CS152 Section 6

Out-of-Order Execution
Week of March 4, 2024

Agenda

- Out-of-order execution

Admin

- Hope the midterm went well!
- Lab 2 due March 8th
- PS3 due March 18th
- Lab 3 is due March 20th

Why is Out-of-Order Execution Useful?

- Exploit *instruction-level parallelism* (ILP) to keep processor busy
  - Make suboptimal code run fast
- *Dynamically* schedule around long-latency instructions

```
ld x2, 0(x1)          # cache miss: 200 cycles
add x5, x3, x4
ld x7, 4(x6)
```

- Initiate long-latency instructions earlier

What Limits OoO Performance?

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```

- Want to issue instruction C right after A, but cannot reorder it earlier due to WAR hazard on B (`f3`)
- Suppose only four F registers exist, and it is not feasible for compiler to choose `f2` as the destination of C since `f2` is read by a later instruction

What Limits OoO Performance?

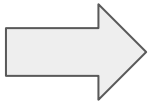
- **WAW/WAR** hazards
  - Caused by reuse of limited set of architectural (named) registers
  - Would not exist if an infinite number of registers were available
  - Not a “true” data dependency
- How can x86 (8 “GPRs”) and x86-64 (16 GPRs) implementations achieve high performance?
- How can we use more registers than what the ISA specifies?

Register Renaming

- Main idea: Decouple **architectural** registers (used for expressing dataflow) from **physical** registers (used for storage)
  - For each in-flight instruction, rename the destination register with a unique tag that refers to a separate buffer to hold result
  - Somehow maintain relationship between tags and ISA registers

Register Renaming

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```



```
fmul P4, P0, P2
fadd P5, P3, P4
fmul P6, P2, P3
fadd P7, P6, P4
```

Rename Table		
	Initial	**Final**
f0	P0	P5
f1	P1	P4
f2	P2	P2
f3	P3	P7

Tomasulo's Algorithm

- On instruction **dispatch** (in program order):
  1. Allocate reservation station (RS) entry
  2. If source register has "present" (P) bit set in register file (RF) entry, copy value into tag/data field in RS and set P bit for operand
  3. Otherwise, copy tag from RF into RS and clear P bit for operand
  4. Replace RF entry for destination register with tag assigned to RS entry (tag_{dest})
- Prior to **execution**:
  1. For missing operands, monitor result bus for tag match; replace tag with value; set P
  2. When all operands are present, issue to functional unit
- On **completion**:
  1. Broadcast <tag_{dest}, result> on result bus for RF and other RS entries to consume
  2. Deallocate RS entry

Q1: Tomasulo's Algorithm

Simulate execution of the following code using Tomasulo's Algorithm:

```
A: fmul f1, f0, f2    # produces value V3
B: fadd f0, f3, f1    # produces value V4
C: fmul f3, f2, f3    # produces value V5
D: fadd f3, f3, f1    # produces value V6
```

- At most one instruction dispatched per cycle
  - Can begin execution the same cycle as dispatch if all operands are present
- Fully-pipelined functional units
  - 2-cycle floating-point add latency
  - 3-cycle floating-point multiply latency
- Broadcasting result takes another cycle
  - Result bus can broadcast two results simultaneously

Cycle 1

Dispatched instruction: A

On instruction **dispatch** (in program order):

1. Allocate reservation station (RS) entry
2. If source register has “present” (P) bit set in register file (RF) entry, copy value into tag/data field in RS and set P bit for operand
3. Otherwise, copy tag from RF into RS and clear P bit for operand
4. Replace RF entry for destination register with tag assigned to RS entry (tag_{dest})

Reservation Stations

	p	tag/data	p	tag/data
T0				
T1				
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3	1	V0	1	V1
T4				

Register File

	p	tag/data
f0	1	V0
f1	0	T3
f2	1	V1
f3	1	V2

Adder

stage	dest tag
1	
2	

Multiplier

stage	dest tag
1	T3
2	
3	

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Cycle 2

Dispatched instruction: B

On instruction **dispatch** (in program order):

