Lecture 11: Pipeline Control and Hazards

John Lazzaro
(www.cs.berkeley.edu/~lazzaro)

Dave Patterson
(www.cs.berkeley.edu/~patterson)

[Adapted from Mary Jane Irwin’s slides www.cse.psu.edu/~cg431 ]
Recap last lecture

- All recent processors use pipelining

- Pipelining doesn’t help \textit{latency} of single task, it helps \textit{throughput} or \textit{bandwidth} of entire workload
  - Multiple tasks operating simultaneously using different resources

- Potential speedup = \textit{Number of pipe stages}

- Pipeline rate limited by \textit{slowest} pipeline stage
  - Unbalanced lengths of pipe stages reduces speedup
  - Time to “fill” pipeline and time to “drain” it reduces speedup

- Must detect and resolve hazards
  - Stalling negatively affects throughput

- Today: pipeline control, including hazards
## Control Settings

<table>
<thead>
<tr>
<th></th>
<th>EX Stage</th>
<th>MEM Stage</th>
<th>WB Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Pipeline registers written every clock cycle (like PC)**

Control same for all instructions in IF and ID stages:
fetch instruction, increment PC
MIPS Pipeline Data and Control Paths

Instruction Memory

Read Address

Register File

Read Addr 1
Read Addr 2
Write Addr
Write Data

Sign Extend

16
32

RegWrite

Read Data 1
Read Data 2
Write Data

ALU

ALUSrc

ALUOp

RegDst

0
1

Data Memory

Address
Read Data
Write Data
MemWrite
MemRead

Control

ID/EX

EX/MEM

Branch

Add

Shift left 2

Add

MEM/WB

IF/ID

1
0

Add

4

Control

RegWrite

Ex/ID

RegWrite

MemtoReg

PCSsrc

MemtoReg

MemtoReg
Review: One Way to “Fix” a Data Hazard

add r1, r2, r3

stall

stall

sub r4, r1, r5

and r6, r7, r1

Can fix data hazard by waiting – stall – but affects throughput
Review: Another Way to “Fix” a Data Hazard

Can fix data hazard by forwarding results as soon as they are available to where they are needed.

Instr. Order

add r1, r2, r3

sub r4, r1, r5

and r6, r7, r1

or r8, r1, r1

sw r4, 100(r1)
Data Forwarding (aka Bypassing)

- Any data dependence line that goes backwards in time
  - EX stage generating R-type ALU results or effective address calculation
  - MEM stage generating lw results
- Forward by taking the inputs to the ALU from any pipeline register rather than just ID/EX by
  - adding multiplexors to the inputs of the ALU so can pass Rd back to either (or both) of the EX’s stage Rs and Rt ALU inputs
    - 00: normal input (ID/EX pipeline registers)
    - 10: forward from previous instr (EX/MEM pipeline registers)
    - 01: forward from instr 2 back (MEM/WB pipeline registers)
  - adding the proper control hardware
- With forwarding can run at full speed even in the presence of data dependencies
Data Forwarding Control Conditions (1/4)

1. **EX/MEM hazard:**
   
   if \((EX/MEM.\text{RegisterRd} == ID/EX.\text{RegisterRs}))\)
   
   \[\text{ForwardA} = 10\]
   
   - “RegisterRd” is number of register to be written (RD or RT)
   - “RegisterRs” is number of RS register
   - “RegisterRt” is number of RT register
   - “ForwardA, ForwardB” controls forwarding muxes

   if \((EX/MEM.\text{RegisterRd} = ID/EX.\text{RegisterRt}))\)
   
   \[\text{ForwardB} = 10\]

2. **MEM/WB hazard:**

   if \((MEM/WB.\text{RegisterRd} == ID/EX.\text{RegisterRs}))\)
   
   \[\text{ForwardA} = 01\]

   if \((MEM/WB.\text{RegisterRd} == ID/EX.\text{RegisterRt}))\)
   
   \[\text{ForwardB} = 01\]

*What’s wrong with this hazard control?*  
*(When might it forward when it shouldn’t?)*  
*(Which sequences would reveal this bug?)*
Data Forwarding Control Conditions (2/4)

1. **EX/MEM** hazard:
   
   ```
   if (EX/MEM.RegWrite
       and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
       ForwardA = 10
   if (EX/MEM.RegWrite
       and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
       ForwardB = 10
   ```

   Forwards the result from the previous instr. to either input of the ALU provided it writes.

2. **MEM/WB** hazard:
   
   ```
   if (MEM/WB.RegWrite
       and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
       ForwardA = 01
   if (MEM/WB.RegWrite
       and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
       ForwardB = 01
   ```

   Forwards the result from the second previous instr. to either input of the ALU provided it writes.

   What’s wrong with this hazard control?  
   (When might it forward when it shouldn’t?)  
   (Which sequences would reveal this bug?)
Data Forwarding Control Conditions (3/4)

1. EX/MEM 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

   Forwards the result from the previous instr. to either input of the ALU provided it writes and != R0.

2. MEM/WB 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

   Forwards the result from the second previous instr. to either input of the ALU provided it writes and != R0.

What’s wrong with this hazard control?
Administrivia

- Start Lab 3; Plan due Thursday evening; meet with TA Friday
- Reading Chapter 6, sections 6.5 to 6.9 for this week
- Midterm Tue Oct 12 5:30 - 8:30 in 101 Morgan (you asked for it)
  - Northwest corner of campus, near Arch and Hearst
  - Midterm review Sunday Oct 10, 7 PM, 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Nothing electronic: no calculators, cell phones, pagers, ...
  - Meet at LaVal’s Northside afterwards for Pizza
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded? More recent result!
Corrected Data Forwarding Control Conditions

2. MEM/WB hazard:
   \[
   \text{if (MEM/WB.RegWrite } \\
   \text{ and (MEM/WB.RegisterRd != 0) } \\
   \text{ and (MEM/WB.RegisterRd == ID/EX.RegisterRs) } \\
   \text{ and (EX/MEM.RegisterRd != ID/EX.RegisterRs } \\
   \text{ || ~ EX/MEM.RegWrite)) } \\
   \text{ ForwardA = 01}
   \]

   \[
   \text{if (MEM/WB.RegWrite } \\
   \text{ and (MEM/WB.RegisterRd != 0) } \\
   \text{ and (MEM/WB.RegisterRd == ID/EX.RegisterRt) } \\
   \text{ and (EX/MEM.RegisterRd != ID/EX.RegisterRt } \\
   \text{ || ~ EX/MEM.RegWrite)) } \\
   \text{ ForwardB = 01}
   \]

   Forward if this instruction writes AND its not writing R0 AND this dest reg == source AND in between instr either dest. reg. doesn’t match OR it doesn’t write reg.
Datapath with Forwarding Hardware
Forwarding with Load-use Data Hazards

lw r1, 100(r2)
sub r4, r1, r5
and r6, r1, r5
and $r6, r1, r5$
xor r4, r1, r5
Load-use Hazard Detection Unit

- Need a hazard detection unit in the ID stage that inserts a stall between the load and its use

2. ID Hazard Detection
   if (ID/EX.MemRead
      and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
         or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
   stall the pipeline

- The first line tests to see if the instruction is a load; the next two lines check to see if the destination register of the load in the EX stage matches either source registers of the instruction in the ID stage
- After this 1-cycle stall, the forwarding logic can handle the remaining data hazards
Stall Hardware

- In addition to the hazard detection unit, we have to implement the stall

- Prevent the IF and ID stage instructions from making progress down the pipeline, done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection unit controls the writing of the PC and IF/ID registers

- The instructions in the back half of the pipeline starting with the EX stage must be flushed (execute `noop`)
  - Must deassert the control signals (setting them to 0) in the EX, MEM, and WB control fields of the ID/EX pipeline register.
  - Hazard detection unit controls the multiplexor that chooses between the real control values and 0’s.
  - Assume that 0’s are benign values in datapath: nothing changes
Adding the Hazard Hardware

Instruction Memory
- Read Address

Control
- IF/ID
- Add
- 4

Hazard Unit

Register File
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Addr 2
- Read Data 2
- Write Addr
- Write Data

Sign Extend
- 16
- 32

ALU
- ALU cntrl
- Shift left 2
- Add

Data Memory
- Address
- Read Data
- Write Data

Forward Unit

Branch

IF/ID
- Control

ID/EX
- 0
- 1

EX/MEM
- Add
- 1
- 0

MEM/WB

Forward Unit

PCCsrc
Adding the Hazard Hardware

![Diagram of pipeline control and hazard unit](image)

- **Instruction Memory**
  - Read Address
- **Register File**
  - Read Addr 1
  - Read Addr 2
  - Read Data 1
  - Write Addr
  - Write Data
- **ALU**
  - Shift left 2
  - Add
  - ALU cntrl
- **ID/EX**
  - ID/EX.RegisterRt
  - ID/EX.MemRead
- **Instruction Fetch/ID**
  - Control
- **Control Unit**
  - Branch
- **Data Memory**
  - Address
  - Read Data
  - Write Data
- **Forward Unit**
  - Forward Unit
- **ID/EX**
  - MEM/WB

Pipelining and hazard detection are crucial for managing the flow of instructions and data through the pipeline stages.
Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit to the memory access stage
  - Should avoid stalling on such a load
Where else need to forward?

- What about this code?
  
  ```
  addu $r5, ...
  sw $r5, ...
  ```

  Or

  ```
  addu $r5, ...
  subu $r6, ...
  sw $r5, ...
  ```

- Need forwarding path

- Elaboration on page 412 shows how if have separate mux for immediate vs. forwarded value, can use EX stage forwarding control to pass proper result to EX/MEM register for sw to store
Control Hazards

- When the flow of instruction addresses is not what the pipeline expects; incurred by change of flow instructions
  - Conditional branches \((\text{beq}, \text{bne})\)
  - Unconditional branches \((\text{j})\)

- Possible solutions
  - Stall
  - Move decision point earlier in the pipeline
  - Predict
  - Delay decision (requires compiler support)

- Control hazards occur less frequently than data hazards; there is nothing as effective against control hazards as forwarding is for data hazards
Datapath Branch and Jump Hardware
Jumps Incur One Stall

- Jumps not decoded until ID, so one stall is needed

Fortunately, jumps are very infrequent – only 2% of the SPECint instruction mix
Review: Branches Incur Three Stalls

Can fix branch hazard by waiting – stall – but affects throughput.
Moving Branch Decisions Earlier in Pipe

- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path

- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall cycles to one (like with jumps)
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can’t be done until after RegFile read, so comparing and updating the PC adds a comparator, an and gate, and a 3x1 mux to the ID timing path
  - Need forward hardware in ID stage

- For longer pipelines, decision points are later in the pipeline, incurring more stalls, so we need a better solution
Early Branch Forwarding Issues

- Bypass of source operands from the EX/MEM
  If \((\text{IDcontrol.Branch} \text{ and (EX/MEM.RegisterRd} \neq 0) \text{ and (EX/MEM.RegisterRd} = \text{IF/ID.RegisterRs}))\)
    \[\text{ForwardC} = 1\]
  If \((\text{IDcontrol.Branch} \text{ and (EX/MEM.RegisterRd} \neq 0) \text{ and (EX/MEM.RegisterRd} = \text{IF/ID.RegisterRt}))\)
    \[\text{ForwardD} = 1\]

- MEM/WB “forwarding” is taken care of by the normal RegFile write before read operation
- If the instruction immediately before the branch produces one of the branch compare source operands, then a stall will be required since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation
Supporting ID Stage Branches

IF/ID
- IF/ID
- Add

Control
- Control
- Branch
- PCSrc

Hazard Unit
- Hazard Unit
- 0
- 0

RegFile
- RegFile
- Read Addr 1
- Read Addr 2
- Read Data 1
- Write Addr 1
- Write Data 1
- Write Data

IF.Flush
- IF.Flush
- 0

Shift
- Shift
- Shift left 2

Add
- Add

Data Memory
- Data Memory
- Read Data 2
- Write Data

ALU
- ALU
- cntrl
- cntrl

Forward Unit
- Forward Unit
- Forward Unit
- Forward Unit
- Forward Unit

EX/MEM
- EX/MEM
- 1
- 1

MEM/WB
- MEM/WB
- 1
- 0
Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome.

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch is taken does the pipeline stall.
   - If taken, flush instructions in the pipeline after the branch:
     - in IF, ID, and EX if branch logic in MEM – three stalls
     - in IF if branch logic in ID – one stall
   - ensure that those flushed instructions haven’t changed machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite or RegWrite)
   - restart the pipeline at the branch destination
Flush with Misprediction (Not Taken)

To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a \texttt{noop})

4 beq $1, $2, 2
8 sub $4, $1, $5
Flushes with Misprediction (Not Taken)

- 4 beq $1, $2, 2
- 8 flub $4, $1, $5
- 16 and $6, $1, $7
- 20 or r8, $1, $9

To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a *noop*).
Branch Prediction, con’t

- Resolve branch hazards by **statically** assuming a given outcome and proceeding

2. **Predict taken** – always predict branches will be taken
   - Predict taken always incurs a stall (if branch destination hardware has been moved to the ID stage)

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance

- With more hardware, possible to try to predict branch behavior **dynamically** during program execution

3. **Dynamic branch prediction** – predict branches at run-time using run-time information
Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage, addressed by the lower bits of the PC, contains a bit that tells whether the branch was taken the last time it was executed:
  - Bit may predict incorrectly (may be from a different branch with the same low order PC bits, or may be a wrong prediction for this branch) but the doesn’t affect correctness, just performance
  - If the prediction is wrong, flush the incorrect instructions in pipeline, restart the pipeline with the right instructions, and invert the prediction bit

- The BHT predicts when a branch is taken, but does not tell where its taken to:
  - A branch target buffer (BTB) in the IF stage can cache the branch target address (or !even! the branch target instruction) so that a stall can be avoided
1-bit Prediction Accuracy

1-bit predictor in loop is incorrect twice when not taken

- Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code

1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)

2. As long as branch is taken (looping), prediction is correct

3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time
2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

Loop: 1\textsuperscript{st} loop instr
2\textsuperscript{nd} loop instr
\hspace{1cm} ...
\hspace{1cm} ...
last loop instr
bne $1,$2,Loop
fall out instr
2-bit Predictors

A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

- Right 9 times
- Wrong on loop
- Fall out
- Right on 1st iteration
- Taken
- Not taken
- Predict
- Taken
- Not Taken
- Predict
- Not Taken

Loop:
1st loop instr
2nd loop instr
.
.
last loop instr
bne $1,$2,Loop
fall out instr
Delayed Decision

- First, move the branch decision hardware and target address calculation to the ID pipeline stage.

- A delayed branch always executes the next sequential instruction – the branch takes effect after that next instruction.
  - MIPS software moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay.

- As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot.
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches.
  - Growth in available transistors has made dynamic approaches relatively cheaper.
Scheduling Branch Delay Slots

A. From before branch

\[
\begin{align*}
\text{add} & \quad \text{$1,$2,$3} \\
\text{if} & \quad \text{$2=0$ then} \\
\quad & \quad \text{delay slot} \\
\end{align*}
\]

becomes

\[
\begin{align*}
\text{if} & \quad \text{$2=0$ then} \\
\quad & \quad \text{add $1,$2,$3} \\
\end{align*}
\]

B. From branch target

\[
\begin{align*}
\text{sub} & \quad \text{$4,$5,$6$} \\
\text{if} & \quad \text{$1=0$ then} \\
\quad & \quad \text{delay slot} \\
\quad & \quad \text{add $1,$2,$3$} \\
\quad & \quad \text{if $1=0$ then} \\
\quad & \quad \text{sub $4,$5,$6$} \\
\end{align*}
\]

becomes

\[
\begin{align*}
\text{add} & \quad \text{$1,$2,$3$} \\
\text{if} & \quad \text{$1=0$ then} \\
\quad & \quad \text{sub $4,$5,$6$} \\
\end{align*}
\]

C. From fall through

\[
\begin{align*}
\text{add} & \quad \text{$1,$2,$3$} \\
\text{if} & \quad \text{$1=0$ then} \\
\quad & \quad \text{delay slot} \\
\quad & \quad \text{sub $4,$5,$6$} \\
\end{align*}
\]

becomes

\[
\begin{align*}
\text{add} & \quad \text{$1,$2,$3$} \\
\text{if} & \quad \text{$1=0$ then} \\
\quad & \quad \text{sub $4,$5,$6$} \\
\end{align*}
\]

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the \text{sub} instruction may need to be copied, increasing IC
- In B and C, must be okay to execute \text{sub} when branch fails
Brain storm on bugs (if time permits)

- Depending on branch solution (move to ID, delayed, static prediction, dynamic prediction), where are bugs likely to hide?
  1.
  2.
  ...
- How can you write tests to uncover these likely bugs?
  1.
  2.
  ...
- Once it passes a test, don’t need to run it again?
In Conclusion

- Data dependencies in pipelines often solved by forwarding
- Need to be sure prior instructions will write, destination matches source, and no earlier instruction has priority
- Need forwarding hardware every place where can forward, stall if stage needs to wait for result
  - EX stage, MEM stage for store, ID stage for early branch
- Loads require stall since overlap EX and MEM stages
  - Branches may require stall too
- Control hazards improved via delayed branch/jump in ISA, static prediction for branches, dynamic prediction for branches
  - If predict, hard part of design is recovering from misprediction