

CS250P: Computer Systems Architecture

Achieving Correct Pipelining



Sang-Woo Jun

Fall 2023

A problematic example

- ❑ What should be stored in data+8? 3, right?

```
la t0, data
lw s0, 0(t0)
lw s1, 4(t0)
add s2, s0, s1
sw s2, 8(t0)
data:
> .word 1 2
```

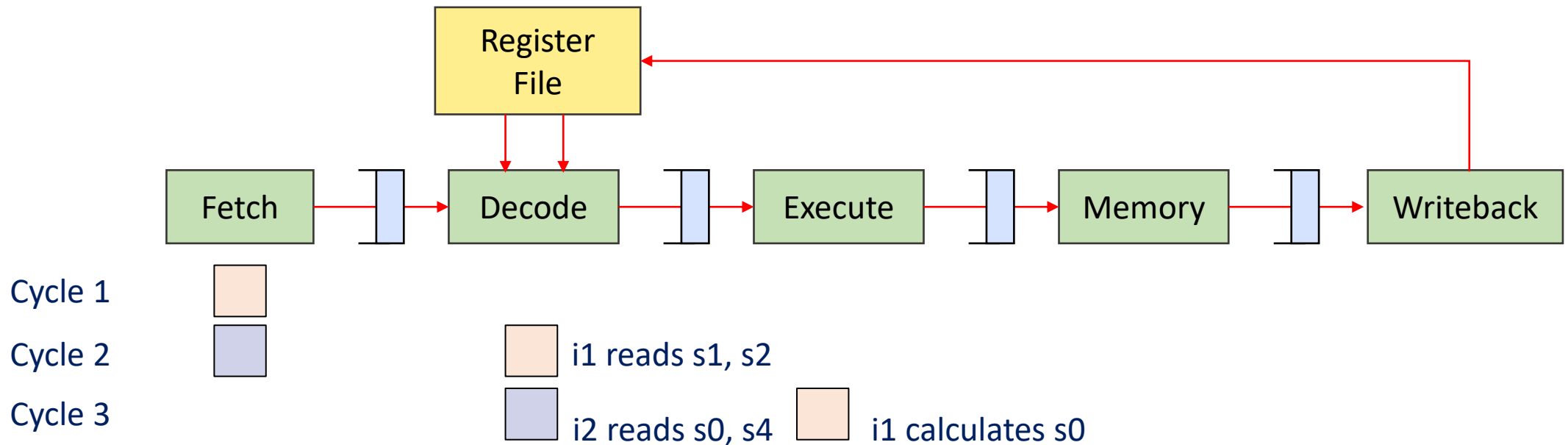
- ❑ Assuming zero-initialized register file, our pipeline will write zero

Why? "Hazards"

Hazard #1: Read-After-Write (RAW) Data hazard

□ When an instruction depends on a register updated by a previous instruction's execution results

- e.g.,
i1: add s0, s1, s2
i2: add s3, s0, s4

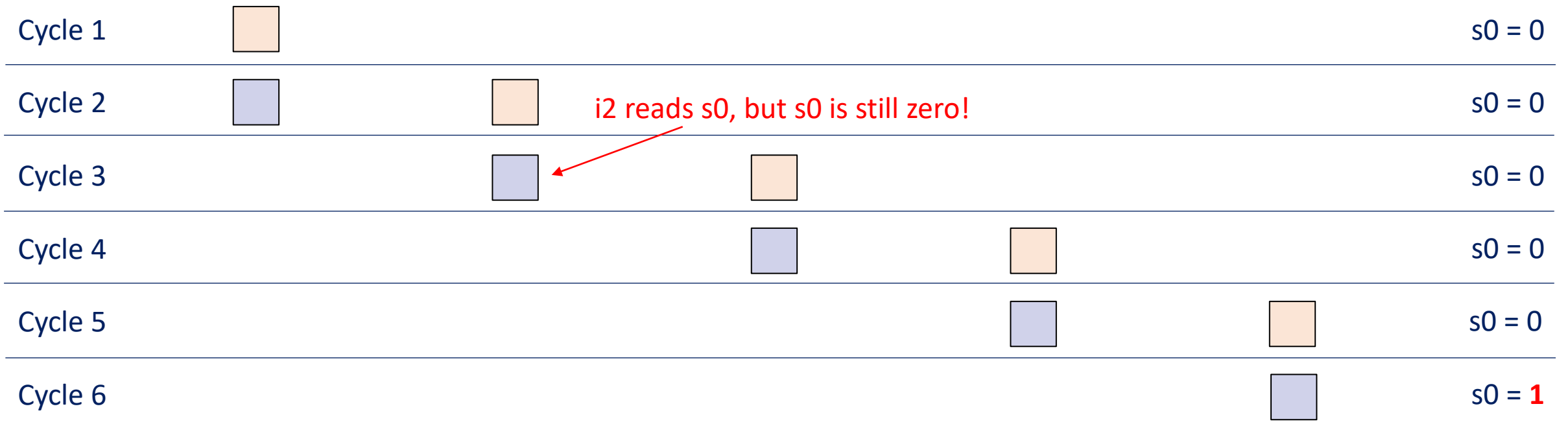
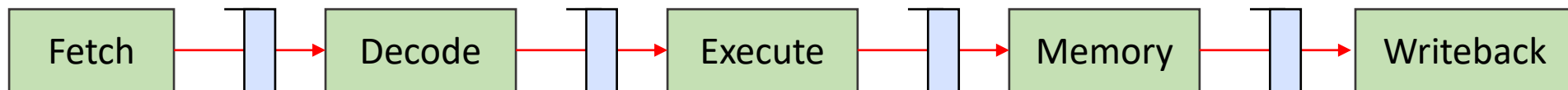


Hazard #1: Read-After Write (RAW) Hazard

i1: addi s0, zero, 1

i2: addi s1, s0, 0

s0 should be 1, s1 should be 1

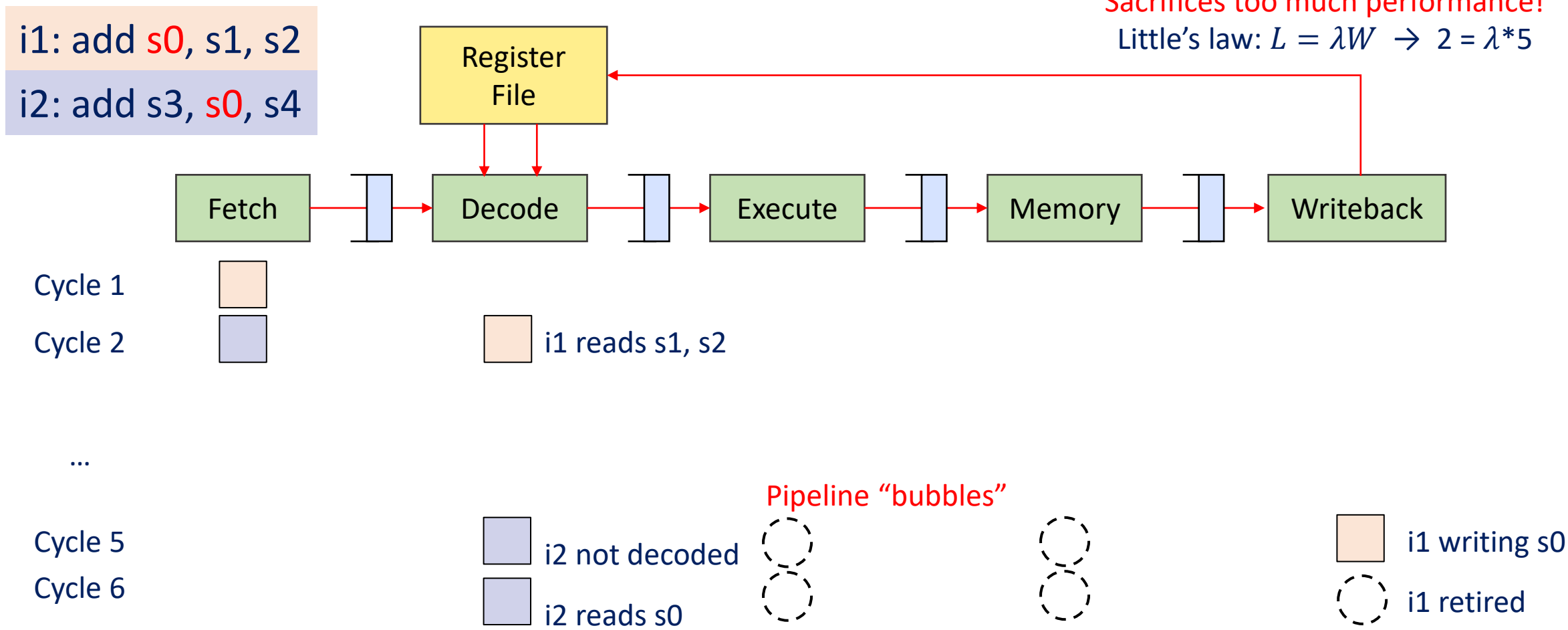


Solution #1: Stalling

- ❑ The processor can choose to stall decoding when RAW hazard detected

Sacrifices too much performance!

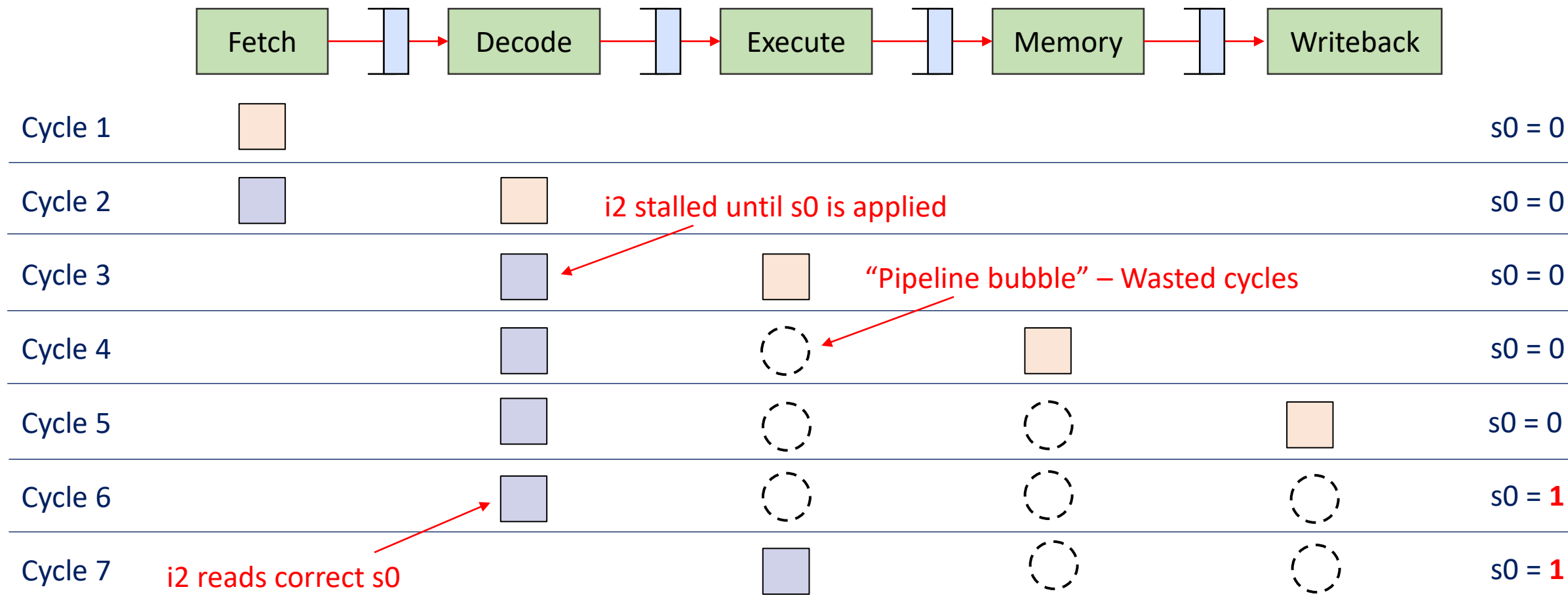
Little's law: $L = \lambda W \rightarrow 2 = \lambda * 5$



Solution #1: Stalling

i1: addi s0, zero, 1

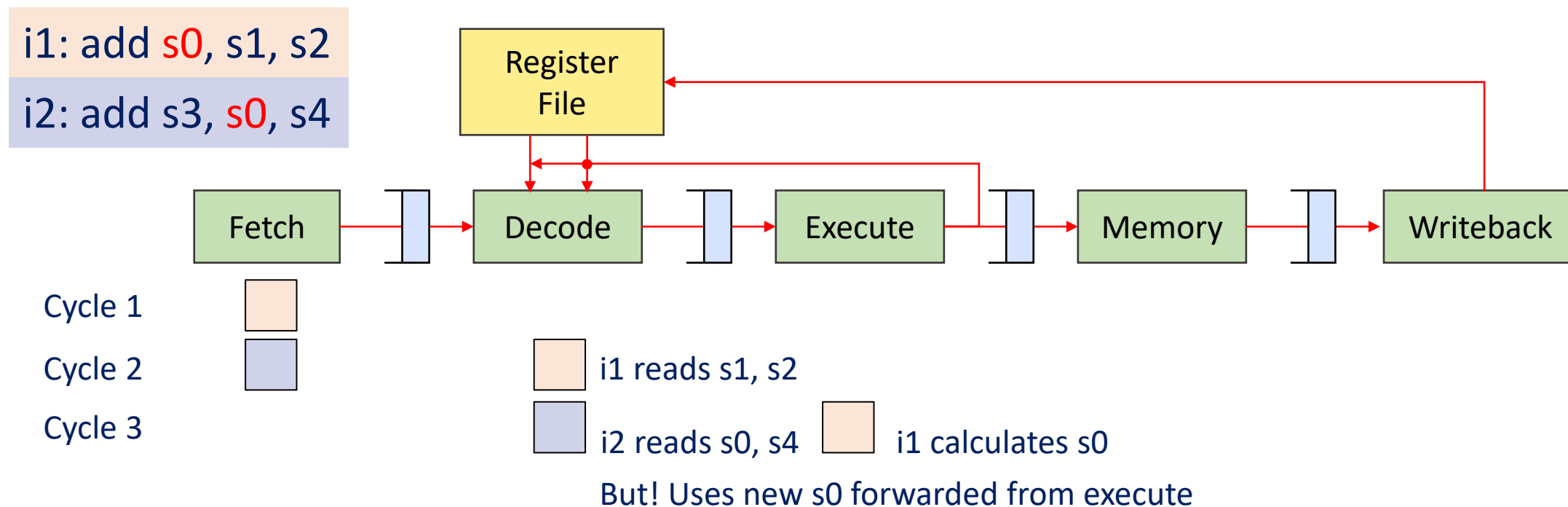
i2: addi s1, s0, 0



Sacrifices too much performance!

Solution #2: Forwarding (aka Bypassing)

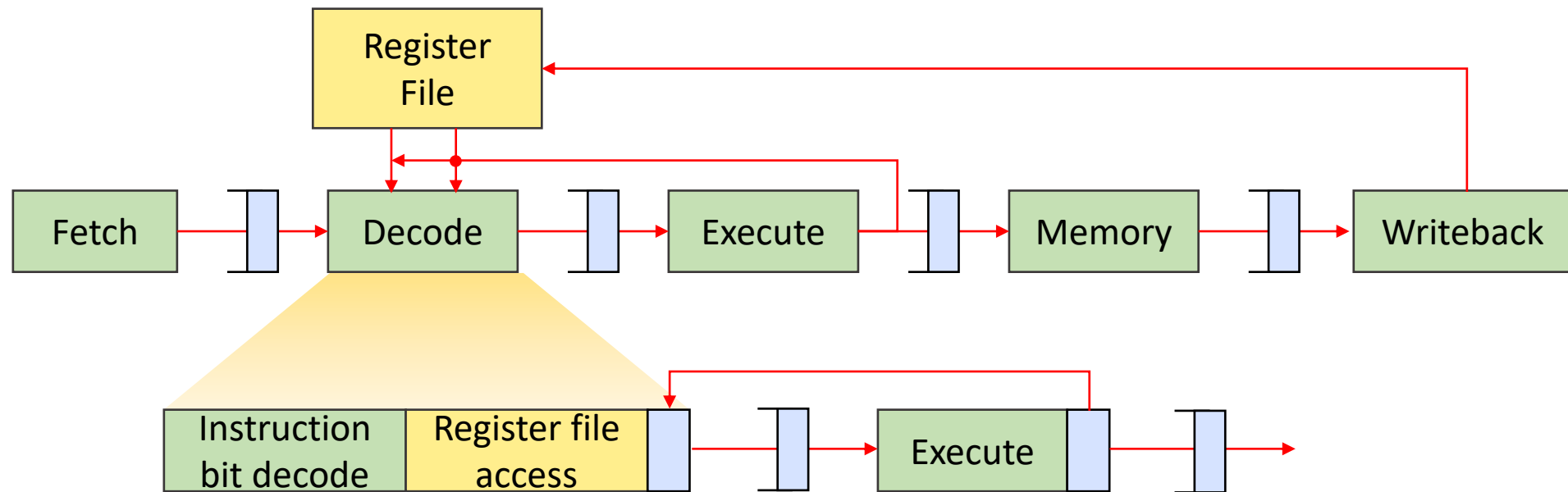
- Forward execution results to input of decode stage
 - New values are used if write index and a read index is the same



No pipeline stalls!

