Computer Organization and Design

Chapter 4 Part 1

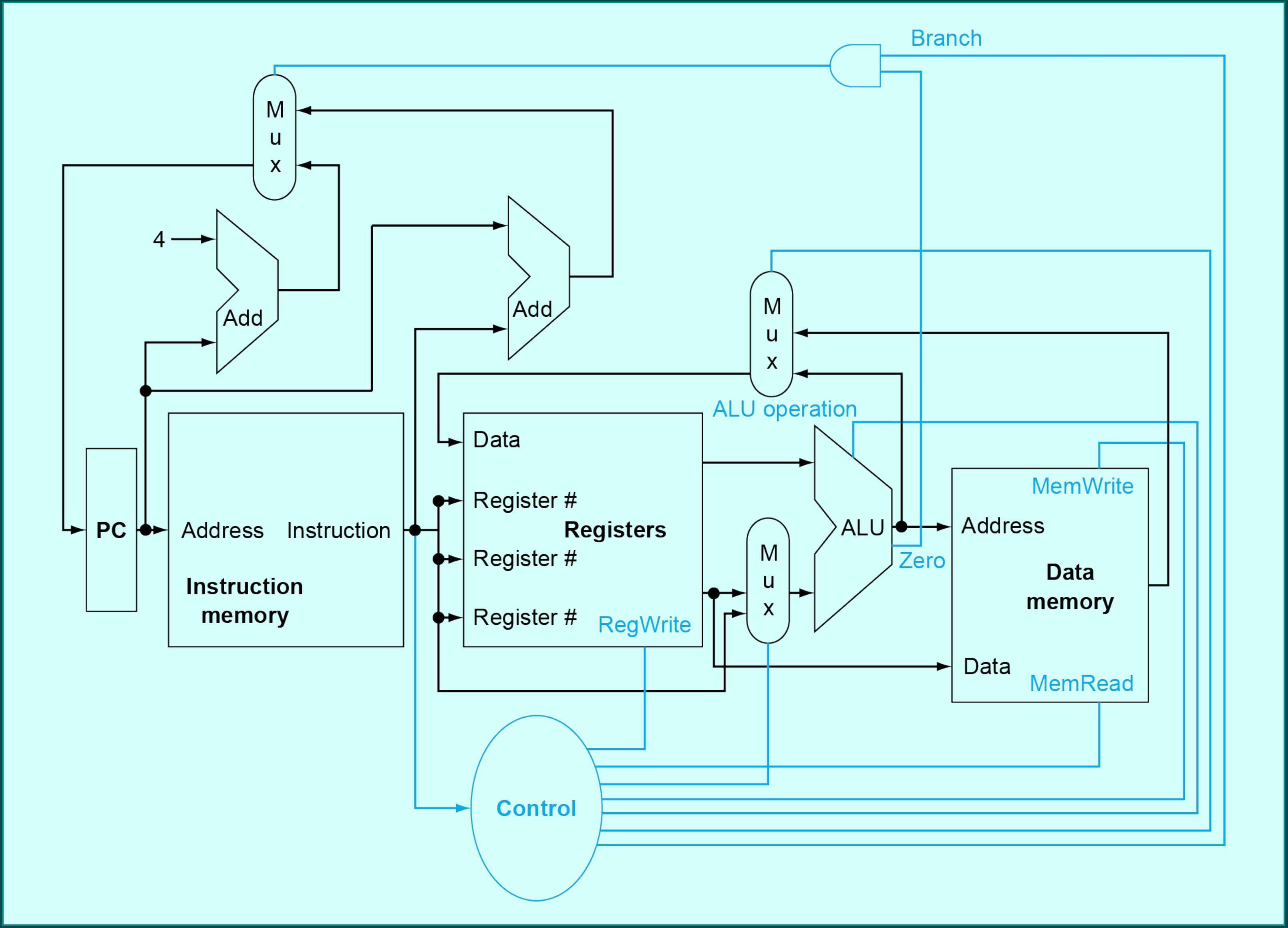
The Processor:

Datapath, Control

The Instruction Cycle

- The process that defines the steps for instruction execution in the computer
- Most all architectures implement the same basic process
  - Some may define it fewer or more steps
- RISC-V instruction cycle contains 5 steps
  - (1) Instruction fetch, increment PC
  - (2) Instruction decode, operand fetch
  - (3) Instruction execution, memory address computation, branch completion
  - (4) Memory access
  - (5) Register write-back or memory read completion

Basic RISC-V Datapath Implementation

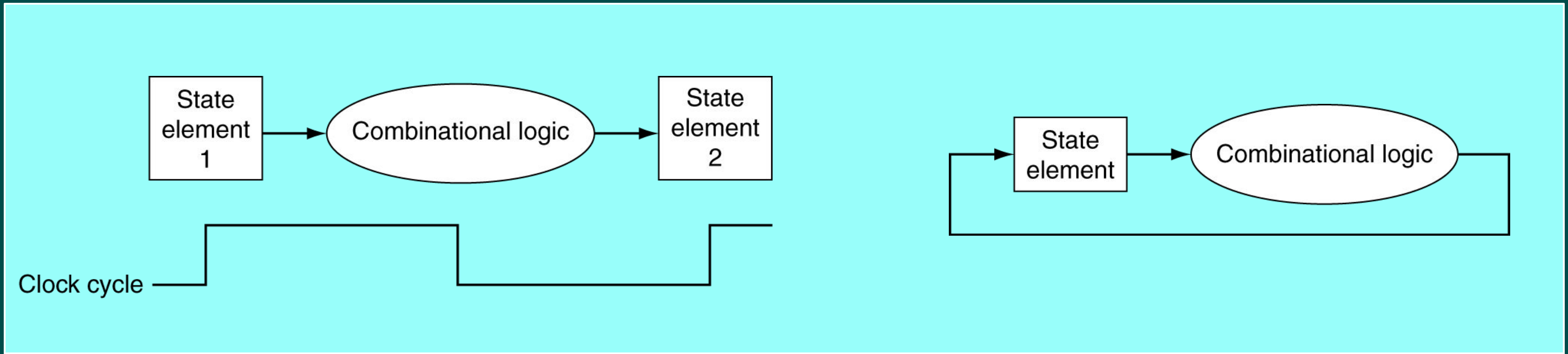


Logic Design Elements

- Two basic categories of logic circuits
  - Combinational elements
    - Operates on data
    - Output is a function of the current input
    - Example: ALU, multiplexers
  - State (sequential) elements
    - Storage component
    - Output is a function of both the stored value and input
    - Example: registers, memory

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period



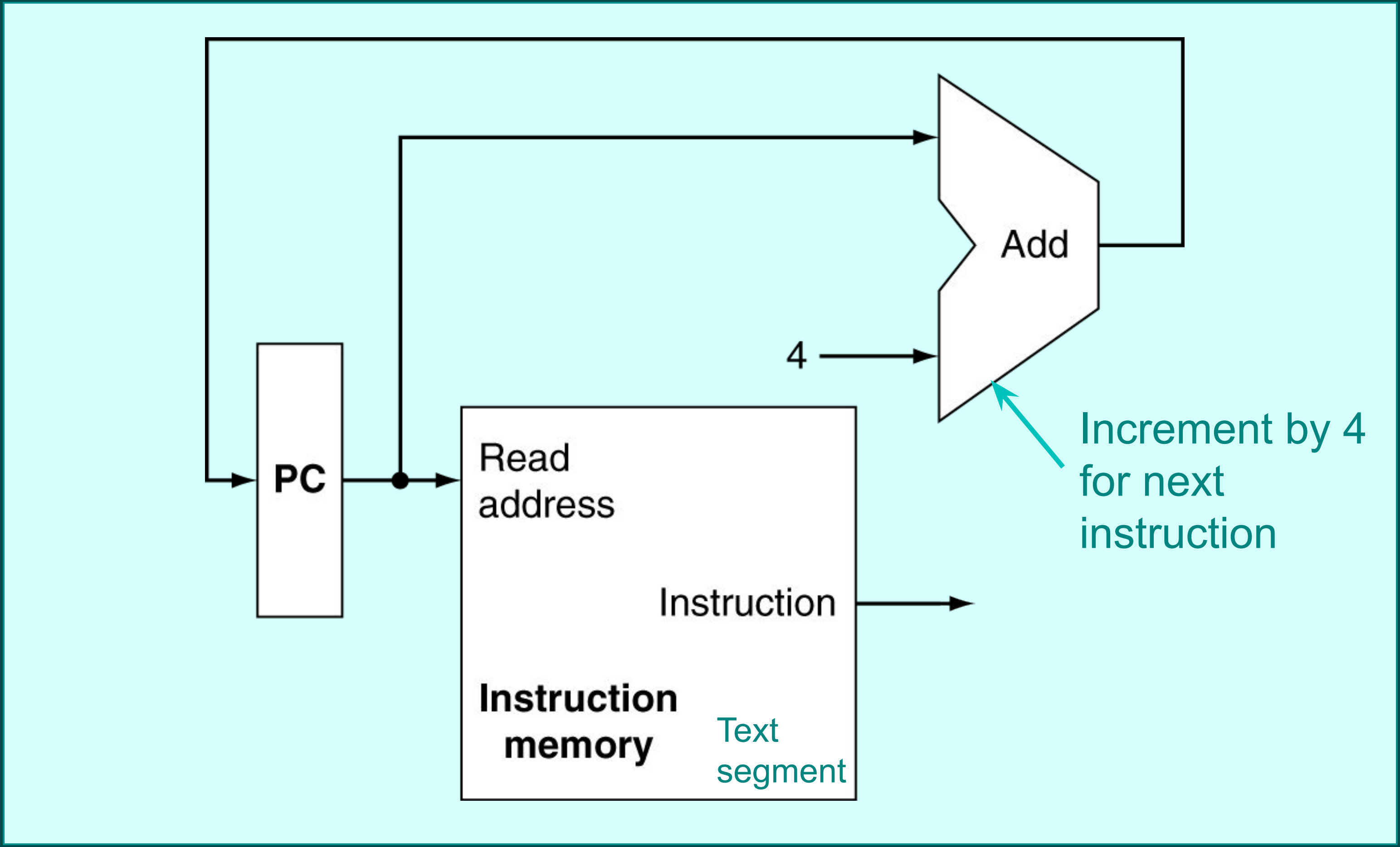
- Edge-triggered methodology allows contents of register to be read, send the value through combinational logic and write output to register in the same clock cycle.

