Processor Architecture: Pipeline dependencies

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

A3 due Thursday, let us know about partners and/or slip days before the due date!

Midterm in one week, counts for 25% of final grade

• Material from today may be included on the exam
• Some review today
• Even more review next class
• 15% partial credit for “I don’t know”, but must erase or cross out anything else on that question
Pipeline Trade-offs

- **Pros:** Decrease the total execution time (Increase the “throughput”).
- **Cons:** Increase the latency of each instruction as new registers are needed between pipeline stages.

![Pipeline Diagram]

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

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

300 ps 20 ps

Combinational logic Reg

Clock

Clock
Throughput

• The rate at which the processor can finish executing an instruction (at the steady state).

Throughput of this 3-stage processor is 1 instruction every 120 ps, or 8.3 Giga (billion) Instructions per Second (GIPS).
Aside: Unbalanced Pipeline

• A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

Cycle time: 120 ps
Delay: 360 ps
Thruput: 8.3 GIPS
Aside: Unbalanced Pipeline

- A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

Cycle time: 120 ps
Delay: 360 ps
Thruput: 8.3 GIPS
Aside: Unbalanced Pipeline

- A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

**Cycle time: 120 ps**

- Delay: 360 ps
- Thrupt: 8.3 GIPS

---

**Cycle time: 170 ps**
Aside: Unbalanced Pipeline

- A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

Rotation 1:

Cycle time: 120 ps
Delay: 360 ps
Thruput: 8.3 GIPS

Rotation 2:

Cycle time: 170 ps
Delay: 510 ps
Aside: Unbalanced Pipeline

- A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

**Cycle time:** 120 ps  
**Delay:** 360 ps  
**Thruput:** 8.3 GIPS

**Cycle time:** 170 ps  
**Delay:** 510 ps  
**Thruput:** 5.9 GIPS
Aside: Unbalanced Pipeline

• A pipeline’s delay is limited by the slowest stage. This limits the cycle time and the throughput.

Cycle time: 170 ps
Delay: 510 ps
Thrupt: 5.9 GIPS
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

![Diagram showing pipeline with stages and copy units]
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

What logic do you need there?

Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

What logic do you need there?

Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.
Aside: Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
Aside: Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
- This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.
Aside: Mitigating Unbalanced Pipeline

• Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
• This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.
• The cycle time is reduced to 70 ps (as opposed to 120 ps) at the cost of extra hardware.
Pipeline Stages

Fetch
- Use PC to read instruction
- **Compute new PC for non-jump instructions**

Decode
- Read program registers

Execute
- Operate ALU
- **Compute new PC for jump instructions**

Memory
- Read or write data memory

Write Back
- Update register file
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn’t know its outcome until after its Execute stage

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

Fetch
- Use PC to read instruction
- **Compute new PC for non-jump instructions**

Decode
- Read program registers
- **Compute new PC for jump instructions**

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file
Better Pipelining

Fetch
- Use PC to read instruction
- **Compute new PC for non-jump instructions**

Decode
- Read program registers
- **Compute new PC for jump instructions**

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file
Saving One Cycle

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

• Jump instruction example below:
  
  • `jne L1` determines whether `irmovq $1, %rax` should be executed
  
  • But `jne` doesn’t know its outcome until after its Execute stage

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

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach

- Hardware mechanisms
  - Stalling (Think of it as hardware automatically inserting nops)
  - Branch Prediction
  - Return Address Stack
Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

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

Also takes a guess of the jump direction
Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?
If prediction is correct: pipeline moves forward without stalling

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

Also takes a guess of the jump direction
Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?
If prediction is correct: pipeline moves forward without stalling
If mispredicted: kill mis-executed instructions, start from the correct target

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

Also takes a guess of the jump direction
Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?
If prediction is correct: pipeline moves forward without stalling
If mispredicted: kill mis-executed instructions, start from the correct target

Static Prediction
- Always Taken
- Always Not-taken

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

Observation (Assumption really): 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 (Assumption really): 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
<do_A>
.corner_case:
<do_B>
ret
```
Static Prediction

Observation (Assumption really): 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.

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

Mostly not taken
Static Prediction

Observation (Assumption really): 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>
    <do_B>
    ret
```

`<before>`

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

`<after>`

Mostly not taken
Static Prediction

Observation (Assumption really): 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_B>
RET

.L1: <body>
CMPQ B, A
JL .L1
<after>
```
Static Prediction

Observation (Assumption really): 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., \texttt{if-else}): always predict not-taken
• Backward jumps (i.e., loop): always predict taken

\begin{verbatim}
cmpq    %rsi,%rdi
jle     .corner_case
<do_A>
.corner_case:    Mostly not taken
<do_B>
ret

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

 Mostly taken
\end{verbatim}
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

  \[
  \text{for } (i=0; i < 5; i++) \{ \ldots \}
  \]
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

\[
\text{for (i=0; i <5; i++) {...}}
\]

<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

```
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++) {...}
```
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 \textit{twice in a row} before changing our mind

\[
\text{for (i=0; i <5; i++) {...}}
\]

<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>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>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>
<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++) {...}
```

<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>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</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>T</td>
<td>N</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 \textit{twice in a row} before changing our mind

\begin{verbatim}
for (i=0; i <5; i++) {...}
\end{verbatim}
More Advanced Dynamic Prediction

• Look for past histories \textit{across instructions}
• Branches are often correlated
  • Direction of one branch determines another

\begin{align*}
\text{cond1 branch not-taken means (} x \leq 0 \text{) } &\quad x = 0 \\
\text{branch taken} &\quad \begin{align*}
\text{if (cond1) } x &= 3 \\
\text{if (cond2) } y &= 19 \\
\text{if } (x \leq 0) &\quad z = 13
\end{align*}
\end{align*}
What Happens If We Mispredict?

Cancel instructions when mispredicted

- Assuming we 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  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
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
A Subtle Data Dependency

• Jump instruction example below:
  • jne L1 determines whether irmovq $1, %rax should be executed
  • But jne doesn’t know its outcome until after its Execute stage.

Why?

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

- Jump instruction example below:
  - jne L1 determines whether irmovq $1, %rax should be executed
  - But jne doesn’t know its outcome until after its Execute stage.

Why?

- There is a data dependency between xorg and jne. The “data” is the status flags.

```
xorg %rax, %rax
jne L1         # Not taken
irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx # Target
irmovq $3, %rax # Target + 1
```
In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.
Data Hazards happen when:

- Result does not feed back around in time for next operation
Data Dependencies in Pipeline Machines

Data Hazards happen when:

- Result does not feed back around in time for next operation
Data Dependencies: No Nop

0x000: irmovq $10, %rdx

0x00a: irmovq $3, %rax

0x014: addq %rdx, %rax

0x016: halt

Remember registers get updated in the Write-back stage
Data Dependencies: No Nop

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

Remember registers get updated in the Write-back stage

addq reads wrong %rdx and %rax
Data Dependencies: 1 Nop

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

addq still reads wrong %rdx and %rax
Data Dependencies: 2 Nop’s

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>irmovq $10,%rdx</td>
<td>1</td>
</tr>
<tr>
<td>0x00a</td>
<td>irmovq $3,%rax</td>
<td>2</td>
</tr>
<tr>
<td>0x014</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>0x015</td>
<td>nop</td>
<td>4</td>
</tr>
<tr>
<td>0x016</td>
<td>addq %rdx,%rax</td>
<td>5</td>
</tr>
<tr>
<td>0x018</td>
<td>halt</td>
<td>6</td>
</tr>
</tbody>
</table>

addq reads the correct %rdx, but %rax still wrong
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`

`addq` reads the correct `%rdx` and `%rax`
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
- Why is it good for the hardware to do so anyways?

Inst0

Fetch  Reg  Decode  Reg  Execute  Reg  Memory  Reg  Write back  Reg
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
  • Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
- Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

• Why is it good for the hardware to do so anyways?
Can we have the hardware automatically generate a nop?
- Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
  • Why is it good for the hardware to do so anyways?

Fetch | Decode | bubble (nop) | Execute | Memory | Write back | Reg | Reg | Reg | Reg
Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?

![Diagram of the instruction pipeline with stages and bubbles.

- Fetch
- Decode
- Execute
- Memory
- Write back

Inst0 Inst1 bubble (nop) Inst2 Inst3

Stall

Reg Reg Reg Reg Reg Reg
Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
  • Why is it good for the hardware to do so anyways?
Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?
• Why is it good for the hardware to do so anyways?

Fetch  |  Reg  |  Decode  |  Reg  |  Execute  |  Reg  |  Memory  |  Reg  |  Write back  |  Reg
Inst4  |       | Inst3    |       | Inst2     |       | bubble (nop) |     |             |
Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?
How are Stall and Bubble Implemented in Hardware?
How are Stall and Bubble Implemented in Hardware?

Normal

Input = y

Output = x

stall = 0

bubble = 0
How are Stall and Bubble Implemented in Hardware?

Normal

Input = y  Output = x

Rising clock

Output = y

stall = 0  bubble = 0
How are Stall and Bubble Implemented in Hardware?

**Normal**

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

**Stall**

- Input = y
- Output = x
- stall = 1
- bubble = 0
How are Stall and Bubble Implemented in Hardware?

**Normal**

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

**Stall**

- Input = y
- Output = x
- Stall = 1
- Bubble = 0
How are Stall and Bubble Implemented in Hardware?

**Normal**

Input = y, Output = x

\[
\text{Output} = x
\]

- stall = 0
- bubble = 0

**Stall**

Input = y, Output = x

\[
\text{Output} = x
\]

- stall = 1
- bubble = 0

**Bubble**

Input = y, Output = x

\[
\text{Output} = x
\]

- stall = 0
- bubble = 1
How are Stall and Bubble Implemented in Hardware?

**Normal**

Input = y  
Output = x  

\[ \text{Rising clock} \]

\[ \text{Output = } y \]

\[ \text{Input = } y \]

\[ \text{Output = } x \]

\[ \text{Rising clock} \]

\[ \text{Output = } y \]

\[ \text{stall} = 0 \]

\[ \text{bubble} = 0 \]

**Stall**

Input = y  
Output = x  

\[ \text{Rising clock} \]

\[ \text{Output = } x \]

\[ \text{Input = } y \]

\[ \text{Output = } x \]

\[ \text{Rising clock} \]

\[ \text{Output = } x \]

\[ \text{stall} = 1 \]

\[ \text{bubble} = 0 \]

**Bubble**

Input = y  
Output = x  

\[ \text{Rising clock} \]

\[ \text{Output = } \text{nop} \]

\[ \text{Input = } y \]

\[ \text{Output = } x \]

\[ \text{Rising clock} \]

\[ \text{Output = } \text{nop} \]

\[ \text{stall} = 0 \]

\[ \text{bubble} = 1 \]
Detecting Stall Condition

0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop

*bubble*
0x016: addq %rdx,%rax
0x018: halt

- Using a “scoreboard”. Each register has a bit.
- Every instruction that writes to a register sets the bit.
- Every instruction that reads a register would have to check the bit first.
  - If the bit is set, then generate a bubble
  - Otherwise, free to go!!
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` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

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

- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```
Data Forwarding Example #2

Register \%rdx
- Forward from the memory stage

Register \%rax
- Forward from the execute stage

0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
Data Forwarding Example #2

Register %rdx
- Forward from the memory stage

Register %rax
- Forward from the execute stage
Data Forwarding Example #2

Register %rdx

- Forward from the memory stage

Register %rax

- Forward from the execute stage
Out-of-order Execution

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

Long-latency instruction.
Forces the pipeline to stall.

\[
\begin{align*}
\mathbf{r}_0 &= \mathbf{r}_1 + \mathbf{r}_2 \\
\mathbf{r}_3 &= \text{MEM}[\mathbf{r}_0] \\
\mathbf{r}_4 &= \mathbf{r}_3 + \mathbf{r}_6 \\
\mathbf{r}_7 &= \mathbf{r}_5 + \mathbf{r}_1 \\
\ldots
\end{align*}
\]

- Compiler could do this, but has limitations
- Generally done in hardware
Out-of-order Execution

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

Long-latency instruction.
Forces the pipeline to stall.

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

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

\[
\begin{align*}
  r_4 &= r_3 + r_6
\end{align*}
\]
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

\[
\begin{align*}
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r4 &= r3 + r6 \\
    r6 &= r5 + r1 \\
\end{align*}
\]

Is this correct?

\[
\begin{align*}
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r6 &= r5 + r1 \\
    r4 &= r3 + r6 \\
\end{align*}
\]
Out-of-order Execution

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

Is this correct?

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

\[
\begin{align*}
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r4 &= r3 + r6 \\
    r6 &= r5 + r1 \\
    \quad \vdots \\
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r4 &= r3 + r6 \\
    r4 &= r5 + r1 \\
    \quad \vdots \\
\end{align*}
\]

Is this correct?

\[
\begin{align*}
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r4 &= r3 + r6 \\
    r6 &= r5 + r1 \\
    \quad \vdots \\
    r0 &= r1 + r2 \\
    r3 &= \text{MEM}[r0] \\
    r4 &= r3 + r6 \\
    r4 &= r5 + r1 \\
    \quad \vdots \\
\end{align*}
\]
Out-of-order Execution

\[
\begin{align*}
\text{r0} &= \text{r1} + \text{r2} \\
\text{r3} &= \text{MEM}[\text{r0}] \\
\text{r4} &= \text{r3} + \text{r6} \\
\text{r6} &= \text{r5} + \text{r1} \\
\end{align*}
\]

Is this correct?

\[
\begin{align*}
\text{r0} &= \text{r1} + \text{r2} \\
\text{r3} &= \text{MEM}[\text{r0}] \\
\text{r4} &= \text{r3} + \text{r6} \\
\text{r4} &= \text{r5} + \text{r1} \\
\end{align*}
\]

\[
\begin{align*}
\text{r0} &= \text{r1} + \text{r2} \\
\text{r3} &= \text{MEM}[\text{r0}] \\
\text{r4} &= \text{r3} + \text{r6} \\
\text{r4} &= \text{r3} + \text{r6} \\
\end{align*}
\]

“Tomasolu Algorithm” is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.