Control

• Must describe hardware to compute 4-bit ALU control input
  – given instruction type
    00 = lw, sw
    01 = beq
    10 = arithmetic
  – function code for arithmetic

• Describe it using a truth table (can turn into gates):

<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>lw</td>
<td>0</td>
<td>1</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>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>arithmetic</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Our Simple Control Structure

• All of the logic is combinational
• We wait for everything to settle down, and the right thing to be done
  – ALU might not produce “right answer” right away
  – we use write signals along with clock to determine when to write
• Cycle time determined by length of the longest path

We are ignoring some details like setup and hold times
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (200ps),
  - ALU and adders (100ps),
  - register file access (50ps)

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:

Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional "internal" registers
Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
  - Example:
    - The add instruction changes a register.
    - Register specified by bits 15:11 of instruction.
    - Instruction specified by the PC.
    - New value is the sum ("op") of two registers.
    - Registers specified by bits 25:21 and 20:16 of the instruction

\[
\text{Reg[Memory[PC][15:11]]} \leftarrow \text{Reg[Memory[PC][25:21]] \text{ op} Reg[Memory[PC][20:16]]}
\]

- In order to accomplish this we must break up the instruction.
  (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

\[
\text{Reg[Memory[PC][15:11]]} \leftarrow \text{Reg[Memory[PC][25:21]] \text{ op} Reg[Memory[PC][20:16]]}
\]

- Could break down to:
  - IR \leftarrow Memory[PC]
  - A \leftarrow Reg[IR[25:21]]
  - B \leftarrow Reg[IR[20:16]]
  - ALUOut \leftarrow A \text{ op} B
  - Reg[IR[20:16]] \leftarrow ALUOut

- We forgot an important part of the definition of arithmetic!
  - PC \leftarrow PC + 4

Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)
- Result: Our book's multicycle Implementation!

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”

\[ IR \leftarrow \text{Memory}[PC]; \]
\[ PC \leftarrow PC + 4; \]

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:

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

- 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:

\[ \text{ALUOut} \leftarrow A + \text{sign-extend}(IR[15:0]); \]

- R-type:

\[ \text{ALUOut} \leftarrow A \circ B; \]

- 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

\[ \text{Reg}[IR[15:11]] \leftarrow \text{ALUOut}; \]

The write actually takes place at the end of the cycle on the edge.
Write-back step

- $Reg[IR[20:16]] \leftarrow MDR,$

*Which instruction needs this?*