1. Allocate reservation station (RS) entry
2. If source register has “present” (P) bit set in register file (RF) entry, copy value into tag/data field in RS and set P bit for operand
3. Otherwise, copy tag from RF into RS and clear P bit for operand
4. Replace RF entry for destination register with tag assigned to RS entry (tag_{dest})

Reservation Stations

	p	tag/data	p	tag/data
T0	1	V2	0	T3
T1				
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3	1	V0	1	V1
T4				

Register File

	p	tag/data
f0	0	T0
f1	0	T3
f2	1	V1
f3	1	V2

Adder

stage	dest tag
1	
2	

Multiplier

stage	dest tag
1	
2	T3
3	

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Cycle 3

Dispatched instruction: C

On instruction **dispatch** (in program order):

1. Allocate reservation station (RS) entry
2. If source register has “present” (P) bit set in register file (RF) entry, copy value into tag/data field in RS and set P bit for operand
3. Otherwise, copy tag from RF into RS and clear P bit for operand
4. Replace RF entry for destination register with tag assigned to RS entry (tag_{dest})

Reservation Stations

	p	tag/data	p	tag/data
T0	1	V2	0	T3
T1				
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3	1	V0	1	V1
T4	1	V1	1	V2

Register File

	p	tag/data
f0	0	T0
f1	0	T3
f2	1	V1
f3	0	T4

Adder

stage	dest tag
1	
2	

Multiplier

stage	dest tag
1	T4
2	
3	T3

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Cycle 4

Dispatched instruction: D

On **completion**:

1. Broadcast <tag_{dest}, result> on result bus for RF and other RS entries to consume
2. Deallocate RS entry
3. For missing operands, monitor result bus for tag match; replace tag with value; set P

Reservation Stations

	p	tag/data	p	tag/data
T0	1	V2	1	V3
T1	0	T4	1	V3
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3				
T4	1	V1	1	V2

Register File

	p	tag/data
f0	0	T0
f1	1	V3
f2	1	V1
f3	0	T1

Adder

stage	dest tag
1	T0
2	

Multiplier

stage	dest tag
1	
2	T4
3	

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Cycle 5

Dispatched instruction: N/A

On **completion**:

1. Broadcast <tag_{dest}, result> on result bus for RF and other RS entries to consume
2. Deallocate RS entry
3. For missing operands, monitor result bus for tag match; replace tag with value; set P

Reservation Stations

	p	tag/data	p	tag/data
T0	1	V2	1	V3
T1	0	T4	1	V3
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3				
T4	1	V1	1	V2

Register File

	p	tag/data
f0	0	T0
f1	1	V3
f2	1	V1
f3	0	T1

Adder

stage	dest tag
1	
2	T0

Multiplier

stage	dest tag
1	
2	
3	T4

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Cycle 6

Dispatched instruction: N/A

On **completion**:

1. Broadcast <tag_{dest}, result> on result bus for RF and other RS entries to consume
2. Deallocate RS entry
3. For missing operands, monitor result bus for tag match; replace tag with value; set P

Reservation Stations

	p	tag/data	p	tag/data
T0				
T1	1	V5	1	V3
T2				

Reservation Stations

	p	tag/data	p	tag/data
T3				
T4				

Register File

	p	tag/data
f0	1	V4
f1	1	V3
f2	1	V1
f3	0	T1

Adder

stage	dest tag
1	T1
2	

Multiplier

stage	dest tag
1	
2	
3	

```
A: fmul f1, f0, f2    # V3
B: fadd f0, f3, f1    # V4
C: fmul f3, f2, f3    # V5
D: fadd f3, f3, f1    # V6
```

Tomasulo’s Algorithm

Q: Why can’t the reservation station entry for an instruction be deallocated immediately on issue?

```
A: fmul f4, f0, f1     # Dispatched and issued immediately; RS is freed
B: fmul f5, f2, f3     # Allocated same RS as A before A has written back
```

