Action Items:
• Assignment 3 is due tomorrow, midnight
• Mid-term: March 8
Announcement

• Programming Assignment 3 is due on tomorrow, midnight
• Mid-term exam: March 8; in class
• Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml
Announcement

• Programming Assignment 3 is due on tomorrow, midnight
• Mid-term exam: March 8; in class
• Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml
Pipeline Example

System Characteristics

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Delay for each instruction: 320 ps
- Can push a new instruction every 320 ps
- Throughput of the system: 3.12 Giga Instructions Per Second (GIPS)
3-Stage Pipelined Version

System Characteristics

- Delay for each instruction: 360 ps (60 ps in loading registers)
- Can push a new instruction every 120 ps
- Throughput of the system: 8.33 Giga Instructions Per Second (GIPS)
Operating a Pipeline

Clock

OP1

OP2

OP3

Time

0 120 240 360 480 640

Clock

Comb. logic

A

R e g

Comb. logic

B

R e g

Comb. logic

C

R e g

100 ps 20 ps 100 ps 20 ps 100 ps 20 ps

100 ps

20 ps

100 ps

20 ps

100 ps

20 ps

Clock

239
Operating a Pipeline

Clock

OP1

A

B

C

OP2

A

B

C

OP3

A

B

C

Time

0 120 240 360 480 640

100 ps 20 ps 100 ps 20 ps 100 ps 20 ps

Comb. logic A

Reg

Comb. logic B

Reg

Comb. logic C

Reg

Clock

100 ps 20 ps 110 ps 20 ps 110 ps 20 ps
Operating a Pipeline

Clock

OP1
A  B  C

OP2
A  B  C

OP3
A  B  C

Time
0  120  240  360  480  640

100 ps  20 ps  100 ps  20 ps  100 ps  20 ps

Comb. logic A
Reg
Comb. logic B
Reg
Comb. logic C
Reg

Clock
Operating a Pipeline

Clock
OP1
A B C
OP2
A B C
OP3
A B C

Time
0 120 240 360 480 640

Comb. logic A
Reg
Comb. logic B
Reg
Comb. logic C
Reg

100 ps 20 ps 100 ps 20 ps 100 ps 20 ps

Clock
Pipeline Stages

Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file
Basic Pipelining Summary

• Pros: Increase throughput by breaking up long combinational paths
• Cons: May increase latency. Need new registers
  • Can’t do better than slowest component (bottleneck)
Interleaving

- Use multiple copies of the slow component
- Interleave between them
  - Cycle by cycle
- Throughput goes up
- N-way interleaving equivalent to N-stage pipeline in performance
Today: Making the Pipeline Really Work

• Control Dependencies
  • Inserting Nops
  • Stalling
  • Delay Slots
  • Branch Prediction

• Data Dependencies
  • Inserting Nops
  • Stalling
  • Out-of-order execution
Control Dependency

• **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.

• Inserting Nops wastes pipeline slots

• Can we do better than that?

• Two strategies:
  • Delay slots
  • Branch Prediction

```assembly
xorg %rax, %rax
jne L1            # Not taken
irmovq $1, %rax   # Fall Through
L1  irmovq $4, %rcx # Target
irmovq $3, %rax   # Target + 1
```
Control Dependency

• **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
• Inserting Nops wastes pipeline slots
• Can we do better than that?
• Two strategies:
  • Delay slots
  • Branch Prediction

```
xorg %rax, %rax
jne L1            # Not taken
irmovq $1, %rax   # Fall Through
L1  irmovq $4, %rcx # Target
irmovq $3, %rax   # Target + 1
```
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

```assembly
xorg %rax, %rax
jne L1            # Not taken
irmovq $1, %rax   # Fall Through
L1  irmovq $4, %rcx  # Target
irmovq $3, %rax   # Target + 1
```
**Control Dependency**

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

```assembly
xorg %rax, %rax          # F D E
jne L1                  # Not taken  F D
  irmovq $1, %rax        # Fall Through
L1  irmovq $4, %rcx     # Target
  irmovq $3, %rax       # Target + 1
```
Control Dependency

• **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.

