CPE 631 Review: Pipelining

Electrical and Computer Engineering
University of Alabama in Huntsville
Aleksandar Milenkovic, milenka@ece.uah.edu
http://www.ece.uah.edu/~milenka

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

Laundry Example (by David Patterson)
- 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

Task Order

30 40 40 40 20

A B C D

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Op | Rs1| Rs2| Rd | 110| 6 | 5 | 0  |

#### Register-Immediate

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Op | Rs1| Rd | immediate | 0  |

#### Branch

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Op | Rs1| Rd | immediate | immediate | 0  |

#### Jump / Call

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Op | | | target | 0  |

### 5 Steps of MIPS Datapath

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

#### Data stationary control

- Local decode for each instruction phase/pipeline stage

### Visualizing Pipeline

- Time (clock cycles)
- Instr. Order
- CC 1, CC 2, CC 3, CC 4, CC 5, CC 6, CC 7
Instruction Flow through Pipeline

Time (clock cycles)

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.IR8...10];
- ID/EX.B ← Regs[IF/ID.IR11...15];
- ID/EX.Imm ← (IF/ID.IR16)16 # if IF/ID.IR16...31;
- ID/EX.NPC ← IF/ID.NPC;
- ID/EX.IR ← IF/ID.IR;

Stage MEM
- 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.Aluout ← ID/EX.NPC + (ID/EX.Imm<< 2);
  - EX/MEM.cond ← (ID/EX.A func 0);

Stage WB
- 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;

DLX Pipeline Definition: IF, ID

DLX Pipeline Definition: IE

DLX Pipeline Definition: MEM, WB
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

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

Data Hazard on R1

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

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

  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.

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

  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

- **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

- **Forwarding**
  
  \[ \text{add r1,r2,r3} \]
  \[ \text{sub r4,r1,r3} \]
  \[ \text{and r6,r1,r7} \]
  \[ \text{or r8,r1,r9} \]
  \[ \text{xor r10,r1,r11} \]

  Time (clock cycles)

  - All instructions take 5 stages, and
  - Writes are always in stage 5
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)

Time (clock cycles)

- Forward R1 from MEM/WB.ALUOUT to DM input
- Forward R1 from EX/MEM.ALUOUT to Zero

Forwarding to Zero

- Forward R1 from EX/MEM.ALUOUT to Zero
- Forward R1 from MEM/WB.ALUOUT to Zero

Instruction Order

Forwarding to DM input (cont’d)

Forward R1 from MEM/WB.ALUOUT to DM input

Time (clock cycles)

- Forward R1 from MEM/WB.LMD to memory input (memory output to memory input)
Data Hazard Even with Forwarding

Time (clock cycles)

Instr Order

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

Data Hazard Even with Forwarding

Time (clock cycles)

Instr Order

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  | Rb,c |
| ADD | Ra,Rb,Rc |
| LW  | Rb,e |
| LW  | Re,e |
| SW  | a,Re |
| LW  | Rf,f |
| SUB | Rd,Re,Rf |
| SW  | d,Rd |

Fast code:

| LW  | Rb,b |
| LW  | Rb,c |
| LW  | Rb,e |
| ADD | a,Re |
| LW  | Rf,f |
| SUB | Rd,Re,Rf |
| SW  | d,Rd |

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,r9
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
  - 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

Branch not Taken

Branch is untaken (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
  - branch instruction
  - sequential successor
  - Branch delay of length n
  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- 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>

Becomes

→ 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

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>0.7 (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 Branch CPI speedup v. 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

Extended MIPS Pipeline

- DLX pipe with three unpipelined, FP functional units

In reality, the intermediate results are probably not cycled around the EX unit; instead the EX stages has some number of clock delays larger than 1
Extended MIPS Pipeline (cont’d)

- **Initiation or repeat interval**: number of clock cycles that must elapse between issuing two operations
- **Latency**: the number of intervening clock cycles between an instruction that produces a result and an instruction that uses the result

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data Memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP/Integer Multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP/Integer Divide</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

Extended MIPS Pipeline (cont’d)

- Multiple outstanding FP operations
  - FP/I Adder and Multiplier are fully pipelined
  - FP/I Divider is not pipelined
- Pipeline timing for independent operations

Hazards and Forwarding in Longer Pipes

- Structural hazard: divide unit is not fully pipelined
  - detect it and stall the instruction
- Structural hazard: number of register writes can be larger than one due to varying running times
- WAW hazards are possible
- Exceptions!
  - instructions can complete in different order than they were issued
  - RAW hazards will be more frequent
Examples

- Stalls arising from RAW hazards

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Three instructions that want to perform a write back to the FP register file simultaneously

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D F2, F4, F6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

Solving Register Write Conflicts

- First approach: track the use of the write port in the ID stage and stall an instruction before it issues
  - use a shift register that indicates when already issued instructions will use the register file
  - if there is a conflict with an already issued instruction, stall the instruction for one clock cycle
  - on each clock cycle the reservation register is shifted one bit
- Alternative approach: stall a conflicting instruction when it tries to enter MEM or WB stage
  - we can stall either instruction
  - e.g. give priority to the unit with the longest latency
  - Pros: does not require to detect the conflict until the entrance of MEM or WB stage
  - Cons: complicates pipeline control; stalls now can arise from two different places

Solving WAW Hazards

- First approach: delay the issue of load instruction until ADD.D enters MEM
- Second approach: stamp out the result of the ADD.D by detecting the hazard and changing the control so that ADDD does not write; LD issues right away
- Detect hazard in ID when LD is issuing
  - stall LD, or
  - make ADDD no-op
- Luckily this hazard is rare
Hazard Detection in ID Stage

- Possible hazards
  - hazards among FP instructions
  - hazards between an FP instruction and an integer instr.
    - FP and integer registers are distinct, except for FP load-stores, and FP-integer moves
  - Assume that pipeline does all hazard detection in ID stage

Hazard Detection in ID Stage (cont’d)

- Check for structural hazards
  - wait until the required functional unit is not busy and make sure that the register write port is available
- Check for RAW data hazards
  - wait until source registers are not listed as pending destinations in a pipeline register that will not be available when this instruction needs the result
- Check for WAW data hazards
  - determine if any instruction in A1, A4, M1, M7, D has the same register destination as this instruction; if so, stall the issue of the instruction in ID

Forwarding Logic

- Check if the destination register in any of EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB pipeline registers is one of the source registers of a FP instruction
- If so, the appropriate input multiplexer will have to be enabled so as to choose the forwarded data