18-447: Computer Architecture
Lecture 13: Out-of-Order Execution

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2012, 2/29/2012
Reminder: Homeworks

- Homework 4
  - Out today
  - Due March 19
  - Strongly recommended to complete in preparation for the exam
Reminder: Lab Assignments

- Lab Assignment 4
  - Out today
  - Due March 23 – Start early.
Don’t Forget: Feedback Sheet for Us

- Please turn in today
Midterm I Next Wednesday

- March 7, in class (12:30-2:20pm sharp)
- Closed book, closed notes
- Can bring a single 8.5x11” cheat sheet
- All topics we covered can be on the exam
  - Lectures, labs, homeworks, required readings
- A variety of problem solving, design, and tradeoff analysis questions
- Come to class early (12:25pm)
  - We will end at 2:20pm sharp
Readings for Today

- P&H Chapter 4.9-4.11
  - More advanced pipelining
  - Interrupts and exception handling
  - Out-of-order and superscalar execution concepts
CALCM Seminar Today 4-5pm

- Feb 29, 2012, Wednesday
- Hamerschlag Hall D-210

- Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design
- Ioan Stefanovici, Toronto
Review of Last Lecture

- N-modular redundancy

- Control dependence handling in pipelined machines
  - Wrap up branch prediction: global, local, hybrid predictors
  - Predicated execution
  - Multipath execution

- Precise exceptions/interrupts
Today

- Wrap up precise exceptions
- Out-of-order execution
Pipelining and Precise Exceptions: Preserving Sequential Semantics
Review: Multi-Cycle Execution

- Not all instructions take the same amount of time for “execution”

- Idea: **Have multiple different functional units that take different number of cycles**
  - Can be pipelined or not pipelined
  - Can let independent instructions to start execution on a different functional unit before a previous long-latency instruction finishes execution
Review: Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

<table>
<thead>
<tr>
<th>Instruction</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL R4 ← R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R3 ← R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL R2 ← R5, R6</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R4 ← R5, R6</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is wrong with this picture?
  - What if FMUL incurs an exception?
  - Sequential semantics of the ISA NOT preserved!
Review: Exceptions vs. Interrupts

- Exceptions versus interrupts
- Cause
  - Exceptions: internal to the running thread
  - Interrupts: external to the running thread
- When to Handle
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check
- Priority: process (exception), depends (interrupt)
- Handling Context: process (exception), system (interrupt)
Precise Exceptions/Interrupts

- The architectural state should be consistent when the exception/interrupt is ready to be handled

1. All previous instructions should be completely retired.

2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state
Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. dataflow

- Aids software debugging

- Enables (easy) recovery from exceptions, e.g. page faults

- Enables (easily) restartable processes

- Enables traps into software (e.g., software implemented opcodes)
Ensuring Precise Exceptions in Pipelining

- **Idea:** Make each operation take the same amount of time

<table>
<thead>
<tr>
<th>FMUL R3 ← R1, R2</th>
<th>ADD R4 ← R1, R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
<tr>
<td>F D E E E E E E E E E E E E W</td>
<td></td>
</tr>
</tbody>
</table>

- **Downside**
  - What about memory operations?
  - Each functional unit takes 500 cycles?
Solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing

Recommended Reading
Solution I: Reorder Buffer (ROB)

- **Idea:** Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory
What’s in a ROB Entry?

- Need valid bits to keep track of readiness of the result(s)

<table>
<thead>
<tr>
<th>V</th>
<th>DestRegID</th>
<th>DestRegVal</th>
<th>StoreAddr</th>
<th>StoreData</th>
<th>PC</th>
<th>Valid bits for reg/data + control bits</th>
<th>Exc?</th>
</tr>
</thead>
</table>
Reorder Buffer: Independent Operations

- Results first written to ROB, then to register file at commit time

- What if a later operation needs a value in the reorder buffer?
  - Read reorder buffer in parallel with the register file. How?
A register value can be in the register file, reorder buffer, (or bypass paths)
Simplifying Reorder Buffer Access

- Idea: Use indirection
- Access register file first
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry
- Access reorder buffer next

- What is in a reorder buffer entry?

<table>
<thead>
<tr>
<th>V</th>
<th>DestRegID</th>
<th>DestRegVal</th>
<th>StoreAddr</th>
<th>StoreData</th>
<th>PC/IP</th>
<th>Control/valid bits</th>
<th>Exc?</th>
</tr>
</thead>
</table>

- Can it be simplified further?
What is Wrong with This Picture?

