Review: Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result

3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

   • Normal data bus: data + destination (“go to” bus)
   • Common data bus: data + source (“come from” bus)
     - 64 bits of data + 4 bits of Functional Unit source address
     - Write if matches expected Functional Unit (produces result)
     - Does the broadcast

Review: Renaming

• Purpose of Renaming: removing “Anti-dependencies”
  - Get rid of WAR and WAW hazards, since these are not “real” dependencies

• Implicit Renaming: i.e. Tomasulo
  - Registers changed into values or response tags
  - We call this “implicit” because space in register file may or may not be used by results!

• Explicit Renaming: more physical registers than needed by ISA.
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

• With Explicit Renaming:
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    » Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler
How important is renaming?
Consider execution without it

1. LD F2, 34(R2)
lateness 1
2. LD F4, 45(R3)
long
3. MULTD F6, F4, F2
3
4. SUBD F8, F2, F2
1
5. DIVD F4, F2, F8
4
6. ADDD F10, F6, F4
1

In-order: 1 (2,1) . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . 2 3 . . 3 5 . . . 5 6 6

Out-of-order execution did not allow any significant improvement!

---

How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Which features of a program limit the number of instructions in the pipeline?

Control transfers

Out-of-order dispatch by itself does not provide any significant performance improvement!

---

Little’s Law

Throughput \( T \) = Number in Flight \( N \) / Latency \( L \)

Example:

- 4 floating point registers
- 8 cycles per floating point operation

⇒ maximum of ½ issue per cycle without renaming!

---

Instruction-level Parallelism via Renaming

1. LD F2, 34(R2)
lateness 1
2. LD F4, 45(R3)
long
3. MULTD F6, F4, F2
3
4. SUBD F8, F2, F2
1
5. DIVD F4', F2, F8
4
6. ADDD F10, F6, F4'
1

In-order: 1 (2,1) . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

Any antidependence can be eliminated by renaming.
(relabeling ⇒ additional storage)

Can be done either in Software or Hardware.
**Review: Loop Example Cycle 9**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
<th>Busy</th>
<th>Addr</th>
<th>Fu</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0</td>
<td>0</td>
<td>R1</td>
<td>Yes</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>F4</td>
<td>F0</td>
<td>F2</td>
<td>Yes</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>F4</td>
<td>0</td>
<td>R1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>F4</td>
<td>F0</td>
<td>F2</td>
<td>Yes</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>F4</td>
<td>0</td>
<td>R1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:**

- S1, S2, S3

**Register result status**

<table>
<thead>
<tr>
<th>Clock</th>
<th>R1</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Dataflow graph constructed completely in hardware
- Renaming detaches early iterations from registers

---

**Data-Flow Architectures**

- Basic Idea: Hardware represents direct encoding of compiler dataflow graphs:

  
  Input: a, b  
  y := (a+b)/x  
  x := (a*(a+b)) + b  

  output: y, x

- Data flows along arcs in “Tokens”.
- When two tokens arrive at compute box, box “fires” and produces new token.
- Split operations produce copies of tokens

---

**Paper by Dennis and Misunas**

- Instruction Cell
- Operation Unit 0
- Operation Unit m-1
- Instruction Cell 0
- Instruction Cell 1
- Instruction Cell n-1
- Memory
- Data Packets
- Operand 1
- Operand 2

**Administrative**

- No class on Monday (President’s Day)
- Midterm I: Wednesday 3/18  
  Location: 310 Soda Hall  
  TIME: 6:00—9:00  
  – Can have 1 sheet of 8½x11 handwritten notes – both sides  
  – No microfiche of the book!
- This info is on the Lecture page (has been)
- Meet at LaVal’s afterwards for Pizza and Beverages  
  – Great way for me to get to know you better  
  – I’ll Buy!
Problem: “Fetch” unit

Stream of Instructions To Execute

Instruction Fetch with Branch Prediction

Out-Of-Order Execution Unit

Correctness Feedback On Branch Results

- Instruction fetch decoupled from execution
- Often issue logic (+ rename) included with Fetch

Branches must be resolved quickly for loop overlap!

- In our loop-unrolling example, we relied on the fact that branches were under control of “fast” integer unit in order to get overlap!

Loop:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0</td>
<td>0</td>
</tr>
<tr>
<td>MULTD</td>
<td>F4</td>
<td>F0</td>
</tr>
<tr>
<td>SD</td>
<td>F4</td>
<td>0</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1</td>
<td>Loop</td>
</tr>
</tbody>
</table>

- What happens if branch depends on result of multd??
  - We completely lose all of our advantages!
  - Need to be able to “predict” branch outcome.
  - If we were to predict that branch was taken, this would be right most of the time.
- Problem much worse for superscalar machines!

Prediction: Branches, Dependencies, Data

- Prediction has become essential to getting good performance from scalar instruction streams.
- Next time, we will discuss predicting branches
- However, architects are now predicting everything: data dependencies, actual data, and results of groups of instructions:
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty close with control hazards already...
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
- Prediction ⇒ Compressible information streams?
- Next few sets of readings are about prediction!

