CS2100 Computer Organisation

The Processor: Datapath
(AY2016/2017) Semester 2
Road Map: Part II

- Performance
- Assembly Language
- Processor: Datapath
- Processor: Control
- Pipelining
- Cache

- Processor Datapath
  - Generic Execution Stages
  - MIPS Execution Stages
  - Constructing Datapath
Building a Processor: Datapath & Control

- Two major components for a processor:
  - Datapath
    - Collection of components that process data
    - Performs the arithmetic, logical and memory operations
  - Control
    - Tells the datapath, memory, and I/O devices what to do according to program instructions
MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
  - **Arithmetic and Logical operations**
    - add, sub, and, or, addi, andi, ori, slt
  - **Data transfer instructions**
    - lw, sw
  - **Branches**
    - beq, bne
  - Shift instructions (**sll, srl**) and J-type instructions (**j**) will not be discussed:
    - Left as exercises 😊
Recap: Instruction Execution Cycle

- **Fetch:**
  - Get instruction from memory
  - Address is in Program Counter (PC) Register

- **Decode:**
  - Find out the operation required

- **Operand Fetch:**
  - Get operand(s) needed for operation

- **Execute:**
  - Perform the required operation

- **Result Write (Store):**
  - Store the result of the operation
MIPS Instruction Executions

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
  - The standard steps are performed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Result Write</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $3, $1, $2</code></td>
<td><strong>standard</strong></td>
<td></td>
<td><code>Result = opr1 + opr2</code></td>
<td><code>Result stored in $3</code></td>
</tr>
<tr>
<td><code>lw $3, 20( $1 )</code></td>
<td></td>
<td><strong>standard</strong></td>
<td><code>MemAddr = opr1 + opr2</code></td>
<td><code>Memory data stored in $3</code></td>
</tr>
<tr>
<td><code>beq $1, $2, ofst</code></td>
<td></td>
<td><strong>standard</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **opr** = Operand
- **MemAddr** = Memory Address
- **ofst** = offset

Processor: Datapath
# 5-STAGE MIPS EXECUTION

**Design changes:**
- Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
- Split *Execute* into **ALU** (Calculation) and **Memory Access**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode &amp; Operand Fetch</th>
<th>ALU</th>
<th>Memory Access</th>
<th>Result Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read inst. at [PC]</td>
<td>oRead [$1] as opr1</td>
<td>Result = opr1 + opr2</td>
<td>Use MemAddr to read from memory</td>
<td>Result stored in $3</td>
</tr>
<tr>
<td></td>
<td>oRead [$2] as opr2</td>
<td>oRead [$1] as opr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oUse 20 as opr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oRead [$1] as opr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oRead [$2] as opr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Taken = (opr1 == opr2)?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target = PC + Label*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Processor: Datapath**

- **ADD $3, $1, $2**
- **lw $3, 20( $1 )**
- **beq $1, $2, label**
Let's Build a MIPS Processor

- What we are going to do:
  - Look at each stage closely, figure out the requirements and processes
  - Sketch a high level block diagram, then zoom in for each elements
  - With the simple starting design, check whether different type of instructions can be handled:
    - Add modifications when needed

➔ Study the design from the viewpoint of a designer, instead of a "tourist" 😊
Fetch Stage: Requirements

Instruction Fetch Stage:

1. Use the Program Counter (PC) to fetch the instruction from memory
   - PC is implemented as a special register in the processor

2. Increment the PC by 4 to get the address of the next instruction:
   - How do we know the next instruction is at PC+4?
   - Note the exception when branch/jump instruction is executed

Output to the next stage (Decode):

- The instruction to be executed
Element: Instruction Memory

- Storage element for the instructions
  - Recall: sequential circuit
  - Has an internal state that stores information
  - Clock signal is assumed and not shown

- Supply instructions given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the right ➔
Element: Adder

- Combinational logic to implement the addition of two numbers

- **Inputs:**
  - Two 32-bit numbers $A$, $B$

- **Output:**
  - Sum of the input numbers, $A + B$

- Just a 32-bit version of the adder discussed in first part of the course 😊
The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
  - How can it work properly?

