Chapter 5
State

- Combinational Elements: *Adder*
- State Elements: *Storage*
Flip-Flop

- One-bit Memory

Something to Remember  \[ \text{D} \rightarrow \text{Q} \]  What I Remember

Remember Now!  \[ \text{Q}' \rightarrow \text{Q} \]  What I Remember

- 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.”
State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?
  - State changes when the clock rises.
Counter: A State Machine
Clocks and State Machines

- 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
Register File

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

- Include the functional units we need for each instruction

Why do we need this stuff?
Datapath For Fetching

- PC
  - Read address
  - Instruction
  - Instruction memory

Add

4
Datapath For R-Type Instructions

Instruction

Read register 1
Read register 2
Write register
Write data

Read data 1
Read data 2

RegWrite

3 ALU operation

ALU

Zero

ALU result

Read reg 1
Read reg 2
Write reg

op sreg1 sreg2 destreg shft amt funct

6 5 5 5 5 6
Datapath For Loading and Storing

Instruction

Registers

Read register 1
Read register 2
Write register
Write data

MemWrite

MemRead

Data memory

Address
Write data

ALU
ALU result

Zero

Read data 1
Read data 2

RegWrite

Sign extend

32

16

3

ALU operation

Write register

Read Reg 1
Read Reg 2
Others
Loads

op Rbase Rdest Offset

6 5 5 16
Datapath for Branching

1. Instruction
2. Read register 1
3. Read register 2
4. Write register
5. Write data
6. PC + 4 from instruction datapath
7. Add Sum (Branch target)
8. Shift left 2
9. ALU operation
10. ALU Zero
11. To branch control logic
12. RegWrite
13. Sign extend
14. 16
15. 32

CSc 314 · T W Bennet · Mississippi College
Datapath Combined
Control

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

• 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

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

<table>
<thead>
<tr>
<th>From Main</th>
<th>Func</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>XXXXXX</td>
<td>0010</td>
</tr>
<tr>
<td>X1</td>
<td>XXXXXX</td>
<td>0110</td>
</tr>
<tr>
<td>1X</td>
<td>XX0000</td>
<td>0010</td>
</tr>
<tr>
<td>1X</td>
<td>XX0010</td>
<td>0110</td>
</tr>
<tr>
<td>1X</td>
<td>XX0100</td>
<td>0000</td>
</tr>
<tr>
<td>1X</td>
<td>XX0101</td>
<td>0001</td>
</tr>
<tr>
<td>1X</td>
<td>XX1010</td>
<td>0111</td>
</tr>
</tbody>
</table>
Main Control

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

<table>
<thead>
<tr>
<th>op</th>
<th>sreg1</th>
<th>sreg2</th>
<th>destreg</th>
<th>shft 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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>Rbase</th>
<th>Rdest</th>
<th>Offset</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></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>Mem-toReg</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>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>Op5</th>
<th>Op4</th>
<th>Op3</th>
<th>Op2</th>
<th>Op1</th>
<th>Op0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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.
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
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.
Full Multi-Cycle 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"

\[
\begin{align*}
\text{IR} &= \text{Memory[PC]}; \\
\text{PC} &= \text{PC} + 4;
\end{align*}
\]

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:

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

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

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

- Branch:
  
  \[
  \text{if } (A==B) \text{ PC } = ALUOut;
  \]
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
  
  \[ \text{Reg}[\text{IR}[15-11]] = \text{ALUOut} ; \]

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

- $Reg[IR[20-16]] = MDR$;

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></td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] II (IR[25-0] &lt;&lt; 2)</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: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multi-Cycle Operation
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

1

Adder

Microprogram counter

Address select logic

Inputs from instruction register opcode field
Address Select Logic

Next Address (State)

11 10 01 00

Data Out
ROM 2
Addr In

Data Out
ROM 1
Addr In

Sequence Control
(From microinstruction)

Curr Address + 1
Op Code (From IR)

Microcontroller Address Select Logic
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>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td></td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
<td></td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td></td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td></td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
<td></td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 0</td>
<td></td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 01</td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>ExtShift</td>
<td>ALUSrcB = 10</td>
<td></td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rd field of the IR as the register number and</td>
</tr>
<tr>
<td></td>
<td>RegDst = 1,</td>
<td></td>
<td>the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write MDR</td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rt field of the IR as the register number and</td>
</tr>
<tr>
<td></td>
<td>RegDst = 0,</td>
<td></td>
<td>the contents of the MDR as the data.</td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td>lrd = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>lrd = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite,</td>
<td></td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td></td>
<td>lrd = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCSource = 00</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<td>PCWrite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCSource = 01</td>
<td></td>
<td>If the Zero output of the ALU is active, write the PC with the contents</td>
</tr>
<tr>
<td></td>
<td>PCWriteCond</td>
<td></td>
<td>of the register ALUOut.</td>
</tr>
<tr>
<td>jump address</td>
<td>PCSource = 10</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td></td>
<td>PCWrite</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td></td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCtl = 00</td>
<td></td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td></td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td></td>
<td>Dispatch using the ROM 2.</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
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
## 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.
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.
Exception Datapath