18-447
Computer Architecture
Lecture 11: Precise Exceptions, State Maintenance, State Recovery

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 2/10/2014
Announcements

- Homework 2 due Wednesday (Feb 12)
- Lab 3 available online (due Feb 21)
Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts


Readings for Next Few Lectures (II)

Readings Specifically for Today


  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
## Remember: Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently
CALL AND RETURN PREDICTION

- **Direct calls are easy to predict**
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- **Returns are indirect branches**
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - **Observation:** Usually a return matches a call
  - **Idea:** Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
  - Accurate most of the time: 8-entry stack $\rightarrow$ > 95% accuracy
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets

Conditional (Direct) Branch
Indirect Jump

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

br.cond TARGET
R1 = MEM[R2]
branch R1
Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: **Predict the last resolved target as the next fetch address**
  + Simple: Use the BTB to store the target address
  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: **Use history based target prediction**
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  + More accurate
  -- An indirect branch maps to (too) many entries in BTB
    -- Conflict misses with other branches (direct or indirect)
    -- Inefficient use of space if branch has few target addresses
More Ideas on Indirect Branches?

- Virtual Program Counter prediction
  - Idea: Use conditional branch prediction structures iteratively to make an indirect branch prediction
  - i.e., devirtualize the indirect branch in hardware

- Curious?
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

How do we do this?
- BTB hit $\rightarrow$ indicates that the fetched instruction is a branch
- BTB entry contains the “type” of the branch

What if no BTB?
- Bubble in the pipeline until target address is computed
- E.g., IBM POWER4
Issues in Branch Prediction (II)

- **Latency:** Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower
“Superscalar” processors
- attempt to execute more than 1 instruction-per-cycle
- must fetch multiple instructions per cycle

Consider a 2-way superscalar fetch scenario
(case 1) Both insts are not taken control flow inst
  - nPC = PC + 8
(case 2) One of the insts is a taken control flow inst
  - nPC = predicted target addr
  - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
  - If the 1\textsuperscript{st} instruction is the predicted taken branch
    \( \rightarrow \) nullify 2\textsuperscript{nd} instruction fetched
Multiple Instruction Fetch: Concepts

Fetch 1 inst/cycle

- Downside:
  Flynn’s bottleneck
  If you fetch 1 inst/cycle, you cannot finish >1 inst/cycle

Fetch 4 inst/cycle

Two major approaches

1) VLIW
   Compiler decides what rests
   can be executed in parallel
   → Simple hardware

2) Superscalar
   Hardware detects dependencies
   between instructions that are fetched in the same cycle.
Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multithreading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
  - Predicated execution
  - Multipath execution
Pipelining and Precise Exceptions: Preserving Sequential Semantics
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
Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply
    
    | FMUL R4 ← R1, R2 | F D E E E E E E E E E E W |
    | ADD R3 ← R1, R2  | F D E W                     |
    | FMUL R2 ← R5, R6 | F D E W                     |
    | ADD R4 ← R5, R6  | F D E W                     |

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

  - FMUL R3 ← R1, R2
  - ADD R4 ← R1, R2

- **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?
Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding 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?
Aside: 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 $\rightarrow$ ROB entry ID
  - Architectural register ID $\rightarrow$ 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 Tradeoffs

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

- **Disadvantages**
  - 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 disadvantages**
  - History buffer
  - Future file
  - Checkpointing
Solution II: History Buffer (HB)

- **Idea:** Update the register file 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 register
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency
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 updates
  - 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
  - Need to copy arch. reg. file to future file on an exception
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, copy architectural file to future file, and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
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
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 are much more common
    → need fast state recovery to minimize performance impact of mispredictions
How Fast Is State Recovery?

- Latency of state recovery affects
  - Exception service latency
  - Interrupt service latency
  - Latency to supply the correct data to instructions fetched after a branch misprediction

- Which ones above need to be fast?

