Chapter 5:
The Processor:

Datapath and Control

Overview

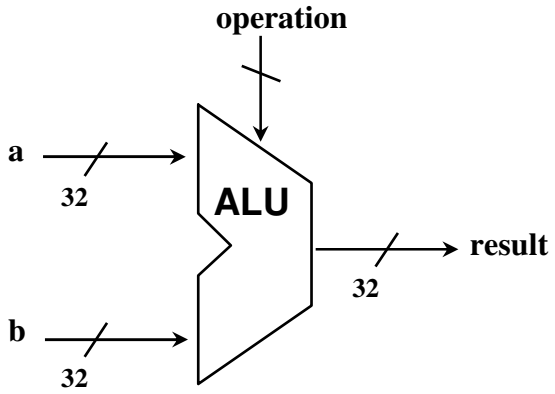
- Logic Design Conventions
- Building a Datapath and Control Unit
  - Different Implementations of MIPS instruction set
- A simple implementation of a processor
- Multicycle Implementation
- Exceptions

Review

Logic Design Conventions

- Almost ready to move into chapter 5 and start building a processor

- First, let's review Boolean Logic and build the ALU
  - To support 1-add for MIPS instruction set and use 32 of them



Review:
Boolean Logic (summary)

- Boolean operations
  - a AND b
    - True only when a is true and b is true
  - a OR b
    - True when either a is true or b is true, or both are true
  - NOT a
    - True when a is false, and vice versa

Review: Boolean Logic (continued)

- Boolean expressions
  - Constructed by combining together Boolean operations
    - Example: (a AND b) OR ((NOT b) AND (NOT a))
- Truth tables capture the output/value of a Boolean expression
  - A column for each input plus the output
  - A row for each combination of input values

Boolean Logic (continued)

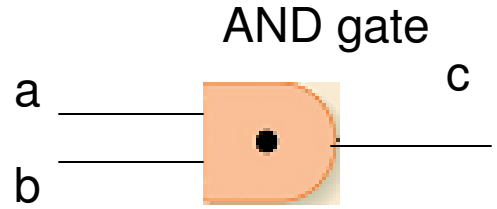
- Example:

  (a AND b) OR ((NOT b) and (NOT a))

INPUT						*OUTPUT*	
a	*b*	$\bar{a}$	$\bar{b}$	*a.b*	$\bar{b}.\bar{a}$	$(a.b) + (\bar{b}.\bar{a})$	*value*
0	0	1	1	0	1	1	*1*
0	1	1	0	0	0	0	*0*
1	0	0	1	0	0	0	*0*
1	1	0	0	1	0	1	*1*

Gates

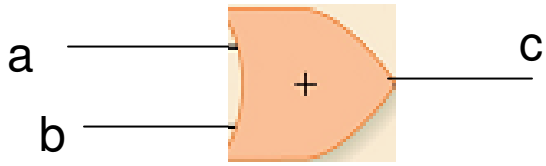
- Gates
  - Hardware devices built from transistors to mimic Boolean logic
- AND gate



  - Two input lines, one output line
  - Outputs a 1 when both inputs are 1

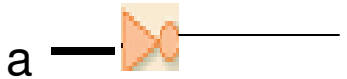
Gates (continued)

- OR gate
  - Two input lines, one output line
  - Outputs a 1 when either input is 1



OR gate

- NOT gate
  - One input line, one output line
  - Outputs a 1 when input is 0 and vice versa



NOT gate

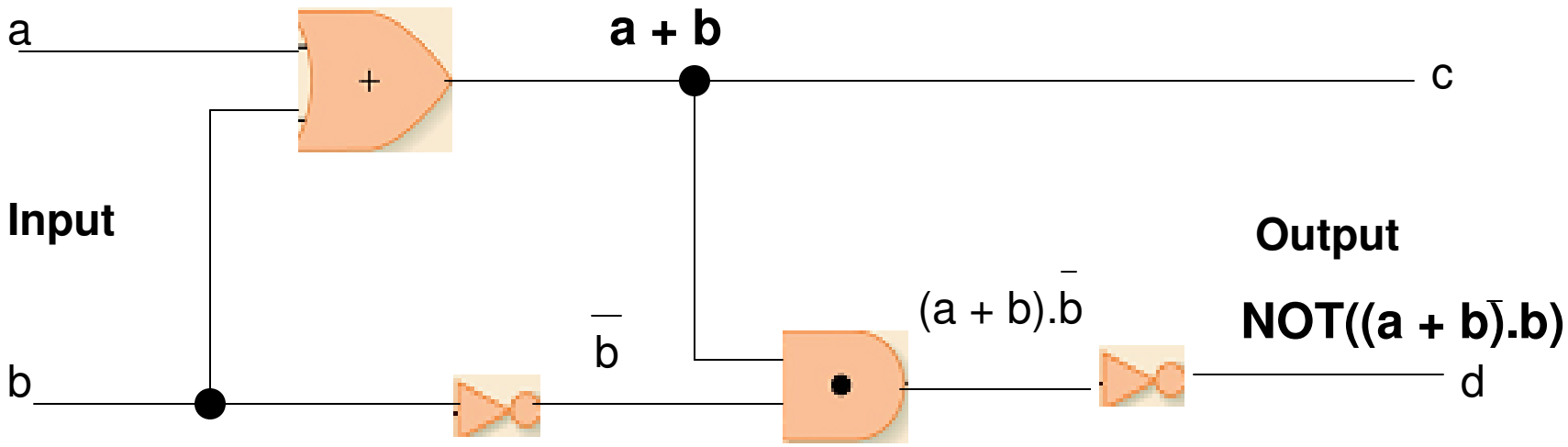
Gates (continued)

