

Pipeline Architecture since 1985

- Last time, we completed the 5-stage pipeline MIPS.
 - Processors like this were first shipped around 1985
 - Still a fundamentally solid design
- Nevertheless, there have been advances in the past 25 years.
 - Deeper Pipelines
 - Dynamic Branch Prediction
 - Branch Target Buffers (removing the taken branch penalty)
 - Multiple Issue / Superscalar
 - Out-of-order Scheduling
- I will briefly overview these ideas
 - For more complete coverage, see CS433

ADVANCED
COMPUTER
ARCHITECTURES

NO HANDOUT
GO TO LAB SECTION
IF YOU HAVEN'T
FINISHED LAB 10

Recall our equation for execution time

- Make things faster by making any component smaller!!

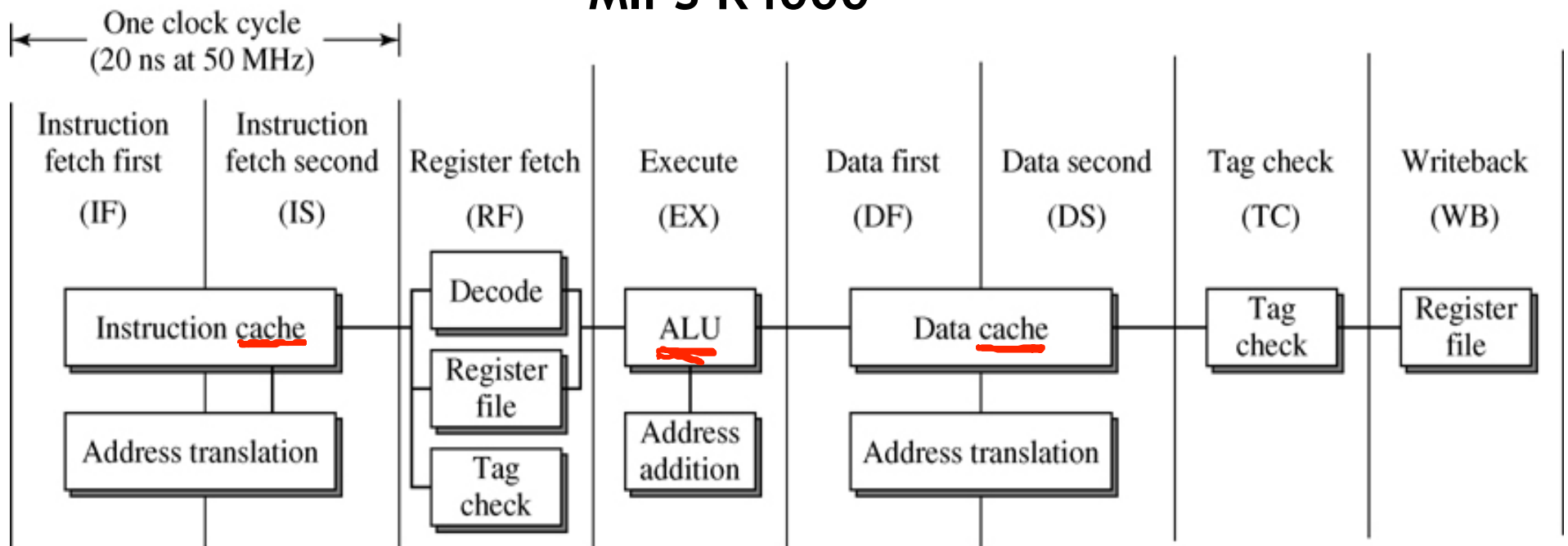
$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

Hardware can affect these

- We proposed pipelining to reduce clock cycle time.
 - If some is good, more is better right?

“Superpipelining”

MIPS R4000



More Superpipelining

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP	TC Fetch	Drive	Alloc	Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Figs	Br Ck	Drive			

Historical data from Intel's processors

Pipeline depths and frequency at introduction.

