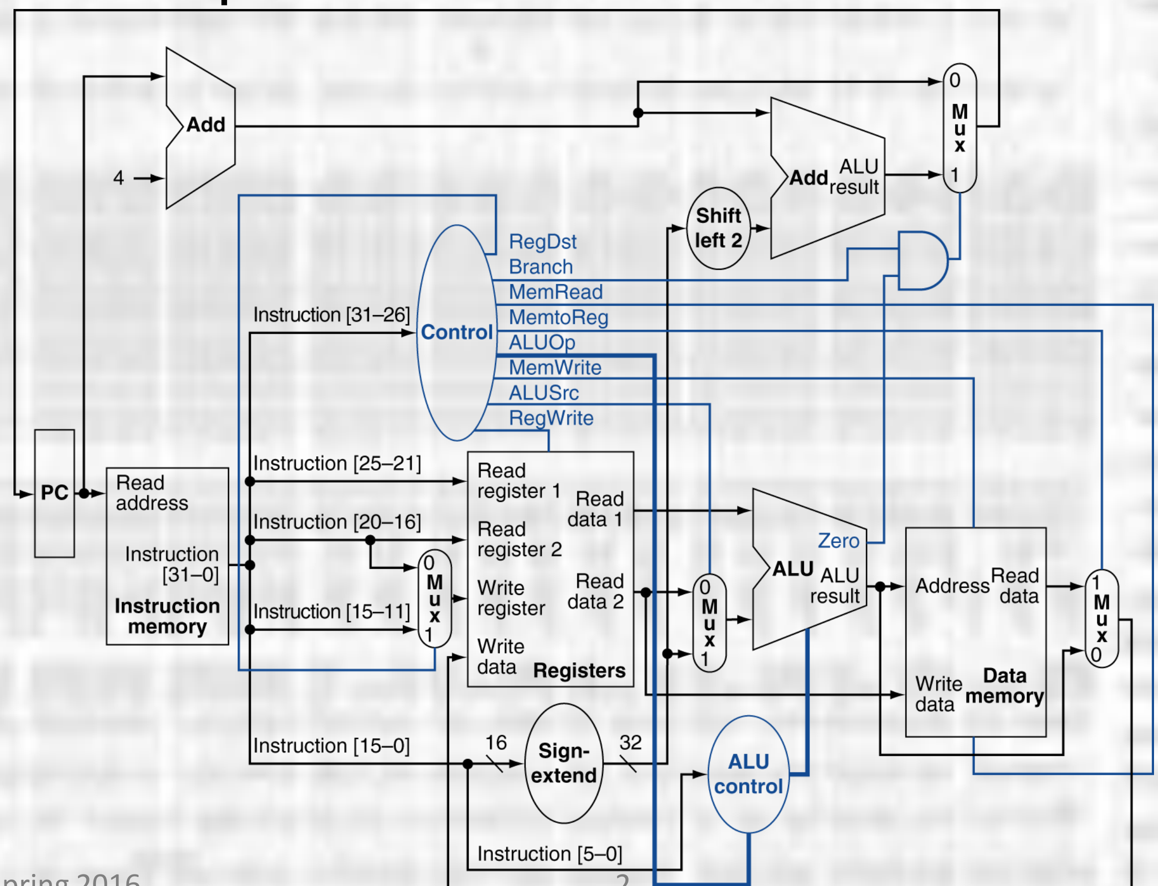
ELE 455/555
Computer System Engineering

Section 2 – The Processor

Class 5 – Parallel Processing and Pipelines

Pipelining
Overview

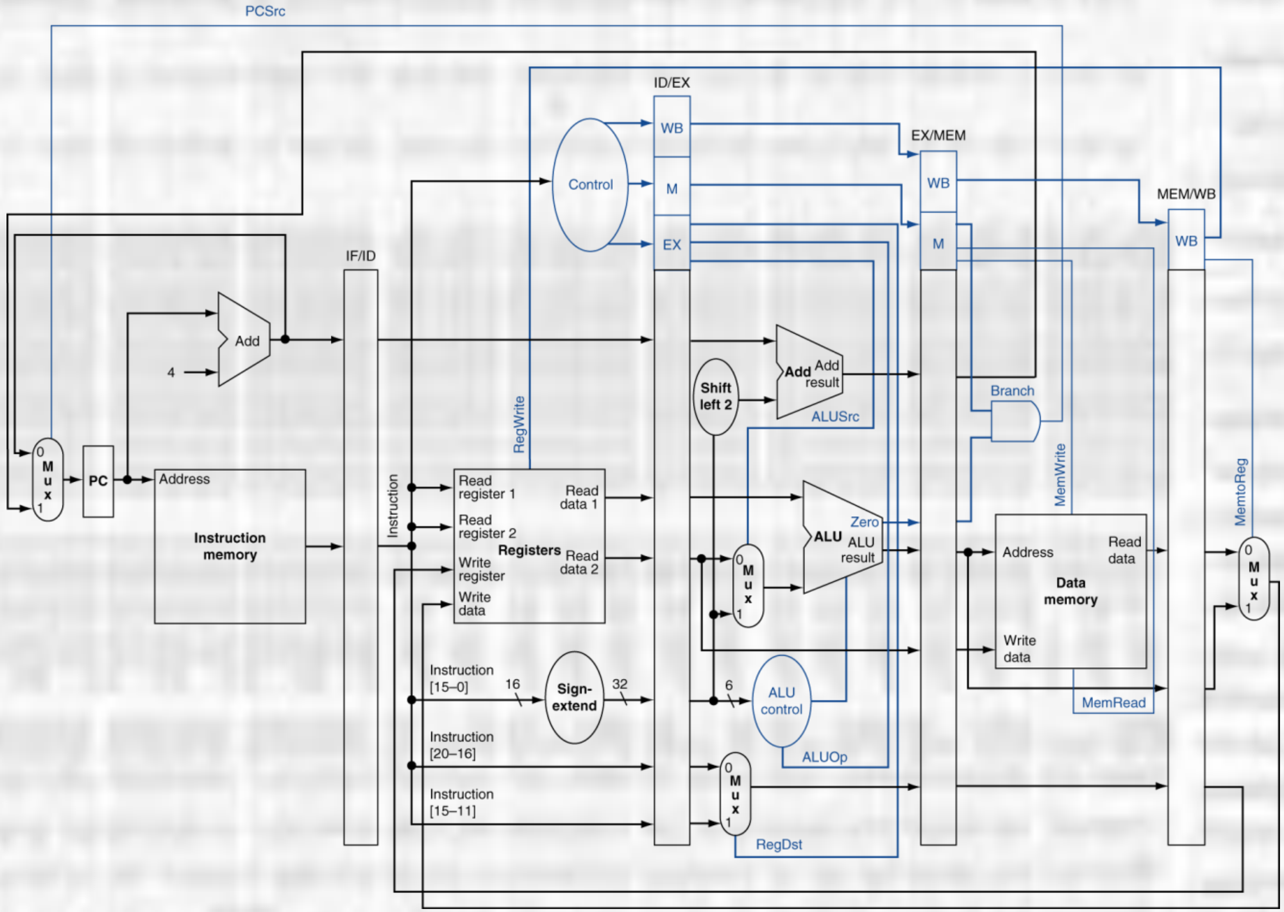
- Simple Datapath



Pipelining

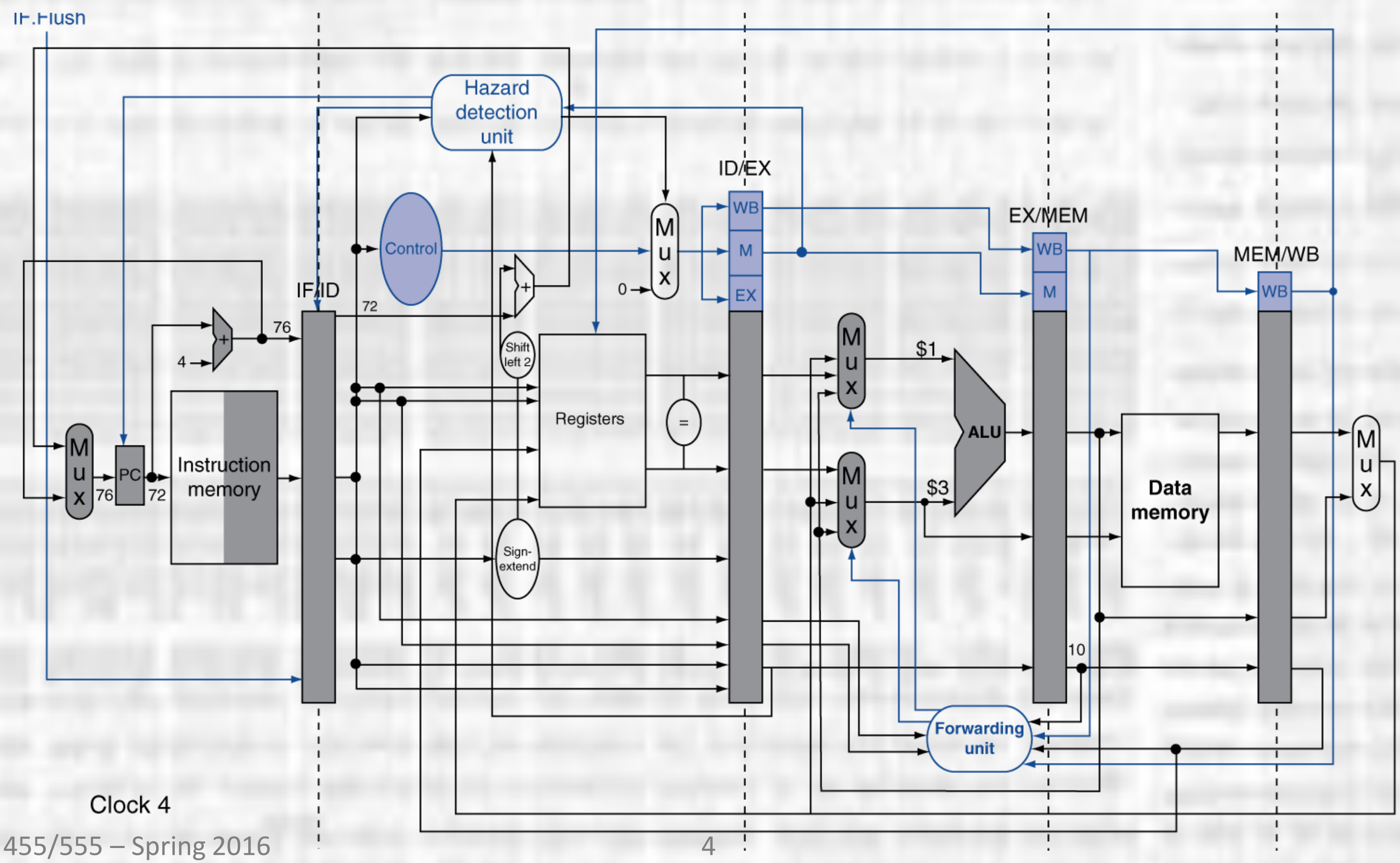
Overview

- Simple Pipeline



Pipelining
Control Hazards

- Pipeline with Hazard Detection and Forwarding



Pipelining

Exceptions

- Exceptions are unplanned events
  - Interrupts and Exceptions – sometimes used interchangeably
  - Interrupts
    - events originated from outside the processor core
      - external interrupt pin
      - A/D complete interrupt
      - Input capture interrupt
  - Exceptions
    - events originated from inside the processor core
      - software interrupt (OS)
      - illegal instruction
