Control Hazards

Branches disrupt the flow of instructions in the pipeline

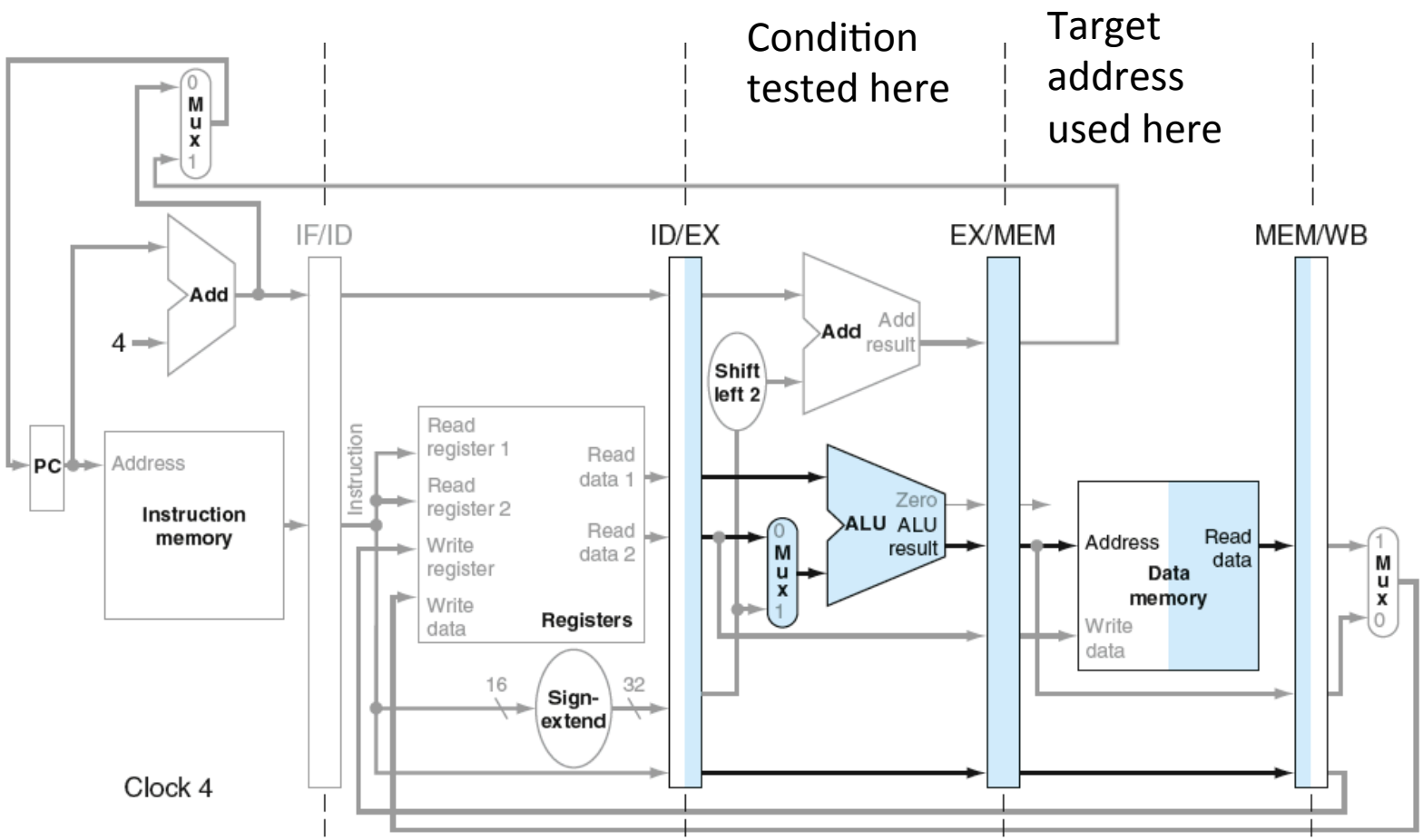
- The branch target address is computed in stage 3
- The branch condition is evaluated in stage 3
- The branch takes effect in stage 4

Instructions behind a taken branch must not complete

- Stages 1 through 3 must be flushed
- This creates 3 bubbles (a 3-cycle penalty)

The effect is called a control hazard

Control Hazards



If condition is true, the PC is loaded with the target address when beq is in stage 4

Control Hazard Example

```
        sub  $11,$6,$5

        beq   $11,$0,skip

        add $4,$7,$3

        sw    $9,4($7)

        add   $9,$2,$9
              .
              .
              .
skip:   or    $8,$4,$0
```

If $11 = 0, the 3 instructions that follow beq should not complete.
The next instruction to execute should be or $8,$4,$0 instruction

Control Hazard Example

Cycle	IF	ID	EX	MEM	WB
1	sub $11,$6,$5				
2	beq $11,$0,skip	sub $11,$6,$5			
3	add $4,$7,$3	beq $11,$0,skip	sub $11,$6,$5		
4	sw $9,4($7)	add $4,$7,$3	beq $11,$0,skip	sub $11,$6,$5	
5	add $9,$2,$9	sw $9,4($7)	add $4,$7,$3	beq $11,$0,skip	sub $11,$6,$5
6	or $8,$4,$0				beq $11,$0,skip

Flushing adds 3 extra clock cycles in this case

This is known as the branch penalty

Control Hazard Options

1. Delay fetching instructions until branch behavior is known
   - Still causes 3-cycle penalty (outcome is known in stage 4)
2. Employ delayed branches
   - Fill delay slots with instructions needing to execute in any case
   - Compiler or programmer fills delay slots
   - Use NOPs if useful instructions cannot be identified
3. Evaluate the branch condition early
   - Requires computing target address in stage 2
   - Requires comparator in stage 2
4. Predict the behavior of the branch instruction
   - Requires recording previous branch behavior
   - Flushing is still required if prediction is wrong

Branch Delays

The behavior of an unconditional branch is known
- The branch is decoded in stage 2
- The instruction following the branch is in stage 1
- This instruction could be flushed

Allowing the instruction to execute avoids a bubble
- The instruction is said to occupy the “branch delay slot”
- Compilers can move an instruction into the delay slot
- It is easier to fill a single slot than to fill 3

Delay Slot Example

```
        li    $9,16
        li    $10,0xFFF
loop:   lw    $4,0($12)
        and   $5,$7,$10
        add   $5,$4,$5
        addi  $9,$9,-1
        bne   $9,$0,loop
        nop
```

```
        li    $9,16
        li    $10,0xFFF
loop:   lw    $4,0($12)
        and   $5,$7,$10
        add   $5,$4,$5
        bne   $9,$0,loop
        addi  $9,$9,-1
```

Branch delay slot instruction executes whether branch is taken or not
A cycle is saved by filling the slot with a useful instruction

Delay Slot Example

```
        li    $9,16
        li    $10,0xFFF
loop:   lw    $4,0($12)
        and   $5,$7,$10
        add   $5,$4,$5
        addi  $9,$9,-1
        bne   $9,$0,loop
        nop
```

```
        li    $9, 15
        li    $10,0xFFF
loop:   lw    $4,0($12)
        and   $5,$7,$10
        add   $5,$4,$5
        bne   $9,$0,loop
        addi  $9,$9,-1
```

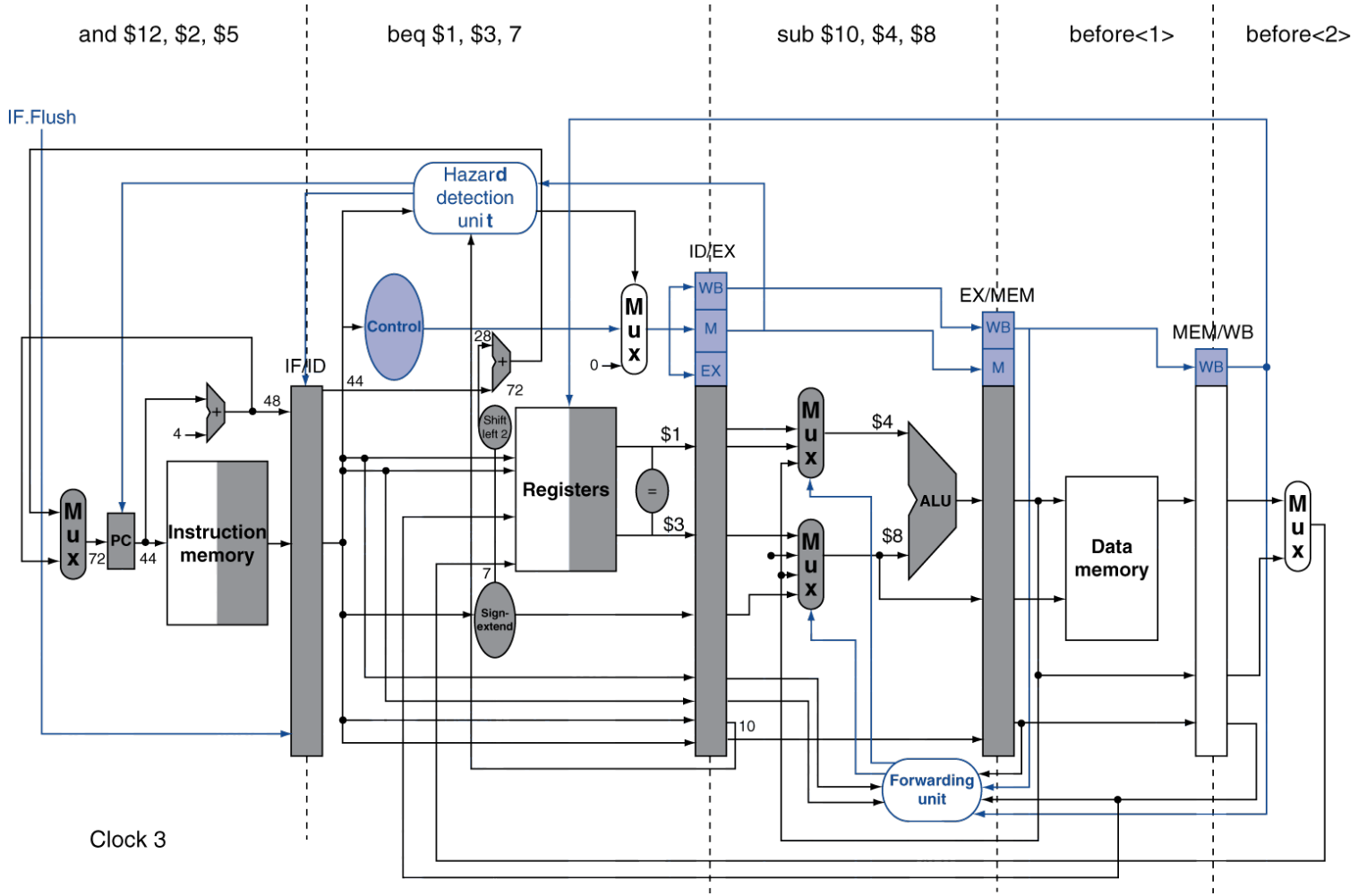
The initial value in the loop control register has been set to 15 so that the number of loop iterations will be the same as for the original code.

Early Evaluation

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Early Evaluation

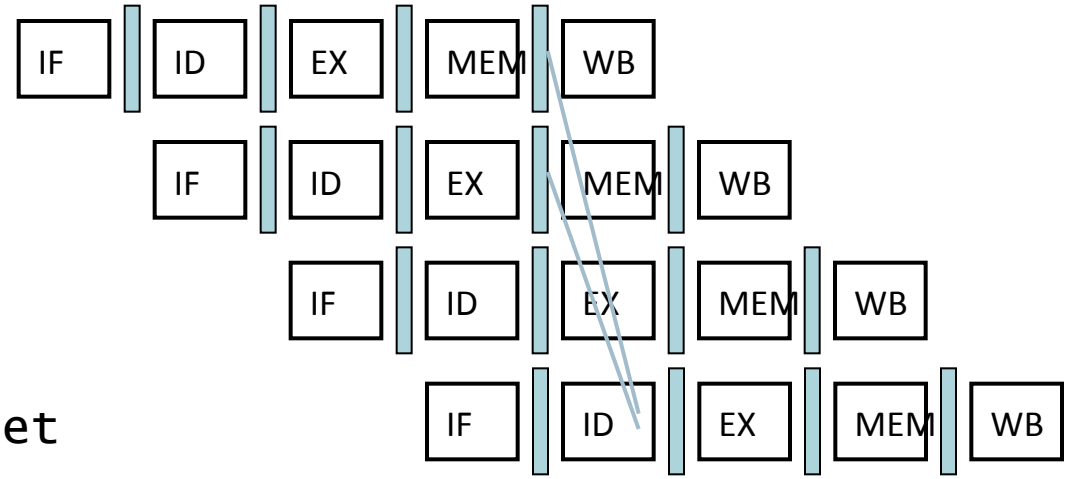


Delay slot is still needed since condition is tested in stage 2

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

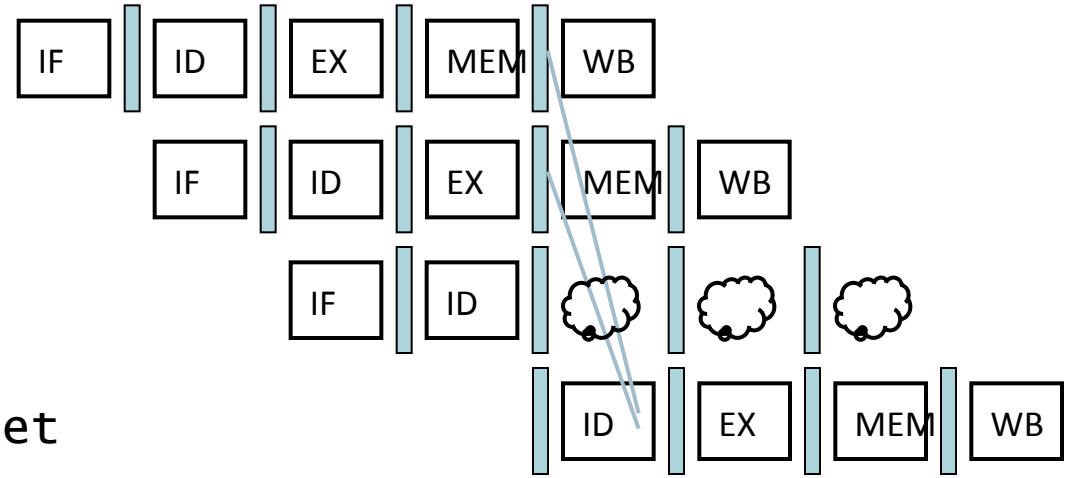


- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

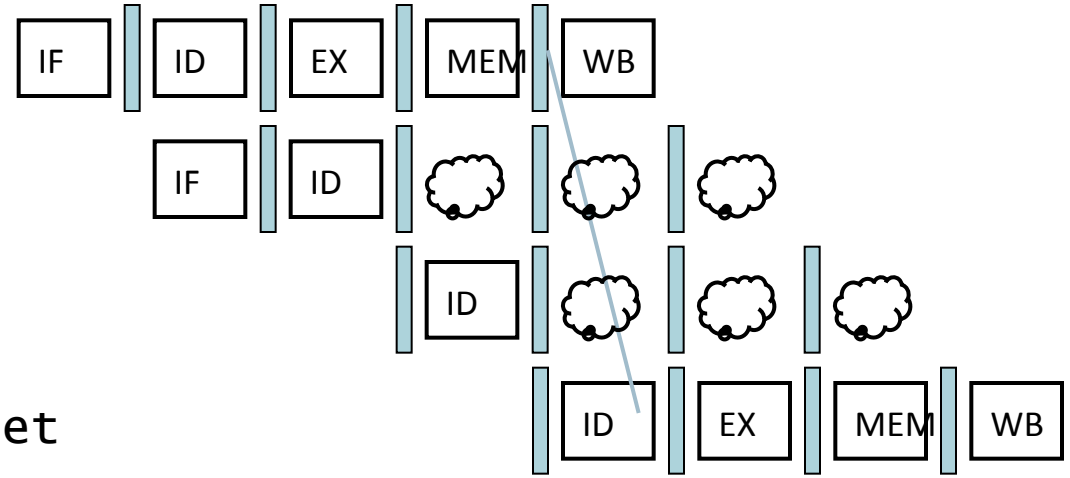
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Control Hazard Example

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



Control Hazards

Branches have more impact on deeper and superscalar pipelines

- More stages may have to be flushed
- Superscalar pipelines process multiple instructions per stage

Branch prediction will be examined as another technique

Branch Prediction

Studies have shown:
20% to 30% of program instructions involve branching
About 65% of branches are taken

Instructions along the predicted path are speculative
the work done must be undone if the prediction is wrong

There are two options for branch prediction:
- static prediction & dynamic prediction
  - Static prediction can be done by the compiler
  - Dynamic prediction requires extra hardware

Static Branch Prediction

Static (fixed) Prediction
The prediction is always the same (taken or not taken)

Forward branches may be predicted not taken

Backwards branches may be predicted taken
e.g. at end of loops

Once actual behavior is known, work may have to be undone

Dynamic Branch Prediction

Dynamic Branch Prediction Increases accuracy of prediction

Previous history of branch behavior is recorded
Shows if branch was taken or not when last encountered
- Branch prediction buffer
- Branch History table (BHT)
- Decode History table (DHT)

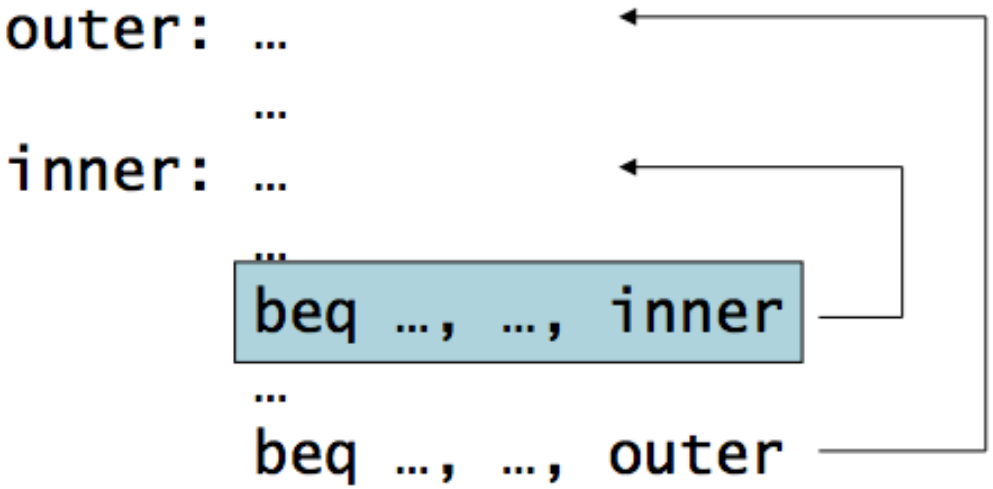
These tables are high-speed buffers (caches)
- address of branch instruction is used to access them

Branch Prediction

BHT is accessed as soon as PC is updated
- Hits only occur for branch instructions already in the table
- Hits provide the predicted target address
- Non-branch instructions cause misses
- Can be accessed before branch enters the pipeline