• Inserting Nops wastes pipeline slots

• Can we do better than that?

• Two strategies:
  • Delay slots
  • Branch Prediction

```assembly
xorg %rax, %rax
jne L1 # Not taken
nop
irmovq $1, %rax # Fall Through
L1 irmovq $4, %rcx # Target
irmovq $3, %rax # Target + 1
```
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

```assembly
1  xorg %rax, %rax
2  jne L1            # Not taken
3    nop
4   irmovq $1, %rax  # Fall Through
L1  irmovq $4, %rcx  # Target
    irmovq $3, %rax  # Target + 1
```
Control Dependency

• **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.

• Inserting Nops wastes pipeline slots

• Can we do better than that?

• Two strategies:
  • Delay slots
  • Branch Prediction

```
xorg %rax, %rax
jne L1               # Not taken
nop                  
nop                  
irmovq $1, %rax     # Fall Through
L1                   
irmovq $4, %rcx     # Target
irmovq $3, %rax     # Target + 1
```
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.

- Inserting Nops wastes pipeline slots

- Can we do better than that?

- Two strategies:
  - Delay slots
  - Branch Prediction

```
  xorg %rax, %rax
  jne L1           # Not taken
  nop
  nop
  irmovq $1, %rax  # Fall Through
L1  irmovq $4, %rcx # Target
  irmovq $3, %rax  # Target + 1
```
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.

- Inserting Nops wastes pipeline slots

- Can we do better than that?

- Two strategies:
  - Delay slots
  - Branch Prediction

```assembly
xor %rax, %rax
jne L1          # Not taken
nop
nop
irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx # Target
irmovq $3, %rax  # Target + 1
```
Delay Slots

```
xorg %rax, %rax
jne L1
nop
nop
irmovq $1, %rax  # Fall Through
L1  irmovq $4, %rcx  # Target
irmovq $3, %rax  # Target + 1
Can we make use of the 2 wasted slots?
```
Delay Slots

Can we make use of the 2 wasted slots?

```assembly
xorg %rax, %rax
jne L1
nop
nop
irmovq $1, %rax   # Fall Through
irmovq $4, %rcx   # Target
irmovq $3, %rax   # Target + 1
```

```c
if (cond) {
    do_A();
} else {
    do_B();
}
do_C();
```
Delay Slots

Can we make use of the 2 wasted slots?

xorg %rax, %rax
jne L1
nop
nop

irmovq $1, %rax  # Fall Through
L1  irmovq $4, %rcx  # Target
irmovq $3, %rax  # Target + 1

Have to make sure do_C doesn't depend on do_A and do_B!!!

if (cond) {
    do_A();
} else {
    do_B();
}
do_C();
### Delay Slots

Can we make use of the 2 wasted slots?

```assembly
xorg  %rax, %rax  

jne L1
nop
nop

irmovq $1, %rax   # Fall Through

L1  irmovq $4, %rcx   # Target
irmovq $3, %rax   # Target + 1
```

Can we make use of the 2 wasted slots?

```c
A less obvious example

```do_C();
if (cond) {
    do_A();
} else {
    do_B();
}
```
Delay Slots

Can we make use of the 2 wasted slots?

A less obvious example

dob();
if (cond) {
    do_A();
} else {
    do_B();
}

\[\begin{array}{ccccccc}
\text{add A, B} & \text{or C, D} & \text{sub E, F} & \text{jle 0x200} & \text{add A, C} \\
\end{array}\]
Delay Slots

Can we make use of the 2 wasted slots?

A less obvious example

```c
xorg %rax, %rax
jne L1
nop
nop

irmaovq $1, %rax  # Fall Through
irmaovq $4, %rcx  # Target
irmaovq $3, %rax  # Target + 1

L1
```

```c
\textbf{do\_C();}
  \textbf{if \ (cond)} \{
    \textbf{do\_A();}
  \} \textbf{else} \{
    \textbf{do\_B();}
\}
```

```c
\begin{align*}
\text{add A, B} & \quad \text{add A, B} \\
\text{or C, D} & \quad \text{sub E, F} \\
\text{sub E, F} & \quad \text{jle 0x200} \\
\text{jle 0x200} & \quad \text{or C, D} \\
\text{add A, C} & \quad \text{add A, C}
\end{align*}
```
Delay Slots

Can we make use of the 2 wasted slots?

A less obvious example

```
  do_C();
  if (cond) {
    do_A();
  } else {
    do_B();
  }
```

Why don’t we move the sub instruction?
Branch Prediction

Static Prediction
- Always Taken
- Always Not-taken

Dynamic Prediction
- Dynamically predict taken/not-taken for each specific jump instruction

If prediction is correct: pipeline moves forward without stalling
If mispredicted: kill mis-executed instructions, start from the correct target
Static Prediction
Static Prediction

Observation: Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.
Static Prediction

Observation: Two uses of jumps

• People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
• People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

cmpq %rsi,%rdi
jle .corner_case
<do_A>
.corner_case:
<do_B>
ret
Static Prediction

Observation: Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

cmpq %rsi,%rdi
jle .corner_case
<do_A>
.corner_case:
<do_B>
ret

Mostly not taken
Static Prediction

Observation: Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```
cmpq  %rsi,%rdi
jle   .corner_case
<do_A>
.corner_case:
<do_B>
ret

<before>

.L1:  <body>
cmpq  B, A
jl    .L1
<after>
```

Mostly not taken
Static Prediction

Observation: Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```assembly
.cmpq    %rsi,%rdi
.jle     .corner_case

.corner_case:
  .do_A

.ret

.L1: <body>
  .cmpq    B, A
  .jl .L1

<before>

Mostly not taken

Mostly taken
```
Static Prediction

Observation: Two uses of jumps

• People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
• People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

Strategy:

• Forward jumps (i.e., if-else): always predict not-taken
• Backward jumps (i.e., loop): always predict taken

```
cmpq    %rsi,%rdi
jle     .corner_case
<do_A>
.corner_case:
<do_B>
ret
```

```
.L1: <body>
cmpq B, A
jl .L1
<after>
```
Static Prediction

Knowing branch prediction strategy helps us write faster code

• Any difference between the following two code snippets?
• What if you know that hardware uses the always non-taken branch prediction?

```c
if (cond) {
    do_A()
} else {
    do_B()
}
```

```c
if (!cond) {
    do_B()
} else {
    do_A()
}
```
Dynamic Prediction

• Simplest idea:
  • If last time taken, predict taken; if last time not-taken, predict not-taken
  • Called 1-bit branch predictor
  • Works nicely for loops
Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```c
for (i=0; i < 5; i++) {...}
```
Dynamic Prediction

• Simplest idea:
  • If last time taken, predict taken; if last time not-taken, predict not-taken
  • Called 1-bit branch predictor
  • Works nicely for loops

  for (i=0; i < 5; i++) {...}

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Outcome</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Actual Outcome</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
</tbody>
</table>
Dynamic Prediction

• Simplest idea:
  • If last time taken, predict taken; if last time not-taken, predict not-taken
  • Called 1-bit branch predictor
  • Works nicely for loops

```plaintext
for (i=0; i <5; i++) {...}
```

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Outcome</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Actual Outcome</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
</tbody>
</table>
Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict.
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind.
Dynamic Prediction

• With 1-bit prediction, we change our mind instantly if mispredict
• Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```c
for (i=0; i <5; i++) {...}
```

<table>
<thead>
<tr>
<th>Predict with 1-bit</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Outcome</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>Predict with 2-bit</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Dynamic Prediction

• With 1-bit prediction, we change our mind instantly if mispredict
• Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```
Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```plaintext
for (i=0; i < 5; i++) {...}
```

<table>
<thead>
<tr>
<th>Predict with 1-bit</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Outcome</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Predict with 2-bit</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
```
Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict.
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind.