      - overflow

Pipelining
Exceptions

- Response to Exceptions

  - Must save the current instruction
    - May be the offending instruction
    - If not – it is the instruction you want to return to
    - Saved in the Exception Program Counter (EPC)

  - Transfer control of the processor to an exception handing routine
    - Need to know what the exception is
      - cause register
      - vectored interrupt
    - Correct the issue if possible
      - Return to program execution (using value in EPC)
    - If not correctable
      - Kill program
      - Return to program execution (using value in EPC)
      - Abort

Pipelining
Exceptions

- Cause Register
  - Register with a bit to indicate each identifiable exception type
  - Exception routine (at a fixed memory location)
    - Reads the Cause Register to determine what type of exception has occurred
    - Responds to the identified exception
  - Can support more than 1 simultaneous exception
    - Routine can build priority into it’s response
  - MIPS uses this approach

Pipelining
Exceptions

- Vectored Interrupt
  - Each interrupt type points the PC to a specific memory location
  - Exception routines (at pre-defined memory locations)
    - Know the cause because each is targeted at a specific cause
    - Responds to it's specific exception
  - Can support more than 1 simultaneous exception
    - New vectors interrupt running exception routines
    - Logic prioritizes exception on the same clock cycle

Pipelining

Exceptions

- Exceptions in a Pipeline

  - Control Hazard

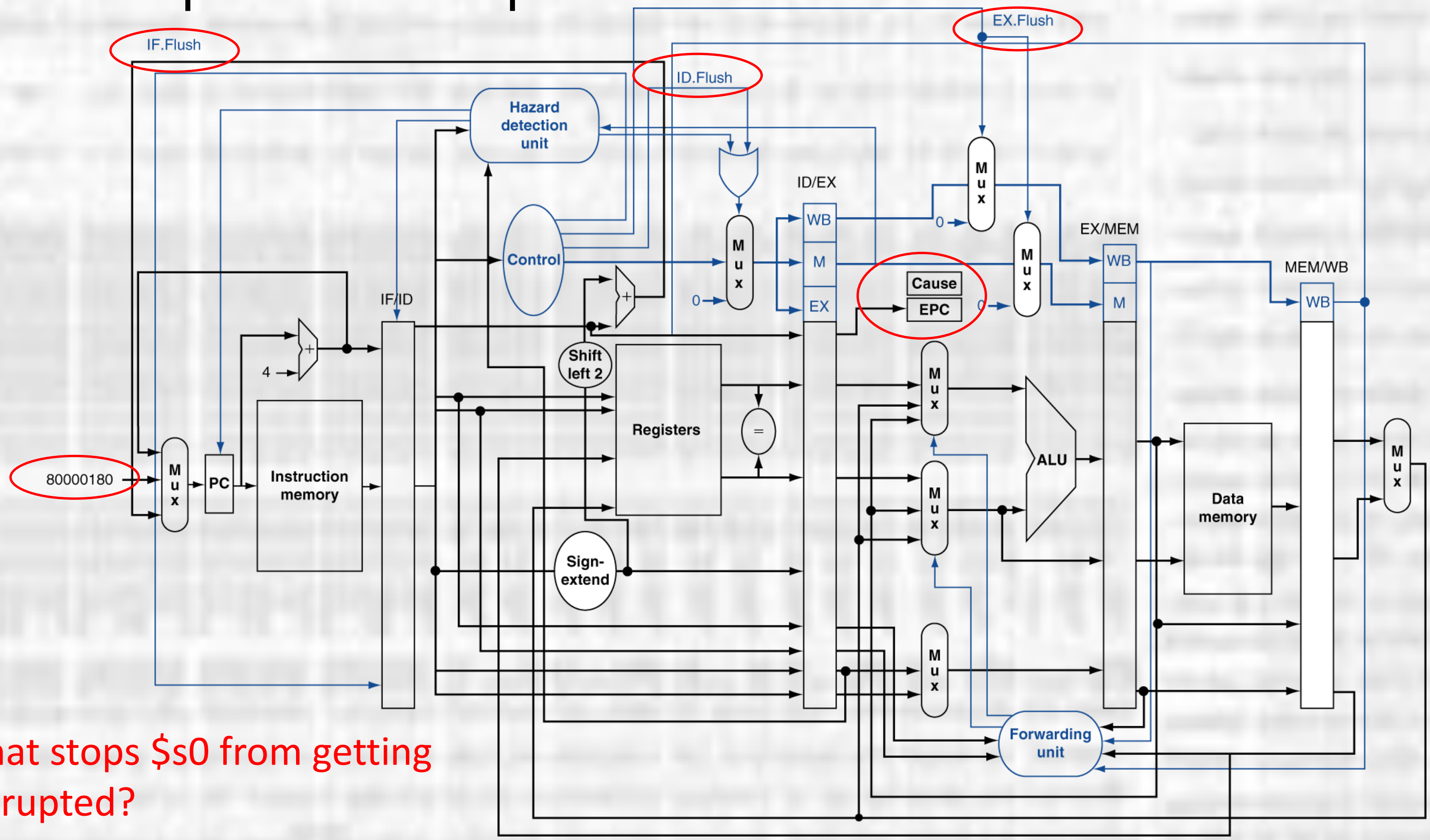
  - Consider an overflow
    add $s0,$s0,$t0