DHT is only accessed after branch is decoded
- It is not checked for non-branch instructions
- hits provide the predicted target instruction

Loops & Branch Prediction



Let's base prediction on a recorded bit (=0 if not taken, =1 if taken)

Assume 9 iterations of inner loop, 1st prediction & 9th are wrong

For each iteration of outer loop, inner beq is mispredicted twice

A 2-bit predictor would be better

2-Bit Prediction

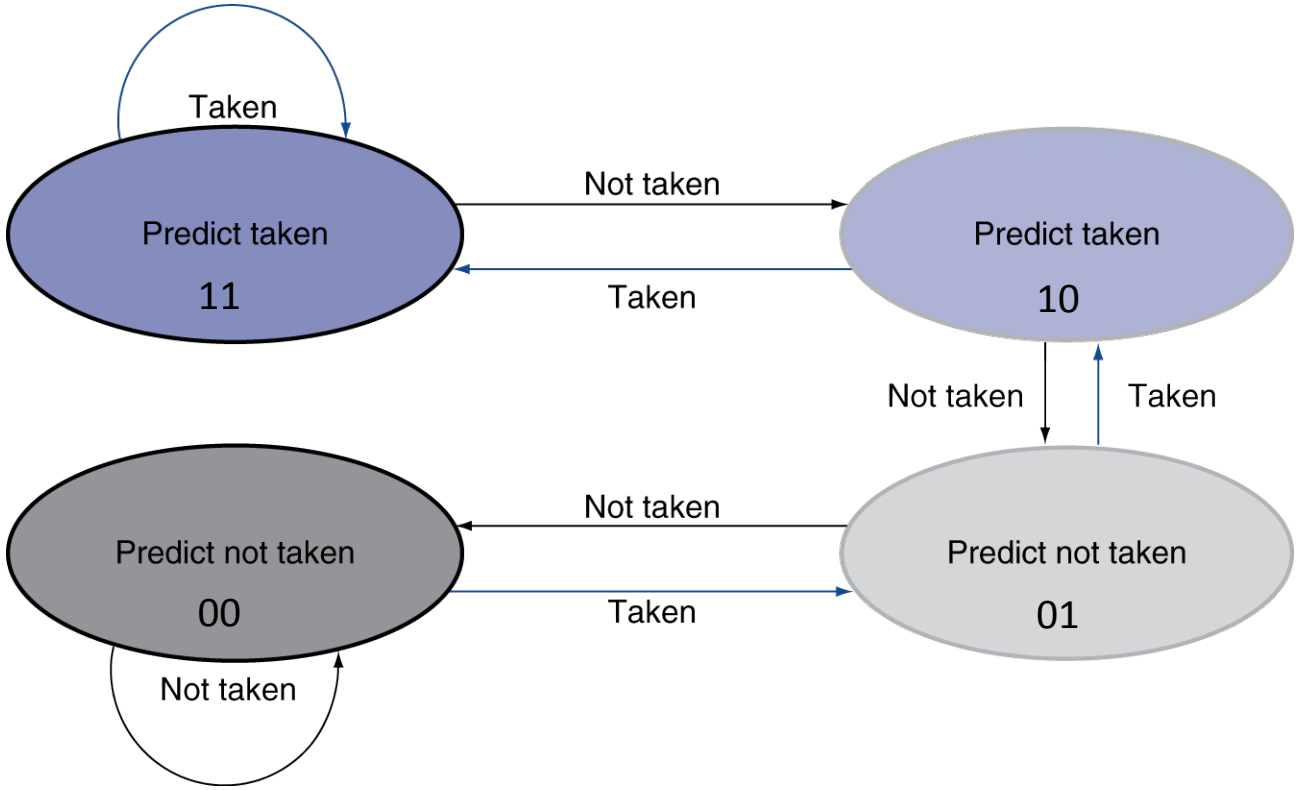
2-bit predictor	Meaning
00	Strongly not taken
01	Weakly not taken
10	Weakly taken
11	Strongly taken

On 1st encounter bits change from 00 to 01 (for previous example)
On 2nd encounter bits change from 01 to 10
On 3rd encounter bits change from 10 to 11
On 9th encounter bits change from 11 to 10

Inner prediction is wrong three times for 1st iteration of outer loop
But wrong only once for each remaining outer loop iteration

2-Bit Prediction

- Only change prediction on two successive mispredictions



Exceptions & Pipelining

Exceptions affect the pipeline like a function call
- control is diverted to the exception handler
- interrupts are a particular type of exception

Exceptions are triggered by unexpected events
- detecting invalid instructions during decode
- arithmetic overflow
- memory errors
- syscall
- interrupts from external I/O controllers

Exceptions & Pipelining

System Coprocessor (CP0) manages MIPS exceptions
- EPC is the exception program counter (CP0 $14)
- Holds address of offending (or interrupted) instruction

Cause register (CP0 $13) indicates problem

Control is transferred to handler at 0x800000018
- handler takes the appropriate action
- resumes program using address in EPC
- terminates program if problem cannot be resolved

Exceptions & Pipelining

Transfer to exception handler causes a control hazard
  program instructions are flushed from pipeline
  this creates pipeline bubbles

The effect on pipeline is similar to a mispredicted branch

Exception Example

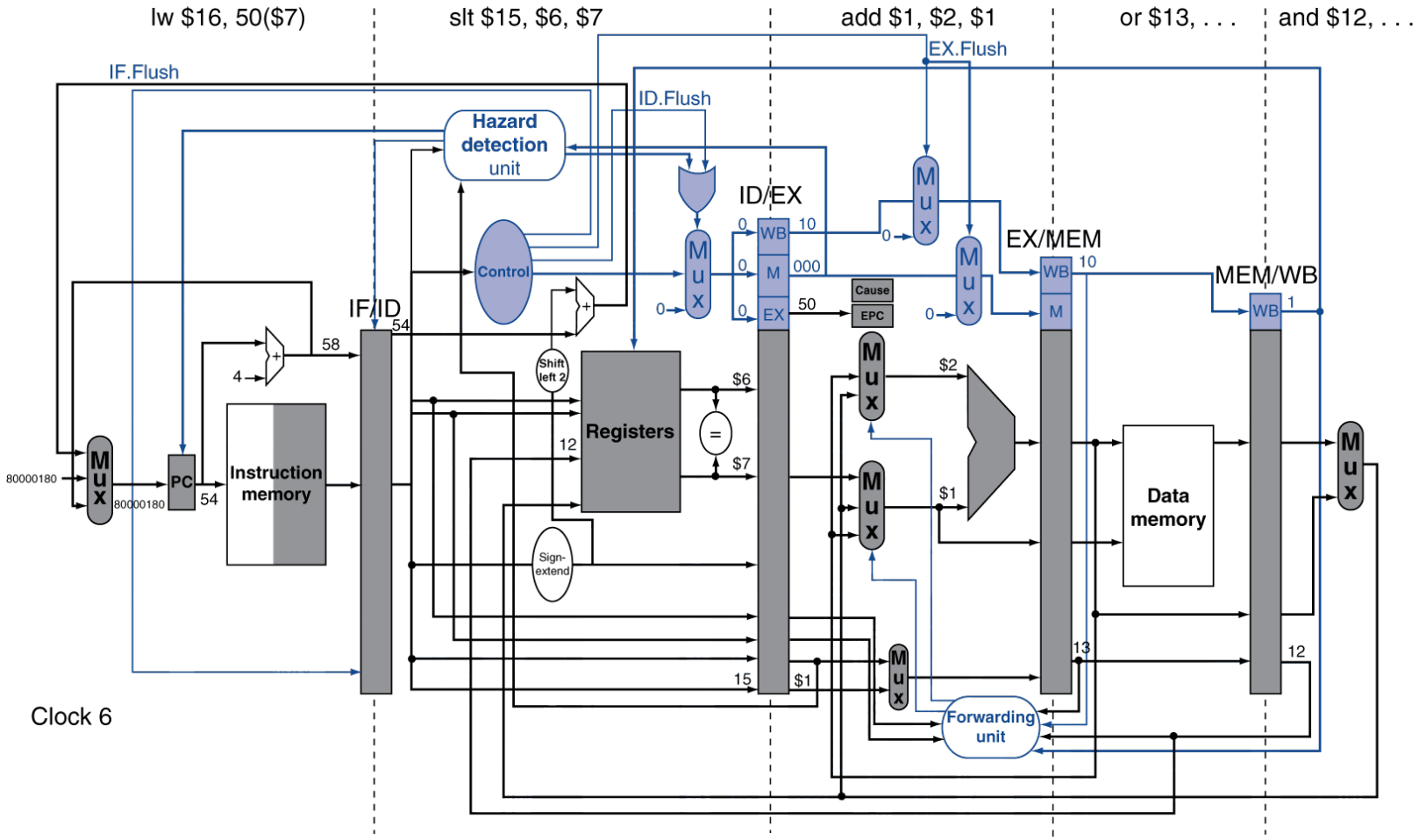
- Exception on add in

```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4C      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
…
```

