Designing a Pipelined CPU

Review -- Single Cycle CPU

Review -- Multiple Cycle CPU

Review -- Instruction Latencies

• Single-Cycle CPU

Load | Fetch Reg/Dec Exec Mem Wr

• Multiple Cycle CPU

Load | Fetch Reg/Dec Exec Mem Wr
Add | Fetch Reg/Dec Exec Wr
Instruction Latencies and Throughput

- **Single-Cycle CPU**
  
<table>
<thead>
<tr>
<th>Load</th>
<th>Fetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

- **Multiple Cycle CPU**
  
  Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5
  
<table>
<thead>
<tr>
<th>Load</th>
<th>Fetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

- **Pipelined CPU**
  
  Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7  Cycle 8
  
<table>
<thead>
<tr>
<th>Load</th>
<th>Fetch</th>
<th>Reg/Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>Wr</th>
</tr>
</thead>
</table>

Pipelining Advantages

- Higher *maximum* throughput
- Higher *utilization* of CPU resources

- But, more complicated *datapath*, more complex control(?)

Pipelining Advantages

<table>
<thead>
<tr>
<th>CPU Design Technology</th>
<th>Control Logic</th>
<th>Peak Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle CPU</td>
<td>Combinational Logic</td>
<td>1</td>
</tr>
<tr>
<td>Multiple-Cycle CPU</td>
<td>FSM or Microprogram</td>
<td>1</td>
</tr>
<tr>
<td>Pipelined CPU</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Pipelining in Modern CPUs

- CPU Datapath
- Arithmetic Units
- System Buses
- Software (at multiple levels)
- etc...
A Pipelined Datapath

IF: Instruction fetch
ID: Instruction decode and register fetch
EX: Execution and effective address calculation
MEM: Memory access
WB: Write back

Mixed Instructions in the Pipeline

lw IM Reg DM Reg
lw IM Reg DM Reg
lw IM Reg DM Reg
lw IM Reg DM Reg

CSE 141, S2'06
Jeff Brown
Pipeline Principles

- All instructions that share a pipeline must have the same *stages* in the same *order*.
  - therefore, *add* does nothing during Mem stage
  - *sw* does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse

---

CSE 141, S2'06

Jeff Brown
The Pipeline, with controls

But....

Pipelined Control

- can’t use microprogram.
- FSM not really appropriate.
- Combinational logic!
  - signals generated once, but follow instruction through the pipeline

Pipelined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>MemtoReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
The Pipeline with Control Logic

The Pipeline in Execution

Is it really that easy?

- What happens when...
  - add $3, $10, $11
  - lw $8, 1000($3)
  - sub $11, $8, $7
The Pipeline in Execution

add $s10, $s1, $s2  sub $s11, $s8, $s7  lw $s8, 1000($s3)  add $s3, $s10, $s11 Write Back

Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

Pipelining Key Points

- ET = IC * CPI * CT
- We achieve high throughput without reducing instruction latency.
- Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).
- Pipelining uses combinational logic (and registers to propagate) to generate control signals.
- Pipelining creates potential hazards.