Notes and Updates

- **Office hours:** Tues 3:30-5, Wed 9-10:30
  - Come see me for midterm questions
  - I'll be out of town Wed 10:30-Friday night
  - Guest lecture on Thursday: Branch Prediction
    - It's not in the book much PLEASE PAY ATTENTION!

- New homework posted!
- Informal Surveys...
  - I'll remind you later in class
- Questions?
Self Check!

• I’ve got a new design for my hardware. Because I hired sub-par engineers, my pipeline breakdown takes 7 stages:
  - IF, ID, EX1, EX2, EX3, MEM, WB

• How many cycles must my hardware stall for:
  Sub $2, $3, $1
  And $12, $2, $5

Can you draw the all the “stall possibilities”?

IF, ID, EX1, EX2, EX3, MEM, WB

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
<th>CC7</th>
<th>CC8</th>
<th>CC9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $14, $2, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult $14, $2, $2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reducing Data Hazards Through Forwarding

add $2, $3, $4

or $5, $3, $2

We could avoid stalling if we could get the ALU output from "add" to ALU input for the "or"

EX Hazard:

if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd \neq 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd \neq 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
Is That All??

- How many hazards did we have to deal with in this pipeline?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
```

Reducing Data Hazards Through Forwarding

```
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
```
Data Forwarding

- Forwarding (just shown) handles two types of data hazards
  - EX hazard
  - MEM hazard
- We’ve already handled the third type (WB) hazard by using a transparent reg file
  - if the register file is asked to read and write the same register in the same cycle, the reg file allows the write data to be forwarded to the output.

Eliminating Data Hazards via Forwarding

Forwarding can grab the defined register value from anywhere in pipeline even though it has not yet been written to register file.
Does Forwarding eliminate all hazards??

NO! You may need to stall after loads
Try this one...

Show stalls and forwarding for this code
add $3, $2, $1

lw $4, 100($3)

and $6, $4, $3

sub $7, $6, $2

Self Check...

• Can you write the “code” (comparisons of pipeline registers) that detects when a value needs to be forwarded from the MEM phase of a load?

• Can you write the code that detects when a bubble must happen?
Some other specific forwarding situations

- Page 412 of book
- Loads forwarding to stores

- AddI not in the datapath shown in book, extra mux needed for it
  - Could you figure out what extra forwarding would be needed for addi?

Data Hazard Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.
- Data dependences cause data hazards.
- Data hazards can be solved by:
  - software (no-ops)
  - hardware stalling
  - hardware forwarding
- Our processor, and indeed all modern processors, use a combination of forwarding and stalling.
Informal Surveys

- Why do I ask you to do these things?

Branch Hazards
in the Pipelined Processor

\[
\begin{align*}
\text{add } & \$3, \$2, \$1 \\
\text{sub } & \$1, \$2, \$5 \\
\text{beq } & \$1, \$3, \text{ yupEqual} \\
\text{mult } & \$5, \$1, \$2 \\
\text{jump } & \text{goOn} \\
\text{yupEqual: } & \text{mult } \$5, \$1, \$3 \\
\text{goOn: } & \text{or } \$10,\$11,\$12 \\
\end{align*}
\]
Dependences

- **Data dependence**: one instruction is dependent on another instruction to provide its operands.
- **Control dependence** (aka branch dependences): one instruction determines whether another gets executed or not.
- **Control dependences are particularly critical with conditional branches.**

  add $5, $3, $2
  sub $6, $5, $2
  beq $6, $7, somewhere
  and $9, $3, $1

Branch Hazards

- **Data dependences caused data hazards**
  - Sometimes... When?

- **Branch dependences can result in branch hazards (aka control hazards)**
  - Sometimes...
    - What do we need to figure out to know when?
When are branches resolved?

Branch target address is put in PC during:
Correct "next" instruction is fetched during:

Branch Hazards

These instructions shouldn't be executed!

Finally, the right instruction
Dealing With Branch Hazards

- **Software solution**
  - insert no-ops (I don’t think any processors do this)

- **Hardware solutions**
  - stall until you know which direction branch goes
  - guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
    - static branch prediction: base guess on instruction type
    - dynamic branch prediction: base guess on execution history