Datapath Components

- The datapath is constructed with the components that implement the various instruction types
  - ALU, load/stores, branches
- Each step of the instruction cycle relates to a set of hardware components
  - Interconnecting the components for each step builds the datapath for the architecture
  - The datapath sequence follows the flow of the instruction cycle

Instruction Fetch

- Use the program counter to read an instruction from memory and increment the PC

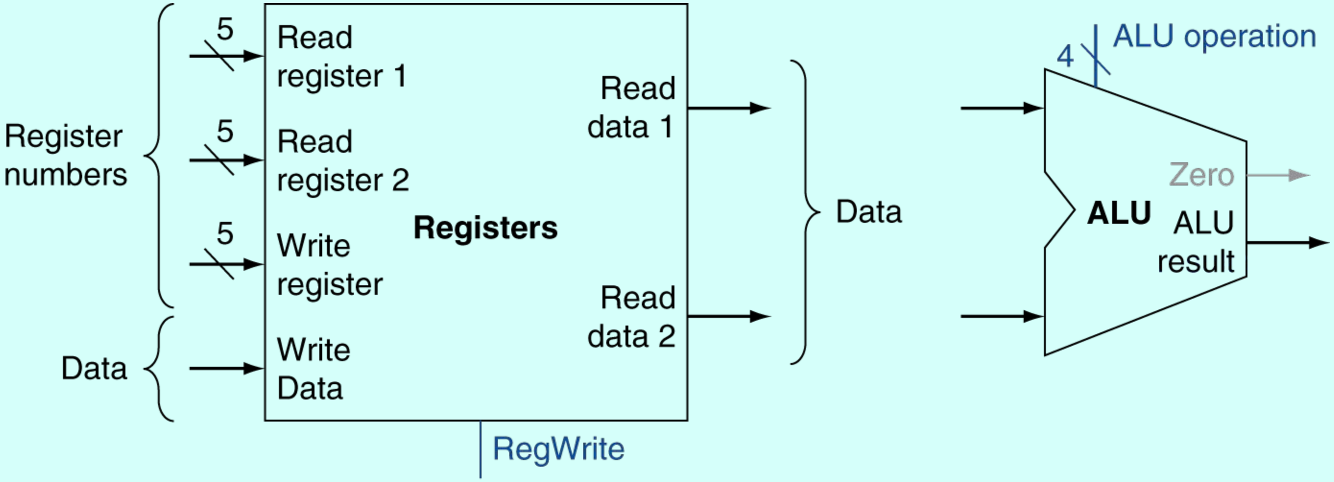


Increment by 4 for next instruction

Text segment

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

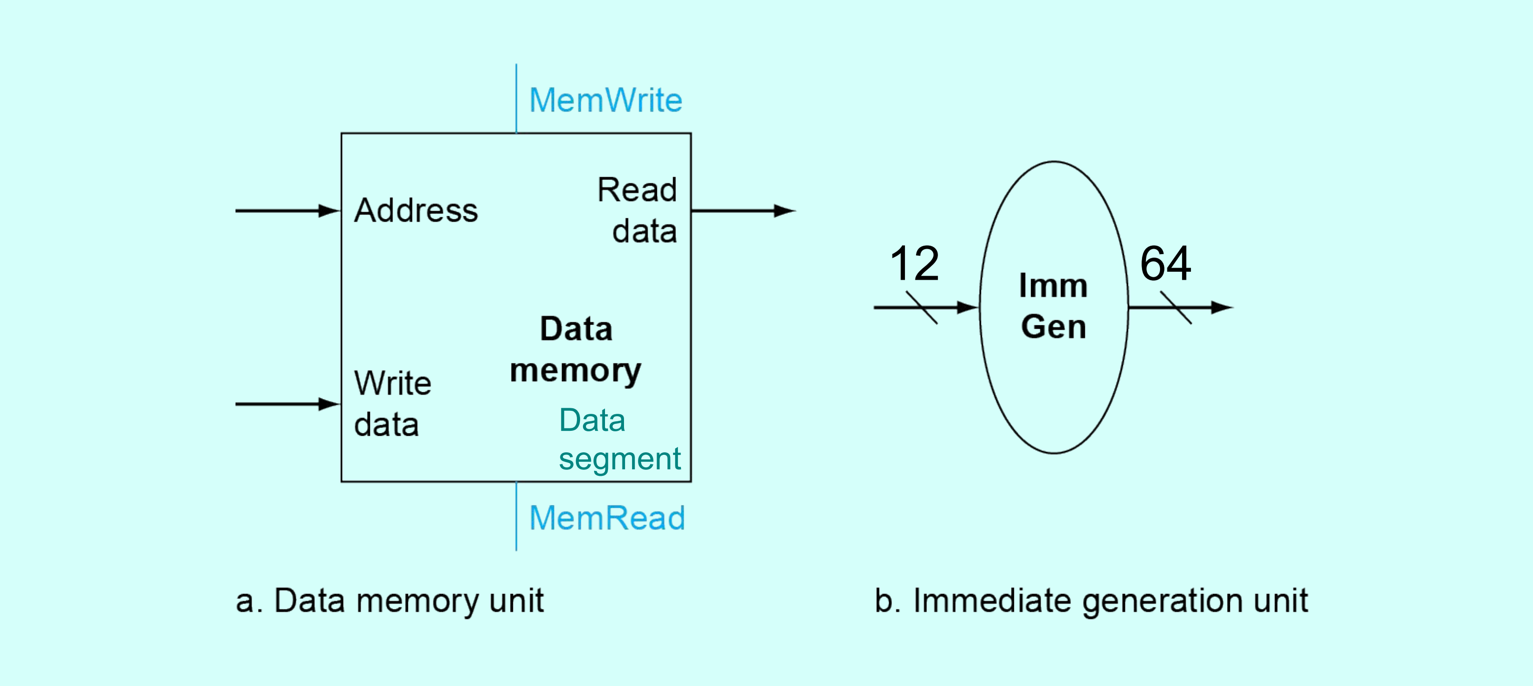
Access registers for input operands and to write result

b. ALU

ALU uses input values to perform operation specified

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset to 32 or 64 bits
- Load: Read memory and update register
- Store: Write register value to memory



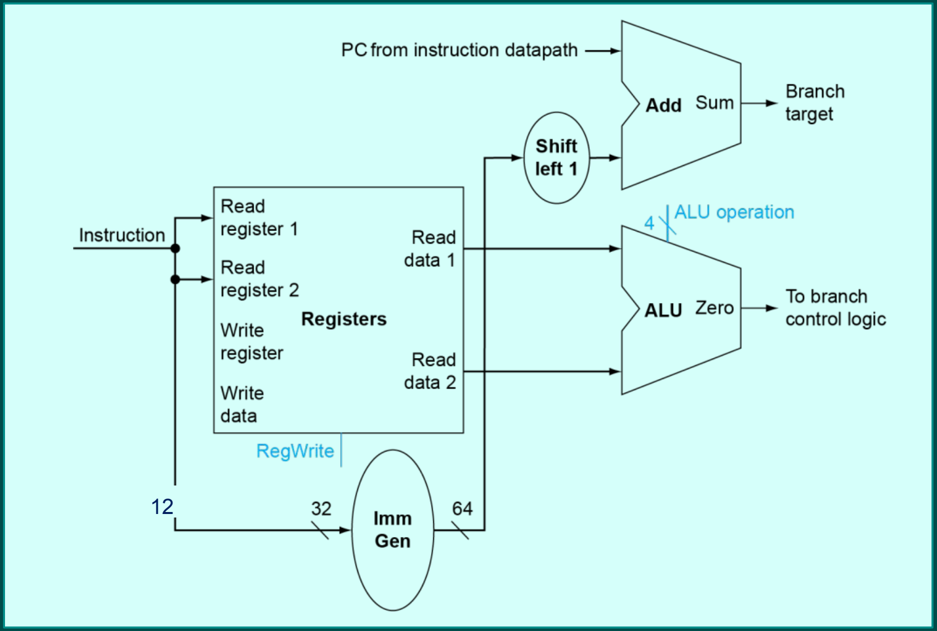
a. Data memory unit

b. Immediate generation unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
  - Calculate target address
- Sign-extend displacement (offset)
- Shift left 1 place (halfword displacement*)
- Add to PC value

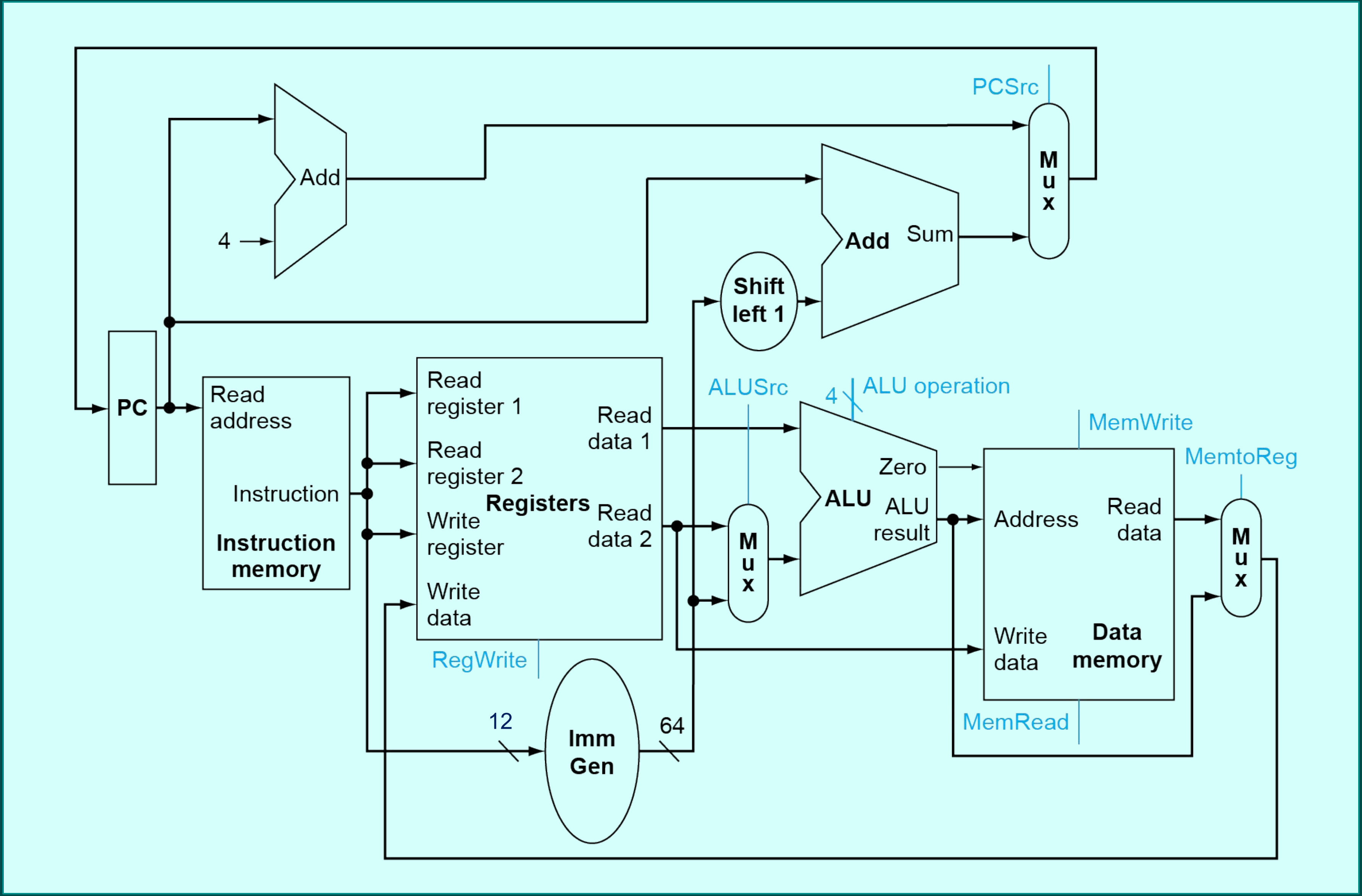
*The halfword displacement inserts the 0 that isn't physically stored in the 12 bit offset in the instruction.



