CS61c Fall 2021
Lecture 14
Pipelining RISC-V
Review

- **Controller**
  - Tells universal datapath how to execute each instruction

- **Instruction timing**
  - Set by instruction complexity, architecture, technology
  - Pipelining increases clock frequency, “instructions per second”
    - But does not reduce time to complete instruction

- **Performance measures**
  - Different measures depending on objective
    - Response time
    - Jobs / second
    - Energy Efficiency (joules/operation, joules/instruction)
Processor

Processor

Control

Datapath

PC

Registers

Arithmetic & Logic Unit (ALU)

Memory

Enable?

Read/Write

Address

Write

Data

Read

Data

Program

Bytes

Data

Processor-Memory Interface
Pipelining doesn’t help *latency* of single task, it helps *throughput* of entire workload.

- Multiple tasks operating simultaneously using different resources.
- Potential speedup = Number pipe stages.
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example.
  - With lots of laundry, approaches 4X.
Agenda

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

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

### Single Cycle vs. Pipelining

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>t_{step}</code> = 100 ... 200 ps</td>
<td></td>
<td><code>t_{cycle} = 200 ps</code></td>
</tr>
<tr>
<td>Register access only 100 ps</td>
<td></td>
<td>All cycles same length</td>
</tr>
<tr>
<td><strong>Instruction time, <code>t_{instruction}</code></strong></td>
<td>= <code>t_{cycle} = 800 ps</code></td>
<td>1000 ps</td>
</tr>
<tr>
<td><strong>Clock rate, <code>f_s</code></strong></td>
<td>1/800 ps = 1.25 GHz</td>
<td>1/200 ps = 5 GHz</td>
</tr>
<tr>
<td><strong>Relative speed</strong></td>
<td>1 x</td>
<td>4 x</td>
</tr>
</tbody>
</table>
RISC-V Pipeline

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

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

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

Resource use in a particular time slot

Resource use of instruction over time
Single-Cycle RISC-V RV32I Datapath

Diagram of a single-cycle RISC-V RV32I datapath. The diagram shows the interaction between various components such as IMEM, ALU, Reg[], DMEM, and PCsel. The diagram includes signals and state transitions for different parts of the datapath, such as PC, ALU, IMEM, and DMEM.
Pipelining RISC-V RV32I Datapath

Instruction Fetch (F)

Instruction Decode/Register Read (D)

ALU Execute (X)

Memory Access (M)

Write Back (W)
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
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
Question:

\[
\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{time}}{\text{cycle}}
\]

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

<table>
<thead>
<tr>
<th></th>
<th>Instructions /program</th>
<th>Cycles/instruction</th>
<th>Time/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>

\[1/\text{throughput}\]
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
  • therefore two different instructions might be accessing the register file on the same cycle!

• Avoid structural hazard by having separate “ports”
  • two independent read ports and one independent write port
  • Reads from one instruction and writes from another can happen simultaneously
Structural Hazard: Memory Access

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

Instruction sequence:

- add t0, t1, t2
- sw t0, 4(t5)
- slt t6, t0, t3
- or t3, t4, t5
- lw t0, 8(t3)
Caches: relatively 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 at least separate instruction/data caches
- Multi-ported register file
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction
  - limited operands per 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?

Instruction sequence:

- $add \ t0, t1, t2$
- $or \ t3, t4, t5$
- $slt \ t6, t4, t3$
- $sw \ t0, 4(t3)$
- $lw \ t0, 8(t3)$
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>5</td>
<td>5/9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Instruction sequence:

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

- Stalls reduce performance
- But stalls might be 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 t0

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

Forwarding: grab operand from pipeline stage, rather than register file
Forwarding (aka Bypassing)

- Access result before it is stored in a register
- Requires extra connections in the datapath
1) Detect Need for Forwarding (example)

- **Add $t0$, $t1$, $t2$**
  - $\text{add } t0, t1, t2$

- **Or $t3$, $t0$, $t5$**
  - $\text{or } t3, t0, t5$

- **Sub $t6$, $t0$, $t3$**
  - $\text{sub } t6, t0, t3$

Comparison of destination of older instructions in pipeline with sources of new instruction in decode stage.