`f4` and `f5` now assigned the same tag in regfile, causing instruction A to incorrectly clobber `f5` on writeback

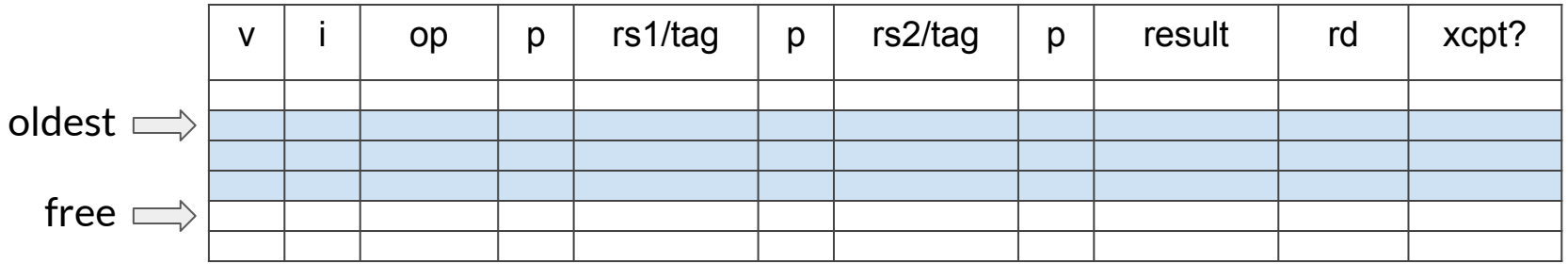
Tomasulo’s Algorithm

Q: Why are exceptions *imprecise* in this implementation?

- Register file is irrevocably modified on dispatch
- No mechanism to recover original value of destination register if instruction causes an exception

How to Regain Precise Exceptions?

Reorder Buffer (ROB) separates *commit* from *completion*:



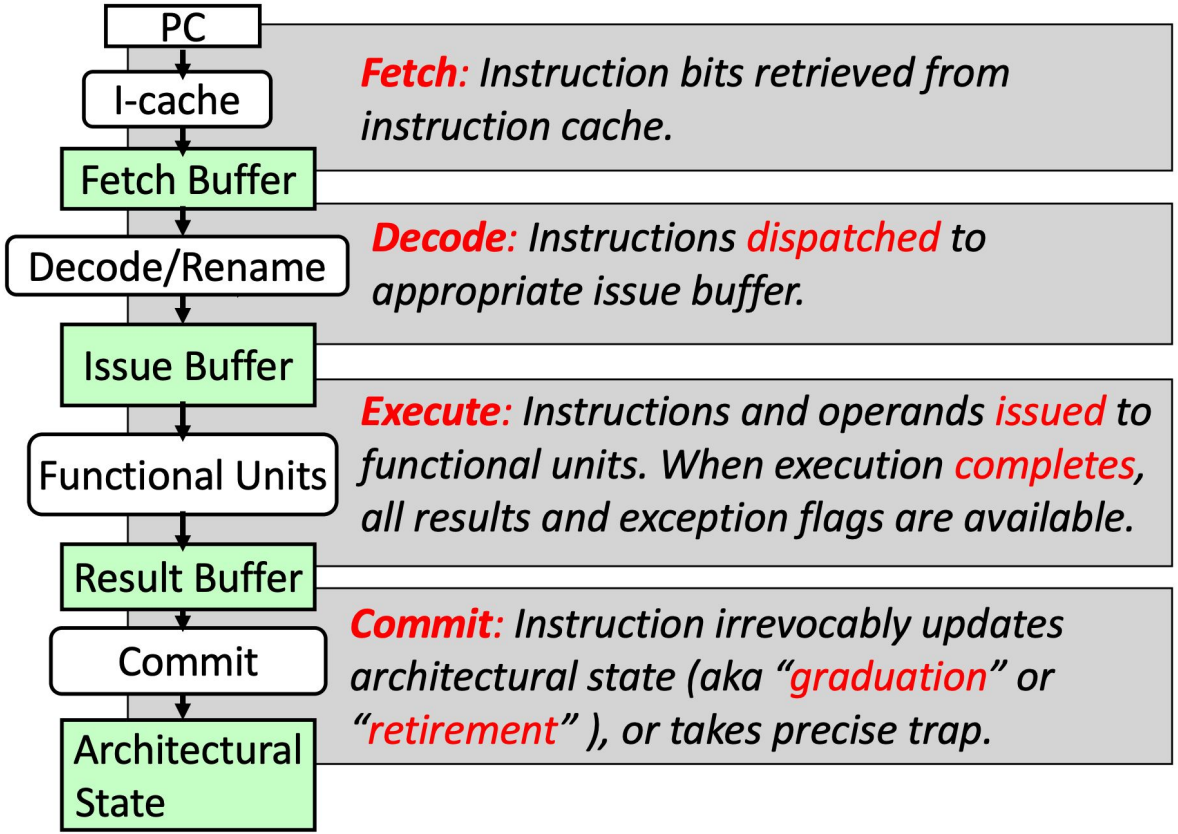
- *Completion*: Result available (out-of-order)
- *Commit*: Architectural state updated (in-order)