Solution #2: Forwarding details

- ❑ May still require stalls for a deeper pipeline microarchitecture
 - If execute took many cycles?
- ❑ Adds combinational path from execute to decode
 - But does not imbalance pipeline very much! (But it does a little bit)



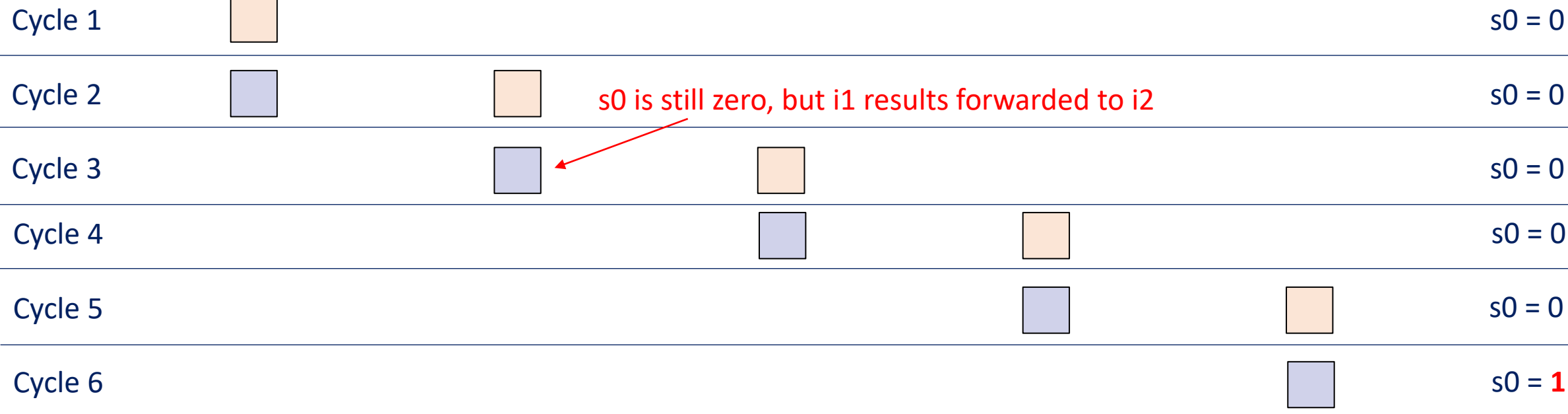
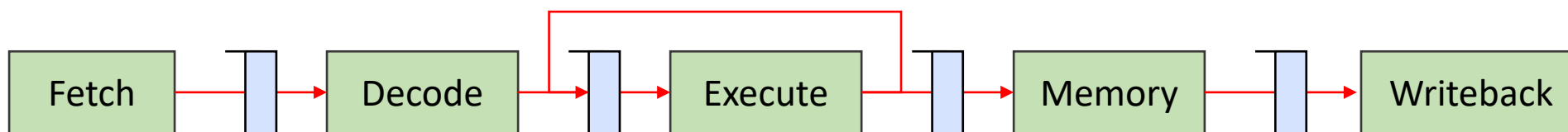
Combinational path only to end of decode stage! (decode/regfile access does not depend on forwarded data)

Solution #2: Forwarding

i1: addi s0, zero, 1

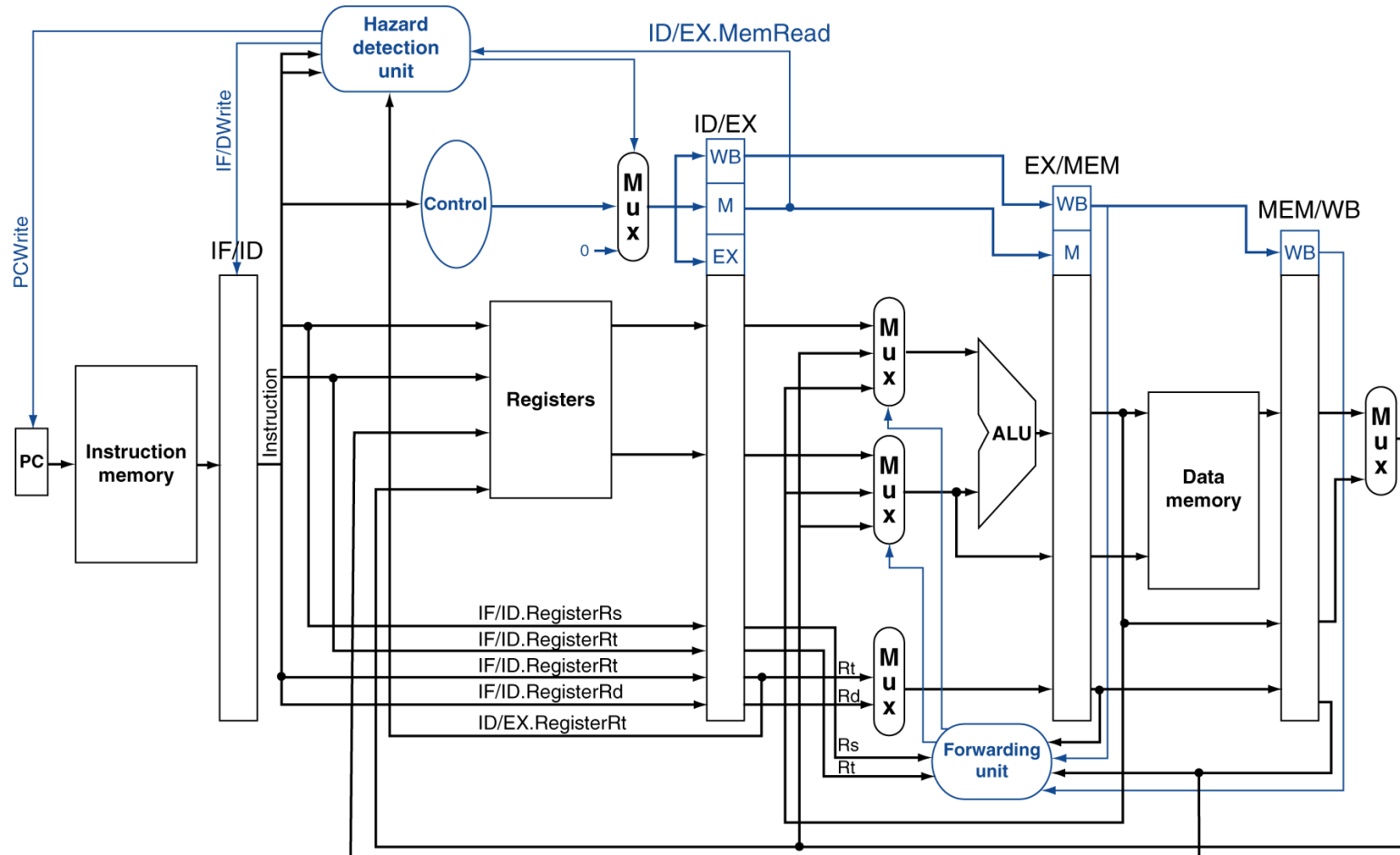
i2: addi s1, s0, 0

results forwarded to decode within same cycle



Forwarding is possible in this situation because the answer (s0 = 1) exists somewhere in the processor!

Datapath with Hazard Detection



Not very intuitive... We'll visit it with code at a discussion section

Hazard #2: Load-Use Data Hazard

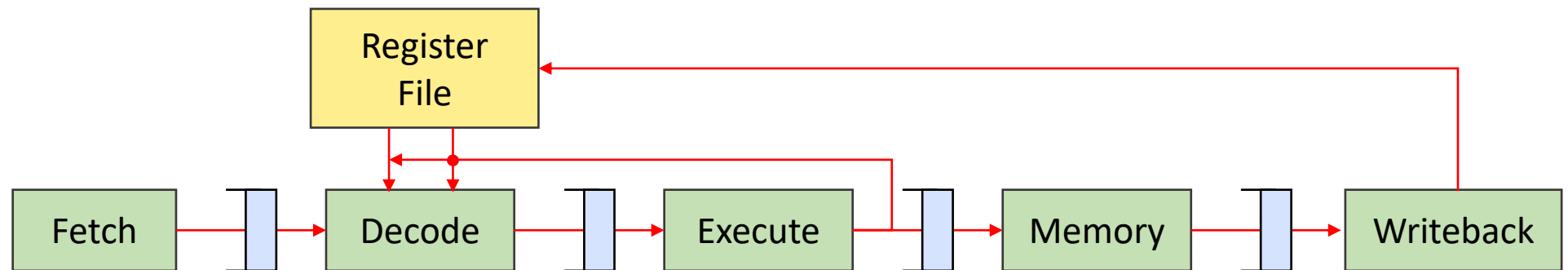
□ When an instruction depends on a register updated by a previous instruction

○ e.g., `i1: lw s0, 0(s2)`

`i2: addi s1, s0, 1`

□ Forwarding doesn't work here, as loads only materialize at writeback

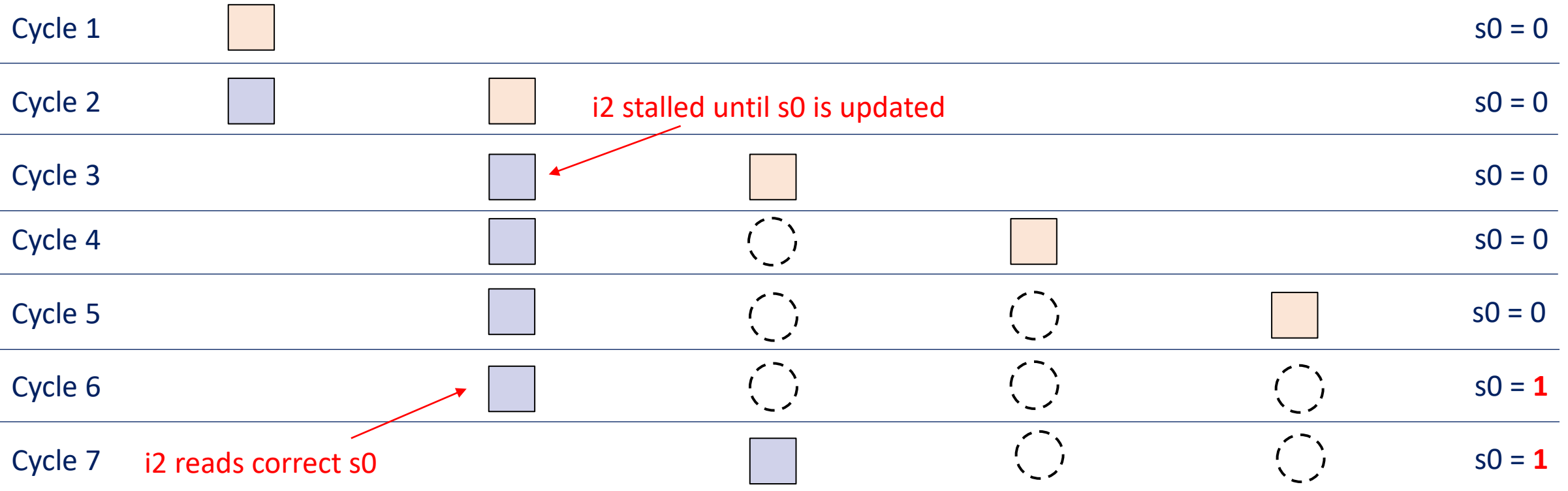
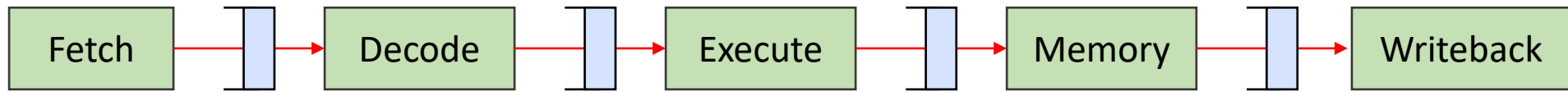
○ Only architectural choice is to stall



Hazard #2: Load-Use Data Hazard

i1: lw s0, 0(s2)

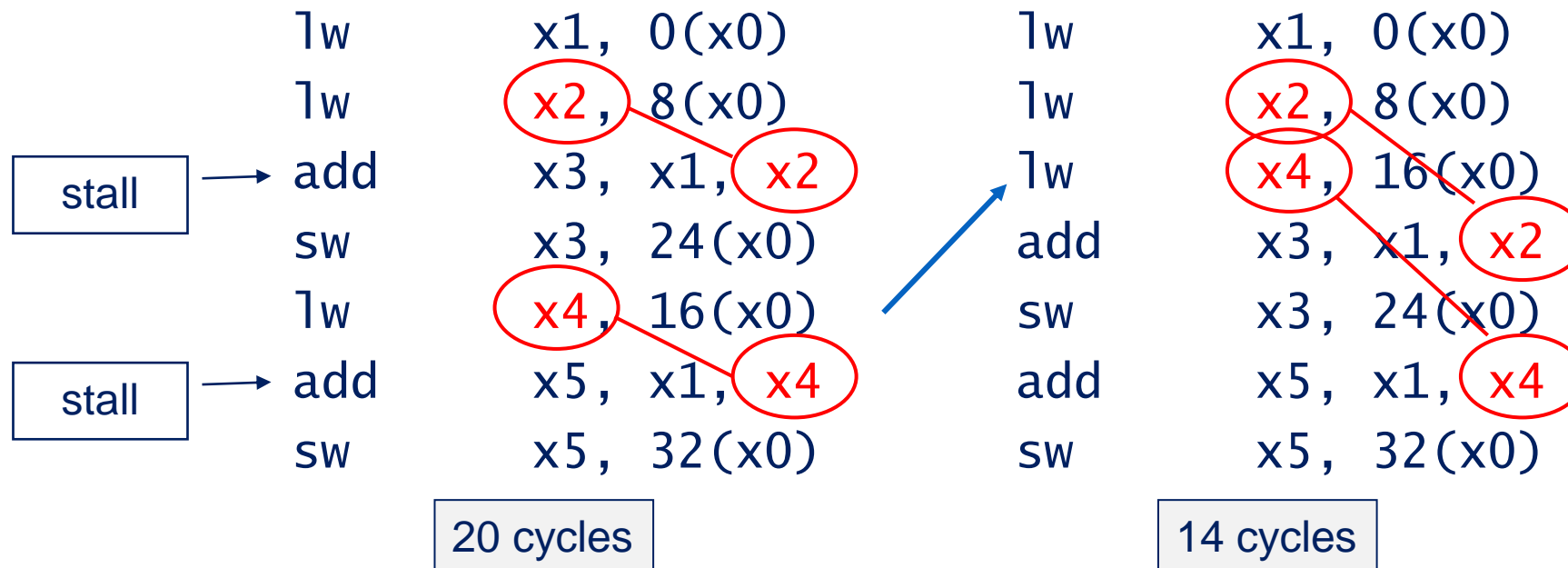
i2: addi s1, s0, 1



Forwarding is not useful because the answer (s0 = 1) exists outside the chip (memory)

A non-architectural solution: Code scheduling by compiler

- ❑ Reorder code to avoid use of load result in the next instruction
- ❑ e.g., $a = b + e$; $c = b + f$;



Compiler does best, but not always possible!

Review: A problematic example

```
la t0, data
lw s0, 0(t0)
lw s1, 4(t0)
add s2, s0, s1
sw s2, 8(t0)
data:
> .word 1 2
```

← RAW hazard
← RAW hazard
← RAW hazard
← Load-Use hazard
← RAW hazard

- Note: “la” is not an actual RISC-V instruction
 - Pseudo-instruction expanded to one or more instructions by assembler
 - e.g., `auipc x5,0x1`
`addi x5,x5,-4 # ← RAW hazard!`

Other potential data hazards

Dangerous if a later instruction's state access can happen before an earlier instruction's access

❑ Read-After-Write (RAW) Hazard

- Obviously dangerous! -- Writeback stage comes after decode stage
- (Later instructions' reads **can** happen before earlier instructions' write)

❑ Write-After-Write (WAW) Hazard

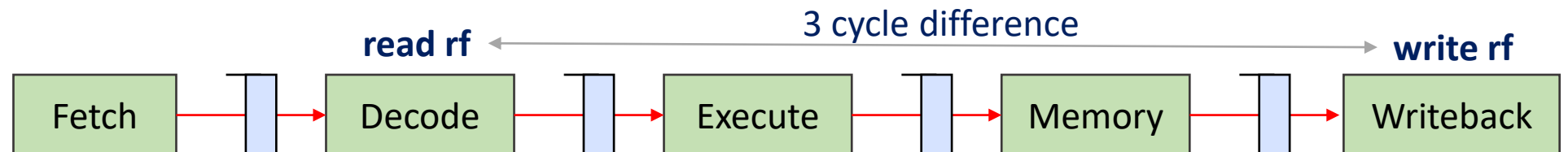
- No hazard for in-order processors

❑ Write-After-Read (WAR) Hazard

- No hazard for in-order processors -- Writeback stage comes after decode stage
- (Later instructions' reads **cannot** happen before earlier instructions' write)

❑ Read-After-Read (RAR) Hazard?