Combining the Datapath Components

- This initial design of the datapath supports the core RISC-V architecture.
- Combining the components in such a way is intended to have each instruction execute in one clock cycle.
- Each datapath element can only do one function at a time
  - Hence, the instruction and data memories must be separate entities
- Use multiplexers where alternate data sources are used for different instructions

Full Datapath



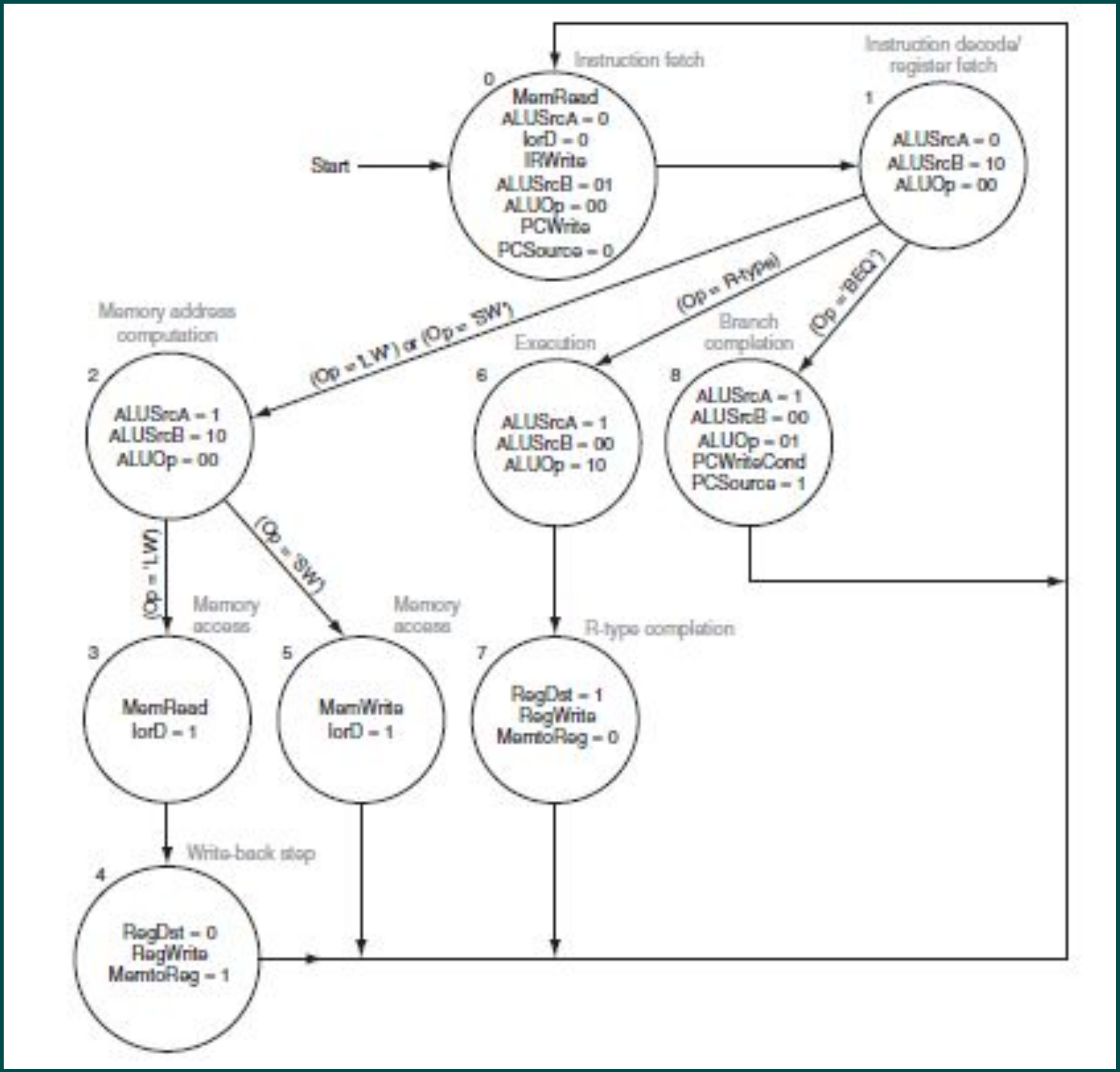
Control Signals

- The components of the datapath - various registers, the ALU, and memory - must be activated in a sequence with specific timing constraints
- Control signals perform the task of timing and activation
- Each control signal has a name based on the part of the datapath it controls
- Each instruction contains the binary code that represents what control signals need to be produced at what times
  - Operation, register or memory

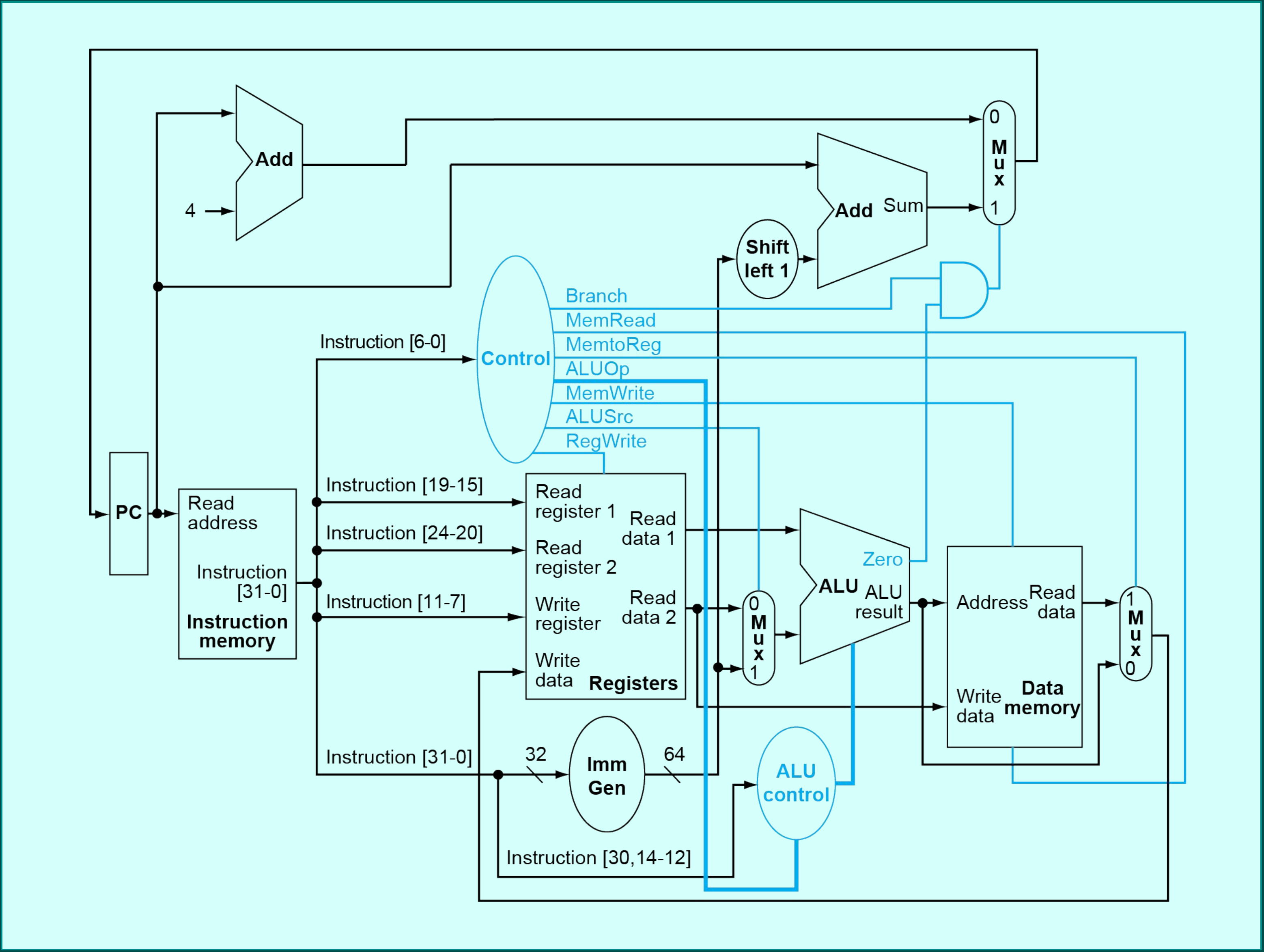
Finite State Machine for Control Unit Design

Depending on the instruction, the FSM defines the state and possible transitions to the next state.

Each state defines the control signals issued from that specific state.



Datapath with Control



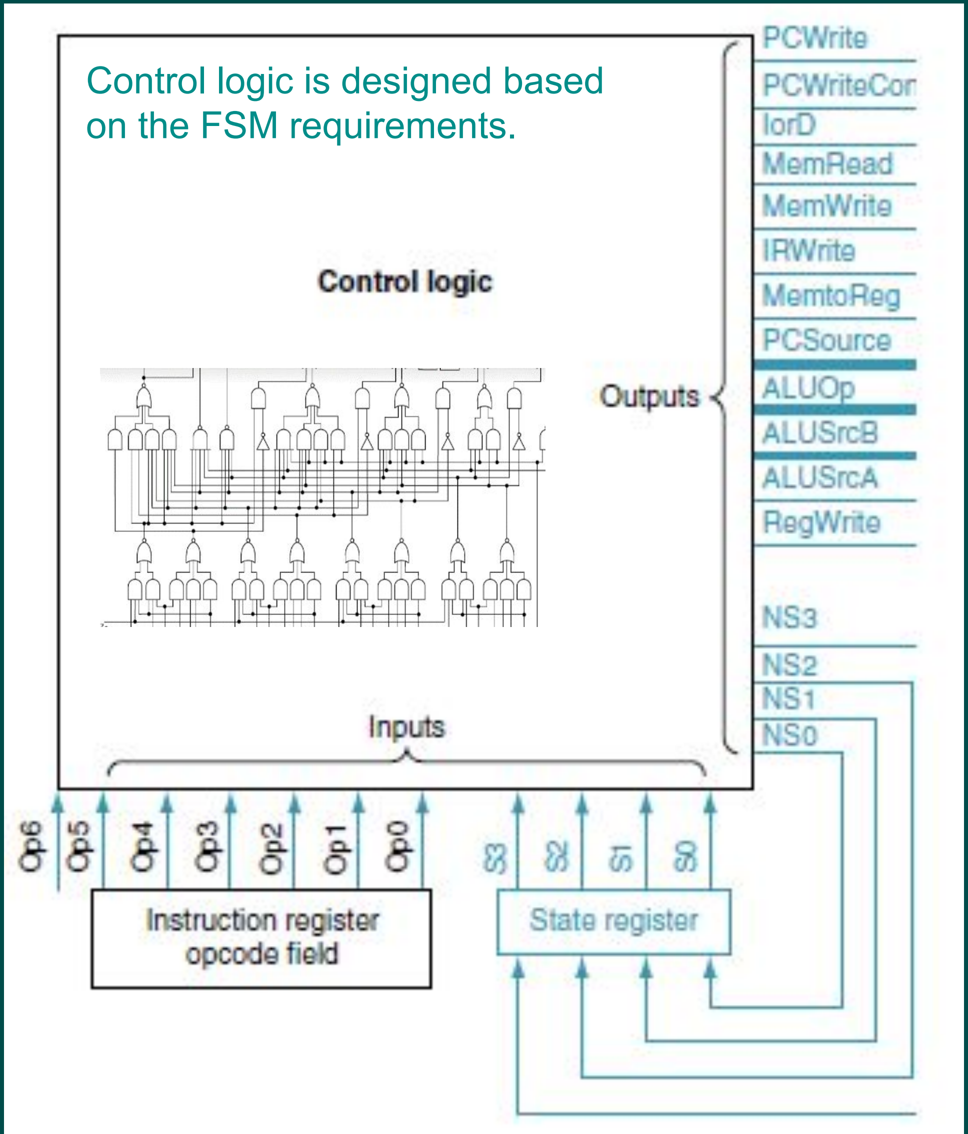
Control Unit

- Control signals are issued throughout the datapath by a control unit
- The control unit is the most complex part of microprocessor design
- The control unit directs the operation of the datapth by providing timing and control signals
- Two basic designs of control units:
  - Hardwired – a combinational circuit where the input logic signals are transformed into a set of output logic signals
  - Microprogrammed – the output logic control signals are stored as binary control variables in a special memory contained within the control unit

Hardwired Control Units

- The predominant design for most contemporary microprocessors
- The control unit is implemented through use of combinational logic circuits
  - A finite number of gates that generate control signals based on instruction opcodes
  - Hardwired control units are generally faster than the microprogrammed designs
- This design uses a fixed architecture
  - Requires changes in the wiring if the instruction set is modified or changed
  - A controller that uses this approach can operate at high speed but has minimal flexibility