What about Precise Exceptions/Interrupts?

- Both Scoreboard and Tomasulo have:
  - In-order issue, out-of-order execution, out-of-order completion
- Recall: An interrupt or exception is precise if there is a single instruction for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- Need way to resynchronize execution with instruction stream (i.e. with issue-order)
  - Easiest way is with in-order completion (i.e. reorder buffer)
  - Other Techniques (Smith paper): Future File, History Buffer
### Exception Handling (In-Order Five-Stage Pipeline)

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

### Phases of Instruction Execution

- **Fetch**: Instruction bits retrieved from cache.
- **Decode**: Instructions placed in appropriate issue (aka "dispatch") stage buffer
- **Execute**: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
- **Commit**: Instruction irrevocably updates architectural state (aka "graduation" or "completion").

### Complex In-Order Pipeline: Precise Exceptions

- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Instructions commit in order, simplifies precise exception implementation
- How to prevent increase latency for single-cycle ops?
  - Bypassing
  - However: can be very expensive
- Other downside: no out-of-order execution

### In-Order Commit for Precise Exceptions

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- **Commit** (write-back to architectural state, i.e., regfile & memory) is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)
**Reorder Buffer**

- **Idea:**
  - Record instruction issue order
  - Allow them to execute out of order
  - Reorder them so that they commit in-order
- **On issue:**
  - Reserve slot at tail of ROB
  - Record dest reg, PC
  - Tag u-op with ROB slot
- **Done execute**
  - Deposit result in ROB slot
  - Mark exception state
- **WB head of ROB**
  - Check exception, handle
  - Write register value, or
  - Commit the store

**Reorder Buffer + Forwarding**

- **Idea:**
  - Forward uncommitted results to later uncommitted operations
- **Trap**
  - Discard remainder of ROB
- **Opfetch / Exec**
  - Match source reg against all dest regs in ROB
  - Forward last (once available)

**Reorder Buffer + Forwarding + Speculation**

- **Idea:**
  - Issue branch into ROB
  - Mark with prediction
  - Fetch and issue predicted instructions speculatively
  - Branch must resolve before leaving ROB
  - Resolve correct
    - Commit following instr
  - Resolve incorrect
    - Mark following instr in ROB as invalid
    - Let them clear

**History File: a hardware undo log**

- **Maintain issue order, like ROB**
- **Each entry records dest reg and old value of dest. Register**
  - What if old value not available when instruction issues?
- **FUs write results into register file**
  - Forward into correct entry in history file
- **When exception reaches head**
  - Restore architected registers from tail to head
Future file

- Idea
  - Arch registers reflect state at commit point
  - Future register reflect whatever instructions have completed
  - On WB update future
  - On commit update arch
  - On exception
    - Discard future
    - Replace with arch

Dest w/I ROB

IFetch
Opfetch/Dcd
Future
Reg
Write Back

What are the hardware complexities with reorder buffer (ROB)?

- How do you find the latest version of a register?
  - As specified by Smith paper, need associative comparison network
  - Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value
  - Need as many ports on ROB as register file

Four Steps of Speculative Tomasulo

1. Issue—get instruction from FP Op Queue
   - If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)
2. Execution—operate on operands (EX)
   - When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)
3. Write result—finish execution (WB)
   - Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit—update register with reorder result
   - When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)

Tomasulo With Reorder buffer:

FP Op Queue
Reorder Buffer
Registers

Done?

FP adders
FP multipliers
Reservation Stations
To Memory
from Memory

Newest
Oldest

ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

LD F0,10(R2)
Tomasulo With Reorder buffer:

Reorder Buffer

FP Op Queue

-- M[10] ST 0(R3), F4 Y
F0 ADDD F0, F4, F6 N
F4 M[10] LD F4, 0(R3) Y
-- BNE F2, <, > N
F2 DIVD F2, F10, F6 N
F10 ADDD F10, F4, F0 N
F0 LD F0, 10(R2) N

Done?

Registers

FP adders

FP multipliers

To Memory

FP adders

FP multipliers

Reservation Stations

0

1 10+R2

Dest

Dest

from Memory

Dest

Dest

Dest

Dest

Done?

Tomasulo With Reorder buffer:

Reorder Buffer

FP Op Queue

-- M[10] ST 0(R3), F4 Y
F0 ADDD F0, F4, F6 N
F4 M[10] LD F4, 0(R3) Y
Tomasulo With Reorder buffer:

Reorder Buffer

FP Op Queue

-- M[10] ST 0(R3), F4 Y
F0 ADDD F0, F4, F6 N
F4 M[10] LD F4, 0(R3) Y
-- BNE F2, <, > N
F2 DIVD F2, F10, F6 N
F10 ADDD F10, F4, F0 N
F0 LD F0, 10(R2) N

Done?

Registers

FP adders

FP multipliers

To Memory

FP adders

FP multipliers

Reservation Stations

0

1 10+R2

Dest

Dest

from Memory

Dest

Dest

Dest

Dest

Done?