- No hazard within processor

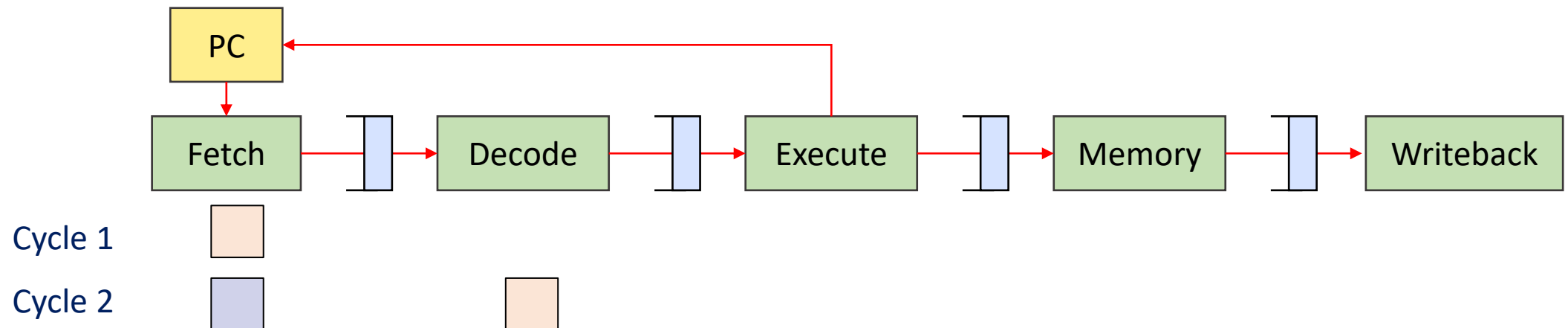


Hazard #3: Control hazard

- ❑ Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - e.g., Still working on decode stage of branch

i1: beq s0, zero, elsewhere

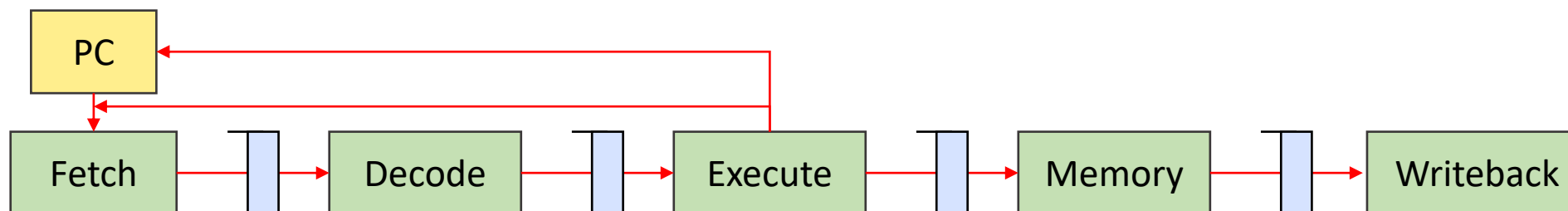
i2: addi s1, s0, 1



Should I load this or not?

Control hazard (partial) solutions

- ❑ Branch target address can be forwarded to the fetch stage
 - Without first being written to PC
 - Still may introduce (one less, but still) bubbles



- ❑ Decode stage can be augmented with logic to calculate branch target
 - May imbalance pipeline, reducing performance
 - Doesn't help if instruction memory takes long (cache miss, for example)

Aside: An awkward solution: Branch delay slot

- ❑ In a 5-stage pipeline with forwarding, one branch hazard bubble is injected in best scenario
- ❑ Original MIPS and SPARC processors included “branch delay slots”
 - One instruction after branch instruction was executed regardless of branch results
 - Compiler will do its best to find something to put there (if not, “nop”)
- ❑ Goal: Always fill pipeline with useful work
- ❑ Reality:
 - Difficult to always fill slot
 - Deeper pipelines meant one measly slot didn't add much (Modern MIPS has 5+ cycles branch penalty!)

But once it's added, it's forever in the ISA...
One of the biggest criticisms of MIPS

CS250P: Computer Systems Architecture

Achieving Correct Pipelining

-- Branch Prediction



Sang-Woo Jun

Fall 2023

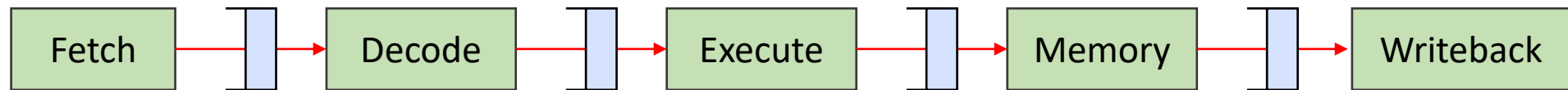
Eight great ideas

- Design for Moore's Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy



Control hazard and pipelining

- ❑ Solving control hazards is a fundamental requirement for pipelining
 - Fetch stage needs to keep fetching instructions without feedback from later stages
 - Must keep pipeline full somehow!
 - ... Can't know what to fetch



Cycle 1 Fetch PC = 0

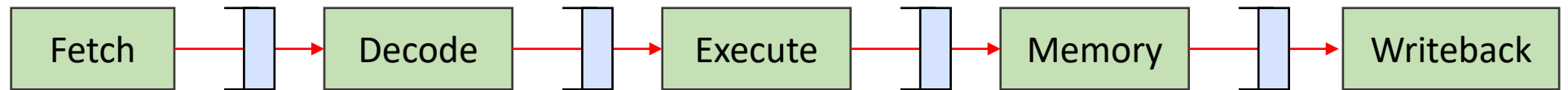
Cycle 2 Fetch PC = ...? Decode PC = 0

Control hazard (partial) solution

Branch prediction

- ❑ We will try to predict whether branch is taken or not
 - If prediction is correct, great!
 - If not, we somehow do not apply the effects of mis-predicted instructions
 - (Effectively same performance penalty as stalling in this case)
 - Very important to have mispredict detection before any state change!
 - Difficult to revert things like register writes, memory I/O
- ❑ Simplest branch predictor: Predict not taken
 - Fetch stage will keep fetching $pc \leq pc + 4$ until someone tells it not to

Predict not taken example



```
addi t1, zero, 3
addi t2, zero, 3
beq t1, t2, skip
sw t3, 0(t0)
ret
skip:
sw t2, 0(t0)
ret
```



Fetch correct branch

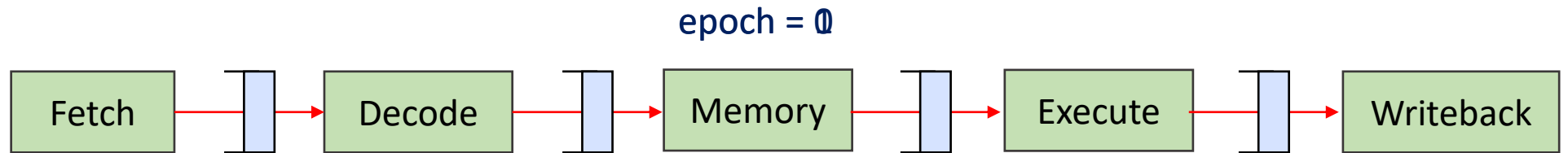
Pipeline bubbles

No state update before Execute stage detects misprediction
(Fetch and Decode stages don't write to register)

How to handle mis-predictions?

- ❑ Implementations vary, each with pros and cons
 - Sometimes, execute sends a combinational signal to all previous stages, turning all instructions into a “nop”
- ❑ A simple method is “epoch-based”
 - All fetched instructions belong to an “epoch”, represented with a number
 - Instructions are tagged with their epoch as they move through the pipeline
 - In the case of mis-predict detection, global epoch is increased, and future instructions from previous epochs are ignored

Predict not taken example with epochs

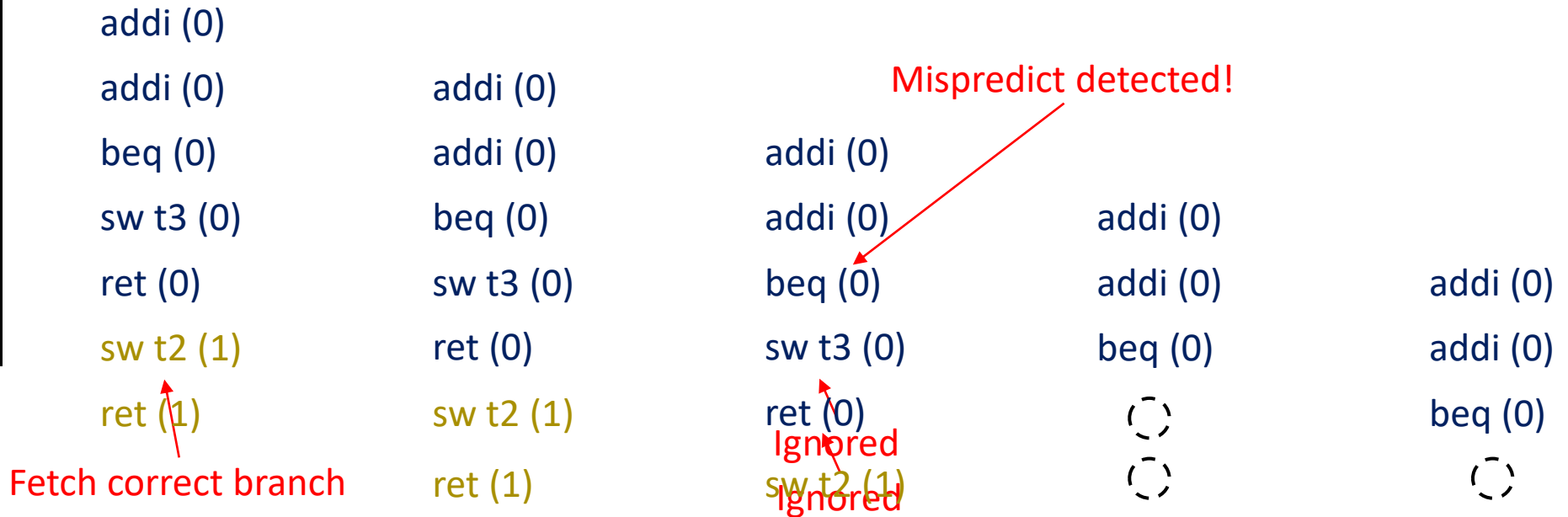


```

addi t1, zero, 3
addi t2, zero, 3

beq t1, t2, skip
sw t3, 0(t0)
ret

skip:
sw t2, 0(t0)
ret
    
```



Some classes of branch predictors

❑ Static branch prediction

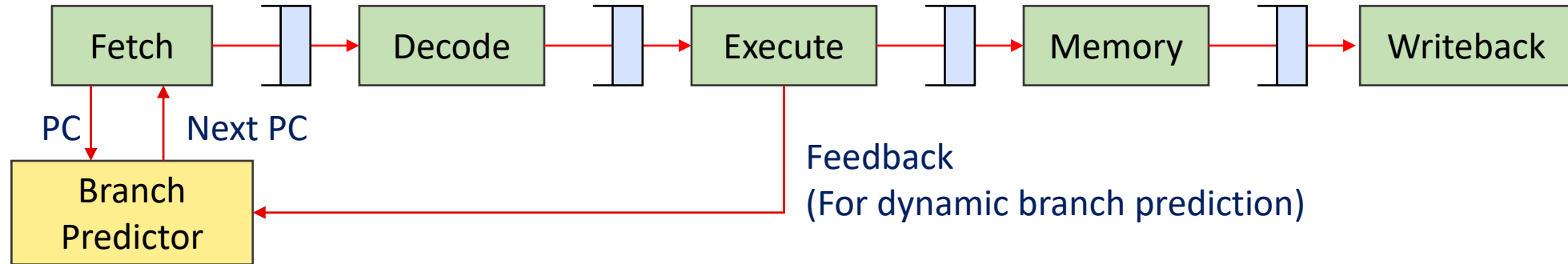
- Based on typical branch behavior
- Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken

❑ Dynamic branch prediction

- Hardware measures actual branch behavior
 - e.g., record recent history (1-bit “taken” or “not taken”) of each branch in a fixed size “branch history table”
- Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

Many many different methods, Lots of research, some even using neural networks!

Pipeline with branch prediction

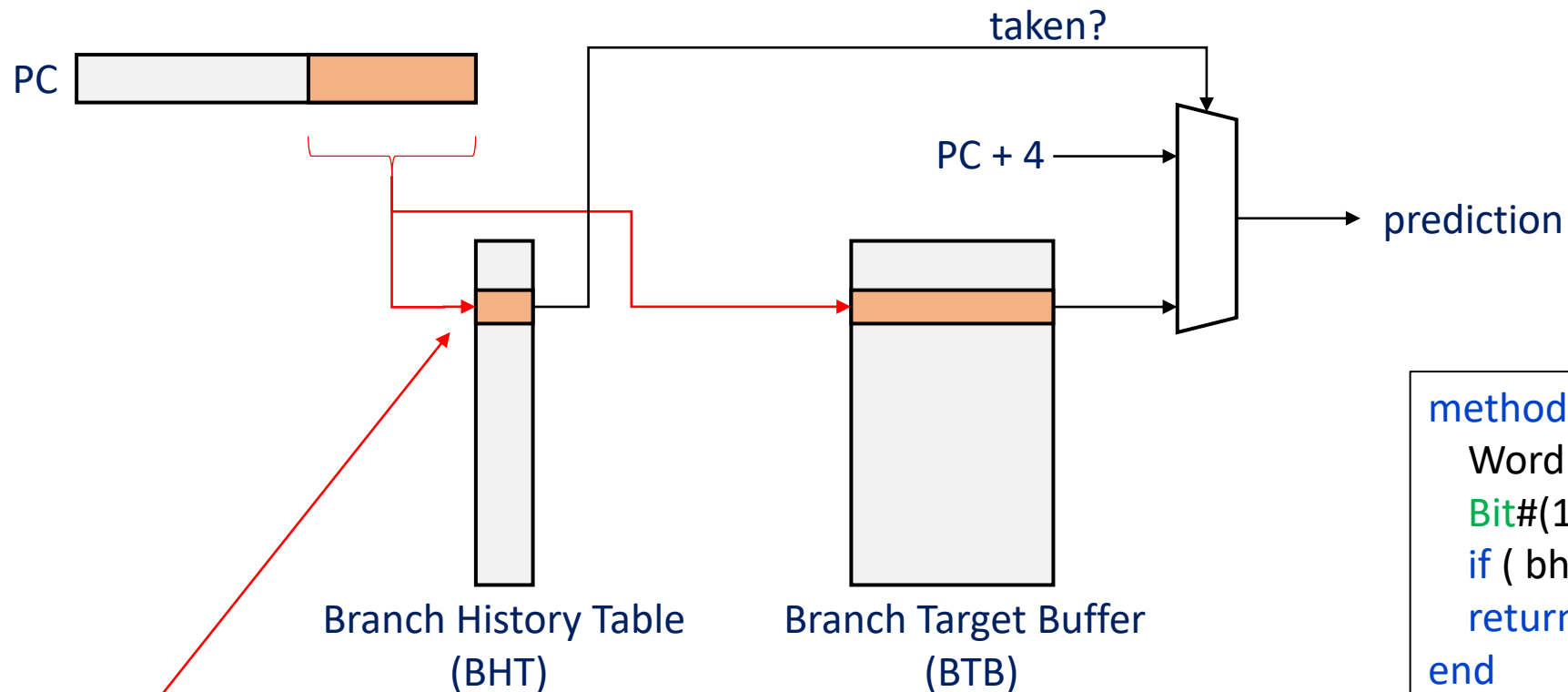


- ❑ Branch predictor predicts what should be the next PC
 - Typically based on the current PC as input
- ❑ Dynamic branch predictors adapt to program using feedback
- ❑ If prediction is correct, great! If not, make sure mispredicted instructions don't effect state
 - We looked at the epoch method of doing this (2 bubbles!)

