Review: Sequential Logic + (Non-)blocking assignments

Must be careful mixing zero-time blocking assignments and edge-triggering: Probably won’t do what you expect when connecting it to other things!

```verilog
module FF (CLK,Q,D);
    input D, CLK;
    output Q, reg Q;
    always @ (posedge CLK)
        Q <= D;
endmodule // FF
```

Good: Doesn’t output until “after edge”

```verilog
module FF (CLK,Q,D);
input D, CLK;
output Q; reg Q;
always @ (posedge CLK)
    Q = #5 D;
endmodule // FF
```

Good: Outputs 5 units “after edge”

```verilog
module FF (CLK,Q,D);
    input D, CLK;
    output Q, reg Q;
    always @ (posedge CLK)
        #5 Q = D;
endmodule // FF
```

Probably Not what you Expect:
- Hold time of 5 units
- glitches < 5 units ignored

Review: MULTIPLY HARDWARE Version 3

° 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg (shift right), (0-bit Multiplier reg)

```
Multiplicand

32-bit ALU

Shift Right

Product [Multiplier]

64 bits

Control

```

° 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

```
Divisor

32-bit ALU

Shift Left

Remainder [Quotient]

64 bits

Write

Control

```

° Multiplication and Division can use same hardware!
Review: Booth’s Algorithm

Alternate representation

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>0001111100</td>
<td>sub (1)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>0001111000</td>
<td>none (0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>0001111000</td>
<td>add (1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>0001111000</td>
<td>none (0)</td>
</tr>
</tbody>
</table>

Examples (8 bits):

- 3 = 11111101 ⇒ 000000000111 = -4 + 2 - 1
- 14 = 00001110 ⇒ 00010010 = 16 - 2
- 23 = 00010111 ⇒ 00111001 = 32 - 16 + 8 - 1

MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>$1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>$1,$2,$3</td>
<td>$1 = $2 ^ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>$1,$2,$3</td>
<td>$1 = <del>(</del>$2</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>$1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg. constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>$1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>$1,$2,10</td>
<td>$1 = ~$2 ^ 10</td>
<td>Logical XOR reg. constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>$1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>shift right logical</td>
<td>$1,$2,10</td>
<td>$1 = $2</td>
<td>&gt;&gt; 10</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>$1,$2,10</td>
<td>$1 = $2</td>
<td>&gt;&gt; 10</td>
</tr>
<tr>
<td>shift left logical</td>
<td>$1,$2,3</td>
<td>$1 = $2</td>
<td>&lt;&lt; 3</td>
</tr>
<tr>
<td>shift right logical</td>
<td>$1,$2,3</td>
<td>$1 = $2</td>
<td>&gt;&gt; 3</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>$1,$2,3</td>
<td>$1 = $2</td>
<td>&gt;&gt; 3</td>
</tr>
</tbody>
</table>

Shifters

Two kinds:

- **Logical (RIGHT OR LEFT)**—value shifted in is always “0”
  - “0” → msb lsb “0”

- **Arithmetic**—(RIGHT ONLY), sign extend
  - msb lsb “0”

Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Barrel Shifter

Technology-dependent solutions: transistor per switch

Diagram showing the barrel shifter with inputs A6, A5, A4, A3, A2, A1, and A0, and outputs D3, D2, D1, and D0.
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
  - Processor
    - Control
    - Datapath
  - Memory
  - Input
  - Output

- Today’s Topic: Design a Single Cycle Processor
  - Machine design
  - Arithmetic (L4-6)
  - Inst. set design (L1-2)
  - Technology (L3)

The Big Picture: The Performance Perspective

- Performance of a machine is determined by:
  - CPI
    - Instruction count
    - Clock cycle time
    - Clock cycles per instruction

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction

- Today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: Long cycle time

How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements
   - the meaning of each instruction is given by the register transfers
   - datapath must include storage element for ISA registers
     - possibly more
   - datapath must support each register transfer

2. Select set of datapath components and establish clocking methodology

3. Assemble datapath meeting the requirements

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic

The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
    - 31 26 21 16 11 6 0
    - op  rs rt rd shamt funct
    - 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits
  - I-type
    - 31 26 21 16 0
    - op  rs rt immediate
    - 6 bits 5 bits 5 bits 16 bits
  - J-type
    - 31 26
    - op  target address
    - 6 bits 26 bits

- 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
Step 1a: The MIPS-lite Subset for today

- **ADD and SUB**
  - addU rd, rs, rt
  - subU rd, rs, rt

- **OR Immediate**
  - ori rt, rs, imm16

- **LOAD and STORE Word**
  - lw rt, rs, imm16
  - sw rt, rs, imm16

- **BRANCH**
  - beq rs, rt, imm16

Logical Register Transfers

- **RTL gives the meaning of the instructions**

- **All start by fetching the instruction**
  \[
  \text{op} | \text{rs} | \text{rt} | \text{rd} | \text{shamt} | \text{funct} = \text{MEM}[\text{PC}]
  \]
  \[
  \text{op} | \text{rs} | \text{rt} | \text{Imm16} = \text{MEM}[\text{PC}]
  \]

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>( R[rd] \leftarrow R[rs] + R[rt]; \quad \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>SUBU</td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; \quad \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>ORi</td>
<td>( R[rt] \leftarrow R[rs]</td>
</tr>
<tr>
<td>LOAD</td>
<td>( R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext(Imm16)}]; \quad \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[R[rs] + \text{sign_ext(Imm16)}] \leftarrow R[rt]; \quad \text{PC} \leftarrow \text{PC} + 4 )</td>
</tr>
<tr>
<td>BEQ</td>
<td>( \text{if} ( R[rs] == R[rt] ) \text{ then PC} \leftarrow \text{PC} + 4 + \text{sign_ext(Imm16)}</td>
</tr>
<tr>
<td></td>
<td>( \text{else PC} \leftarrow \text{PC} + 4 )</td>
</tr>
</tbody>
</table>

