A Complete Datapath for R Type Instructions

- Lw, Sw, Add, Sub, And, Or, Slt can be performed
- For j (jump) we need an additional multiplexor

**What Else is Needed in Data Path**

- Support for j and jr
  - For both of them PC value need to come from somewhere else
  - For J, PC is created by 4 bits (31:28) from old PC, 26 bits from IR (27:2) and 2 bits are zero (1:0)
  - For JR, PC value comes from a register
- Support for JAL
  - Address is same as for J inst
  - OLD PC needs to be saved in register 31
- And what about immediate operand instructions
  - Second operand from instruction, but without shifting
- Support for other instructions like lw and immediate inst write

**Control**

- For each instruction
  - Select the registers to be read (always read two)
  - Select the 2nd ALU input
  - Select the operation to be performed by ALU
  - Select if data memory is to be read or written
  - Select what is written and where in the register file
  - Select what goes in PC
- Information comes from the 32 bits of the instruction
- Example: add $8, $17, $18

**Adding Control to DataPath**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**ALU Control**

- ALU's operation based on instruction type and function code
  - e.g., what should the ALU do with any instruction
- Example: lw $1, 100($2)

We are ignoring some details like setup and hold times
Other Control Information

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq, 10 = arithmetic
    - 11 = Jump
  - function code for arithmetic
- Control can be described using a truth table:

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp Funct field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>010</td>
</tr>
</tbody>
</table>

Implementation of Control

- Simple combinational logic to realize the truth tables

A Complete Datapath with Control

Datapath with Control and Jump Instruction

Timing: Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:
### Instruction Format

<table>
<thead>
<tr>
<th>Bit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-26</td>
<td>REG 1</td>
</tr>
<tr>
<td>25-21</td>
<td>REG 2</td>
</tr>
<tr>
<td>20</td>
<td>LOAD ADDRESS</td>
</tr>
<tr>
<td>15-11</td>
<td>ADD/RD</td>
</tr>
<tr>
<td>10-6</td>
<td>ADD/RD</td>
</tr>
<tr>
<td>5-0</td>
<td>OFFSET</td>
</tr>
</tbody>
</table>

### Operation for Each Instruction

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>1. READ INST</td>
</tr>
<tr>
<td>SW</td>
<td>1. READ INST</td>
</tr>
<tr>
<td>R-Type</td>
<td>1. READ INST</td>
</tr>
<tr>
<td>BR-Type</td>
<td>1. READ INST</td>
</tr>
<tr>
<td>JMP-Type</td>
<td>1. READ INST</td>
</tr>
</tbody>
</table>

### Multicycle Approach

- We will be reusing functional units
  - Break up the instruction execution in smaller steps
  - Each functional unit is used for a specific purpose in one cycle
- Balance the work load
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- At the end of cycle, store results to be used again
  - Need additional registers
- Our control signals will not be determined solely by instruction
  - E.g., what should the ALU do for a “subtract” instruction?
- We'll use a finite state machine for control

### Review: finite state machines

- Finite state machines:
  - A set of states and
  - Next state function (determined by current state and the input)
  - Output function (determined by current state and possibly input)
- We'll use a Moore machine (output based only on current state)

### Multi-Cycle DataPath Operation

<table>
<thead>
<tr>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>MEM</td>
</tr>
<tr>
<td>REG FILE</td>
</tr>
</tbody>
</table>

### 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.
- Can be described succinctly using RTL "Register-Transfer Language"

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
- RTL:

  \[ A = \text{Reg}[IR[25-21]]; \]
  \[ B = \text{Reg}[IR[20-16]]; \]
  \[ \text{ALUOut} = PC + (\text{sign-extend}(IR[15-0]) \ll 2); \]

- 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:
  \[ \text{ALUOut} = A + \text{sign-extend}(IR[15-0]); \]
- R-type:
  \[ \text{ALUOut} = A \text{ op } B; \]
- Branch:
  \[ \text{if (A==B) PC = ALUOut}; \]

Step 4 (R-type or memory access)

- Loads and stores access memory
  \[ \text{MDR} = \text{Memory}[\text{ALUOut}]; \]
  or
  \[ \text{Memory}[\text{ALUOut}] = B; \]
- R-type instructions finish
  \[ \text{Reg}[IR[15-11]] = \text{ALUOut}; \]

Write back step

- \[ \text{Reg}[IR[20-16]] = \text{MDR}; \]

What about all the other instructions?

Summary:

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>[ IR = \text{Memory}[PC]; ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/regular fetch</td>
<td>[ A = \text{Reg}[IR[25-21]]; ] [ B = \text{Reg}[IR[20-16]]; ] [ \text{ALUOut} = PC + (\text{sign-extend}(IR[15-0]) \ll 2); ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>[ \text{ALUOut} = A \text{ op } B; ] [ \text{ALUOut} = A \text{ op } \text{sign-extend}(IR[15-0]); ] [ \text{PC} = \text{ALUOut}; ]</td>
<td></td>
<td>[ \text{PC} = \text{PC}[31-28] ] [ \text{IR}[25-16]; ]</td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>[ \text{Reg}[IR[15-11]] = \text{ALUOut}; ] [ \text{Load MDR = Memory}[\text{ALUOut}]; ] [ \text{Load MDR = Memory}[\text{ALUOut}]; ]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read corruption</td>
<td>[ \text{Load Reg}[IR[15-11]] = \text{MDR}; ]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>