```c
for (i=0; i < 5; i++) {...}
```
More Advanced Dynamic Prediction

- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

\[
\text{cond1 branch not-taken means } (x \leq 0) \\
\text{branch taken}
\]

\[
x = 0 \\
\text{if (cond1) } x = 3 \\
\text{if (cond2) } y = 19 \\
\text{if (x } \leq 0) \text{ z = 13}
\]
How to Keep The Predictions?

- Branch Target Buffer (BTB)
  - A separate memory that’s not visible to programmers
  - Keep tracks of the target for each branch instruction

<table>
<thead>
<tr>
<th>Instruction Memory</th>
<th>instruction word</th>
<th>Branch target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cancel instructions when mispredicted
  • Detect branch not-taken in execute stage
  • On following cycle, replace instructions in execute and decode by bubbles
  • No side effects have occurred yet
Today: Making the Pipeline Really Work

- Control Dependencies
  - Inserting Nops
  - Stalling
  - Delay Slots
  - Branch Prediction

- Data Dependencies
  - Inserting Nops
  - Stalling
  - Out-of-order execution
Data Dependencies

1  irmovq $50, %rax
2  addq %rax, %rbx
3  mrmovq 100(%rbx), %rdx
Data Dependencies

1. \texttt{irmovq $50, \%rax}
2. \texttt{addq \%rax, \%rbx}
3. \texttt{mrmovq 100(\%rbx), \%rdx}
Data Dependencies

1. `irmovq $50, %rax`
2. `addq %rax, %rbx`
3. `mrmovq 100(%rbx), %rdx`
Data Dependencies

1. `irmovq $50, %rax`
2. `addq %rax, %rbx`
3. `mrmovq 100(%rbx), %rdx`

- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact
Data Dependencies: No Nop

0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
Data Dependencies: 1 Nop

0x000: irmovq $10, %rdx
0x00a: irmovq $3, %rax
0x014: nop
0x015: addq %rdx, %rax
0x017: halt
Data Dependencies: 2 Nop’s

0x000:  irmovq $10,%rdx
0x00a:  irmovq $3,%rax
0x014:  nop
0x015:  nop
0x016:  addq %rdx,%rax
0x018:  halt

Cycle 6

W

R[%rax] ← 3

D

valA ← R[%rdx]=10
valB ← R[%rax]=0

Error
Data Dependencies: 3 Nop’s

0x000: `irmovq $10,%rdx`
0x00a: `irmovq $3,%rax`
0x014: `nop`
0x015: `nop`
0x016: `nop`
0x017: `addq %rdx,%rax`
0x019: `halt`

Cycle 6

- `W`
- `R[%rax] ← 3`

Cycle 7

- `D`
- `valA ← R[%rdx] = 10`
- `valB ← R[%rax] = 3`
Normal Execution

Fetch | Reg | Decode | Reg | Execute | Reg | Memory | Reg | Write back | Reg
Normal Execution

Inst0

Fetch | Reg | Decode | Reg | Execute | Reg | Memory | Reg | Write back | Reg
Normal Execution

Inst1
- Fetch
- Reg

Inst0
- Decode
- Reg
- Execute
- Reg
- Memory
- Reg
- Write back
- Reg
Normal Execution

Inst2
- Fetch

Inst1
- Decode

Inst0
- Execute
- Memory
- Write back
<table>
<thead>
<tr>
<th>Inst3</th>
<th>Inst2</th>
<th>Inst1</th>
<th>Inst0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Diagram showing the stages of normal execution: Fetch, Decode, Execute, Memory, Write back.
Normal Execution

- Inst4: Fetch
- Inst3: Decode
- Inst2: Execute
- Inst1: Memory
- Inst0: Write back

Reg Reg Reg Reg Reg
Normal Execution

Fetch | Decode | Execute | Memory | Write back
Normal Execution

Fetch → Decode → Execute → Memory → Write back → Reg

Inst4 → Inst3 → Inst2
Normal Execution

Fetch  Reg  Decode  Reg  Execute  Reg  Memory  Reg  Write back  Reg
Normal Execution

- Fetch
- Decode
- Execute
- Memory
- Write back

Inst4
Stalling Illustration

Fetch → Decode → Execute → Memory → Write back → Reg
Stalling Illustration

Inst0

Fetch \rightarrow \text{Reg} \rightarrow \text{Decode} \rightarrow \text{Reg} \rightarrow \text{Execute} \rightarrow \text{Reg} \rightarrow \text{Memory} \rightarrow \text{Reg} \rightarrow \text{Write back} \rightarrow \text{Reg}