  - reduce the branch delay

- **Software/hardware solution**
  - delayed branch: Always execute instruction after branch.
    - Compiler puts something useful (or a no-op) there.
Stalling for Branch Hazards

- All branches waste 3 cycles.
  - Seems wasteful, particularly when the branch isn’t taken.
- It’s better to guess whether branch will be taken
  - Easiest guess is “branch isn’t taken”

Assume Branch Not Taken

- works pretty well when you’re right - no wasted cycles
Assume Branch Not Taken

- same performance as stalling when you’re wrong

\[
\text{beq } $4, $0, \text{there and } $12, $2, $5 \text{ or ... add ... there: sub $12, $4, $2}
\]

Some other static BP strategies

1. “backwards” = negative displacement field
   - loops (which branch backwards) are usually executed multiple times.
   - “if-then-else” often takes the “then” (no branch) clause.

2. Compiler makes educated guess
   - sets “predict taken/not taken” bit in instruction
Informal Surveys

- Turn in!

Reducing the Branch Delay
it’s easy to reduce stall to 2-cycles
One-cycle branch misprediction penalty

- Target computation & equality check in ID phase.
- This figure also shows flushing hardware.

"Optimized" Stalling for Branch Hazards: Assume branch is taken
Optimized Stalling for Branch Hazards (with branching in ID stage)

beq $4, $0, there

and $12, $2, $5

or...

Stalling for Branch Hazards with branching in ID stage

beq $4, $0, there

and $12, $2, $5

there: sub
Eliminating the Branch Stall

- There’s no rule that says we have to branch immediately. We could wait an extra instruction before branching.
- The original SPARC and MIPS processors used a to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, whether the branch is taken or not!

Finding an instruction to eliminate the branch stall

- add $3, $2, $1
- sub $1, $2, $5
- beq $1, $3, yupEqual
- mult $5, $1, $2
- jump goOn
- yupEqual: mult $5, $1, $3
- goOn: or $10,$11,$12
- ....
Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
  - Need earlier instruction that doesn’t affect the branch
  - OR:

- If you can’t find anything, you must put a \textit{nop} to insure correctness.

- Worked well for early RISC machines.
  - Doesn’t help recent processors much
  - E.g., MIPS R10000, has a 5-cycle branch penalty, and executes 4 instructions per cycle.
  - The Pentium 4 branch penalty is 19 cycles!

- Meanwhile, delayed branch is a permanent part of the ISA.
Branch Prediction

- Static branch prediction isn’t good enough when mispredicted branches waste 10 or 20 instructions.
- Dynamic branch prediction keeps a brief history of what happened at each branch.

Measuring branch prediction effectiveness

- Misprediction Rate =

- Instructions between mispredictions
  - Intuitive idea of how frequently the pipeline stalls due to branch effects
  - Also affected by frequency of branches overall in code
Dynamic Branch Prediction

Branch history table (BHT)

program counter

This ‘1’ bit means, “the last time the program counter ended with 0100 and a beq instruction was seen, the branch was taken.” Hardware guesses it will be taken again.

for (i=0;i<10;i++) {
  ...
  ...
}

loop top:
  ...
  ...
  add $i, $i, #1
  beq $i, #10, loop top

Mispredict Rate with a BHT

- Assume BHT initialized to zeros - what is predictions are made the FIRST time a branch is encountered?

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Prediction</th>
<th>Actual Branch Direction</th>
</tr>
</thead>
</table>

Misprediction Rate with BHT:
Misprediction Rate with static
Not taken:
Mispredict Rate with a BHT

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Prediction</th>
<th>Actual Branch Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

big loop:
011000  loop top: ...

... 
add $i$, $i$, #1 
beq $i$, #10, loop top 
add $j$, $j$, #1 
beq $j$, #100,000,000 big loop

Misprediction Rate with BHT:

Inaccuracy with BHTs

for (i=0; i<10; i++) {
  ...
  Lots of intervening code
  for (ii = 0; ii<10; i++)
    ...
    Lots of intervening code
}

How can this branch mispredict EVERY ITERATION?
Two-bit predictors are even better
(Branch prediction is a hot research topic)

This state means, "the last two branches at this location were taken."

This one means, "the last two branches at this location were not taken."

Two-bit predictors give better loop prediction

<table>
<thead>
<tr>
<th></th>
<th>Iteration</th>
<th>Prediction</th>
<th>Actual Branch Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is a different 1-bit prediction scheme called a 2-bit saturating counter

Misprediction Rate: