Lecture 5: ISA Implementation

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

Basic Pipelining

- Pipelining is the organizational implementation technique that has been responsible for the most dramatic increases in computer performance
- Overview of basic pipelining
  - DLX without pipelining
  - What is pipelining?
  - Computing pipeline speedup
  - Clocking pipelines
  - DLX with pipelining
  - Pipeline hazards
  - Handling interrupts
DLX review: Instruction format summary

R-type instructions

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs1</td>
<td>rs2</td>
<td>rd</td>
<td>func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Register ALU ops

I-type instructions

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rs1</td>
<td>rd</td>
<td>IMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Load/store
- ALU immediate
- Cond. branch
- Jump register
- JALR

J-type instructions

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Jump
- JAL

The Basic RISC datapath without pipelining

[Diagram of the Basic RISC datapath]
Data Flow:
The details

<table>
<thead>
<tr>
<th>Reg-Reg ALU</th>
<th>Reg-immed ALU</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IR = IMem[PC]</td>
<td>NPC = PC+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = Regs[IR25..21]</td>
<td>B = Regs[IR20..16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU = A op B</td>
<td>ALU = A op IR15..0</td>
<td>ALU = A + IR15..0</td>
<td>ALU = NPC + IR15..0</td>
<td>ALU = NPC + IR25..0</td>
<td></td>
</tr>
<tr>
<td>Data = DMem(ALU)</td>
<td>DMem(ALU) = B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Regs:
- [IR15..11] = ALU
- [IR20..16] = ALU

Regs:
- [IR20..16] = Data;

PC = NPC
PC = cond?
ALU : NPC
PC = ALU

All operations are combinatorial except the shaded boxes, which store results.

Pipelining

- Exploits instruction-level parallelism by overlapping the execution of consecutive instructions

Unpipelined

Pipelined

![Pipelining Diagram](image_url)
Implementing pipelining

Pipelining: Computing the speedup

- Time per instruction
  - \( TPI = CPI \times cycle\_time \)
  - We can think about pipelining as reducing either CPI or cycle\_time

- Ideal Speedup
  - Speedup = \( \frac{TPI_{without\ pipeline}}{TPI_{with\ pipeline}} \) = number of pipeline stages
  - Requires all stages to be perfectly balanced
  - No latch overhead
  - No stall cycles

- The speedup from pipelining is limited
  - \( CPI_{real} = CPI_{ideal} + CPI_{stall} \)
  - \( CCT_{real} = Time_{longest\ pipestage} + Time_{latch\ overhead} \)
Pipeline example:
3 stages

Assume a 2 nsec latch delay.

**Unpipelined**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ns</td>
<td>16 ns</td>
<td>14 ns</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 ns</td>
<td>16 ns</td>
<td>14 ns</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 ns</td>
<td>18 ns</td>
<td>18 ns</td>
<td>18 ns</td>
</tr>
</tbody>
</table>

Latency = 42 nsec  
Throughput = 1/42 nsec

**Pipelined**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ns</td>
<td>2</td>
<td>16 ns</td>
<td>2</td>
</tr>
<tr>
<td>10 ns</td>
<td>2</td>
<td>16 ns</td>
<td>2</td>
</tr>
<tr>
<td>18 ns</td>
<td>18 ns</td>
<td>18 ns</td>
<td>18 ns</td>
</tr>
</tbody>
</table>

Latency = 3 x 18 = 54 nsec (longer!)  
Throughput = 1/18 nsec (2.3x, not 3x)

---

**Clock overhead and reliable clocking**

Single-phase edge-triggered flipflops (data sampled on rising edge)

\[ \text{Period} \geq T_{\text{logic}} + T_{\text{setup}} + T_{\text{prop}} + T_{\text{skew}} \]

Tskew from: wire delay, buffers, non-vertical edges with varying trigger points
Pipelining DLX: What are the stages?

- Divide into stages so that
  - Time of each stage is about the same
  - Each stage is used at most once in each instruction
  - Registers can be easily defined to hold the state information between stages
  - Not so many stages that latch overhead dominates
  - Not so few stages that we give up performance that could be gained

- For DLX, choose 5 stages:
  - IF: Instruction fetch
  - ID (&RF): Instruction decode and register fetch
  - EX: execution of ALU operation
  - MEM: Memory access
  - WB: Write back to register file

Pipelined Instruction Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
<th>i+4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Looks Simple!
Rules for pipeline registers

- Each stage must be independent, so the inter-stage registers must hold:
  - data values
  - control signals, including
    - Decoded instruction fields
    - MUX controls
    - ALU controls
- Think of the register file as two independent units:
  - Read file, accessed in ID
  - Write file, accessed (simultaneously!) in WB
- There is no “final” set of registers after WB, (WB/IF) because the instruction is finished and all results are recorded in permanent machine state (register file, memory, and PC).
Pipelined dataflow:
The details

<table>
<thead>
<tr>
<th>Reg-Reg ALU</th>
<th>Reg-immed ALU</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IF**
- \( IR2 = \text{IR}[25..21]; \)
- \( PC2 = PC + 4; \)

**ID**
- \( A3 = \text{Regs}[IR2[25..21]]; B3 = \text{Regs}[20..16]; \)
- \( IR3 = IR2; PC3 = PC2; \)
- \( IM3 = IR2[15] + IR2[14..0]; \)

**EX**
- \( ALU4 = A3 + IM3; \)
- \( IR4 = IR3; PC4 = PC3; \)

**MEM**
- \( IR5 = IR4; PC5 = PC4; \)
- \( WB5 = \text{DMem}[ALU4]; \)
- \( DMem[ALU4] = MD4; \)
- \( IR5 = IR4; PC5 = PC4; \)

**WB**
- \( Din = WB; \)
- \( Din = WB; \)
- \( Din = WB; \)