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 2016

[ 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 ARM processor

• Temporal parallelism
• Divide single-cycle processor into 5 stages:
  – Fetch
  – Decode
  – Execute
  – Memory
  – Writeback
• Add pipeline registers between stages
Single-Cycle vs Pipelined

**Single-Cycle**

![Single-Cycle Diagram]

**Pipelined**

![Pipelined Diagram]
Pipeline datapath abstraction

LDR R2, [R0, #40]
ADD R3, R9, R10
SUB R4, R1, R5
AND R5, R12, R13
STR R6, [R1, #20]
ORR R7, R11, #42
Single-cycle & pipelined datapath

- **WA3** must arrive at same time as **Result**
- Register file written on falling edge of **CLK**
Remove adder by using PCPlus4F after PC has been updated to PC+4
Assumes writing happens (e.g., in first half of clock cycle) before reading
Tracing LDR in its journey: 1st cycle
Tracing LDR in its journey: 2nd cycle
Tracing LDR in its journey: 3rd cycle
Tracing LDR in its journey: 4th cycle
Tracing LDR in its journey: 5th cycle
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>LDR</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>STR</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>data op</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>Branch</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Single-cycle versus pipeline performance

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

Program execution order (in instructions)

LDR R1, [R5]
LDR R2, [R6]
LDR R3, [R7]

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

Program execution order (in instructions)

LDR R1, [R5]
LDR R2, [R6]
LDR R3, [R7]
Pipeline speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions\(_{\text{pipelined}}\) = Time between instructions\(_{\text{nonpipelined}}\) / 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
Reminder of single-cycle control
Modifications to pipeline control

- Control signals derived from instruction
  - Same as in single-cycle implementation
  - Control delayed to proper pipeline stage
- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage
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 ARMs 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
2. Data Hazards: compute-use

ADD R1, R4, R5
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
Data Hazard: load-use

LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
Handling data hazards

A. Compile-time techniques
B. Forward data at run time
C. Stall the processor at run time
A. Data hazard elimination using compile-time techniques (nop)

- Insert enough nops until result is ready (wastes cycles)
A. Data hazard elimination using compile-time techniques (code rescheduling)

- Reorder code to avoid use of load result in the next instruction
- Compiler must be aware of pipeline structure

```
ADD R1, R4, R5
ADD R8, R1, R3
AND R9, R6, R1
SUB R10, R2, R7
```

Rescheduling saved one cycles!
B. Data hazard elimination using data forwarding/bypassing during runtime

- Don’t wait for result to be stored in a register ➔ forward the results whenever the results
- Requires extra connections in the datapath

ADD R1, R4, R5
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7

- Check if register read in Execute stage matches register written in Memory or Writeback stage
- If so, forward result
Circuitry for forwarding
To forward or not to forward!

- **Execute** stage register matches **Memory** stage register?
  
  Match\_1E\_M = (RA1E == WA3M)
  
  Match\_2E\_M = (RA2E == WA3M)

- **Execute** stage register matches **Writeback** stage register?
  
  Match\_1E\_W = (RA1E == WA3W)
  
  Match\_2E\_W = (RA2E == WA3W)

- If it matches, forward result:

  ```
  if (Match\_1E\_M • RegWriteM) ForwardAE = 10;
  else if (Match\_1E\_W • RegWriteW) ForwardAE = 01;
  else ForwardAE = 00;
  ```

  **ForwardBE same but with Match2E**
Double data hazard

- Consider the sequence:
  
  $$
  \text{ADD } R1, R1, R2 \\
  \text{ADD } R1, R1, R3 \\
  \text{ADD } R1, R1, R4 \\
  $$

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Give priority to EX results. That is, only fwd from MEM if EX hazard condition isn’t true
Pipelining hazards

- Structural hazards
  2. Data hazard
  3. Control hazard

  - Compile-time techniques
  - Forward data at run time
  - Stall the processor at run time
Stalling

LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
Forwarding is not going to eliminate all hazards

LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
C. Data hazard elimination by stalling

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9  CC 10

Program execution order (in instructions)

LDR R1, [R4, #40] IM Reg
and becomes nop

AND R8, R1, R3

ORR R9, R6, R1

SUB R10, R1, R7

clock cycle wasted – necessary for correctness
Stalling HW
To stall or not to stall!

- Is either source register in the Decode stage the same as the one being written in the Execute stage?

\[ Match_{12D\_E} = (RA1D == WA3E) \ | \ (RA2D == WA3E) \]

- Is a \texttt{LDR} in the Execute stage AND \( Match_{12D\_E} \)?

\[ \text{ldrstall} = Match_{12D\_E} \bullet \text{MemtoRegE} \]

\[ \text{StallF} = \text{StallD} = \text{FlushE} = \text{ldrstall} \]
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

• **B:**
  – branch not determined until the Writeback stage of pipeline
  – Instructions after branch fetched before branch occurs
  – These 4 instructions must be flushed if branch happens

• **Writes to PC (R15) similar**
Control hazards

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken (4)
  - May be reduced by determining BTA earlier
Early branch resolution

• **Determine BTA in Execute stage**
  – Branch misprediction penalty = 2 cycles

• **Hardware changes**
  – Add a branch multiplexer before PC register to select BTA from ALUResultE
  – Add BranchTakenE select signal for this multiplexer (only asserted if branch condition satisfied)
  – PCSrcW now only asserted for writes to PC
Pipelined processor with early BTA
Control hazards with early BTA

20  B 3C
24  AND R8, R1, R3
28  ORR R9, R6, R1
2C  SUB R10, R1, R7
30  SUB R11, R1, R8
34  ...
64  ADD R12, R3, R4

Flush these instructions
Control stalling logic

- **PCWrPendingF = 1** if write to PC in Decode, Execute or Memory
  \[ PCWrPendingF = PCSrcD + PCSrcE + PCSrcM \]

- **Stall Fetch** if **PCWrPendingF**
  \[ StallF = ldrStallD + PCWrPendingF \]

- **Flush Decode** if **PCWrPendingF** OR PC is written in Writeback OR branch is taken
  \[ FlushD = PCWrPendingF + PCSrcW + BranchTakenE \]

- **Flush Execute** if branch is taken
  \[ FlushE = ldrStallD + BranchTakenE \]

- **Stall Decode** if **ldrStallD** (as before)
  \[ StallD = ldrStallD \]
ARM Pipelined Processor with Hazard Unit
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

- No: it is not a branch
  - Next PC = PC + 4

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

MUX → PC → IM → Pipeline reg → rest of pipeline

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

[floorplan of Pentium processor]
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

```asm
MOV R1, #0 ; R1 = sum
MOV R0, #0 ; R0 = i

FOR ; for (i=0; i<10; i=i+1)
    CMP R0, #10
    BGE DONE
    ADD R1, R1, R0 ; sum = sum + i
    ADD R0, R0, #1
    B 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