Microprocessor	Year	Clock Rate	Pipeline Stages
i486	1989	25 MHz	5
Pentium	1993	66 MHz	5
Pentium Pro	1997	200 MHz	10
P4 Willamette	2001	2000 MHz	22
P4 Prescott	2004	3600 MHz	31
Core 2 Conroe	2006	2930 MHz	14
Core 2 Yorkfield	2008	2930 MHz	16
Core i7 Gulftown	2010	3460 MHz	16

What
Happened?



There is a cost to deep pipelines

Microprocessor	Year	Clock Rate	Pipeline Stages	Power
i486	1989	25 MHz	5	5W
Pentium	1993	66 MHz	5	10W
Pentium Pro	1997	200 MHz	10	29W
P4 Willamette	2001	2000 MHz	22	75W
P4 Prescott	2004	3600 MHz	31	103W
Core 2 Conroe	2006	2930 MHz	14	75W
Core 2 Yorkfield	2008	2930 MHz	16	95W
Core i7 Gulftown	2010	3460 MHz	16	130W

POWER

- **Two effects:**
 - Diminishing returns: **pipeline register latency becomes significant**
 - **Negatively impacts CPI** (longer stalls, more instructions flushed)

Mitigating CPI loss 1: Dynamic Branch Prediction

- “Predict not-taken” is cheap, but
 - Some branches are almost always taken
 - Like loop back edges.

loop: bgt
j ← taken

- What fraction of time will the **highlighted** branch mispredict?

```
for (int i = 0 ; i < 1000 ; i ++){  
    for (int j = 0 ; j < 10 ; j ++){  
        // do something  
    }  
}
```

something
bit # 10, 11, loop

pred

	T	T	T	T	T	T	T	T	NT	T	T
	T	T	T	T	T	T	T	T	T	NT	T

a) 0% b) 10% c) 20% **d) 90%** e) 100%

Mitigating CPI loss 1: Dynamic Branch Prediction

- It turns out, instructions tend to do the same things over and over again
 - Idea: Use past history to predict future behavior
 - First attempt:
 - Keep 1 bit per branch that remembers last outcome
- What fraction of time will the **highlighted** branch mispredict?

```
for (int i = 0 ; i < 1000 ; i ++ ) {  
    for (int j = 0 ; j < 10 ; j ++ ) {  
        // do something  
    }  
}
```

a) 0% b) 10% **c) 20%** d) 90% e) 100%

Two-bit branch prediction

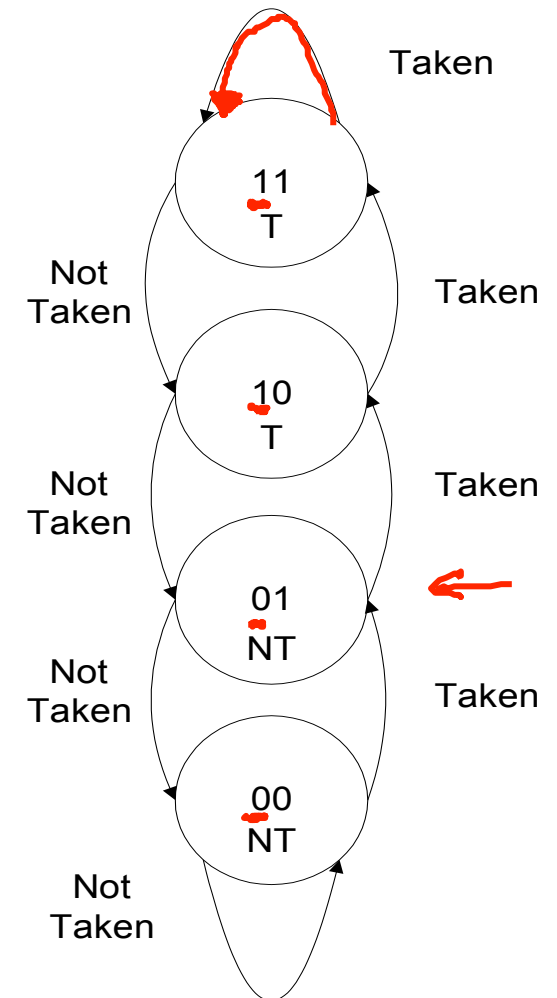
- Solution: add longer term memory (hysteresis)

- Use a *saturating* 2-bit counter:
 - Increment when branch taken
 - Decrement when branch not-taken
 - Use **top bit** as prediction

- How often will the branch mispredict?