Hardware Implementation

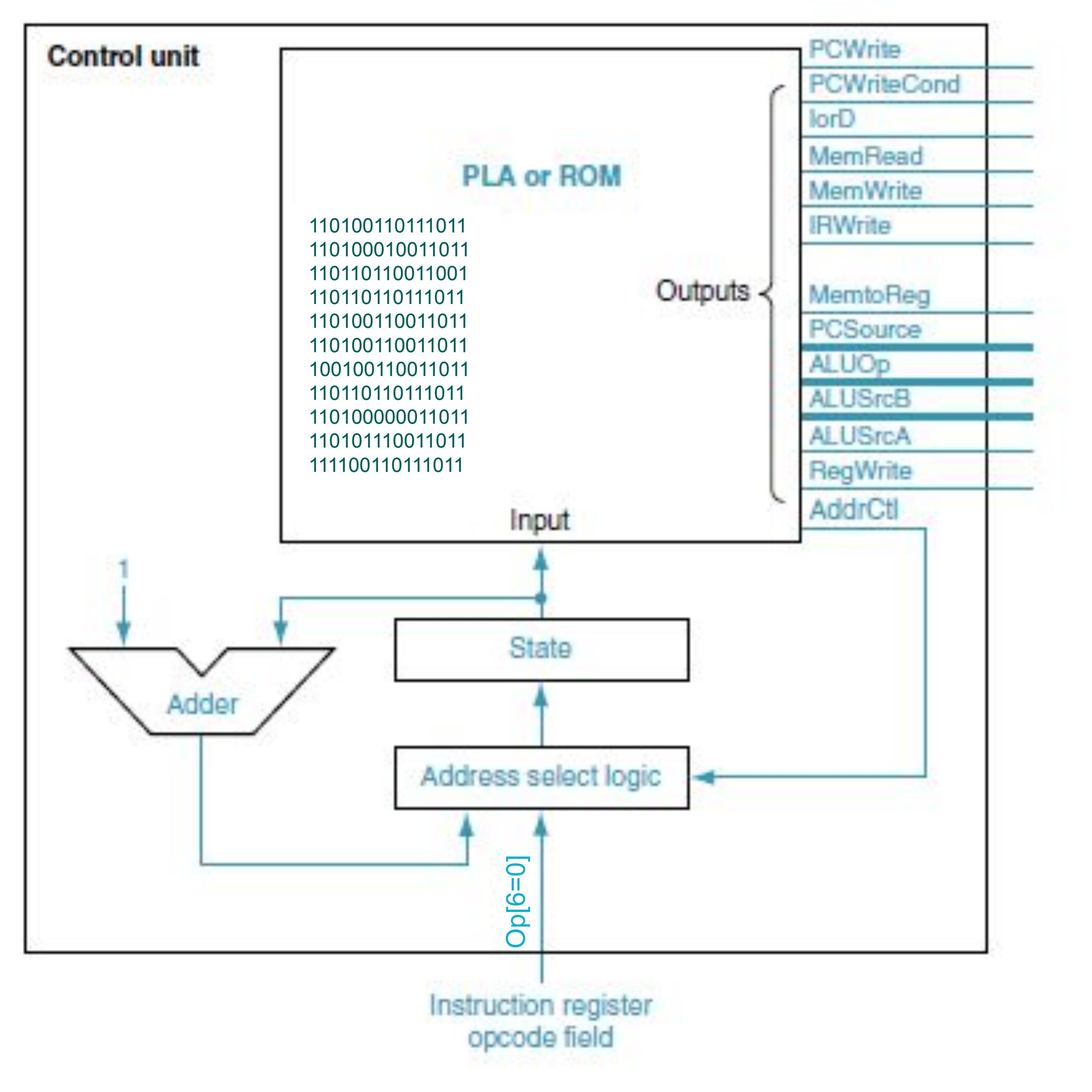


Control logic is designed based on the FSM requirements.

Microprogrammed Control Units

- Microprogrammed control units use ideas from programming to create a method of generating the control signals
- Control signals are encoded into microinstructions that are executed in sequence
  - Each sequence of microinstructions is called a microprogram or microcode
- Each machine instruction has a corresponding microprogram that is stored in a special internal memory called a control store
  - accessing the control store executes the microprogram to generate the control signals to the datapath

Microprogrammed Implementation



```
110100110111011
110100010011011
110110110011001
110110110111011
110100110011011
110100110011011
100100110011011
110110110111011
110100000011011
110101110011011
111100110111011
```

Hybrid Control Unit Designs

- There are control units that use ideas from both hardwired and microprogrammed designs
  - Microprogrammed control units were primarily used in CISC
  - Hardwired control units are primarily used in RISC
- The microprocessors that use hybrid control units are typically those that incorporate both CISC and RISC features
  - ex. Intel Sandy Bridge, Ivy Bridge, Haswell CPUs
  - ex. AMD Zen microarchitecture
    - Micro-operations from decoded instructions are stored in a trace cache
    - When an instruction is fetched, the trace cache is checked and the micro-ops are issued if present

Translating Control Units

- Translates each single machine instruction into a sequence of simpler instructions
- Advantage is that an “out-of-order” computer can be simpler in the bulk of its logic, while handling complex multi-step instructions
- x86 Intel CPUs translate complex CISC x86 instructions to more RISC-like internal micro-operations
- The "front" of the control unit manages the translation of instructions
- The "back" of the CU is an out-of-order processing unit that issues the micro-operations and operands to the execution units and data paths.

Exceptions and Interrupts

- Designing the control unit is the most challenging part of designing the processor.
- Complicating the design is how to implement exceptions and interrupts
  - These are events other than branching that disrupt the normal flow of program execution
  - Exception: any unexpected change in control flow regardless of whether the cause is internal or external to the CPU
  - Interrupt: an exception that comes from outside the processor
  - Note: in some architectures and texts, the terms exception and interrupt are used interchangeably!

Exceptions

- Exceptions can occur in many circumstances
  - generated when an unexpected event occurs that causes the currently executing instruction stream to be interrupted
  - Example: invalid memory address
- When an exception occurs, the address of the instruction that caused the exception is saved in a register called the exception program counter (EPC)
- Once saved, the computer jumps to a predefined address to begin executing another segment of code to handle the exception
  - This code is called the exception handler routine
- Saving the address allows returning to the program after the exception is handled
  - Only if recovery from the error is possible
  - Otherwise, the program is terminated with errors

Exception Handler Routine

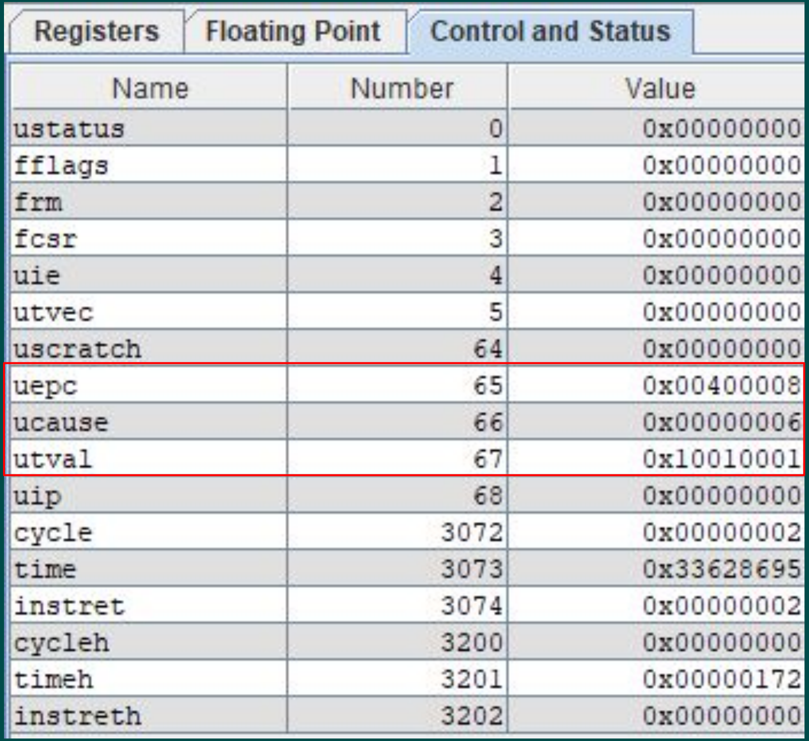
- The program (or code sequence) that executes to handle the exception is called an exception handler
- Exception handlers are found at various levels of the system to attempt to prevent failures due to either program or hardware problems
- Programming languages may have built-in exception handling tools
  - .NET, Java, C++, Python, Ruby, Ada
  - Programmer incorporates or builds exception routines or methods using these tools

Exception Processing

- When an exception occurs
  - Status of processor switches from user to kernel (system) mode
    - Status register value modified to reflect this
  - EPC register is loaded with the address of the instruction that was executing when the exception occurred
    - Establishes a pointer to the instruction if re-execution is needed
    - Otherwise, the EPC value is incremented to bypass re-executing the excepting instruction
  - Cause register is loaded with the numeric code of the type of exception

RARS Exception

- Control and Status tab next to registers lists subset of csr registers
  - csr = Control and status registers
  - RISC-V specification defines 4096 total csr registers
    - some are for future use and allow for expansion



Name	Number	Value
ustatus	0	0x00000000
fflags	1	0x00000000
frm	2	0x00000000
fcsr	3	0x00000000
uie	4	0x00000000
utvec	5	0x00000000
uscratch	64	0x00000000
uepc	65	0x00400008
ucause	66	0x00000006
utval	67	0x10010001
uip	68	0x00000000
cycle	3072	0x00000002
time	3073	0x33628695
instret	3074	0x00000002
cycleh	3200	0x00000000
timeh	3201	0x00000172
instreth	3202	0x00000000