Tomasulo With Reorder buffer:

Reorder Buffer

FP Op Queue

-- M[10] ST 0(R3), F4 Y
F0 ADDD F0, F4, F6 N
F4 M[10] LD F4, 0(R3) Y
-- BNE F2, <, > N
F2 DIVD F2, F10, F6 N
F10 ADDD F10, F4, F0 N
F0 LD F0, 10(R2) N

Done?

Registers

FP adders

FP multipliers

To Memory

FP adders

FP multipliers

Reservation Stations

0

1 10+R2

Dest

Dest

from Memory

Dest

Dest

Dest

Dest

Done?

What about memory hazards???

Memory Disambiguation: Sorting out RAW Hazards in memory

• Question: Given a load that follows a store in program order, are the two related?
  – (Alternatively: is there a RAW hazard between the store and the load)?
  Eg: st 0(R2), R5
  ld R6, 0(R3)

• Can we go ahead and start the load early?
  – Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  – We might want to issue/begin execution of both operations in same cycle.
  – Today: Answer is that we are not allowed to start load until we know that address 0(R2) ≠ 0(R3)
  – Next Week: We might guess at whether or not they are dependent (called “dependence speculation”) and use reorder buffer to fixup if we are wrong.
Hardware Support for Memory Disambiguation

• Need buffer to keep track of all outstanding stores to memory, in program order.
  – Keep track of address (when becomes available) and value (when becomes available)
  – FIFO ordering: will retire stores from this buffer in program order

• When issuing a load, record current head of store queue (know which stores are ahead of you).

• When have address for load, check store queue:
  – If any store prior to load is waiting for its address, stall load.
  – If load address matches earlier store address (associative lookup), then we have a memory-induced RAW hazard:
    » store value available ⇒ return value
    » store value not available ⇒ return ROB number of source
  – Otherwise, send out request to memory

• Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Relationship between precise interrupts and speculation:

• Speculation is a form of guessing
  – Branch prediction, data prediction
  – If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  – This is exactly same as precise exceptions!

• Branch prediction is a very important!
  – Need to “take our best shot” at predicting branch direction.
  – If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    » Consider 4 instructions per cycle
    » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!

• Technique for both precise interrupts/exceptions and speculation: in-order completion or commit
  – This is why reorder buffers in all new processors

Quick Recap: Explicit Register Renaming

• Make use of a physical register file that is larger than number of registers specified by ISA

• Keep a translation table:
  – ISA register => physical register mapping
  – When register is written, replace table entry with new register from freelist.
  – Physical register becomes free when not being used by any instructions in progress.
Explicit register renaming: R10000 Freelist Management

- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000

Current Map Table

Free list

• Physical register file larger than ISA register file
• On issue, each instruction that modifies a register is allocated new physical register from freelist
• Used on: R10000, Alpha 21264, HP PA8000

Current Map Table

Free list

- Note that physical register P0 is “dead” (or not “live”) past the point of this load.
  - When we go to commit the load, we free up

Checkpoint at BNE instruction
Explicit register renaming:
R10000 Freelist Management

Advantages of Explicit Renaming
- **Decouples** renaming **from scheduling**:
  - Pipeline can be exactly like “standard” DLX pipeline (perhaps with multiple operations issued per cycle)
  - Or, pipeline could be tomasulo-like or a scoreboard, etc.
  - Standard forwarding or bypassing could be used
- Allows data to be fetched from single register file
  - No need to bypass values from reorder buffer
  - This can be important for balancing pipeline
- Many processors use a variant of this technique:
  - R10000, Alpha 21264, HP PA8000
- Another way to get precise interrupt points:
  - All that needs to be “undone” for precise break point is to undo the table mappings
  - Provides an interesting mix between reorder buffer and future file
    » Results are written immediately back to register file
    » Registers names are “freed” in program order (by ROB)

Dynamic Scheduling for Out-Of-Order Superscalar
- How to issue two instructions and keep in-order instruction issue for Tomasulo? 
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point
- Issue 2X Clock Rate, so that issue remains in order
- Only FP loads might cause dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load Queue to avoid WAR, WAW
  - Called “**decoupled architecture**”
Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Multiported rename logic: must be able to rename same register multiple times in one cycle!
  - Rename logic one of key complexities in the way of multiple issue!
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
  - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)

Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- HW determines address conflicts
- HW better branch prediction
- HW maintains precise exception model
- HW does not execute bookkeeping instructions
- Works across multiple implementations
- SW speculation is much easier for HW design

Superscalar v. VLIW

- Smaller code size
- Binary compatibility across generations of hardware
  - Simplified Hardware for decoding, issuing instructions
  - No Interlock Hardware (compiler checks?)
  - More registers, but simplified Hardware for Register Ports (multiple independent register files?)
Summary

- **DataFlow view:**
  - Data triggers execution rather than instructions triggering data

- **Dynamic hardware schemes can unroll loops dynamically in hardware**
  - Form of limited dataflow
  - Register renaming is essential

- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

- **Precise Interrupts:**
  - Must commit things back in order
  - Reorder buffer: temporarily holds results until commit possible
  - Toss out things to achieve precise interrupt point