CS 6354: Homework 1 Related / Precise Interrupts

19 September 2016

To read more...

This day's paper:

Smith and Pleszkan, "Implementation of Precise Interrupts in Pipelined Processors"

Materials on virtual memory/demand paging (a reason we need precise interrupts):

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, sections B.5-B.6 (graduate text)

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, section 1.7, sections 9.1-7 (undergrad text)

Supplementary readings on precise interrupts:

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, section 3.6

Shin and Lipatsi, *Modern Processor Design*, section 4.3.5

Homework 1

data cache sizes

instruction cache sizes

TLB size

data cache associativity

TLB associativity

throughput (read/write; sequential/random)

latency (maximum)

prefetching?

two of many choices

On paper reviews

What was your most significant insight from the paper?

Example good answers:

> Implementing precise interrupts requires degrading processor performance
>
> To get good performance in a heavily pipelined processor, we need to allow instructions to use values from prior instructions that aren't finished yet.
>
> One can provide precise interrupts by allowing instructions to complete out-of-order, but maintaining a consistent state to restore if an interrupt occurs.

Not-so-good answers:

> There are several ways to implement precise interrupts.

Reasons for precise interrupts

virtual memory

debugging

multiprocessing (timer interrupts)

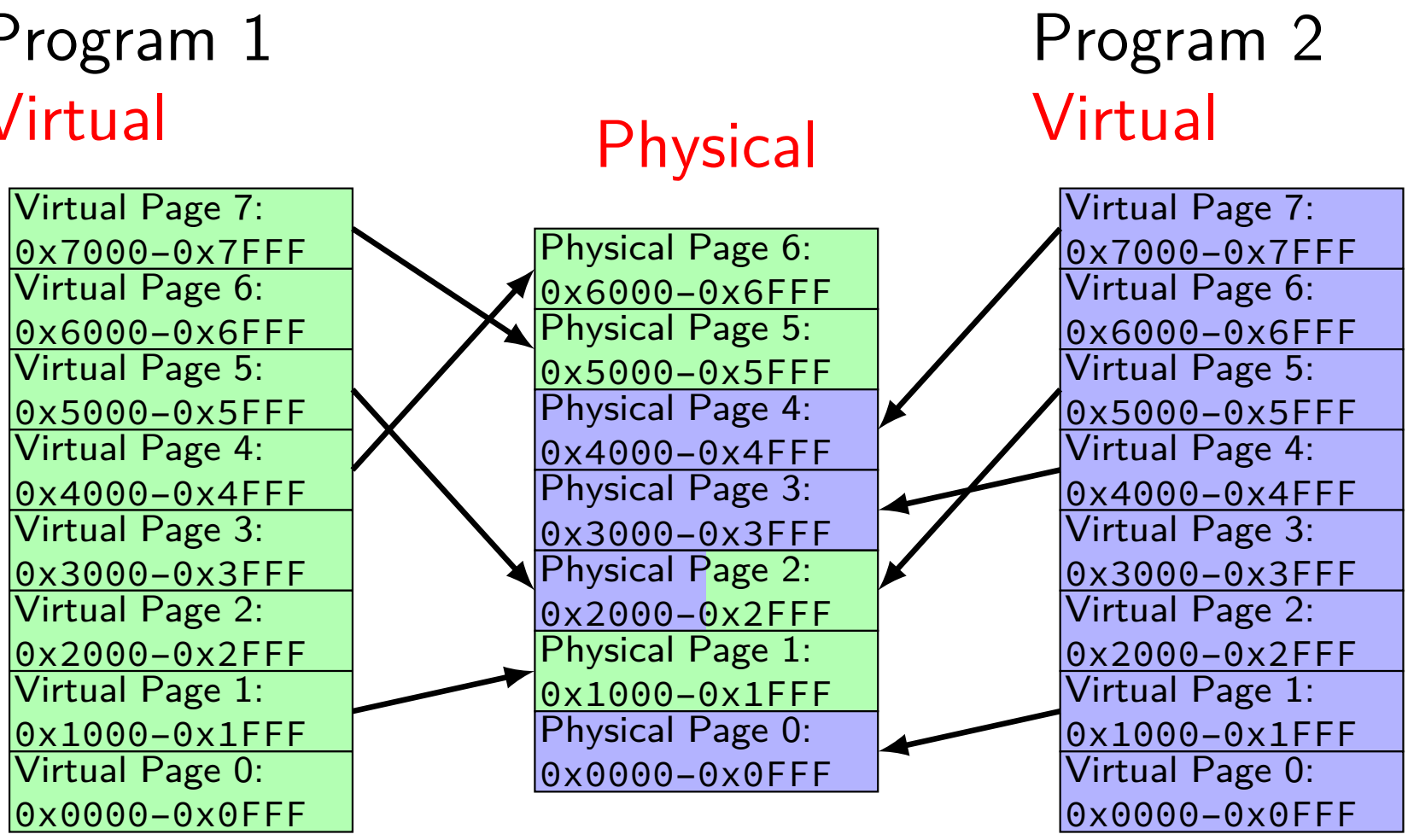
virtual machines — trap-and-emulate

- VirtualBox, VMWare, Xen, parallels...

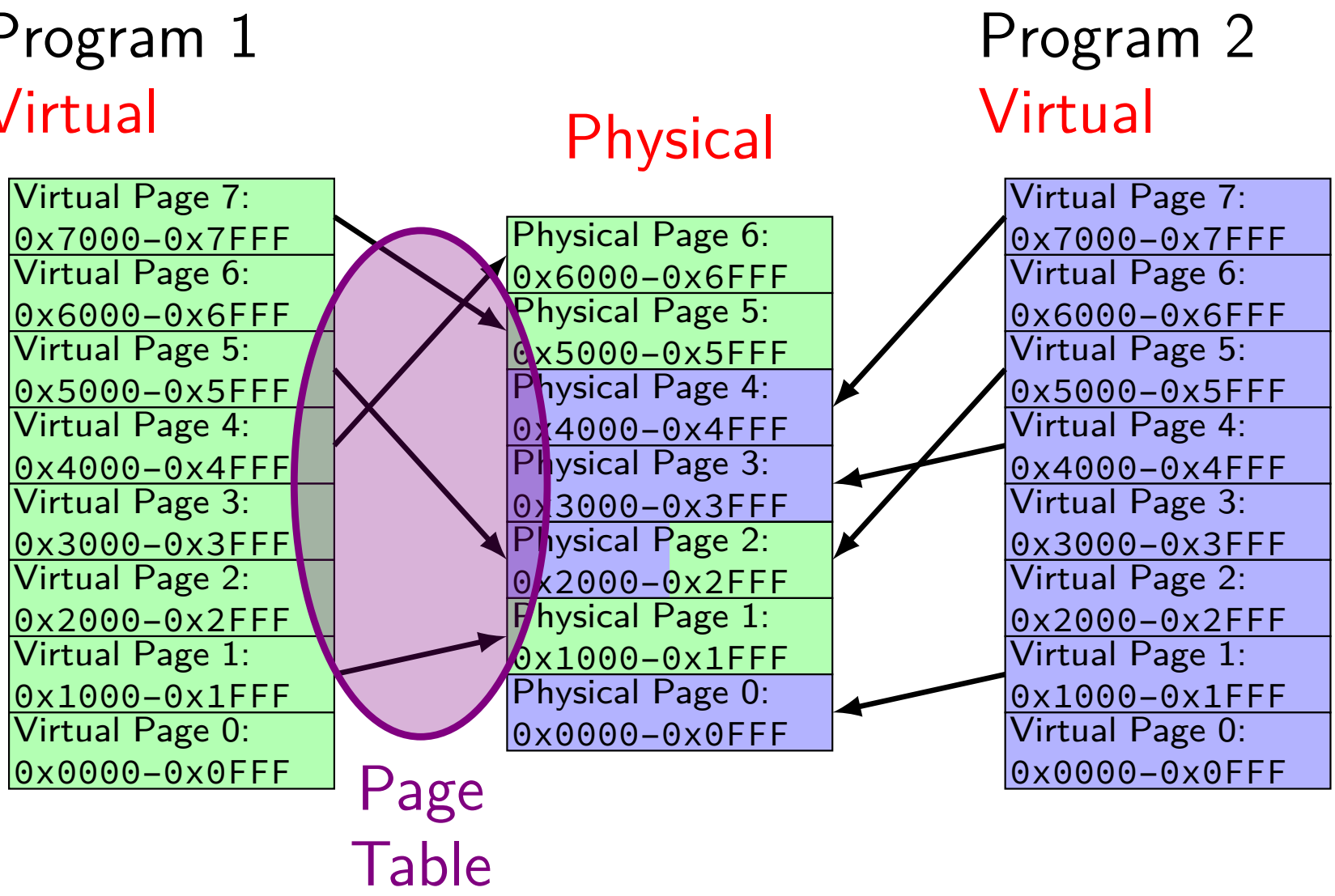
future hardware — trap-and-emulate

- example: floating point support

Virtual memory



Virtual memory

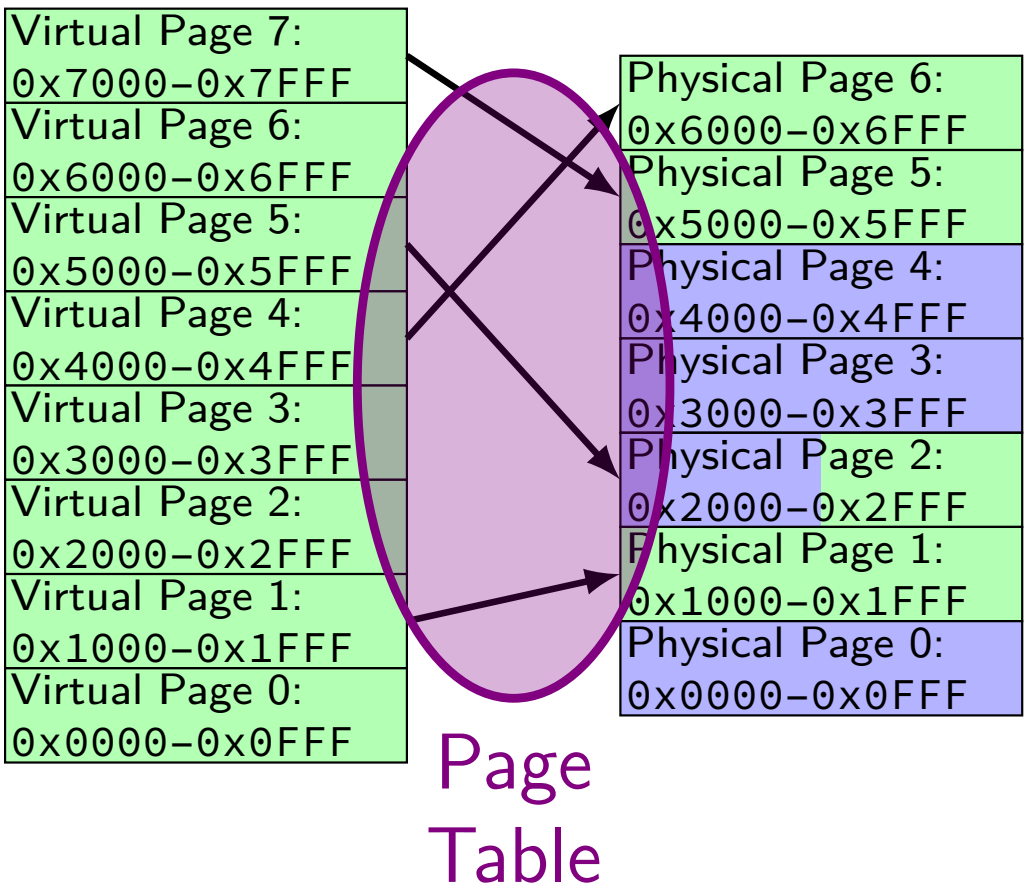


Virtual memory

Program 1

Virtual

Physical



virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

Page tables and the TLB (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB
(fast cache)

tag (virt.)	physical	last use
7	5	
1	1	most recent

TLB access pattern (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag (virt.)	physical	last use
7	5	most recent
1	1	
4	6	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	
load 0x5F05	virtual page 5, offset 0xF05	
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag (virt.)	physical	last use
7	5	
1	1	most recent
4	6	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag (virt.)	physical	last use
7	5	
1	1	
5 (was 4)	2 (was 6)	most recent

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 6)
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag (virt.)	physical	last use
7	5	most recent
1	1	
5	2	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 6)
load 0x7D57	virtual page 7, offset 0xD57	hit
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (1)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag (virt.)	physical	last use
7	5	
1	1	
5	2	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 6)
load 0x7D57	virtual page 7, offset 0xD57	hit
load 0x5143	virtual page 5, offset 0x143	hit

TLB access pattern (2)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag	physical	last use
7	5	most recent
1	1	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	
load 0x5F05	virtual page 5, offset 0xF05	
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (2)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag	physical	last use
7	5	
1	1	most recent

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (2)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag	physical	last use
5	2	most recent
1	1	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 7)
load 0x7D57	virtual page 7, offset 0xD57	
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (2)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag	physical	last use
5	2	
7	5	most recent

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 7)
load 0x7D57	virtual page 7, offset 0xD57	miss (replace 1)
load 0x5143	virtual page 5, offset 0x143	

TLB access pattern (2)

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	yes	2
4	yes	6
3	no	–
2	no	–
1	yes	1
0	no	–

fully-associative TLB (fast cache)

tag	physical	last use
5	2	most recent
7	5	

access pattern

memory access	split of address	result
load 0x7050	virtual page 7, offset 0x050	hit
load 0x1003	virtual page 1, offset 0x003	hit
load 0x5F05	virtual page 5, offset 0xF05	miss (replace 7)
load 0x7D57	virtual page 7, offset 0xD57	miss (replace 1)
load 0x5143	virtual page 5, offset 0x143	hit

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	no	–
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	no	–
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	`program.exe @ 0x4000`
2	yes	`program.exe @ 0x3000`
1	yes	`program.exe @ 0x2000`
0	no	–

access pattern

memory access	split of address	result
load `0x1004`	virtual page 1, offset `0x004`	
load `0x1008`	virtual page 1, offset `0x008`	
load `0x7FF4`	virtual page 7, offset `0xFF4`	
load `0x100C`	virtual page 1, offset `0x00C`	
load `0x7FF0`	virtual page 7, offset `0xEE0`	

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	no	–
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1 (just loaded)
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	`program.exe @ 0x4000`
2	yes	`program.exe @ 0x3000`
1	yes	`program.exe @ 0x2000`
0	no	–

access pattern

memory access	split of address	result
load `0x1004`	virtual page 1, offset `0x004`	not present, page fault
load `0x1008`	virtual page 1, offset `0x008`	
load `0x7FF4`	virtual page 7, offset `0xFF4`	
load `0x100C`	virtual page 1, offset `0x00C`	
load `0x7FF0`	virtual page 7, offset `0xEE0`	

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	yes	5 (just allocated)
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	`program.exe @ 0x4000`
2	yes	`program.exe @ 0x3000`
1	yes	`program.exe @ 0x2000`
0	no	–

access pattern

memory access	split of address	result
load `0x1004`	virtual page 1, offset `0x004`	not present, page fault
load `0x1008`	virtual page 1, offset `0x008`	present
load `0x7FF4`	virtual page 7, offset `0xFF4`	
load `0x100C`	virtual page 1, offset `0x00C`	
load `0x7FF0`	virtual page 7, offset `0xEE0`	

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	program.exe @ 0x4000
2	yes	program.exe @ 0x3000
1	yes	program.exe @ 0x2000
0	no	–

access pattern

memory access	split of address	result
load 0x1004	virtual page 1, offset 0x004	not present, page fault
load 0x1008	virtual page 1, offset 0x008	present
load 0x7FF4	virtual page 7, offset 0xFF4	not present, page fault
load 0x100C	virtual page 1, offset 0x00C	
load 0x7FF0	virtual page 7, offset 0xEE0	

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	program.exe @ 0x4000
2	yes	program.exe @ 0x3000
1	yes	program.exe @ 0x2000
0	no	–

access pattern

memory access	split of address	result
load 0x1004	virtual page 1, offset 0x004	not present, page fault
load 0x1008	virtual page 1, offset 0x008	present
load 0x7FF4	virtual page 7, offset 0xFF4	not present, page fault
load 0x100C	virtual page 1, offset 0x00C	present
load 0x7FF0	virtual page 7, offset 0xEE0	

Demand Paging

Page Table (slow memory)

virtual	present?	physical
7	yes	5
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

OS's information

virtual	valid?	location on disk
7	yes	(none, allocate 0s)
6	no	–
5	no	–
4	no	–
3	yes	program.exe @ 0x4000
2	yes	program.exe @ 0x3000
1	yes	program.exe @ 0x2000
0	no	–

access pattern

memory access	split of address	result
load 0x1004	virtual page 1, offset 0x004	not present, page fault
load 0x1008	virtual page 1, offset 0x008	present
load 0x7FF4	virtual page 7, offset 0xFF4	not present, page fault
load 0x100C	virtual page 1, offset 0x00C	present
load 0x7FF0	virtual page 7, offset 0xEE0	present

Caches on Caches

memory as software-managed cache for disk

- operating system makes all decisions

page faults are hardware support for the software

- way to ask operating system to make decision

illusion of much larger memory

(with worse performance — disk/SSD is slow)

The Page Fault Handler

operating system code

called when user code wants not present virtual page

operating system often restarts instruction

> restart $\approx$ reset registers, jump to
> after making the page present

A Page Fault

User Program

```
$sp ← 0x7FF4
Mem[$sp+4] ← $a0
$a0 ← 0x2000
Mem[$sp+8] ← $a1
```

Page Table (slow memory)

virtual	**present?**	**physical**
7	no	–
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

A Page Fault

User Program

```
$sp ← 0x7FF4
Mem[$sp+4] ← $a0
$a0 ← 0x2000
Mem[$sp+8] ← $a1
```

Page Table (slow memory)

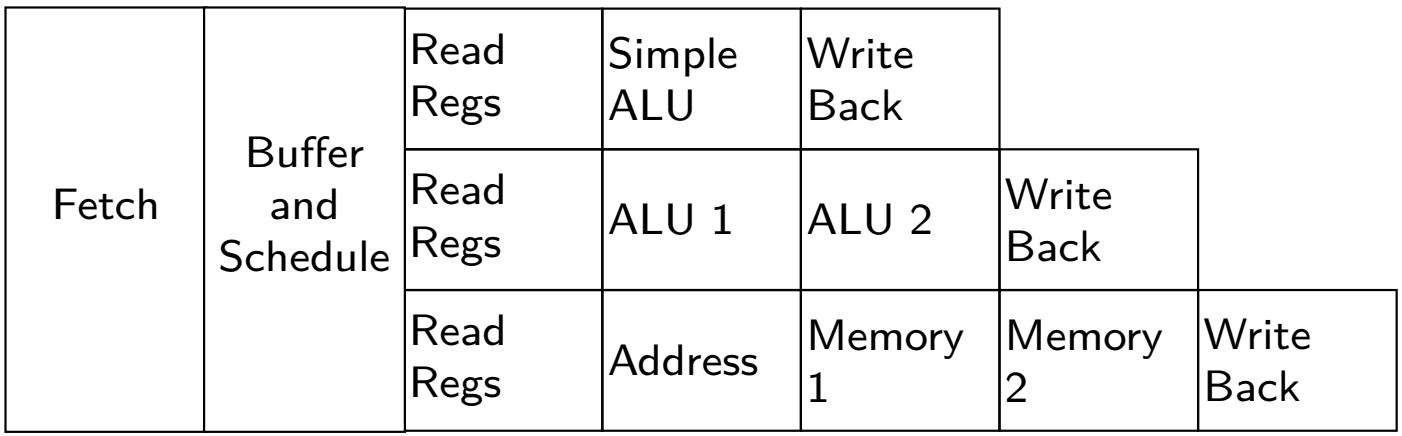
virtual	present?	physical
7	yes	5 (just allocated)
6	no	–
5	no	–
4	no	–
3	no	–
2	no	–
1	yes	1
0	no	–

Operating System

```
page_fault_handler:
    ; address of Mem[$sp + 4] ← $a0 in $exception_pc (special register)
    ; faulting address 0x7FF4 in $bad_addr (special register)
    save user registers
    insert page table entry based on $bad_addr
    restore user registers
    rfi ; return from interrupt —— jump to $exception_pc
```