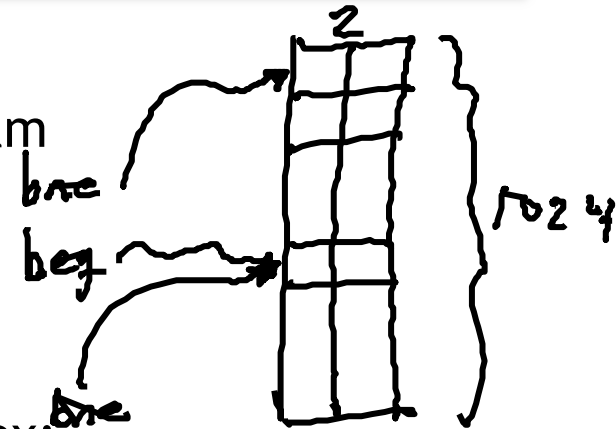
T T T T T T T T T NT T T T T T ..
 0 1 2 3 3 3 3 3 3 3 2 3 3 3 3 3 ..
 pred T T T T T T T T T T T T T T T ..

a) 0% b) 10% c) 20% d) 90% e) 100%



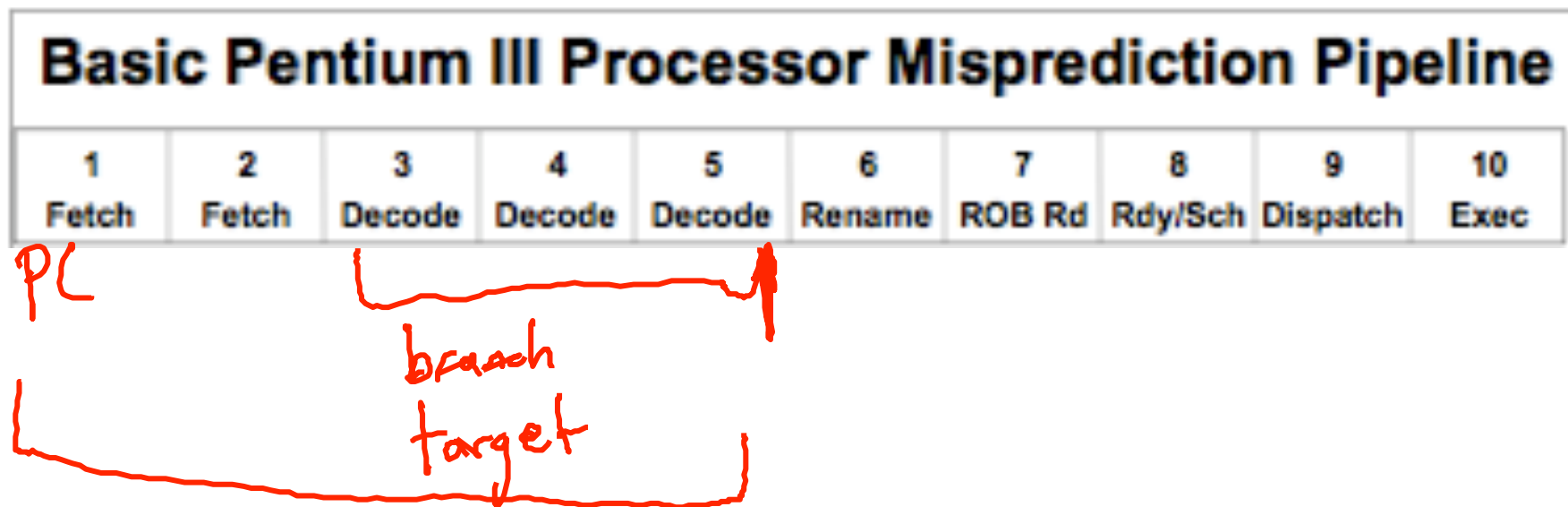
Branch prediction tables

- Too expensive to keep 2 bits per branch in the program
- Instead keep a fixed sized table in the processor
 - Say 1024 2-bit counters.
- “Hash” the program counter (PC) to construct an index:
 - Index = $(\underline{PC} \gg 2) \wedge (\underline{PC} \gg 12) \wedge 0x3ff$
- Multiple branches will map to the same entry (*interference*)
 - But generally not at the same time
 - Programs tend to have working sets.



