Advanced Computer Architectures - Notes

260236

March 2024

Preface

Every theory section in these notes has been taken from two sources:

- Computer Architecture: A Quantitative Approach. [1]

About:

GitHub repository

Contents

1 Basic Concepts

This section is designed to review old concepts that are fundamental to this course.

1.1 Pipelining

1.1.1 MIPS Architecture

MIPS (**Microprocessor without Interlocked Pipelined Stages**) is a family of Reduced Instruction Set Computer (RISC). It is based on the concept of **executing only simple instruction in a reduced basic cycle to optimize the performance of CISC**[1] CPUs.

MIPS is a **load-store architecture** (or a register–register architecture), which means it is an Instruction Set Architecture (ISA[2]) that divides **instructions into two categories**:

- **Memory access** (load and store between memory and registers; load data from memory to registers; store data from registers to memory):

$$\text{Memory} \xrightarrow{\textbf{load}} \text{Registers}$$
$$\text{Memory} \xleftarrow{\textbf{store}} \text{Registers}$$

- ALU operations (which only occur between registers).

Finally, **MIPS** is also a **Pipeline Architecture**. It means that **it can execute a performance optimization technique based on overlapping the execution of multiple instructions derived from a sequential execution flow**.

[1] CISC processors use simple and complex instructions to complete any given task. Instead, the RISC processor uses the approach of increasing internal parallelism by executing a simple set of instructions in a single clock cycle (see more here).

[2] Instruction Set Architecture (ISA) is a part of the abstract model of a computer, which generally defines how software controls the CPU.

Reduced Instruction Set of MIPS Processor

The instruction set of the MIPS processor is the following:

- ALU instructions:
  - **Sum** between **two registers**:

    ```
    add $s1, $s2, $s3        # $s1 <- $s2 + $s3
    ```

    Take the values from `s2` and `s3`, make the sum and save the result on `s1`.
  - **Sum** between **register and constant**:

    ```
    addi $s1, $s1, 4         # $s1 <- $s1 + 4
    ```

    Take the value from `s1`, make the sum between `s1` and `4`, and save the result on `s1`.
- Load/Store instructions:
  - **Load**

    ```
    lw $s1, offset ($s2)     # $s1 <- Memory[$s2 + offset]
    ```

    From the `s2` register, calculate the index on the memory with the `offset`, take the value and store it in the `s1` register.
  - **Store**

    ```
    sw $s1, offset ($s2)     # Memory[$s2 + offset] <- $s1
    ```

    Take the value from the `s1` register, take the value from the `s2` register, calculate the index on the memory with the `offset`, and store the value taken from s1 in the memory.
- Branch instructions to control the instruction flow:
  - **Conditional branches**
    Only if the condition is true (branch on equal):

    ```
    beq $s1, $s2, L1     # if $s1 == $s1 then goto L1
    ```

    Only if the condition is false (branch on not equal):

    ```
    bne $s1, $s2, L1     # if $s1 != $s2 then goto L1
    ```
  - **Unconditional jumps**. The branch is always taken.
    Jump:

    ```
    j L1     # jump to L1
    ```

    Jump register:

    ```
    jr $s1  # jump to address contained in $s1
    ```

Formats of MIPS 32-bit Instructions

The previous instructions are divided into **three types**:

- Type **R (Register)**: ALU instructions.
- Type **I (Immediate)**: Load/Store instructions and Conditional branches.
- Type **J (Jump)**: Unconditional jumps instructions.

Every instruction **starts with a 6-bit opcode**. In addition to the opcode:

- R-type instructions specify:
  - **Three registers**: `rs`, `rt`, `rd`
  - A **shift amount field**: `shamt`
  - A **function field**: `funct`
- I-type instructions specify:
  - **Two registers**: `rs`, `rt`
  - **16-bit immediate value**: `offset/immediate`
- J-type instructions specify:
  - **26-bit jump target**: `address`

	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
	31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	offset/immediate		
J	op	address				

Figure 1: MIPS 32-bit architecture.