Setup: a more complex pipeline

multiple dynamic issue pipeline

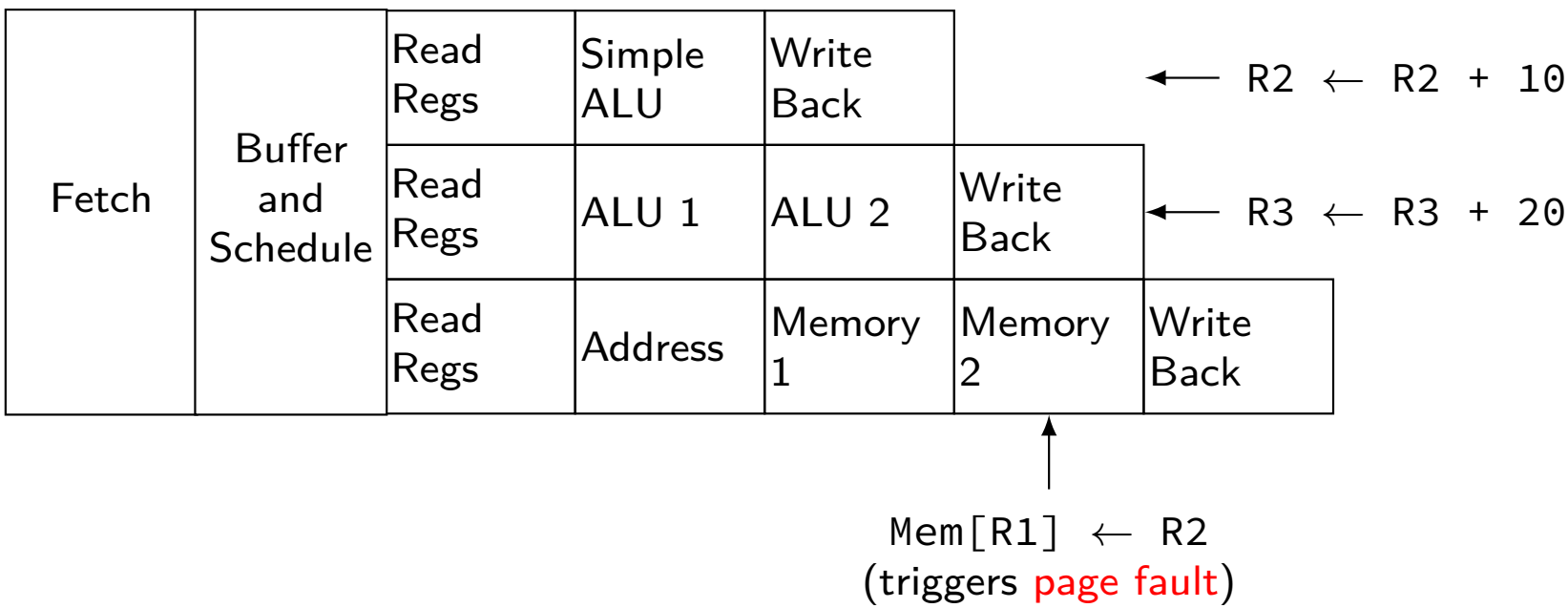


The Precise Interrupts Problem

User Program

```
R1 ← 0x7000
Mem[R1] ← R2        // Mem[0x7000] ← R2
R2 ← R2 + 10
Mem[4 + R1] ← R3 // Mem[0x7004] ← R3
R3 ← R3 + 20
```

multiple dynamic issue pipeline

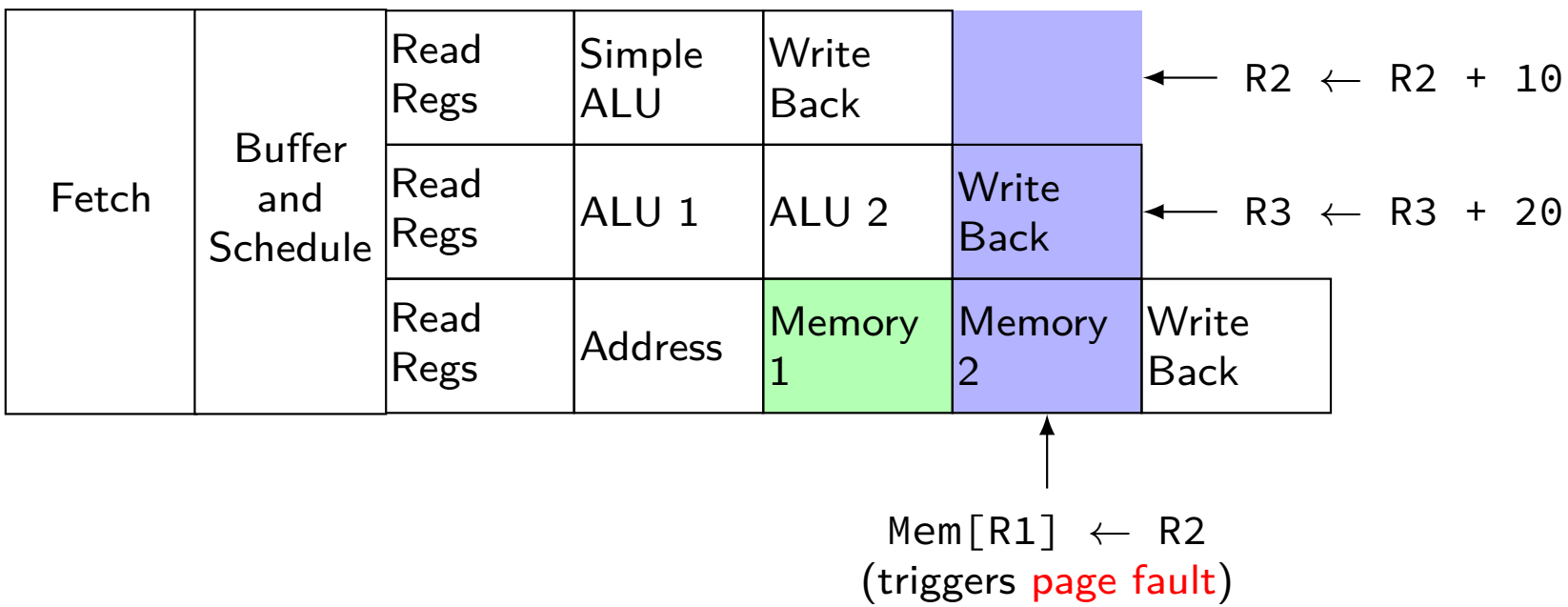


The Precise Interrupts Problem

User Program

```
R1 ← 0x7000
Mem[R1] ← R2          // Mem[0x7000] ← R2
R2 ← R2 + 10
Mem[4 + R1] ← R3      // Mem[0x7004] ← R3
R3 ← R3 + 20
```

multiple dynamic issue pipeline

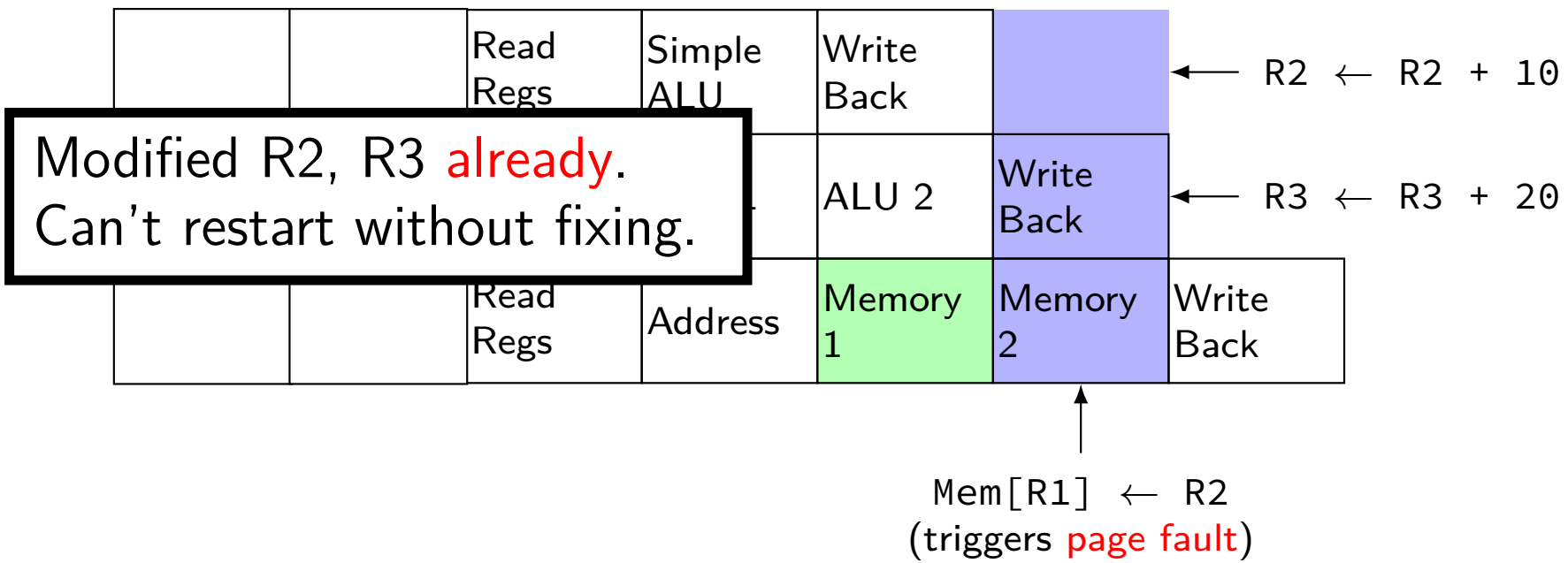


The Precise Interrupts Problem

User Program

```
R1 ← 0x7000
Mem[R1] ← R2          // Mem[0x7000] ← R2
R2 ← R2 + 10
Mem[4 + R1] ← R3      // Mem[0x7004] ← R3
R3 ← R3 + 20
```

multiple dynamic issue pipeline



Precise Interrupt Options

in-order completion

reorder buffer

future file

history buffer

Precise Interrupt Options

in-order completion

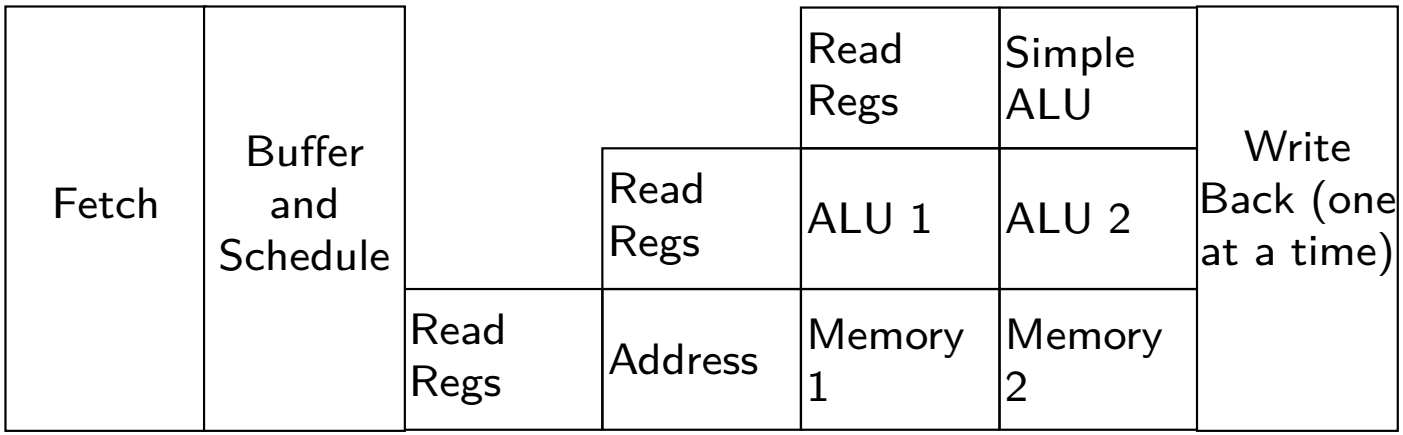
reorder buffer

future file

history buffer

In-order completion

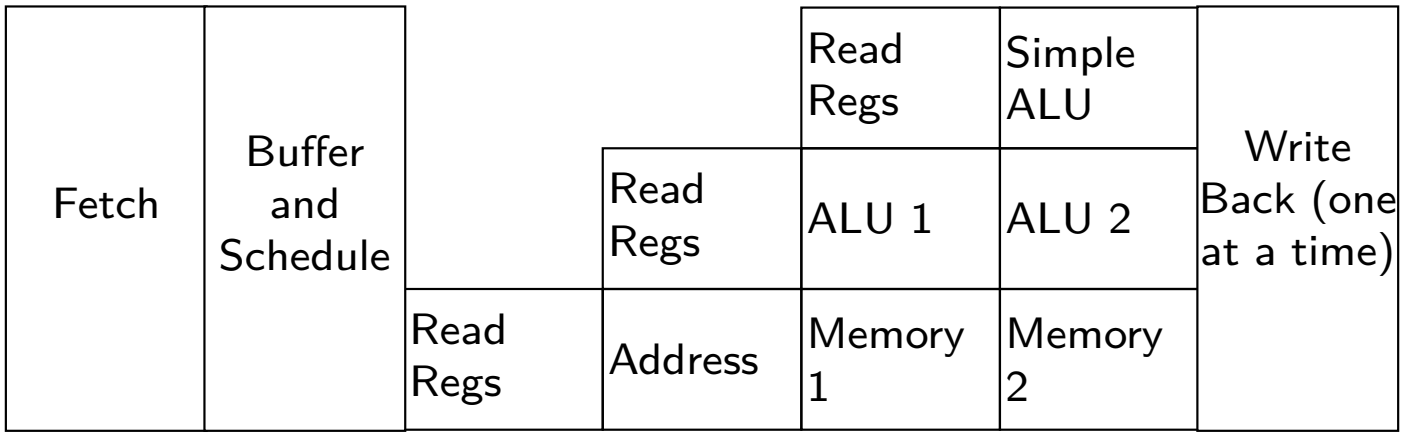
multiple dynamic issue pipeline



require instructions to write back in order

don't schedule if it wouldn't

In-order completion: Example

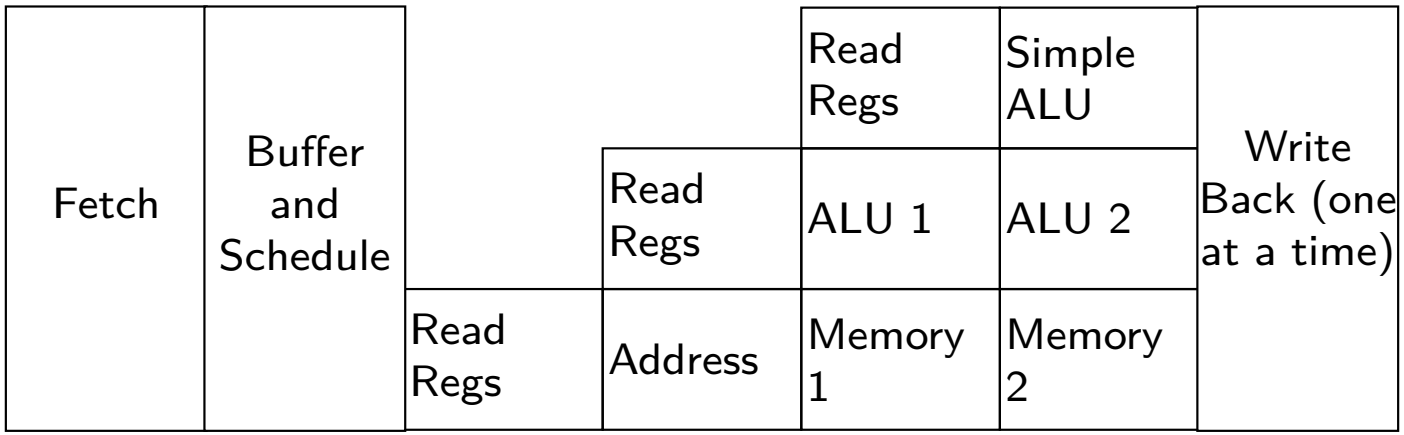



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.