- Handler

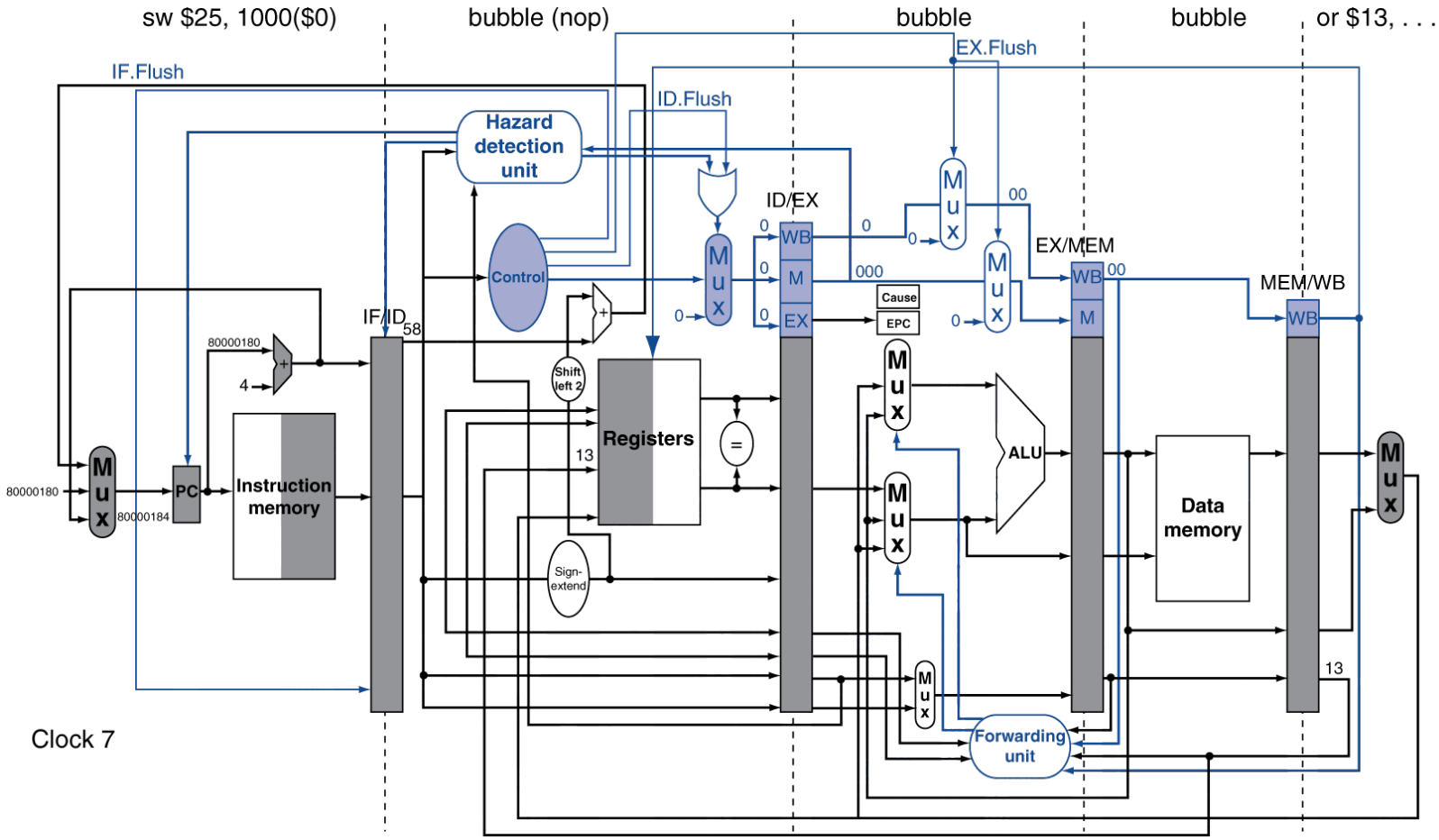
```
80000180    sw    $25, 1000($0)
80000184    sw    $26, 1004($0)
…
```

Exception Example



Flush control signal inserts bubbles

Exception Example



Flush control signal inserts bubbles

Precise vs Imprecise Exceptions

Some systems support precise exceptions
- no register writes after the offending instruction
- leaves the system in a consistent state

Imprecise exceptions
- instruction causing exception is identified
- but succeeding instructions may complete
- allows the system to be in an inconsistent state

Superpipelining

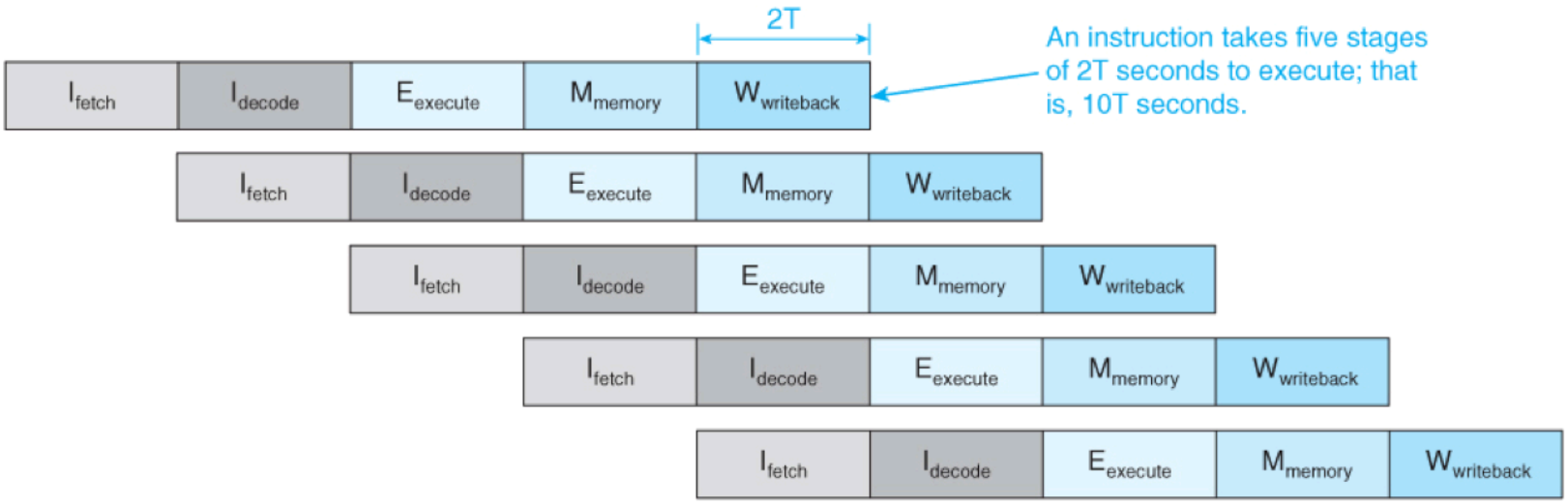
Superpipelining allows more instructions to be overlapped

Operations may not all require a complete clock cycle
- register reads or writes
- check for hits in cache
- decoding an opcode

The pipeline can run faster than the external clock rate

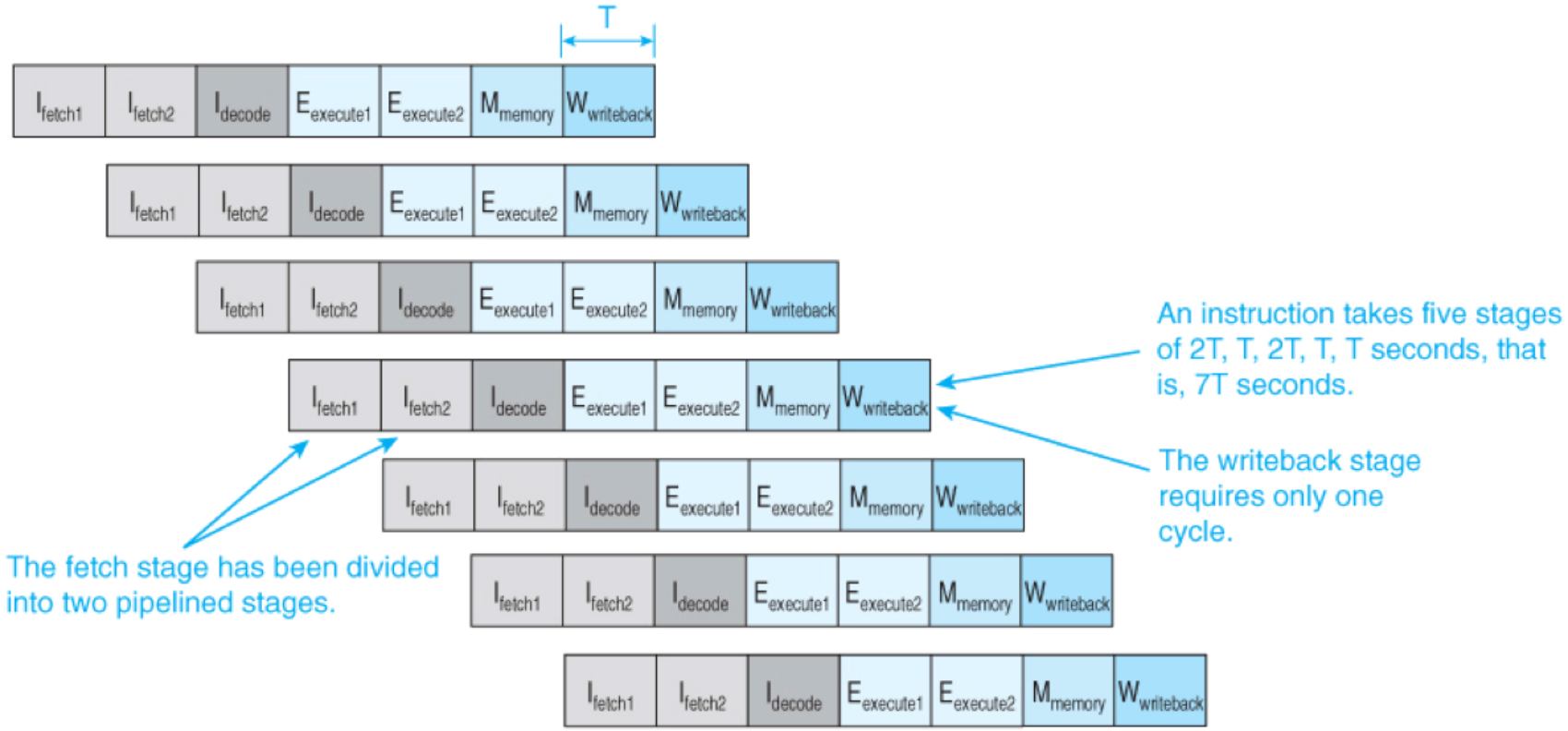
Superpipelines have some stages subdivided
- a different instruction is in each phase with a stage