Stalling Illustration

Inst1     Inst0

Fetch     Reg     Decode     Reg     Execute     Reg     Memory     Reg     Write back     Reg
Stalling Illustration

Inst2: Fetch
Inst1: Decode
Inst0: Execute
Memory
Write back
Stalling Illustration

Inst3
Fetch

Inst2
Decode

Inst1
Execute

Inst0
Memory

Write back

Reg

Reg

Reg

Reg

Reg
Stalling Illustration

Inst3: Fetch
Inst2: Decode
bubble (nop)
Inst1: Execute
Inst0: Memory
Write back

Reg
Reg
Reg
Reg
Reg
Stalling Illustration

<table>
<thead>
<tr>
<th>Inst3</th>
<th>Inst2</th>
<th>bubble (nop)</th>
<th>bubble (nop)</th>
<th>Inst1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td>Write back</td>
</tr>
</tbody>
</table>
Stalling Illustration

<table>
<thead>
<tr>
<th>Inst4</th>
<th>Inst3</th>
<th>Inst2</th>
<th>bubble (nop)</th>
<th>bubble (nop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td>Write back</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
</tbody>
</table>

- Inst4: Fetch
- Inst3: Decode
- Inst2: Execute
- bubble (nop)
- bubble (nop)
Stalling Illustration

Fetch → Decode → Execute → Memory → Write back → bubble (nop)
Stalling Illustration

Fetch | Reg
---|---
Decode | Reg
Execute | Reg
Memory | Reg
Write back | Reg
Inst2 | Reg
Inst3 | Reg
Inst4 | Reg
Pipeline Register Mode to Support Stall and Bubble
Pipeline Register Mode to Support Stall and Bubble

Normal

Input = y
Output = x

stall = 0
bubble = 0
Pipeline Register Mode to Support Stall and Bubble

Normal

\[ \text{Input} = y \quad \text{Output} = x \]

\[ \text{stall} = 0 \quad \text{bubble} = 0 \]

Rising clock

\[ \text{Output} = y \]
Pipeline Register Mode to Support Stall and Bubble

**Normal**

- Input = y
- Output = x
- Stall = 0
- Bubble = 0

**Stall**

- Input = y
- Output = x
- Stall = 1
- Bubble = 0
Pipeline Register Mode to Support Stall and Bubble

Normal

Input = y
Output = x

stall = 0
bubble = 0

Rising clock

Output = y

Stall

Input = y
Output = x

stall = 1
bubble = 0

Rising clock

Output = x
Pipeline Register Mode to Support Stall and Bubble

Normal

Input = y  Output = x

stall = 0  bubble = 0

Rising clock

Output = y

Stall

Input = y  Output = x

stall = 1  bubble = 0

Rising clock

Output = x

Bubble

Input = y  Output = x

stall = 0  bubble = 1

Output = x

32
Pipeline Register Mode to Support Stall and Bubble

**Normal**

- Input = y
- Output = x
- stall = 0
- bubble = 0

**Stall**

- Input = y
- Output = x
- stall = 1
- bubble = 0

**Bubble**

- Input = y
- Output = x
- stall = 0
- bubble = 1

- Output = x
- Output = y
- Output = nop
Stalling for Data Dependencies

• If instruction follows too closely after one that writes register, slow it down
• Hold instruction in decode
• Think of it as dynamically injecting nops into execute stage
Stalling X3

0x000: irmovq $10,%rdx

0x00a: irmovq $3,%rax
  bubble
  bubble
  bubble

0x014: addq %rdx,%rax

0x016: halt
Detecting Stall Condition

0x000: irmovq $10, %rdx
0x00a: irmovq $3, %rax
0x014: nop
0x015: nop  \textit{bubble}
0x016: addq %rdx, %rax
0x018: halt

Cycle 6

\begin{align*}
W_{\text{dstE}} &= \%rax \\
W_{\text{valE}} &= 3
\end{align*}
Data Forwarding

Naïve Pipeline

- Register isn’t written until completion of write-back stage
- Source operands read from register file in decode stage
- The decode stage can’t start until the write-back stage finishes

Observation

- Value generated in execute or memory stage

Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage
Data Forwarding Example

- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as `valB` for decode stage

