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

6 PM 7 8 9 10 11 Midnight

30 40 40 40 20

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?

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

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

Steps of MIPS Datapath (cont’d)

5 Steps of MIPS Datapath

Visualizing Pipeline
Instruction Flow through Pipeline

Time (clock cycles)

Add R1, R2, R3
Nop
Nop
Add R1, R2, R3

Lw R4, 0(R2)

Sub R6, R5, R7

Xor R9, R8, R1

Instruction Flow through Pipeline

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[16...10]];
  - ID/EX.B ← Regs[IF/ID.IR[11...15]];
  - ID/EX.Imm ← (IF/ID.IR[16...31])# IF/ID.IR[16...31];
  - ID/EX.NPC ← IF/ID.NPC;
  - ID/EX.IR ← IF/ID.IR;

DLX Pipeline Definition: IE

- **ALU**
  - EX/MEM.IR ← ID/EX.IR;
  - EX/MEM.ALUOUT ← ID/EX.A func ID/EX.B; or
  - EX/MEM.ALUOUT ← ID/EX.A func ID/EX.Imm;
  - EX/MEM.cond ← 0;

- **load/store**
  - EX/MEM.IR ← ID/EX.IR;
  - EX/MEM.B ← ID/EX.B;
  - EX/MEM.ALUOUT ← ID/EX.A + ID/EX.Imm;
  - EX/MEM.cond ← 0;

- **branch**
  - EX/MEM.NPC ← ID/EX.A + ID/EX.Imm;
  - EX/MEM.cond ← (ID/EX.A func 0);

DLX Pipeline Definition: MEM, WB

- **Stage MEM**
  - ALU
    - MEM/WB.IR ← EX/MEM.IR;
    - MEM/WB.ALUOUT ← EX/MEM.ALUOUT;
  - load/store
    - MEM/WB.IR ← EX/MEM.IR;
    - MEM/WB.LMD ← Mem[EX/MEM.ALUOUT] or Mem[EX/MEM.ALUOUT] ← EX/MEM.B;

- **Stage WB**
  - ALU
    - Regs[MEM/WB.IR[16...10]] ← MEM/WB.ALUOUT; or
    - Regs[MEM/WB.IR[16...10]] ← MEM/WB.ALUOUT;
  - load
    - Regs[MEM/WB.IR[16...10]] ← MEM/WB.LMD;
It's 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

- Time (clock cycles)
  - Cycle 1: Load
  - Cycle 2: Instr 1
  - Cycle 3: Instr 2
  - Cycle 4: Instr 3
  - Cycle 5: Stall
  - Cycle 6: Instr 4
  - Cycle 7

One Memory Port/Structural Hazards (cont'd)

- Data Hazard on R1
  - Time (clock cycles)
    - IF: add r1, r2, r3
    - ID: sub r4, r1, r3
    - EX: and r6, r1, r7
    - MEM: or r8, r1, r9
    - WB: xor r10, r1, r11
Three Generic Data Hazards

- **Read After Write (RAW)**
  - InstrJ tries to read operand before InstrI writes it
  - I: add r1, r2, r3
  - J: sub r4, r1, r3
  - Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

- **Write After Read (WAR)**
  - InstrJ writes operand before InstrI reads it
  - 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

- **Write After Write (WAW)**
  - InstrJ writes operand before InstrI writes it.
  - 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)
- Forwarding to avoid data hazard
- I: sub r1, r2, r3
- J: add r2, r1, r3
- K: mul r6, r1, r7
- L: add r8, r1, r9
- M: xor r10, r1, r11
Forwarding to DM input (cont’d)

Forward R1 from MEM/WB.ALUOUT to DM input

Time (clock cycles)

Forward R1 from EX/MEM.ALUOUT to DM input

Time (clock cycles)

Forward to Zero

Forwarding to Zero

Forward R1 from EX/MEM.ALUOUT to Zero

Time (clock cycles)

Forward R1 from EX/MEM.ALUOUT to Zero

Time (clock cycles)
Data Hazard Even with Forwarding

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.

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:
\[
\begin{align*}
&\text{LW} \quad \text{Rb}, b \\
&\text{LW} \quad \text{Rc}, c \\
&\text{ADD} \quad \text{Ra}, \text{Rb}, \text{Rc} \\
&\text{SW} \quad a, \text{Ra} \\
&\text{LW} \quad \text{Re}, e \\
&\text{LW} \quad \text{Rf}, f \\
&\text{SUB} \quad \text{Rd}, \text{Re}, \text{Rf} \\
&\text{SW} \quad d, \text{Rd}
\end{align*}
\]

Fast code:
\[
\begin{align*}
&\text{LW} \quad \text{Rb}, b \\
&\text{LW} \quad \text{Rc}, c \\
&\text{LW} \quad \text{Re}, e \\
&\text{ADD} \quad \text{Ra}, \text{Rb}, \text{Rc} \\
&\text{ADD} \quad \text{Ra}, \text{Re}, \text{Rf} \\
&\text{LW} \quad \text{Rf}, f \\
&\text{SUB} \quad \text{Rd}, \text{Re}, \text{Rf} \\
&\text{SW} \quad d, \text{Rd}
\end{align*}
\]

Control Hazard on Branches

Three Stage Stall

10: beq r1, r3, 36
14: and r2, r3, r5
18: or r6, r1, r7
22: add r8, r1, r2
36: 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

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

Pipelined MIPS Datapath

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

Branch not Taken

- Time [clocks]
- 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

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

Four Branch Hazard Alternatives

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

Scheduling the branch delay slot:

**From Before**

- Delay slot is scheduled with an independent instruction from before the branch
- Best choice, always improves performance

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

```
if (R2 = 0) then <ADD R1, R2, R3>
```
### 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>
```

Becomes
```
ADD R1, R2, R3
if (R2=0) then
  <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

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

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

For simple RISC pipeline, CPI = 1:

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

Example 3: Evaluating Branch Alternatives

(for 1 program)

- Scheduling Branch CPI speedup v. stall
  - Scheme penalty 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

Example 4: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads&Stores are 40% of instructions executed