Pipelining and Hazards

Instructor: Steven Ho
Great Idea #4: Parallelism

- **Parallel Requests**
  Assigned to computer
e.g. search “Garcia”

- **Parallel Threads**
  Assigned to core
e.g. lookup, ads

- **Parallel Instructions**
  > 1 instruction @ one time
e.g. 5 pipelined instructions

- **Parallel Data**
  > 1 data item @ one time
e.g. add of 4 pairs of words

- **Hardware descriptions**
  All gates functioning in
parallel at same time

---

**Software**

**Hardware**

Warehouse Scale Computer

Leverage Parallelism & Achieve High Performance

---

Computer

Core … Core

Memory

Input/Output

---

Instruction Unit(s)

A₀+B₀  A₁+B₁  A₂+B₂  A₃+B₃

Functional Unit(s)

---

Cache Memory

---

Logic Gates
Review of Last Lecture

• Implementing controller for your datapath
  – Take decoded signals from instruction and generate control signals

• Pipelining improves performance by exploiting Instruction Level Parallelism
  – 5-stage pipeline for RISC-V: IF, ID, EX, MEM, WB
  – Executes multiple instructions in parallel
  – Each instruction has the same latency
  – What can go wrong???
Agenda

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

Add $t_0, t_1, t_2$

Or $t_3, t_4, t_5$

SLL $t_6, t_0, t_3$

### Timing

<table>
<thead>
<tr>
<th>Single Cycle</th>
<th>Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{step} = 100 \ldots 200$ ps</td>
<td>$t_{cycle} = 200$ ps</td>
</tr>
<tr>
<td>Register access only 100 ps</td>
<td>All cycles same length</td>
</tr>
<tr>
<td>$t_{instruction} = t_{cycle} = 800$ ps</td>
<td>1000 ps</td>
</tr>
<tr>
<td>1/800 ps = 1.25 GHz</td>
<td>1/200 ps = 5 GHz</td>
</tr>
</tbody>
</table>

**Relative speed**

- Single Cycle: 1 x
- Pipelining: 4 x
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

Resource use of instruction over time:

- $t_{\text{instruction}} = 1000 \text{ ps}$
- $t_{\text{cycle}} = 200 \text{ ps}$

Resource use in a particular time slot:
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)
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
Graphical Pipeline Representation

- RegFile: right half is read, left half is write

Time (clock cycles)
Question: Which of the following signals (buses or control signals) for RISC-V does NOT need to be passed into the EX pipeline stage for a beq instruction?

(A) BrUn
(B) MemWr
(C) RegWr
(D) WBSel

beq t0 t1 Label
Agenda

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

Hazards Ahead!
Pipelining Hazards

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

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

2) Data hazard
   – Data dependency between instructions
   – Need to wait for previous instruction to complete its data write

3) Control hazard
   – Flow of execution depends on previous instruction
Agenda

- RISC-V Pipeline
- 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 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
1. Structural Hazards

- Conflict for use of a resource

Can we read and write to registers simultaneously?
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
Regfile Structural Hazards

• Two alternate solutions:
  1) Build RegFile with independent read and write ports (what you will do in the project; good for single-stage)
  2) Double Pumping: split RegFile access in two! Prepare to write during 1\textsuperscript{st} half, write on \textit{falling} edge, read during 2\textsuperscript{nd} half of each clock cycle
    • Will save us a cycle later...
    • Possible because RegFile access is \textit{VERY} fast (takes less than half the time of ALU stage)

• Conclusion: Read and Write to registers during same clock cycle is okay
Memory Structural Hazards

• Conflict for use of a resource

Time (clock cycles)

Instr 1
Load
Instr 2
Instr 3
Instr 4

Trying to read (and maybe write) same memory twice in same clock cycle
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
Administrivia

• Proj2-2 due 7/13, HW3/4 due 7/16
  – 2-2 autograder being run
• Guerilla Session Tonight! 4-6pm
• HW0-2 grades should now be accurate on glookup
• Project 3 released tomorrow night!
• Supplementary review sessions starting
  – First one this Sat. (7/14) 12-2p, Cory 540AB
Agenda

• RISC-V Pipeline
• Hazards
  – Structural
  – Data
    • R-type instructions
    • Load
  – Control
• Superscalar processors
2. Data Hazards (1/2)

- Consider the following sequence of instructions:

  add  t0, t1, t2
  sub  t4, t0, t3
  and  t5, t0, t6
  or   t7, t0, t8
  xor  t9, t0, t10

- Stored during WB
- Read during ID
2. Data Hazards (2/2)

- Data-flow *backward* in time are hazards

```
add t0, t1, t2
sub t4, t0, t3
and t5, t0, t6
or  t7, t0, t8
xor t9, t0, t10
```

Time (clock cycles)
Solution 1: Stalling

Problem: Instruction depends on result from previous instruction

- \texttt{\textbf{add} t0, t1, t2}
- \texttt{\textbf{sub} t4, t0, t3}

Effectively NOP: affected pipeline stages do nothing.
Stalls and Performance

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

• Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it’s not stored in RegFile yet

**Example Instructions:**
- `add t0, t1, t2`
- `sub t4, t0, t3`
- `and t5, t0, t6`
- `or t7, t0, t8`
- `xor t9, t0, t10`

**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

Program execution order (in instructions)

- add $s0, $t0, $t1
- sub $t2, $s0, $t3
Detect Need for Forwarding

(example)

Compare destination of older instructions in pipeline with sources of new instruction in decode stage. Must ignore writes to x0!