Dynamic branch prediction

- ❑ Two questions about a PC address being fetched
 - Will this instruction cause a branch?
 - If so, where will it branch to?
 - Both information are needed to predict-fetch a branch
- ❑ Two architectural entities for predicting the answer to these questions
 - Branch History Table (BHT)
 - Whether this instruction is an instruction, and if it causes a branch
 - Branch Target Buffer (BTB)
 - Which address this instruction will jump to
 - (There are many variations – This is a common example)

Dynamic branch prediction



```
method Word predict(Word pc) begin
  Word next_pc = pc + 4;
  Bit#(10) lsb = truncate(pc);
  if ( bht[lsb] ) next_pc = btb(lsb);
  return next_pc;
end
```

Execute stage updates BHT and BTB
with actual behavior (if it is a branch instruction)

Why truncate PC? BHT/BTB is typically small! (2048 elements or so)
Different branches may map to same buffer element... ☹️

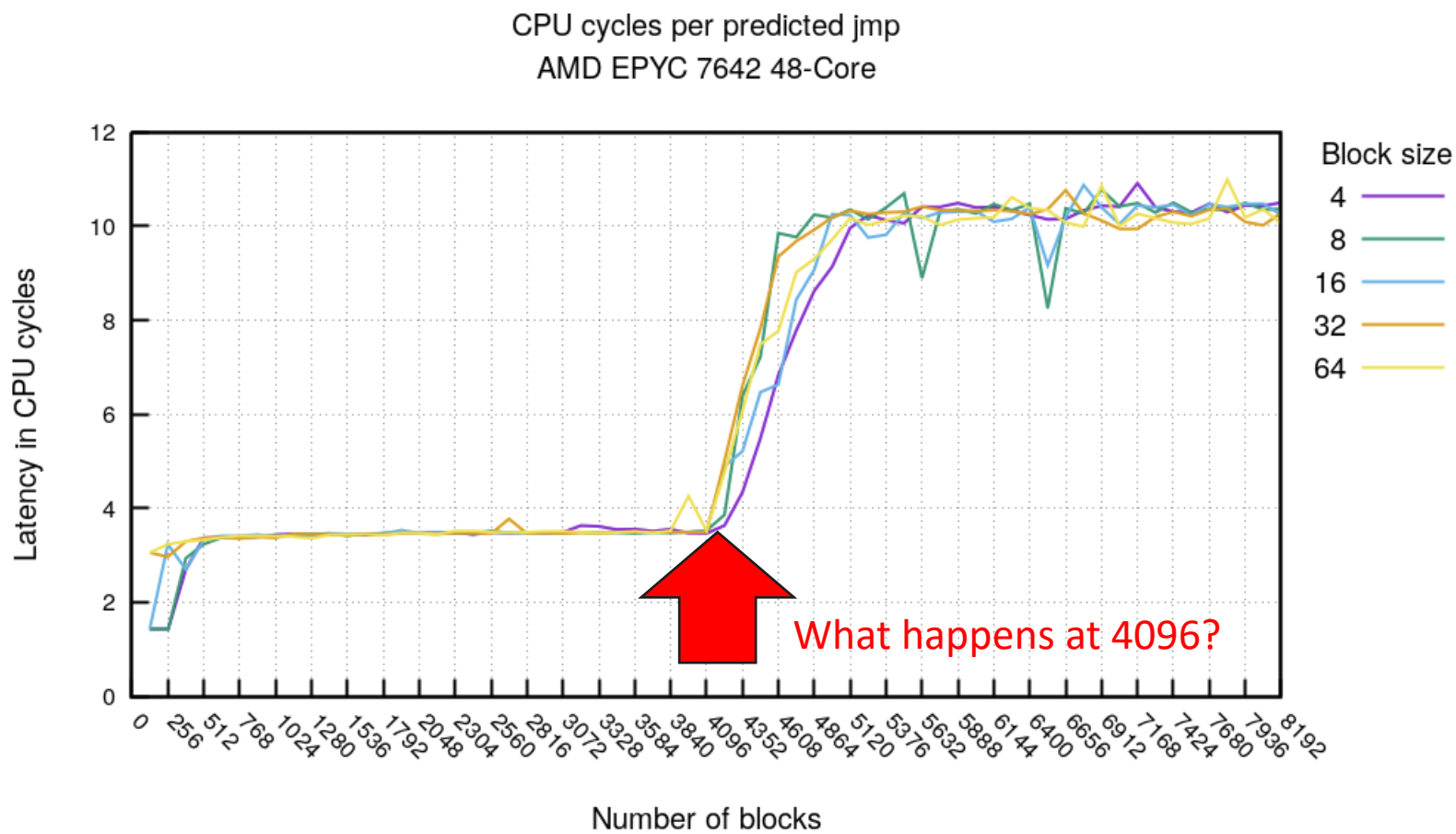
Back to the three questions

- ❑ Is it a branch instruction?
 - Execute updates BHT if it is a branch instruction
- ❑ Is the branch taken?
 - BHT stores if the branch was taken last time
- ❑ Where does the branch go?
 - BTB stores where it went to last time

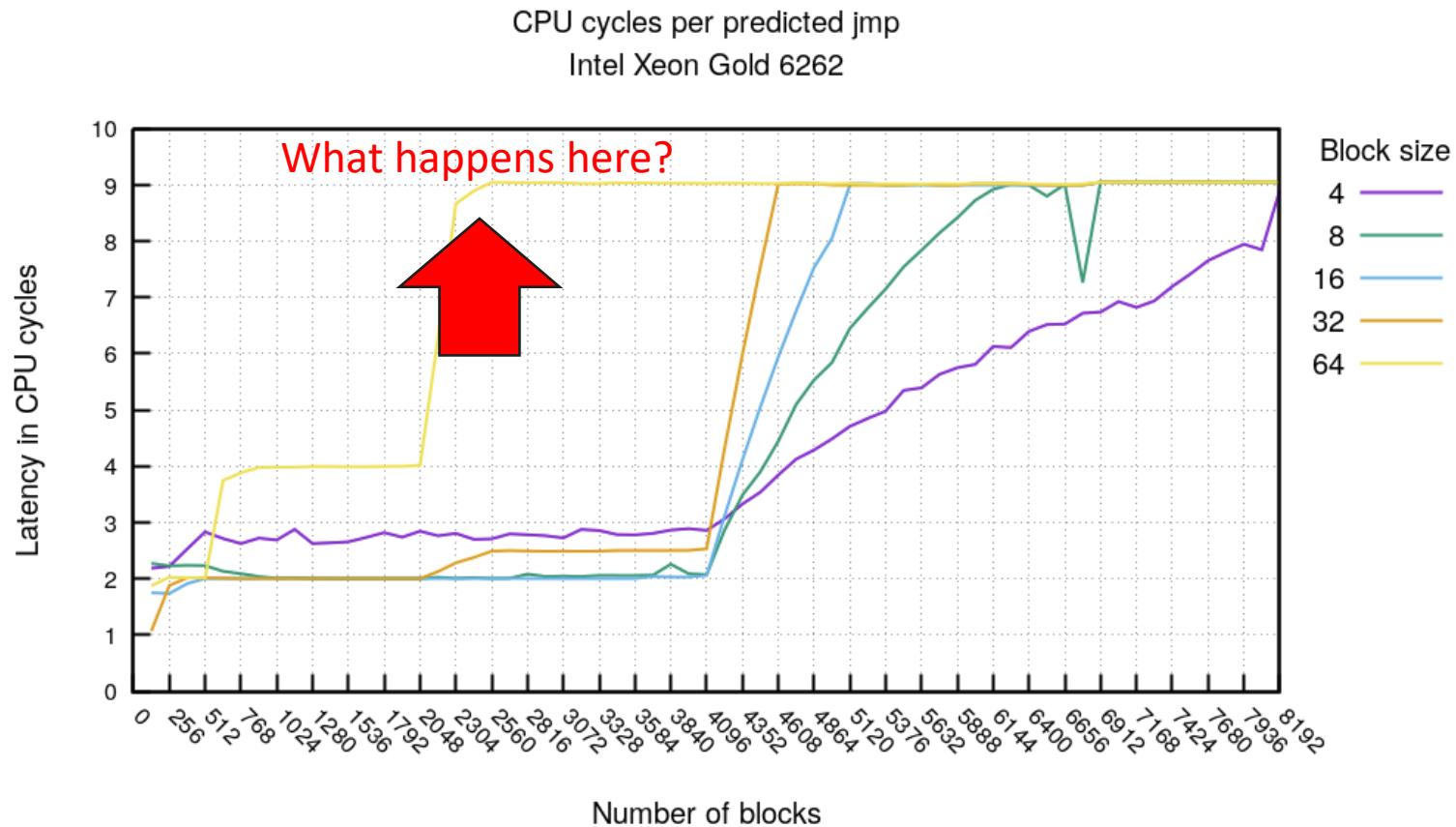
- ❑ Of course, all three are merely predictions!

Impact of branch predictors on performance

```
const char *getCountry(int cc) {  
    if(cc == 1) return "A1";  
    if(cc == 2) return "A2";  
    if(cc == 3) return "O1";  
    if(cc == 4) return "AD";  
    if(cc == 5) return "AE";  
    if(cc == 6) return "AF";  
    if(cc == 7) return "AG";  
    if(cc == 8) return "AI";  
    ...  
    if(cc == 252) return "YT";  
    if(cc == 253) return "ZA";  
    if(cc == 254) return "ZM";  
    if(cc == 255) return "ZW";  
    if(cc == 256) return "XK";  
    if(cc == 257) return "T1";  
    return "UNKNOWN";  
}
```



Impact of branch predictors on performance



Answer:

“Xeon BTB is 8-way set-associative”

-- will re-visit after talking about caches

jmp instructions placed evenly 64 bytes apart
will harm performance...

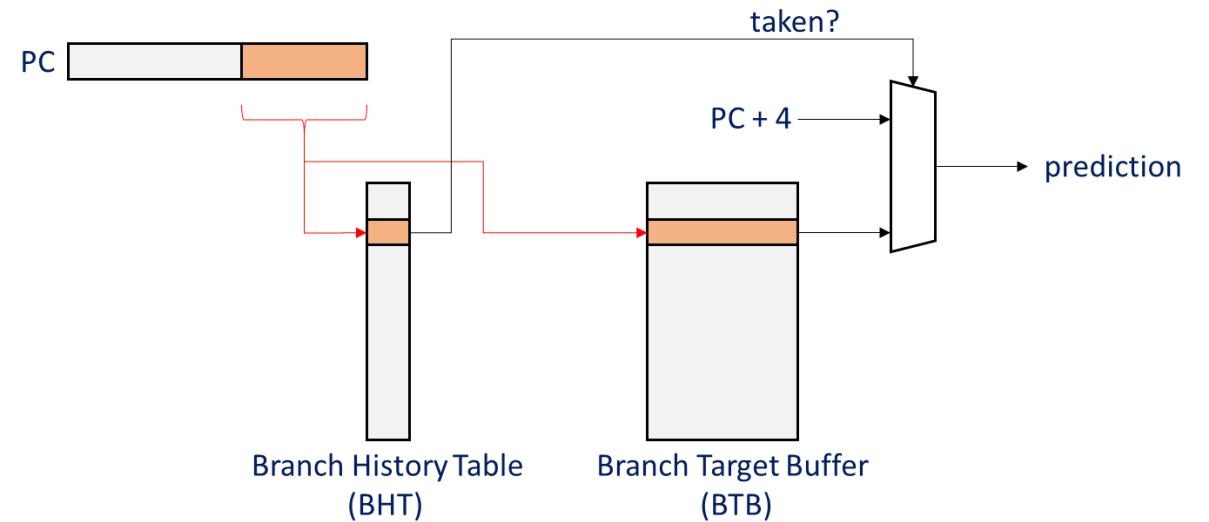
Simple example: 1-bit predictor

❑ BHT has one-bit entries

- Most recently taken/not taken
- (“Last time predictor”)
- Does this work well?

❑ How many mispredicts with these taken (T), not taken (N) sequences?

- TTTTNNNNN TTTTNNNNN
- TNTNTNTN TNTNTNTN
- for (i = 0 ... 2) {
 for (j = 0 ... 2) {
 }
 }
 } Mispredict at j = 0 (T), j = 2 (N)



Simple example: 2-bit predictor

- ❑ BHT has two bits – Single outlier does not change future predictions
 - 00: Strongly not taken, 01: Not taken, 10: Taken, 11: Strongly taken
 - Taken branch increases number, not taken branch decreases number
 - Counter saturates! Taken after 11 -> 11, Not taken after 00 -> 00
- ❑ How many mispredicts with these taken (T), not taken (N) sequences?
 - TTTTNNNN TTTTNNNN
 - TNTNTNTN Initialized to 01: TNTNTNTN
 - for (i = 0 ... 2) { Initialized to 10: TNTNTNTN
 - for (j = 0 ... 2) {
 - }
 - }

Mispredict once at i = 0 && j = 0 (T), j = 2 (N),

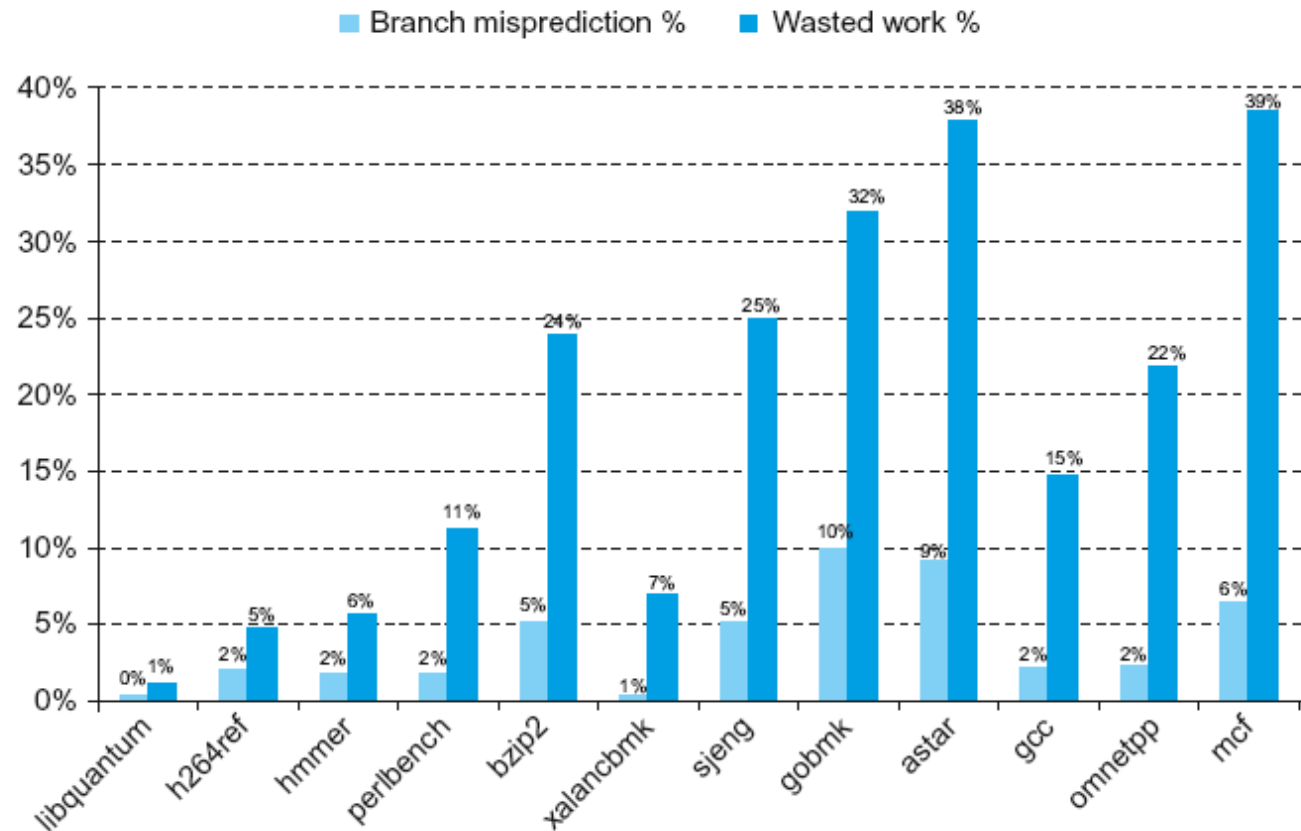
In reality, most SPEC benchmarks record ~90% accuracy with 2-bit predictor

Branch prediction and performance

- ❑ Effectiveness of branch predictors is crucial for performance
 - Spoilers: On SPEC benchmarks, modern predictors routinely have 98+% accuracy
 - Of course, less-optimized code may have much worse behavior
- ❑ Branch-heavy software performance depends on good match between software pattern and branch prediction
 - Some high-performance software optimized for branch predictors in target hardware
 - Or, avoid branches altogether! (Branchless code)

In the real-world: Core i7 performance

- Branch predictors work pretty well!
 - But deep/wide pipelines result in high mispredict overhead



Aside: Impact of branches

“[This code] takes ~12 seconds to run. But on commenting line 15, not touching the rest, the same code takes ~33 seconds to run.”