    - Prevent $s0 from being stored with the wrong result
    - Complete any instructions in front of the add in the pipeline
    - Flush any instructions after the add in the pipeline
    - Save PC for the add instruction into the EPC
    - Save the cause in the Cause Register
    - Transfer control to the exception handler routine

  - Looks almost like a mis-predicted taken branch

Pipelining
Exceptions

- Exceptions in a Pipeline



What stops $s0 from getting corrupted?

Pipelining

Exceptions

- Exceptions in a Pipeline - 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)
	…	

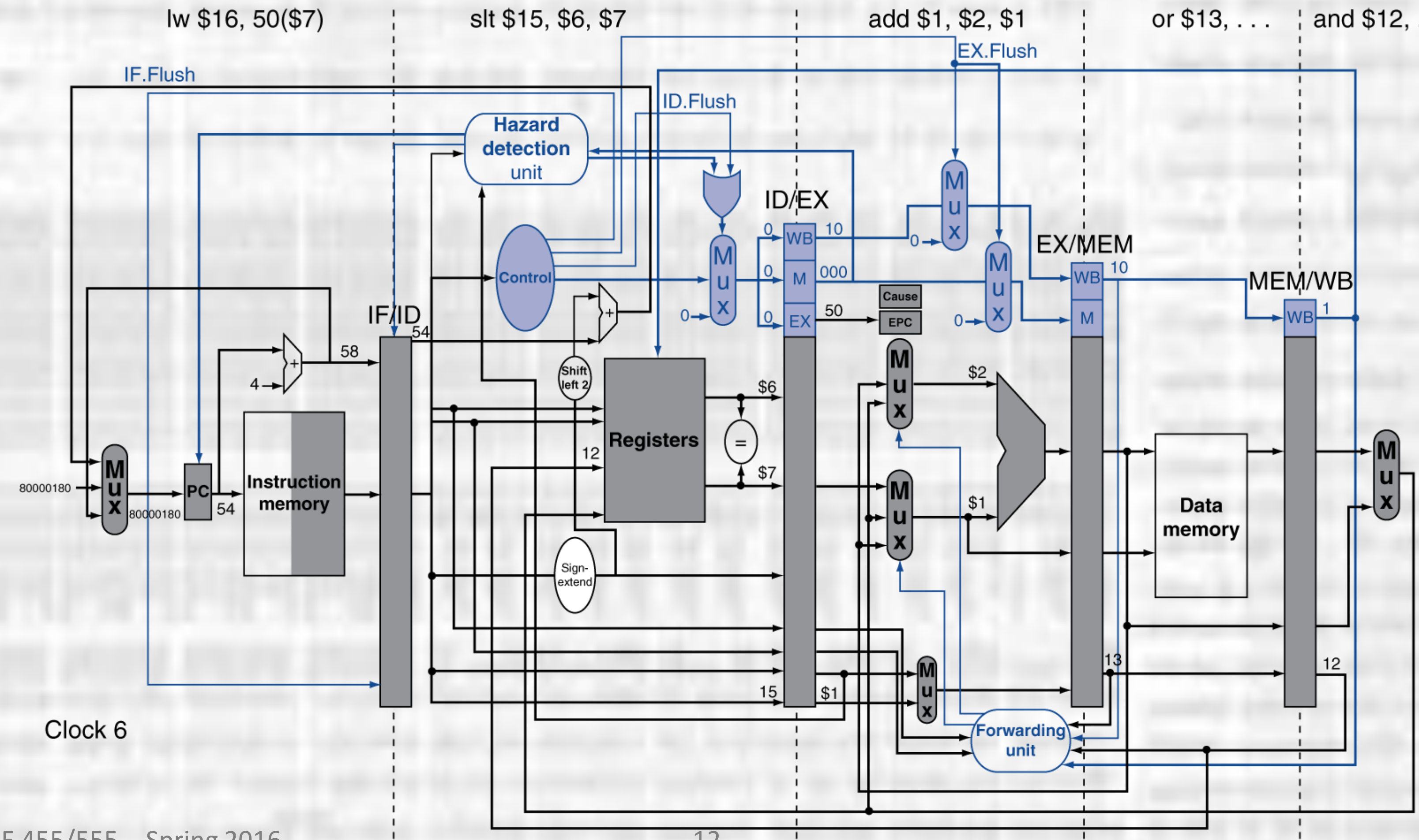
  - Handler

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

Pipelining

Exceptions

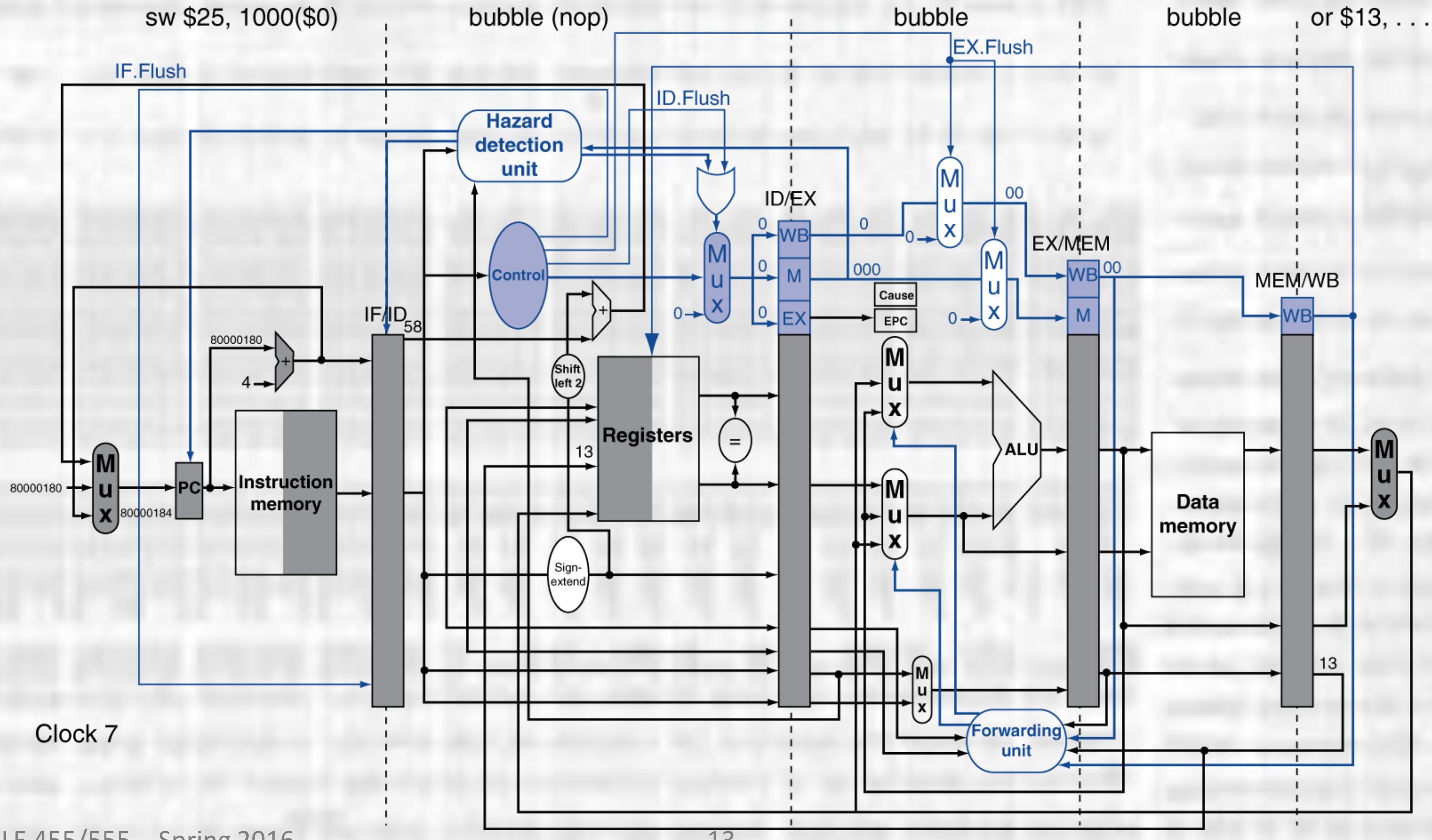
- Exceptions in a Pipeline - example



Pipelining

Exceptions

- Exceptions in a Pipeline - example



Pipelining

Exceptions

- Multiple Exceptions in a Pipeline
  - Pipelining overlaps multiple instructions
    - Could have multiple exceptions at once
  - Simple approach: deal with exception from earliest instruction
    - Flush subsequent instructions
    - “Precise” exceptions
  - In complex pipelines
    - Multiple instructions issued per cycle
    - Out-of-order completion
    - Maintaining precise exceptions is difficult!

Pipelining
Exceptions

- Multiple Exceptions in a Pipeline
  - Pipelining overlaps multiple instructions
    - Could have multiple exceptions at once