*Must ignore writes to x0!*
Example Forwarding Path

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

Program execution order (in instructions)

- `lw $2, 20($1)`
- `and becomes nop`
- `and $4, $2, $5`
- `or $8, $2, $6`
- `add $9, $4, $2`

Stall

repeat and instruction and forward
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 \( D=A+B; \ E=A+C; \)

Original Order:

- lw t1, 0(t0)
- lw t2, 4(t0)
- add t3, t1, t2
- sw t3, 12(t0)
- lw t4, 8(t0)
- add t5, t1, t4
- sw t5, 16(t0)

Alternative:

- lw t1, 0(t0)
- lw t2, 4(t0)
- lw t4, 8(t0)
- add t3, t1, t2
- sw t3, 12(t0)
- add t5, t1, t4
- sw t5, 16(t0)

Stall! Stall! 13 cycles 11 cycles
Cat Break: From Thomas Thurston
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

```assembly
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: ...

.....

Taken branch

Guess next PC!

Check guess correct
Implementing Branch Prediction...

- This is a CS152 topic, but some ideas:
  - Branch prediction is critical for performance on deeper pipelines/superscalar as the "Misprediction penalty" is vastly higher

- Keep a branch prediction buffer/cache: Small memory addressed by the lowest bits of PC
  - During instruction decode, if branch: Look up whether branch was taken last time?
    - If yes, compute $\text{PC} + \text{offset}$ and fetch that (or store actual branch target address from last time)
    - If no, stick with $\text{PC} + 4$
  - If branch hasn't been seen before
    - Assume forward branches are not taken, backward branches are taken
  - Update state on predictor with results of branch when it is finally calculated
More on the Branch Predictor...

- **What to do on the first time you see a branch?**
  - RISC-V: Assume forward is not-taken, backwards will be taken
    - Forward == IF statement, backwards == loop
  - Compiler can then use this as a hint when structuring code

- **Jumps are never mispredicted.... But JALR can be!**
  - Maintain a similar cache for JALR instructions
  - One for **JALR RA RX**
    - Used to predict object-oriented function calls
  - One for **JALR X0 RA** as a stack
    - Used to handle function returns
Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors
Increasing Single Processor Core 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)
     • Depth limited by max clock rate and power dissipation

3. Multi-issue “superscalar” processor
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: EG, memory/cache misses

- CS152 discusses these techniques!
Out Of Order Superscalar Processor

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Functional units

Integer

Integer

Floating point

Load-store

Commit unit

In-order issue

Out-of-order execute

In-order commit

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

CPI = 1

CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.
And That Is A Beast...

- 6 separate functional units
  - 3x ALU
  - 3 for memory operations
- 20-24 stage pipeline
- Aggressive branch prediction and other optimizations
  - Massive out-of-order capability: Can reorder up to 128 micro-operation instructions!
- And yet it still barely averages a 1 on CPI!
And then Multicore...

• Don't just have one super-scaler processor core, have multiple ones!

• Past approach: Take your best single-core and replicate it...
  • Problem: Those superscalar beasts consume a huge amount of energy

• New approach: Big/Little
  • Some of your processor cores are the single-core beasts...
  • Some are a more energy-optimized design
    • Far less energy/instruction, but fewer instructions/second
  • When absolute performance isn't needed, run tasks on the energy-efficient cores

• Started in cellular phones but now desktop & laptop processors (Intel "Alder Lake", Apple M1) adopt this approach
Pipelining and ISA Design

- **RISC-V ISA designed for pipelining**
  - All instructions are 32-bits in the RV-32 ISA
    - Easy to fetch and decode in one cycle
      - Variant additions add 16b and 64b instructions, but can tell by looking at just the first bits what type it is
    - Versus x86: 1- to 15-byte instructions
      - Requires additional pipeline stages for decoding instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle
And Apple M1

• The following is informed speculation...
  • But the M1 seems to have delivered an >25% performance boost over comparable x86s

• The M1 is ARM rather than Intel x86
  • Arm is best described as "Mostly-RISC"
  • 32 general purpose registers, a load/store architecture, and an easy instruction decoding
    • Probably reduces branch mispredict penalty by a couple of clock cycles
  • But a fair bit of ISA bloat
    • Patterson & Hennessy ARM edition uses a subset of the ISA

• Where is the performance from?
  • Wide decode: 8 instructions/cycle
    • Latest Intel is only 6
  • Monstrous L1 caches, flatter cache hierarchy
    • L1: 192/128kB vs Intel’s 32/48kB ("Alder Lake")
  • Huge reordering: 630 deep out-of-order, 100+ memory loads/stores in process
  • Oh, and 32 general purpose registers to make the compiler happy!
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
- Superscalar processors use multiple execution units for additional instruction level parallelism
  - Performance benefit highly code dependent