- Abstraction in hardware design
  - Map hardware devices to Boolean logic
  - Design more complex devices in terms of logic, not electronics

Building Computer Circuits

- A circuit is a collection of logic gates:
  - Transforms a set of binary inputs into a set of binary outputs
  - Values of the outputs depend only on the current values of the inputs
- Combinational circuits have no cycles in them (no outputs feed back into their own inputs)

Circuit Diagram (Summary)



•Every output in a circuit diagram can be represented as a Boolean Expression

A Circuit Construction Algorithm
ALU

- Sum-of-products algorithm is one way to design circuits:
  - Truth table → Boolean expression → gate layout

A Circuit Construction Algorithm (continued)

- Sum-of-products algorithm
  - Truth table captures every input/output possible for circuit
  - Repeat process for each output line
    - Build a Boolean expression using AND and NOT for each 1 of the output line
    - Combine together all the expressions with ORs
    - Build circuit from whole Boolean expression

Construction of Addition Circuit

1-ADD Algorithm

13 + 14

	011000 ← Carry Bit	c_i
13 →	001101	a_i
+		
14 →	001110	b_i
27 →	**011011**	s_i

An Addition Circuit

add $s1, $s2, $s3

- Addition circuit
  - Adds two unsigned binary integers, setting output bits and an overflow
  - Built from 1-bit adders (1-ADD)
  - Starting with rightmost bits, each pair produces
    - A value for that order
    - A carry bit for next place to the left

An Addition Circuit (continued)

- 1-ADD truth table
  - Input
    - One bit from each input integer: a_i, b_i
    - One carry bit (always zero for rightmost bit): c_i
  - Output
    - One bit for output place value: s_i
    - One "carry" bit: c_{i+1}

1-ADD Circuit

Sub-expression construction (**AND & NOT Gates**)

Inputs			outputs	
a_i	b_i	c_i	S_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$\bar{a} . \bar{b} . c$

$\bar{a} . b . \bar{c}$

$a . \bar{b} . \bar{c}$

$a . b . c$

1-ADD Circuit

Sub-expression construction (**AND & NOT Gates**)

OUTPUT: S_i

$$(\bar{a}.\bar{b}.c) + (\bar{a}.b.\bar{c}) + (a.\bar{b}.\bar{c}) + (a.b.c)$$

Construct the circuit diagram for the sub-expression

1-ADD Circuit

Sub-expression construction (**AND & NOT Gates**)

Inputs			outputs	
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	**1**	0
0	1	0	**1**	0
0	1	1	0	1
1	0	0	**1**	0
1	0	1	0	1
1	1	**0**	0	1
1	1	1	**1**	1

$\bar{a}.b.c$

$a.\bar{b}.c$

$a.b.\bar{c}$

$a.b.c$

1-ADD Circuit

Sub-expression construction (**AND & NOT Gates**)

OUTPUT: c_{i+1}

$$(\overline{a}.b.c) + (a.\overline{b}.c) + (a.b.\overline{c}) + (a.b.c)$$

Construct the circuit diagram for the sub-expression

An Addition Circuit (continued)

- Building the full adder
  - Put rightmost bits into 1-ADD, with zero for the input carry
  - Send 1-ADD's output value to output, and put its carry value as input to 1-ADD for next bits to left
  - Repeat process for all bits

Control Circuits

- Do not perform computations
- Choose order of operations or select among data values
- Major types of controls circuits
  - Multiplexors
    - Select one of inputs to send to output
  - Decoders
    - Sends a 1 on one output line, based on what input line indicates

Control Circuits (continued)

- Multiplexor form
  - 2^N regular input lines
  - N selector input lines
  - 1 output line
- Multiplexor purpose
  - Given a code number for some input, selects that input to pass along to its output
  - Used to choose the right input value to send to a circuit (ALU, Registers..)

Practice

Problem: Consider a logic function with three inputs: A, B, and C.

Output D is true if at least one input is true
Output E is true if exactly two inputs are true
Output F is true only if all three inputs are true

Show the truth table for these three functions.

Show the Boolean equations for these three functions.

Show an implementation consisting of AND, OR and NOT gates.

The Processor: *Datapath and Control*

Intro

Recall CPU Performance equation:

CPU time in program = Instruction Count x ***CPI*** x ***Clock cycle time***

Best Performance → Optimal (minimum) CPU time

- The ISA and Compiler determine the Instruction Count
- **The Processor determines:**
  - Clock cycles Per Instruction (*CPI*)
  - Seconds per clock cycle (*Clock cycle time*)

Build a processor that exploits the MIPS ISA

The Processor: *Datapath and Control*

Basic MIPS Implementation

- Let's look at a subset of core MIPS ISA:
  - The memory-reference instructions: *load word (lw), store word (sw)*
  - The arithmetic-logical instructions: `add, sub, and, or, slt`
  - The control flow instructions: *branch equal* (`beq`) and *jump* (`j`)

Create *Datapath and design the Control*
for the three instruction classes

The Processor: *Datapath and Control*

Basic MIPS Implementation flow

- Let's examine the steps involved to implement the instruction classes

1. Generic Steps

[*assume the memory unit stores instructions and supply the instruction given an address*]