How do the three state maintenance methods fare in terms of recovery latency?
  - Reorder buffer
  - History buffer
  - Future file
Branch State Recovery Actions and Latency

- **Reorder Buffer**
  - Wait until branch is the oldest instruction in the machine
  - Flush entire pipeline

- **History buffer**
  - Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file
  - Flush instructions in pipeline younger than the branch

- **Future file**
  - Wait until branch is the oldest instruction in the machine
  - Copy arch. reg. file to future file
  - Flush entire pipeline
Can We Do Better?

- **Goal:** Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved.

- **Idea:** Checkpoint the frontend register state at the time a branch is fetched and keep the checkpointed state updated with results of instructions older than the branch.

Checkpointing

- When a branch is decoded
  - Make a copy of the future file and associate it with the branch

- When an instruction produces a register value
  - All future file checkpoints that are younger than the instruction are updated with the value

- When a branch misprediction is detected
  - Restore the checkpointed future file for the mispredicted branch when the branch misprediction is resolved
  - Flush instructions in pipeline younger than the branch
  - Deallocate checkpoints younger than the branch
Checkpointing

- Advantages?

- Disadvantages?
Registers versus Memory

- So far, we considered mainly registers as part of state

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Maintaining Speculative Memory State: 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
Out-of-Order Execution
(Dynamic Instruction Scheduling)
An 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)?

```
IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5
```

```
LD R3 ← R1 (0)
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5
```

- **Answer:** First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later *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 (scheduling, or execution)
  - Solution: out-of-order dispatch (scheduling, or 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 + precise exceptions:**

  ![In-order Dispatch Diagram]

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

- **Out-of-order dispatch + precise exceptions:**

  ![Out-of-order Dispatch Diagram]

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

- **16 vs. 12 cycles**
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 to execute
   - 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, need to dispatch the instruction to its functional unit (FU)
   - Instruction wakes up if all sources are ready
   - If multiple instructions are awake, need to select one per FU
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
- Initially in Intel Pentium Pro, AMD K5
- Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15
Two Humps in a Modern Pipeline

- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)
General Organization of an OOO Processor

Tomasulo’s Machine: IBM 360/91
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></th>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R9</td>
<td></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

\[
\begin{align*}
&\text{mul} \ R1, R2 \rightarrow R3 \\
&\text{add} \ R3, R4 \rightarrow R5 \\
&\text{add} \ R2, R6 \rightarrow R7 \\
&\text{add} \ R8, R9 \rightarrow R10 \\
&\text{mul} \ R7, R10 \rightarrow R11 \\
&\text{add} \ R5, R11 \rightarrow R5
\end{align*}
\]

- \text{mul} \ takes \ 6 \ cycles
- \text{add} \ takes \ 4 \ cycles

How many cycles total with data forwarding?
" ?"
Exercise Continued

FD123456W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W

↑
Execution timeline w/ scoreboard

31 cycles

FD123456W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W
FD1234W

25 cycles
Exercise Continued

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

Tomasulo's algorithm + full forwarding

20 cycles
How It Works

Register Alias Table

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of writer

Reservation Stacks for ADDER

SRC1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SRC2

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume adder & multiplier have separate buses

Value tag

Value tag
Cycle 0

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R11</td>
<td>1</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

- Initial contents of the register cache table
- Record store status are all invalid
Cycle 2:

- MUL R1, R2 → R3 reads its sources from the RAT.
- It writes to its destination in the RAT (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:

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

- MUL at X starts execution
- ADD R3, R4 → RS gate renamed and placed into the ADDER register stations

End of cycle 3:

- 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 X1)

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 registers enabled after the value becomes ready.

