Chapter 5

State

- Combinational Elements: Adder
- State Elements: Storage

Flip-Flop

- One-bit Memory

\[ \begin{array}{c|c|c}
\text{Something to Remember} & \text{D} & \text{Q} \\
\hline
\text{Remember Now!} & \text{Q} & \text{What I Remember} \\
\end{array} \]

- Edge-Triggered

Register

A collection of flip-flops
State Machines

- Stored bit describes the state of the object.
- A clock generates a regular pattern of voltage changes.
- The state changes when the clock “ticks.”

Counter: A State Machine

At each tick, the machine does the “next” thing.
- The faster the clock, the faster the machine.
- The clock must be slow enough for changes to propagate back to the registers during a clock cycle.
- This is what those computer ads are talking about.
The Processor: Datapath & Control

- We’re ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw`, `sw`
  - arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
  - control flow instructions: `beq`, `j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers

Datapath and Control

- Datapath: Moving, storing, and creating data.
- Control: Making each of those happen at right place and time.

Building the Datapath

- Parts.
- Sub-assemblies.
- Complete datapath.

Register File

- Built using D flip-flops

[Diagram of a register file circuit]
Register File

- Note: we still use the real clock to determine when to write

Datapath For Fetching

Simple Implementation

- Include the functional units we need for each instruction

Datapath For R-Type Instructions

Why do we need this stuff?
Datapath For Loading and Storing

Datapath Combined

Datapath for Branching

Control

- Signals the devices what they must do.
- Interprets the instruction.
- A large combinational circuit.
Add Control Units

Inputs: Low six bits of the instruction (function code), and two bits sent from the main controller.

Outputs: ALU Function selection.

ALU Control Signals

<table>
<thead>
<tr>
<th>ALU Control Input</th>
<th>ALU Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>and</td>
</tr>
<tr>
<td>0001</td>
<td>or</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>set on less than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Secondary Controller

From Main | Func | Output |
---|---|---|
00 | XXXXXX | 0010 |
X1 | XXXXXX | 0110 |
1X | XX0000 | 0010 |
1X | XX0010 | 0110 |
1X | XX0100 | 0000 |
1X | XX0101 | 0001 |
1X | XX1010 | 0111 |

Determine what ALU needs to do based on the main controller’s direction and the function code from the instruction.
Main Control

- Input: High six bits of the instruction (op code).

<table>
<thead>
<tr>
<th>op</th>
<th>seg1</th>
<th>seg2</th>
<th>destreg</th>
<th>shift</th>
<th>amt</th>
<th>funct</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td>R</td>
</tr>
</tbody>
</table>

- Outputs: RegDst, Branch, MemRead, MemtoReg, AULOp, MemWrite, ALUSrc, RegWrite.

Main Control: Output Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mem-Read</th>
<th>Mem-Write</th>
<th>ALU Branch</th>
<th>ALU Op1</th>
<th>ALU Op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Main Control: Input Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op Code: First Six Instruction Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Op5</td>
</tr>
<tr>
<td>ee</td>
<td></td>
</tr>
<tr>
<td>R-Format</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
</tr>
</tbody>
</table>

Just a large, combinational circuit.

Single-Cycle Implementation
**Single-Cycle**

Problems:
- The cycle must be long enough for the longest instruction.
- Components must be duplicated, as the second ALU.

A Solution:
- Shorter cycle.
- Multiple cycles per instruction.
- Different numbers of cycles for different instructions.

**Changes for Multi-Cycle**

- A single memory unit for both instructions and data.
- A single ALU, rather than an ALU and two adders.
- Add internal registers to hold the results of each cycle for the next.

**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

**High-Level View**
Full Multi-Cycle Implementation

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"

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

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

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 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:

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

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

Write-back step

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

What about all the other instructions?