- use the program counter (PC) to supply instruction address
- get the instruction from memory
- read registers
- use the instruction to decide exactly what to do

The Processor: *Datapath and Control*

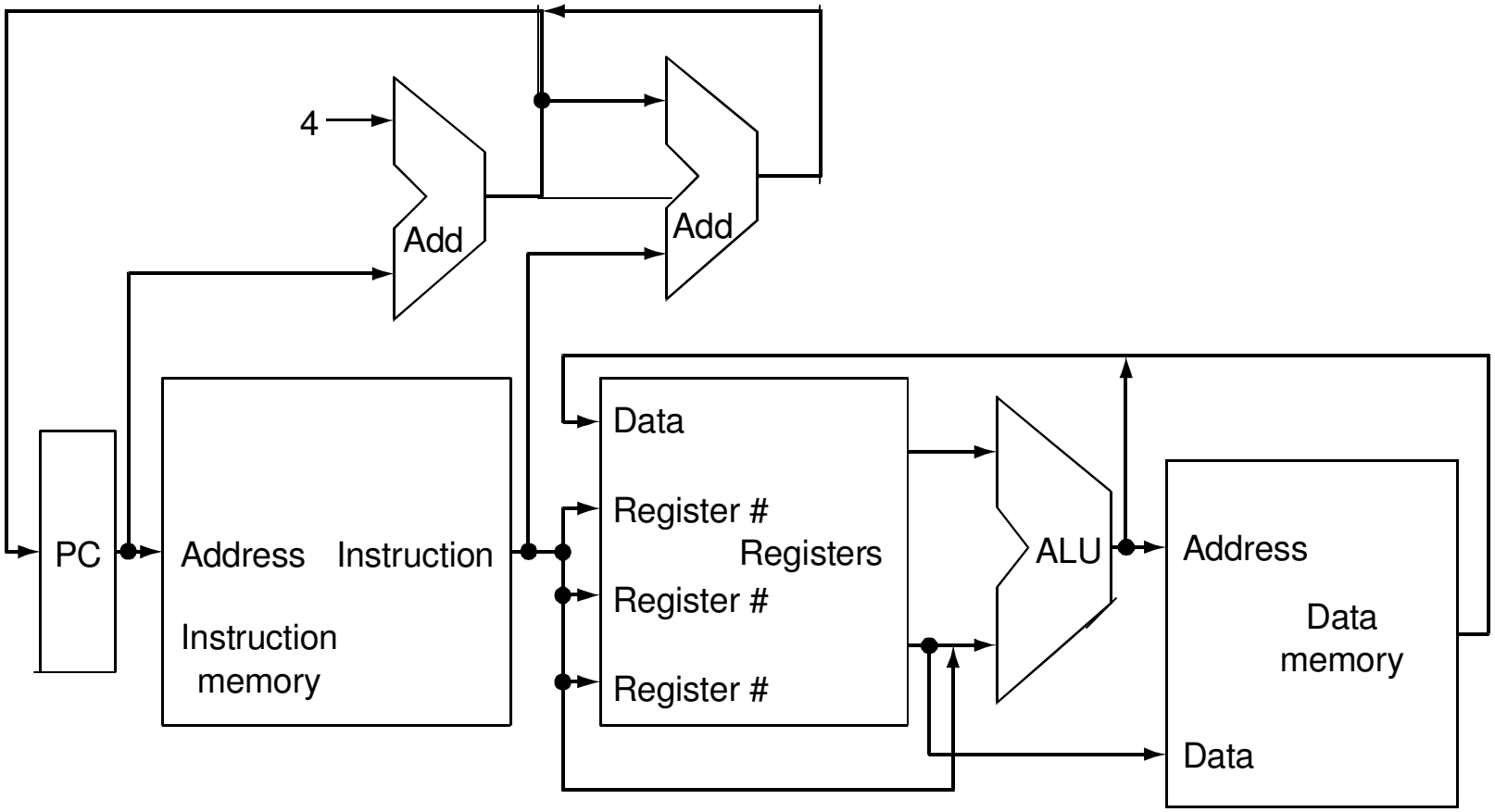
Basic MIPS Implementation flow

2. **Use ALU after reading the register(s)**
   - The memory-reference instructions: *address calculation*
   - The arithmetic-logical instructions: *execute operation*
   - Branch instruction: *Comparison*

3. **Now Complete Implementation based on instruction class**
   - The memory-reference instructions: *access memory?*
   - The arithmetic-logical instructions: *access register?*
   - Branch: *access PC?*