When to predict branches?

- Need:
 - PC (to access predictor)
 - To know it is a branch (must have decoded the instruction)
 - The branch target (computed from the instruction bits)



- How many flushes on a not taken prediction? ~~8~~
- How many flushes on a taken prediction?
- Is this the best we can do? **No**

Edit: If branch target computed in Cycle 5, the 4 cycles of flushes

Mitigating CPI loss 1: Branch Target Buffers

- Need:

- PC ← Already have at fetch.
 - To know it is a branch
 - The branch target
- Can remember and make available at fetch

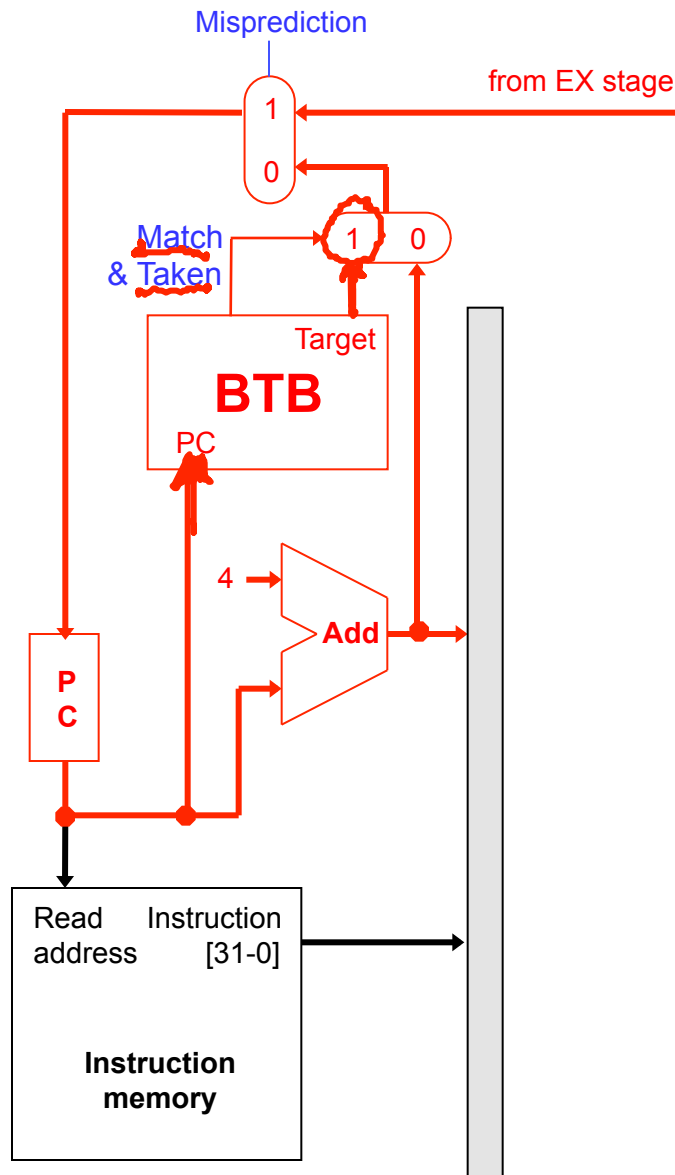
- Create a table: Branch Target Buffer

PCx
HASH
Index

PC	2-bit counter	target
PC ₁		

- Allocate an entry whenever a branch is taken (& not already present)

BTB accessed in parallel with reading the instruction



- If matching entry found, and ...
- 2-bit counter predicts taken
 - Redirect fetch to branch target
 - Instead of $PC+4$
- What is the taken branch penalty?
 - (i.e., how many flushes on a predicted taken branch?)

