Pipelining RISC-V
Administrivia

- Project 2B is due this Today (3/4)!
- We will be releasing exam study resources this week for the midterm.
- We will also release information about the exam later this week.
- Project 1 Clobber Information Released
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 per task
Levels of Representation/Interpretation

- **High Level Language Program (e.g., C)**
- **Assembly Language Program (e.g., RISC-V)**
- **Machine Language Program (RISC-V)**
- **Compiler**
- **Assembler**
- **Machine Interpretation**
- **Hardware Architecture Description (e.g., block diagrams)**
- **Architecture Implementation**
- **Logic Circuit Description (Circuit Schematic Diagrams)**

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Anything can be represented as a number, i.e., data or instructions.

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Pipelining Overview (Review)

- 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
## Pipelining with RISC-V

<table>
<thead>
<tr>
<th>Phase</th>
<th>Pictogram</th>
<th>$t_{step}$ Serial</th>
<th>$t_{cycle}$ Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Reg Rea</td>
<td></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>ALU</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>200 ps</td>
<td>200 ps</td>
</tr>
<tr>
<td>Register Write</td>
<td></td>
<td>100 ps</td>
<td>200 ps</td>
</tr>
</tbody>
</table>

$t_{instruction}$

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

$t_{cycle}$

800 ps

1000 ps
Pipelining with RISC-V

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

- **Single Cycle**
  - Timing: \( t_{step} = 100 \ldots 200 \text{ ps} \)
  - Instruction time, \( t_{instruction} = t_{cycle} = 800 \text{ ps} \)
- **Pipelining**
  - Timing: \( t_{cycle} = 200 \text{ ps} \)
  - Instruction time: 1000 ps
  - CPI (Cycles Per Instruction): \(~1 \text{ (ideal)}, >1 \text{ (actual)}\)
  - Clock rate, \( f_s \): \(1/800 \text{ ps} = 1.25 \text{ GHz} \) vs \(1/200 \text{ ps} = 5 \text{ GHz} \)
  - Relative speed: \(1 \times, 4 \times\)
Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?

add t0, t1, t2
or t3, t4, t5
sll t6, t0, t3
sw t0, 4(t3)
lw t0, 8(t3)
addi t2, t2, 1
RISC-V Pipeline

Instruction sequence:

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

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

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

Resource use in a particular time slot

Resource use of instruction over time
Single Cycle Datapath

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back

PC → instruction memory → registers → ALU → Data memory

rd, rs, rt, imm
Pipeline registers

- Need registers between stages
- To hold information produced in previous cycle
More Detailed Pipeline:
(Note, slightly different ISA)
IF for Load, Store, …
ID for Load, Store, …
EX for Load
MEM for Load
WB for Load – Oops!

Wrong register number!
Corrected Datapath for Load
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
Pipelined Control

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

- A hazard is a situation that prevents starting the next instruction in the next clock cycle

- **Structural** hazard
  - A required resource is busy (e.g. needed in multiple stages)

- **Data** hazard
  - Data dependency between instructions
  - Need to wait for previous instruction to complete its data read/write

- **Control** hazard
  - Flow of execution depends on previous instruction
Structural Hazard

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

• **Solution 1:** Instructions take it in 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:

- add t0, t1, t2
- or t3, t4, t5
- slt t6, t0, t3
- sw t0, 4(t3)
- lw t0, 8(t3)
Instruction and Data Caches

Processor
- Control
- Datapath
  - PC
  - Registers
  - Arithmetic & Logic Unit (ALU)

Memory (DRAM)
- Program
- Data

Instruction Cache
Data Cache
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
• RISC ISAs (including RISC-V) designed to avoid structural hazards
  • e.g. at most one memory access/instruction
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, t0, 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

Instruction sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0, $t1, $t2</td>
<td></td>
</tr>
<tr>
<td>or $t3, $t4, $t5</td>
<td></td>
</tr>
<tr>
<td>slt $t6, $t0, $t3</td>
<td></td>
</tr>
<tr>
<td>sw $t0, 4($t3)</td>
<td></td>
</tr>
<tr>
<td>lw $t0, 8($t3)</td>
<td></td>
</tr>
</tbody>
</table>

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

Value of \( s0 \)

|   | 5 | 5 | 5 | 5 | 5/9 | 9 | 9 | 9 | 9 | 9 |

- 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:
  - effectively NOP: affected pipeline stages do “nothing”
Stalls and Performance

• Stalls reduce performance
  • But stalls may be required to get correct results

• Compiler can rearrange code or insert NOPs (writes to register x0) 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
Detect Need for Forwarding
(example)

add \( t_0, t_1, t_2 \)

or \( t_3, t_0, t_5 \)

sub \( t_6, t_0, t_3 \)

Compare destination of older instructions in pipeline with sources of new instruction in decode stage. Must ignore writes to \( x_0 \)!
Forwarding Path for RA to ALU
Actual Forwarding Path Location…

- We forward with two muxes just after RS1x and RS2x pipeline registers
- The output of the read registers
- Select either the register, the output of the ALU\textsubscript{m} register, or the output of the MEM/WB register
Agenda

• Hazards
  • Structural
  • Data
    • R-type instructions
  • Load
• Control
• Superscalar processors
Load Data Hazard

Program execution order (in instructions)

lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7

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

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

repeat and instruction and forward

Stall
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 \texttt{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 instr!

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)

9 cycles

7 cycles
Agenda

• Hazards
• Structural
• Data
  • R-type instructions
  • Load
• Control
• Superscalar processors
Control Hazards

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

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: .....
Implementing Branch Prediction...

- This is a 152 topic, but some ideas
  - Doesn't matter much on a 5 stage pipeline: 2 cycles on a branch even with no prediction
    - And even with a branch predictor we will probably take a cycle to decide during ID, so we'd be 2 or 1 not 2 or 0...
  - But 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
  - If branch: Look up whether you took the branch the last time?
    - If yes, compute $\text{PC} + \text{offset}$ and fetch that
    - 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
Agenda

• 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 “superscalar” processor
Superscalar Processor

- Multiple issue “superscalar”
  - Replicate pipeline stages $\Rightarrow$ 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

In-order issue

Functional units

Integer

Integer

Floating point

Load-store

Out-of-order execute

Commit unit

In-order commit

P&H p. 340
Benchmark: CPI of 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!
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 bytes 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
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