Lecture 05: Overcoming Data Hazards

Mohammad Banikazemi

[Slides from Prof. Mary Jane Irwin, PSU Adapted from Computer Organization and Design, Patterson & Hennessy, © 2005, UCB]
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

Time to fill the pipeline
Review: Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - **structural hazards**: attempt to use the same resource by two different instructions at the same time
  - **data hazards**: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
  - **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch instructions

- Can always resolve hazards by **waiting**
  - pipeline control must detect the hazard
  - and take action to resolve hazards
A Single Memory Would Be a Structural Hazard

- Fix with separate instr and data memories (I$ and D$)
How About Register File Access?

Time (clock cycles)

Inst 1

add $1, 

Inst 2

add $2,$1,
How About Register File Access?

Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

Clock edge that controls register writing.

Clock edge that controls loading of pipeline state registers.
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- Read before write data hazard

Instr. Order

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALU</th>
<th>IM</th>
<th>Reg</th>
<th>DM</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $4,$1,$5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $6,$1,$7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $8,$1,$9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor $4,$1,$5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

```
add $1,  
sub $4,$1,$5  
and $6,$1,$7  
or $8,$1,$9
xor $4,$1,$5
```

- Read After Write (RAW) data hazard
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

Instruction Order

lw $1,4($2)
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5

- Load-use (RAW) data hazard
One Way to “Fix” a Data Hazard

Can fix data hazard by waiting – stall – but impacts CPI

```
add $1, IM Reg ALU IM Reg DM Reg
sub $4,$1,$5 IM Reg ALU IM Reg DM Reg
and $6,$1,$7 IM Reg ALU IM Reg DM Reg
```

Instruction Order

stall
stall
Another Way to “Fix” a Data Hazard

Fix data hazards by forwarding results as soon as they are available to where they are needed.

Instr. Order

add $1,
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5
Another Way to “Fix” a Data Hazard

Fix data hazards by forwarding results as soon as they are available to where they are needed.

Instr. Order

add $1, sub $4,$1,$5

and $6,$1,$7

or $8,$1,$9

xor $4,$1,$5
Forwarding with Load-use Data Hazards

lw $1, 4($2)
sub $4, $1, $5
and $6, $1, $7
or $8, $1, $9
xor $4, $1, $5
Forwarding with Load-use Data Hazards

Will still need one stall cycle even with forwarding
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards
One Way to “Fix” a Control Hazard

Fix branch hazard by waiting – stall – but affects CPI
Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - Let it take two cycles (since it doesn’t use the DM stage)

- What if the data memory access is twice as slow as the instruction memory?
  - Make the clock twice as slow or …
  - Let data memory access take two cycles (and keep the same clock rate)
Sample Pipeline Alternatives

- **ARM7**
  - IM
  - Reg
  - EX
  - PC update
  - IM access
  - decode
  - reg access
  - ALU op
  - DM access
  - shift/rotate
  - commit result
  - (write back)

- **StrongARM-1**
  - IM
  - Reg
  - ALU
  - DM
  - Reg

- **XScale**
  - IM1
  - IM2
  - Reg
  - SHFT
  - ALU
  - DM1
  - DM2
  - PC update
  - BTB access
  - start IM access
  - decode
  - reg 1 access
  - ALU op
  - DM write
  - reg write
  - IM access
  - shift/rotate
  - reg 2 access
  - start DM access
  - exception
**Corrected Datapath to Save RegWrite Addr**

- Need to preserve the destination register address in the pipeline state registers

![Datapath Diagram]
Corrected Datapath to Save RegWrite Addr

- Need to preserve the destination register address in the pipeline state registers
MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages
## Control Settings

<table>
<thead>
<tr>
<th></th>
<th>EX Stage</th>
<th>MEM Stage</th>
<th>WB Stage</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
<td>ALU Src</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Data Forwarding (aka Bypassing)

- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle.

- For ALU functional unit: the inputs can come from any pipeline register rather than just from ID/EX by:
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX’s stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes

- Other functional units may need similar forwarding logic (e.g., the DM)

- With forwarding can achieve a CPI of 1 even in the presence of data dependencies
Data Forwarding Control Conditions

1. **EX/MEM hazard:**
   - If \((\text{EX/MEM.RegWrite} \land (\text{EX/MEM.RegisterRd} \neq 0) \land (\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}))\)
     \(\Rightarrow\) ForwardA = 10
   - If \((\text{EX/MEM.RegWrite} \land (\text{EX/MEM.RegisterRd} \neq 0) \land (\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt}))\)
     \(\Rightarrow\) ForwardB = 10

2. **MEM/WB hazard:**
   - If \((\text{MEM/WB.RegWrite} \land (\text{MEM/WB.RegisterRd} \neq 0) \land (\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}))\)
     \(\Rightarrow\) ForwardA = 01
   - If \((\text{MEM/WB.RegWrite} \land (\text{MEM/WB.RegisterRd} \neq 0) \land (\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}))\)
     \(\Rightarrow\) ForwardB = 01

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

Forwards the result from the second previous instr. to either input of the ALU.
Forwarding Illustration

Instr. Order

add $1,

sub $4,$1,$5

and $6,$7,$1

EX/MEM hazard forwarding
MEM/WB hazard forwarding
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?

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

![Diagram showing data flow and potential hazards](image-url)
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?

Instr. Order

add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
Corrected Data Forwarding Control Conditions

2. MEM/WB hazard:
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRd)
and (MEM/WB.RegisterRd = ID/EX.RegisterRd))
    ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Datapath with Forwarding Hardware
Datapath with Forwarding Hardware
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 and a mux to the memory access stage
Forwarding with Load-use Data Hazards

lw $1,4($2)
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5
Forwarding with Load-use Data Hazards

\[ \text{lw} \hspace{2mm} \$1,4($2) \]
\[ \text{add} \hspace{2mm} \$4,\$1,\$5 \]
\[ \text{and} \hspace{2mm} \$6,\$1,\$5 \]
\[ \text{and} \hspace{2mm} \$8,\$1,\$9 \]
\[ \text{xor} \hspace{2mm} \$4,\$1,\$5 \]

IM \rightarrow Reg \rightarrow DM \rightarrow Reg \rightarrow ALU \rightarrow IM \rightarrow Reg \rightarrow DM \rightarrow Reg \rightarrow ALU \rightarrow IM \rightarrow Reg \rightarrow DM \rightarrow Reg
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 now in the EX stage is a \texttt{lw}; the next two lines check to see if the destination register of the \texttt{lw} matches either source register of the instruction in the ID stage (the load-use instruction)

- After this one cycle stall, the forwarding logic can handle the remaining data hazards
Stall Hardware

- Along with the Hazard Unit, we have to implement the stall
- Prevent the instructions in the IF and ID stages from progressing 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 \((\text{PC.write})\) and IF/ID \((\text{IF/ID.write})\) registers
- Insert a “bubble” between the \(\text{l w}\) instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a \(\text{noop}\) in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 \((\text{noop})\). The Hazard Unit controls the mux that chooses between the real control values and the 0’s.
- Let the \(\text{l w}\) instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline
Adding the Hazard Hardware
Adding the Hazard Hardware

Instruction Memory

Read Address

Control

Hazard Unit

IF/ID

Add

4

ID/EX

Read Addr 1

Read Addr 2

Data 1

Data 2

Register File

Write Addr

Write Data

Read

16

32

Sign Extend

ID/EX.

RegisterRt

EX/MEM

Shift left 2

Add

ALU

Shift left 2

ALU ctrl

Data Memory

Address

Read Data

Write Data

Forward Unit

PCSrc

Branch

MEM/WB

Forward Unit

ID/EX.

MemRead

ALU cntrl

ID/EX.

Rt
Next Lecture and Reminders

Next lecture
- Reducing branch costs
  - Reading assignment – PH, Chapter 6.6 and 6.8

Reminders
- Midterm Exam ~30%
  - Wed. March 21, 5:00-6:50pm, WWH 1013
- HW4 due February 21th
- Lab1 due February 28th