“(running time may vary on different machines, but the proportion will stay the same).”

```
11     for (int c = 0; c < arraySize; ++c)
12         data[c] = rnd.nextInt() % 256;
13
14     // With this, the next loop runs faster
15     Arrays.sort(data);
16
17     // Test
18     long start = System.nanoTime();
19     long sum = 0;
20
21     for (int i = 0; i < 100000; ++i) {
22         // Primary loop
23         for (int c = 0; c < arraySize; ++c) {
24             if (data[c] >= 128)
25                 sum += data[c];
26         }
27     }
28
29     System.out.println((System.nanoTime() - start) / 1000000000.0);
30     System.out.println("sum: " + sum);
```

Aside: Impact of branches

```
for (int i = 0 ; i < len ; i++) {  
    if (nums[0][i] * nums[1][i] != 0) {  
        arbitrary++;  
    }  
    /* Slower because it involves two branches  
    if (nums[0][i] != 0 && nums[1][i] != 0) {  
        arbitrary++;  
    }  
    */  
}
```

Aside: Branchless programming

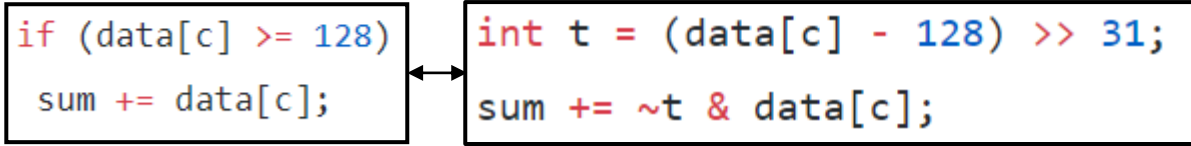
```
// Branch - Random
seconds = 10.93293813

// Branch - Sorted
seconds = 5.643797077

// Branchless - Random
seconds = 3.113581453

// Branchless - Sorted
seconds = 3.186068823
```

```
11  for (int c = 0; c < arraySize; ++c)
12      data[c] = rnd.nextInt() % 256;
13
14  // With this, the next loop runs faster
15  Arrays.sort(data);
16
17  // Test
18  long start = System.nanoTime();
19  long sum = 0;
20
21  for (int i = 0; i < 100000; ++i) {
22      // Primary loop
23      for (int c = 0; c < arraySize; ++c) {
24          if (data[c] >= 128)
25              sum += data[c];
26      }
27  }
28
29  System.out.println((System.nanoTime() - start) / 1000000000.0);
30  System.out.println("sum: " + sum);
```



CS250P: Computer Systems Architecture

Performance Profiling with PerfTools



Sang-Woo Jun

How To Evaluate Our Approaches?

- ❑ Say, we made a performance engineering change in our program
 - ...And performance decreased by 10%
 - Why? Can we know?
- ❑ Many tools provide profiling capabilities
 - gprof, OProfile, Valgrind, VTune, PIN, ...
- ❑ We will talk about perf, part of perf tools
 - Native support in the Linux kernel
 - Straightforward PMC (Performance Monitoring Counter) support

Aside:

Performance Monitoring Counters (PMC)

- ❑ Problem: How can we measure architectural events?
 - L1 cache miss rates, branch mis-predicts, total cycle count, instruction count, ...
 - No way for software to know
 - Events happen too often for software to be counting them
- ❑ Solution: PMCs (Sometimes called Hardware Performance Counters)
 - Dozens of special registers that can each be programmed to count an event
 - Privileged registers, only accessible by kernel
 - Supported PMCs differ across models and designs
- ❑ Usage
 - Program PMC, read PMC, run piece of code, read PMC, compare read values

Linux Perf

- ❑ Performance analysis tool in Linux
 - Natively supported by kernel
 - Supports profiling a VERY wide range of events: PMC to kernel events
 - Note: needs sudo to do most things
- ❑ Many operation modes: top, stat, record, report, ...
 - Supported events found in “sudo perf list”

```
List of pre-defined events (to be used in -e):
```

```
branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]
bus-cycles [Hardware event]
cache-misses [Hardware event]
cache-references [Hardware event]
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
ref-cycles [Hardware event]
```

```
⋮
page-faults OR faults [Software event]
task-clock [Software event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-loads [Hardware cache event]
L1-dcache-stores [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
LLC-load-misses [Hardware cache event]
LLC-loads [Hardware cache event]
```

Linux Perf: Stat

❑ Default command prints some useful information

○ “sudo perf stat ls”

❑ More events can be traced using -e

○ sudo perf stat -e task-clock,page-faults,cycles,instructions,branches,branch-misses,LLC-loads,LLC-load-misses ls

Performance counter stats for 'ls':

0.652008	task-clock (msec)	#	0.805 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
104	page-faults	#	0.160 M/sec
2,797,861	cycles	#	4.291 GHz
2,245,082	instructions	#	0.80 insn per cycle
444,095	branches	#	681.119 M/sec
16,749	branch-misses	#	3.77% of all branches

0.000810402 seconds time elapsed

Performance counter stats for 'ls':

0.681485	task-clock (msec)	#	0.781 CPUs utilized
102	page-faults	#	0.150 M/sec
2,921,152	cycles	#	4.286 GHz
2,217,325	instructions	#	0.76 insn per cycle
439,589	branches	#	645.046 M/sec
16,608	branch-misses	#	3.78% of all branches
9,736	LLC-loads	#	14.286 M/sec
3,269	LLC-load-misses	#	33.58% of all LL-cache hits

0.000872088 seconds time elapsed

Linux Perf: Record, Report

- ❑ Log events with “record”, interactively analyze it with “report”
 - `sudo perf record -e cycles,instructions,L1-dcache-loads,L1-dcache-load-misses [...]`
 - Creates “perf.data”
- ❑ “sudo perf report” reads “perf.data”

```
Available samples
8 cycles
8 instructions
8 L1-dcache-loads
7 L1-dcache-load-misses
```



```
Samples: 8 of event 'cycles', Event count (approx.): 2476964
Overhead Command Shared Object Symbol
96.20% tail [kernel.kallsyms] [k] memcpy_erms
3.80% perf [kernel.kallsyms] [k] perf_event_addr_filters_exec
0.00% perf [kernel.kallsyms] [k] native_write_msr
0.00% tail [kernel.kallsyms] [k] native_write_msr
```

This is where most cycles are spent!

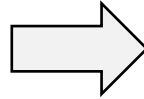
```
Samples: 7 of event 'L1-dcache-load-misses', Event count (approx.): 36818
Overhead Command Shared Object Symbol
86.85% tail [kernel.kallsyms] [k] copy_page
12.36% perf [kernel.kallsyms] [k] perf_iterate_ctx
0.74% perf [kernel.kallsyms] [k] perf_event_addr_filters_exec
0.04% perf [kernel.kallsyms] [k] native_write_msr
0.00% tail [kernel.kallsyms] [k] native_write_msr
```

This is where most L1 cache misses are!

Loop unrolling: A compiler solution to branch hazards

```
for ( i = 0 to 15 ) foo();
```

Loop unrolling



```
for ( i = 0 to 3 ) {  
    foo();  
    foo();  
    foo();  
    foo();  
}
```

Potentially **16** branch mispredicts
Even without mispredicts,
branch instruction consume **16** cycles

Potentially **4** branch mis-predicts
Without mis-predicts,
branch instruction consume **4** cycles

We can do this manually, or tell the compiler to do its best

- GCC flags `-funroll-loops`, `-funroll-all-loops`
- How much to unroll depends on heuristics within compiler

Code example: Counting numbers

□ How fast is the following code?

- a and b are initialized to `rand()%256`
- cnt is 100,000,000
- Compiled with GCC `-O3`

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) cnt++;  
}
```

□ This code takes 0.44s on my desktop (i5 @ 3 GHz)

- Each loop takes 13.2 cycles ($3 \text{ GHz} * 0.44 / 100,000,000$)
- Can we do better? *My x86 is 4-way superscalar!*

Optimization attempt #1: Loop unrolling

- ❑ There are three potential branch instruction locations
 - “i < cnt”, “a[i] < 128”, and b[i] < 128”
- ❑ Is the bottleneck the “for” loop?
 - Let’s try giving -funroll-all-loops

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) lcnt++;  
}
```

- ❑ Performance increased from 0.44s to ~0.43s.
 - Better, but not by much

Identifying the bottleneck

- ❑ We predict the “if” statements are the bottlenecks
 - Each of the two branch instructions has a 50% chance of being taken
 - Branch prediction very inefficient!

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 && b[i] < 128 ) lcnt++;  
}
```

- ❑ Performance improves when comparison becomes skewed
 - 0.44s when comparing against 128 (50%)
 - 0.27s when comparing against 64 (25%), 0.17s with 32

Optimization attempt #2: Branchless code

- ❑ Let's try getting rid of the "if" statement. How?
- ❑ Some knowledge of architectural treatment of numbers is required
 - x86 represents negative numbers via two's complement
 - "1" == 0x1, "-1" == 0xffffffff
 - "1>>31" == 0x0, "-1>>31" == 0xffffffff
- ❑ "(v-128)>>31"
 - if v >= 128: 0x0
 - v < 128: 0xffffffff

So many more instructions! Will this be faster?

```
for ( int i = 0; i < cnt; i++ ) {  
>  lcnt += ( (((a[i] - 128)>>31)&1) * (((b[i] - 128)>>31)&1) );  
}
```

Comparing Performance Numbers

Name	Elapsed (s)
Vanilla	0.44 s
Branchless	0.06 s

Branch predictor is almost always correct



Vanilla: Total misses: 57 M out of 3,623 M

Overhead	Command	Shared Object	Symbol
87.38%	a.out	a.out	[.] main
9.80%	a.out	libc-2.27.so	[.] __random
1.48%	a.out	libc-2.27.so	[.] __random_r
0.36%	a.out	[kernel.kallsyms]	[k] __pagevec_lru_add_fn
0.29%	a.out	[kernel.kallsyms]	[k] get_page_from_freelist

~2 cycles per loop! 8 Operations with 4 way superscalar...

Branchless: Total misses: 7 M out of 3,514 M

Over 7x performance!

Overhead	Command	Shared Object	Symbol
77.47%	a.out	libc-2.27.so	[.] __random
10.13%	a.out	libc-2.27.so	[.] __random_r
3.12%	a.out	[kernel.kallsyms]	[k] get_page_from_freelist
2.86%	a.out	[kernel.kallsyms]	[k] __pagevec_lru_add_fn
1.78%	a.out	[kernel.kallsyms]	[k] __handle_mm_fault
0.74%	a.out	a.out	[.] main
0.70%	a.out	libc-2.27.so	[.] rand

Interestingly, loop with only one comparator is automatically optimized by compiler

```
for ( int i = 0; i < cnt; i++ ) {  
>   if ( a[i] < 128 ) lcnt ++;  
}
```

Shows same performance as the branchless one

CS250P: Computer Systems Architecture

Achieving Correct Pipelining

-- Superscalar



Sang-Woo Jun

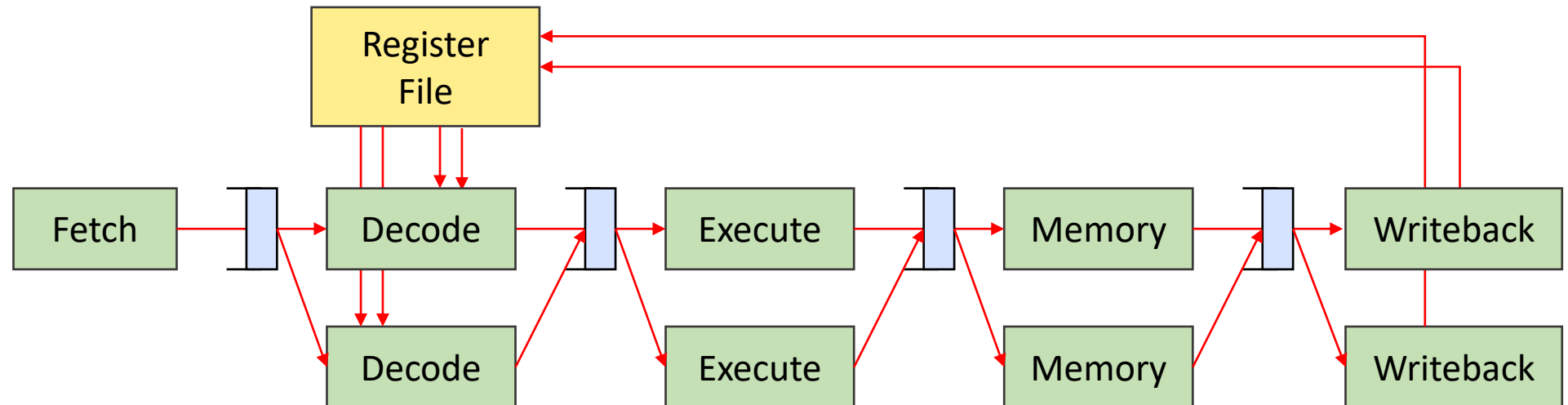
Fall 2023

Superscalar Processing

- ❑ An ideally pipelined processor can handle up to one instructions per cycle
 - Instructions Per Cycle (IPC) = 1, Cycles Per Instruction (CPI) = 1
- ❑ Superscalar wants to process multiple instruction per cycle
 - $IPC > 1$, $CPI < 1$
 - An N-way superscalar processor handles N instructions per cycle
 - Requires multiple pipeline hardware instances/resources
 - Hardware performs dependency checking on-the-fly between concurrently-fetched instructions

Pipeline for superscalar processing

- ❑ Multiple copies of the datapath supports multiple instructions/cycle
- ❑ Register file needs many more ports
- ❑ Actually requires a complex scheduler in the decode stage!



Superscalar has concurrent hazards

- ❑ What if two concurrently issued instructions have dependencies?
 - No choice but to stall the dependent instruction...
 - ... in an in-order pipeline! ← Topic for another day
- ❑ Data hazards
 - e.g., “addi s1, s0, 1” and “addi s2, s1, 1” issued at the same time?
 - Forwarding won't work here! Both instructions in decode stage at the same time
 - Scheduler must stagger “addi s2, s1, 1”, sacrificing performance
- ❑ Control hazards
 - e.g., How to handle a beq, followed by another instruction?
 - Branch prediction, as usual

Results in very complex control logic! (Chip resources/cost!)

In-order superscalar example

Ideal IPC = 2 (2-Way superscalar)

lw t0, 40(\$s0)

add t1, \$s1, \$s2

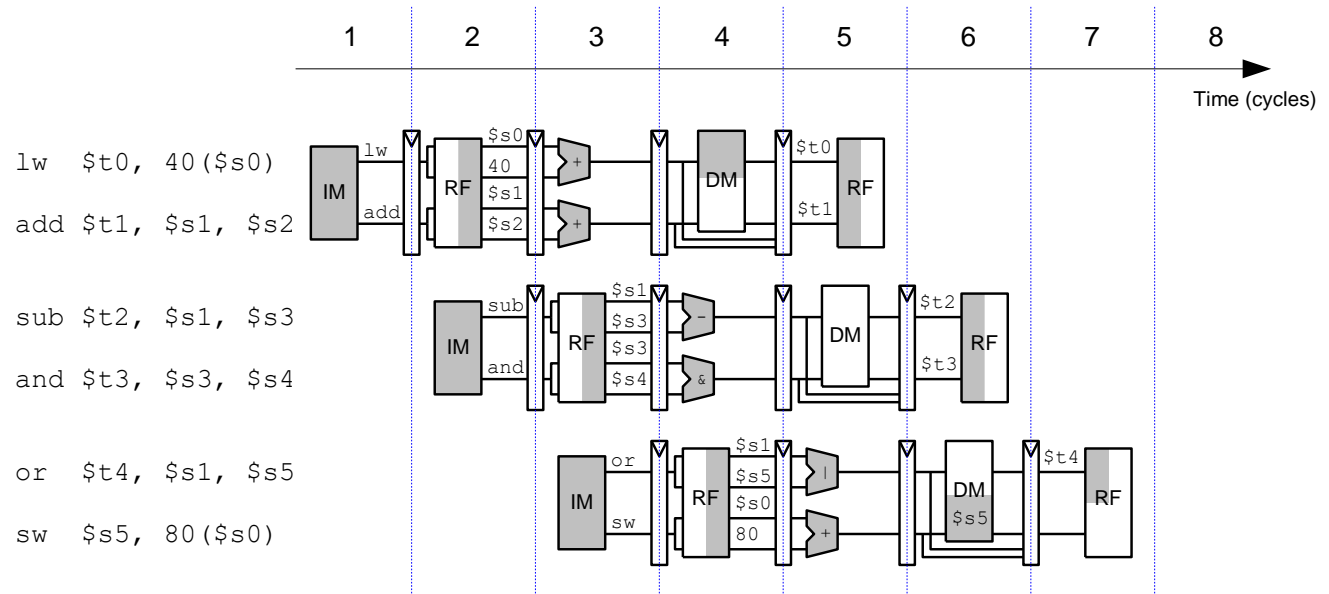
sub t2, s1, s3

and t3, s3, s4

or t4, s1, s5

sw s5, 80(\$s0)

No dependencies between
any instructions



Actual IPC = 2 (6 instructions issued in 3 cycles)

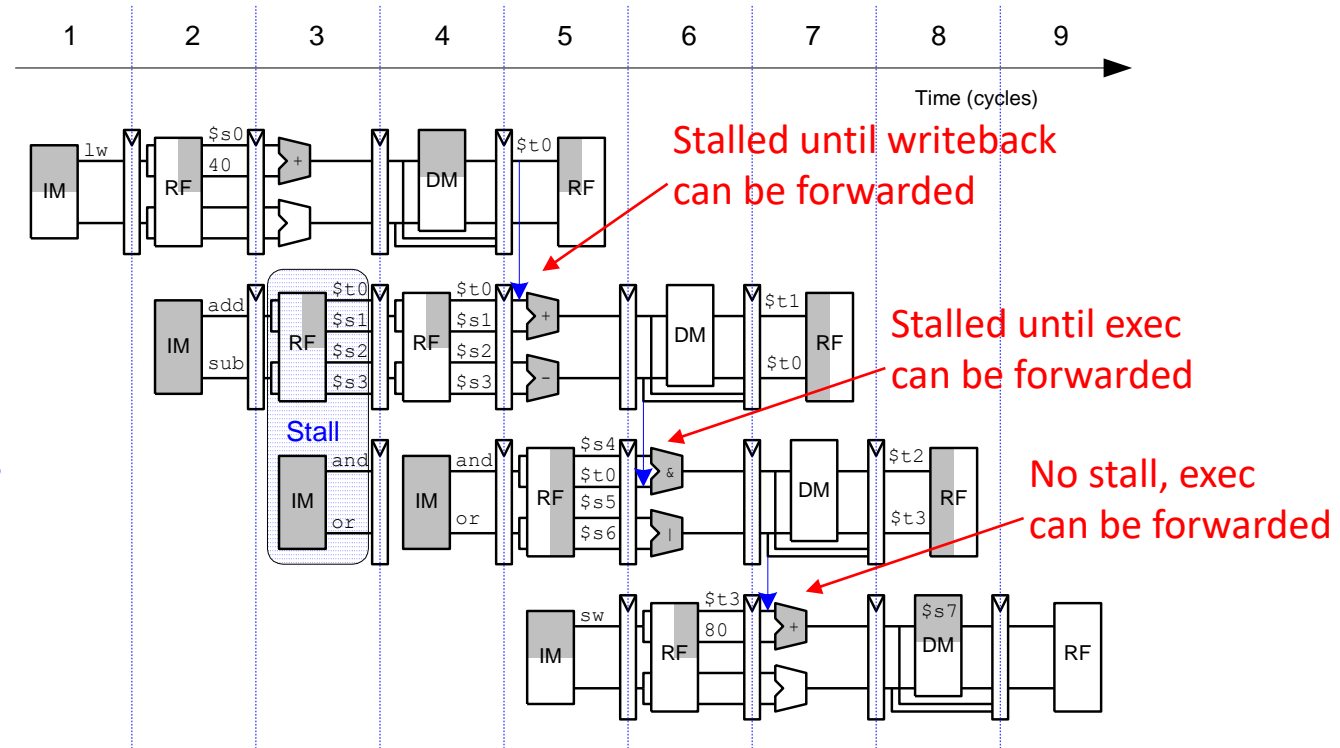
In-order superscalar with dependencies

Ideal IPC = 2 (2-Way superscalar)