- What is R4’s value at the end?
  - The first FMUL’s result
  - Output dependency not respected
Register Renaming with a Reorder Buffer

- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - They exist due to lack of register ID’s (i.e. names) in the ISA

- The register ID is renamed to the reorder buffer entry that will hold the register’s value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register

- This eliminates anti- and output- dependencies
  - Gives the illusion that there are a large number of registers
In-Order Pipeline with Reorder Buffer

- **Decode (D):** Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E):** Instructions can complete out-of-order
- **Completion (R):** Write result to reorder buffer
- **Retirement/Commit (W):** Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
Reorder Buffer Pros and Cons

- **Pro**
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependencies

- **Con**
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection $\rightarrow$ increased latency and complexity

- **Other solutions aim to eliminate the cons**
  - History buffer
  - Future file
  - Checkpointing
Solution III: Future File (FF) + ROB

- **Idea:** *Keep two register files (speculative and architectural)*
  - Arch reg file: Updated in program order for precise exceptions
    - Use a reorder buffer to ensure in-order updated
  - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)

- Future file is used for fast access to latest register values (speculative state)
  - Frontend register file

- Architectural file is used for state recovery on exceptions (architectural state)
  - Backend register file
Future File

- **Advantage**
  - No need to read the values from the ROB (no CAM or indirection)

- **Disadvantage**
  - Multiple register files
In-Order Pipeline with Future File and Reorder Buffer

- **Decode (D):** Access future file, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E):** Instructions can complete out-of-order
- **Completion (R):** Write result to reorder buffer and future file
- **Retirement/Commit (W):** Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
Checking for and Handling Exceptions in Pipelining

- When the **oldest instruction ready-to-be-retired is detected to have caused an exception**, the control logic
  - Recovers architectural state (register file, IP, and memory)
  - Flushes all younger instructions in the pipeline
  - Saves IP and registers (as specified by the ISA)
  - Redirects the fetch engine to the exception handling routine
    - Vectored exceptions
Out-of-Order Execution
(Dynamic Instruction Scheduling)
Review: In-order pipeline

- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to a functional unit
Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

  \[
  \begin{align*}
  &\text{IMUL } R3 \leftarrow R1, R2 \\
  &\text{ADD } R3 \leftarrow R3, R1 \\
  &\text{ADD } R1 \leftarrow R6, R7 \\
  &\text{IMUL } R5 \leftarrow R6, R8 \\
  &\text{ADD } R7 \leftarrow R3, R5 \\
  &\text{LD } R3 \leftarrow R1 (0) \\
  &\text{ADD } R3 \leftarrow R3, R1 \\
  &\text{ADD } R1 \leftarrow R6, R7 \\
  &\text{IMUL } R5 \leftarrow R6, R8 \\
  &\text{ADD } R7 \leftarrow R3, R5
  \end{align*}
  \]

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later \textit{independent} instructions cannot get executed

- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture
Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen THREE:
  - 1. Fine-grained multithreading
  - 2. Value prediction
  - 3. Compile-time instruction scheduling/reordering
- What are the disadvantages of the above three?
- Any other way to prevent dispatch stalls?
  - Actually, you have briefly seen the basic idea before
    - Dataflow: fetch and “fire” an instruction when its inputs are ready
    - Problem: in-order dispatch (issue, execution)
    - Solution: out-of-order dispatch (issue, execution)
Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source “values” of each instruction in the resting area

- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation
In-order vs. Out-of-order Dispatch

- In order dispatch:

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>STALL</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>STALL</td>
<td>D</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>STALL</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Tomasulo + precise exceptions:

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>WAIT</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>WAIT</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 16 vs. 12 cycles

- IMUL  R3 ← R1, R2
- ADD   R3 ← R3, R1
- ADD   R1 ← R6, R7
- IMUL  R5 ← R6, R8
- ADD   R7 ← R3, R5
Enabling OoO Execution

1. Need to link the consumer of a value to the producer
   - **Register renaming**: Associate a “tag” with each data value

2. Need to buffer instructions until they are ready
   - Insert instruction into *reservation stations* after renaming

3. Instructions need to keep track of readiness of source values
   - **Broadcast the “tag”** when the value is produced
   - Instructions **compare their “source tags”** to the broadcast tag
     → if match, source value becomes ready

4. When all source values of an instruction are ready, dispatch the instruction to functional unit (FU)
   - What if more instructions become ready than available FUs?
Tomasulo’s Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units

- What is the major difference today?
  - **Precise exceptions**: IBM 360/91 did NOT have this

- Variants used in most high-performance processors
  - Most notably Pentium Pro, Pentium M, Intel Core(2), AMD K series,
  - Alpha 21264, MIPS R10000, IBM POWER5, Oracle UltraSPARC T4
Two Humps in a Modern Pipeline