These two are completely orthogonal.
Cycle 4

**cycle 4:**  
- ADD R2, R6 → R7 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
- It is executed before the add in A

![Diagram](image-url)
## Cycle 7

### Table:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>~ 1</td>
</tr>
<tr>
<td>R2</td>
<td>~ 2</td>
</tr>
<tr>
<td>R3</td>
<td>X ~</td>
</tr>
<tr>
<td>R4</td>
<td>~ 4</td>
</tr>
<tr>
<td>R5</td>
<td>d ~</td>
</tr>
<tr>
<td>R6</td>
<td>~ 6</td>
</tr>
<tr>
<td>R7</td>
<td>b ~</td>
</tr>
<tr>
<td>R8</td>
<td>~ 8</td>
</tr>
<tr>
<td>R9</td>
<td>~ 9</td>
</tr>
<tr>
<td>R10</td>
<td>c ~</td>
</tr>
<tr>
<td>R11</td>
<td>y ~</td>
</tr>
</tbody>
</table>

### Diagram:

- **Graph A**: 3 nodes connected as follows: A → B → C
- **Graph B**: 4 nodes connected as follows: 1 ~ 1 ~ 2 ~ 2 ~ 3 ~ 9 ~ 9 ~ 2
- **Graph C**: nodes connected as follows: 0 b ~ 0 c ~

---

* 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 bits accordingly

  → (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 bits 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)
Out-of-Order Execution with Precise Exceptions

- **Idea:** Use a reorder buffer to reorder instructions before committing them to architectural state.

- An instruction updates the register alias table (essentially a future file) when it completes execution.

- An instruction updates the architectural register file when it is the oldest in the machine and has completed execution.
Out-of-Order Execution with Precise Exceptions

- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)
Enabling OoO Execution, Revisited

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

2. Buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming

3. Keep track of readiness of source values of an instruction
   - **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)
   - **Wakeup and select/schedule** the instruction
Summary of OOO Execution Concepts

- Register renaming eliminates false dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch
An out-of-order engine dynamically builds the dataflow graph of a piece of the program
- which piece?

The dataflow graph is limited to the instruction window
- Instruction window: all decoded but not yet retired instructions

Can we do it for the whole program?

Why would we like to?

In other words, how can we have a large instruction window?

Can we do it efficiently with Tomasulo’s algorithm?
Dataflow Graph for Our Example

\[
\begin{align*}
\text{MUL} & \quad R3 \leftarrow R1, R2 \\
\text{ADD} & \quad R5 \leftarrow R3, R4 \\
\text{ADD} & \quad R7 \leftarrow R2, R6 \\
\text{ADD} & \quad R10 \leftarrow R8, R9 \\
\text{MUL} & \quad R11 \leftarrow R7, R10 \\
\text{ADD} & \quad R5 \leftarrow R5, R11
\end{align*}
\]
State of RAT and RS in Cycle 7

end of cycle 7:

<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>R5</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>~6</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td>~8</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td>~9</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>~1</td>
<td>~1</td>
<td>~4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>~2</td>
<td>1</td>
<td>~6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>~8</td>
<td>1</td>
<td>~9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>~0</td>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x1 ~1 1 1 ~2
0 b ~0 0 c ~

* All 6 resistors removed.
- Note what happened to R5
Dataflow Graph

MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)

Nodes: operations performed by the instruction
Arcs: tags in Tomasulo’s algorithm
Restricted Data Flow

- An out-of-order machine is a “restricted data flow” machine
  - Dataflow-based execution is restricted to the microarchitecture level
  - ISA is still based on von Neumann model (sequential execution)

- Remember the data flow model (at the ISA level):
  - Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
Questions to Ponder

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - **Latency tolerance**: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently.

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - **What limits the latency tolerance scalability of Tomasulo’s algorithm?**
    - **Active/instruction window size**: determined by register file, scheduling window, reorder buffer.
Registers versus Memory, Revisited

- So far, we considered register based value communication between instructions

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores need to be handled after their execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine
When do you schedule a load instruction in an OOO engine?

- Problem: A younger load can have its address ready before an older store’s address is known
- Known as the memory disambiguation problem or the unknown address problem

Approaches

- Conservative: Stall the load until all previous stores have computed their addresses (or even retired from the machine)
- Aggressive: Assume load is independent of unknown-address stores and schedule the load right away
- Intelligent: Predict (with a more sophisticated predictor) if the load is dependent on the/any unknown address store