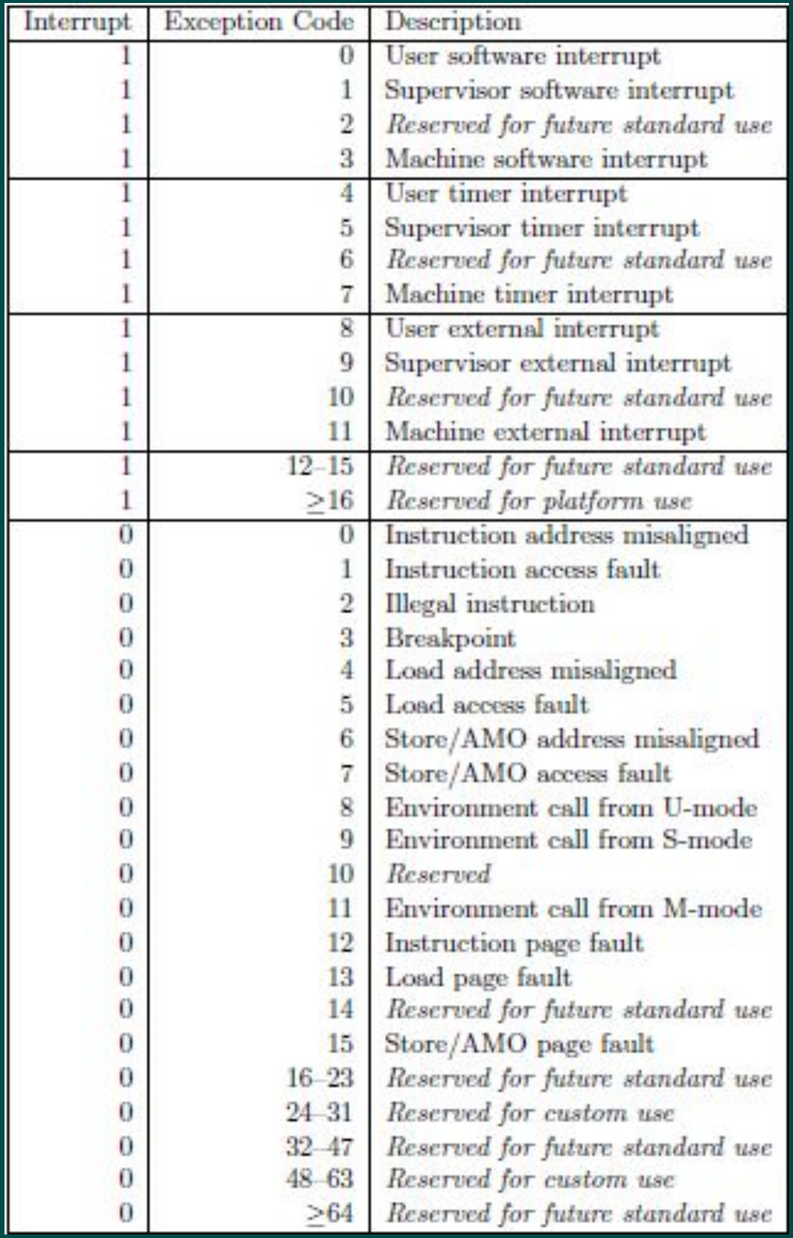
uepc = User exception program counter

ucause = User cause

utval = User trap value

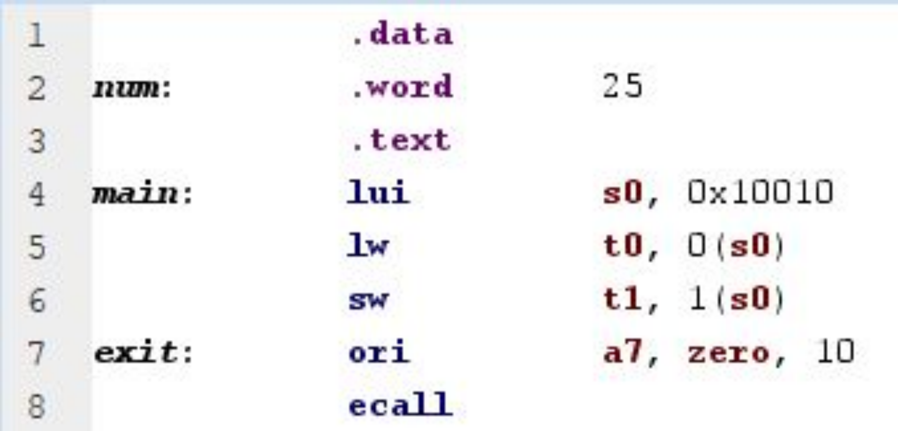
RISC-V Exception Cause Values

- Table of user cause codes for the cause register in the csr



Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Example from RARS:

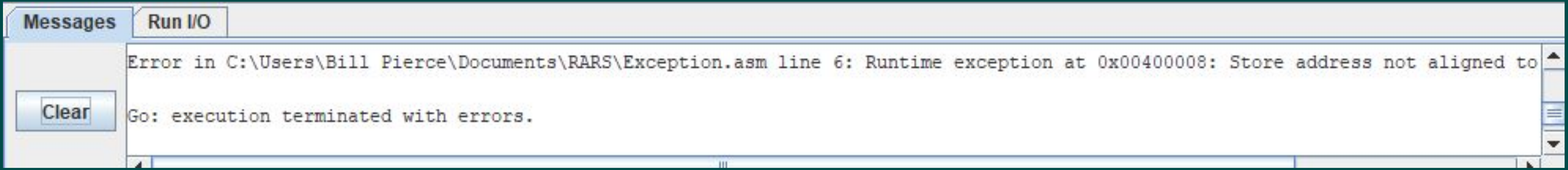


```
1                 .data
2  num:           .word       25
3                 .text
4  main:          lui         s0, 0x10010
5                 lw          t0, 0(s0)
6                 sw          t1, 1(s0)
7  exit:          ori         a7, zero, 10
8                 ecall
```

Store word instruction on line 6 violates alignment restriction - offset must be multiple of 4.

Offset = 1 is an error and generates an exception

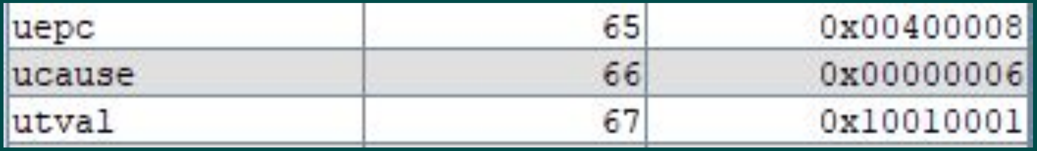
RARS Messages Window



```
Error in C:\Users\Bill Pierce\Documents\RARS\Exception.asm line 6: Runtime exception at 0x00400008: Store address not aligned to

Go: execution terminated with errors.
```

CSR for above program:



uepc	65	0x00400008
ucause	66	0x00000006
utval	67	0x10010001

uepc = 0x00400008: address of instruction generating the exception
ucause = 0x00000006: exception code for store address misaligned
utval = 0x10010001: address in error for memory access

Computer Organization and Design

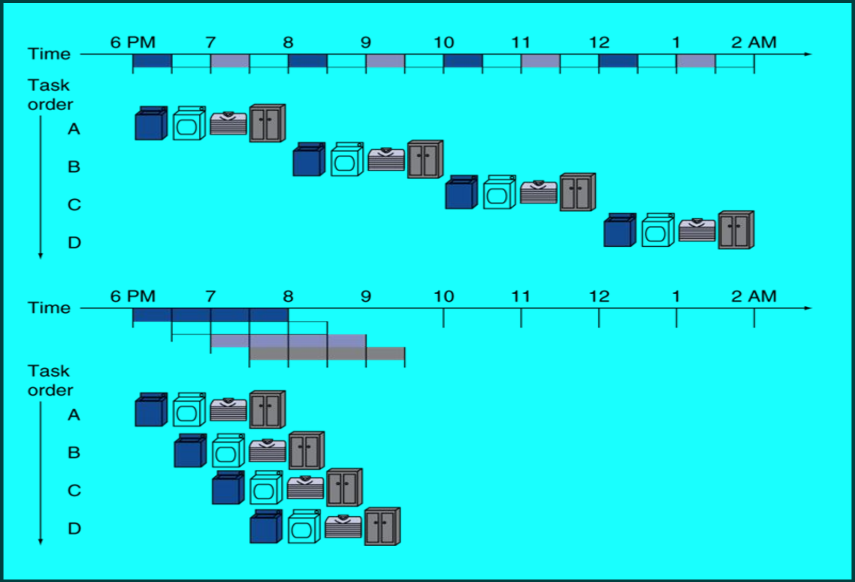
Chapter 4 Part 2

The Processor:

Pipelining

Pipelining Concept

- Pipelining is an implementation technique where multiple instructions are overlapped in execution
- Pipelining is one of the key implementation techniques used in all high-performance CPUs
- Pipelining is based on real life processes
  - Laundry example
  - Assembly lines



Pipelining Principles and Terminology

- Pipelining does not improve the execution time (latency) of any single instruction but it does improve the throughput of the entire program
- The pipeline is defined in stages, each stage implements a step of an instruction’s execution
- The pipeline rate is limited by the slowest stage
- In a pipelined system, multiple tasks are operating simultaneously (parallelism)

Speedup

- The speedup of a pipeline is the ratio between the time for a non-pipelined execution and the pipeline execution
- The potential speedup is equal to the number of stages
- Unbalanced lengths of pipeline stages reduces speedup (not all stages take the same amount of time)
- The time to “fill” the pipeline and the time to “drain” it reduces speedup

RISC-V Pipeline

- Recall that the RISC-V instruction cycle defines five steps:
  - Instruction fetch cycle (IF)
  - Instruction decode & register fetch cycle (ID)
  - Execution, memory address computation, or branch completion (EX)
  - Memory access or R-type instruction completion cycle (MEM)
  - Write-back cycle (WB)
- The RISC-V pipeline is based around the steps in the instruction cycle
  - Pipeline will have 5 stages

Pipeline Diagrams

- Pipelines are visually represented as diagrams showing instructions and their progression through the pipeline

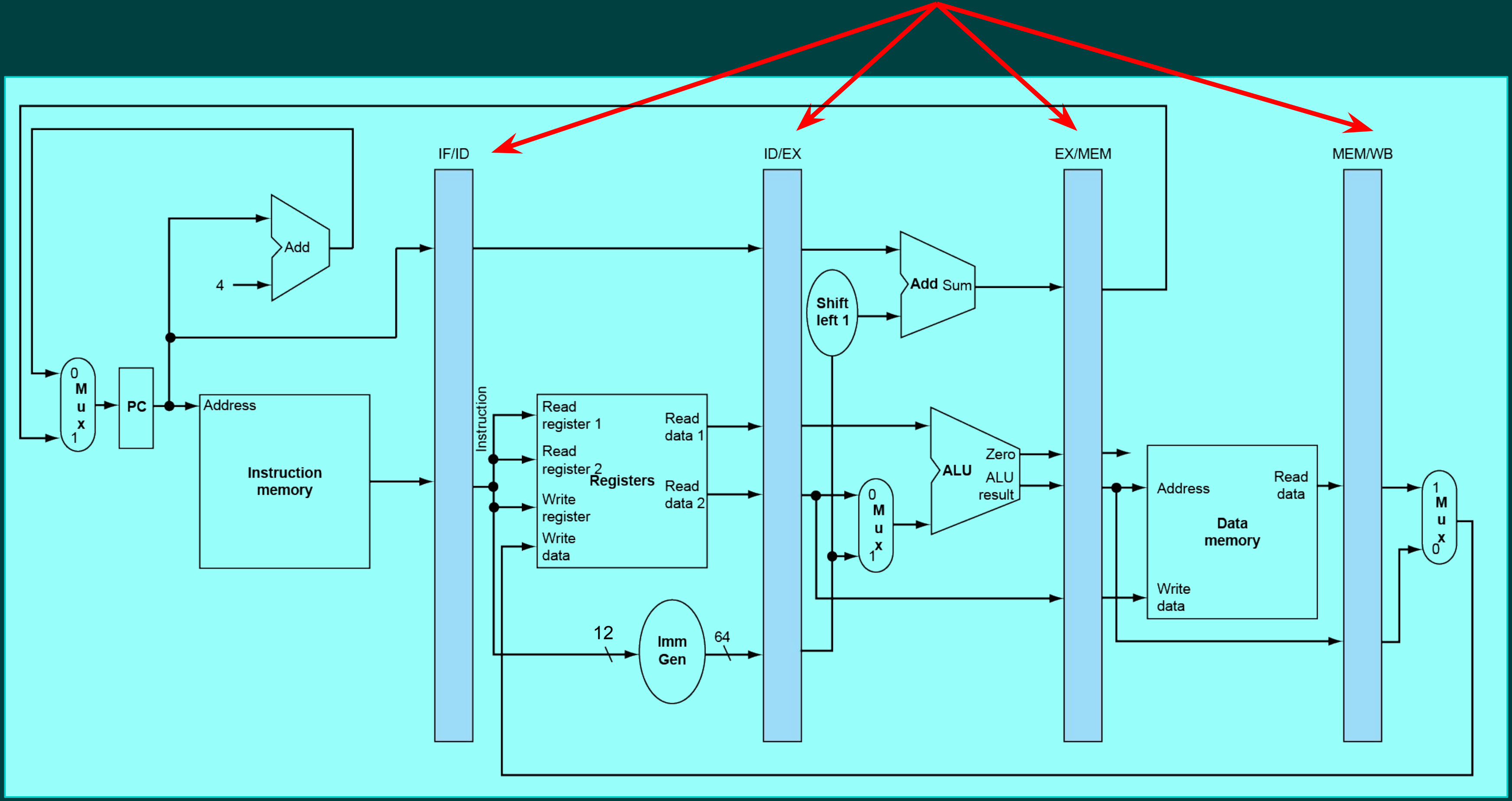
< - - - - - - - - - - - - - Clock Cycles - >