- **Magic of clock:**
  - PC is read during the first half of the clock period and it is updated with PC+4 at the next rising clock edge

![Diagram of the processor datapath showing the reading and updating of the PC over time.](image-url)
Decode Stage: Requirement

- **Instruction Decode Stage:**
  - Gather data from the instruction fields:
    1. Read the **opcode** to determine instruction type and field lengths
    2. Read data from all necessary registers
      - Can be two (e.g. **add**), one (e.g. **addi**) or zero (e.g. **j**)

- **Input from previous stage (Fetch):**
  - Instruction to be executed

- **Output to the next stage (Execute):**
  - Operation and the necessary operands
Decode Stage: Block Diagram

Fetch Stage → Inst. → Decode Stage:
- Register Number
  - 5: Read register 1
  - 5: Read register 2
  - 5: Write register
- Register File
  - Read data 1
  - Read data 2
  - Operands
- Data

Execute Stage

Collection of registers, known as register file

Processor: Datapath
Element: Register File

- A collection of 32 registers:
  - Each 32-bit wide and can be read/written by specifying register number
  - Read at most two registers per instruction
  - Write at most one register per instruction

- **RegWrite** is a control signal to indicate:
  - Writing of register
  - 1 (True) = Write, 0 (False) = No Write
Decode Stage: R-Type Instruction

add $8, $9, $10

Notation:
Inst [Y:X] = bits X to Y in Instruction

Processor: Datapath
Decode Stage: I-Type Instruction

**Instruction:**

\[ \text{addi } \$21, \$22, -50 \]

**Problems:**
- Destination \( \$21 \) is in the "wrong position"
- **Read Data 2** is an immediate value, not from register
Decode Stage: Choice in Destination

**addi $21, $22, -50**

**Solution (Wr. Reg. No.):**
Use a **multiplexer** to choose the correct write register number based on instruction type.

**RegDst:**
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number.
Recap: Multiplexer

- **Function:**
  - Selects one input from multiple input lines

- **Inputs:**
  - $n$ lines of same width

- **Control:**
  - $m$ bits where $n = 2^m$

- **Output:**
  - Select $i^{th}$ input line if control=$i$

Control=0 $\rightarrow$ select $\text{in}_0$
Control=3 $\rightarrow$ select $\text{in}_3$
**Solution (Rd. Data 2):**
Use a multiplexer to choose the correct operand 2. Sign extend the 16-bit immediate value to 32-bit.
Decode Stage: Load Word Instruction

- Try it out: "lw $21, -50($22)"
- Do we need any modification?
Decode Stage: **Branch Instruction**

- Example: "\texttt{beq }\$9, \$0, 3"
  - Need to calculate branch outcome and target at the same time!
  - We will tackle this problem in the ALU Stage 😊
Decode Stage: Summary

Processor: Datapath
**ALU Stage: Requirement**

- **Instruction ALU Stage:**
  - ALU = Arithmetic-Logic Unit
  - Perform the real work for most instructions here
    - Arithmetic (e.g. `add`, `sub`), Shifting (e.g. `sll`), Logical (e.g. `and`, `or`)
    - Memory operation (e.g. `lw`, `sw`): Address calculation
    - Branch operation (e.g. `bne`, `beq`): Perform register comparison and target address calculation

- **Input from previous stage (Decode):**
  - Operation and Operand(s)

- **Output to the next stage (Memory):**
  - Calculation result
ALU Stage: Block Diagram

- Decode Stage
- ALU
- Memory Stage

Operands → ALU → ALU result

Logic to perform arithmetic and logical operations
Element: Arithmetic Logical Unit

- **ALU (Arithmetic-logical unit)**
  - Combinational logic to implement arithmetic and logical operations

- **Inputs:**
  - Two 32-bit numbers

- **Control:**
  - 4-bit to decide the particular operation

- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero

### ALU Control Function Table

<table>
<thead>
<tr>
<th>ALUcontrol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>slt</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
**ALU Stage: Non-Branch Instructions**

- We can handle non-branch instructions easily:

```
add $8, $9, $10
```

---

**Processor: Datapath**

![Datapath Diagram]

**ALUcontrol**:
Set using opcode + funct field (more in next lecture)
ALU Stage: Branch Instructions

- Branch instruction is harder as we need to perform two calculations:

- Example: "beq $9, $0, 3"

  1. Branch Outcome:
     - Use ALU unit to compare the register
     - The 1-bit "isZero?" signal is enough to handle equal / not equal check (how?)