Superpipelining



Conventional pipeline with cycle time = 2T

Superpipelining



Speedup = 10/7 = 1.43

Superpipeline with cycle time = T (twice the rate)

Superpipelining

Superpipelines use finer granularity
  instruction throughput increases
  the cost is a higher clock rate

Other parts of the datapath may require a slower rate
    precludes running the entire system at a higher clock rate

Hazards & mispredictions have a greater impact
    more stages have to be flushed

More stages cause more interstage delays
    more pipeline registers

Superpipelining

Superpipelines overlap more instructions

Superpipelining provides a modest performance increase

Superscalar provides a greater benefit

Superscalar systems can include superpipelining

Superscalar System Overview

Pipelining's goal is a throughput of 1 instruction per cycle
  data hazards & control hazards make this difficult

Scalar systems handle 1 instruction per stage
  each stage requires one clock cycle
  only one instruction is started each cycle
  all instructions go through all stages
  some instructions may take longer than without the pipeline
  however the instruction throughput is increased

Superscalar System Overview

Superscalar systems aim for 2 or more instructions per cycle
 extra hardware executes multiple instruction per cycle
 multiple pipelines can operate in parallel
 or each stage can process multiple instructions at one time

Multiple instructions are fetched at one time
  requires a wider CPU-to-memory bus

Multiple instructions are decoded together
  dependencies between the instructions are detected
  independent instructions are issued to their execute units

Superscalar System Overview

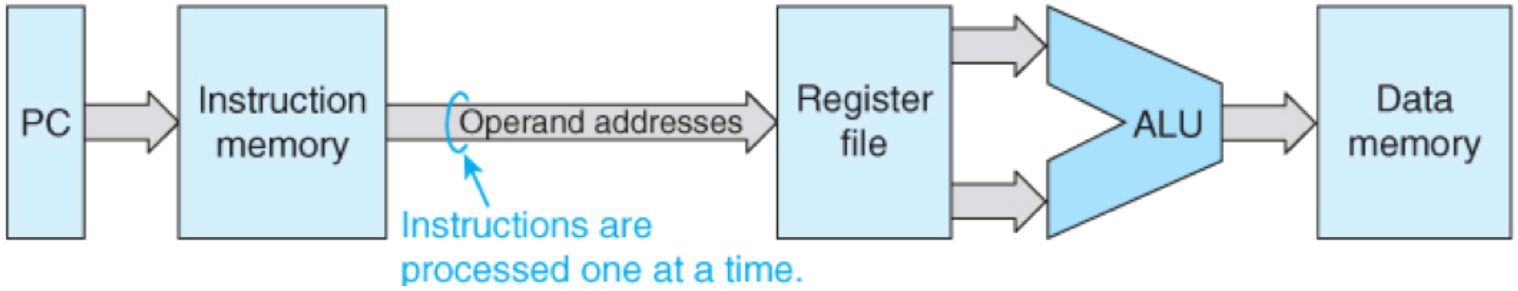
Multiple execute units process instructions at the same time
- multiple ALUs for integer operations
- multiple floating point units
- load/store unit (to compute memory addresses)
- branch unit to analyze branch conditions & make predictions

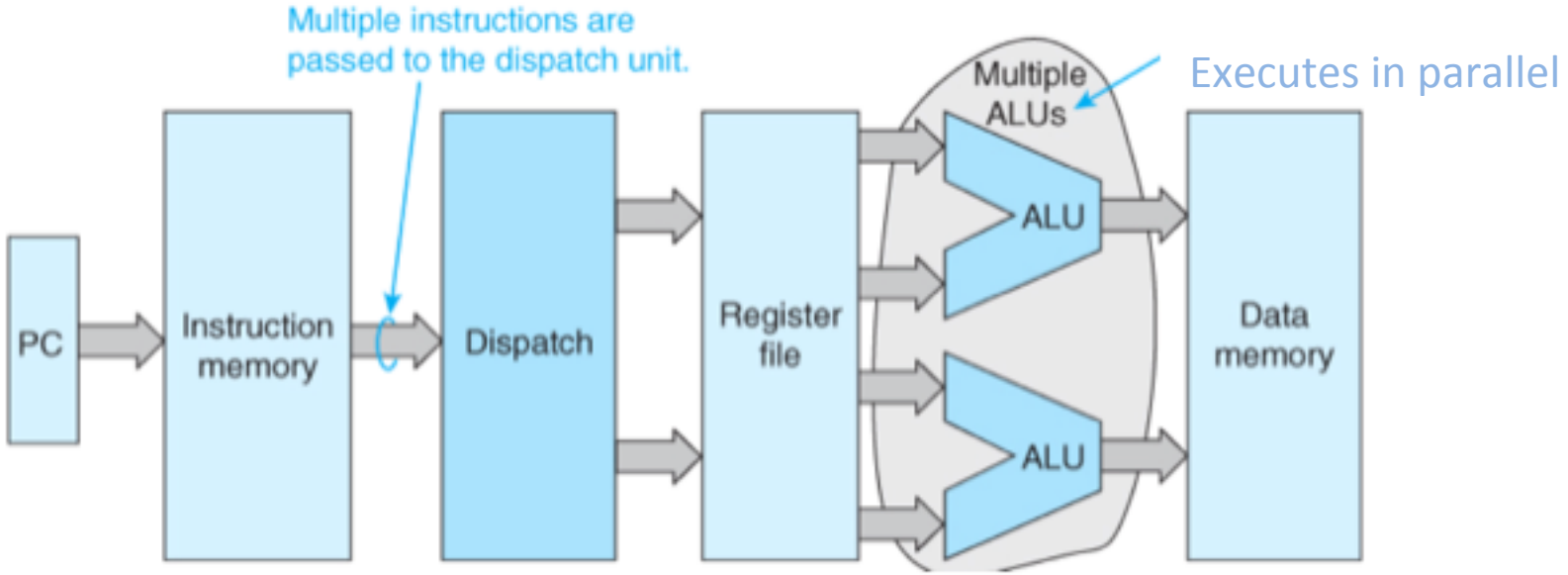
Resource Hazards are possible with superscalar systems
- the required unit may not be available for next instruction
- required registers may already be in use

Out-of-order execution may be needed for good performance
- instructions may start in a different order than in program
- later instructions may complete before earlier ones

Superscalar System Overview

Scalar Pipeline





Superscalar Pipeline

Superscalar System Overview

An m-way superscalar system has m parallel pipelines
  m-fold increases in performance are seldom achieved

The Pentium P5 had 2 integer pipelines
  the 2nd pipeline could only be used about 30% of the time

The original Pentium used superscalar operation
  two 5-stage integer pipelines U and V
  U pipe included a shifter not in the V pipe
  a 6-stage floating point unit

Superscalar System Overview

Compilers generate machine instructions in program order

Superscalar processors *dispatch* multiple instructions in parallel
- operands are obtained from multiported register files
- Instructions that have operands can execute in parallel
- The execute units needed must be available

The program semantics (meaning must be preserved)

Sequence 1

```
add   $11,$12,$13
add   $14,$11,$13
add   $15,$16,$17
```

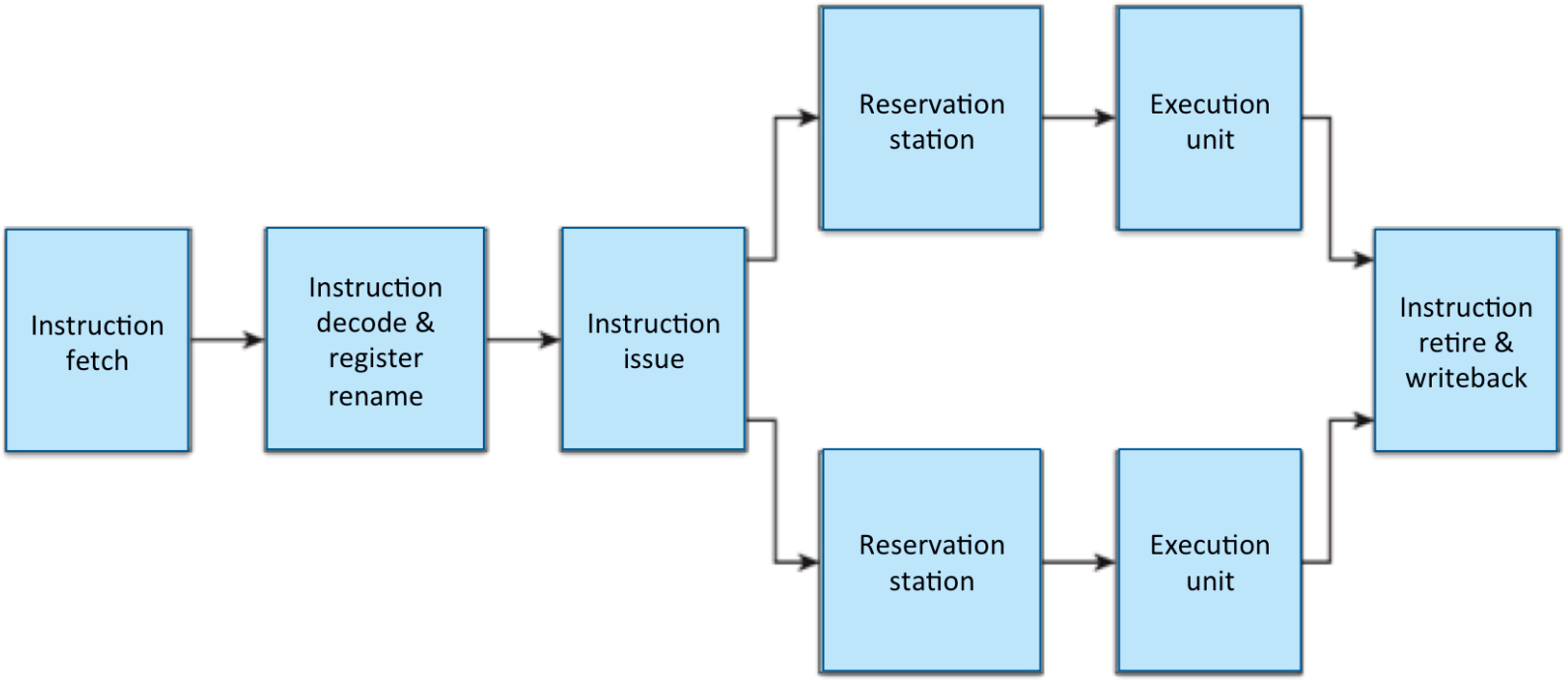
Sequence 2

```
add   $11,$12,$13
add   $15,$16,$17
add   $14,$11,$13
```