	1	2	3	4	5	6	7	8
Inst. I	IF	ID	EX	MEM	WB			
Inst. I+1		IF	ID	EX	MEM	WB		
Inst. I+2			IF	ID	EX	MEM	WB	
Inst. I+3				IF	ID	EX	MEM	WB

RISC-V Pipelined Datapath

Each stage of the datapath is separated by a set of pipeline registers that work to stage data values between pipeline stages

Pipeline Registers



Timing The Pipeline

- Not all stages take the same amount of time
- Total time for an instruction is the sum of times for each stage that the instruction uses

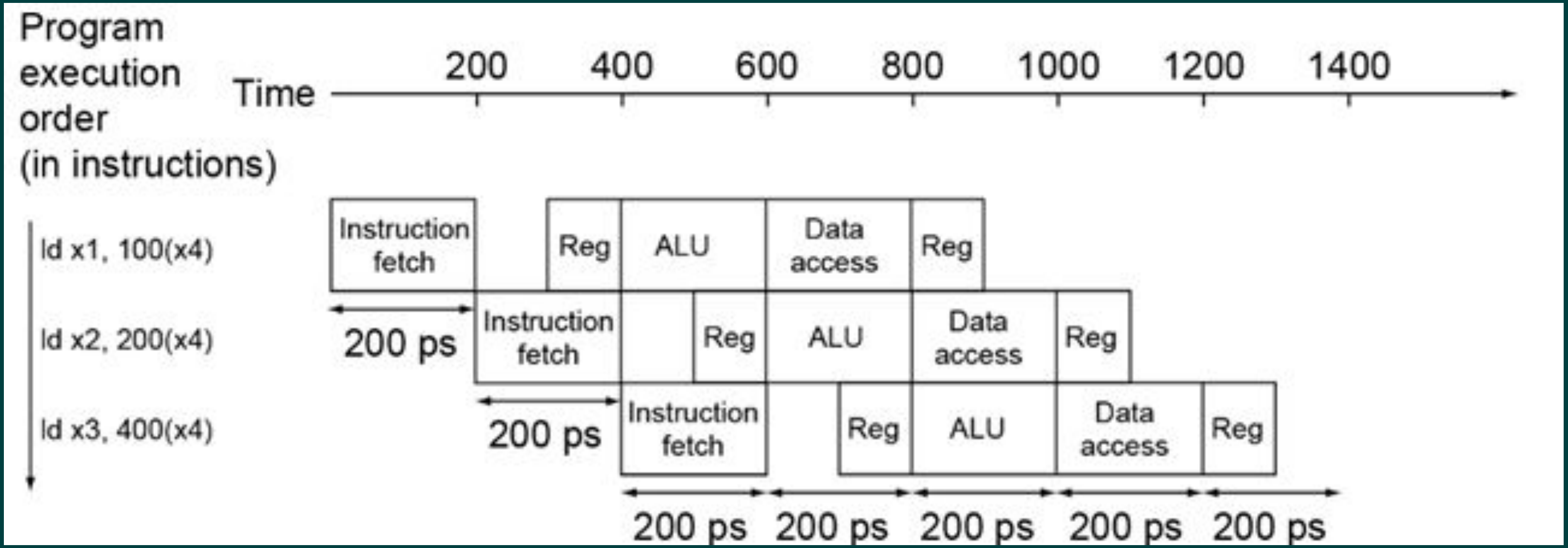
Table showing hypothetical stage latencies

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
Load word	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
Branch	200 ps	100 ps	200 ps			500 ps

- However, since the clock cycle period is fixed, relative to the time required for the longest stage, each instruction’s total execution time is the total time through all the pipeline stages
  - Mix of instructions sequencing through the pipeline varies

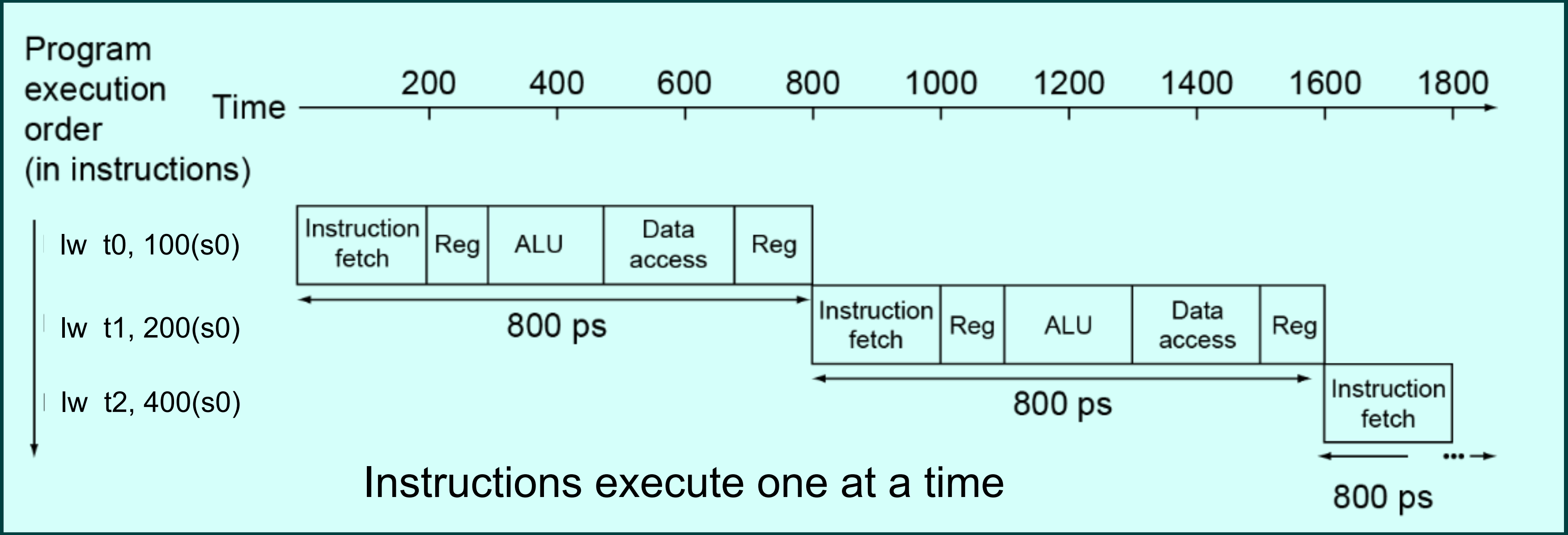
Instructions in the Pipeline

- The following diagram shows three load word instructions executed in the RISC-V pipeline using sample unit times as stated in the previous table

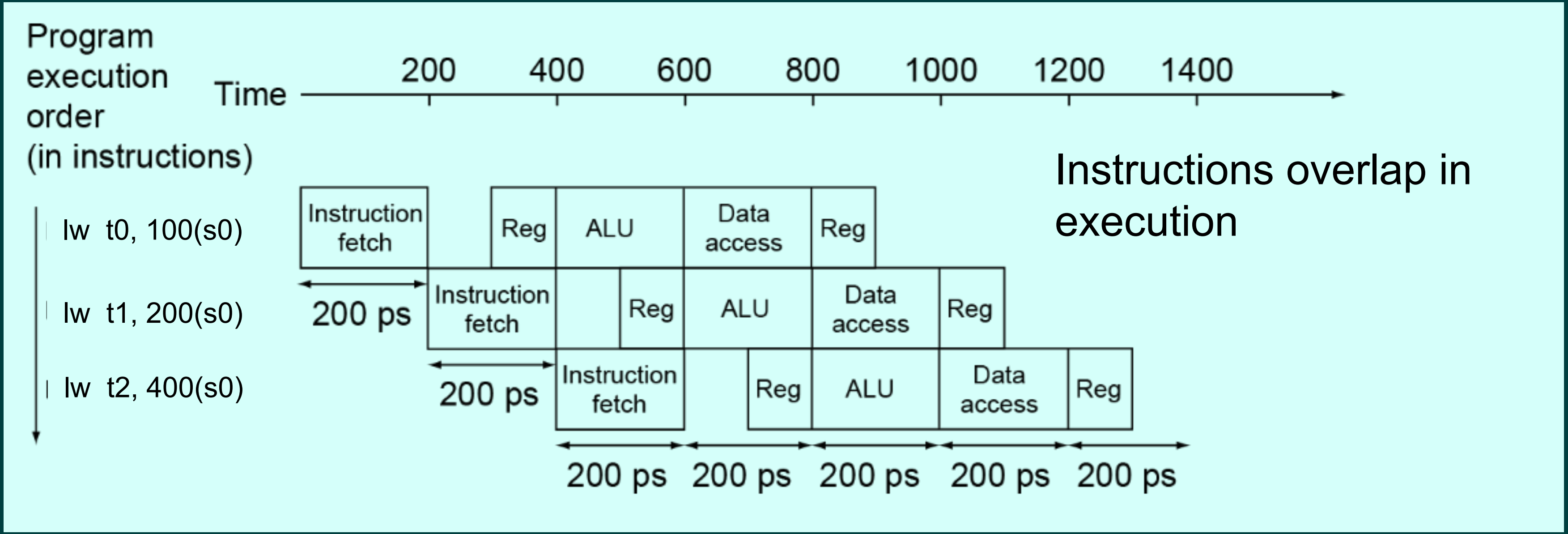


Register reads occur during the second half of the clock cycle
Register writes occur during the first half of the clock cycle

Single Cycle Datapath vs. Pipeline



Instructions execute one at a time



Instructions overlap in execution

Performance Comparison
Single-cycle vs Pipeline

- Single-cycle time = time for longest instruction
- Each load word instruction in a non-pipelined datapath requires 800 ps
- Strict sequential execution of three load word instructions would require 2400 ps
- In a pipelined datapath, each stage time = time of longest stage
  - All stages take 200 ps; 5 stages = 1000 ps
  - So, each load word instruction requires 1000 ps
- However, instructions overlap by starting a new instruction every 200 ps

Performance Comparison (continued)

- Total execution time for three load word instructions = 1400 ps
- We can see that the execution time for these three instructions is shorter but this does not provide a true measure of pipelined performance
- General pipeline speedup formula:

$$\boxed{Time\ between\ \ instructions_{Pipelined} = \frac{Time\ between\ instructions_{Nonpipelined}}{Number\ of\ pipeline\ stages}}$$

Performance Comparison (continued)

- Non-pipelined: the time between the 1st and 4th instruction is

  3 x 800 ps = 2400 ps

- Pipelined: the time between the 1st and 4th instruction is

  3 x 200 ps = 600 ps

- Pipeline speedup:

  2400/600 = 4

  even though there are 5 stages

- Speedup of 4 vs. 5 due to filling and draining the pipeline
  - Decrease in the theoretical (ideal) speedup

Instruction Sets for Pipelining

- Most RISC instruction sets (i.e. RISC-V) were designed for pipelining
  - All instructions are same length (32 bits)
    - Easier to fetch and decode in first two stages
  - Regular instruction formats
    - Can decode and read registers in one step
  - Memory operands only appear in loads and stores
    - Calculation of memory address occurs in execute stage followed by memory access in next stage
  - Operands must be aligned in memory
    - Memory transfers occur in a single pipeline stage

Pipeline Hazards

- Hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle
- Hazards reduce the performance of pipelines
- Three types of hazards
  - Structural
  - Control
  - Data