Scan (or click) the QR code below to view the table in high quality:

Phases of execution of MIPS Instructions

Every instruction in the MIPS subset can be implemented in **at most 5 clock cycles (phases)** as follows:

1. **Instruction Fetch (IF)**
   (a) **Send** the **content** of Program Counter (PC) register to the Instruction Memory (IM);
   (b) **Fetch** the current **instruction** from Instruction Memory;
   (c) **Update** the Program Counter to the **next sequential address** by adding the value 4 to the Program Counter (4 because each instruction is 4 bytes!).
2. **Instruction Decode and Register Read (ID)**
   (a) Make the **fixed-filed recording** (**decode the current instruction**);
   (b) **Read** from the Register File (RF) of one or two registers corresponding to the registers specified in the instruction fields;
   (c) Sign-extension of the offset field of the instruction in case it is needed.
3. **Execution (EX)**. The ALU operates on the operands prepared in the previous cycle depending on the instruction type (see more details after this list):
   - **Register-Register** ALU instructions: ALU **executes the specified operation** on the operands read from the Register File.
   - **Register-Immediate** (Register-Constant) ALU instructions: ALU executes the specified operation on the first operand read from Register File and the sign-extended immediate operand.
   - **Memory Reference**: ALU adds the base register and the offset to calculate the **effective address**.
   - **Conditional Branches**: ALU compares the two registers read from Register File and computes the possible **branch target address** by adding the sign-extended offset to the incremented Program Counter.
4. **Memory Access (ME)**. It depends on the operation performed:
   - Load. Instructions require a **read access to the Data Memory using the effective address**.
   - Store. Instruction require a **write access to the Data Memory using the effective address** to write the data **from the source register read from the Register File**.
   - Conditional branches can **update the content of the Program Counter** with the branch target address, if the conditional test yielded true.

5. **Write-Back (WB)**. It depends on the operation performed:
   (a) **Load** instructions **write the data read from memory in the destination register of the Register File**.
   (b) **ALU** instructions **write the ALU results into the destination register of the Register File**.

Execution (EX) details

- **Register-Register ALU instructions**. Given the following pattern (where `op` can be the operators `add/addi` (+) or `sub/subi` (-), but not `mult` (×) or `div` (÷) because they required some special registers and therefore more phases):

```
op $x, $y, $z   # e.g. op=add  =>  $x <- $y + $z
```

  Cost: 4 clock cycles

  1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
  2. Fixed-Field Decoding and read from Register File the registers: `y` and `z`;
  3. Execution (EX), ALU performs the operation `op` (`$ y op $ z`);
  4. Write-Back (WB), ALU writes the result into the destination register `x`.

- **Memory Reference**
  - Load. Given the following pattern:

```
lw $x, offset ($y)  # $x <- M[$y + offset]
```

    Cost: 5 clock cycles

    1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
    2. Fixed-Field Decoding and read of Base and register `y` from Register File (RF);
    3. Execution (EX), ALU adds the base register and the offset to calculate the effective address: `y + offset`;
    4. Memory Access (ME), read access to the Data Memory (DM) using the effective (`y + offset`) address;
    5. Write-Back (WB), write the data read from memory in the destination register of the Register File (RF) `x`.

- Store. Given the following pattern:

```
sw $x, offset ($y)  # M[$y + offset] <- $x
```

Cost: 4 clock cycles

1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
2. Fixed-Field Decoding and read of Base register `y` and source register `x` from Register File (RF);
3. Execution (EX), ALU adds the base register and the offset to calculate the effective address: `y + offset`;
4. Memory Access (WB), write the data read from memory in the destination register of the Register File (RF) `M(y + offset)`.

- **Conditional Branch**. Given the following pattern:

```
beq $x, $y, offset
```

Cost: 4 clock cycles

1. Instruction Fetch (IF) and update the Program Counter (next sequential address);
2. Fixed-Field Decoding and read of source registers `x` and `y` from Register File (RF);
3. Execution (EX), ALU compares two registers `x` and `y` and compute the possible branch target address by adding the sign-extended offset to the incremented Program Counter: `PC + 4 + offset`;
4. Memory Access (ME), update the content of the Program Counter with the branch target address (we assume that the conditional test is true).

1.1.2 Implementation of MIPS processor - Data Path

Implementing a MIPS processor isn't difficult. On the following page we show three different diagrams: the first is a very high level data path to allow the reader to understand how it works; the second is more detailed, but without the CU (Control Unit); the third is the complete data path and it also includes the CU (in red).