```
lw t0, 40(s0)
add t1, t0,$s1
sub t0, s2, s3
and t2, s4, t0
or t3, s5, s6
sw s7, 80(t3)
```

Dependencies across many instructions!

```
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

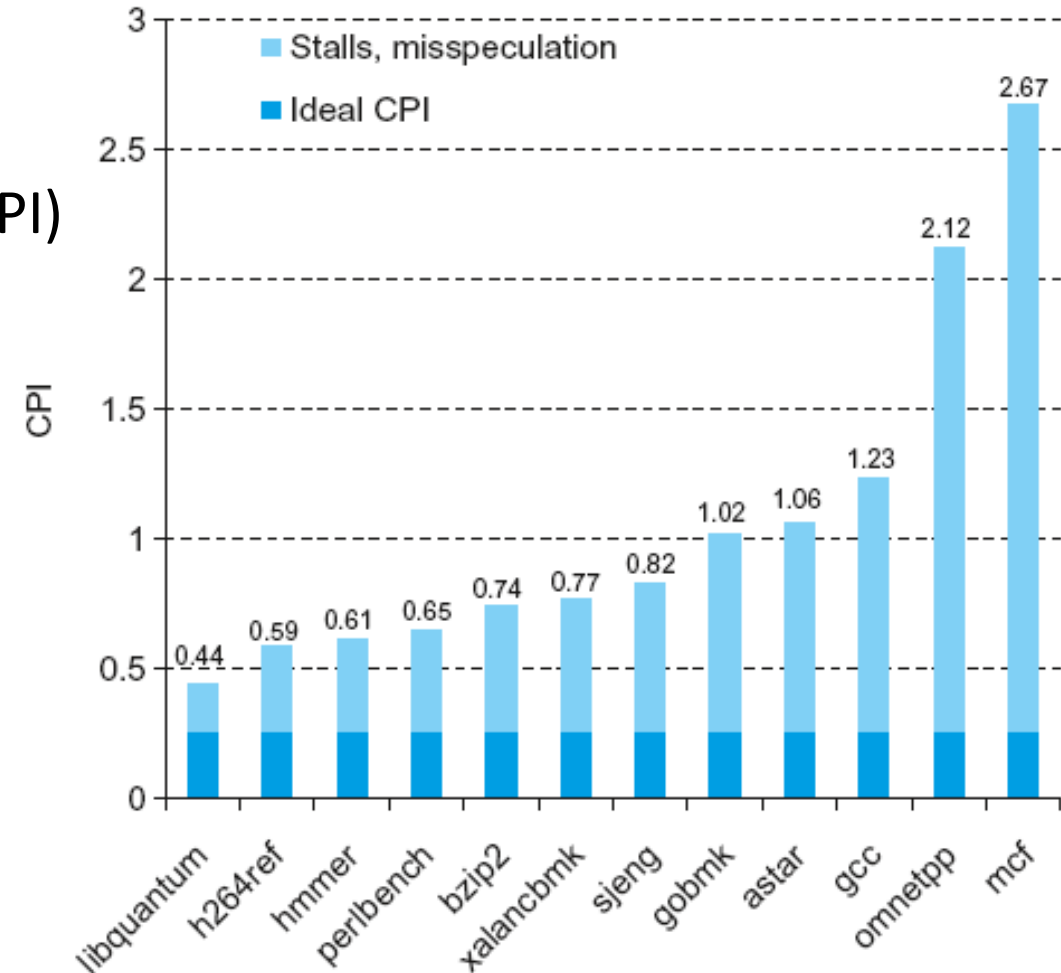


Actual IPC = 1.2 (6 instructions issued in 5 cycles)

In the real-world: Core i7 performance

- ❑ Core i7 has a 4-way *Out-of-Order* Superscalar pipeline
 - Ideally, 0.25 Cycles Per Instruction (CPI)
 - Dependencies and misprediction typically results in much lower performance

Is it worth it? Or do we want just more, simpler cores?
Depends on your target area (servers? phones?) and profiling results...



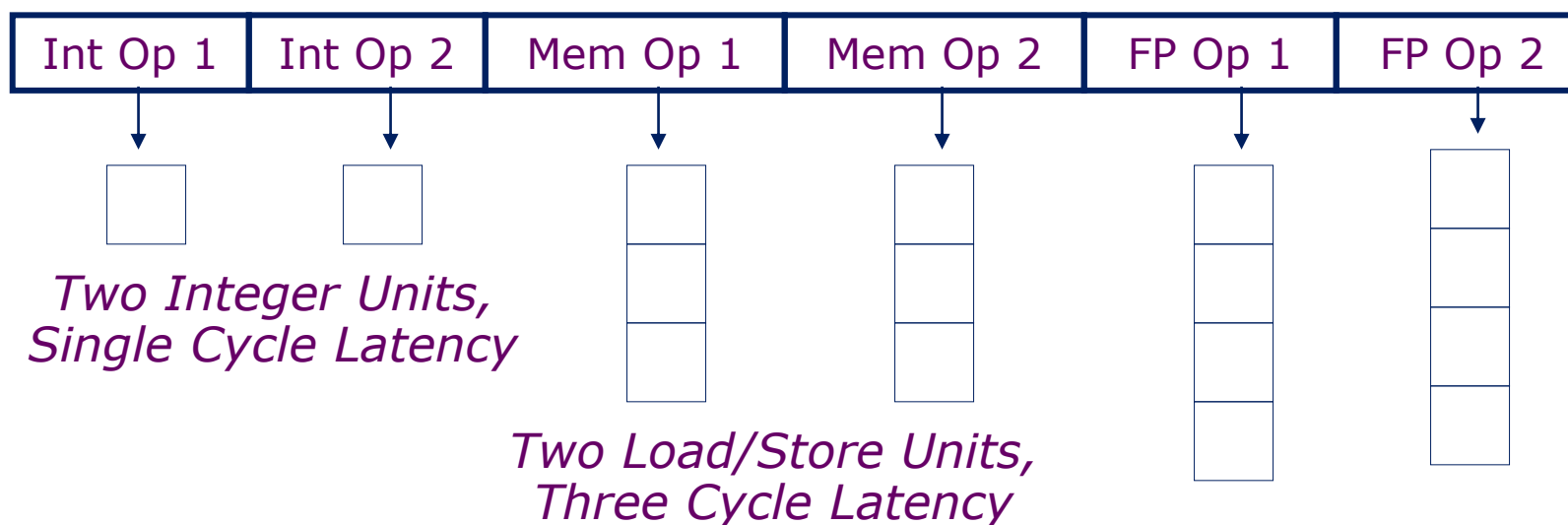
Very Long Instruction Word (VLIW)

- ❑ Superscalar does not change the ISA
 - Complicates hardware in charge of detecting dependencies!
- ❑ What if we changed the ISA, and made the compiler manage ILP?

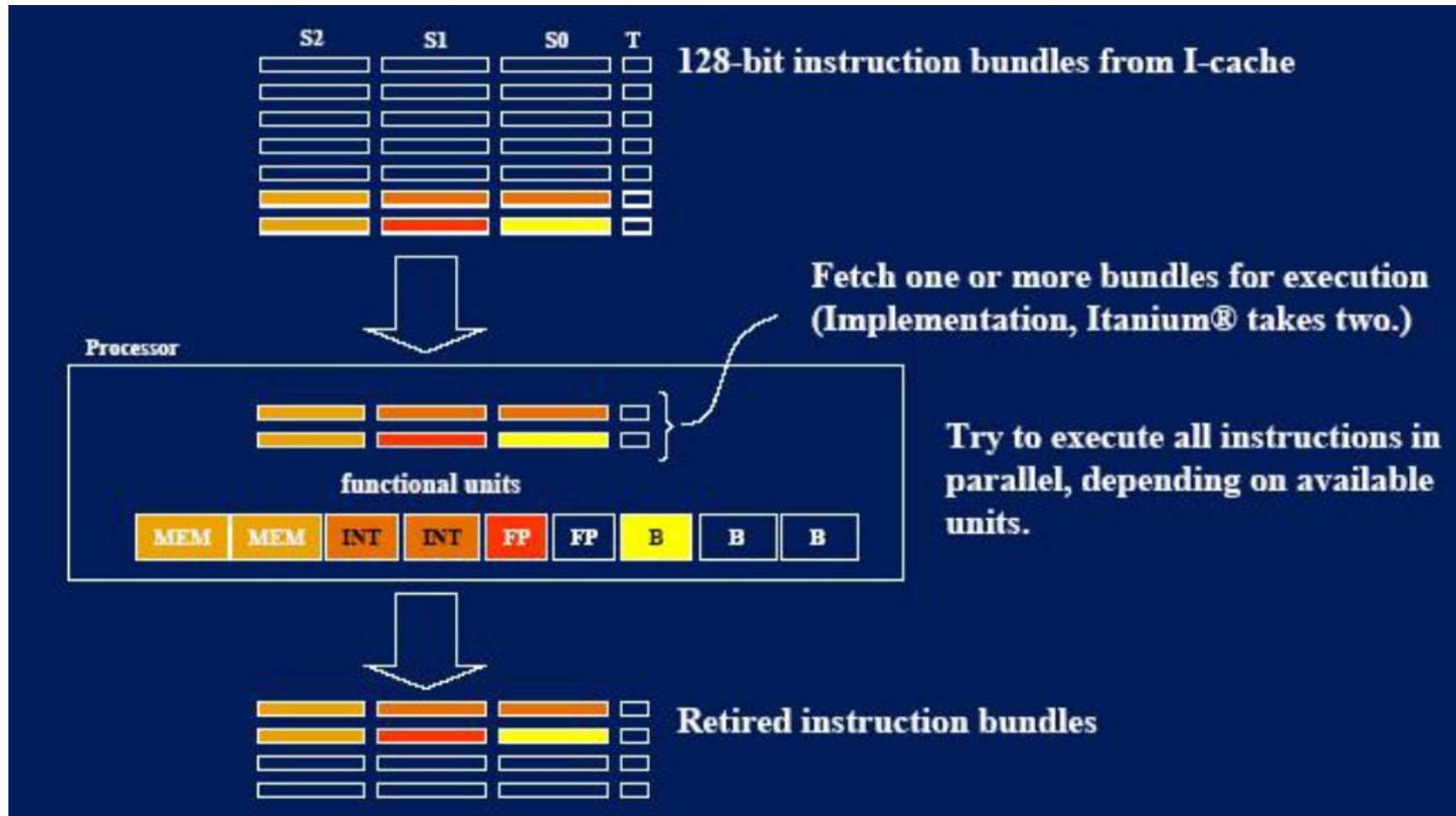
- ❑ Not in x86/RISC-V/ARM/...
 - Sometimes as accelerator extensions!
 - (RISC-V “V” extension)

Very Long Instruction Word (VLIW)

- ❑ Multiple instructions packaged into a Very Long Instruction
 - Sometimes “bundle”
- ❑ Each execution operation slot has a fixed function (ALU, Mem, FP, etc)
- ❑ Compiler’s responsibility to create efficient instructions
 - Inter-slot dependency is not checked by hardware!

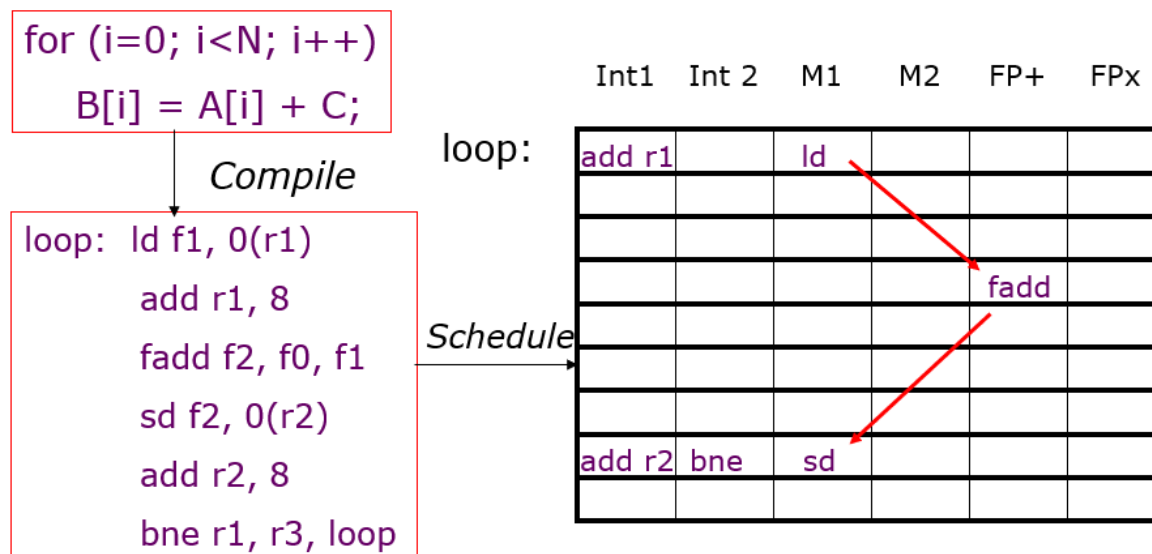


Intel Explicitly Parallel Instruction Computing (EPIC, Itanium)



VLIW Characteristics

- ❑ Very good performance for computation-intensive code
- ❑ Very bad performance for code with many dependencies/hazards!
 - Much more sensitive to hazards than single-issue pipelines
 - Example: short loops



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

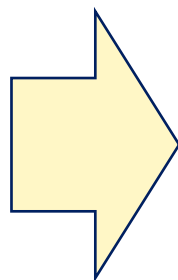
Compiler's job is important!