Terminology for OoOE

- **Dispatch vs. Issue**
  - **Dispatch:** instruction decoded and enters ROB/Reservation Station
  - **Issue:** instruction “issued” for execution (enters EX Unit)
  - Some papers and diagrams will use the term Issue for what we refer to as Dispatch

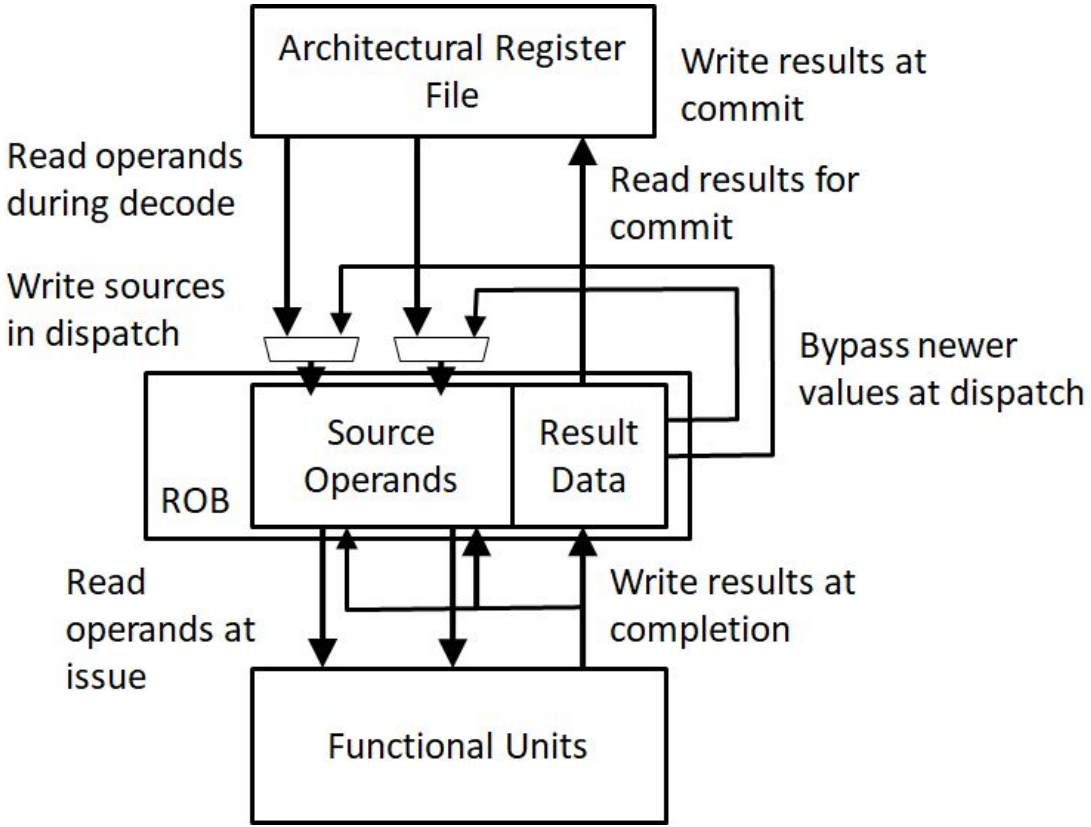
- **Completion vs. Commit**
  - Precise Exceptions (In-order commit despite OoO completion)
  - **“Retired”** also used to mean **“committed”**