In-order completion: Example

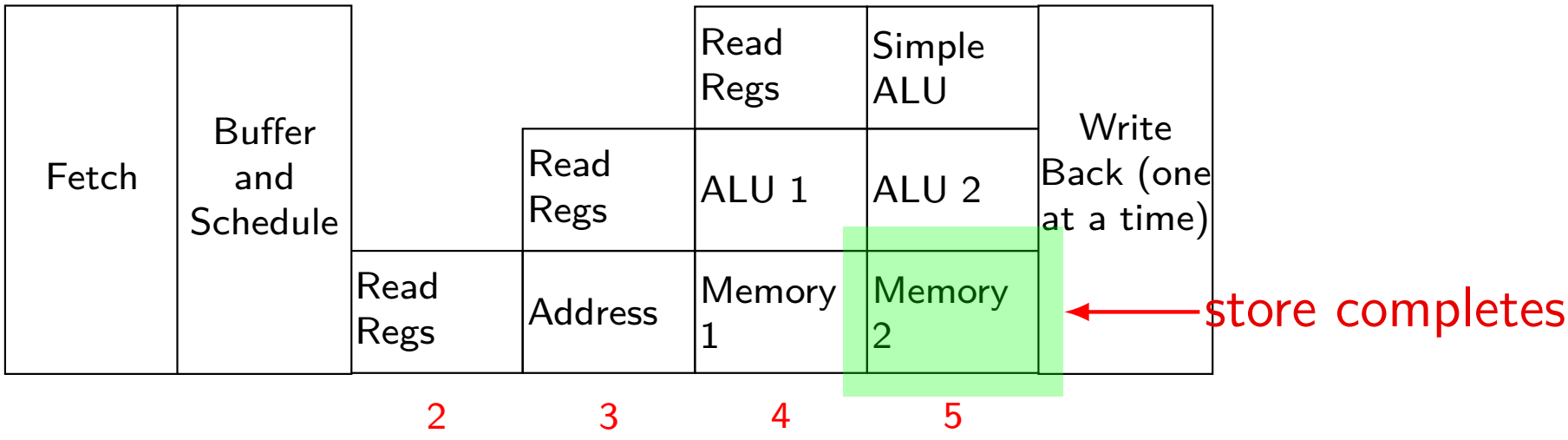



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1 + 3 = 4$	(1)

In-order completion: Example

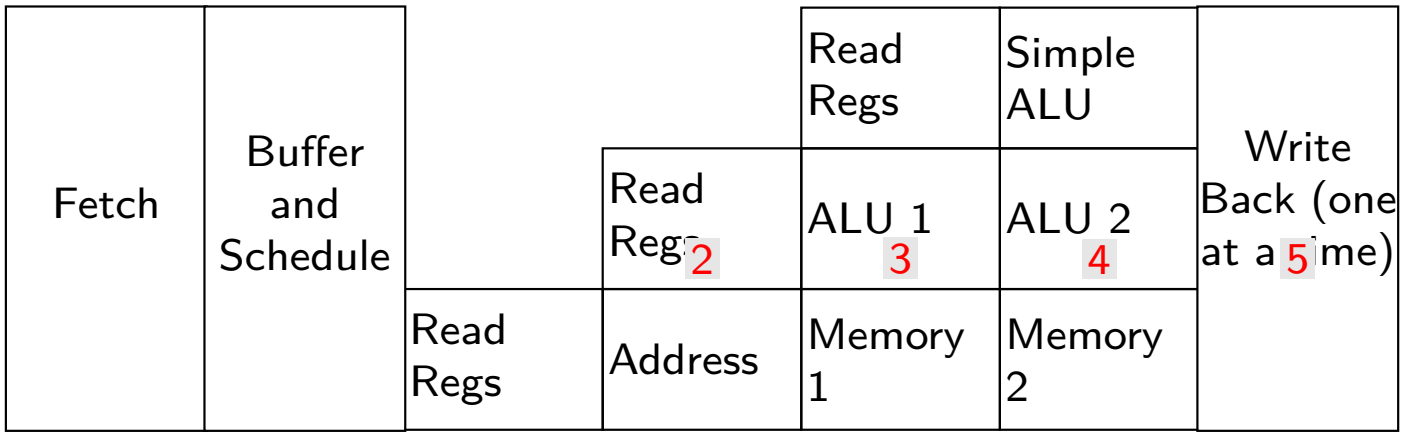



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)

