An Abstract View of the Critical Path

- This affects how much you can overclock your PC!

Critical Path (Load Operation) =
Delay clock through PC (IFs) +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to perform a 32-bit Add +
Data Memory Access Time +
Stable Time for Register File Write

Recap: Meaning of the Control Signals

- ExtOp: “zero”, “sign”
- ALUsrc: 0 => reg; 1 => Mem
- ALUctr: “add”, “sub”, “or”
- WR: 1 => write register

RTL: The Add Instruction

- MEM[PC] Fetch the instruction from memory
- PC = PC + 4 Calculate the next instruction’s address
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions

Instruction Fetch Unit at the End of Add

- PC = PC + 4
  - This is the same for all instructions except: Branch and Jump

Single Cycle Datapath during Or Immediate?

- \( R[rt] = R[rs] \text{ OR ZeroExt}[\text{imm16}] \)

The Single Cycle Datapath during Add

- \( R[rd] = R[rs] + R[rt] \)

The Single Cycle Datapath during Load?

- \( R[rt] = \text{Data Memory} (R[rs] + \text{SignExt}[\text{imm16}]) \)
The Single Cycle Datapath during Load

- \( R[rt] = \text{Data Memory} (R[rs] + \text{SignExt}[\text{imm16}]) \)

The Single Cycle Datapath during Store

- Data Memory \( \{R[rs] + \text{SignExt}[\text{imm16}]\} = R[rt] \)

The Single Cycle Datapath during Branch?

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

Instruction Fetch Unit at the End of Branch

- Instruction \( \{\text{Zero} = 1\} \)
  - Direct MUX select?
  - Branch / not branch
  - Let’s pick 2nd option

Q: What logic gate?
A Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register Transfer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R[r] &lt;- R[s] + R[t]; PC &lt;- PC + 4</td>
<td></td>
</tr>
<tr>
<td>ALUsrc = RegB, ALUctrl = “add”, RegDst = rd, RegWr, nPC_sel = “=4”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>R[r] &lt;- R[s] - R[t]; PC &lt;- PC + 4</td>
<td></td>
</tr>
<tr>
<td>ORI</td>
<td>R[r] &lt;- R[s] + zero_ext(1n1); PC &lt;- PC + 4</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>R[r] &lt;- MEME R[s] + sign_ext(1n16); PC &lt;- PC + 4</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>MEME R[s] + sign_ext(1n16); &lt;- R[r]; PC &lt;- PC + 4</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>if (R[r] == R[t]) then PC &lt;- PC + sign_ext(1n16); else PC &lt;- PC + 4</td>
<td></td>
</tr>
</tbody>
</table>

The Single Cycle Datapath during Jump

- New PC = (PC[31..28], target address, 00)

The Single Cycle Datapath during Jump

- New PC = (PC[31..28], target address, 00)
**Instruction Fetch Unit at the End of Jump**

- New PC = \( \{ PC[31..28], \text{target address, 00} \} \)

- Jump: 
  - `Inst Memory Addr` 
  - `Instructions[31:0]`

- `aPC_SEL`, `Zero`, `aPC_MUX_SEL`

- How do we modify this to account for jumps?

**Have CL to implement Jump on paper now**

- `Inst31` 
  - `Inst30` 
  - `Inst29` 
  - `Inst28` 
  - `Inst27` 
  - `Inst26` 
  - `Inst25` 
  - `Inst01` 
  - `Inst00` 

  - Jump

**Peer Instruction**

- A. Our ALU is a synchronous device
- B. We should use the main ALU to compute PC=PC+4
- C. The ALU is inactive for memory reads or writes.

**And in Conclusion... Single cycle control**

- 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

- Control is the hard part

- MIPS makes that easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location

- Operations always on registers/immediates