Abstract High-level View

Basic (our subset) MIPS Implementation flow



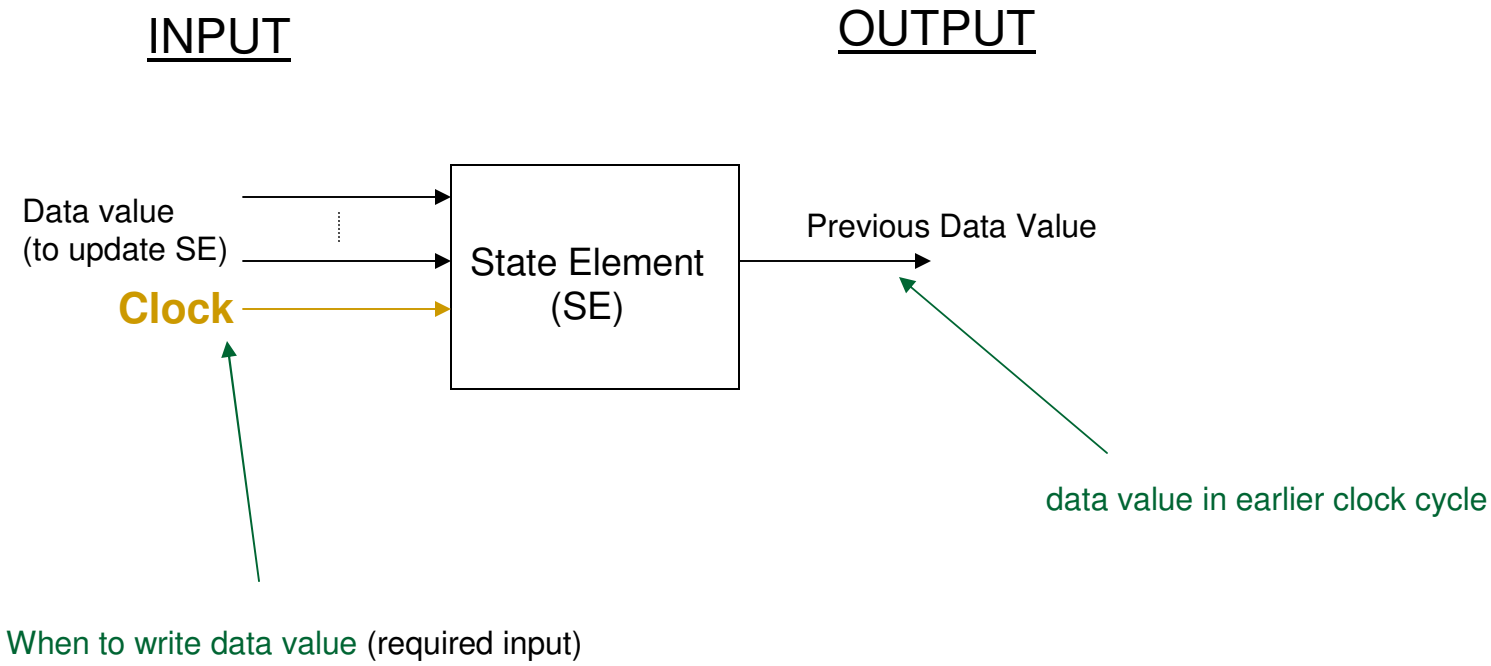
What is wrong with this flow?

Logic Design Conventions

MIPS Implementation

- Two types of logic elements:
  - elements that operate on data values (Combinational)
    - No memory
    - *output state* depends only on *current input* state
    - ALU: and-gate, or-gate and not-gate are combinational
  - elements that contain state (State Elements)
    - Store the state of the bit
    - Output depends on the stored bit
    - State-elements (*flip-flops, latches*)
    - Instruction, Data Memories, Registers