  - Simple approach: deal with exception from earliest instruction
    - Flush subsequent instructions
    - “Precise” exceptions
  - In complex pipelines
    - Multiple instructions issued per cycle
    - Out-of-order completion
    - Maintaining precise exceptions is difficult!
    - → Imprecise Exceptions
      - Let the handler figure it out!

Instruction Level Parallelism

Overview

- Instruction Level Parallelism (ILP)
  - Allows more instructions to complete per period of time
    - → higher throughput → higher performance
  - Pipelining is our first example of ILP
    - Increase performance by making deeper pipelines
      - Cut the work into smaller pieces and run the clock faster
      - More instructions complete for a fixed unit of time
    - There is a limit
      - Deeper pipelines have higher costs for branches and exceptions

Instruction Level Parallelism

Overview

- ILP – Multiple Issue

  - Create multiple parallel pipeline stages
  - Start multiple instructions on each clock cycle

  - Static Multiple Issue
    - Compiler organizes instructions into groups
    - Creates “issue slots” – instructions that can be executed in parallel
    - Compiler responsible for detecting and avoiding hazards

  - Dynamic Multiple Issue
    - CPU examines the incoming instruction stream
    - Groups instructions into issue slots
    - CPU responsible for detecting and avoiding hazards
      - Compiler can make the job easier

