Lecture 13: Pipelining

John Wawrzynek & Nick Weaver

http://inst.eecs.berkeley.edu/~cs61c/sp18
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control
• Superscalar processors
Recap: Pipelining with RISC-V

Instruction sequence:
- `add t0, t1, t2`
- `or t3, t4, t5`
- `sll t6, t0, t3`

### Single Cycle
- **Timing**:
  - $t_{step} = 100 \ldots 200$ ps
  - $t_{cycle} = 200$ ps
- Register access only 100 ps

### Pipelining
- **Instruction time**, $t_{instruction}$:
  - $= t_{cycle} = 800$ ps
  - 1000 ps
- **Clock rate**, $f_s$:
  - $1/800$ ps = 1.25 GHz
  - $1/200$ ps = 5 GHz
- **Relative speed**:
  - 1 x
  - 4 x
RISC-V Pipeline

Resource use in a particular time slot

Resource use of instruction over time

add t0, t1, t2
or t3, t4, t5
slt t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
addi t2, t2, 1

Instruction sequence

$\tau_{\text{cycle}} = 200 \text{ ps}$

$\tau_{\text{instruction}} = 1000 \text{ ps}$

Lecture 13: Pipelining

CS 61c
Single-Cycle RISC-V RV32I Datapath
Pipelining RISC-V RV32I Datapath

Instruction Fetch (F)
Instruction Decode/Register Read (D)
ALU Execute (X)
Memory Access (M)
Write Back (W)
Pipelined RISC-V RV32I Datapath

Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline

Must pipeline instruction along with data, so control operates correctly in each stage
Each stage operates on different instruction

Pipeline registers separate stages, hold data for each instruction in flight
The table below shows the changes in different metrics when pipelining the single-cycle processor. The changes are categorized as decrease, increase, or same. Fill in the blanks to indicate the correct changes for each metric:

<table>
<thead>
<tr>
<th></th>
<th>Instructions (\text{/program})</th>
<th>Cycles (\text{/instruction})</th>
<th>Time (\text{/cycle})</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>decrease</td>
<td>decrease</td>
<td>same</td>
</tr>
<tr>
<td>B</td>
<td>same</td>
<td>increase</td>
<td>decrease</td>
</tr>
<tr>
<td>C</td>
<td>same</td>
<td>same</td>
<td>decrease</td>
</tr>
<tr>
<td>D</td>
<td>increase</td>
<td>decrease</td>
<td>increase</td>
</tr>
</tbody>
</table>

Pipelining the single-cycle processor can increase processor performance by:

- Fill in the blanks in the table with the correct changes.
- Understand the relationship between the number of instructions, cycles per instruction, and time per cycle when pipelining is applied to the processor.
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control
• Superscalar processors
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages
Hazards Ahead
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control
• Superscalar processors
Structural Hazard

• **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource

• **Solution 1:** Instructions take turns to use resource, some instructions have to stall

• **Solution 2:** Add more hardware to machine

• Can always solve a structural hazard by adding more hardware
Regfile Structural Hazards

• Each instruction:
  – can read up to two operands in decode stage
  – can write one value in writeback stage

• Avoid structural hazard by having separate “ports”
  – two independent read ports and one independent write port

• Three accesses per cycle can happen simultaneously
Structural Hazard: Memory Access

- Instruction and data memory used simultaneously
  - ✓ Use two separate memories

Instruction sequence:

1. add t0, t1, t2
2. or t3, t4, t5
3. slt t6, t0, t3
4. sw t0, 4(t3)
5. lw t0, 8(t3)
Instruction and Data Caches

Caches: small and fast “buffer” memories
Structural Hazards – Summary

• Conflict for use of a resource
• In RISC-V pipeline with a single memory
  – Load/store requires data access
  – Without separate memories, instruction fetch would have to *stall* for that cycle
    ▪ All other operations in pipeline would have to wait
• Pipelined datapaths require separate instruction/data memories
  – Or separate instruction/data caches
• RISC ISAs (including RISC-V) designed to avoid structural hazards
  – e.g. at most one memory access/instruction
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control
• Superscalar processors
Data Hazard: Register Access

- Separate ports, but what if write to same value as read?
- Does `sw` in the example fetch the old or new value?

```
add t0, t1, t2
or t3, t4, t5
slt t6, t4, t3
sw t0, 4(t3)
lw t0, 8(t3)
```
Register Access Policy

- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value
- Indicated in diagram by shading

Might not always be possible to write then read in same cycle, especially in high-frequency designs.
Data Hazard: ALU Result

s0 holds “5” then add instr changes s0 to “9”

Value of s0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>5/9</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

- add s0, t0, t1
- sub t2, s0, t0
- or t6, s0, t3
- xor t5, t1, s0
- sw s0, 8(t3)

Without some fix, sub and or will calculate wrong result!
Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction
  - add $s0, t0, t1
  - sub $t2, $s0, $t3

- Bubble:
  - stall dependent instruction
    - effectively NOP: affected pipeline stages do “nothing”
Stalls and Performance

• Stalls reduce performance
  – But stalls are required to get correct results

• Compiler could try to arrange code to avoid hazards and stalls
  – Requires knowledge of the pipeline structure
Solution 2: Forwarding

Value of \( t_0 \)

- Add \( t_0, t_1, t_2 \)
- Or \( t_3, t_0, t_5 \)
- Sub \( t_6, t_0, t_3 \)
- Xor \( t_5, t_1, t_0 \)
- Sw \( t_0, 8(t_3) \)