Phases of Instruction Execution



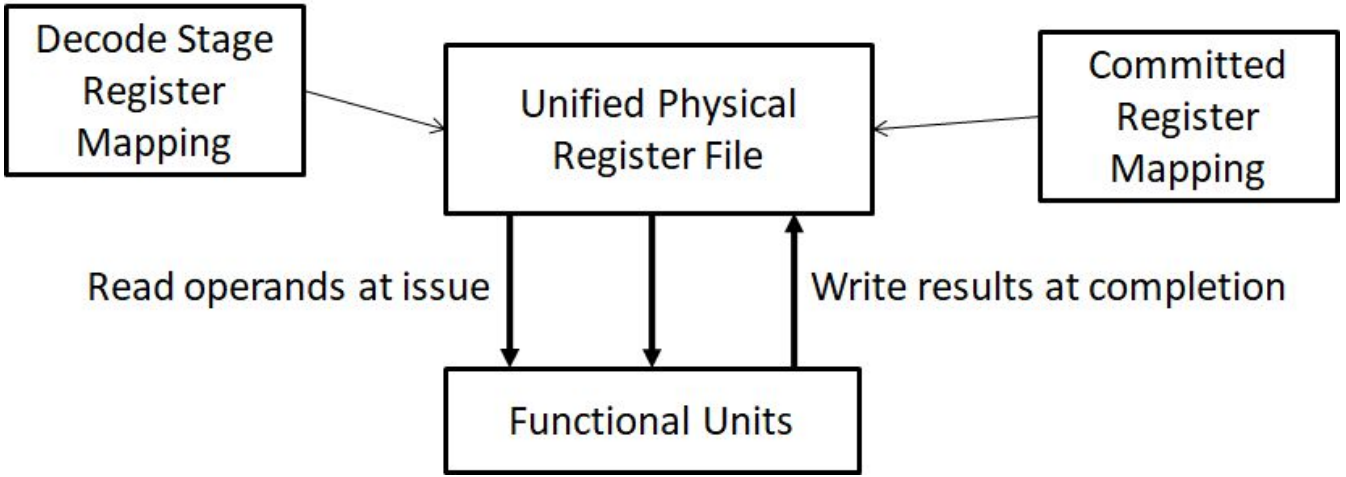
Data-in-ROB

- ROB holds both tags and data
- Separate architectural register file



Unified Physical Register File

- Physical register file holds both committed and temporary values
- Only tags held in ROB



Q2: Renaming with Unified PRF

- On **dispatch**:
  1. Allocate new physical register for destination from free list
  2. Update decode-stage mapping
- On **commit**:
  1. Update architectural mapping
  2. Deallocate previous physical register for destination; re-add to free list
- On **exception**:
  1. Repair decode-stage rename table by un-renaming in reverse order; walk through ROB entries from newest to oldest (MIPS R10k approach)

Renaming with Unified PRF (Q2)

Instruction	Architectural Destination Register	Physical Destination Register	Freed Register
`ld x2, 0(x4)`	x2	P2	P11
`sw x2, 0(x3)`	-	-	-
`addi x4, x4, 0x4`	x4	P4	P6
`addi x3, x3, 0x4`	x3	P10	P5
`bne x4, x1, loop`	-	-	-
`ld x2, 0(x4)`	x2	p7	P2
`sw x2, 0(x3)`	-	-	-
`addi x4, x4, 0x4`	x4	p1	P4
`addi x3, x3, 0x4`	x3	p0	P10
`bne x4, x1, loop`	-	-	-

