CS 61C: Great Ideas in Computer Architecture

Pipelining Hazards

Instructor: Justin Hsia
Great Idea #4: Parallelism

Software
• Parallel Requests
  Assigned to computer
e.g. search “Katz”
• 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

Hardware
• Warehouse Scale Computer
  Leverage Parallelism & Achieve High Performance

Hardware descriptions
All gates functioning in parallel at same time
Review of Last Lecture

• Implementing controller for your datapath
  – Take decoded signals from instruction and generate control signals
  – Use “AND” and “OR” Logic scheme

• Pipelining improves performance by exploiting Instruction Level Parallelism
  – 5-stage pipeline for MIPS: IF, ID, EX, MEM, WB
  – Executes multiple instructions in parallel
  – Each instruction has the same latency
  – What can go wrong???
Question: Are the following statement TRUE or FALSE?

1) Adding a new instruction will NOT require changing any of your existing control logic.

2) Adding a new control signal will NOT require changing any of your existing control logic.
Review: Pipelined Datapath
Graphical Pipeline Representation

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

Time (clock cycles)

Instr Order

Load
Add
Store
Sub
Or
Question: Which of the following signals (buses or control signals) for MIPS-lite does NOT need to be passed into the EX pipeline stage?

- [ ] PC + 4
- [ ] MemWr
- [ ] RegWr
- [x] imm16
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 read/write

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

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
1. Structural Hazards

• Conflict for use of a resource
• MIPS pipeline with a single memory?
  – Load/Store requires memory access for data
  – Instruction fetch would have to *stall* for that cycle
    • Causes a pipeline “*bubble*”
• Hence, pipelined datapaths require separate instruction/data memories
  – Separate L1 I$ and L1 D$ take care of this
Structural Hazard #1: Single Memory

- Trying to read the same memory twice in the same clock cycle.
Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) Split RegFile access in two: Write during 1\textsuperscript{st} half and Read during 2\textsuperscript{nd} half of each clock cycle
     - Possible because RegFile access is VERY fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports

- **Conclusion:** Read and Write to registers during same clock cycle is okay
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
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
2. Data Hazards (2/2)

- Data-flow *backward* in time are hazards

*Time (clock cycles)*

- add $t0,$t1,$t2
- sub $t4,$t0,$t3
- and $t5,$t0,$t6
- or $t7,$t0,$t8
- xor $t9,$t0,$t10
Data Hazard Solution: Forwarding

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

\[
\text{add } \$t0,\$t1,\$t2 \\
\text{sub } \$t4,\$t0,\$t3 \\
\text{and } \$t5,\$t0,\$t6 \\
\text{or } \$t7,\$t0,\$t8 \\
\text{xor } \$t9,\$t0,\$t10
\]
Datapath for Forwarding (1/2)

• What changes need to be made here?
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
Administrivia

• HW 4 due tonight
• Project 2 Part 2 due Sunday
• Project 3 will be released Thursday or Friday
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
Data Hazard: Loads (1/4)

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

```
lw $t0,0($t1)
sub $t3,$t0,$t2
```

- 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”

\[
\text{l}w \; \$t0, \; 0(\$t1) \quad \text{sub} \; \$t3, \$t0, \$t2 \quad \text{and} \; \$t5, \$t0, \$t4 \\
\text{or} \; \$t7, \$t0, \$t6
\]

Schematically, this is what we want, but in reality stalls done “horizontally”

How to stall just *part* of pipeline?
Data Hazard: Loads (3/4)

- Stall is equivalent to \texttt{nop}

\texttt{lw \$t0, 0(\$t1)}

\texttt{nop}

\texttt{sub \$t3,\$t0,\$t2}

and \texttt{\$t5,\$t0,\$t4}

or \texttt{\$t7,\$t0,\$t6}
Data Hazard: Loads (4/4)

• Slot after a load is called a *load delay slot*
  – If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  – Letting the hardware stall the instruction in the delay slot is equivalent to putting a *nop* in the slot (except the latter uses more code space)

• **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!
Code Scheduling to Avoid Stalls

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

• MIPS code for \( A = B + E; \quad C = B + F; \)

```plaintext
# Method 1:
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)
```

13 cycles

```
# Method 2:
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)
```

11 cycles
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
3. Control Hazards

• **Branch** \( (\text{beq}, \text{bne}) \) **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

• **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?