In-order completion: Example



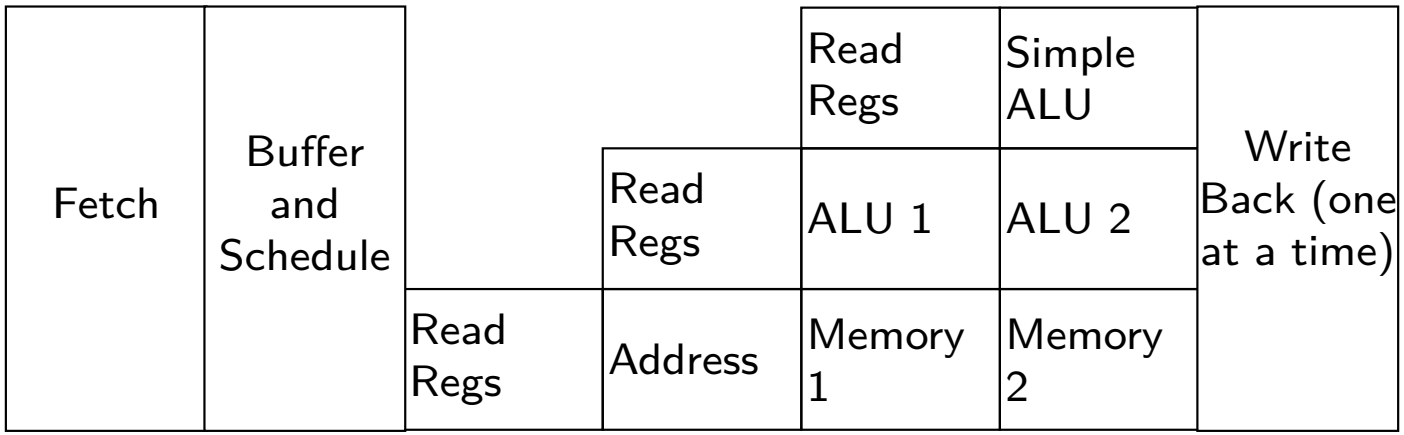

```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1 + 3 = 4$	(1)
1	memory	$1 + 4 = 5$	(2)
~~1~~	~~fancy ALU~~	~~$1 + 4 = 5$~~	~~(3)~~

(3) would complete with (2) — disallowed

In-order completion: Example

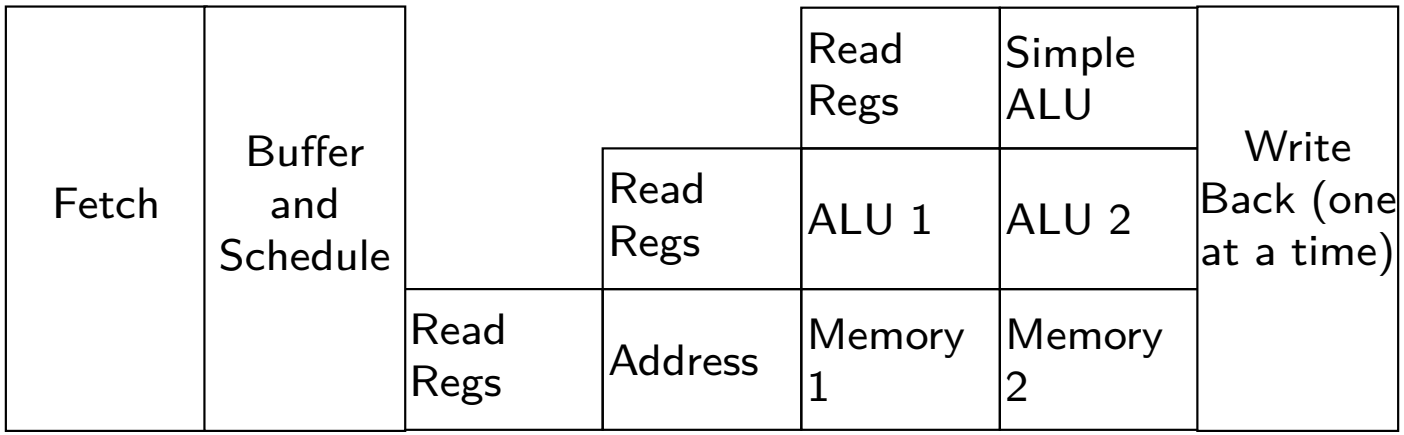



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1 + 3 = 4$	(1)
1	memory	$1 + 4 = 5$	(2)
3	simple ALU	$3 + 3 = 6$	(3)

In-order completion: Example

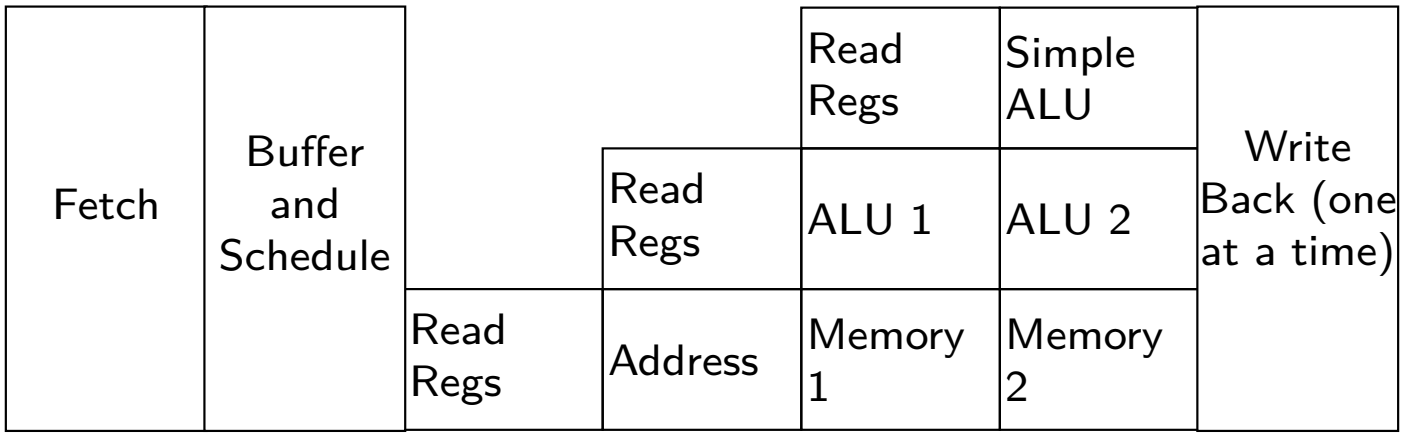



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)

In-order completion: Example

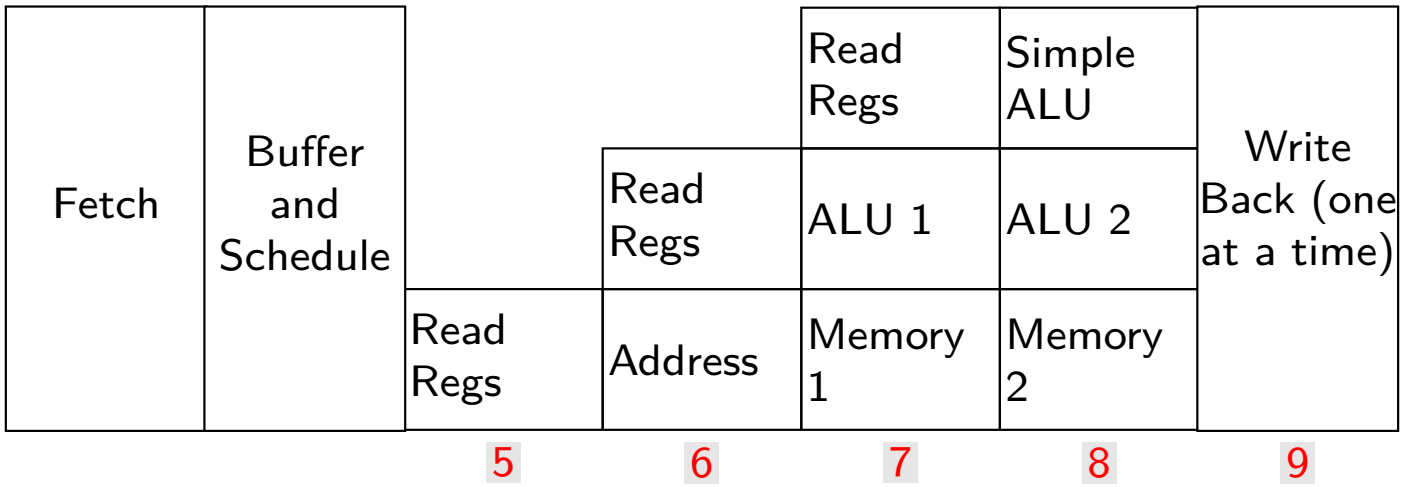



```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)

