CPE 631 Lecture 03: Review: Pipelining, Memory Hierarchy

Electrical and Computer Engineering
University of Alabama in Huntsville

Outline

- Pipelined Execution
- 5 Steps in MIPS Datapath
- Pipeline Hazards
  - Structural
  - Data
  - Control
Laundry Example

- Four loads of clothes: A, B, C, D
- Task: each one to wash, dry, and fold
- Resources
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” reduce speedup
Computer Pipelines

- Execute billions of instructions, so throughput is what matters
- What is desirable in instruction sets for pipelining?
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand many places in instruction format vs. registers located in same place?

A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- Memory access only via load/store instructions
- 32 32-bit GPR (R0 contains zero)
- 3-address, reg-reg arithmetic instruction; registers in same place
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS

Register-Register

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26 25</th>
<th>2120</th>
<th>16 15</th>
<th>1110</th>
<th>6 5</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td>Opx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26 25</th>
<th>2120</th>
<th>16 15</th>
<th></th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26 25</th>
<th>2120</th>
<th>16 15</th>
<th></th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26 25</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Steps of MIPS Datapath

Instruction Fetch

Instr. Decode

Execute Addr. Calc

Memory Access

Write Back

- Instruction Fetch
- Instr. Decode
- Execute Addr. Calc
- Memory Access
- Write Back

5/5
Steps of MIPS Datapath (cont’d)

- Data stationary control
  - local decode for each instruction phase / pipeline stage

Visualizing Pipeline
Instruction Flow through Pipeline

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC 1</td>
</tr>
<tr>
<td>Add R1, R2, R3</td>
</tr>
<tr>
<td>Lw R4, 0 (R2)</td>
</tr>
<tr>
<td>Sub R6, R5, R7</td>
</tr>
<tr>
<td>Xor R9, R8, R1</td>
</tr>
<tr>
<td>Nop</td>
</tr>
<tr>
<td>CC 2</td>
</tr>
<tr>
<td>Add R1, R2, R3</td>
</tr>
<tr>
<td>Lw R4, 0 (R2)</td>
</tr>
<tr>
<td>Add R1, R2, R3</td>
</tr>
<tr>
<td>Lw R4, 0 (R2)</td>
</tr>
<tr>
<td>Sub R6, R5, R7</td>
</tr>
<tr>
<td>Nop</td>
</tr>
<tr>
<td>CC 3</td>
</tr>
<tr>
<td>Sub R6, R5, R7</td>
</tr>
<tr>
<td>For R9, R8, R1</td>
</tr>
<tr>
<td>Nop</td>
</tr>
<tr>
<td>CC 4</td>
</tr>
<tr>
<td>Add R1, R2, R3</td>
</tr>
<tr>
<td>Lw R4, 0 (R2)</td>
</tr>
<tr>
<td>Add R1, R2, R3</td>
</tr>
<tr>
<td>Nop</td>
</tr>
<tr>
<td>Nop</td>
</tr>
<tr>
<td>Nop</td>
</tr>
</tbody>
</table>

DLX Pipeline Definition: IF, ID

- **Stage IF**
  - IF/ID.IR ← Mem[PC];
  - if EX/MEM.cond {IF/ID.NPC, PC ← EX/MEM.ALUOUT}
    else {IF/ID.NPC, PC ← PC + 4};

- **Stage ID**
  - ID/EX.A ← Regs[IF/ID.IR_{6...10}];
  - ID/EX.B ← Regs[IF/ID.IR_{11...15}];
  - ID/EX.Imm ← (IF/ID.IR_{16})^{16} ## IF/ID.IR_{16...31};
  - ID/EX.NPC ← IF/ID.NPC;
  - ID/EX.IR ← IF/ID.IR;
DLX Pipeline Definition: IE

- **ALU**
  - \( \text{EX/MEM.IR} \leftarrow \text{ID/EX.IR} \);
  - \( \text{EX/MEM.ALUOUT} \leftarrow \text{ID/EX.A func ID/EX.B} \) or \( \text{EX/MEM.ALUOUT} \leftarrow \text{ID/EX.A func ID/EX.Imm} \);
  - \( \text{EX/MEM.cond} \leftarrow 0 \);

- **load/store**
  - \( \text{EX/MEM.IR} \leftarrow \text{ID/EX.IR} \);
  - \( \text{EX/MEM.B} \leftarrow \text{ID/EX.B} \);
  - \( \text{EX/MEM.ALUOUT} \leftarrow \text{ID/EX.A + ID/EX.Imm} \);
  - \( \text{EX/MEM.cond} \leftarrow 0 \);

- **branch**
  - \( \text{EX/MEM.NPC} \leftarrow \text{ID/EX.A + ID/EX.Imm} \);
  - \( \text{EX/MEM.cond} \leftarrow (\text{ID/EX.A func } 0) \);

DLX Pipeline Definition: MEM, WB

- **Stage MEM**
  - **ALU**
    - \( \text{MEM/WB.IR} \leftarrow \text{EX/MEM.IR} \);
    - \( \text{MEM/WB.ALUOUT} \leftarrow \text{EX/MEM.ALUOUT} \);
  - **load/store**
    - \( \text{MEM/WB.IR} \leftarrow \text{EX/MEM.IR} \);
    - \( \text{MEM/WB.LMD} \leftarrow \text{Mem[EX/MEM.ALUOUT]} \) or \( \text{Mem[EX/MEM.ALUOUT]} \leftarrow \text{EX/MEM.B} \);

- **Stage WB**
  - **ALU**
    - \( \text{Regs[MEM/WB.IR}_{16...20}\} \leftarrow \text{MEM/WB.ALUOUT} \) or \( \text{Regs[MEM/WB.IR}_{11...15}\} \leftarrow \text{MEM/WB.ALUOUT} \);
  - **load**
    - \( \text{Regs[MEM/WB.IR}_{11...15}\} \leftarrow \text{MEM/WB.LMD} \);
Its Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

One Memory Port/Structural Hazards

![Diagram showing time (clock cycles) with instructions and memory access]

- Load
- Instr 1
- Instr 2
- Instr 3
- Instr 4

14/01/2004 UAH-CPE631
One Memory Port/Structural Hazards (cont’d)

Time (clock cycles)

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7
---|---|---|---|---|---|---
Load | IFetch | Reg | ALU | DMem | Reg | Reg
Instr 1 | | | | | | | Bubble
Instr 2 | | | | | | | Bubble
Stall | | | | | | | Bubble
Instr 3 | | | | | | | Bubble

14/01/2004  UAH-CPE631

Data Hazard on R1

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11

14/01/2004  UAH-CPE631
Three Generic Data Hazards

- **Read After Write (RAW)**
  InstrJ tries to read operand before InstrI writes it
  
  \[ I: \text{add } r1, r2, r3 \]
  \[ J: \text{sub } r4, r1, r3 \]

- **Caused by a “Dependence”** (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- **Write After Read (WAR)**
  InstrJ writes operand before InstrI reads it
  
  \[ I: \text{sub } r4, r1, r3 \]
  \[ J: \text{add } r1, r2, r3 \]
  \[ K: \text{mul } r6, r1, r7 \]

- **Called an “anti-dependence”** by compiler writers. This results from reuse of the name “r1”.
- **Can’t happen in MIPS 5 stage pipeline because:**
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Three Generic Data Hazards

- **Write After Write (WAW)**
  InstrJ writes operand before InstrI writes it.

  \[
  \text{I: sub r1, r4, r3} \\
  \text{J: add r1, r2, r3} \\
  \text{K: mul r6, r1, r7}
  \]

- Called an “output dependence” by compiler writers
- This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

Forwarding to Avoid Data Hazard

- Time (clock cycles)

  `[Diagram of forwarding to avoid data hazard.]`

  - `add r1, r2, r3`
  - `sub r4, r1, r3`
  - `and r6, r1, r7`
  - `or r8, r1, r9`
  - `xor r10, r1, r11`
**HW Change for Forwarding**

Diagram showing the flow of data through the processor pipeline stages: ID/EX, EX/MEM, MEM/WB.

- Forward R1 from EX/MEM.ALUOUT to ALU input (lw)
- Forward R1 from MEM/WB.ALUOUT to ALU input (sw)
- Forward R4 from MEM/WB.LMD to memory input (memory output to memory input)

**Forwarding to DM input**

- Forward R1 from EX/MEM.ALUOUT to ALU input (lw)
- Forward R1 from MEM/WB.ALUOUT to ALU input (sw)
- Forward R4 from MEM/WB.LMD to memory input (memory output to memory input)

Order:

- lw R4,0(R1)
- sw 12(R1),R4

Clock cycles:

- cc1 IM
- cc2 IM
- cc3 IM
- cc4 DM
- cc5 IM
- cc6 DM
- cc7 IM

14/01/2004  UAH-CPE631  25

14/01/2004  UAH-CPE631  26
Forwarding to DM input (cont’d)

Forward R1 from MEM/WB.ALUOUT to DM input

Time (clock cycles)

1. add $R1, R2, R3$
2. sw 0($R4$), $R1$

Forward R1 from EX/MEM.ALUOUT to Zero

Time (clock cycles)

1. add $R1, R2, R3$
2. bneq $R1, 50$
3. sub $R4, R5, R6$
4. bneq $R1, 50$
Data Hazard Even with Forwarding

Time (clock cycles)

- lw r1, 0(r2)
- sub r4, r1, r6
- and r6, r1, r7
- or r8, r1, r9

Data Hazard Even with Forwarding

Time (clock cycles)

- lw r1, 0(r2)
- sub r4, r1, r6
- and r6, r1, r7
- or r8, r1, r9
Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[ a = b + c; \]
\[ d = e - f; \]

assuming \( a, b, c, d, e, \) and \( f \) in memory.

**Slow code:**

- \[ LW \ Rb, b \]
- \[ LW \ Rc, c \]
- \[ ADD \ Ra, Rb, Rc \]
- \[ SW \ a, Ra \]
- \[ LW \ Re, e \]
- \[ LW \ Rf, f \]
- \[ SUB \ Rd, Re, Rf \]
- \[ SW \ d, Rd \]

**Fast code:**

- \[ LW \ Rb, b \]
- \[ LW \ Rc, c \]
- \[ ADD \ Ra, Rb, Rc \]
- \[ LW \ Re, e \]
- \[ LW \ Rf, f \]
- \[ SUB \ Rd, Re, Rf \]
- \[ SW \ a, Ra \]
- \[ SW \ d, Rd \]

Control Hazard on Branches
Three Stage Stall

10: \[ \text{beq} \ r1, r3, 36 \]
14: \[ \text{and} \ r2, r3, r5 \]
18: \[ \text{or} \ r6, r1, r7 \]
22: \[ \text{add} \ r8, r1, r9 \]
36: \[ \text{xor} \ r10, r1, r11 \]
**Example: Branch Stall Impact**

- If 30% branch, Stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

---

**Pipelined MIPS Datapath**

- **Instruction Fetch**
- **Instr. Decode Reg. Fetch**
- **Execute Addr. Calc**
- **Memory Access**
- **Write Back**

- **Data stationary control**
  - local decode for each instruction phase / pipeline stage
Four Branch Hazard Alternatives

- **#1**: Stall until branch direction is clear
- **#2**: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction

Branch not Taken

Branch is not taken (determined during ID), we have fetched the fall-through and just continue ⇒ no wasted cycles

Branch is taken (determined during ID), restart the fetch from at the branch target ⇒ one cycle wasted
Four Branch Hazard Alternatives

#3: Predict Branch Taken
- Treat every branch as taken
- 53% MIPS branches taken on average
- But haven’t calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
- Make sense only when branch target is known before branch outcome

#4: Delayed Branch
- Define branch to take place AFTER a following instruction
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this
Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address:
    only valuable when branch taken
  - From fall through:
    only valuable when branch not taken

Scheduling the branch delay slot: From Before

- Delay slot is scheduled with an independent instruction from before the branch
- Best choice, always improves performance
Scheduling the branch delay slot: From Target

- Delay slot is scheduled from the target of the branch
- Must be OK to execute that instruction if branch is not taken
- Usually the target instruction will need to be copied because it can be reached by another path ⇒ programs are enlarged
- Preferred when the branch is taken with high probability

SUB R4,R5,R6
...
ADD R1,R2,R3
if(R1=0) then
<Delay Slot>

... 
ADD R1,R2,R3
if(R2=0) then
<Delay Slot>

SUB R4,R5,R6

Scheduling the branch delay slot: From Fall Through

- Delay slot is scheduled from the taken fall through
- Must be OK to execute that instruction if branch is taken
- Improves performance when branch is not taken

ADD R1,R2,R3
if(R2=0) then
<Delay Slot>
SUB R4,R5,R6

ADD R1,R2,R3
if(R2=0) then
<Delay Slot>
<SUB R4,R5,R6>
Delayed Branch Effectiveness

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Example: Branch Stall Impact

- Assume CPI = 1.0 ignoring branches
- Assume solution was stalling for 3 cycles
- If 30% branch, Stall 3 cycles

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>70%</td>
<td>1</td>
<td>.7 (37%)</td>
</tr>
<tr>
<td>Branch</td>
<td>30%</td>
<td>4</td>
<td>1.2 (63%)</td>
</tr>
</tbody>
</table>

=> new CPI = 1.9, or almost 2 times slower
Example 2: Speed Up Equation for Pipelining

\[ \text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst} \]

\[ \text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]

For simple RISC pipeline, CPI = 1:

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}} \]

Example 3: Evaluating Branch Alternatives (for 1 program)

- Scheduling scheme: Branch penalty, CPI, speedup v. stall
- Stall pipeline: 3, 1.42, 1.0
- Predict taken: 1, 1.14, 1.26
- Predict not taken: 1, 1.09, 1.29
- Delayed branch: 0.5, 1.07, 1.31

Conditional & Unconditional = 14%, 65% change PC
<table>
<thead>
<tr>
<th>Example 4: Dual-port vs. Single-port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A:</td>
</tr>
<tr>
<td>Dual ported memory (&quot;Harvard Architecture&quot;)</td>
</tr>
<tr>
<td>Machine B:</td>
</tr>
<tr>
<td>Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate</td>
</tr>
<tr>
<td>Ideal CPI = 1 for both</td>
</tr>
<tr>
<td>Loads&amp;Stores are 40% of instructions executed</td>
</tr>
</tbody>
</table>