Multi Cycle Datapath

Slides courtesy of Professor Tod Amon, Southern Utah University, with minor modifications by Nathan Sprague
**ALU control lines**

0000 = and  
0001 = or  
0010 = add  
0110 = subtract  
0111 = slt  
1100 = NOR

---

**FIGURE 5.13** The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don’t-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don’t-care terms and are replaced with XX in the truth table.
Simple combinational logic
Control lines based on opcode

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Problems with single cycle implementation...
Mult-cycle design...
Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.

```plaintext
IR <= Memory[PC];
PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch

\[
\begin{align*}
A & \leftarrow \text{Reg}[\text{IR}[25:21]]; \\
B & \leftarrow \text{Reg}[\text{IR}[20:16]]; \\
\text{ALUOut} & \leftarrow \text{PC} + (\text{sign} - \text{extend}(\text{IR}[15:0])) \ll 2; \\
\end{align*}
\]

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)
Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:
  \[ \text{ALUOut} \leftarrow A + \text{sign-extend}(\text{IR}[15:0]); \]

- R-type:
  \[ \text{ALUOut} \leftarrow A \text{ op B}; \]

- Branch:
  \[ \text{if } (A==B) \text{ PC } \leftarrow \text{ALUOut}; \]
Step 4 (R-type or memory-access)

- Loads and stores access memory

  \[
  \text{MDR} \leftarrow \text{Memory}[\text{ALUOut}]; \\
  \text{or} \\
  \text{Memory}[\text{ALUOut}] \leftarrow B; \\
  \]

- R-type instructions finish

  \[
  \text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut}; \\
  \]
Write-back step

- $\text{Reg[IR[20:16]]} \leftarrow \text{MDR};$

Which instruction needs this?
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we’ve accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification
Note:
- don't care if not mentioned
- asserted if name only
- otherwise exact value

How many state bits will we need?
Finite State Machine for Control

Implementation:

- PCWrite
- PCWriteCond
- IorD
- MemRead
- MemWrite
- IRWrite
- MemtoReg
- PCSource
- ALUOp
- ALUSrcB
- ALUSrcA
- RegWrite
- RegDst
- NS3
- NS2
- NS1
- NS0
PLA Implementation