CS352H: Computer Systems Architecture

Topic 9: MIPS Pipeline - Hazards

October 1, 2009
Data Hazards in ALU Instructions

■ Consider this sequence:
  sub $2, $1,$3
  and $12,$2,$5
  or $13,$6,$2
  add $14,$2,$2
  sw $15,100($2)

■ We can resolve hazards with forwarding
  ■ How do we detect when to forward?
Dependencies & Forwarding

Program execution order (in instructions)

1. sub $2, $1, $3
2. and $12, $2, $5
3. or $13, $6, $2
4. add $14, $2, $2
5. sw $15, 100($2)

Time (in clock cycles)

<table>
<thead>
<tr>
<th>Value of register $2:</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register

- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt

- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0
Forwarding Paths

b. With forwarding
Forwarding Conditions

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Double Data Hazard

- Consider the sequence:
  
  add $1,$1,$2
  add $1,$1,$3
  add $1,$1,$4

- Both hazards occur
  
  - Want to use the most recent

- Revise MEM hazard condition
  
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
 ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
 ForwardB = 01
Datapath with Forwarding
Load-Use 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

Need to stall for one cycle
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    - (ID/EX/RegisterRt = IF/ID/RegisterRt))
- If detected, stall and insert bubble
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Program execution order (in instructions)

Iw $2, 20($1)

and becomes nop

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

lw $2, 20($1)
and becomes nop
and $4, $2, $5 stalled in ID
or $8, $2, $6 stalled in IF
add $9, $4, $2
Datapath with Hazard Detection
Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Branch Hazards

If branch outcome determined in MEM

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

Example: branch taken

36: sub $10, $4, $8
40: beq $1, $3, 7
44: and $12, $2, $5
48: or $13, $2, $6
52: add $14, $4, $2
56: slt $15, $6, $7
...
72: lw $4, 50($7)
Example: Branch Taken

and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $8
before<1>
before<2>

Clock 3

IF.ID

44

Control

Hazard
detection
unit

ID.EX

M

EX

M

EX/MEM

MEM/WB

Instruction memory

Registers

ALU

Data memory

Forwarding unit

University of Texas at Austin    CS352H    Computer Systems Architecture    Fall 2009    Don Fussell    20
Example: Branch Taken

```
Iw $4, 50($7)  

Bubble (nop)  

beq $1, $3, 7  

sub $10, ...  

before<1>  
```

Clock 4
Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

\begin{verbatim}
add $1, $2, $3
add $4, $5, $6
...
beq $1, $4, target
\end{verbatim}

Can resolve using forwarding
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
- Need 1 stall cycle

lw $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

lw $1, addr
beq stalled
beq stalled
beq $1, $0, target
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction
Inner loop branches mispredicted twice!

outer: …
  ...
inner: …
  ...
  beq ..., ..., inner
  ...
  beq ..., ..., outer

Mispredict as taken on last iteration of inner loop
Then mispredict as not taken on first iteration of inner loop next time around
2-Bit Predictor

- Only change prediction on two successive mispredictions
Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately
Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard
Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We’ll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180
An Alternate Mechanism

- **Vectored Interrupts**
  - Handler address determined by the cause

- **Example:**
  - Undefined opcode: \( \text{C000 0000} \)
  - Overflow: \( \text{C000 0020} \)
  - …: \( \text{C000 0040} \)

- **Instructions either**
  - Deal with the interrupt, or
  - Jump to real handler
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  add $1, $2, $1
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware
Pipeline with Exceptions
Exception Properties

- **Restartable exceptions**
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch

- **PC saved in EPC register**
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust
Exception Example

- Exception on `add` in
  
  ```
  40    sub  $11, $2, $4
  44    and  $12, $2, $5
  48    or  $13, $2, $6
  4C    add  $1, $2, $1
  50    slt  $15, $6, $7
  54    lw  $16, 50($7)
  ...
  ```

- Handler
  
  ```
  80000180    sw  $25, 1000($0)
  80000184    sw  $26, 1004($0)
  ...
  ```
Exception Example

lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, ... and $12, ...

Clock 6
Exception Example

sw $25, 1000($0)

bubble (nop)

bubble

bubble

or $13, ...
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions
Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots