Superscalar* Club Meeting #1

*we really mean: superscalar speculative out-of-order

James C. Hoe
Department of ECE
Carnegie Mellon University
Assumptions

- You have taken 18-447
- You have read *Metaflow*
- You have tried to read *R10K*
- You are here to learn
- You will put in the effort
Goal

• Achieve an RTL-precise understanding of superscalar speculative out-of-order register dataflow—as with 5-stage pipeline in 18-447
  – know you could make a working design and know what it does (how good is it?)
  – know shortcomings and limits in the simplifications you chose to make
  – know what you have tried to but not figured out
  – have a gut-feel for the boundary between known-unknown and unknown-unknown
Plan of Attack

• Focus
  – mainly on register dataflow
  – lightly on memory dataflow
  – not at all on i-fetch (a well decoupled subject both conceptually and physically)

• Path
  – further develop concepts in L20 we didn’t have time for
  – study Metaflow DRIS to flesh out conceptual-level understanding
  – study how things were really done in R10K
  – play with an RTL-precise executable model (in C++)
Let’s Get Started
Parallelism Defined

- **$T_1$** (work measured in time):
  - time to do work with 1 PE
- **$T_\infty$** (critical path):
  - time to do work with infinite PEs
  - $T_\infty$ bounded by dataflow dependence
- Average parallelism:
  
  $$P_{avg} = \frac{T_1}{T_\infty}$$

- For a system with $p$ PEs
  
  $$T_p \geq \max\{\frac{T_1}{p}, T_\infty\}$$

- When $P_{avg} \gg p$
  
  $$T_p \approx \frac{T_1}{p},$$ aka “linear speedup”
**ILP: Instruction-Level Parallelism**

- Average **ILP** = \( \frac{T_1}{T_\infty} \)
  = no. instruction / no. cyc required

  **code1:** \( \text{ILP} = 1 \)
  i.e., must execute serially

  **code2:** \( \text{ILP} = 3 \)
  i.e., can execute at the same time

<table>
<thead>
<tr>
<th>code1:</th>
<th>r1 ← r2 + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r3 ← r1 / 17</td>
</tr>
<tr>
<td></td>
<td>r4 ← r0 - r3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code2:</th>
<th>r1 ← r2 + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r3 ← r9 / 17</td>
</tr>
<tr>
<td></td>
<td>r4 ← r0 - r10</td>
</tr>
</tbody>
</table>
Superscalar Speculative Out-of-Order Execution
Exploiting **ILP** for Performance

Scalar in-order pipeline with forwarding

- operation latency (OL) = 1 base cycle
- peak **IPC** = 1 \(//\) no concurrency
- required **ILP** \(\geq 1\) to avoid stall

[Internal Use Draft] Superscalar Club Meeting #1, Slide 9, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Superpipelined Execution

\[ \text{OL} = M \text{ minor-cycle}; \text{ same as 1 base cycle} \]

peak \( \text{IPC} = 1 \) per minor-cycle // has concurrency though

required \( \text{ILP} \geq M \)

Achieving full performance requires always finding \( M \) “independent” instructions in a row
Superscalar (Inorder) Execution

\[ \text{OL} = 1 \text{ base cycle} \]
peak \( \text{IPC} = N \)
required \( \text{ILP} \geq N \)

Achieving full performance requires finding \( N \) “independent” instructions on every cycle.
In-order Superscalar

Pipe A

I-cache

Reg File Read

ALU

D-cache

Reg File Write

Pipe B

PC

2 X fetch bandwidth

2 X read ports

2 X Logic

Can't always double resources

2 X write ports

No!
Limitations of Inorder Pipeline

- Achieved **IPC** of inorder pipelines degrades rapidly as **NxM** approaches **ILP**
- Despite high concurrency potential, pipeline never full due to frequent dependency stalls!!
Out-of-Order Execution

- **ILP** is scope dependent

\[
\begin{align*}
\text{ILP}=1 & \\
r1 & \leftarrow r2 + 1 \\
r3 & \leftarrow r1 / 17 \\
r4 & \leftarrow r0 - r3 \\
r11 & \leftarrow r12 + 1 \\
r13 & \leftarrow r19 / 17 \\
r14 & \leftarrow r0 - r20
\end{align*}
\]

\[
\begin{align*}
\text{ILP}=2 & \\
r1 & \leftarrow r2 + 1 \\
r3 & \leftarrow r1 / 17 \\
r4 & \leftarrow r0 - r3 \\
r11 & \leftarrow r12 + 1 \\
r13 & \leftarrow r19 / 17 \\
r14 & \leftarrow r0 - r20
\end{align*}
\]

Accessing **ILP**=2 requires (1) larger scheduling window but also (2) out-of-order execution (even Lab 4 can go slightly OOO)
Superscalar Speculative Out-of-Order Execution
von Neuman vs Dataflow

- Consider a von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

\[
\begin{align*}
  v &:= a + b; \\
  w &:= b \times 2; \\
  x &:= v - w; \\
  y &:= v + w; \\
  z &:= x \times y;
\end{align*}
\]

- Dataflow program instruction ordering implied by data dependence
  - instruction specifies who receives the result
  - instruction executes when operands received
  - no program counter, no intermediate state

[dataflow figure and example from Arvind]
Instruction “Micro-Dataflow”

• Maintain a buffer of many pending instructions, a.k.a. reservation stations (RSs)
  – wait for functional unit to be free
  – wait for register RAW hazards to resolve (i.e., required input operands to be produced)

• Decouple execution order from who is first in line (program order)
  – select inst’s in RS whose operands are available
  – give preference to older instructions (heuristical)

• A completing instruction frees pending, RAW-dependent instructions to execute

*What about WAW and WAR?*
IBM 360/91 FP Module [1967]

ignore for now

out-of-order issue window
Tomasulo’s Algorithm [IBM 360/91, 1967]

- Dispatch an instruction to a RS slot after decode
  - decode received from RF either operand value or placeholder RS-tag
  - mark RF dest with RS-tag of current inst’s RS slot
- Inst in RS can issue when all operand values ready
- Completing instruction, in addition to updating RF dest, broadcast its RS-tag and value to all RS slots
- RS slot holding matching RS-tag placeholder pickup value
# Tomasulo’s Algorithm

<table>
<thead>
<tr>
<th>Instruction state</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue FP operation</td>
<td>Station r empty</td>
<td>if (RegisterStat[r].Qi!=0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qi ← RegisterStat[r].Qi}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vj ← Regs[r]; RS[r].Qi ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (RegisterStat[r].Qi!=0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qk ← RegisterStat[r].Qk}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vk ← Regs[r]; RS[r].Qk ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes; RegisterStat[rd].Qj ← r;</td>
</tr>
<tr>
<td>Load or store</td>
<td>Buffer r empty</td>
<td>if (RegisterStat[r].Qi!=0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qj ← RegisterStat[r].Qj}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vj ← Regs[r]; RS[r].Qj ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes;</td>
</tr>
<tr>
<td>Load only</td>
<td></td>
<td>RS[r].Qj ← r;</td>
</tr>
<tr>
<td>Store only</td>
<td></td>
<td>if (RegisterStat[r].Qi!=0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qk ← RegisterStat[r].Qk}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vk ← Regs[r]; RS[r].Qk ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes;</td>
</tr>
<tr>
<td>Execute FP operation</td>
<td>(RS[r].Qi = 0) and (RS[r].Qk = 0)</td>
<td>Compute result: operands are in Vj and Vk</td>
</tr>
<tr>
<td>Load-store step 1</td>
<td>RS[r].Qj = 0 &amp; r is head of load-store queue</td>
<td>RS[r].Qj = 0 &amp; r is head of load-store queue</td>
</tr>
<tr>
<td>Load step 2</td>
<td></td>
<td>RS[r].A ← RS[r].Vj ← RS[r].A;</td>
</tr>
<tr>
<td>Write Result FP operation or load</td>
<td>Execution complete at r &amp; CDB available</td>
<td>∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RS[x].Qj=r) {RS[x].Qj ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RS[x].Qk=r) {RS[x].Vj ← result;RS[x].Qk ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RS[x].Qk=r) {RS[x].Vj ← result;RS[x].Qk ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← no;</td>
</tr>
<tr>
<td>Store</td>
<td>Execution complete at r &amp;</td>
<td>Mem(RS[r].A) ← RS[r].Vj;</td>
</tr>
<tr>
<td></td>
<td>CDB available</td>
<td>RS[r].Qj ← 0;</td>
</tr>
</tbody>
</table>

- **RS entry**
  - **Busy**: in use
  - **Op**: operand
  - **Vj**: op1 value
  - **Vk**: op2 value
  - **Qj**: op1 RS-tag
  - **Qk**: op2 RS-tag

Case (RegisterStat)
- 0: RF val current RS-tag:
  - to be produce by corresponding instruction

[Internal Use Draft] Superscalar Club Meeting #1, Slide 20, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Tomasulo’s Algorithm (caption)

Figure 2.12 Steps in the algorithm and what is required for each step. For the issuing instruction, \( rd \) is the destination, \( rs \) and \( rt \) are the source register numbers, \( imm \) is the sign-extended immediate field, and \( r \) is the reservation station or buffer that the instruction is assigned to. \( RS \) is the reservation station data structure. The value returned by an FP unit or by the load unit is called \( result \). \( RegisterStat \) is the register status data structure (not the register file, which is \( Regs [] \)). When an instruction is issued, the destination register has its \( Qi \) field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the \( V \) fields. Otherwise, the \( Q \) fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the \( Q \) fields. The \( Q \) fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the \( CDB \) is available, it can do its write back. All the buffers, registers, and reservation stations whose value of \( Qi \) or \( Qk \) is the same as the completing reservation station update their values from the \( CDB \) and mark the \( Q \) fields to indicate that values have been received. Thus, the \( CDB \) can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in \( Execute \), and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step, if there is a pending branch already in the pipeline. In Section 2.6, we will see how speculation support removes this restriction.

[Hennessy&Patterson, CAAQA]
Invariants

RegisterStat[\textbf{rs}].Qi\neq0 \Rightarrow

\text{let } q = \text{RegisterStat}[\textbf{rs}].Qi \text{ in }
\begin{align*}
& \text{RS}[q].Busy = \text{true } \land \\
& \text{rs is logical dest of inst in } \text{RS}[q] \\
\end{align*}

?RegisterStat[\textbf{rs}].Qi=0 \Rightarrow \forall q \ ( \text{rs is not dest of inst in } \text{RS}[q] )?

\text{RS}[r].Busy \land \text{RS}[r].Qj\neq0 \Rightarrow

\text{let}\ \{q = \text{RS}[r].Qj; rs=operand of inst in } \text{RS}[r]\ \}\text{ in }
\begin{align*}
& \text{RS}[q].Busy \land \\
& \text{rs is logical dest of inst in } \text{RS}[q] \land \\
& \text{inst in } \text{RS}[q] \text{ is older than inst in } \text{RS}[r] \land \\
& \text{no } \text{RS}[s] \text{ hold inst older than } \text{RS}[q] \text{ younger than } \text{RS}[r] \text{ and has } \textbf{rs} \text{ as dest}
\end{align*}
RAW Example: \[ i: R2 \leftarrow R0 + R4 \]

\[ j: R8 \leftarrow R0 + R2 \]
**RAW Example:**

**Cyc #1: issue $i$**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>6.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

**Adder (i)**

**Cyc #2: $i$ in RS 1 broadcasts tag and result: $CBD=<<1, 16.0>>$**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>1</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Adder (i)**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>2</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Cyc #3:**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>2</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Adder (i)**

**Note:**

- Assume 1 cyc-add, 2-cyc mult
- Issue up to 1 add & 1 mult per cycle
- Complete up to 1 add & 1 mult per cycle
WAW and WAR in Tomasulo??

- No WAW and WAR in 5-stage pipeline because
  - single write stage
  - write stage at the end (later than any read stage)
  - in-order progression in pipeline
### WAR Example:

#### Cyc #1:

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Cyc #2:

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Adder**

#### Cyc #3:

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Adder**

### Instructions:

- **i:** $R4 \leftarrow R0 \times R8$
- **j:** $R0 \leftarrow R4 \times R2$
- **k:** $R2 \leftarrow R2 + R8$

#### FLR Table:

<table>
<thead>
<tr>
<th>FLR</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### FLR Table:

<table>
<thead>
<tr>
<th>FLR</th>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**WAR Example:**

**Cyc #1: issue i**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Adder**

**Cyc #2: issue j & k (assume dual issue)**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3.5</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Adder (k)**

**Cyc #3: RS 1 and 4 completes CBD=<<1,11.3>> & <<4,46,8>>**

[Internal Use Draft] Superscalar Club Meeting #1, Slide 27, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
WAW Example:

Cyc #1:

\[ i: R4 \leftarrow R0 \times R8 \]
\[ j: R2 \leftarrow R0 + R4 \]
\[ k: R4 \leftarrow R0 + R8 \]
\[ l: R8 \leftarrow R4 \times R8 \]

Cyc #2:
Adder

Cyc #3:
Adder

Adder

[Internal Use Draft] Superscalar Club Meeting #1, Slide 28, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
WAW Example:

**Cyc #1: issue \(i\) and \(j\)**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>(j) 1</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder

\[\begin{array}{c|ccc}
RS & Tag & Sink & Tag & Src \\
\hline
\(j\) 1 & 0 & 6.0 & 4 & -- \\
2 & & & & \\
3 & & & & \\
\end{array}\]

**Cyc #2: issue \(k\) and \(l\)**

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>(j) 1</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder (k)

\[\begin{array}{c|ccc}
RS & Tag & Sink & Tag & Src \\
\hline
\(j\) 1 & 0 & 6.0 & 4 & -- \\
2 & & & & \\
3 & & & & \\
\end{array}\]

**Cyc #3: RS 2 and 4 completes:** \(CBD=<<(2, 13.8)>> \& \<<(4, 46, 8)>>

<table>
<thead>
<tr>
<th>RS</th>
<th>Tag</th>
<th>Sink</th>
<th>Tag</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>(j) 1</td>
<td>0</td>
<td>6.0</td>
<td>0</td>
<td>46.2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>6.0</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder (k)

\[\begin{array}{c|ccc}
RS & Tag & Sink & Tag & Src \\
\hline
\(j\) 1 & 0 & 6.0 & 0 & 46.2 \\
2 & & 6.0 & 0 & 7.8 \\
3 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>7.8</td>
</tr>
</tbody>
</table>

**Mult/Div (i)**

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(i\) 4 & 0 & 6.0 & 0 & 7.8 \\
5 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

**Mult/Div (i)**

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(i\) 4 & 0 & 6.0 & 0 & 7.8 \\
5 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

**Cyc #2:** issue \(k\) and \(l\)

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(j\) 1 & 0 & 6.0 & 0 & 46.2 \\
2 & & 6.0 & 0 & 7.8 \\
3 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

**Mult/Div (i)**

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(i\) 4 & 0 & 6.0 & 0 & 7.8 \\
5 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

**Mult/Div (i)**

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(i\) 4 & 0 & 6.0 & 0 & 7.8 \\
5 & & & & \\
\end{array}\]

**FLR**

<table>
<thead>
<tr>
<th>Busy</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

**WAW?**

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(j\) 1 & 0 & 6.0 & 0 & 46.2 \\
2 & & 6.0 & 0 & 7.8 \\
3 & & & & \\
\end{array}\]

Adder (k)

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(j\) 1 & 0 & 6.0 & 0 & 46.2 \\
2 & & 6.0 & 0 & 7.8 \\
3 & & & & \\
\end{array}\]

Adder (k)

\[\begin{array}{c|ccc}
RS & Tag | Sink | Tag | Src \\
\hline
\(j\) 1 & 0 & 6.0 & 0 & 46.2 \\
2 & & 6.0 & 0 & 7.8 \\
3 & & & & \\
\end{array}\]

[Internal Use Draft] Superscalar Club Meeting #1, Slide 29, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Out-of-order yes, but . . . .

- IPC ≤ 1
  - superscalar execute but actually about hiding multiply latency
  - IPC<<1 when ILP=1: successive RAW-dependent instructions cannot issue back-to-back

- A lot of duplicated operand values stored

- No precise RF state on exception
  - on WAR example: if $J$ raises exception after $K$ already wrote back to RF; can’t undo $K$’s RF update

- No way to rewind if speculative execution

Need still more powerful magic!!
Superscalar Speculative Out-of-Order Execution
Register Renaming
Removing False Dependencies

• With out-of-order execution comes WAW and WAR hazards

• Anti and output dependencies are false dependencies on register names rather than data

\[
\begin{align*}
  r_3 & \leftarrow r_1 \text{ op } r_2 \\
  r_5 & \leftarrow r_3 \text{ op } r_4 \\
  r_3 & \leftarrow r_6 \text{ op } r_7
\end{align*}
\]

• With infinite number of registers, anti and output dependencies avoidable by using a new register for each new value
Register Renaming: Example

Original

\[ r_1 \leftarrow r_2 / r_3 \]
\[ r_4 \leftarrow r_1 \ast r_5 \]
\[ r_1 \leftarrow r_3 + r_6 \]
\[ r_3 \leftarrow r_1 - r_5 \]

Renamed

\[ r_1 \leftarrow r_2 / r_3 \]
\[ r_4 \leftarrow r_1 \ast r_5 \]
\[ r_8 \leftarrow r_3 + r_6 \]
\[ r_9 \leftarrow r_8 - r_5 \]
On-the-fly HW Register Renaming

- Maintain mapping from ISA reg. names to physical registers
- When decoding an instruction that updates ‘rₓ’:
  - allocate unused physical register tᵧ to hold inst result
  - set new mapping from ‘rₓ’ to tᵧ
  - younger instructions using ‘rₓ’ as input finds tᵧ
- De-allocate a physical register for reuse when it is never needed again?

\[
\begin{align*}
  r₁ &\leftarrow r₂ / r₃ \\
  r₄ &\leftarrow r₁ * r₅ \\
  r₁ &\leftarrow r₃ + r₆
\end{align*}
\]
Reorder Buffer
Program State Views

Figure 5-1. Illustration of In-Order, Lookahead, and Architectural States

from [Johnson, 1990]
Instruction Reorder Buffer (ROB)

- Program-order bookkeeping (circular buffer)
  - instructions enter and leave in program order
  - tracks 10s to 100s of in-flight instructions in different stages of execution
- Dynamic juggling of state and dependency
  - oldest finished instruction “commit” architectural state updates on exit
  - all ROB entries considered “speculative” due to potential for exceptions and mispredictions
In-order vs Speculative State

- In-order state:
  - cumulative architectural effects of all instructions committed in-order so far
  - can never be undone!!
- Speculative state, as viewed by a given inst in ROB
  - in-order state + effects of older inst’s in ROB
  - effects of some older inst’s may be pending
- Speculative state effects must be reversible
  - remember both in-order and speculative values for an RF register (may have multiple speculative values)
  - store inst updates memory only at commit time
- Discard younger speculative state to rewind execution to oldest remaining inst in ROB
ROB Rename Registers

from [Gonzalez, et al., 2010]

to “form” architectural state for decode

Architectural Register File
committed
in-order state

Reorder Buffer
look-ahead state

one physical array?
Tomasulo + Speculative Execution

ignore for now
## Tomasulo’s Algorithm + ROB

<table>
<thead>
<tr>
<th>Status</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue all instructions</td>
<td>if (RegisterStat[rs].Busy) /* in-flight instr. writes rs */</td>
<td>{h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */</td>
</tr>
<tr>
<td></td>
<td>{RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}</td>
<td>} else {RS[r].Qj ← h;} /* wait for instruction */</td>
</tr>
<tr>
<td>Reservation station (r)</td>
<td>else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};</td>
<td></td>
</tr>
<tr>
<td>and ROB (b) both available</td>
<td>RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no;</td>
<td></td>
</tr>
<tr>
<td>FP operations</td>
<td>if (RegisterStat[rt].Busy) /* in-flight instr writes rt */</td>
<td>{h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */</td>
</tr>
<tr>
<td>and stores</td>
<td>{RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;}</td>
<td>} else {RS[r].Qk ← h;} /* wait for instruction */</td>
</tr>
<tr>
<td></td>
<td>else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};</td>
<td></td>
</tr>
<tr>
<td>FP operations</td>
<td>RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;</td>
<td></td>
</tr>
<tr>
<td>Loads</td>
<td>RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;</td>
<td></td>
</tr>
<tr>
<td>Stores</td>
<td>RS[r].A ← imm;</td>
<td></td>
</tr>
</tbody>
</table>

### ROB Entry:

- is **Busy / Instruction / logical Dest reg / dest Value / value** is **Ready**
- RegisterStat: reg is **Busy / renamed to Reorder** buffer entry #

[Internal Use Draft] Superscalar Club Meeting #1, Slide 42, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
### Tomasulo’s Algorithm + ROB

<table>
<thead>
<tr>
<th>Execute</th>
<th>( \text{FP op} ) ( (\text{RS}[r].Qj == 0) ) and ( (\text{RS}[r].Qk == 0) )</th>
<th>Compute results—operands are in ( V_j ) and ( V_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load step 1</td>
<td>( (\text{RS}[r].Qj == 0) ) and there are no stores earlier in the queue</td>
<td>( \text{RS}[r].A \leftarrow \text{RS}[r].V_j + \text{RS}[r].A );</td>
</tr>
<tr>
<td>Load step 2</td>
<td>Load step 1 done and all stores earlier in ROB have different address</td>
<td>Read from ( \text{Mem} {\text{RS}[r].A} )</td>
</tr>
<tr>
<td>Store</td>
<td>( (\text{RS}[r].Qj == 0) ) and store at queue head</td>
<td>( \text{ROB}[h].Address \leftarrow \text{RS}[r].V_j + \text{RS}[r].A );</td>
</tr>
<tr>
<td>Write result</td>
<td>Execution done at ( r ) and CDB available</td>
<td>( b \leftarrow \text{RS}[r].\text{Dest}; \text{RS}[r].\text{Busy} \leftarrow \text{no}; )</td>
</tr>
<tr>
<td></td>
<td>( \forall x { \text{if} \ (\text{RS}[x].Qj == b) \ { \text{RS}[x].V_j \leftarrow \text{result}; \text{RS}[x].Qj \leftarrow 0 } }; )</td>
<td>( \forall x { \text{if} \ (\text{RS}[x].Qk == b) \ { \text{RS}[x].V_k \leftarrow \text{result}; \text{RS}[x].Qk \leftarrow 0 } }; )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{ROB}[b].\text{Value} \leftarrow \text{result}; \text{ROB}[b].\text{Ready} \leftarrow \text{yes} );</td>
</tr>
<tr>
<td>Store</td>
<td>Execution done at ( r ) and ( (\text{RS}[r].Qk == 0) )</td>
<td>( \text{ROB}[h].\text{Value} \leftarrow \text{RS}[r].V_k );</td>
</tr>
<tr>
<td>Commit</td>
<td>Instruction is at the head of the ROB (entry ( h )) and ( \text{ROB}[h].\text{ready} == \text{yes} )</td>
<td>( d \leftarrow \text{ROB}[h].\text{Dest}; /* register dest, if exists */ )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ( (\text{ROB}[h].\text{Instruction} == \text{Branch}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ if (branch is mispredicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ clear \text{ROB}[h], \text{RegisterStat}; fetch branch dest; }; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if ( (\text{ROB}[h].\text{Instruction} == \text{Store}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ \text{Mem}{\text{ROB}[h].\text{Destination}} \leftarrow \text{ROB}[h].\text{Value}; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else /* put the result in the register destination */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ \text{Regs}[d] \leftarrow \text{ROB}[h].\text{Value}; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{ROB}[h].\text{Busy} \leftarrow \text{no}; /* free up ROB entry */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/* free up dest register if no one else writing it */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ( (\text{RegisterStat}[d].\text{Reorder} == h) ) { \text{RegisterStat}[d].\text{Busy} \leftarrow \text{no}; };</td>
</tr>
</tbody>
</table>

---

[Internal Use Draft] Superscalar Club Meeting #1, Slide 43, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Invariants

• You give it try . . .
Speculative Out-of-order Execution

- A mispredicted branch after resolution must be rewound and restarted and fast!
- Much trickier than 5-stage pipeline . . .
  - can rewind to an intermediate speculative state
  - a rewound branch could still be speculative and itself be discarded by another rewind!
  - rewind must reestablish both architectural state (register value) and microarchitecture state (e.g., rename table)
  - rewind/restart must be fast (not infrequent)
- Also need to rewind on exceptions . . . . but easier
ASAP Midpoint ROB Rewind

```
\begin{center}
\begin{tikzpicture}
\draw[blue, thick, ->] (0,0) -- (0,5) node[above] {head};
\draw[blue, thick, ->] (0,0) -- (0,-5) node[below] {tail};
\draw[blue, thick] (0,0) -- (1,0) -- (1,5) -- (0,5);%
\node at (0.5,5) {oldest};
\node at (0.5,-5) {youngest};
\node at (0.5,0) {mispredict};
\end{tikzpicture}
\end{center}
```
Nested Control Flow Speculation
Mis-speculation Recovery can be Speculative
Speculative Rewinding with ROB

- In-order RF state never needs undo’ing ✓
- Lookahead RF state tied to ROB ✓
- Restoring architectural state view (i.e. map table)
  - at decode, record in ROB, the logical dest and the overwritten previous mapping
  - on rewind, walk-back ROB one entry at a time to restore register map table

What happens if previous mapping is to an already retired ROB entry? How to know that?

[Internal Use Draft] Superscalar Club Meeting #1, Slide 49, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Killing In-Flight Instruction

- Tag in-flight instructions by their program-ordered ROB index
- On a mispredicted branch, selectively kill younger instructions based on index comparison
- How to determine if index $i$ “younger than” $j$?

[Diagram showing ROB structure with head, tail, oldest, active, and youngest entries]
What do we know so far

• Out-of-order: yes, except IPC<<1 when ILP=1
• Superscalar: not exactly
  – superscalar execute out of multiple RS
  – everything else discussed as single atomic action
  – how to do more than 1/cyc quickly and cheaply?
• Speculative: yes, except
  – rewind take as long as degree out-of-order
  – still need to a way to decide, given two ROB entry indices which is younger

Building what we said today would only be bigger and slower than your 5-stage pipeline
Reading for Next Week

- Metaflow DRIS
- Memory Dataflow
  - Section 6.4 of Gonzalez, et al.
  - “Memory Reference Instructions”, Metaflow, p65.
  - “Address Queue”, R10K, p34.
  - “Memory Hierarchy”, R10K, p37.
References

• Yeager, MIPS R10K Superscalar Microprocessor, 1996.