Step 1: Requirements of the Instruction Set

- **Memory**
  - instruction & data

- **Registers (32 x 32)**
  - read RS
  - read RT
  - Write RT or RD

- **PC**

- **Extender**

- **Add and Sub register or extended immediate**

- **Add 4 or extended immediate to PC**

Step 2: Components of the Datapath

- **Combinational Elements**

- **Storage Elements**
  - Clocking methodology
Combinational Logic Elements (Basic Building Blocks)

- **Adder**
  
  ![Adder Diagram]
  
  - A \(\rightarrow\) Sum
  - B \(\rightarrow\) Carry

- **MUX**
  
  ![MUX Diagram]
  
  - Select \(\rightarrow\) Y

- **ALU**
  
  ![ALU Diagram]
  
  - OP \(\rightarrow\) Result

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 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 (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) 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 \(\Rightarrow\) 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:**
  
  - Address selects the word to put on Data Out
  - 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 \(\Rightarrow\) Data Out valid after “access time.”
Recall: Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

Administrivia

- LAB 2 due tomorrow at midnight
  - Must demonstrate your lab to TAs in section
- Tomorrow Section in LAB (119 Cory)
  - Must form up groups in tomorrow
  - 4 or 5 person groups
  - No 3 person groups (this means that not all 5-person groups may be able to exist)
- LAB 3/Homework 3 out tomorrow
  - Build a single-cycle processor in Hardware!
  - Start working in Groups
- Today’s material in Chapter 5
- First Midterm: Wednesday, March 10
  - 5:30 – 8:30 in 306 Soda
  - La Val’s Afterwards

Step 3: Assemble DataPath meeting our requirements

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

3a: 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"
3b: Add & Subtract

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)
- Example: \( \text{addU rd, rs, rt} \)
- \( Ra, Rb, \) and \( Rw \) come from instruction’s \( rs, rt, \) and \( rd \) fields
- \( \text{ALUctr} \) and \( \text{RegWr} \): control logic after decoding the instruction

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
</tbody>
</table>

6 bits | 5 bits | 5 bits | 5 bits | 6 bits |

Register-Register Timing: One complete cycle

3c: Logical Operations with Immediate

- \( R[rt] \leftarrow R[rs] \text{ op } \text{ZeroExt}[imm_{16}] \)

3d: Load Operations

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm_{16}]] \)
- Example: \( \text{lw rt, rs, imm}_{16} \)
3e: Store Operations


3f: The Branch Instruction

beq rs, rt, imm16

mem[PC] · Fetch the instruction from memory

Equal <- R[rs] == R[rt] · Calculate the branch condition

if (Equal)

- PC <- PC + 4 + (SignExt(imm16) x 4)
- else
- PC <- PC + 4

Datapath for Branch Operations

beq rs, rt, imm16 · Datapath generates condition (equal)

Putting it all together

Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit

We have everything except control signals (underline)

Today’s lecture will show you how to generate the control signals
Recap: Meaning of the Control Signals

- nPC_MUX_sel: 0 ⇒ PC ← PC + 4
  1 ⇒ PC ← PC + 4 + SignExt(Im16) || 00

- Later in lecture: higher-level connection between mux and branch cond

---

Step 4: An Abstract View of the Implementation

- Ideal Instruction Memory

  Control
  - Instruction: op, rs, rt
  - Control Signals
  - Conditions

  Datapath
  - Ideal Data Memory
  - RW, Ra, Rb: 32 32-bit Registers
  - RW, Ra, Rb

  Clk

RTL: The Add Instruction

- add rd, rs, rt
  - mem[PC]
  - R[rd] ← R[rs] + R[rt]
  - PC ← PC + 4

- Fetch the instruction from memory
- The actual operation
- Calculate the next instruction’s address
**Instruction Fetch Unit at the Beginning of Add**

- Fetch the instruction from Instruction memory: \( \text{Instruction} \leftarrow \text{mem}[\text{PC}] \)
  - This is the same for all instructions

**The Single Cycle Datapath during Add**

- \( R[\text{rd}] \leftarrow R[\text{rs}] + R[\text{rt}] \)

**Instruction Fetch Unit at the End of Add**

- \( \text{PC} \leftarrow \text{PC} + 4 \)
  - This is the same for all instructions except: Branch and Jump

**The Single Cycle Datapath during Or Immediate**

- \( R[\text{rt}] \leftarrow R[\text{rs}] \text{ or ZeroExt}[\text{Imm16}] \)
The Single Cycle Datapath during Branch

If \( R[rs] - R[rt] == 0 \) then Zero <- 1; else Zero <- 0

nPC_sel = "Br"

Instruction Fetch Unit at the End of Branch

If \( Zero == 1 \) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

What is encoding of nPC_sel?
Direct MUX select?
Branch / not branch

Let’s choose second option

nPC_MUX_sel
0 0 1 1
0 1 1 0

A Summary of Control Signals: The Decode Process

This is the process of “decoding” instructions!

Step 5: Assemble Control

This is a hardware implementation issue
We will complete design next lecture
LAB 3 involves implementing this processor
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.”

Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

Next Time:
Multi-cycle Implementation!

Summary

- 5 steps to design a processor
  1. Analyze instruction set => datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic

- MIPS makes it easier
  - Instructions same size
  - Source registers always in same place
  - immediates same size, location
  - Operations always on registers/immediates

- Single cycle datapath => CPI=1, CCT => long

- Next time: implementing control