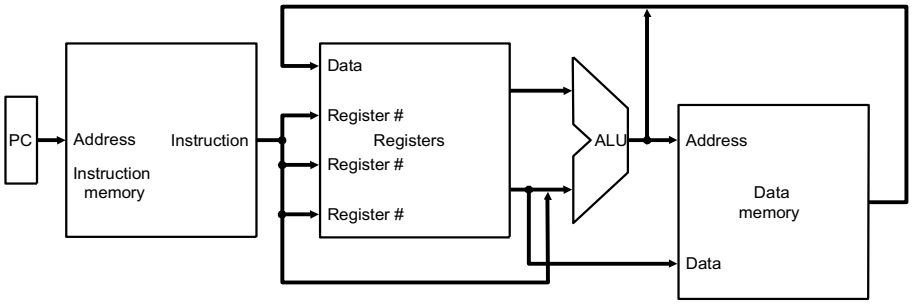

Figure 2: A basic implementation of MIPS data path.

Scan (or click) the QR code below to view the figure 2 in high quality:

Two notes:

- The **Instruction Memory** (read-only memory) is separated from **Data Memory**.
- The 32 general-purpose register are organized in a **Register File** (RF) with 2 read ports and 1 write port.

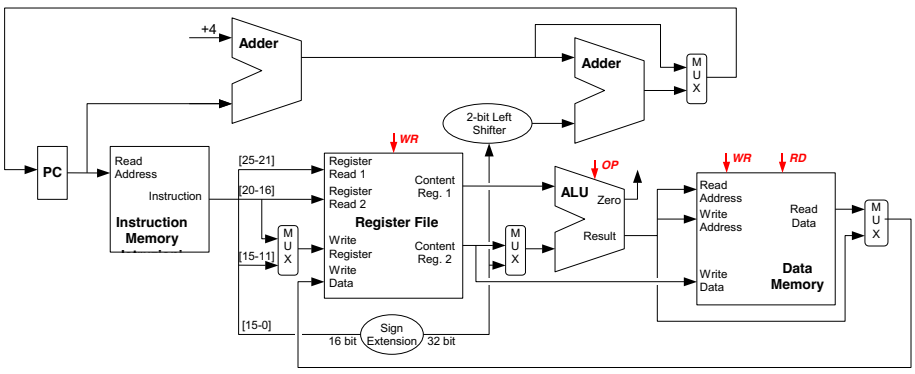

Figure 3: An implementation of MIPS data path (no Control Unit).

Scan (or click) the QR code below to view the figure 3 in high quality:

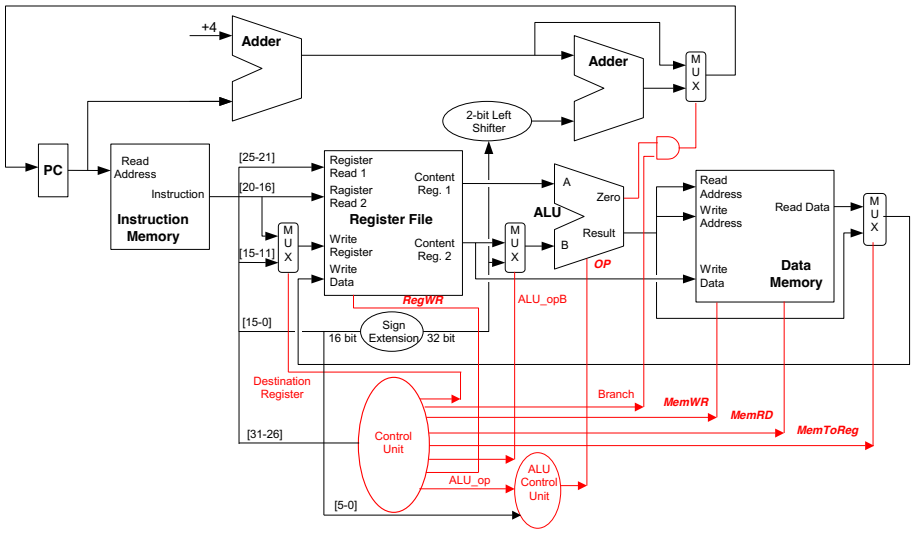

Figure 4: A complete implementation of MIPS data path.

Scan (or click) the QR code below to view the figure 4 in high quality:

References

[1] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

Index