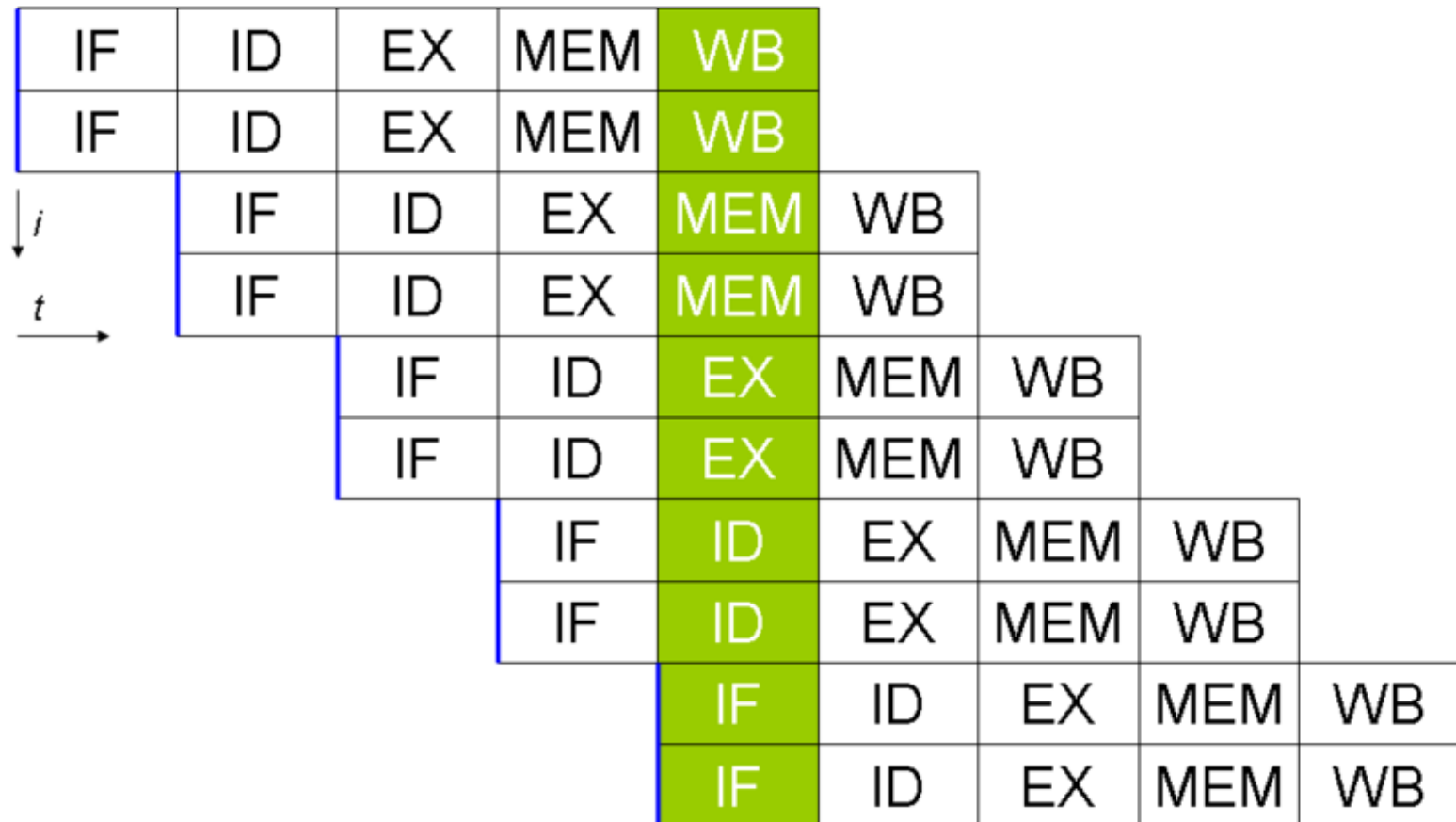
a) 0

Back to our equation for execution time

$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

- Removing stalls & flushes can bring CPI down 1.
 - Can we bring it lower?

