Multi-Cycle Implementation

Today’s Menu
- Multi-Cycle machines
  - Why multi-cycle?
  - Comparative performance
  - Physical and Logical Design of Datapath and Control
  - Microprogramming

Multi-cycle Solution
Idea: Let the FASTEST instruction determine clock period

<table>
<thead>
<tr>
<th>Instr Class 1</th>
<th>Instr Class 2</th>
<th>Instr Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Takes 4 cycles</td>
<td></td>
<td>Takes 2 cycles</td>
</tr>
</tbody>
</table>

Less Wasted Time

Multi-cycle Reality
- We are going to go further than allowing the fastest instruction to determine rate
- We are going to break EVERY instruction up into phases

MIPS Architecture Instruction Classes

<table>
<thead>
<tr>
<th>ALU</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem rd</td>
<td>mem rd</td>
<td>mem rd</td>
<td>mem rd</td>
<td>mem rd</td>
</tr>
<tr>
<td>op code</td>
<td>op code</td>
<td>op code</td>
<td>op code</td>
<td>op code</td>
</tr>
<tr>
<td>1 or 2 reg rd</td>
<td>2 reg rd</td>
<td>2 reg rd</td>
<td>2 reg rd</td>
<td>PC, immrd</td>
</tr>
</tbody>
</table>

Note That...
- Instruction decode and register read must be done for all classes of instructions
- PC+4 can be done right after instruction fetch
- Address generation for Jump can be performed during the decode step
- The same adder can be shared among:
  - Instruction fetch logic (PC = PC + 4)
  - Address generation logic (Address = A + [R[15:0])
  - Branch target calculation (next PC = (PC + 4) + IR[15:0])
  - ALU operation (ALUOutput = A + B)
- Same memory port can be used to access instructions and data
**Recall MIPS Instruction Formats...**

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Field Widths</th>
<th>Register Field RS</th>
<th>Register Field RT</th>
<th>Register Field RD</th>
<th>Immediate Field</th>
<th>Function Field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-type Instructions</strong></td>
<td>20-16</td>
<td>15-11</td>
<td>10-6</td>
<td>5-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load or Store</strong></td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
<td></td>
<td>immediate (address)</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-0</td>
<td></td>
<td>immediate (address)</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>31-26</td>
<td>25-0</td>
<td></td>
<td></td>
<td></td>
<td>immediate (address)</td>
</tr>
</tbody>
</table>

**Execution Steps (1)**

- **Instruction Fetch**
  
  \[ IR = \text{Memory}[PC]; \]
  \[ PC = PC + 4; \]

**Execution Steps (2)**

- **Instruction Decode and Register Fetch**
  
  \[ A = \text{Reg}[IR[25..21]]; \]
  \[ B = \text{Reg}[IR[20..16]]; \]
  \[ \text{Target} = PC + 4 + (\text{signExtend}(IR[15..0]) << 2); \]
Execution Step (3)

- Execution, memory address computation or branch completion
  - Memory Reference
    \[ \text{ALUOut} = A + \text{signExtend}(IR[15..0]) \]
  - Arithmetic/Logical Operation
    \[ \text{ALUOut} = A + B \]
  - Branch
    \[ \text{If } (A == B) \text{ PC } = \text{Target}; \]
  - Jump
    \[ \text{PC} = \text{PC}[31 ..28] || (IR[25..0] << 2); \]

Execution Step (4)

- Memory access or R-type instruction completion
  - Memory Reference
    \[ \text{MDR} = \text{Memory}[	ext{ALUOut}]; \]
    \[ \text{or} \]
    \[ \text{MDR} = \text{Memory}[	ext{ALUOut}] = B; \]
  - Arithmetic/Logical Instructions (R-type)
    \[ \text{Reg}[IR[15..11]] = \text{ALUOut}; \]
  - Branch, Jump
    Nothing

Execution Step (5)

- Memory Read completion (Load only)
  \[ \text{Reg}[IR[20..16]] = \text{MDR}; \]

Finite State Machine Control

Instruction Fetch/Decode and Register Fetch

Memory Access | R-Type Instructions | Branch Instruction | Jump Instruction
**Finite State Machine Control**

- Instruction Fetch/Decode and Register Fetch are the same for all instructions
- Different sub-state machines for
  - Memory Access
  - R-Type
  - Branch
  - Jump

**Multicycle Control**

![Multicycle Control Diagram](image)

**Memory-Reference FSM**

- From State 1
- Memory address computation
- Memory Access
- Write-back step
- Back To State 0

**R-type FSM**

- From State 1
- Execution
- R-type completion
- Back To state 0
**Multicycle Control**

![Multicycle Control Diagram]