Step 4 (R-type or memory-access)

- Loads and stores access memory
  
  \[
  \text{MDR = Memory[ALUOut]}
  \]

- R-type instructions finish
  
  \[
  \text{Reg[IR[15-11]] = ALUOut;}
  \]

  The write actually takes place at the end of the cycle on the edge

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>PC = IR</td>
<td>PC = PC + 4</td>
<td>PC = PC + 4</td>
<td>PC = PC + 4</td>
</tr>
<tr>
<td>Instruction decoder/selector fetch</td>
<td>A = IR[25:16]</td>
<td>B = IR[25:16]</td>
<td>ALUOut = PC + \text{sign-extend}(IR[15:0]) \ll 2</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut = A \text{ op } B</td>
<td>ALUOut = A + \text{sign-extend}(IR[15:0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC + 4</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg[IR[15:11]] = ALUOut</td>
<td>Load MDR = Memory[ALUOut] or Store MDR = Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multi-Cycle Operation

Control Logic

Finite State Machine for Control

- Implementation:

Control Logic

Compute the Outputs from the Inputs

- Lots of gates.
- ROM
- Etc.
An Alternative: Microcode

Microcode storage

Outputs

Datapath control outputs

Input

Sequencing control

Microprogram counter

Address select logic

Inputs from instruction register opcode field

Microcontroller Address Select Logic

CSc 314 - T W Bennet - Mississippi College

Microinstructions

- One large instruction for each state.
- Specifies the signals to generate.
- Specifies the next instruction.

Microinstruction format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Signal active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>000</td>
<td>ALUo = 000</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Sub</td>
<td>010</td>
<td>ALUo = 010</td>
<td>Cause the ALU to subtract.</td>
</tr>
<tr>
<td>Funcode</td>
<td>010</td>
<td>Use the instruction function code to determine ALU control.</td>
<td></td>
</tr>
<tr>
<td>SRC1</td>
<td>A</td>
<td>ALUo = 1</td>
<td>Use the PC or the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>ALUo = 0</td>
<td>Use the PC as the next ALU input.</td>
</tr>
<tr>
<td>SRC2</td>
<td>B</td>
<td>ALUo = 0</td>
<td>Use the second ALU input.</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>ALUo = 1</td>
<td>Use the output of the instruction as the second ALU input.</td>
</tr>
<tr>
<td>Exp</td>
<td>C</td>
<td>ALUo = 1</td>
<td>Use the output of the instruction as the second ALU input.</td>
</tr>
<tr>
<td>Field</td>
<td></td>
<td></td>
<td>Use a register using the field of the IR as the register number and the contents of the ALU as the data.</td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td>Microcode 9</td>
<td>Write a register using the field of the IR as the register number and the contents of the microcode as the data.</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>Microcode 10</td>
<td>Write a register using the field of the IR as the register number and the contents of the microcode as the data.</td>
</tr>
<tr>
<td>Read PC</td>
<td></td>
<td>Read memory using the PC as address, will result in R5 and the data.</td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td></td>
<td>Read memory using the ALU as address, will result in Microcode.</td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td></td>
<td>Write memory using the ALU as address, contents B as the data.</td>
<td></td>
</tr>
<tr>
<td>Microcode</td>
<td></td>
<td>Write output of the microcontroller.</td>
<td></td>
</tr>
<tr>
<td>PC write control</td>
<td>00</td>
<td>If the 5th output of the ALU is 00, write the contents of the PC into memory.</td>
<td></td>
</tr>
<tr>
<td>Jump address</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>001</td>
<td>Choose the next microinstruction to execute.</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>001</td>
<td>Go to the first microinstruction of the next instruction.</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>010</td>
<td>Resume until the end of the instruction.</td>
<td></td>
</tr>
</tbody>
</table>
Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Exceptions and Interrupts

Jump to the O/S upon certain events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Source</th>
<th>MIPS Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Device Request</td>
<td>External</td>
<td>Interrupt</td>
</tr>
<tr>
<td>User Program Calls O/S (Syscall)</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Arithmetic Overflow</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Hardware Malfunction</td>
<td>Either</td>
<td>Either</td>
</tr>
</tbody>
</table>

Types of Exceptions

- Vectored: Jumps to different locations based on type of exception or interrupt.
- Cause Register: Store a code for the cause of the exception in a special register, then jump to a standard place.
- MIPS uses the later.

Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes
MIPS Exception Registers

- EPC: Holds the address of the offending instruction.
- Cause: Holds a code for the cause of the exception.

Exceptions

For the example implementation, we consider two exceptions

- Undefined instruction: Code 0.
- Arithmetic overflow: Code 32.