Instruction Level Parallelism

Overview

- ILP – Speculation
  - Predict what to do with each instruction
    - Start as soon as possible
    - Wait for a hazard to clear
  - Check to see if prediction was correct
    - If right
      - Continue with execution
    - If wrong
      - Back-up and choose the other path – not easy

Instruction Level Parallelism

Overview

- ILP – Speculation
  - SW Speculation
    - Compiler tries to guess at what path the code will take
    - Re-orders instructions to allow multiple issue
    - Adds corrective code for the cases where it is wrong
  - HW Speculation
    - Buffers the results from any speculative paths
    - Releases the results once it is known that the speculation was correct
      - allows WB
    - Dumps the results and backs up if the speculation was incorrect
      - flushes anything still in the pipeline
      - re-issues the correct instructions

Instruction Level Parallelism

Overview

- ILP – Speculation
  - Very Complex
    - Complexity for the compiler
    - Significant HW
    - Can create exceptions
      - speculate on a branch where the address is incorrect if the prediction is incorrect
      - without speculation – the exception would never occur
      - Buffer the exceptions and wait until the status of the speculation is known
    - Must work well or it hurts more than it helps

Instruction Level Parallelism

Multiple Issue

- Static Multiple Issue
  - Compiler groups instructions into issue packets
    - Must understand what resources are available
    - Must avoid creating hazards with-in a packet
    - Must work to avoid any hazards between packets
      - Some may be avoided by the HW – eg. forwarding
    - Pad the packets with NOPs when all else fails
  - Looks like a VLIW pipeline
    - add2 $1, $2, $3, $4 ; $1 = $1 + $2, $3 = $3 +$4
    - ador $1, $2, $3, $4 ; $1 = $1 + $2, $3 = $3 or $4

Instruction Level Parallelism

Multiple Issue

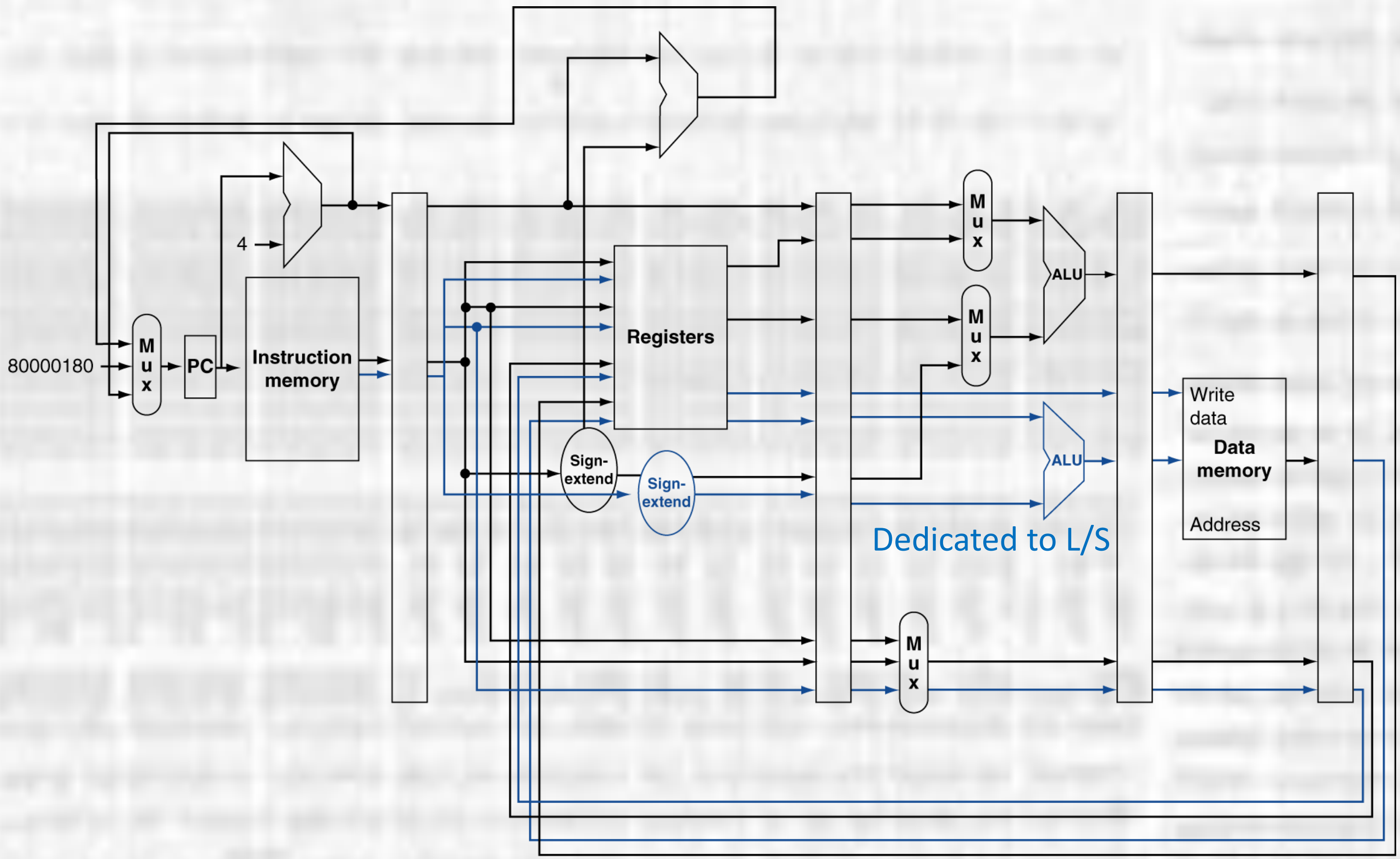
- Static Dual Issue
  - Two execution paths
    - Two issue packets
      - ALU/branch path
      - Load/Store path
    - Pad with nop if one or the other cannot be issued