State Element

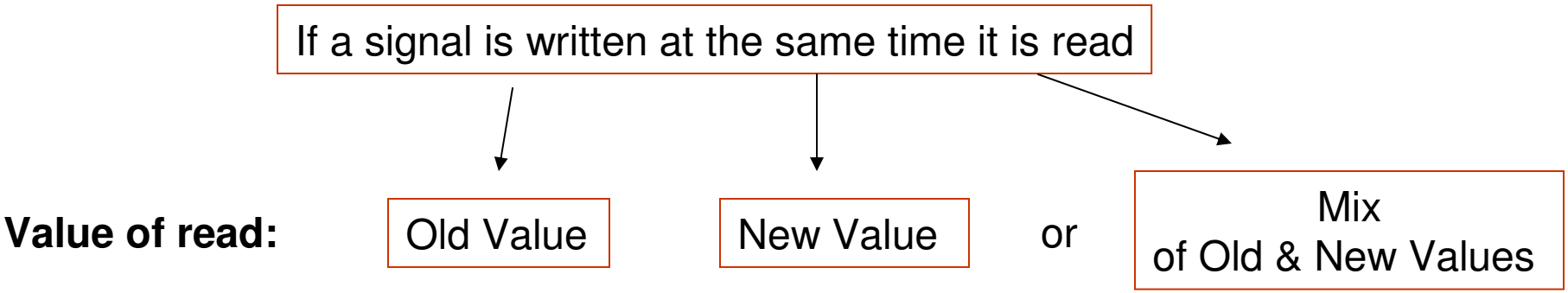


A State Element has a minimum of two inputs and one output

State Elements

Clocking Methodology

- Defines when signals can be read and when they can be written

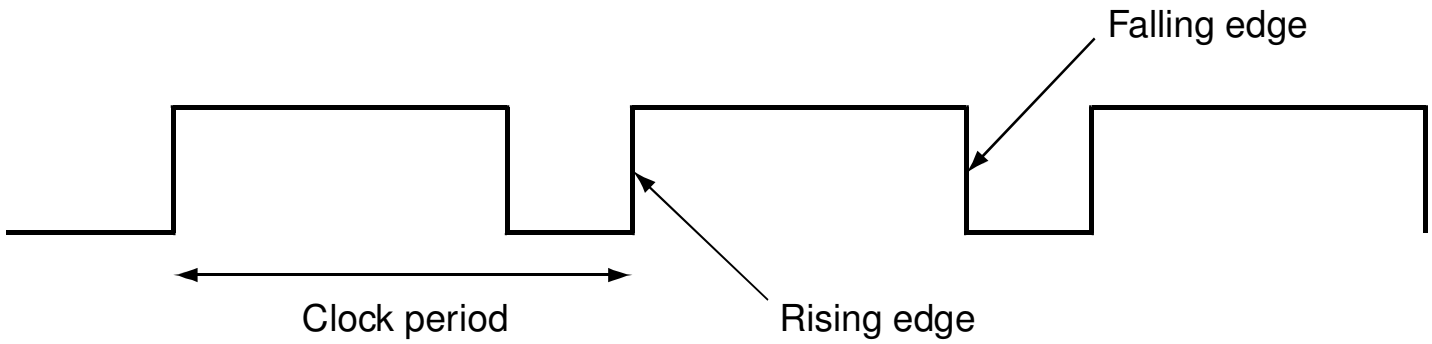


Clocking methodology avoids the scenario

Clocks

Synchronous Logic

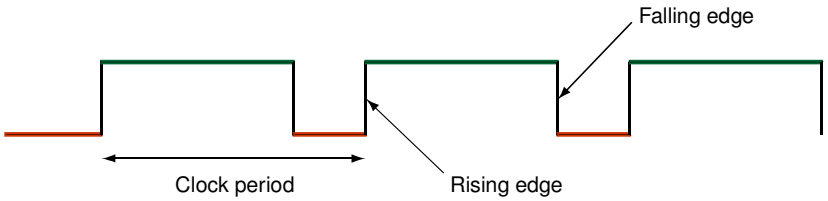
- **When should an element that contains state be updated ?**



State Elements

Synchronous Digital Systems

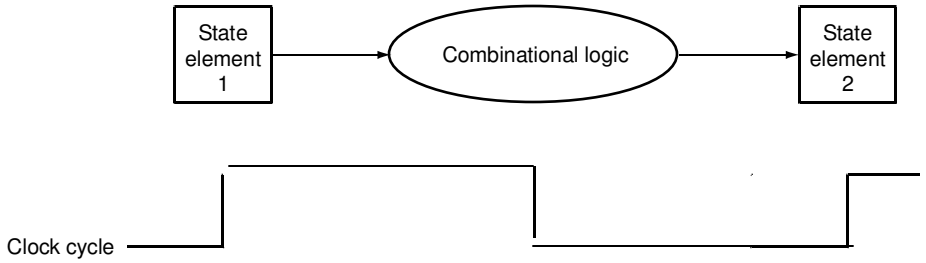
- Clocks
  - Free running signal with a fixed cycle time
  - Decide when an element that contains state should be updated



Clock Period: High and Low

Sequential Logic Design

- Logic Circuit representation:
  - Block of State Elements ^{n}→ Combinational Logic → State Element
  - Signals written into state elements must be valid (Stable)



After Clock edge: SE-1 output changes → Input to Combinational Logic

|

| (Stable signal)

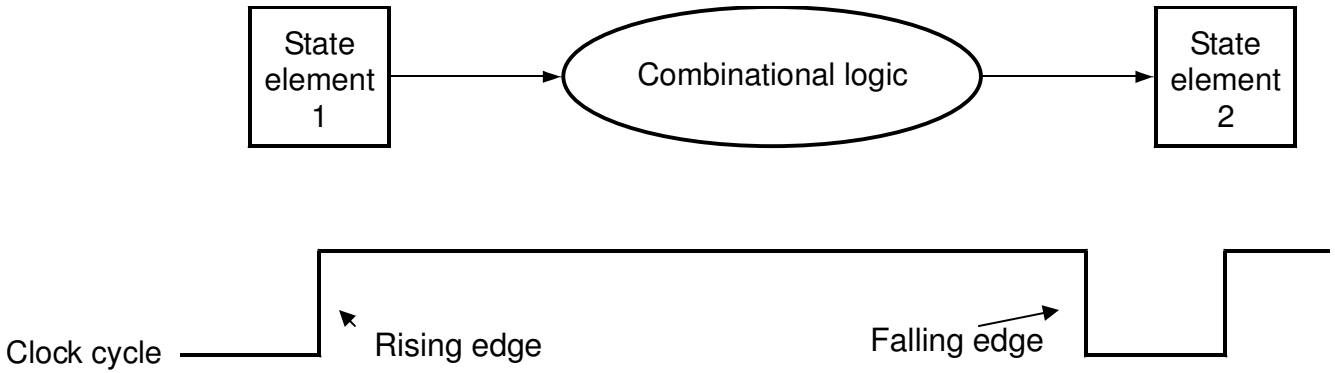
--→ SE-2

Require Long Clock Period Restriction to ensure output of combinational logic is stable (lower bound)

Clocking Methodology

Edge-triggered clocking

- **Edge-triggered Clocking Methodology**
  - **Either Rising edge or Falling Edge of clock is active**
  - **All State Changes occur on the active clock edge**
    - **Sampling of signals is almost instantaneous**
  - **Choice of an active clock edge depends on implementation**