- e.g., Loop unrolling to keep execution units busy

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]    = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```



Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

Schedule

loop:

	Int1	Int 2	M1	M2	FP+	FPx
			ld f1			
			ld f2			
			ld f3			
add r1			ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
add r2	bne	sd f8				

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

Issues with VLIW

- ❑ Execution unit configurations change across models
 - How many Integer units, how many float units, neural units ...?
 - Cannot be binary compatible across models!
 - Unless hardware provides an abstraction layer...?
 - But that would add scheduler overhead, undermining VLIW (Itanium tried a good balance)
- ❑ Dependency/hazards difficult for compiler to manage
 - Too many slots end up empty (low performance, large binary)
- ❑ But when it works well, it works remarkably well
 - e.g., Scientific computing
 - That's why it is often resurrected as potential solution (Itanium, ATi TeraScale, ...)

CS250P: Computer Systems Architecture

Out of Order Processing



Sang-Woo Jun

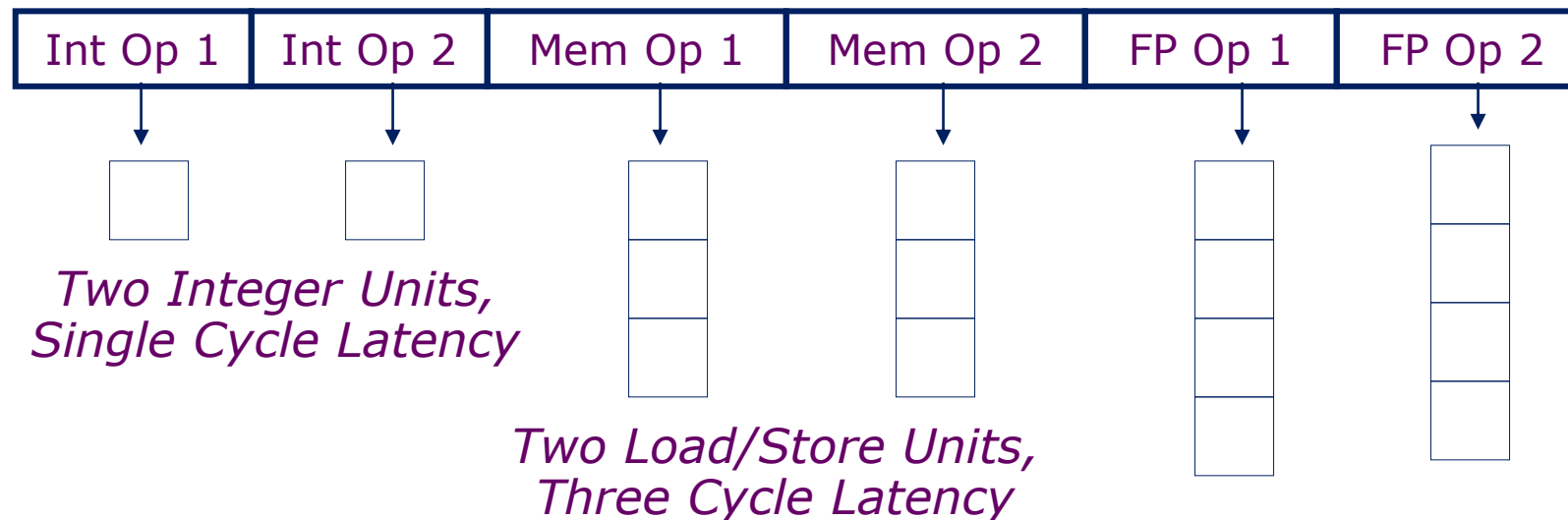
Fall 2023



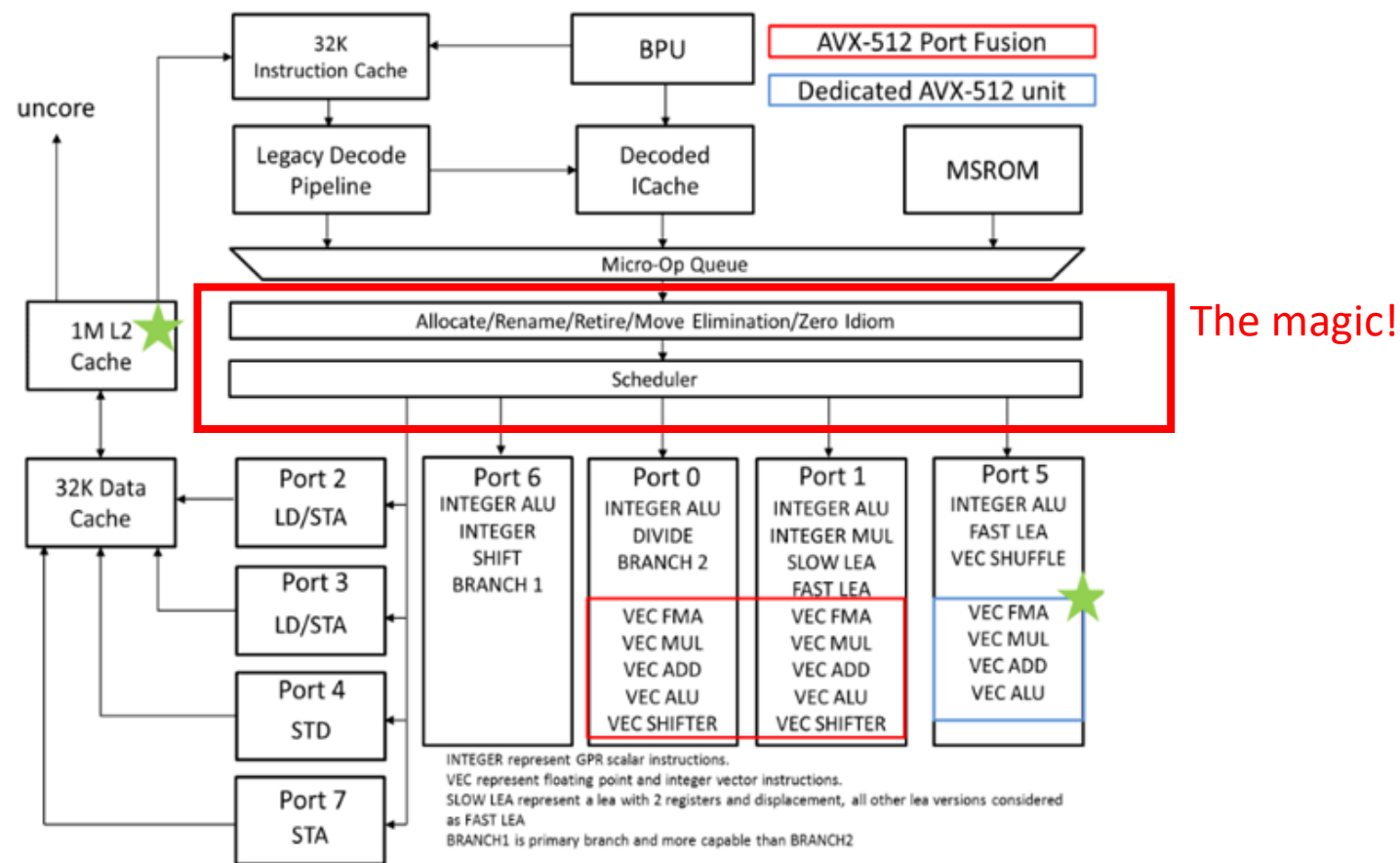
Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Back to Transparent Parallelism

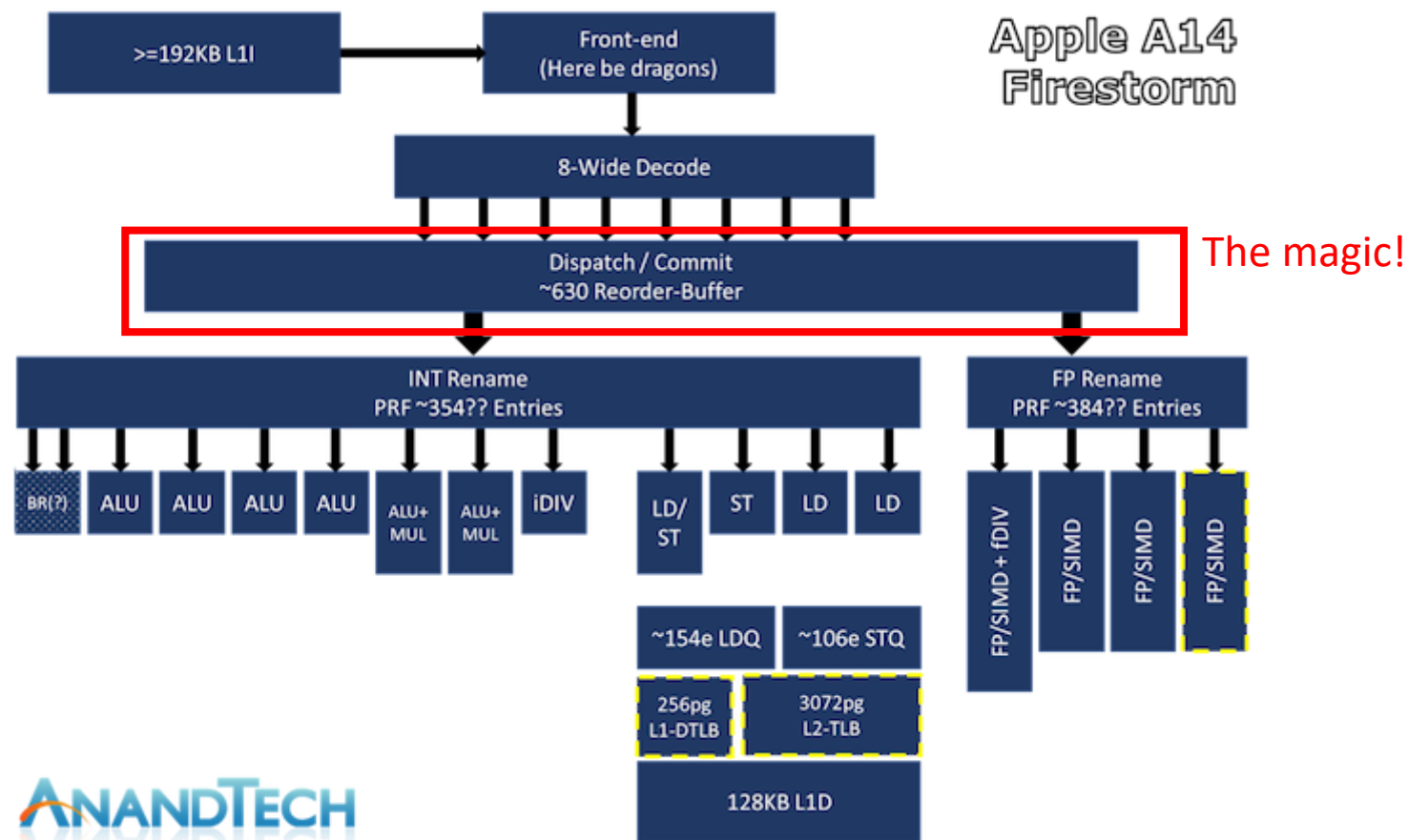
- ❑ Explicit parallelism is not as popular as transparent
 - Everyone wants performance for free!
- ❑ Can we keep execution slots busy, using backwards-compatible single-thread instruction streams?



Skylake-X Microarchitecture (2019)



Apple M1 Microarchitecture (2020)



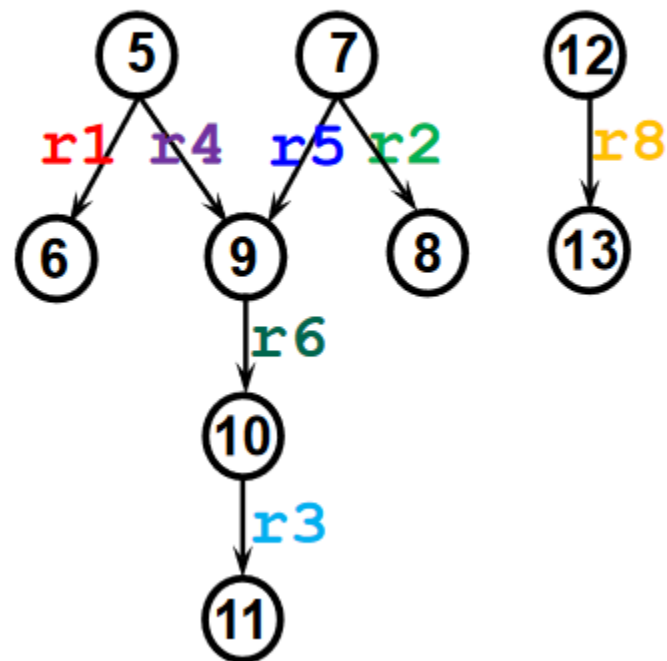
OoO: Determining dependencies

(1) **r8** ← 20
(2) **r1** ← addr1
(3) **r2** ← addr2
(4) **r3** ← addr3

LOOP:

(5) **r4** ← MEM[**r1**]
(6) **r1** ← **r1** + 4
(7) **r5** ← MEM[**r2**]
(8) **r2** ← **r2** + 4
(9) **r6** ← **r4** + **r5**
(10) MEM[**r3**] ← **r6**
(11) **r3** ← **r3** + 4
(12) **r8** ← **r8** - 1
(13) **bnz r8, LOOP**

Instruction dependence graph



Data dependency types: RaW

□ Read-after-Write

- A “true” dependency
- We must wait until r5’s value is materialized... No other choice

(7) $r5 \leftarrow \text{MEM}[r2]$

(9) $r6 \leftarrow r4 + r5$

LOOP:

(5) $r4 \leftarrow \text{MEM}[r1]$

(6) $r1 \leftarrow r1 + 4$

(7) $r5 \leftarrow \text{MEM}[r2]$

(8) $r2 \leftarrow r2 + 4$

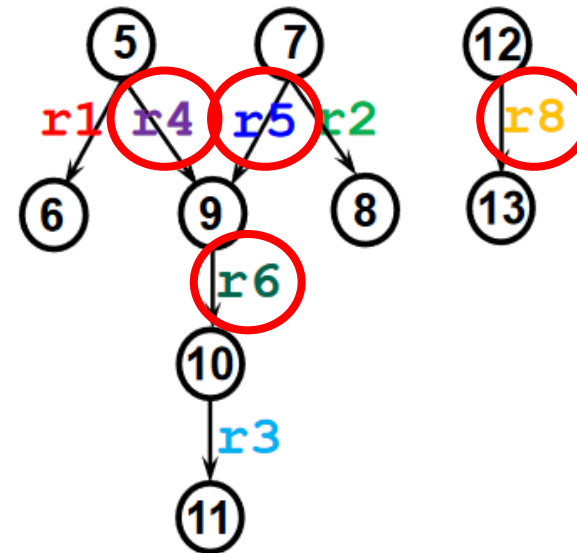
(9) $r6 \leftarrow r4 + r5$

(10) $\text{MEM}[r3] \leftarrow r6$

(11) $r3 \leftarrow r3 + 4$

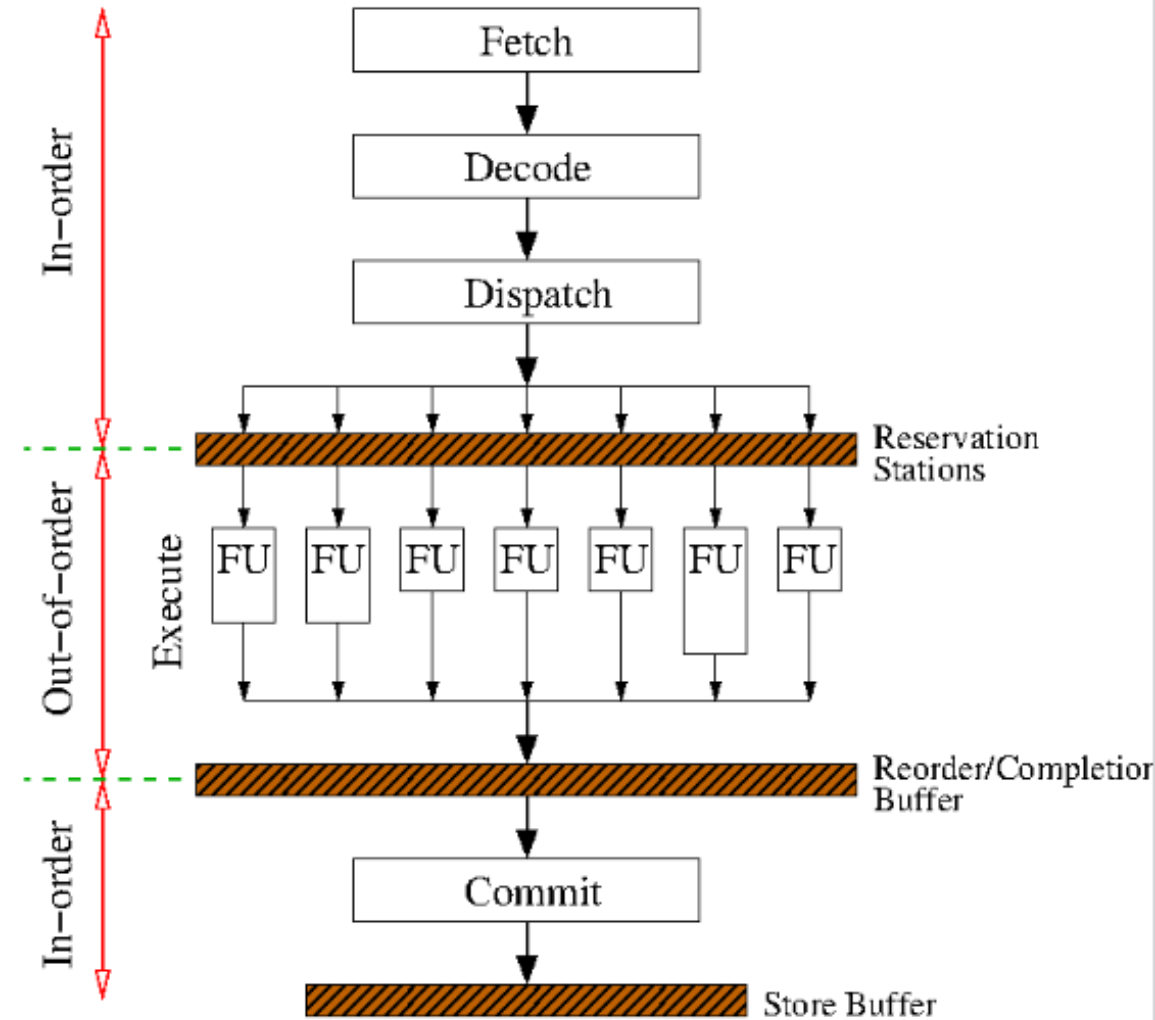
(12) $r8 \leftarrow r8 - 1$

(13) $\text{bnz } r8, \text{ LOOP}$



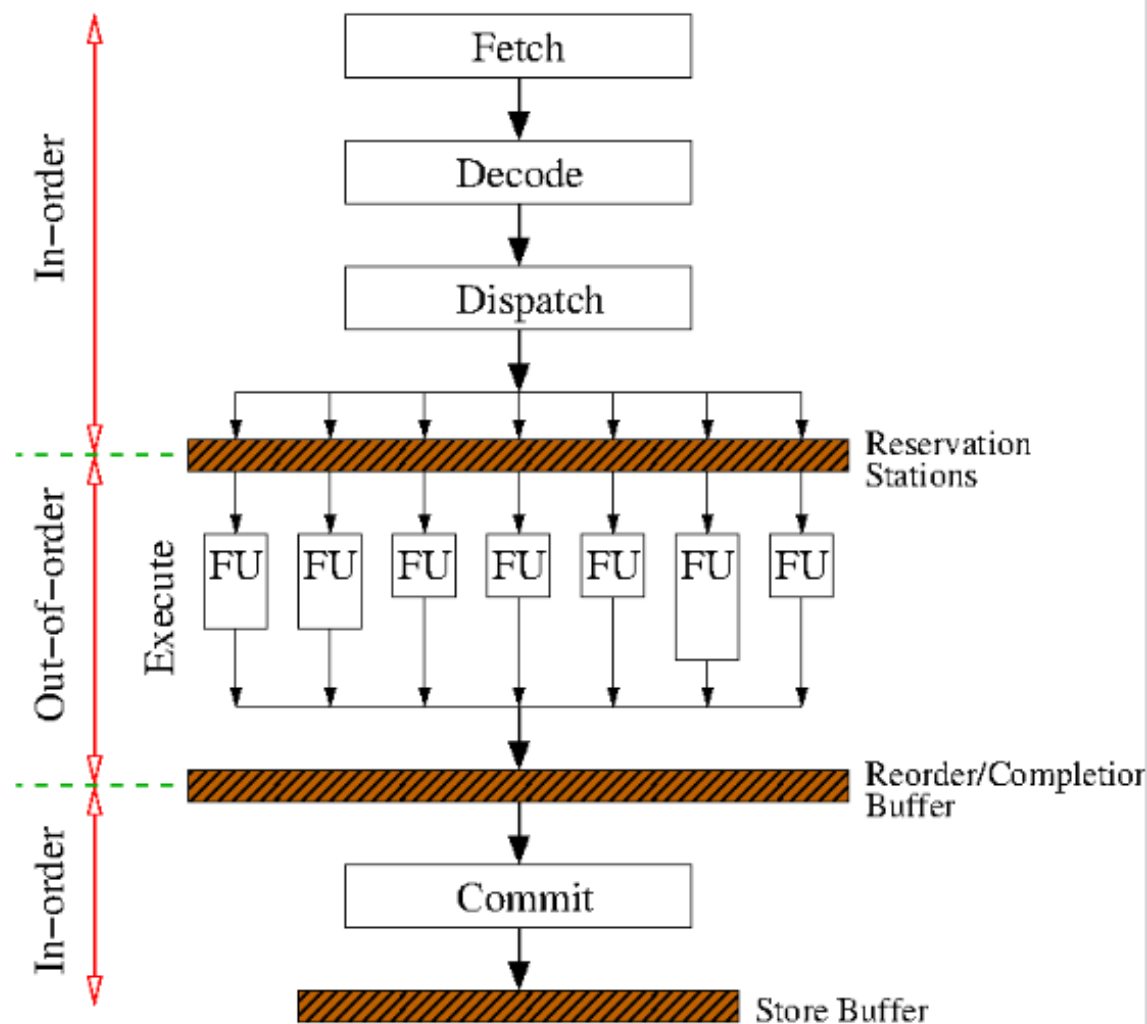
OoO managing dependencies

- ❑ Looks like dispatch+Commit stages added to VLIW
 - Instructions wait at “reservation stations”
 - “Is my input operand being written to”
 - Listens to forwarding paths
 - Out of order



OoO managing dependencies

- ❑ Arithmetic can happen OoO, BUT Commits should happen In-Order!
 - Register writes, memory updates, etc
- ❑ Decoded instructions line up at Reorder Buffer(RoB)
 - Wait until execute results available
 - Wait until branch mispredict ruled out
 - Commits in order of insertion



Many topics we won't go into today!

- ❑ Effectively matching available operands to waiting instructions
 - Looping over instructions is too slow
 - N-to-N broadcast is too expensive (slow clocks!)
 - Tomasulo's algorithm!
- ❑ Precise interrupts become complicated
 - Things are executing OoO, when a breakpoint happens, how do we line things back up for debugging?

Just one more topic: Register renaming

□ Not all dependencies are RaW. Some can be resolved!

○ Write-after-Read (WaR)

- e.g., 5->6
- “Anti-dependence”: r1’s value clobbered after (6)
- If we had used “r9” instead, no dependency!

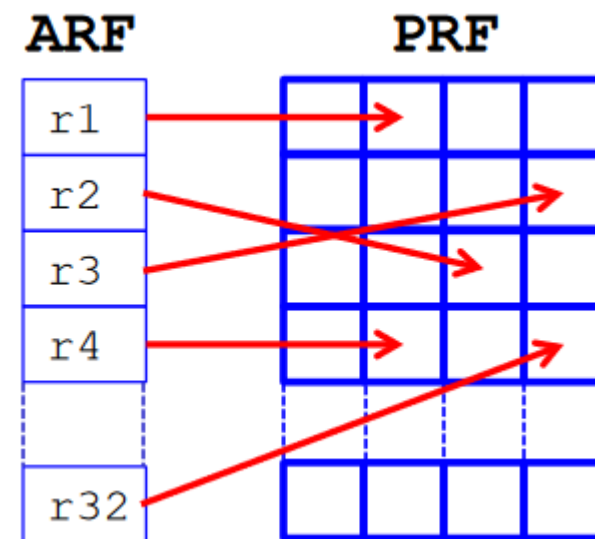
○ Write-after-Write (WaW)

- e.g., 7->7 across loop iteration
- (7) does not read from “r5”...
- If each loop iteration used a different reg, (r9, r10, r11,...) no dependency!

```
(5)  r4 ← MEM[r1]
(6)  r1 ← r1 + 4
(7)  r5 ← MEM[r2]
(8)  r2 ← r2 + 4
(9)  r6 ← r4 + r5
(10) MEM[r3] ← r6
(11) r3 ← r3 + 4
(12) r8 ← r8 - 1
(13) bnz r8, LOOP
```

OoO: Register renaming

- ❑ Two different concepts of registers
 - “Architectural Registers”: Conceptually defined in ISA, software abstraction
 - “RISC-V has 32 registers in the register file”
 - “Physical Registers”: Larger number of registers actually in silicon
 - Scheduler dynamically renames registers to an empty slot in the physical register file
 - When WaW or WaR dependencies are discovered



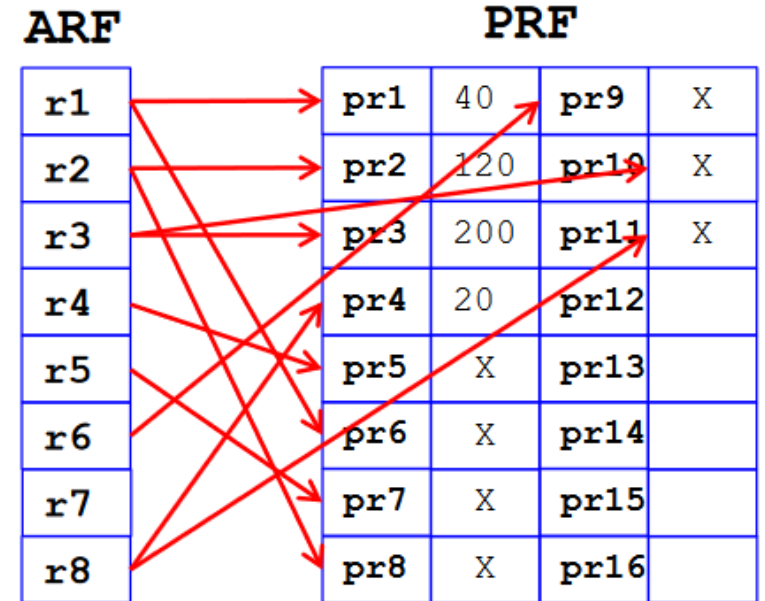
Register renaming: Previous example...

Original code:

```
→ (5) r4 ← MEM[r1]
→ (6) r1 ← r1 + 4
→ (7) r5 ← MEM[r2]
→ (8) r2 ← r2 + 4
→ (9) r6 ← r4 + r5
→ (10) MEM[r3] ← r6
→ (11) r3 ← r3 + 4
→ (12) r8 ← r8 - 1
→ (13) bnz r8, LOOP
```

Post-decode (as seen by RS):

```
(5) pr5 ← MEM[40]
(6) pr6 ← 40 + 4
(7) pr7 ← MEM[120]
(8) pr8 ← 120 + 4
(9) pr9 ← pr5 + pr7
(10) MEM[200] ← pr9
(11) pr10 ← 200 + 4
(12) pr11 ← 20 - 1
(13) bnz pr11, LOOP
```



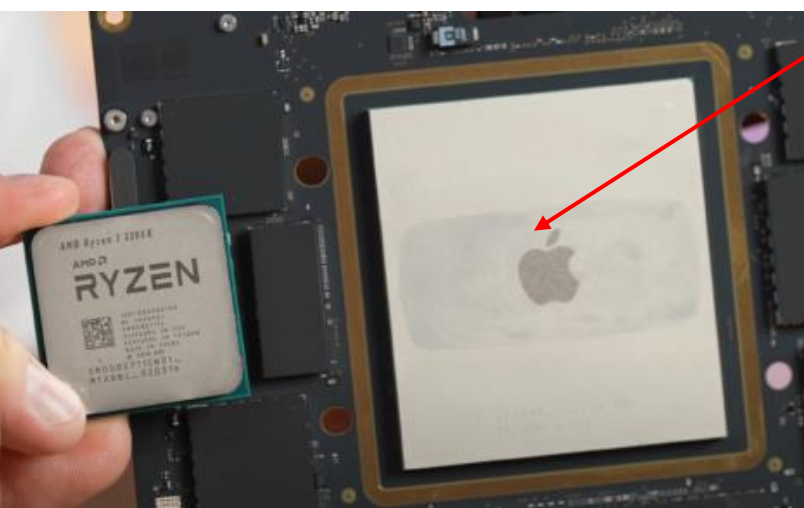
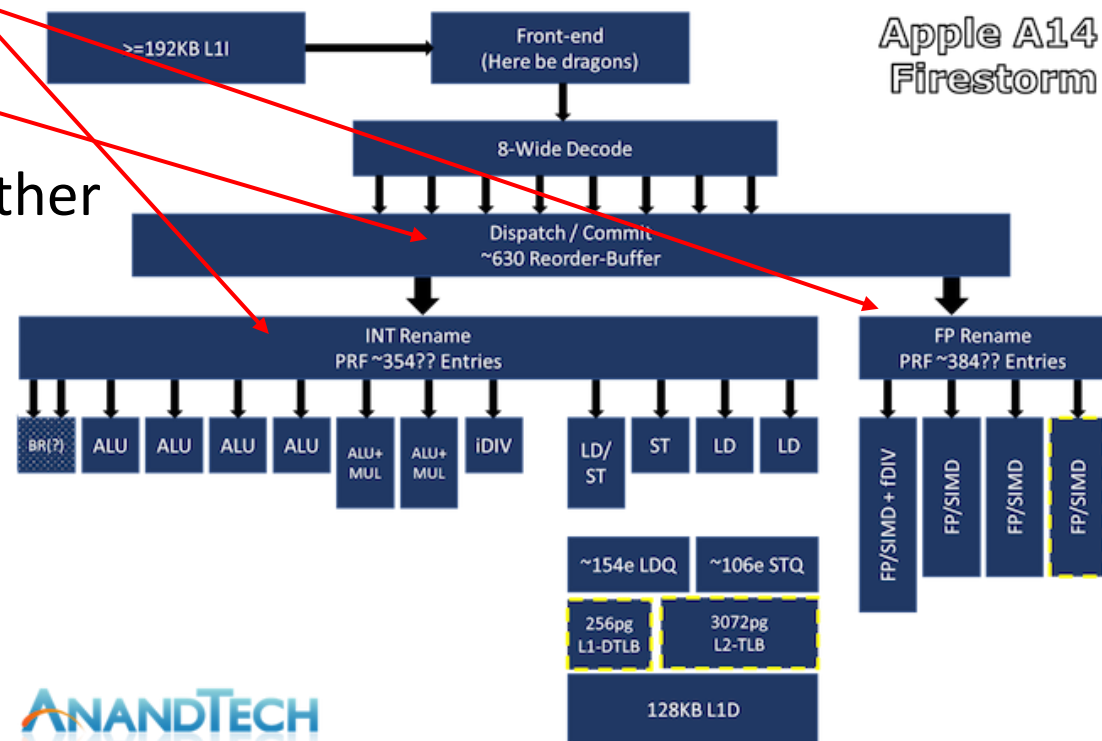
Back to timely example: Apple M1

❑ Really good single-thread performance!

❑ How?

RISC!

- “8-wide decoder” [...] “16 execution units (per core)”
- “(Estimated) 630-deep out-of-order”
- “Unified memory architecture”
- Hardware/software optimized for each other



M1 Ultra
Image source: wccftch

Aside: Macro-op Fusion

- ❑ Multiple (typically 2) instructions can be “fused” into a one
 - Decoder hardware emits one decoded instructions from two
 - This does not affect ISA! Totally transparent to programmer/compiler
- ❑ Why?
 - Smaller number of instructions to process
 - While still maintaining RISC ISA (Also used in CISC / x86 with smaller instructions)
 - Typical criticism of RISC is a larger number of generated instructions for same program
 - (More cycles to execute same program)

rd is immediately “clobbered” by ld
Only one register write persists

Can be fused into one instruction
Without more functionality in the execute stage



```
// rd = array[offset]
add rd, rs1, rs2
ld rd, 0(rd)
```

Aside: Macro-op Fusion

- ❑ RISC-V benchmarks (RV64GC)
 - SPECINT 2006 benchmarks
 - Handful of fusion rules
 - About 5% decrease in executed instruction count
- ❑ Compared against x86-64
 - Without MOP Fusion: 1.16x instructions
 - With MOP Fusion: 1.09x instructions!
- ❑ RISC paradigm but with less instruction overhead!

Programmer: I was not consulted about this!

- ❑ Programmer: “If the processor told me it had parallel processing units, I would have written code optimized for it!”

Modern Processor Topics - Performance

❑ Transparent Performance Improvements

- Pipelining, Caches
- Superscalar, Out-of-Order, Branch Prediction, Speculation, ...
- Covered in CS250A and others

❑ Explicit Performance Improvements

- SIMD extensions, AES extensions, ...
- ...