Structural Hazards

- In a structural hazard, the hardware cannot support the combination of instructions in the same clock cycle
- Usually caused when two instructions contend for use of the same resource within a clock cycle
- Conflicts can be reduced or eliminated with careful design of the hardware and instruction set
  - Compiler design to identify conflicts during compilation
  - Separate instruction and data memory
  - Build in more hardware (multiple execution units)

Control Hazards

- Control hazards are caused by branches and other instructions that change the value of the program counter (PC)
- Changing the PC later in the pipeline affects all the instructions that have already started in the pipeline
- When a change in sequential execution occurs through a branch, instructions already in the pipeline following the branch must be flushed and new instructions loaded from the point of the branch
- Branches not taken do not cause a problem

Control Hazard Solutions

- Solution 1: insert a pipeline stall (bubble) after every branch instruction
  - Delay the next instruction after the branch by one or more clock cycles until you know the result of the branch (taken or not taken)
  - Not a good choice since branches occur relatively frequently in code
    - If/else, loop control
  - Lower utilization of pipeline stages

Control Hazard Solutions

- Solution 2: predict the outcome of the branch (three choices)
  - Assume branches are always are not taken
  - Assume branches are always taken
  - Static prediction – once set, can't change
  - Effectiveness depends on nature of the application
    - How many branches are taken vs. not taken
  - Prediction of branch outcome based on the previous history of one or more branches in the program (dynamic prediction)
    - Must have additional function to record history of branches
    - Possible average of 85% accuracy with 2-bit history

Dynamic Branch Prediction

- Implemented in all commodity microprocessors
  - Intel, AMD, IBM Power, ARM, etc.
- Technique uses a branch prediction buffer
  - Branch History Table (BHT)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken) of branch instructions
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction
- Sophisticated dynamic branch prediction schemes achieve 97% accuracy in predictions

Data Hazards

- Data hazards occur when the current instruction needs the result of a previous instruction that is still executing in the pipeline
  - Consider the following instruction sequence

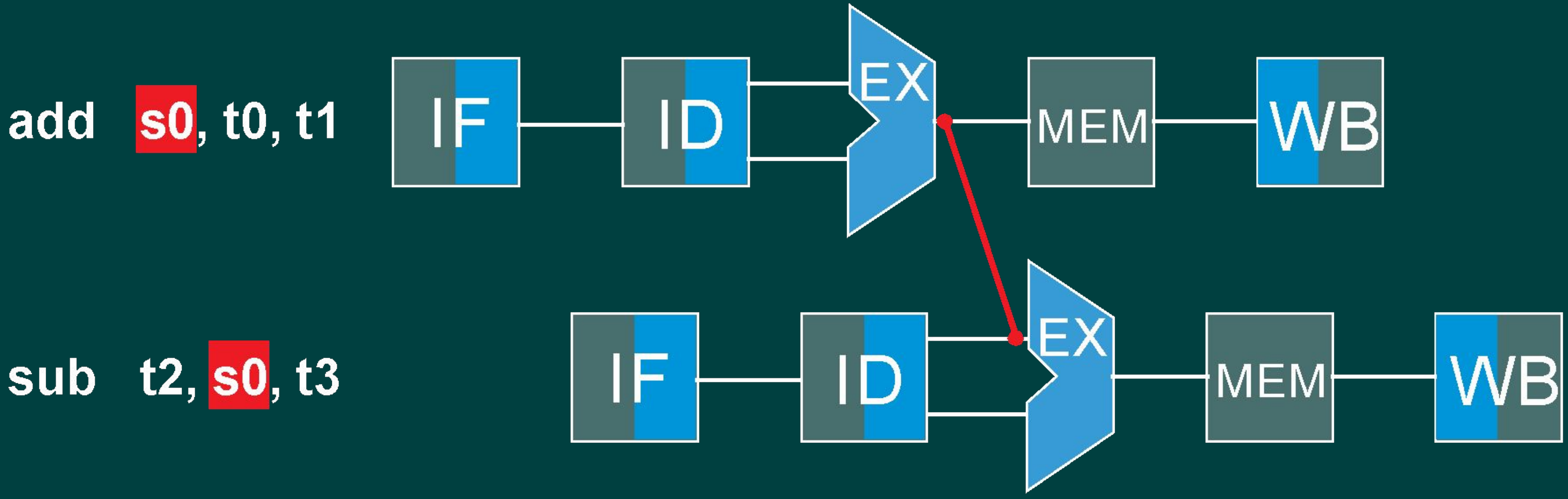
    ```
    add  s0, t0, t1
    sub  t2, s0, t3
    ```

  - The add instruction doesn't write its result to register s0 until the last stage (stage 5)
  - The sub instruction would read register s0 in its second stage
  - We would have to stall the pipeline for three cycles so that the sub instruction would read the correct value from s0

Data Hazards (continued)

- This is too common a problem to attempt to resolve in compilers or assemblers
- The most common solution is to implement data forwarding or data bypassing
- Data forwarding provides a means of making the result of the ALU available directly as input to the ALU stage of the following instruction prior to the write-back stage
- There is still the possibility of needing pipeline stalls on some instruction sequences (e.g. an R-format instruction following a load)

Data Forwarding

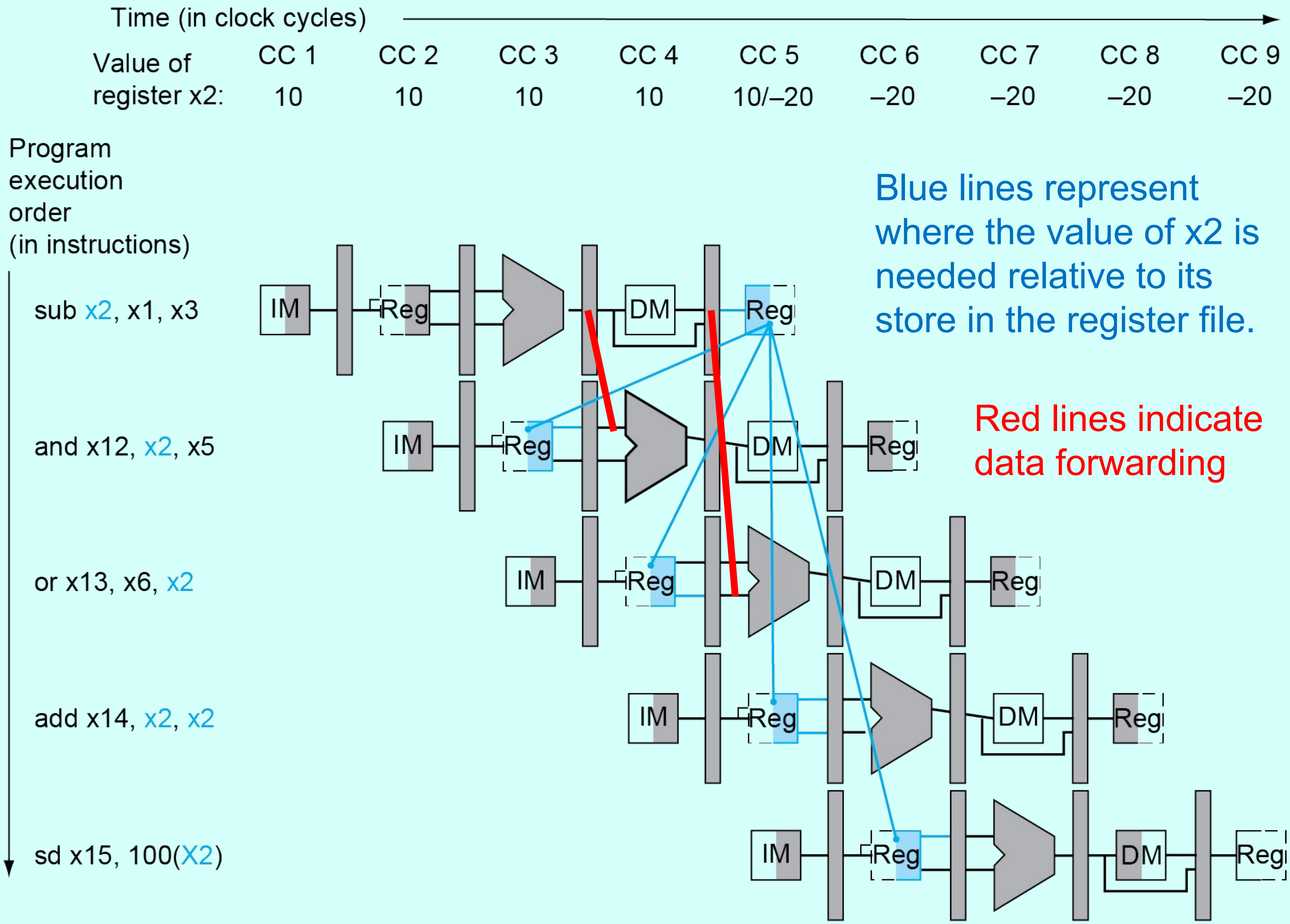


add s0, t0, t1

sub t2, s0, t3

- Special register files are placed on the input of the ALU to temporarily hold values from the ALU output.
- If a forwarded data value is required, it is read from the special register file rather than from the normal register file.

A Pipelined Sequence of Instructions



Time (in clock cycles)

Value of register x2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/–20	–20	–20	–20	–20

Program execution order (in instructions)

sub x2, x1, x3

and x12, x2, x5

or x13, x6, x2

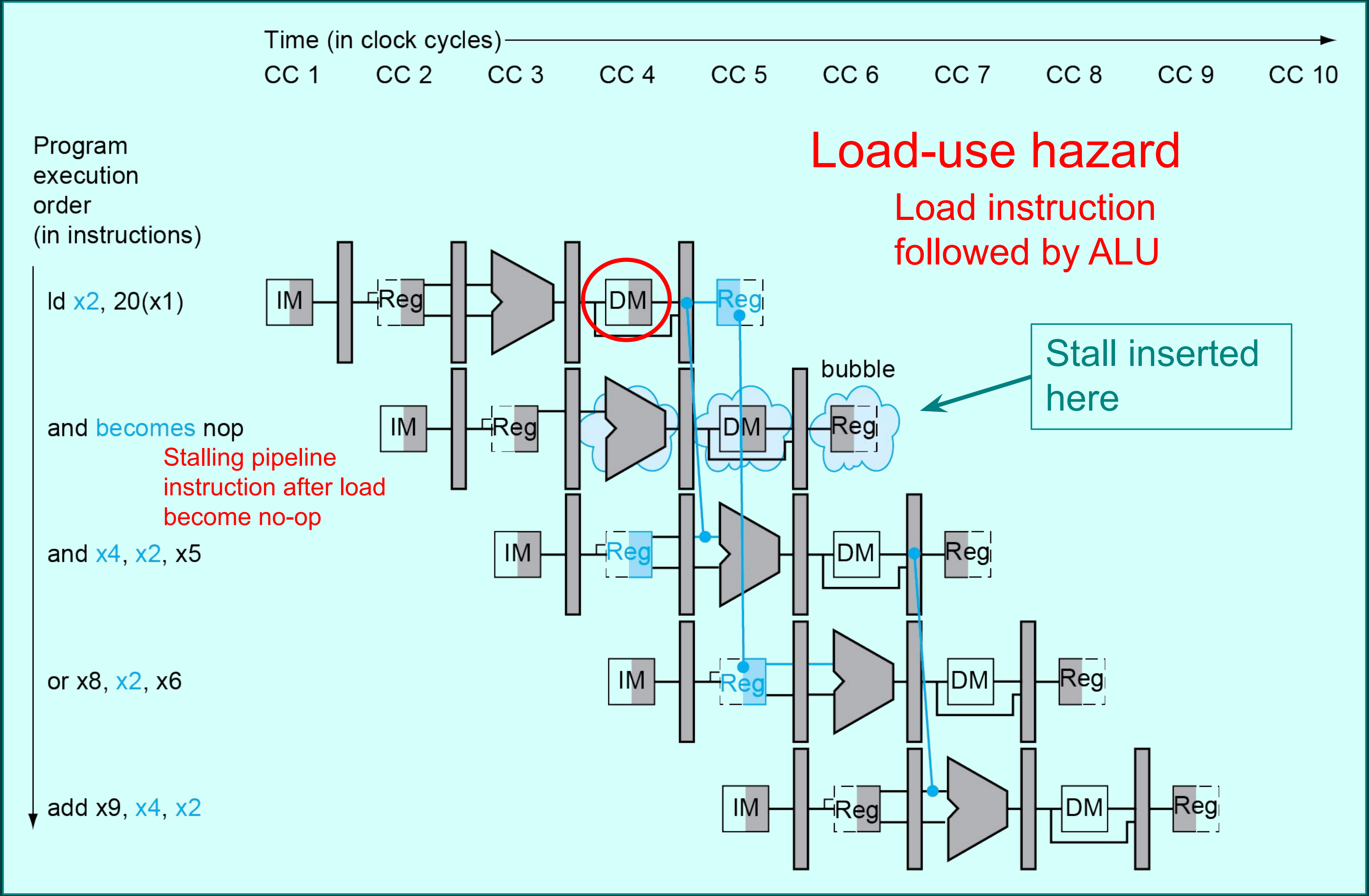
add x14, x2, x2

sd x15, 100(X2)

Blue lines represent where the value of x2 is needed relative to its store in the register file.