All signals from SE -1 to SE -2 propagate in one clock cycle

Sequential Logic

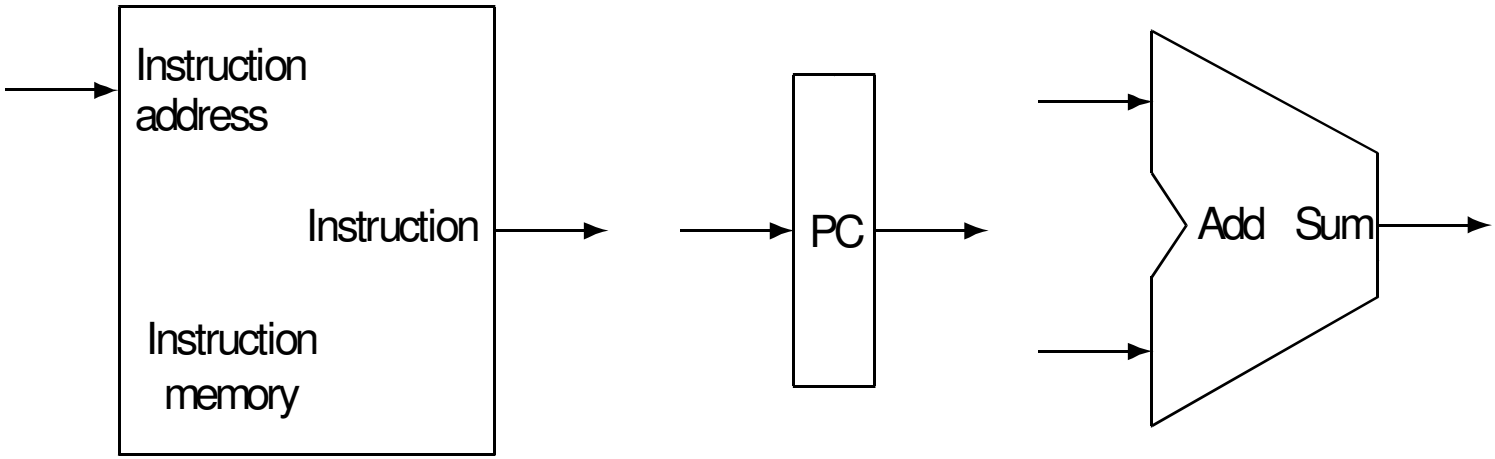
- How does a Sequential Logic circuit operate?

1. **When the clock issues a pulse →**
   - Each SE examines outputs of combinational logic, changes affected 0/1 state to opposite state (takes a few nanoseconds) and transforms to a stable state
2. **The change in state of some of the stored bits triggers changes in some combinational logic and subsequently changes in other SE (unstable state).**
   - Takes few nanoseconds for every output of the combinational logic block to reach a stable state
3. **When all outputs of combinational logic blocks are stable, the clock issues another pulse**
   - Repeat steps 1 and 2.

Require Long Clock Period Restriction to ensure output of combinational logic is stable

Building a Datapath

Store and Access Instructions



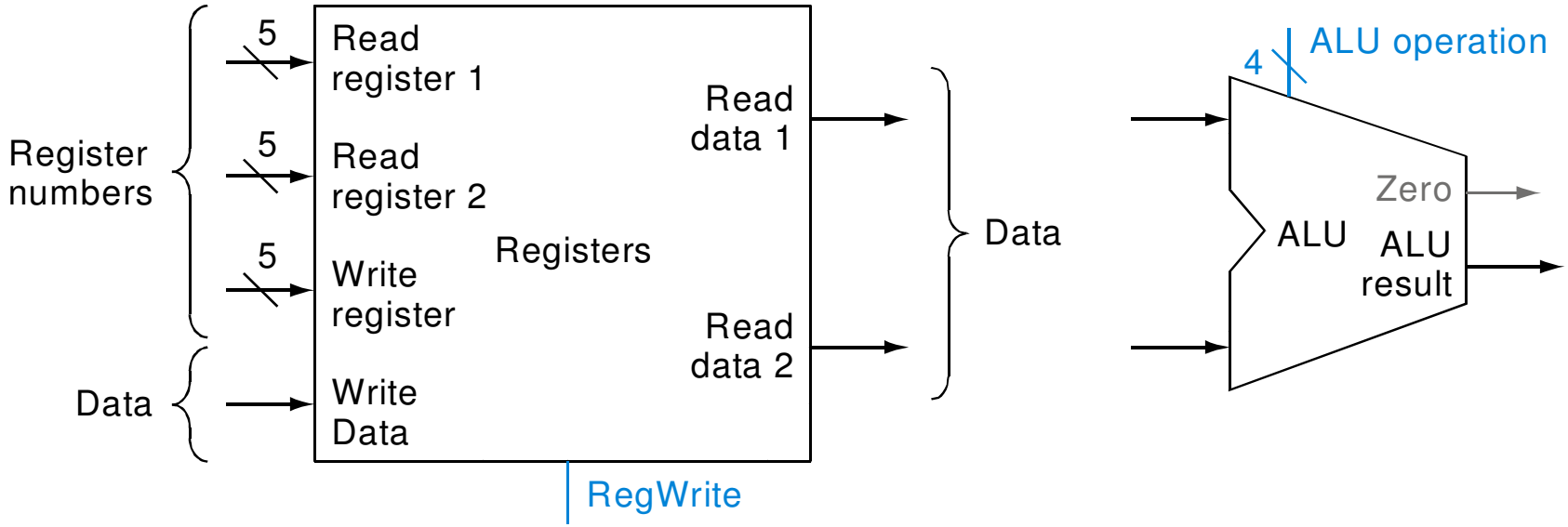
a. Instruction memory b. Program counter c. Adder