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			nop	nop	nop	nop	nop
n + 20	Load/store			IF	ID	EX	MEM	WB

Instruction Level Parallelism

Multiple Issue

- Static Dual Issue



Instruction Level Parallelism

Multiple Issue

- Static Dual Issue - 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

** 5 instructions complete in 4 clock cycles → IPC=1.25 vs. theoretical IPC = 2
Also note – data hazard space is doubled

Instruction Level Parallelism

Multiple Issue

- Static Dual Issue – loop unrolling

  - Replicate the loop body to allow for additional parallelism
    - Reduces loop overhead
    - Increases code size

  - Requires register renaming
    - Use different registers for each un-rolled iteration
    - Requires enough additional registers to avoid aliasing the values
      - called name dependence

Instruction Level Parallelism

Multiple Issue

- Static Dual Issue – example with loop unrolling

```
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:	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

** 14 instructions complete in 8 clock cycles → IPC=1.75 vs. theoretical IPC = 2

Instruction Level Parallelism

Multiple Issue

- Dynamic Multiple Issue – (superscalar)
  - CPU can execute instructions “out of order”
  - CPU decides how many instructions to issue
    - limited by resources
    - avoid hazards
  - Must “commit” results in order
  - Compiler can help make the job easier

Instruction Level Parallelism

Multiple Issue

- Dynamic Multiple Issue

  - Consider

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