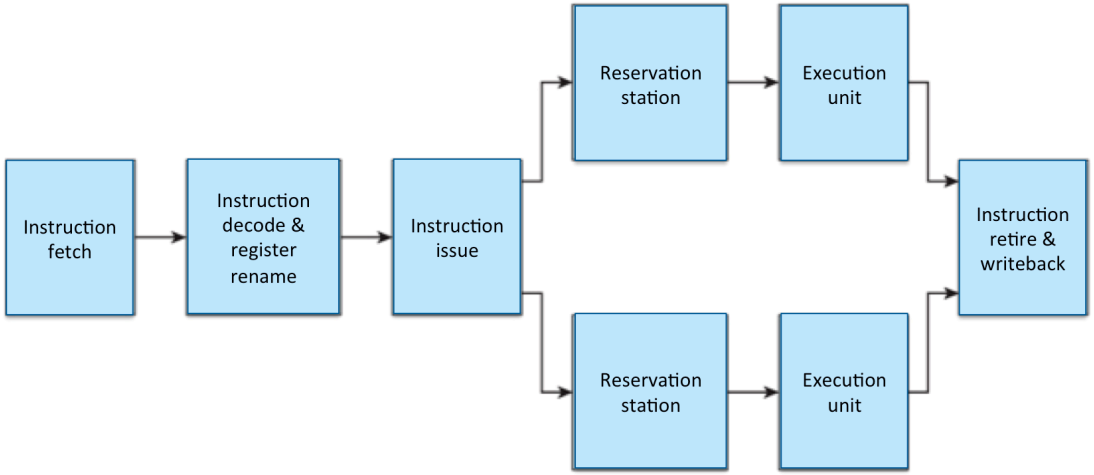
Different instruction order but same meaning or net effect

Superscalar Operation

A more realistic view of a superscalar system is:



Superscalar Operation



Instruction fetch

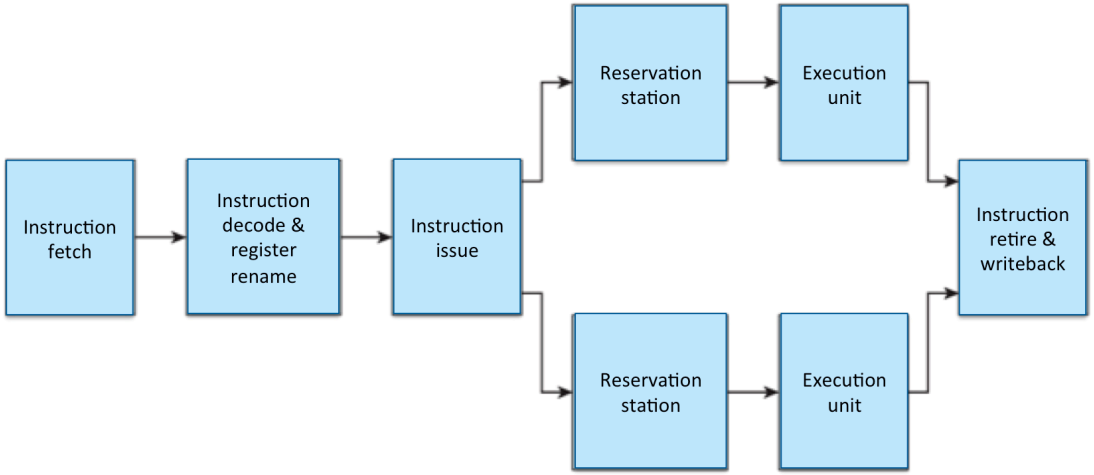
obtains instructions from cache or memory

Instruction Decode

interprets opcodes

substitutes temporary registers to avoid unnecessary stalls

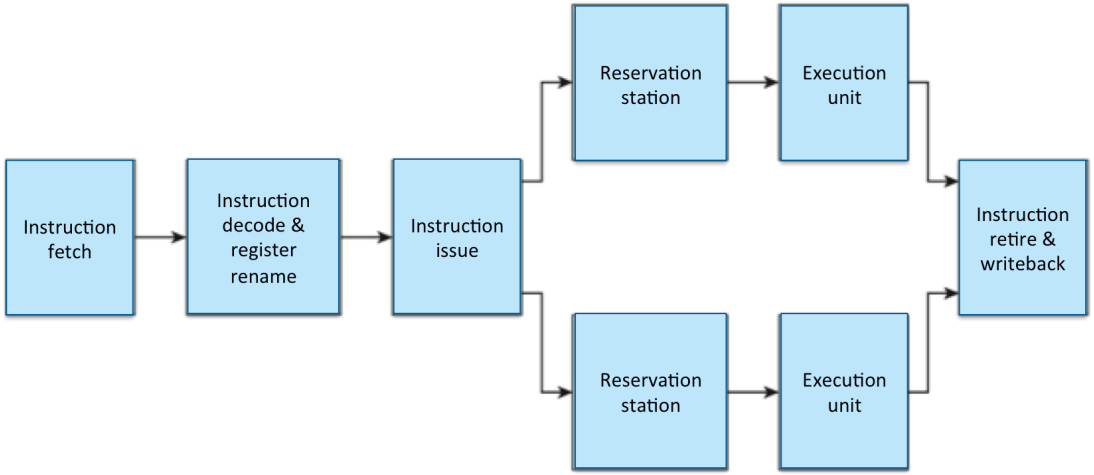
Superscalar Operation



Instruction issue

- ensures as many instructions as possible execute in parallel
- sends instructions to reservation stations (*issues* them)
- Instructions are *dispatched* from the reservation stations
  - once the required input operands are available

Superscalar Operation



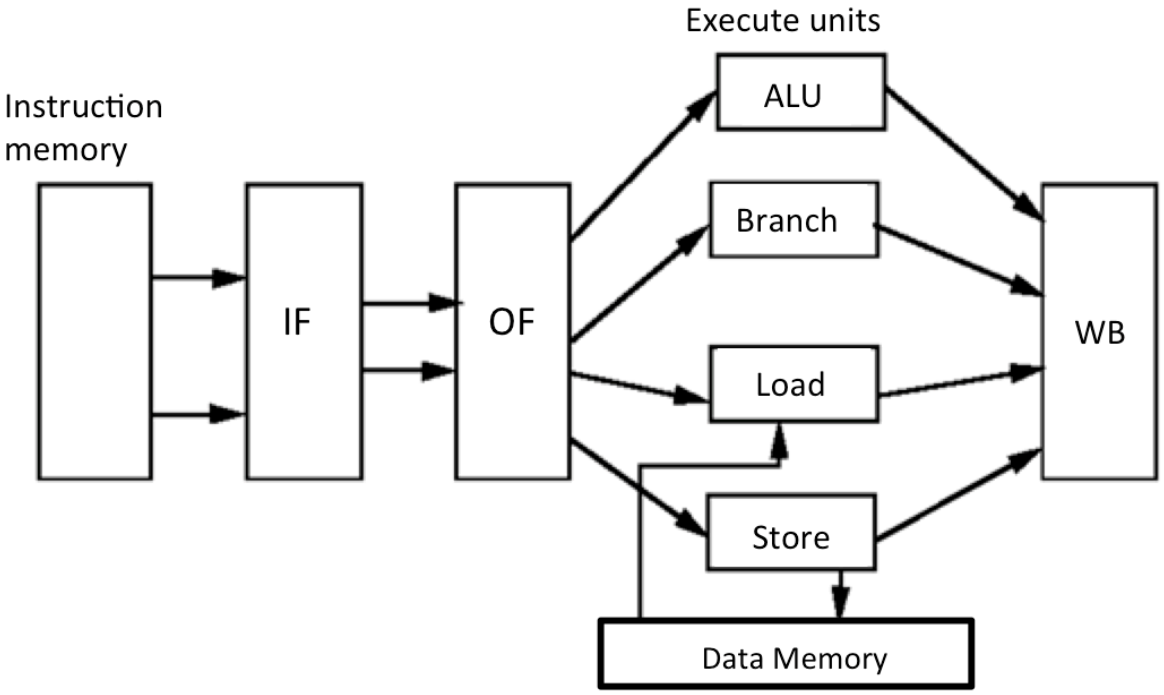
Reservation stations

- Serve as front end buffer to execution units
- Hold instructions that use the corresponding execution unit

Instruction retire

- Writes the results to the destination registers (*commits* them)
- Tells reservation stations when resources are available