In-order completion: Example




```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

In-order completion: An exception

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

In-order completion: An exception

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

out-of-order issue

in-order commit

In-order completion: An exception

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

Exception during (2)?
Detect at cycle 5

register writes/memory stores suppressed

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=1)

time to complete	used?	pipeline	dest reg.	PC
1	no			
2	no			
3	yes	simple ALU	R1	(1)
4	yes	memory	(mem.)	(2)
5	yes	simple ALU	R2	(3)

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=1)

time to complete	used?	pipeline	dest reg.	PC
1	no			
2	no			
3	yes	simple ALU	R1	(1)
4	yes	memory	(mem.)	(2)
5	yes	simple ALU	R2	(3)

not started yet

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=2)

time to complete	used?	pipeline	dest reg.	PC
1	no			
2	yes	simple ALU	R1	(1)
3	yes	memory	(mem.)	(2)
4	yes	simple ALU	R2	(3)
5	yes	memory	(mem.)	(4)

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=3)

time to complete	used?	pipeline	dest reg.	PC
1	yes	simple ALU	R1	(1)
2	yes	memory	(mem.)	(2)
3	yes	simple ALU	R2	(3)
4	yes	memory	(mem.)	(4)
5	yes	simple ALU	R3	(5)

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=4)

time to complete	used?	pipeline	dest reg.	PC
1	yes	memory	(mem.)	(2)
2	yes	simple ALU	R2	(3)
3	yes	memory	(mem.)	(4)
4	yes	simple ALU	R3	(5)
5	yes	memory	R5	(6)

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=4)

time to complete	used?	pipeline	dest reg.	PC
1	yes	memory	(mem.)	(2)
2	yes	simple ALU	R2	(3)
3	yes	memory	(mem.)	(4)
4	yes	simple ALU	R3	(5)
5	yes	memory	R5	(6)

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Result-Shift Register

instrs by write time

cycle issued	pipeline	complete in cycle	instr.
1	simple ALU	$1+3=4$	(1)
1	memory	$1+4=5$	(2)
3	simple ALU	$3+3=6$	(3)
3	memory	$3+4=7$	(4)
5	simple ALU	$5+3=8$	(5)
4	memory	$4+5=9$	(6)

result shift register (t=5)

time to complete	used?	pipeline	dest reg.	PC
1	yes	simple ALU	R2	(3)
2	yes	memory	(mem.)	(4)
3	yes	simple ALU	R3	(5)
4	yes	memory	R5	(6)
5	no			

Rules:

Place entries in result shift register in program order.

Always place after all existing entries.

(Also, don't break data dependencies.)

Precise Interrupt Options

in-order completion

reorder buffer

future file

history buffer

The Re-Order Buffer

multiple dynamic issue pipeline

Fetch	Reserve ROB entry and Schedule	Read Regs	Simple ALU	Write ROB				Commit: Copy from ROB to Regs
		Read Regs	ALU 1	ALU 2	Write ROB			
		Read Regs	Address	Memory 1	Memory 2	Write ROB		

reorder buffer (ROB)

entry #	used?	dest reg.	result	PC
4	yes	R1	(waiting)	(1)
5	yes	(mem.)	(waiting)	(2)
6	yes	R2	(waiting)	(3)
7	yes	(mem.)	(waiting)	(4)
8	yes	R3	(waiting)	(5)
9	yes	R5	(waiting)	(6)

reserved in-order

written whenever ready

committed to regs. in-order

Scheduling with ROB

```
R1 ← 0x7000          // (1)
Mem[R1] ← R2         // (2)
R2 ← R2 + 10         // (3)
Mem[4 + R1] ← R3     // (4)
R3 ← R3 + 20         // (5)
R5 ← Mem[8 + R1]     // (6)
```

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB)

entry #	used?	dest reg.	result	PC
4	yes	R1	(waiting)	(1)
5	yes	(mem.)	(ask mem)	(2)
6	yes	R2	(waiting)	(3)
7	yes	(mem.)	(ask mem)	(4)
8	yes	R3	(waiting)	(5)
9	yes	R5	(waiting)	(6)

Operating the ROB

Look at entry for first instruction in program order

Wait for result of that instruction to be available

Then, copy result to destination register
(or send signal to memory system)

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB) (t=1)

entry #	used?	dest reg.	result	PC
4	yes	R1	(waiting)	(1)
5	yes	(mem.)	(ask cache)	(2)
6	yes	R2	(waiting)	(3)
7	yes	(mem.)	(ask cache)	(4)
8	yes	R3	(waiting)	(5)
9	yes	R5	(waiting)	(6)

Operating the ROB

Look at entry for first instruction in program order

Wait for result of that instruction to be available

Then, copy result to destination register
(or send signal to memory system)

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB) (t=4)

entry #	used?	dest reg.	result	PC
4	yes	R1	0x7000	(1)
5	yes	(mem.)	(ask mem)	(2)
6	yes	R2	(waiting)	(3)
7	yes	(mem.)	(ask mem)	(4)
8	yes	R3	(waiting)	(5)
9	yes	R5	(waiting)	(6)

Operating the ROB

Look at entry for first instruction in program order

Wait for result of that instruction to be available

Then, copy result to destination register
(or send signal to memory system)

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB) (t=4)

entry #	used?	dest reg.	result	PC
~~4~~	~~yes~~	~~R1~~	~~0x7000~~	~~(1)~~
5	yes	(mem.)	(ask mem)	(2)
6	yes	R2	(waiting)	(3)
7	yes	(mem.)	(ask mem)	(4)
8	yes	R3	(waiting)	(5)
9	yes	R5	(waiting)	(6)

copied to R1

Operating the ROB

Look at entry for first instruction in program order

Wait for result of that instruction to be available

Then, copy result to destination register
(or send signal to memory system)

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB) (t=5)

entry #	used?	dest reg.	result	PC
5	yes	(mem.)	(ask mem)	(2)
6	yes	R2	`0x10020`	(3)
7	yes	(mem.)	(ask mem)	(4)
8	yes	R3	`0x10050`	(5)
9	yes	R5	(waiting)	(6)

memory not ready

Operating the ROB

Look at entry for first instruction in program order

Wait for result of that instruction to be available

Then, copy result to destination register
(or send signal to memory system)

instrs by write time

cycle issued	pipeline	to ROB in cycle	instr.
1	simple ALU	$1+3=4$	(1)
2	memory	$2+4=6$	(2)
1	fancy ALU	$1+4=5$	(3)
3	memory	$2+4=7$	(4)
2	simple ALU	$2+3=5$	(5)
1	memory	$1+5=6$	(6)

reorder buffer (ROB) (t=6)

entry #	used?	dest reg.	result	PC
~~5~~	~~yes~~	~~(mem.)~~	~~(ask mem)~~	~~(2)~~
~~6~~	~~yes~~	~~R2~~	~~0x10020~~	~~(3)~~
7	yes	(mem.)	(ask mem)	(4)
8	yes	R3	0x20050	(5)
9	yes	R5	0x34567	(6)

memory ready

ROB and exceptions

result is exception instead of value to store

clear rest of ROB instead of committing to register

then call OS

Exception info needs to be stored in ROB — why?

ROB and forwarding

```
MEM[R6] ← R7   ; (1)
R1 ← R2 + R3   ; (2)
R4 ← R1 + R5   ; (3)
```

reorder buffer increases time to write registers

(3) needs to wait for (1) to finish

otherwise, old value of R1 unless...

bypass to read reorder buffer instead of regfile

likely more delay than MIPS-style forwarding

circular buffer

reorder buffer (ROB) (t=6)

entry #	used?	dest reg.	result	PC
0				
1				
2				

(next to commit) head

3	yes		
4	yes		
5	yes		
6	yes		
7	yes		

tail

ROB and memory

defer stores until at top of ROB

also loads unless they use different addresses than stores

more complicated with multiple processors

Precise Interrupt Options

in-order completion

reorder buffer

future file

history buffer

Future file

maintain two copies of registers

use reorder buffer for one — in-order commit

use other one for forwarding

easier than bypass logic?

Precise Interrupt Options

in-order completion

reorder buffer

future file

history buffer

History buffer

Like reorder buffer but holds prior values

On exception:

- wait for all pending operations to finish
- copy prior values for everything after exception (in program order)

Speculation

predict a branch?

make entry for branch in reorder buffer

if result is not expected — treat like exception

avoid committing speculated register changes

Summary

in-order completion

- restriction on when instructions start

reorder buffer

- but store results in buffer
- 'commit' from buffer to registers in program order
- bypassing to make buffered values available early

future file

- instead of bypassing, maintain two copies of registers

history buffer

- store results out-of-order to registers
- but store prior values in buffer
- copy from buffer on exception