ENGN1640: Design of Computing Systems
Topic 05: Pipeline Processor Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2019

[ material from Patterson & Hennessy and Harris]
Pipelining analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

- Four loads:
  - Speedup
    \[ \frac{16}{7} = 2.3 \]

- Non-stop:
  - Ideal Speedup
    \[ \frac{4n}{n + 3} \approx 4 \]
    - number of stages
Pipelined RISCV processor

- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages
Pipeline Performance

• Assume time for stages is
  – 100ps for register read or write
  – 200ps for other stages

• Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Single-Cycle vs Pipelined

**Single-cycle \((T_c = 800\text{ps})\)**

- Program execution order (in instructions)
  - \(x_1, 100(x4)\)
  - \(x_2, 200(x4)\)
  - \(x_3, 400(x4)\)

**Pipelined \((T_c = 200\text{ps})\)**

- Program execution order (in instructions)
  - \(x_1, 100(x4)\)
  - \(x_2, 200(x4)\)
  - \(x_3, 400(x4)\)
RISC-V Pipelined Datapath

Right-to-left flow leads to hazards
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
Pipeline Operation

• Cycle-by-cycle flow of instructions through the pipelined datapath
  – “Single-clock-cycle” pipeline diagram
    • Shows pipeline usage in a single cycle
    • Highlight resources used
  – c.f. “multi-clock-cycle” diagram
    • Graph of operation over time
• We’ll look at “single-clock-cycle” diagrams for load & store
IF for Load, Store, …
ID for Load, Store, ...
EX for Load
MEM for Load
WB for Load

Wrong register number
Corrected Datapath for Load
EX for Store
MEM for Store
WB for Store
Multi-Cycle Pipeline Diagram

- Traditional form
Multi-Cycle Pipeline Diagram
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle
Pipelined Control (Simplified)
Pipelined Control

• Control signals derived from instruction
  – As in single-cycle implementation
Pipelined Control
Pipeline speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = Time between instructions_{nonpipelined} / Number of stages
  - Ideal speedup (n instructions and s stages) \( \frac{sn}{n + s - 1} \)
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
  - Branches will also reduce the speedup
  - Added pipeline registers reduce the speedup
Pipelining hazards

- Situations that prevent starting the next instruction in the next cycle
  1. Structural hazards
     - A required resource is busy
  2. Data hazard
     - Need to wait for previous instruction to complete its data read/write
  3. Control hazard
     - Deciding on control action depends on previous instruction
1. Structural hazards

- Conflict for use of a resource
- In RISCV pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add x19, x0, x1`
  - `sub x2, x19, x3`

Or
  - `lw x1, 0(x2)`
  - `sub x4, x1, x5`
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath
Load-Use Data Hazard

• Can’t always avoid stalls by forwarding
  – If value not computed when needed
  – Can’t forward backward in time!

Program execution order (in instructions)

Id x1, 0(x2)

sub x4, x1, x5
Handling data hazards

A. Compile-time techniques
B. Forward data at run time
C. Stall the processor at run time
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for \( a = b + e; c = b + f; \)

\[
\begin{array}{l}
\text{lw} \quad x1, 0(x0) \\
\text{lw} \quad x2, 8(x0) \\
\text{add} \quad x3, x1, x2 \\
\text{sw} \quad x3, 24(x0) \\
\text{lw} \quad x4, 16(x0) \\
\text{add} \quad x5, x1, x4 \\
\text{sw} \quad x5, 32(x0)
\end{array}
\]

13 cycles

\[
\begin{array}{l}
\text{lw} \quad x1, 0(x0) \\
\text{lw} \quad x2, 8(x0) \\
\text{lw} \quad x4, 16(x0) \\
\text{add} \quad x3, x1, x2 \\
\text{sw} \quad x3, 24(x0) \\
\text{add} \quad x5, x1, x4 \\
\text{sw} \quad x5, 32(x0)
\end{array}
\]

11 cycles
Data Hazards in ALU Instructions

• Consider this sequence:
  
  sub  x2, x1, x3  
  and x12, x2, x5  
  or  x13, x6, x2  
  add x14, x2, x2  
  sd  x15, 100(x2)

• We can resolve hazards with forwarding
  – How do we detect when to forward?
Dependencies & Forwarding

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>Value of register x2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC 1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

```
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(X2)
```
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2
Detecting the Need to Forward

• But only if forwarding instruction will write to a register!
  – EX/MEM.RegWrite, MEM/WB.RegWrite

• And only if Rd for that instruction is not x0
  – EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0
Forwarding Paths
# Forwarding Conditions