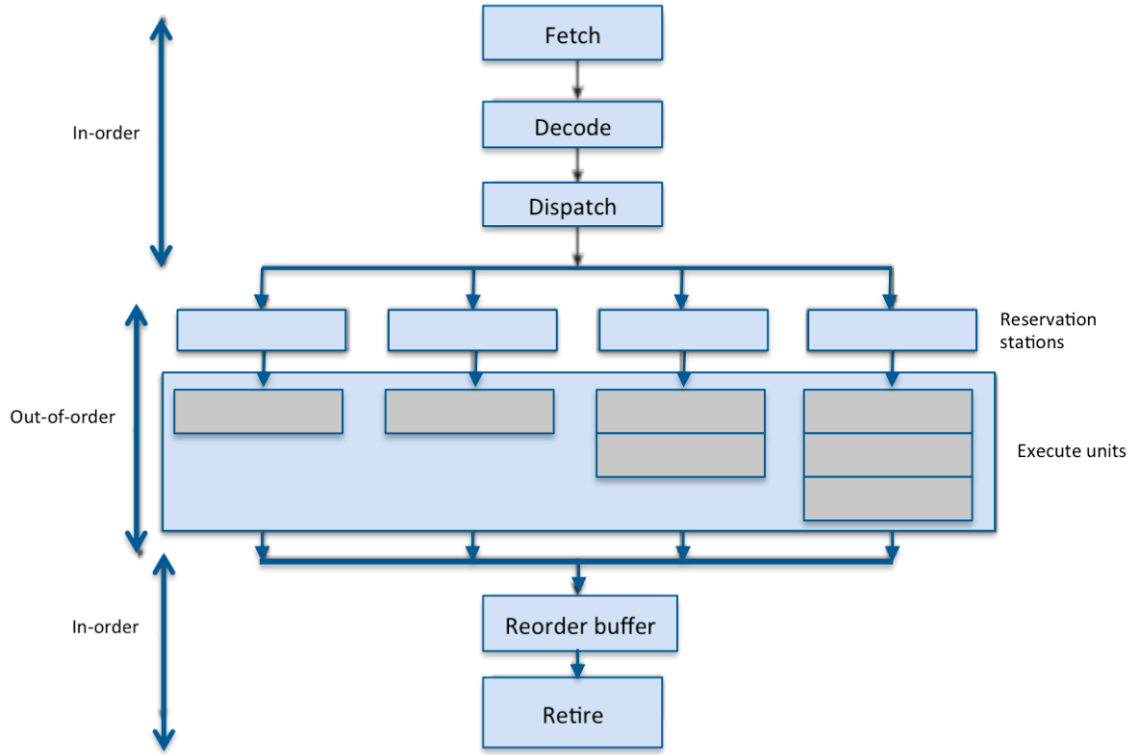
Superscalar Operation



Even with in-order issue, instructions may complete out-of-order

e.g. An add or shift may be issued after a lw or floating point instruction, but may finish executing first.

Superscalar Operation



Reorder buffers hold completed instructions

ensure that the program meaning or outcome is preserved

instructions that complete out-of-order are retired in-order

retire means writing the result to the correct register

Superscalar Operation

Possible issue/completion options:

- in-order issue & in-order completion
  - used in scalar pipeline
- in-order issue & out-of-order completion
- out-of-order issue & in-order completion
- out-of-order issue & out-of-order completion

Changing execution order can cause other types of data dependencies

Types of Dependencies

1. True data dependency (RAW) read after write

```
Sub  $3, $2, $4
Add  $5,$3,$4        # must read $5 after it is written by sub
```

2. Antidependency (WAR) write after read

```
or   $2, $4, $3
sub  $4, $5,$6     # can't write $4 until after the or reads $4
```

The or may be stalled waiting on a resource or on $3

Types of Dependencies

3. Output dependency (WAW) write after write

```
sub   $3, $2,$4
add   $5,$6,$5
slt   $3, $7,$0    # must write $3 after sub writes $3
```

Register Renaming

The use of registers can create apparent dependencies

1. or $3, $3,$5
2. add $4,$3,$2
3. add $3, $5, $2
4. sub $7,$3,$4

WAW exists between 3. and 1. (both write $3)

WAR exists between 3. and 2. (3. overwrites $3 input to 2.)

RAW exists between (1. & 2.), (2. & 4.), (3. & 4.)

Register Renaming

The use of registers can create apparent dependencies

1. or $3_a, $3_a,$5
2. add $4,$3_a,$2
3. add $3_b, $5, $2
4. sub $7,$3_b,$4

3_a and 3_b are unrelated, so a different register can be used for 3_b

- Eliminates false dependencies
- Register renaming performs this dynamic substitution

Register renaming adapts to larger register sets

- Applications use more registers without having to be recompiled
- Recompilation is required without register renaming feature

Multiple Instruction Issue

Instruction-level Parallelism (IPL)
the ability to execute multiple instructions together

Pipelining provides IPL by overlapping instruction execution

Superscalar systems replicate hardware to provide IPL
multiple instructions can be sent through separate pipelines
multiple units can execute instructions simultaneously

Multiple issue means starting more than 1 instruction at a time

Static vs. Dynamic Multiple Issue

Static multiple issue is based on:

- compiler deciding which instruction can execute together
- Independent instructions are grouped into packets or bundles
- This packaging occurs prior to execution (compile time)
- Bundles or packets are assigned to issue slots
- These are also called very long instruction words (VLIW)
- Packets are issued within a single clock cycle
- Packed instructions must map to separate resources
  - Instruction mix may be restricted

Static vs. Dynamic Multiple Issue

Dynamic multiple issue is based on:

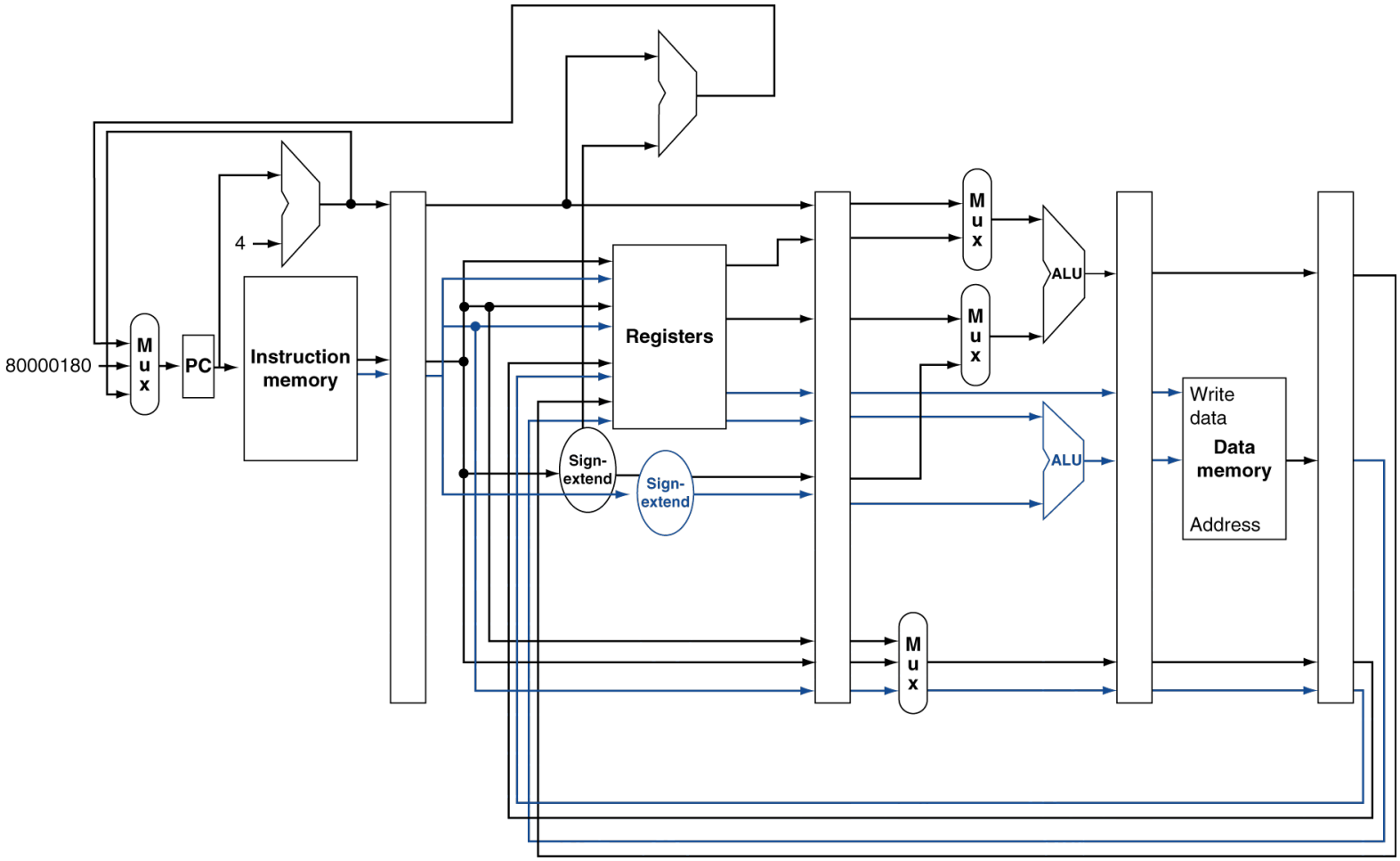
- Control unit deciding which instruction can execute together
- Decisions are made by hardware during execution
- Based on hazards (data, resources, etc.)
- Compiler may assist by reordering machine instructions