```
0x000: irmovq $10, %rdx
0x000a: irmovq $3, %rax
0x014: nop
0x015: nop
0x016: addq %rdx, %rax
0x018: halt
```
Data Forwarding Example #2

0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt

Register %rdx
- Generated by ALU during previous cycle
- Forward from memory as valA

Register %rax
- Value just generated by ALU
- Forward from execute as valB
Forwarding Priority

Multiple Forwarding Choices

- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
Out-of-order Execution

• Compiler could do this, but has limitations
• Generally done in hardware

Long-latency instruction. Forces the pipeline to stall.

\[
\begin{align*}
    r_0 &= r_1 + r_2 \\
    r_3 &= \text{MEM}[r_0] \\
    r_4 &= r_3 + r_6 \\
    r_7 &= r_5 + r_1 \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    r_0 &= r_1 + r_2 \\
    r_3 &= \text{MEM}[r_0] \\
    r_7 &= r_5 + r_1 \\
    \ldots
\end{align*}
\]
Out-of-order Execution

\[ r_0 = r_1 + r_2 \]
\[ r_3 = \text{MEM}[r_0] \]
\[ r_4 = r_3 + r_6 \]
\[ r_6 = r_5 + r_1 \]

...
Out-of-order Execution

\[ r_0 = r_1 + r_2 \]
\[ r_3 = \text{MEM}[r_0] \]
\[ r_4 = r_3 + r_6 \]
\[ r_6 = r_5 + r_1 \]
\[ \ldots \]

Is this correct?

\[ r_0 = r_1 + r_2 \]
\[ r_3 = \text{MEM}[r_0] \]
\[ r_6 = r_5 + r_1 \]
\[ \ldots \]
\[ r_4 = r_3 + r_6 \]
Out-of-order Execution

\[
\begin{align*}
    r_0 &= r_1 + r_2 \\
    r_3 &= \text{MEM}[r_0] \\
    r_4 &= r_3 + r_6 \\
    r_6 &= r_5 + r_1 \\
    \ldots
\end{align*}
\]

Is this correct?

\[
\begin{align*}
    r_0 &= r_1 + r_2 \\
    r_3 &= \text{MEM}[r_0] \\
    r_4 &= r_3 + r_6 \\
    r_6 &= r_5 + r_1 \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    r_0 &= r_1 + r_2 \\
    r_3 &= \text{MEM}[r_0] \\
    r_4 &= r_3 + r_6 \\
    r_4 &= r_5 + r_1 \\
    \ldots
\end{align*}
\]
Out-of-order Execution

\[
\begin{align*}
r_0 &= r_1 + r_2 \\
r_3 &= \text{MEM}[r_0] \\
r_4 &= r_3 + r_6 \\
r_6 &= r_5 + r_1 \\
\end{align*}
\]

Is this correct?

\[
\begin{align*}
r_0 &= r_1 + r_2 \\
r_3 &= \text{MEM}[r_0] \\
r_6 &= r_5 + r_1 \\
r_4 &= r_3 + r_6 \\
\end{align*}
\]
Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

Is this correct?

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r4 = r5 + r1
...
```

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r5 + r1
...
r4 = r3 + r6
```

If you are interested, Google “Tomasolu Algorithm.” It is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.