform the add operation (`ADD`), then store the result back to memory (`SW`).

1.2 Complex Instruction Set Computer (CISC)

CISC architectures are another class of instruction sets that use a (*relatively*) larger more complex set of instructions, allowing for execution of complex operations with a single instruction. CISC implementations might require more specialized or complex hardware, and their instructions may require more than a single clock cycle.

1.3 MIPS Instruction Set

The MIPS is a RISC instruction set developed by MIPS Computer Systems. MIPS uses thirty-two 32-bit general purpose registers and a 32-bit program counter (PC).

Instruction Format

MIPS instructions are typically 32-bit and are divided into three types: R (register), I (immediate), and J (jump). Every instruction starts with a 6-bit opcode. In addition to the opcode, R-type instructions specify three registers (rd, rs, rt), a shift amount field, and a function field; I-type instructions specify two registers and a 16-bit immediate value; J-type instructions follow the opcode with a 26-bit jump target.

Type	-31-		format (bits)			-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Figure 1: The three formats used for the core instruction set.

Instruction Set

Instruction Description	Example Usage	Instruction Type	Format	Opcode	Function
Load Upper Immediate	`lui $t0, 0x1234`	Data Movement	I	15	-
Add	`add $t0, $t1, $t2`	Arithmetic	R	0	32
Subtract	`sub $t0, $t1, $t2`	Arithmetic	R	0	34
Set Less Than	`slt $t0, $t1, $t2`	Arithmetic	R	0	42
Add Immediate	`addi $t0, $t1, 10`	Arithmetic	I	8	-
Set Less Than Immediate	`slti $t0, $t1, 10`	Arithmetic	I	10	-
Bitwise AND	`and $t0, $t1, $t2`	Logical	R	0	36
Bitwise OR	`or $t0, $t1, $t2`	Logical	R	0	37
Bitwise XOR	`xor $t0, $t1, $t2`	Logical	R	0	38
Bitwise NOT	`nor $t0, $zero, $t1`	Logical	R	0	39
Bitwise AND Immediate	`andi $t0, $t1, 0xFF`	Logical	I	12	-
Bitwise OR Immediate	`ori $t0, $t1, 0xFF`	Logical	I	13	-
Bitwise XOR Immediate	`xori $t0, $t1, 0xFF`	Logical	I	14	-
Load Word	`lw $t0, 0($t1)`	Memory Access	I	35	-
Store Word	`sw $t0, 0($t1)`	Memory Access	I	43	-
Jump	`j label`	Control Flow	J	2	-
Jump Register	`jr $ra`	Control Flow	R	0	8
Branch on Less Than Zero	`bltz $t0, label`	Control Flow	I	1	0
Branch if Equal	`beq $t0, $t1, label`	Control Flow	I	4	-
Branch if Not Equal	`bne $t0, $t1, label`	Control Flow	I	5	-
Jump and Link	`jal label`	Control Flow	J	3	-
System Call	`syscall`	System	R	0	12

Figure 2: Selected MIPS Instruction Set

MIPS Supported Addressing Modes

MIPS supports a limited number of addressing modes:

Immediate Addressing

Used in instructions like `addi`, `andi`, `ori`, and `stli`. The operand's immediate value `imm` is specified in the instruction itself. For example:

```
andi $t0, $t1, 0xFF
```

This instruction performs the bitwise AND of the value in register `$t1` and the value `0x00FF`.

Register Addressing

Used in Arithmetic and logical operations, and data movement. The operand's value is located in the specified register. For example:

```
or $t0, $t1, $t2
```

This instruction performs the bitwise OR of the value in register `$t1` and the value in register `$t2`.

Base/Displacement Addressing

Used in memory access instructions like `lw`, `sw`, `lb`, and `sb`. The operands value is stored in the calculated effective address which is calculated as:

$$\text{Effective Address} = \text{Base Register Value} + \text{Offset}$$

For example:

```
lw $t0, 4($t1)
```

This instruction stores `$t0` at the address `($t1 + 8)`.

PC-Relative Addressing

Used for branch instructions like `beq`, `bne`, and `bltz`. The effective address is determined by adding a signed intermediate offset to the current PC value.

For example:

```
beq $t0, $t1, L
```

This instruction branches to `PC + L` if `$t0 == $t1`

Note 1. Since the immediate offset is sign-extended (signed) it has to be shifted by two bits making the actual immediate address: `PC + 4 + (L << 2)`

2 Pipelining Implementation Using RISC

2.1 MIPS Pipeline

A MIPS pipeline is usually divided into five stages, allowing up to five instructions to be processed simultaneously, each at a different stage of execution. The stages are:

1. **Instruction Fetch (IF)** - Fetch the instruction from memory using the Program Counter (PC).
2. **Instruction Decode/Register Fetch (ID)** - Decode the instruction to determine the operation and operands, then read the necessary registers from the register file.
3. **Execute/Address Calculation (EX)** - Perform the required computation using the ALU for arithmetic and logic instructions or calculate the effective addresses for memory operations.
4. **Memory Access (MEM)** - Access data memory for load and store instructions.
5. **Write Back (WB)** - Write the result back to the register file.

With our pipeline stages defined, we can identify the basic hardware components involved in the pipeline:

- **Program Counter (PC)** - Holds the address of the next instruction to fetch. The PC increments after each instruction fetch (unless modified by a branch or jump instruction).
- **Instruction Memory** - Stores the programs instructions. We fetch instructions using their address (provided by the PC).
- **Register File** - This 'file' contains the thirty-two 32 bit general purpose registers. The register file allows for reading two registers and writing one register simultaneously.
- **Arithmetic Logic Unit (ALU)** - Performs all arithmetic and logical operations.
- **Data Memory** - Stores the data used by the program. Data memory is accessed during the MEM sage for load and store operations.
- **Control Unit (CU)** - Generates control signals based on the instruction opcode and function code. The CU directs data flow and operations of the ALU, memory, and registers.
- **Pipeline Registers** - In order to allow simultaneous processing of instructions, each pipeline stage is separated by a pipeline register. The pipeline registers hold intermediate data and control signals between different stages.

Note 2. Not all the hardware components, or even the pipeline stages are involved in every operation. *See examples below.*

Arithmetic Instruction in MIPS Pipeline

The instruction:

```
add $t0, $t1, $t2
```

Would move through the pipeline as follows:

1. **IF** - Fetch `add` instruction
2. **ID** - Decode instruction; read `$t1` and `$t2` from register file.
3. **EX** - ALU adds content of `$t1` and `$t2`.
4. **MEM** - No operation (dummy state).
5. **WB** - Write result to `$t0` in the register file.

Load/Store Instruction in MIPS Pipeline

The instruction:

```
lw $t0, 8($t1)
```

Would move through the pipeline as follows:

1. **IF** - Fetch `lw` instruction
2. **ID** - Decode instruction; read `$t1` and sign-extend offset `8`
3. **EX** - ALU calculates effective address `$t1 + 8`.
4. **MEM** - Read data from memory at effective address.
5. **WB** - Write result to `$t0`.

Branch and Jump Instructions in MIPS Pipeline

The instruction:

```
beq $t0, $t1, L
```

Would move through the pipeline as follows:

1. **IF** - Fetch `beq` instruction
2. **ID** - Decode instruction; read `$t1` and `$t0`. Compute branch target address.
3. **EX** - ALU compares `$t1` and `$t0`. Determine if branch is taken.
4. **MEM** - No operation.
5. **WB** - No operation.

Based on the way we handle instructions we can build the MIPS pipeline with components for all the operations needed.

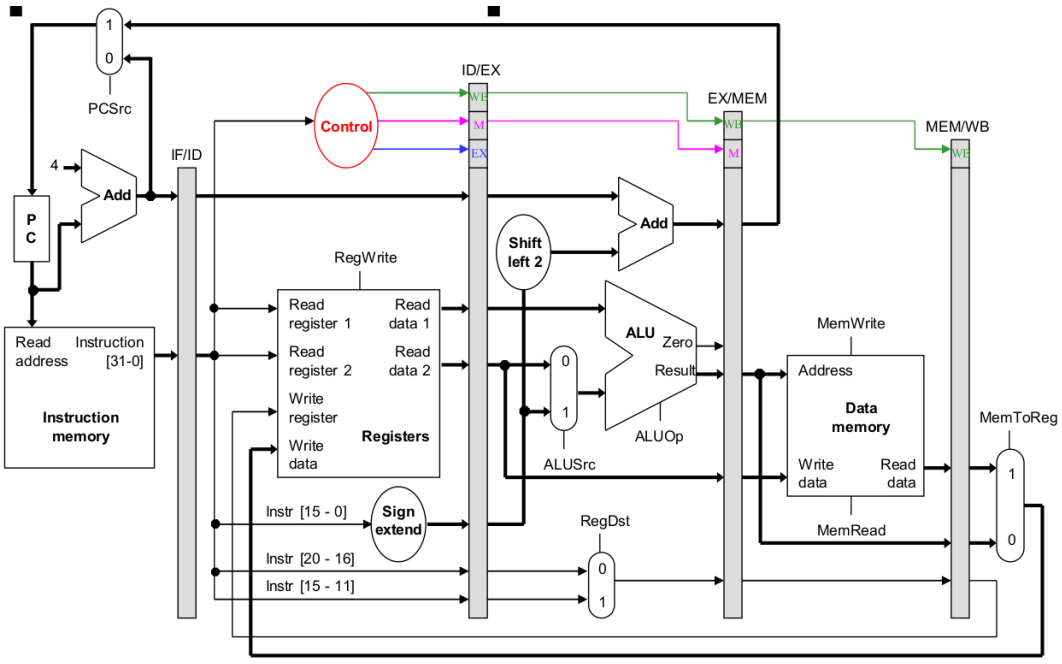

Figure 3: Circuit diagram of a 5-stage MIPS pipeline. Notice the additional ALU units and Sign Extend components necessary for handling different addressing modes.

However, usually we deal with a more abstracted view of the above diagram for simplicity.

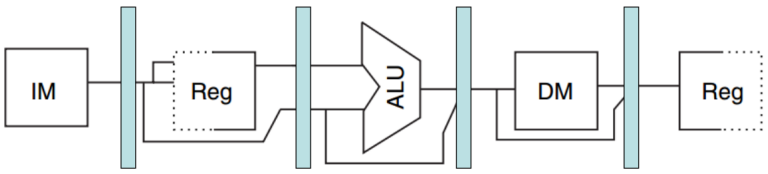

Figure 4: Abstracted/simplified view of 5-stage MIPS pipeline.

References