MIPS Static Multiple Issue

Consider a 2-issue MIPS processor

- Two instructions are fetched and executed together
- Requires 64-bit CPU-to-memory bus
- One can be an integer or branch instruction
- The other can be a load or store instruction
- Instructions are always issued in pairs
- Nop instruction replaces an unused slot within pair
- Required stalls holdup both instructions in a pair

Dual-issue MIPS



Extra resources: updated register file, unit to compute memory address, 2nd sign extension unit

Hazards in 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

```
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:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	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 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- "Superscalar" processors
- CPU selects instructions to execute each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though compiler may still help
  - Code semantics ensured by the CPU

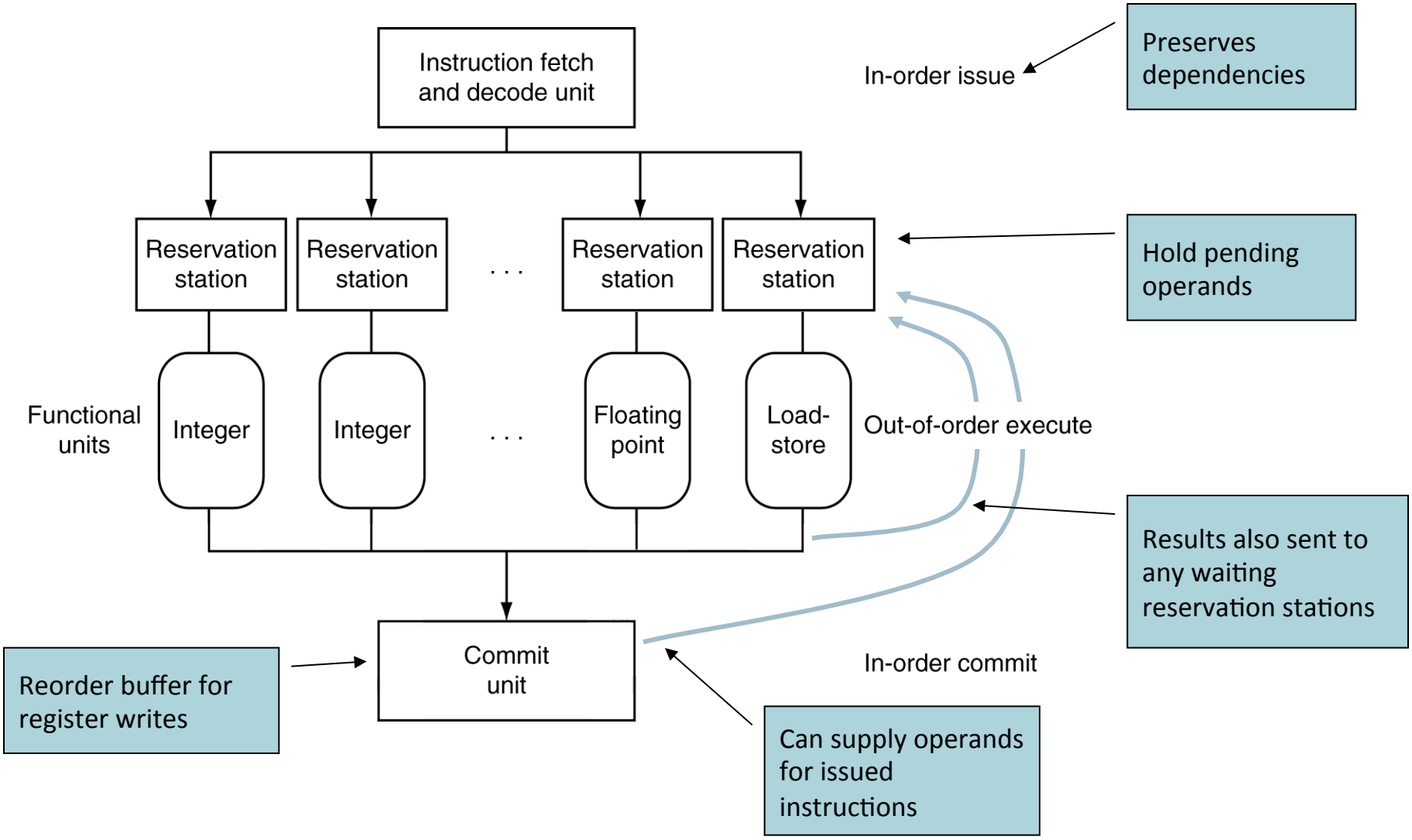
Dynamic Scheduling

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

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

  - Can start sub while addu is waiting for lw

Dynamic Scheduling



Register renaming

- Register renaming via reservation stations (RS) & ROB
  - ROB is reorder buffer
- On instruction issue to reservation station
  - copy available operand to reservation station
  - from register file or from ROB
    - Once done, register can be overwritten
  - When available, operands are provided to RS
    - Come from the function unit producing the result
    - Register update may not be required

Superscalar Operation

Scheduling Example

```
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:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	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

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

```
Loop: addi $s1, $s1,-16
      lw   $t0, 16($s1)
      addu $t0, $t0, $s2
      sw   $t0, 16($s1)
      lw   $t1, 12($s1)
      addu $t1, $t1, $s2
      sw   $t1, 12($s1)
      lw   $t2, 8($s1)
      addu $t2, $t2, $s2
      sw   $t2, 8($s1)
      lw   $t3, 4($s1)
      addu $t3, $t3, $s2
      sw   $t3, 4($s1)
      bne  $s1, $zero, Loop
```

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

(a 0 displacement & the orignal value in $s1 are used in the first lw instruction)

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- "Superscalar" processors
- CPU selects instructions to execute each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though compiler may still help
  - Code semantics ensured by the CPU

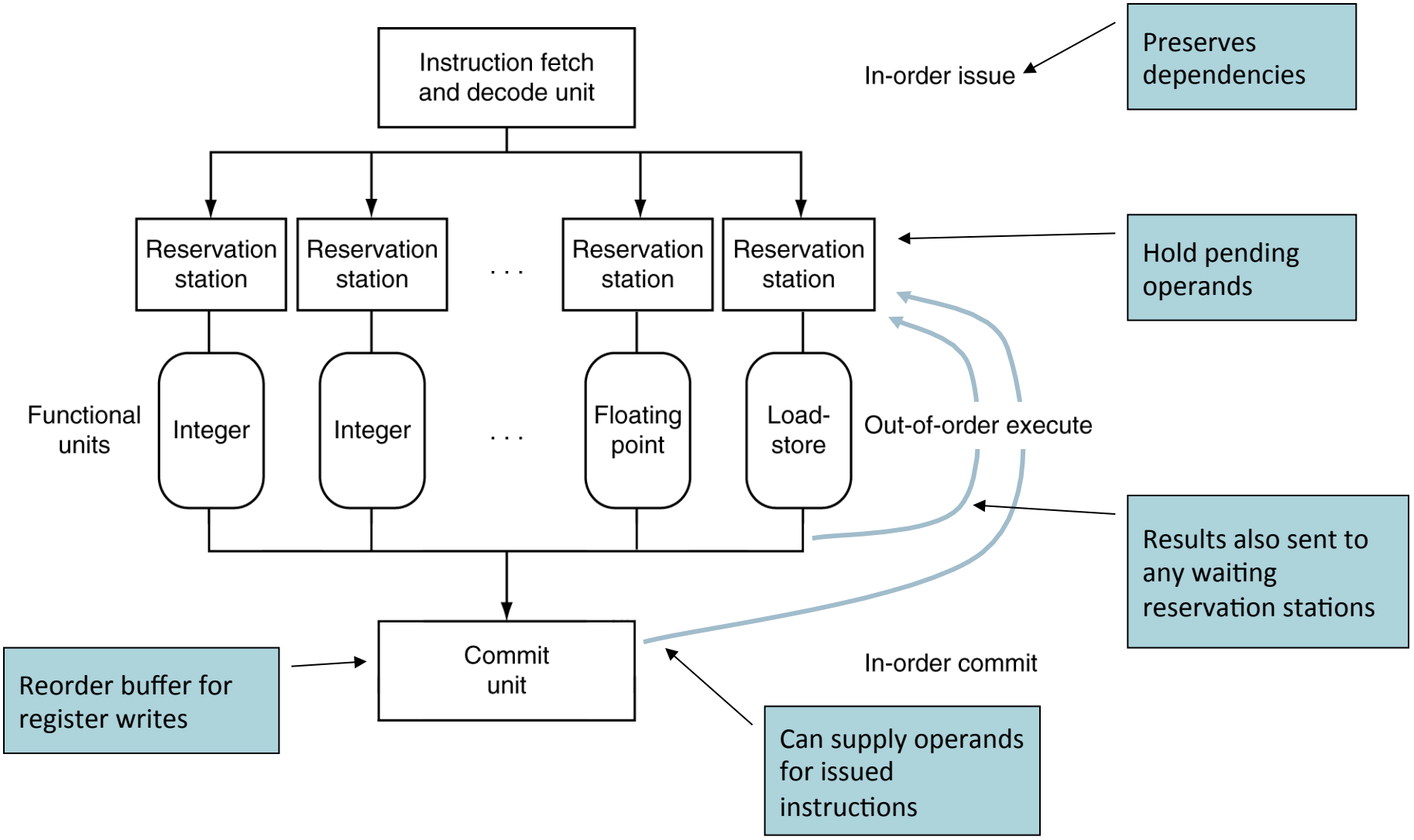
Dynamic Scheduling

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

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

  - Can start sub while addu is waiting for lw

Dynamic Scheduling



Register renaming

- Register renaming via reservation stations (RS) & ROB
  - ROB is reorder buffer
- On instruction issue to reservation station
  - copy available operand to reservation station
  - from register file or from ROB
    - Once done, register can be overwritten
  - When available, operands are provided to RS
    - Come from the function unit producing the result
    - Register update may not be required