Time (clock cycles)

Instr Order
Instr 1
Instr 2
Instr 3
Instr 4

Instr
beq

I$
Reg
DS
Reg

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

• **Option #1:** Insert *special branch comparator* in 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 \texttt{nop} is needed instead of two
  – **Side Note:** This means that branches are idle in EX, MEM, and WB
Improved Branch Stall

• When is comparison result available?

Time (clock cycles)

Instr Order
Instr 1
Instr 2
Instr 3
Instr 4

Only one bubble required now
Datapath for ID Branch Comparator

• What changes need to be made here?
Datapath for ID Branch Comparator

• Handled by hazard detection unit
3. Control Hazard: Branching

• **Option #2: 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?

• Achieve simplest hardware if we predict that all branches are NOT taken
3. Control Hazard: Branching

• **Option #3**: *Branch delay slot*
  
  – Whether or not we take the branch, *always* execute the instruction immediately following the branch
  
  – **Worst-Case**: Put a `nop` in the branch-delay slot
  
  – **Better Case**: Move an instruction from before the branch into the branch-delay slot
    
    • Must not affect the logic of program
3. Control Hazard: Branching

• MIPS uses this *delayed branch* concept
  – Re-ordering instructions is a common way to speed up programs
  – Compiler finds an instruction to put in the branch delay slot \( \approx 50\% \) of the time

• Jumps also have a delay slot
  – Why is one needed?
Delayed Branch Example

Non-delayed Branch:
- \texttt{or $8, $9, $10}\n- \texttt{add $1, $2, $3}\n- \texttt{sub $4, $5, $6}\n- \texttt{beq $1, $4, Exit}\n- \texttt{xor $10, $1, $11}\n
Delayed Branch:
- \texttt{add $1, $2, $3}\n- \texttt{sub $4, $5, $6}\n- \texttt{beq $1, $4, Exit}\n- \texttt{or $8, $9, $10}\n- \texttt{xor $10, $1, $11}\n
Why not any of the other instructions?
Delayed Jump in MIPS

- MIPS Green Sheet for jal:
  \[ R[31] = PC + 8 \]; \( PC = \text{JumpAddr} \)
  - \( PC + 8 \) because of *jump delay slot*!
  - Instruction at \( PC + 4 \) always gets executed before jal jumps to label, so return to \( PC + 8 \)
Get To Know Your Staff

• Category: Movies
Agenda

• Structural Hazards
• Data Hazards
  – Forwarding
• Administrivia
• Data Hazards (Continued)
  – Load Delay Slot
• Control Hazards
  – Branch and Jump Delay Slots
  – Branch Prediction
Dynamic Branch Prediction

• Branch penalty is more significant in deeper pipelines
  – Also superscalar pipelines (discussed tomorrow)

• 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
2-Bit Predictor

• Only change prediction after two successive incorrect predictions
Question: Are the following statement TRUE or FALSE?

1) Thanks to pipelining, I have reduced the time it took me to wash my one shirt

2) Longer pipelines are always a win (since less work per stage & a faster clock)
**Question:** For each code sequences below, choose one of the statements below:

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

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

3:
- `addi $t1,$t0,1`
- `add $t1,$t0,$t0`
- `addi $t2,$t0,5`
- `addi $t4,$t1,5`
- `addi $t3,$t0,2`
- `addi $t3,$t0,4`
- `addi $t5,$t1,5`

☐ No stalls as is

☐ No stalls with forwarding

☐ Must stall
Code Sequence 1

Time (clock cycles)

Instr Order

Instr

Instr

Instr

Instr

Must stall
Code Sequence 2

Time (clock cycles)

Forwarding

no forwarding

Forwarding

No stalls with forwarding

Instr

Order

Instr

Instr

add

addi

addi

add

addi
Code Sequence 3

No stalls as is
Summary

• Hazards reduce effectiveness of pipelining
  – Cause stalls/bubbles
• Structural Hazards
  – Conflict in use of datapath component
• Data Hazards
  – Need to wait for result of a previous instruction
• Control Hazards
  – Address of next instruction uncertain/unknown
  – Branch and jump delay slots