add t0, t1, t2

or t3, t0, t5

sub t6, t0, t3
Forwarding Path

Forwarding Control Logic
Agenda

• RISC-V Pipeline
• Hazards
  – Structural
  – Data
    • R-type instructions
    • Load
  – Control
• Superscalar processors
Data Hazard: Loads (1/4)

• **Recall:** Dataflow backwards in time are hazards

\[
\text{lw } t_0, 0(t_1) \quad \text{sub } t_3, t_0, t_2
\]

• Can’t solve all cases with forwarding
  – Must *stall* instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/4)

• **Hardware** stalls pipeline
  – Called “hardware interlock”

```plaintext
lw t0, 0(t1)
sub t3, t0, t2
and t5, t0, t4
or t7, t0, t6
```

This is what happens in hardware in a “hardware interlock”

Must stall entire pipeline
Data Hazard: Loads (3/4)

• Stall is equivalent to \textit{nop}

\texttt{lw t0, 0(t1)}

\texttt{nop}

\texttt{sub t3, t0, t2}

and \texttt{t5, t0, t4}

or \texttt{t7, t0, t6}
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

\textbf{Idea:} Let the compiler/assembler put an unrelated instruction in that slot $\rightarrow$ no stall!
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:
- \( lw \ t1, 0(t0) \)
- \( lw \ t2, 4(t0) \)
- \( lw \ t4, 8(t0) \)
- \( add t3, t1, t2 \)
- \( sw t3, 12(t0) \)
- \( add t5, t1, t4 \)
- \( sw t5, 16(t0) \)

13 cycles

11 cycles

Stall!
Break!
Agenda

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

• Branch (\texttt{beq}, \texttt{bne}) 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

• \textbf{Simple Solution:} Stall on every branch until we have the new PC value
  – How long must we stall?
Branch Stall

- When is comparison result available?

```
Instr 1
beq
Instr 2
Instr 3
Instr 4
```

Time (clock cycles)

TWO bubbles required per branch!
3. Control Hazard: Branching

- **Option #1:** Moving *branch comparator* to ID stage
  - As soon as instruction is decoded, immediately make a decision and set the new value of the PC
  - **Benefit:** Branch decision made in 2\textsuperscript{nd} stage, so only one *nop* is needed instead of two
  - **Side Note:** Have to compute new PC value in ID instead of EX
    - Adds extra hardware and reduces redundancy
    - Branches are idle in EX, MEM, and WB
Improved Branch Stall

- When is comparison result available?

---

![Diagram showing improved branch stall with comparison result availability]

Only one bubble required now
Data Hazard: Branches!

- **Recall:** Dataflow backwards in time are hazards

\[
\text{add } t_0, t_0, t_1
\]

\[
\text{beq } x_0, t_0, \text{foo}
\]

- Now that \( t_0 \) is needed earlier (ID instead of EX), we can’t forward it to the beq’s ID stage
  - Must *stall* after add, then forward (more hardware)
Observations

• **Takeaway**: Moving branch comparator to ID stage would add extra hardware, reduce redundancy, and introduce new problems

• Can we work with the nature of branches?
  - 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
3. Control Hazard: Branching

• **RISC-V Solution:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary
  – Must cancel (*flush*) all instructions in pipeline that depended on guess that was wrong
  – How many instructions do we end up flushing?
Kill Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx
Branch Prediction

beq t0, t1, label

label:  

....

.

Taken branch

Guess next PC!

Check guess correct
Dynamic Branch Prediction

• Branch penalty is more significant in deeper pipelines

• Use *dynamic branch prediction*
  – Have branch prediction buffer (a.k.a. branch history table) that stores outcomes (taken/not taken) indexed by recent branch instruction addresses
  – To execute a branch
    • Check table and predict the same outcome for next fetch
    • If wrong, flush pipeline and flip prediction
1-Bit Predictor: Shortcoming

• Examine the code below, assuming both loops will be executed multiple times:

```
outer: ...
    ...
inner: ...
    ...
    beq ..., ..., inner
    ...
    beq ..., ..., outer
```

• Inner loop branches are predicted wrong twice!
  – Predict as **taken** on last iteration of inner loop
  – Then predict as **not taken** on first iteration of inner loop next time around
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
Question: For each code sequence below, choose one of the statements below:

1:
- lw $t0, 0($t0)
- add $t1, $t0, $t0

2:
- add $t1, $t0, $t0
- addi $t2, $t0, 5
- addi $t4, $t1, 5

3:
- addi $t1, $t0, 1
- addi $t2, $t0, 2
- addi $t3, $t0, 2
- addi $t3, $t0, 4
- addi $t5, $t1, 5

A  No stalls as is
B  No stalls with forwarding
C  Must stall
Code Sequence 1

Time (clock cycles)

Instr Order

Iw
add
instr
instr
instr

Must stall
Code Sequence 2

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time 1</th>
<th>Time 2</th>
<th>Time 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Forwarding:
- No forwarding
- No stalls with forwarding

7/13/2016
Code Sequence 3

Time (clock cycles)

Instruction Order:
- addi
- addi
- addi
- addi
- addi

ALU

I$  Reg  ALU  D$  Reg
I$  Reg  ALU  D$  Reg
I$  Reg  ALU  D$  Reg
I$  Reg  ALU  D$  Reg
I$  Reg  ALU  D$  Reg

No stalls as is
Agenda

• RISC-V Pipeline
• 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

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 Intel Core i7

P&H p. 350
Summary

• Hazards reduce effectiveness of pipelining
  – Cause stalls/bubbles
• Structural Hazards
  – Conflict in use of a datapath component
• Data Hazards
  – Need to wait for result of a previous instruction
• Control Hazards
  – Address of next instruction uncertain/unknown
• 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 3rd stage, access memory in 4th 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!