Multiple Issue



Issue width over time

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width
i486	1989	25 MHz	5	1
Pentium	1993	66 MHz	5	2
Pentium Pro	1997	200 MHz	10	3
P4 Willamette	2001	2000 MHz	22	3
P4 Prescott	2004	3600 MHz	31	3
Core 2 Conroe	2006	2930 MHz	14	4
Core 2 Yorkfield	2008	2930 MHz	16	4
Core i7 Gulftown	2010	3460 MHz	16	4

Static Multiple Issue

- Compiler groups instructions into *issue packets*
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Pad with nop if necessary

Example: MIPS with Static Dual Issue

- Dual-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - `add $t0, $s0, $s1`
`load $s2, 0($t0)`
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```

Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
  
```

	ALU/branch	Load/store	cycle
Loop:	<code>nop</code>	<code>lw \$t0, 0(\$s1)</code>	1
	<code>addi \$s1, \$s1, -4</code>	<code>nop</code>	2
	<code>addu \$t0, \$t0, \$s2</code>	<code>nop</code>	3
	<code>bne \$s1, \$zero,</code> <code>Loop</code>	<code>sw \$t0, 4(\$s1)</code>	4

- $IPC = 5/4 = 1.25$ (c.f. peak $IPC = 2$)

Loop Unrolling

- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called *register renaming*
 - Avoid loop-carried *anti-dependencies*
 - Store followed by a load of the same register
 - Aka “name dependence”
 - ▶ Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

- $IPC = 14/8 = \underline{1.75}$
 - Closer to 2, but at cost of registers and code size


Dynamic Multiple Issue = Superscalar

- CPU decides whether to issue 0, 1, 2, ... instructions each cycle
 - Avoiding structural and data hazards
- Avoids need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU
 - By stalling appropriately
- Limited benefit without compiler support
 - Adjacent instructions are often dependent

Out-of-order Execution (Dynamic Scheduling)

- Allow the CPU to execute instructions *out of order* to avoid stalls
 - But commit result to registers in order

- Example



```
lw      $t0, 20($s2)
add     $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

- Can start sub while add is waiting for lw

- Why not just let the compiler schedule code?

Out-of-order Scheduling

- Allow the CPU to execute instructions *out of order* to avoid stalls
 - But commit result to registers in order