<table>
<thead>
<tr>
<th>Mux control</th>
<th>Source</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardA = 00</td>
<td>ID/EX</td>
<td>The first ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardA = 10</td>
<td>EX/MEM</td>
<td>The first ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardA = 01</td>
<td>MEM/WB</td>
<td>The first ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
<tr>
<td>ForwardB = 00</td>
<td>ID/EX</td>
<td>The second ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardB = 10</td>
<td>EX/MEM</td>
<td>The second ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardB = 01</td>
<td>MEM/WB</td>
<td>The second ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
</tbody>
</table>
Double Data Hazard

• Consider the sequence:
  add \( x_1, x_1, x_2 \)
  add \( x_1, x_1, x_3 \)
  add \( x_1, x_1, x_4 \)

• Both hazards occur
  – Want to use the most recent

• Revise MEM hazard condition
  – Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
Datapath with Forwarding
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble
How to Stall the Pipeline

• Force control values in ID/EX register to 0
  – EX, MEM and WB do nop (no-operation)
• Prevent update of PC and IF/ID register
  – Using instruction is decoded again
  – Following instruction is fetched again
  – 1-cycle stall allows MEM to read data for ld
    • Can subsequently forward to EX stage
Load-Use Data Hazard

Program execution order (in instructions)

Id x2, 20(x1)
and becomes nop
and x4, x2, x5
or x8, x2, x6
add x9, x4, x2

Time (in clock cycles)
CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9  CC 10

Stall inserted here
Datapath with Hazard Detection
Stalls and Performance

The BIG Picture

• Stalls reduce performance
  – But are required to get correct results
• Compiler can arrange code to avoid hazards and stalls
  – Requires knowledge of the pipeline structure
Data hazard summary

- Compiler can arrange code to avoid hazards and stalls. Requires knowledge of the pipeline structure.
- Forwarding can sometimes avoid stalls at the expense of extra hardware complexity.
- Stalls reduce performance by increasing the average cycles per instruction (CPI). But sometimes are absolutely necessary to get correct results.
3. Control Hazards

• Branch determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on ID stage of branch

• In RISC-V pipeline
  – Need to compare registers and compute target early in the pipeline
  – Add hardware to do it in ID stage
3. Control hazards

- If branch outcome determined in MEM

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:    sub    x10, x4, x8
40:    beq    x1, x3, 16  // PC-relative branch
        // to 40+16*2=72
44:    and    x12, x2, x5
48:    orr    x13, x2, x6
52:    add    x14, x4, x2
56:    sub    x15, x6, x7
...
72:    ld      x4, 50(x7)
```
Example: Branch Taken

and x12, x2, x5

beq x1, x3, 16

sub x10, x4, x8

before<1>

before<2>

Clock 3
Example: Branch Taken
Branch prediction

• Ideal pipelined processor: CPI = 1
• Branch misprediction increases CPI

• **Static branch prediction:**
  – Always not taken
  – Always taken
  – Check direction of branch (forward or backward): If backward, predict taken; else, predict not taken

• **Dynamic branch prediction:**
  – Make a dynamic based on history of branches

In all cases, branch must be executed to see if prediction was correct → if not, flush instructions and resume from correct direction!
Eliminating 2-cycle stall for taken-prediction policy with branch target buffer

- Even with predictor, still need to calculate the target address
  - 2-cycle penalty for a taken branch

- **Branch target buffer**
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately – no 2-cycle penalty
Branch target buffer

Branch PC = Branch target address

Yes: it is a branch and PC = branch target address

No: it is not a branch
Next PC = PC+4
2. Dynamic branch prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction:
  - **Branch prediction buffer** (aka branch history table) indexed by recent branch instruction addresses and stores outcome (taken/not taken)
  - To execute a branch:
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction
1-bit branch predictor

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop
- Prediction bits are added to BTB entries

```
addi x11, x0, 0           ; R1 = sum
addi x10, x0, 0           ; R0 = i
addi x12, x0, 10

FOR:                       ; for (i=0; i<10; i=i+1)
    bge R0, x12, DONE
    add x11, x11, x10       ; sum = sum + i
    addi x10, x10, 1
    beq x0, x0, FOR

DONE
```
Problem with 1-bit predictor

- Inner loop branches mispredicted twice!

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-bit predictor

- Only change prediction on two successive mispredictions
Summary

- Pipelining for speedup
- Ideal speedup = number of stages, but actual speedup depends on delay balance between stages (clock frequency), delays introduced by pipeline registers, and number of stalls (CPI).
- Hazards (structural, data, and control) can increase CPI
- Hazards can be eliminated or mitigated using code reorganization, stalling, flushing, forwarding / bypassing
- Branch prediction, branch prediction buffer, branch target buffer can reduce stalls arising from control hazards