- **Hump 1:** Reservation stations (scheduling window)
- **Hump 2:** Reordering (reorder buffer, aka instruction window or active window)
Tomasulo’s Machine: IBM 360/91

- From memory: load buffers
- From instruction unit: operation bus
- FP registers
- Reservation stations
- Common data bus
- To memory: store buffers
Register Renaming

- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - They exist because not enough register ID’s (i.e. names) in the ISA

- The register ID is renamed to the reservation station entry that will hold the register’s value
  - Register ID → RS entry ID
  - Architectural register ID → Physical register ID
  - After renaming, RS entry ID used to refer to the register

- This eliminates anti- and output- dependencies
  - Approximates the performance effect of a large number of registers even though ISA has a small number
Tomasulo’s Algorithm: Renaming

- Register rename table (register alias table)

<table>
<thead>
<tr>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Tomasulo’s Algorithm

- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!
An Exercise

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine imprecise exceptions (full forwarding)
Exercise Continued

Pipeline structure

```
MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

Mul takes 6 cycles
ADD takes 4 cycles

How many cycles total with data forwarding?

```
F D E W
↓
Can take multiple cycles
```
Exercise Continued

```
FD 1 2 3 4 5 6 W
FD - - - - - D 1 2 3 4 W
F - - - - - D 1 2 3 4 W
FD 1 2 3 4 W
FD - - - - D 1 2 3 4 5 6 W
F - - - - D 1 2 3 4 W