- Example

```
lw      $t0, 20($s2)
add     $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

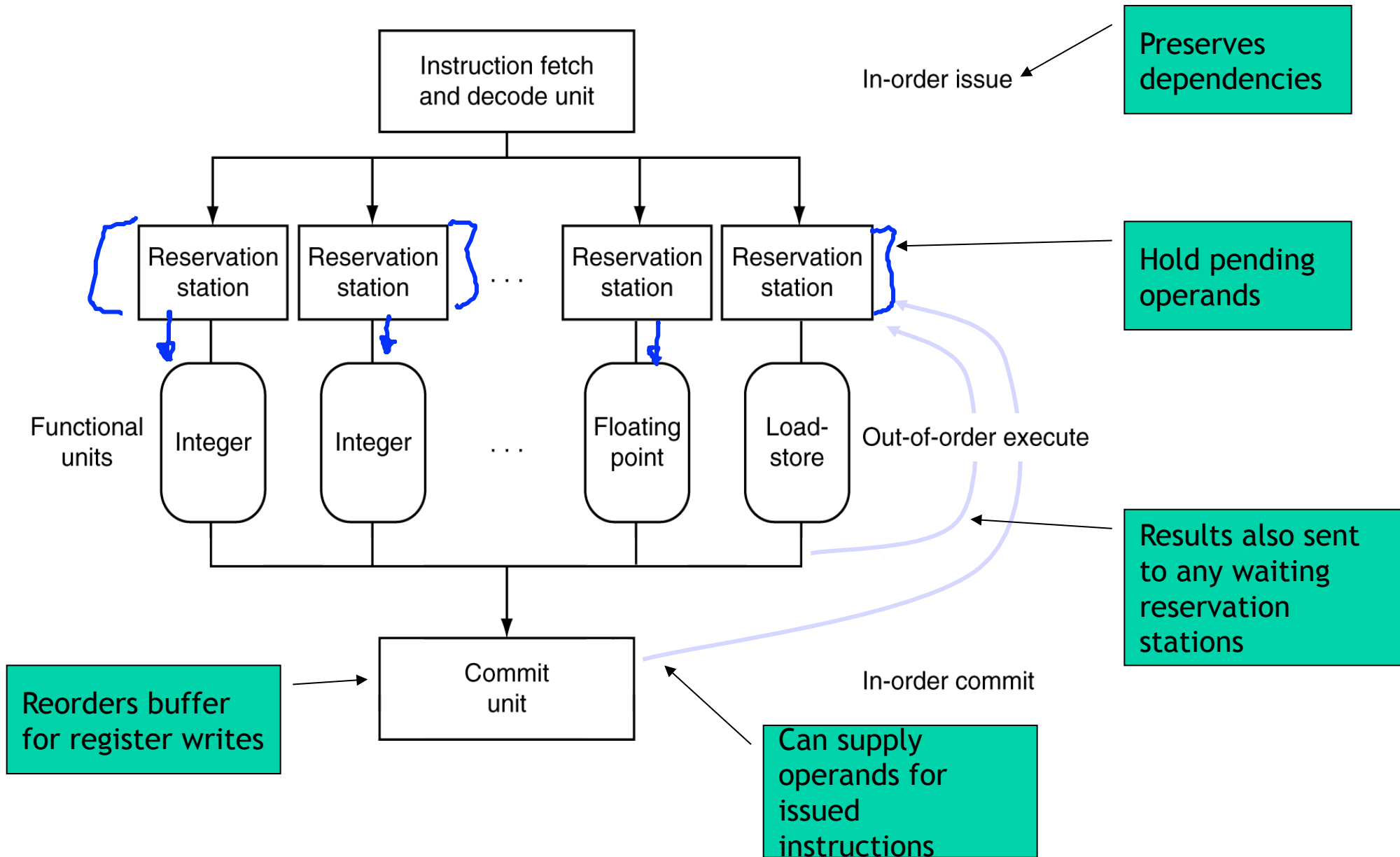
- Can start sub while add is waiting for lw
- Why not just let the compiler schedule code?
 - Not all stalls are predicable
 - e.g., we'll see shortly that memory has variable latency
 - Can't always schedule around branches
 - Branch outcome is dynamically determined (i.e., predicted)
 - Different implementations have different latencies and hazards

Implementing Out-of-Order Execution

Basically, unroll loops in hardware:

1. Fetch instructions in program order ($\leq 4/\text{clock}$)
2. Predict branches as taken/not-taken
3. To avoid hazards on registers, *rename registers* using a set of internal registers (~80 registers)
4. Collection of renamed instructions might execute in a *window* (~60 instructions)
5. Execute instructions with ready operands in 1 of multiple *functional units* (ALUs, FPUs, Ld/St)
6. Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*
7. If predicted branch correctly, *commit* results in program order
8. If predicted branch incorrectly, discard all dependent results and start with correct PC

Dynamically Scheduled CPU



Takeaway points

- The 5-stage pipeline is not a bad mental model for SW developers:
 - Integer arithmetic is cheap
 - Loads can be relatively expensive
 - Especially if there is not other work to be done (e.g., linked list traversals)
 - We'll further explain why starting on Friday
 - Branches can be relatively expensive
 - But, primarily if they are not predictable
- In addition, try to avoid long serial dependences; given double D[10]
 - $((D[0] + D[1]) + (D[2] + D[3])) + ((D[4] + D[5]) + (D[6] + D[7]))$
 - Is faster than:
 - $((((((((D[0] + D[1]) + D[2]) + D[3]) + D[4]) + D[5]) + D[6]) + D[7]))$
- There is phenomenal engineering in modern processors