Forwarding: grab operand from pipeline stage, rather than register file
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
1) Detect Need for Forwarding (example)

add \texttt{t0}, \texttt{t1}, \texttt{t2}

or \texttt{t3}, \texttt{t0}, \texttt{t5}

sub \texttt{t6}, \texttt{t0}, \texttt{t3}

Compare destination of older instructions in pipeline with sources of new instruction in decode stage. Must ignore writes to \texttt{x0}!
Example Forwarding Path

Forwarding Control Logic

Same idea extends to rs2, and to instruction inst_D, inst_M pairing
Administrivia

• *Project 3.1 still due next Wednesday (3/7)*
• Homework 2 due Friday (11:59PM)
• Project party on both *Monday (8-10, Cory 293)* and Wednesday (7-10, Cory 293)
• Guerrilla session Tonight 7-9pm, Barrows 20!
• Midterm 2, March 20, is moved to 8-10PM (was 7-9 on the website)
  • Alternative exam earlier, 6-8PM (so people don’t need to be in exams until midnight :)
  • submit exam conflict form if they haven’t
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control
• Superscalar processors
Load Data Hazard

1 cycle stall unavoidable
forward
unaffected
Stall Pipeline

repeat and instruction and forward
Lecture 13: Pipelining

lw Data Hazard

• Slot after a load is called a load delay slot
  – If that instruction uses the result of the load, then the hardware will stall for one cycle
  – Equivalent to inserting an explicit nop in the slot
    ▪ except the latter uses more code space
  – Performance loss!

• Idea:
  – Put unrelated instruction into load delay slot
  – No performance loss!
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!

- RISC-V code for \( D = A + B; \ E = A + C; \)

```
<table>
<thead>
<tr>
<th>Original Order:</th>
<th>Alternative:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw t1, 0(t0)</td>
<td>lw t1, 0(t0)</td>
</tr>
<tr>
<td>lw t2, 4(t0)</td>
<td>lw t2, 4(t0)</td>
</tr>
<tr>
<td>add t3, t1, t2</td>
<td>add t3, t1, t2</td>
</tr>
<tr>
<td>sw t3, 12(t0)</td>
<td>sw t3, 12(t0)</td>
</tr>
<tr>
<td>lw t4, 8(t0)</td>
<td>lw t4, 8(t0)</td>
</tr>
<tr>
<td>add t5, t1, t4</td>
<td>add t5, t1, t4</td>
</tr>
<tr>
<td>sw t5, 16(t0)</td>
<td>sw t5, 16(t0)</td>
</tr>
</tbody>
</table>
```

13 cycles
11 cycles
Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors
Control Hazards

beq t0, t1, label
sub t2, s0, t5
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)

executed regardless of branch outcome!
executed regardless of branch outcome!!!
PC updated reflecting branch outcome
Observation

• If branch not taken, then instructions fetched sequentially after branch are correct

• If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs
Kill Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx

Taken branch
Convert to NOP
Convert to NOP
PC updated reflecting branch outcome
Reducing Branch Penalties

• Every taken branch in simple pipeline costs 2 dead cycles
• To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
• Only flush pipeline if branch prediction was incorrect
Branch Prediction

beq t0, t1, label

label: .....
Agenda

• RISC-V Pipeline
• Pipeline Control
• Hazards
  – Structural
  – Data
    ▪ R-type instructions
    ▪ Load
  – Control

• Superscalar processors
Increasing Processor Performance

1. Clock rate
   - Limited by technology and power dissipation

2. Pipelining
   - “Overlap” instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards (CPI > 1)

3. Multi-issue “super-scalar” processor
   - Multiple execution units (ALUs)
     - Several instructions executed simultaneously
     - CPI < 1 (ideally)
Superscalar Processor

P&H p. 340
Benchmark: CPI of Intel Core i7

P&H p. 350

CPI = 1

CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.
In Conclusion

• Pipelining increases throughput by overlapping execution of multiple instructions

• All pipeline stages have same duration
  – Choose partition that accommodates this constraint

• Hazards potentially limit performance
  – Maximizing performance requires programmer/compiler assistance
  – E.g. Load and Branch delay slots

• Superscalar processors use multiple execution units for additional instruction level parallelism
  – Performance benefit highly code dependent
Extra Slides
Pipelining and ISA Design

• RISC-V ISA designed for pipelining
  – All instructions are 32-bits
    ▪ Easy to fetch and decode in one cycle
    ▪ Versus x86: 1- to 15-byte instructions
  – Few and regular instruction formats
    ▪ Decode and read registers in one step
  – Load/store addressing
    ▪ Calculate address in 3\(^{rd}\) stage, access memory in 4\(^{th}\) stage
  – Alignment of memory operands
    ▪ Memory access takes only one cycle
Superscalar Processor

• Multiple issue “superscalar”
  – Replicate pipeline stages ⇒ multiple pipelines
  – Start multiple instructions per clock cycle
  – CPI < 1, so use Instructions Per Cycle (IPC)
  – E.g., 4GHz 4-way multiple-issue
    ▪ 16 BIPS, peak CPI = 0.25, peak IPC = 4
  – Dependencies reduce this in practice

• “Out-of-Order” execution
  – Reorder instructions dynamically in hardware to reduce impact of hazards

• CS152 discusses these techniques!