Execution timeline w/ scoreboard
```

31 cycles

```
FD 1 2 3 4 5 6 W
FD E 1 2 3 4 W
F D 1 2 3 4 W
FD 1 2 3 4 W
FD 1 2 3 4 5 6 W
F D 1 2 3 4 W
```

25 cycles
Exercise Continued

Tomasulo's algorithm + full forwarding

20 cycles
How It Works

Register Alias Table

of writer

SRC1

SRC2

Reservation Station for ADDER

Value tag

Assume adder & multiplier have separate buses
### Cycle 0

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>R11</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
Cycle 2:

- MUL $R1, R2 \rightarrow R3$ reads its sources from the RAT.
  - It writes to its destination in the RAT.
  - It renames its destination.
  - It allocates a reservation station entry.
  - It allocates a tag for its destination register.
  - It places its sources in the reservation station entry that is allocated.

End of cycle 2:

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R2$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$R3$</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>$R4$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$R1$</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

- MUL at X becomes ready to execute.
  - What if multiple instructions become ready at the same time?
  - Both of its sources must be valid in the reservation station X.
cycle 3:

- MUL at X starts execution
- ADD R3, R4, RS goes renamed and placed into the ADDR register section

end of cycle 3:

| R1 | 1 ~ 1 |
| R2 | 1 ~ 2 |
| R3 | 0 X ~ 4 |
| R4 | 1 ~ 4 |
| R5 | 0 a ~ |
| R6 | 1 ~ 6 |
| R11| 1 ~ 11 |

- ADD at a cannot be ready to execute because one of its sources is not ready
- It is waiting for the value with the tag X to be broadcast (by the MUL in X)

Aside: Does the tag need to be associated with the RS entry of the producer?
Answer: No: Tag is a tag for the value that is communicated.

RS is a place to hold the values RS enables data flow while they become ready. Like value communicated these two are completely orthogonal.
Cycle 4

cycle 4:  \[ \text{ADD } R2, R6 \rightarrow R7 \text{ gets renamed and placed into } RS \]

end of cycle 4:

- ADD at b becomes ready to execute
  (both sources are ready!)

- At cycle 5, it is sent to the adder out-of-program order
  \[ \rightarrow \text{It is executed before the add in } A \]
Cycle 7

<table>
<thead>
<tr>
<th>R1</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>~</td>
</tr>
<tr>
<td>X2</td>
<td>~</td>
<td>2</td>
</tr>
<tr>
<td>X3</td>
<td>0</td>
<td>~</td>
</tr>
<tr>
<td>X4</td>
<td>0</td>
<td>~</td>
</tr>
<tr>
<td>X5</td>
<td>1</td>
<td>~</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>x2</td>
<td>1</td>
<td>~</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>o</td>
<td>y</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>~</td>
</tr>
</tbody>
</table>

end of cycle 7:

* All 6 meters removed.
- Note what happened to R5
Cycle 8:

- MUL at X and ADD at b
  broadcast their tags and values

  - RS entries waiting for these tags capture the values
    and set the Valid b accordingly

  \[ \rightarrow ( \text{What is needed in HW to accomplish this?}) \]

  CAM on tags that are broadcast for all RS entries & sources

- RAT entries waiting for these tags also capture the values and set the Valid b0s accordingly
An Exercise, with Precise Exceptions

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with reorder buffer (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine with reorder buffer (full forwarding)

<table>
<thead>
<tr>
<th>Add</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>R3 ← R1, R2</td>
</tr>
<tr>
<td>ADD</td>
<td>R5 ← R3, R4</td>
</tr>
<tr>
<td>ADD</td>
<td>R7 ← R2, R6</td>
</tr>
<tr>
<td>ADD</td>
<td>R10 ← R8, R9</td>
</tr>
<tr>
<td>MUL</td>
<td>R11 ← R7, R10</td>
</tr>
<tr>
<td>ADD</td>
<td>R5 ← R5, R11</td>
</tr>
</tbody>
</table>

F D E R W

56
Summary of OOO Execution Concepts

- Renaming eliminates false dependencies

- Tag broadcast enables value communication between instructions $\rightarrow$ dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
    - Limited to the instruction window
  - Can we do it for the whole program? Why would we like to?
    - How can we have a large instruction window efficiently?
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Register Renaming and OoO Execution

- Architectural registers dynamically renamed
  - Mapped to reservation stations

<table>
<thead>
<tr>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>-</td>
<td>V0 1</td>
</tr>
<tr>
<td>R1</td>
<td>S2</td>
<td>new1V1 0</td>
</tr>
<tr>
<td>R2</td>
<td>-</td>
<td>V2 1</td>
</tr>
<tr>
<td>R3</td>
<td>S0</td>
<td>newV3 0</td>
</tr>
<tr>
<td>R4</td>
<td>-</td>
<td>V4 1</td>
</tr>
<tr>
<td>R5</td>
<td>-</td>
<td>V5 1</td>
</tr>
<tr>
<td>R6</td>
<td>-</td>
<td>V6 1</td>
</tr>
<tr>
<td>R7</td>
<td>S4</td>
<td>V7 0</td>
</tr>
<tr>
<td>R8</td>
<td>-</td>
<td>V8 1</td>
</tr>
<tr>
<td>R9</td>
<td>-</td>
<td>V9 1</td>
</tr>
</tbody>
</table>

**Instructions:**

- IMUL R3 ← R1, R2
- IMUL S0 ← V1, V2
- ADD R3 ← R3, R1
- ADD S1 ← S0, V4
- ADD R1 ← R6, R7
- ADD S2 ← V6, V7
- IMUL R3 ← R6, R8
- IMUL S3 ← V6, V8
- ADD R7 ← R3, R9
- ADD S4 ← S3, V9

**Table:**

<table>
<thead>
<tr>
<th>Src1 tag</th>
<th>Src1 value</th>
<th>V?</th>
<th>Src2 tag</th>
<th>Src2 value</th>
<th>V?</th>
<th>Ctl</th>
<th>S?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>-</td>
<td>Retired --- Entry Deallocated ent</td>
<td>nul</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>Retired --- Entry Deallocated ent</td>
<td>add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>Completed --- Wait for Retirement</td>
<td>add</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>-</td>
<td>BROADCAST S3 and new3V3</td>
<td>1</td>
<td>add</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>S3</td>
<td>new3V3</td>
<td>-</td>
<td>V9</td>
<td>1</td>
<td>add</td>
<td></td>
</tr>
</tbody>
</table>
Other Solutions for Precise Exceptions
Solution II: History Buffer (HB)

- **Idea:** Update architectural state when instruction completes, but UNDO UPDATES when an exception occurs

- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head
History Buffer

- **Advantage:**
  - Register file contains up-to-date values. History buffer access not on critical path

- **Disadvantage:**
  - Need to read the old value of the destination
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency
Checkpointing

- **Idea:** Periodically checkpoint the register file state. When exception/interrupt occurs, go back to the most recent checkpoint and re-execute instructions one by one to regenerate exception.

- **State guaranteed to be precise only at checkpoints.**

- **Advantage:**
  - Per-instruction reorder buffer is not needed
  - Allows for aggressive execution between checkpoints

- **Disadvantage:**
  - Interrupt latency depends on distance from checkpoint
  - Number of checkpoints?

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an “exception”
  - Except it is not visible to software

- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction
  - All three state recovery methods can be used

- Difference between exceptions and branch mispredictions?
  - Branch mispredictions more common: need fast recovery
Pipelining Issues: Stores

- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. **Why?**
  - **One idea:** Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - **Store/write buffer:** Similar to reorder buffer, but used only for store instructions
    - Program-order list of un-committed store operations
    - When store is decoded: Allocate a store buffer entry
    - When store address and data become available: Record in store buffer entry
    - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data