  2. Branch Target Address:
     - Introduce additional logic to calculate the address
     - Need PC (from Fetch Stage)
     - Need Offset (from Decode Stage)
Complete ALU Stage

**Example Instruction:** "beq $9, $0, 3"

**PCSrc:** Control Signal to select between (PC+4) or Branch Target
Memory Stage: Requirement

- Instruction **Memory Access Stage:**
  - Only the load and store instructions need to perform operation in this stage:
    - Use memory address calculated by ALU Stage
    - Read from or write to data memory
  - All other instructions remain idle
    - Result from ALU Stage will pass through to be used in Result Store stage if applicable

- Input from previous stage (**ALU**):
  - Computation result to be used as memory address (if applicable)

- Output to the next stage (**Result Write**):
  - Result to be stored (if applicable)
Memory Stage: Block Diagram

ALU Stage

Result

MemWrite

MemRead

Result Store Stage

Address

Read Data

Write Data

Data Memory

Memory which stores data values

Processor: Datapath
Element: **Data Memory**

- **Storage element for the data of a program**

- **Inputs:**
  - Memory Address
  - Data to be written (Write Data) for store instructions

- **Control:**
  - Read and Write controls; only one can be asserted at any point of time

- **Output:**
  - Data read from memory (Read Data) for load instructions
Memory Stage: Load Instructions

- Only relevant parts of Decode & ALU Stage are shown

lw $21, -50($22)
Memory Stage: Store Instructions

- Need **Read Data 2 (Decode)** as the **Write Data**

\[ \text{sw } $21, -50 ($22) \]
Memory Stage: Non-Memory Instructions

- Add a multiplexer to choose the result to be stored

```plaintext
add $8, $9, $10
```
Result Write Stage: Requirement

Instruction **Register Write Stage:**

- Most instructions write the result of some computation into a register
  - Examples: arithmetic, logical, shifts, loads, set-less-than
  - Need destination register number and computation result
- Exceptions are stores, branches, jumps:
  - There are no result to be written
  - These instructions remain idle in this stage

Input from previous stage (**Memory**):

- Computation result either from memory or ALU
Result Write Stage: Block Diagram

- Result Write stage has no additional element:
  - Basically just route the correct result into register file
  - The *Write Register* number is generated way back in the *Decode* Stage
Result Write Stage: Routing

\[ \text{add } \$8, \$9, \$10 \]
The Complete Datapath!

- We have just finished "designing" the datapath for a subset of MIPS instructions:
  - Shifting and Jumping are not supported

- Check your understanding:
  - Take the complete datapath and play the role of controller:
    - See how supported instructions are executed
    - Figure out the correct control signals for the datapath elements

- Coming up next: **Control** (Lecture #15)
Instruction Memory

Instruction

Address

opcode 31:26

rs 25:21

rt 20:16

rd 15:11

shamt 10:6

funt 5:0

000000

01001

01010

01000

00000

10000

Inst [25:21]

Inst [20:16]

Inst [15:11]

Inst [15:0]

PC

Add

4

MUX

Add

Left Shift 2-bit

MUX

PCSrc

Registers

RR1

RD1

RR2

WR

RD2

WD

RegWrite

RegDst

Sign

Extend

ALUcontrol

is0?

ALU

ALUSrc

ALU result

MemWrite

Data Memory

Address

Write

Read

Data

Read

Data

MemToReg

MemRead

Instruction Memory

Instruction

Address

opcode 31:26

rs 25:21

rt 20:16

rd 15:11

shamt 10:6

funt 5:0

000000

01001

01010

01000

00000

10000

Inst [25:21]

Inst [20:16]

Inst [15:11]

Inst [15:0]

PC

Add

4

MUX

Add

Left Shift 2-bit

MUX

PCSrc

Registers

RR1

RD1

RR2

WR

RD2

WD

RegWrite

RegDst

Sign

Extend

ALUcontrol

is0?

ALU

ALUSrc

ALU result

MemWrite

Data Memory

Address

Write

Read

Data

Read

Data

MemToReg

MemRead

Processor: Datapath
Reading Assignment

- The Processor: Datapath and Control
  - 3rd edition: Chapter 5 Sections 5.1 – 5.3
  - 4th edition: Chapter 4 Sections 4.1 – 4.3
End