Designing a Single Cycle Datapath

CPS 104
Lecture 12

Outline of Today’s Lecture

- Homework #4 Due Thursday
- MIPS Simulator due April 14
- Second Mid-term end of March
- Reading Ch 5.1-5.3
- Where are we with respect to the Big picture?
- The Steps of Designing a Processor
- Datapath and timing for Reg-Reg Operations
- Datapath for Logical Operations with Immediate
- Datapath for Load and Store Operations
- Datapath for Branch and Jump Operations

Review: Integer Multiplication & Division

Multiplication
- Series of Shift and Add

Booth’s Algorithm
- (Current, Previous) bits of Multiplier:
  - 0,0: middle of string of 0s; do nothing
  - 0,1: end of a string of 1s; add multiplicand
  - 1,0: start of string of 1s; subtract multiplicand
  - 1,1: middle of string of 1s; do nothing
- Shift Product/Multiplier right 1 bit (as before)

Division
- Series of Shift and Subtract

Review: FP Addition

Example:
- $1.610 \times 10^{-1} + 9.999 \times 10^{1}$

Step 1:
- Align decimal point: $0.016 \times 10^{1} + 9.999 \times 10^{1}$

Step 2:
- Add: $10.015 \times 10^{1}$

Step 3:
- Normalize: $1.0015 \times 10^{2}$

Step 4:
- Round: $1.002 \times 10^{2}$

May need to repeat steps 3 and 4 if result not normal after rounding. (renormalization)

Review: FP Multiplication

1. Add biased exponents, subtract bias
2. Multiply significands
3. Normalize product
4. Round significand
5. Compute sign of product

$.5 \times .75 = 1.000x2^{-1} \times 1.100x2^{-2}$

Review: Rounding

- Rounding with Guard & Round bits

Example:
- $2.35 \times 10^{-1} + 2.34 \times 10^{1}$, using 3 significant digits

Align decimal points (exponents, shift smaller)
- $2.34$
- $0.225$

Guard 5, Round 6

$2.305$

- Round: $2.37 \times 10^{3}$
- Without guard & round bits, result: $2.36 \times 10^{3}$
- Error of 1 Unit in the least significant position
- Why 2 bits?
  - Product could have leading 0, so shift left when normalizing
What is Computer Architecture?

- Coordination of levels of abstraction

```
Application
  ▼ Operating System
  ▼ Compiler
  ▼ Firmware
  ▼ CPU
  ▼ Memory
  ▼ I/O system
  ▼ Instruction Set Architecture, Memory, I/O
  ▼ Hardware
```

- Under a set of rapidly changing technology Forces

The Big Picture: Where are We Now?

• The Five Classic Components of a Computer

```
Processor
  ▼ Control
  ▼ Memory
  ▼ Input
  ▼ Output
```

Today's Topic: Datapath Design

Datapath Design

° How do we build hardware to implement the MIPS instructions?

° Add, LW, SW, Beq, Jump

The MIPS Instruction Formats

° All MIPS instructions are 32 bits long. The three instruction formats:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>11</td>
<td>26</td>
<td>16</td>
<td>0</td>
<td>26</td>
<td>0</td>
</tr>
</tbody>
</table>
```

° The different fields are:

  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

The MIPS Subset (We can’t implement them all!)

```
<table>
<thead>
<tr>
<th>Type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>OR IMM</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>BRANCH</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>11</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>
```

° The Hardware “Program”

```
Instruction Fetch ➔ Instruction Decode ➔ Operand Fetch ➔ Execute ➔ Result Store ➔ Next Instruction
```

How do I build the hardware to implement the MIPS instructions and their sequencing?
Combinational Logic Elements (Basic Building Blocks)

- **Adder**
  - Sums and carries

- **MUX**
  - Selects inputs A or B

- **ALU**
  - Outputs include sum, carry, result, and zero

---

Storage Element: Register (Basic Building Block)

- **Register**
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
    - Write Enable:
      - negated (0): Data Out will not change
      - asserted (1): Data Out will become the same as Data In.

---

Storage Element: Register File

- **Register File consists of 32 registers:**
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW

- **Register is selected by:**
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable is 1

- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after “access time.”

---

Storage Element: Idealized Memory

- **Memory (idealized)**
  - One input bus: Data In
  - One output bus: Data Out

- **Memory word is selected by:**
  - Write Enable = 0: Address selects the word to put on the Data Out bus
  - Write Enable = 1: Address selects the memory word to be written via the Data In bus

- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after “access time.”

---

An Abstract View of the Implementation

- **Instruction Address**
- **Ideal Instruction Memory**
- **Data Address**
- **Ideal Data Memory**
- **Data Out**

---

Clocking Methodology

- **All storage elements are clocked by the same clock edge**
- Cycle Time >= CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- Longest delay path = critical path
An Abstract View of the Critical Path

° Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”

The Steps of Designing a Processor

° Instruction Set Architecture => Register Transfer Language
° Register Transfer Language =>
  - Datapath components
  - Datapath Interconnect
° Datapath components => Control signals
° Control signals => Control logic

Overview of the Instruction Fetch Unit

° The common RTL operations
  - Fetch the instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”

RTL: The ADD Instruction

° add rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - PC <- PC + 4 Calculate the next instruction’s address

RTL: The Load Instruction

° lw rt, rs, imm16
  - mem[PC] Fetch the instruction from memory
  - Addr <- R[rs] + SignExt(imm16) Calculate the memory address
  - R[rt] <- Mem[Addr] Load the data into the register
  - PC <- PC + 4 Calculate the next instruction’s address
### RTL: The Subtract Instruction

- **Sub: rd, rs, rt**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Register operation
- **rs, rt**: Source registers
- **PC**: Program counter
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **PC**: Program counter

### RTL: The OR Immediate Instruction

- **ori rt, rs, imm16**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value

### RTL: The Load Instruction

- **lw rt, rs, imm16**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value

### Datapath for Logical Operations with Immediate

- **R[rt] <- R[rs] or ZeroExt[imm16]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value

### Datapath for Register-Register Operations

- **R[rd] <- R[rs] - R[rt]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers

### Datapath for Register-Register Operations

- **R[rd] <- R[rs] op R[rt]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers

### Datapath for Logical Operations with Immediate

- **R[rt] <- R[rs] or ZeroExt[imm16]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value

### Datapath for Register-Register Operations

- **R[rd] <- R[rs] - R[rt]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **PC**: Program counter
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **PC**: Program counter

### Datapath for Logical Operations with Immediate

- **R[rt] <- R[rs] or ZeroExt[imm16]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value

### Datapath for Register-Register Operations

- **R[rd] <- R[rs] - R[rt]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **PC**: Program counter
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **PC**: Program counter

### Datapath for Logical Operations with Immediate

- **R[rt] <- R[rs] or ZeroExt[imm16]**
- **mem[PC]**: Fetch the instruction from memory
- **rd**: Destination register
- **rs, rt**: Source registers
- **imm16**: Immediate value
Datapath for Load Operations

Example: `lw rt, rs, imm16`

- `R[t] <- Mem[R[rs] + SignExt[imm16]]`

## RTL: The Store Instruction

Example: `sw rt, rs, imm16`

- `mem(PC)`
- `Addr <- R[rs] + SignExt[imm16]`
- `Mem[Addr] <- R[rt]`
- `PC <- PC + 4`

## RTL: The Branch Instruction

Example: `beq rs, rt, imm16`

- `mem(PC)`
- `Cond <- R[rs] - R[rt]`
- `if (COND eq 0) PC  <- PC + 4 + ( SignExt(imm16) x 4 )`
- `else PC  <- PC + 4`

## Binary Arithmetic for the Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: `PC<31:0> = PC<31:0> + 4`
  - Branch operation: `PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4`
- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: `PC<31:2> = PC<31:2> + 1`
  - Branch operation: `PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]`
  - In either case: `Instruction-Memory-Address = PC<31:2> concat “00”`
Next Address Logic: Expensive and Fast Solution

- Using a 30-bit PC:
  - Sequential operation: \( \text{PC}_{31:2} = \text{PC}_{31:2} + 1 \)
  - Branch operation: \( \text{PC}_{31:2} = \text{PC}_{31:2} + 1 + \text{SignExt}[\text{Imm16}] \)
- In either case: Instruction-Memory-Address = \( \text{PC}_{31:2} \text{ concat } 00 \)

RTL: The Jump Instruction

- \( \text{mem[PC]} \)
  - Fetch the instruction from memory
- \( \text{PC} \leftarrow \text{PC} + 4_{31:28} \text{ concat target}_{25:0} \text{ concat } 00 \)
  - Calculate the next instruction’s address

Putting it All Together: A Single Cycle Datapath

- We have everything except control signals.