Red lines indicate data forwarding

Inserting a Stall (Bubble) in the Pipeline



Load-use hazard

Load instruction followed by ALU

Stall inserted here

Stalling pipeline instruction after load become no-op

Example: Data Dependencies

- Identify all the data dependencies in the following code

```
add   s0, a1, a0
add   a0, s0, a1
swa1, 0(s0)
add   s1, s0, a0
```

1. The second instruction is dependent upon the first (s0).
2. The third instruction is dependent upon the first (s0)
3. The fourth instruction is dependent upon the first (s0) and the second (a0)

Example: Data Hazards

- Identify all the data hazards in the following code (asterisks indicate dependency)

```
 add   s0, a1, a0
*add   a0, s0, a1      → Hazard on s0
*sw    a1, 0(s0)       → Hazard on s0
**add  s1, s0, a0      → Hazard on a0
```

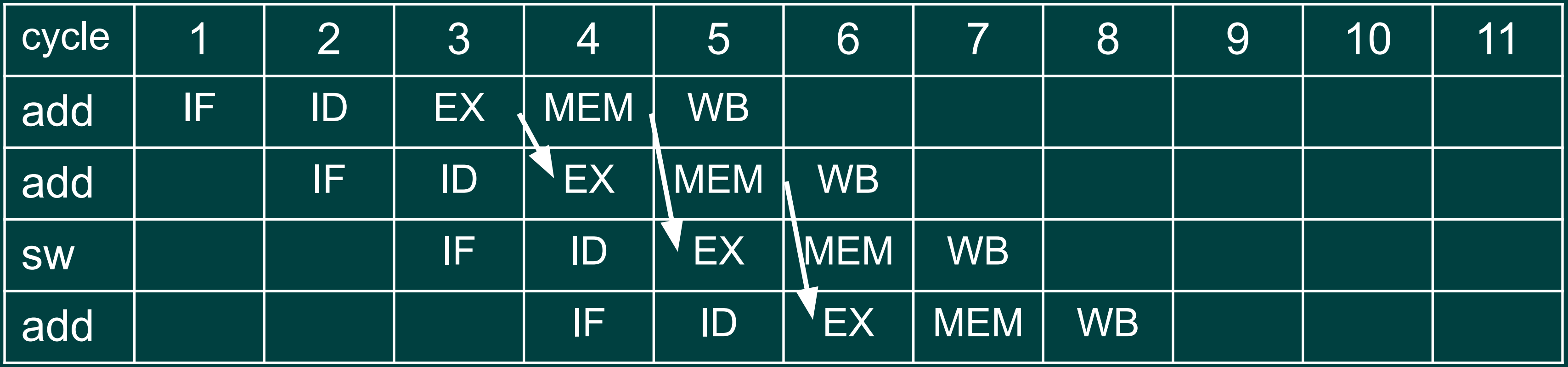
cycle	1	2	3	4	5	6	7	8	9	10	11
add	IF	ID	EX	MEM	WB						
add		IF	ID	stall	stall	EX	MEM	WB			
sw			IF	stall	stall	ID	EX	MEM	WB		
add						IF	ID	stall	EX	MEM	WB

No data forwarding incurs three stall cycles

Forwarding Example: Hazards Eliminated

- Data hazards previous identified in non-data forwarding pipeline are eliminated

```
add   s0, a1, a0
*add  a0, s0, a1
*swa1, 0(s0)
*add  s1, s0, a0
```



cycle	1	2	3	4	5	6	7	8	9	10	11
add	IF	ID	EX	MEM	WB						
add		IF	ID	EX	MEM	WB					
sw			IF	ID	EX	MEM	WB				
add				IF	ID	EX	MEM	WB			

Arrows show data forwarding paths; stalls eliminated

Advanced Pipelining

- Goal is to advance performance by implementing techniques that obtain higher levels of parallelism
- ILP (Instruction Level Parallelism) looks at the instruction sequence to identify parallelism
  - Instructions that are not dependent on each other can be executed in parallel providing you have the hardware to support simultaneous execution
  - Multiple issue is a scheme where more than one instruction begins execution in a single cycle
    - Static multiple issue – compiler generated decisions about which instructions to issue together
    - Dynamic multiple issue – decisions are made during execution by the CPU
    - Only possible in superscalar pipelined processors

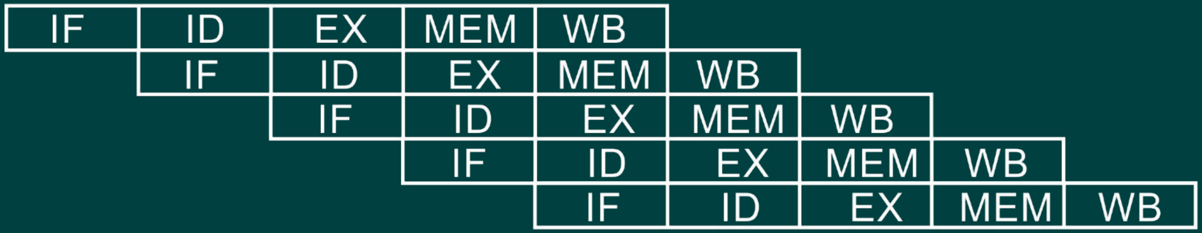
Speculation

- Advanced concept that allows the compiler or CPU to “guess” about the properties of an instruction so as to allow dependent instructions to begin processing.
  - If the guess was correct, there is a gain in performance since all the dependent instructions were executed correctly.
  - If the guess was wrong, there is a performance hit because additional work is required to “undo” the effect of executing the speculated instructions.
- To get the best improvement in performance, a lot of effort goes into the “guessing” of the speculated instructions.
  - Results in a more complex processor
  - Speculation can be done in the compiler or hardware

Advanced Architectures

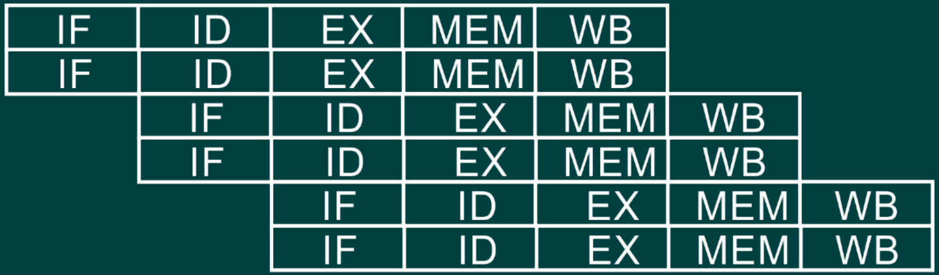
- Superscalar processors
  - dynamic multiple issue processor
  - Instructions are generally issued in-order but can be executed out-of-order
  - Hardware guarantees correctness
  - Most modern processors are superscalar in design

- VLIW (Very Long Instruction Word)
  - Static multiple issue processor
  - Compiler assembles compatible instructions into an instruction word
  - Multiple hardware execution units operate in parallel to execute each instruction word
  - VLIW CPUs implement RISC-like execution units
  - Found in DSPs and embedded media processors



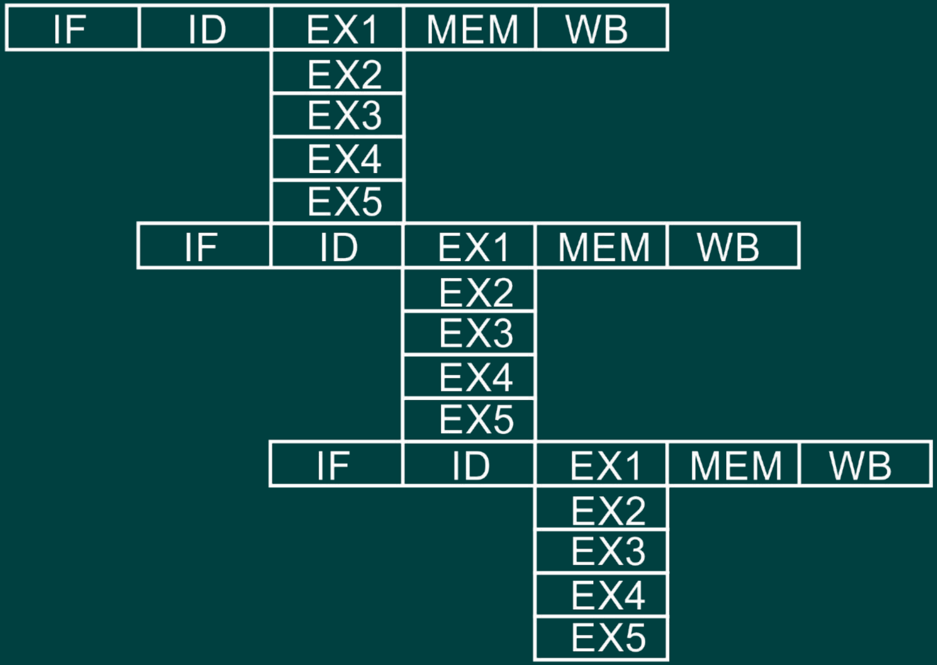
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Scalar pipeline



IF	ID	EX	MEM	WB		
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
		IF	ID	EX	MEM	WB

Superscalar pipeline



IF	ID	EX1	MEM	WB		
		EX2				
		EX3				
		EX4				
		EX5				
	IF	ID	EX1	MEM	WB	
			EX2			
			EX3			
			EX4			
			EX5			
		IF	ID	EX1	MEM	WB
				EX2		
				EX3		
				EX4		
				EX5		

VLIW pipeline

Pipeline architecture comparisons

Terminology / Vocabulary

- Datapath
- Control
- Instruction cycle
- Logic circuit types
  - Combinational
  - Sequential (state)
- Clocking methodology
- Datapath functional elements
- Control signals
- Control units
  - Hardwired
  - Microprogrammed
- Exceptions vs. interrupts
- Pipelining
- Pipeline latency
- Speedup
- Pipeline hazards
  - Structural
  - Control
  - Data
- Hazard solutions
  - Stalling
  - Branch prediction
  - Data forwarding
- ILP (Instruction Level Parallelism
- Speculation
- Superscalar
- VLIW