  - addu must wait for the lw to complete
    - our hazard correction can't fix this
  - sub has no dependencies so it can be issued in parallel with the lw

Instruction Level Parallelism

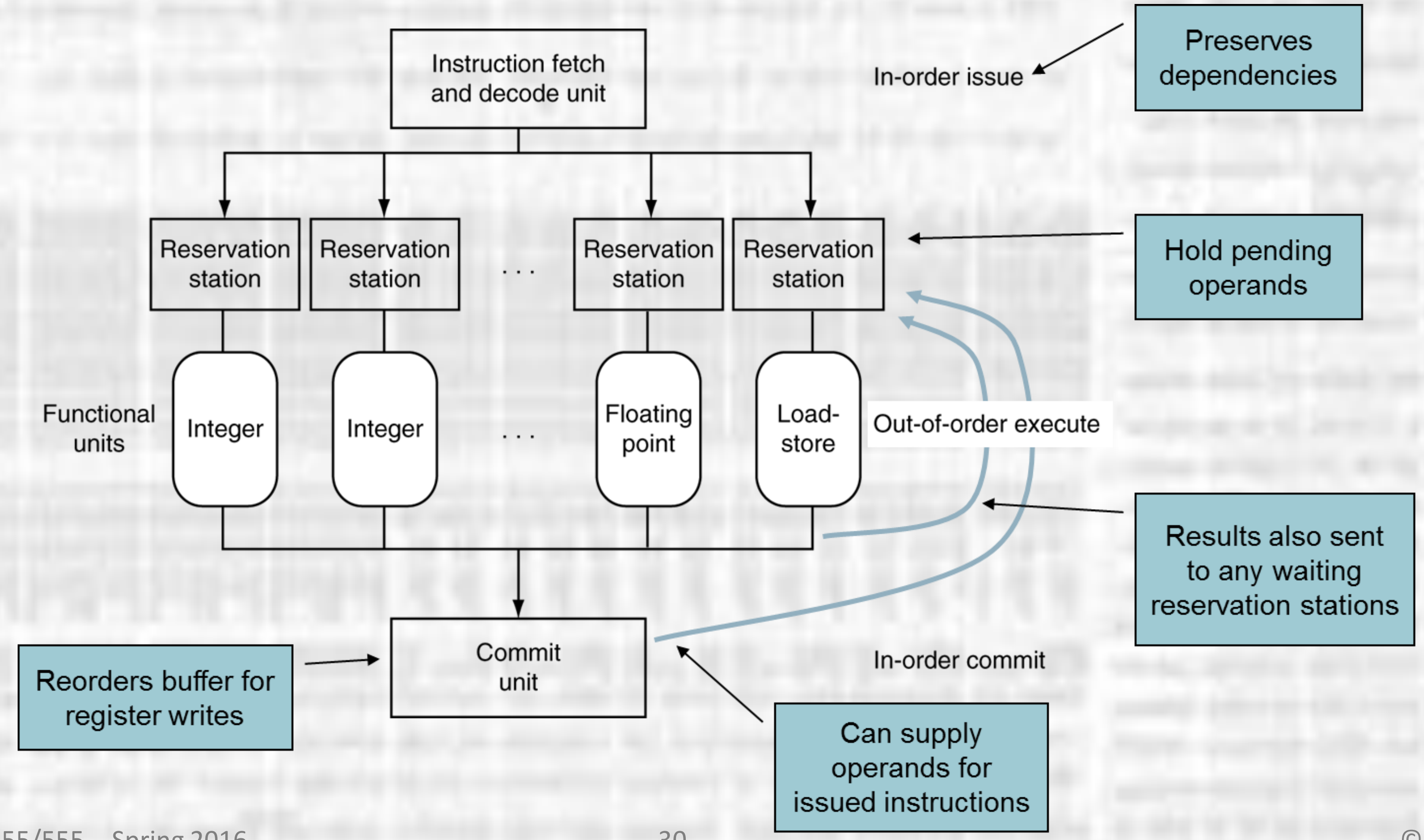
Multiple Issue

- Dynamic Multiple Issue
  - In order issue
    - keeps dependencies in line
  - Reservation station
    - Holds instruction until all dependencies are available
  - Functional Units
    - Execution units
    - May be duplicates
  - Commit
    - Hold on to any writes ready before the appropriate time
      - pending earlier instructions that were scheduled later

Instruction Level Parallelism

Multiple Issue

- Dynamic Multiple Issue

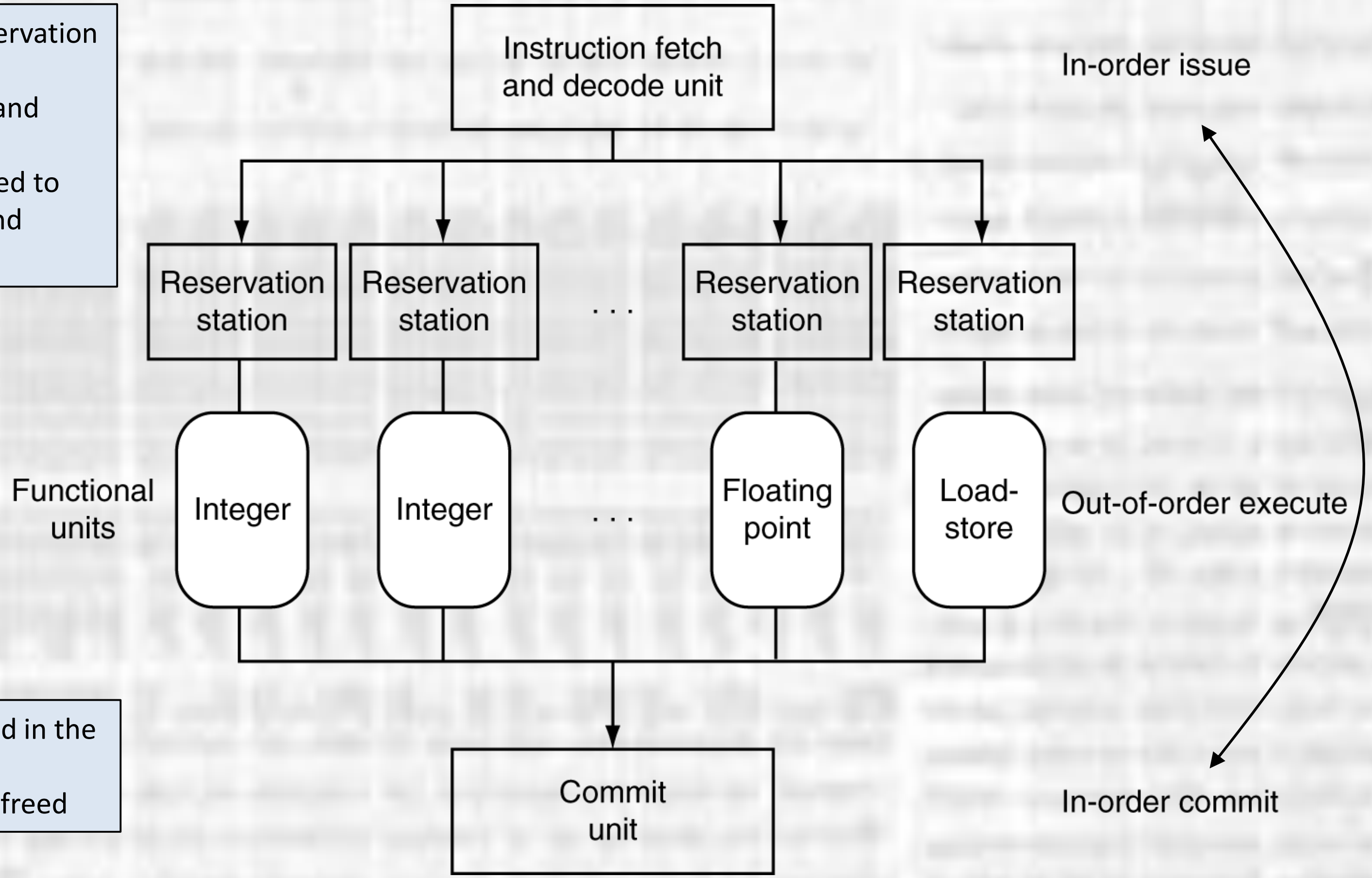


Instruction Level Parallelism

Multiple Issue

- Dynamic Multiple Issue – register renaming

Once the instruction is in the reservation station
-- available operands are copied and register is freed
-- unavailable operands are tracked to their pending execution unit and register is freed



If the waiting value is only needed in the reservation station
-- WB is cancelled and register is freed

Instruction Level Parallelism

Multiple Issue

- Dynamic Multiple Issue – speculation
  - Branches
    - Predict branch direction but don’t commit until result is known
  - Loads
    - Especially important once we start dealing with real memories (caches)
    - Predict the effective address
    - Load the value before completing waiting stores
    - Don’t commit until the prediction result is known

Instruction Level Parallelism

Putting it all together

- Cortex A8