Initial Map Table

Architectural Register	Physical Register
x1	p12
x2	~~P11~~ P2 P7
x3	~~P5,~~ P10 P0
x4	~~P6,~~ P4 P1
x5	P3

Free List

~~p2~~
~~p4~~
~~p10~~
~~p7~~
~~p1~~
~~p0~~
p9
p8
p13
p14

Renaming with Unified PRF (Q2)

Instruction	Architectural Destination Register	Physical Destination Register	Freed Register
`ld x2, 0(x4)`	x2	p2	p11
`sw x2, 0(x3)`			
`addi x4, x4, 0x4`			
`addi x3, x3, 0x4`			
`bne x4, x1, loop`			
`ld x2, 0(x4)`			
`sw x2, 0(x3)`			
`addi x4, x4, 0x4`			
`addi x3, x3, 0x4`			
`bne x4, x1, loop`			

Initial Map Table

Architectural Register	Physical Register
x1	p12
x2	~~P11~~ P2
x3	P5
x4	P6
x5	P3

Free List

~~p2~~
p4
p10
p7
p1
p0
p9
p8
p13
p14

Renaming with Unified PRF (Q2)

Instruction	Architectural Destination Register	Physical Destination Register	Freed Register
`ld x2, 0(x4)`	x2	p2	p11
`sw x2, 0(x3)`	-	-	-
`addi x4, x4, 0x4`	x4	p4	p6
`addi x3, x3, 0x4`	x3	p10	p5
`bne x4, x1, loop`	-	-	-
`ld x2, 0(x4)`	x2	p7	p2
`sw x2, 0(x3)`	-	-	-
`addi x4, x4, 0x4`	x4	p1	p4
`addi x3, x3, 0x4`	x3	p0	p10
`bne x4, x1, loop`	-	-	-

Initial Map Table

Architectural Register	Physical Register
x1	p12
x2	~~P11~~, ~~P2~~, P7
x3	~~P5~~, ~~p10~~, p0
x4	~~P6~~, ~~p4~~, p1
x5	p3

Free List

~~p2~~
~~p4~~
~~p10~~
~~p7~~
~~p1~~
~~p0~~
p9
p8
p13
p14

Renaming with Unified PRF (Q2)

Suppose the second load instruction caused an exception. Show the state of the map table and free list before jumping to the exception handler.

Map Table

Architectural Register	Physical Register
x1	
x2	
x3	
x4	
x5	

Free List

Renaming with Unified PRF (Q2)

Instruction	Architectural Destination Register	Physical Destination Register	Freed Register
`ld x2, 0(x4)`	x2	p2	p11
`sw x2, 0(x3)`	-	-	-
`addi x4, x4, 0x4`	x4	p4	p6
`addi x3, x3, 0x4`	x3	p10	p5
`bne x4, x1, loop`	-	-	-
~~`ld x2, 0(x4)`~~	~~x2~~	~~p7~~	~~p2~~
~~`sw x2, 0(x3)`~~	-	-	-
~~`addi x4, x4, 0x4`~~	~~x4~~	~~p1~~	~~p4~~
~~`addi x3, x3, 0x4`~~	~~x3~~	~~p0~~	~~p10~~
~~`bne x4, x1, loop`~~	-	-	-

Initial Map Table

Architectural Register	Physical Register
x1	p12
x2	~~P11~~, ~~P2~~, ~~P7~~, P2
x3	~~P5~~, ~~p10~~, ~~p0~~, p10
x4	~~P6~~, ~~p4~~, ~~p1~~, p4
x5	p3

Free List

~~p2~~
~~p4~~
~~p10~~
~~p7~~
~~p1~~
~~p0~~
p9
p8
p13
p14