**Performance of Multicycle Implementation**

- Each type of instruction can take a variable # of cycles
- Example
  - Assume the following instruction distributions:
    - loads 5 cycles 22%
    - stores 4 cycles 11%
    - R-type 4 cycles 49%
    - branches 3 cycles 16%
    - jump 3 cycles 2%
  - What’s the average Cycles Per Instruction (CPI)?
    - CPI = (CPU clock cycles/Instruction Count)
    - CPI = (5 cycles * 0.22) + (4 cycles * 0.11) + (4 cycles * 0.49)
    - = (3 cycles * 0.16) + (3 cycles * 0.02)
    - CPI = 4.04 cycles per instruction
- What was the CPI for the single-cycle machine?
  - Single cycle implies 1 clock cycle per instruction --> CPI = 1.0
  - So isn’t the single-cycle machine about 4 times faster?

**So who cares?**

- Imagine:
  - Hundreds of instructions…
  - Tens of different instruction classes (we’ve seen four)
  - Instructions that take anywhere from 1 to 100 cycles to complete
- That’s what any real ISA has
- Now imagine drawing the FSM diagram for that!
- Solution 1: Write Verilog and use synthesis (today)
- Solution 2: Use some programming lessons

**Exploiting the structure for microcode**

- First cycles are the same
- No reconvergence after decode
- Limited Branching

![Exploiting the Structure for Microcode Diagram]
Microcode Word Definition

- What does the microcoded data control?
  - ALU Control: Add, Subt, Func code
  - SRC1: PC, A
  - SRC2: B, 4, Extend, Extshift
  - Register Control: Read, Write ALU, Write MDR
  - Memory: Read PC, Read ALU, Write ALU
  - PCWrite control: ALU, ALUOut-cond, JumpAddress
  - Sequencing: Next entry or next instruction

- Total Word Size >= 13

Microcoded Table

<table>
<thead>
<tr>
<th>Addr</th>
<th>Next?</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Writeto</th>
<th>Register</th>
<th>PCWrite</th>
<th>PCSource</th>
<th>MDRWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (IF)</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1 (ID)</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>2 (RE)</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>3 (R7)</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>4 (IF)</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>5 (ID)</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>6 (ID)</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>

Now we can easily encode very complicated finite state machines via the microcode ROM. Basically we’ve built a very simple sequencer.

Microcoded Instructions in the Real World

- AMD’s Bulldozer core
  - Instruction decoder handles 4 instructions at a time
  - Each one can do either a simple “Fast Path” decode
  - Or it has to use a microcoded ROM to decode into multiple instructions
  (Remember x86 is an ugly instruction set, but internally everything looks like MIPS.)

Why is this nice?

- Reduce complexity of control design
  - Only way to do it before synthesis tools
  - Only way to encode Complex Instruction Sets
- Allows bug fixes, optimizations after real hardware
- Now how do people do this today?
  - Specify Style for:
    - Registers
    - Latches
    - Combinational Logic
    - Finite State Machines

And Never Forget Marketing

- It is easy to sell a 4GHz processor
- It is harder to sell “Yes our processor is only 500MHz, but it actually achieves a lower CPI, and therefore really is higher performance.”
  - MUCH harder: Apple tried to say their 33MHz 68540 was as fast as a 50MHz 486! (which it was, but no one believed them.)
  - Intel designed the whole Pentium 4 around higher clock rates and discovered it was too hot and too slow compared to the slower AMD Opteron. Cost them a fortune.
- This is actually changing…
  - Today people care a lot more about power
  - Who cares how fast your cell phone’s processor is? We only care about battery life and it being “fast enough.”

Exceptions and Interrupts

- Exceptions are *exceptional events* that disrupt the normal flow of a program
- Terminology varies between different machines
- Examples of Interrupts
  - User hitting the keyboard
  - Disk drive asking for attention
  - Arrival of a network packet
- Examples of Exceptions
  - Divide by zero
  - Overflow
  - Page fault
Handling Exceptions and Interrupts

- When do we jump to an exception?
  - Upon detection, invoke the OS to "service the event"
    - Right when it occurs?
    - What about in the middle of executing a multi-cycle instruction?
    - Processor checks for event at the end of every instruction
    - Processor provides EPC & Cause registers to inform OS of cause
      - EPC - Exception Program Counter
      - Holds PC that the OS should jump to when resuming execution
      - Cause Register
      - Holds bit-encoded cause of the exception

Exception Flow

- When an exception (or interrupt) occurs, control is transferred to the OS

Summary

- Single cycle implementations have to consider the worst case delay through the datapath to come-up with the cycle time.
- Multicycle implementations have the advantage of using a different number of cycles for executing each instruction.
- In general, the multicycle machine is better than the single cycle machine, but the actual execution time strongly depends on the workload.
- The most widely used machine implementation is neither single cycle, nor multicycle – it’s the pipelined implementation.
  (Next lecture)