Datapath Elements to store and Access Instructions

Instruction memory is SE (but no write function)→ Combinational logic
PC → SE

Building a Datapath

R-Format ALU Operations

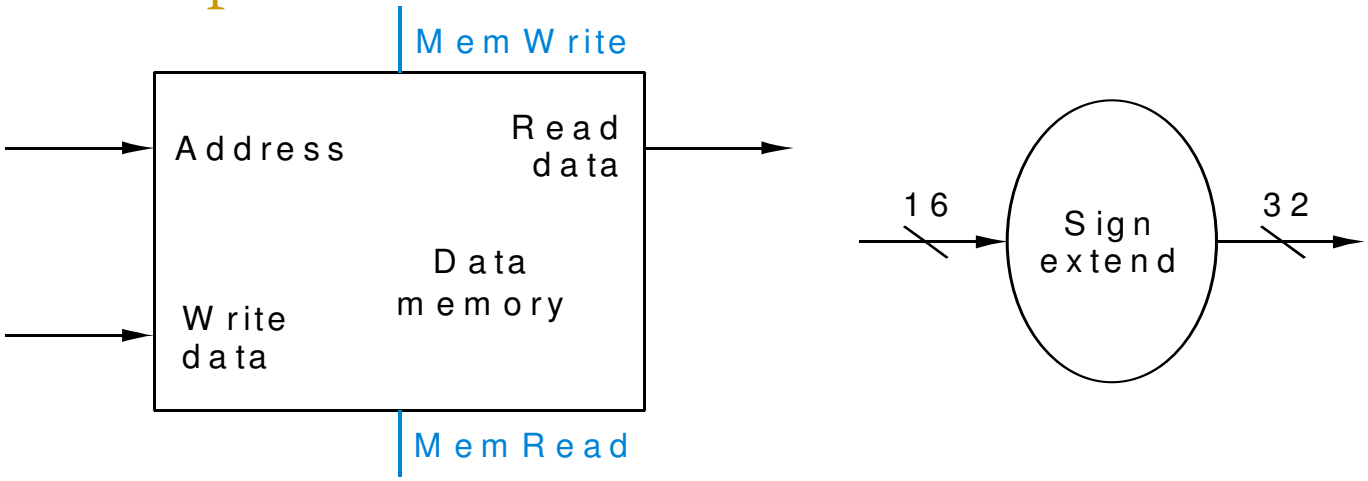


a. Registers b. ALU

Units needed to implement R-format ALU operations
Register File contains 2^5 registers (read/write) → SE

Building a datapath

lw and sw operations



a. Data memory unit b. Sign-extension unit

Additional Units needed to implement lw and sw

Memory Unit is a SE:

Inputs: Address & Write Data

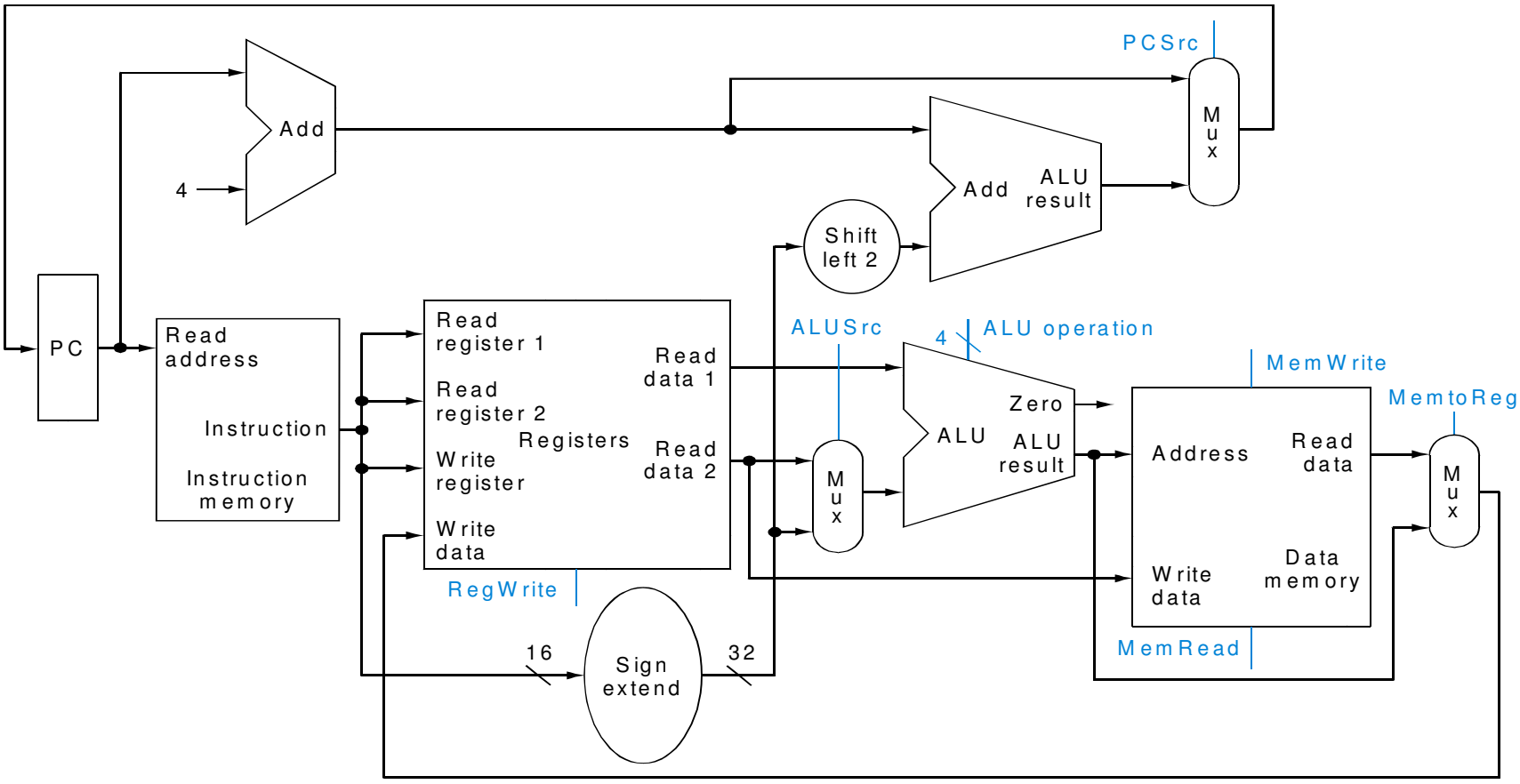
Output: Read Result

Sign Extension Unit:

Converts a 16-bit input to 32-bit output

Building Datapath

- **Use Multiplexors to combine: the components**



A Simple Implementation Scheme
ALU Control

- Let's look at a subset of core MIPS ISA:
  - The memory-reference instructions: *load word (lw), store word (sw)*
  - The arithmetic-logical instructions: `add, sub, and, or, set on less than (slt)`
  - The control flow instructions: *branch equal (*`beq`*)* and `jump (j)`

A Simple Implementation Scheme
ALU Control

- ALU has 4 control Inputs (2^4 combinations)
  - Examine Subset (6 combinations)

ALU Control Input	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

A Simple Implementation Scheme

ALU Control

- **Selecting the operations to perform (ALU, read/write, etc.)**
- **Controlling the flow of data (multiplexor inputs)**
- **Information comes from the 32 bits of the instruction**

What should the ALU do with this instruction ?

add $8, $17, $18 **Instruction Format:**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

Compute *AND, OR, subtract*, add or *slt* depending on 6-bit funct

ALU Control

What should the ALU do with this instruction?

lw $1, 100($2)

35	**2**	**1**	**100**
op	**rs**	**rt**	**16 bit offset**

Compute the memory address…….. But how?

ALU Control

Need a mapping for hardware to compute the 4-bit ALU control input signals

Instruction Type:

opcode	ALUop (is set to)
lw, sw	00
beq	01
arithmetic	10

Function Code for Arithmetic…..

ALU Control Bit Settings

Instruction opcode	ALUop	Instruction Operation	Funct Field	Desired ALU action	ALU control input
lw	00	load word	XXXXXX	add	**0010**
sw	00	store word	XXXXXX	add	**0010**
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	**0010**
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

X ~ don't care bit

ALU Control Bit Settings

- Control Bit Settings depends on:
  - ALUop bits
  - Function field codes for R-type instruction

Instruction opcode	ALUop	Funct Field	ALU control input
lw	00	XXXXXX	0010
sw	00	XXXXXX	0010
Branch equal	01	XXXXXX	0110
R-type	10	100000	0010
R-type	10	100010	0110
R-type	10	100100	0000
R-type	10	100101	0001
R-type	10	101010	0111

ALU Control Logic

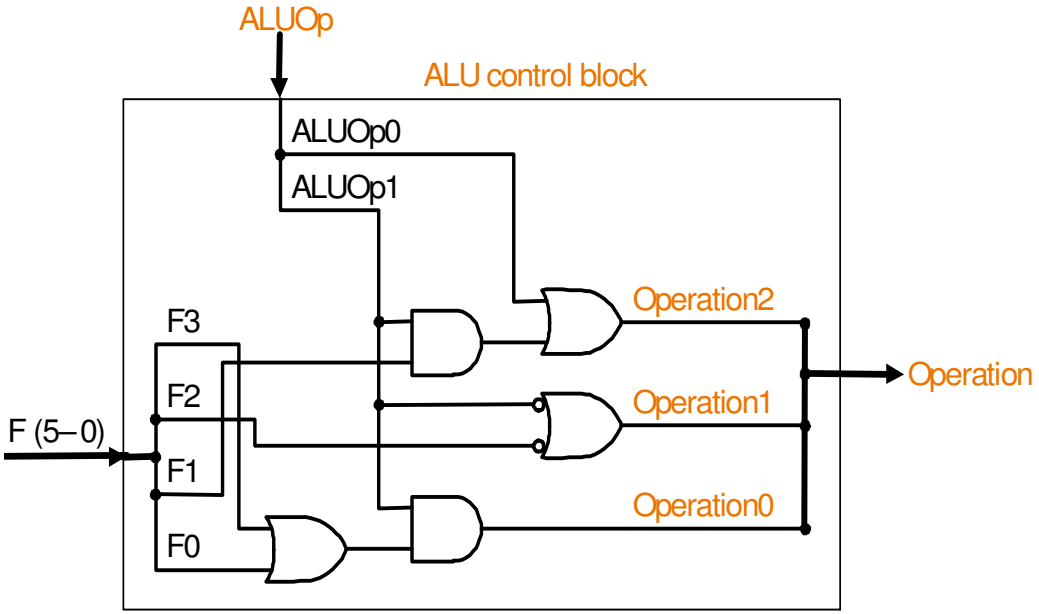
Truth Table (the 3-ALU control bits)

ALUop		Funct Field						Operation (Output)
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

ALU Control function has 3-distinct outputs: operation2 operation1 operation0

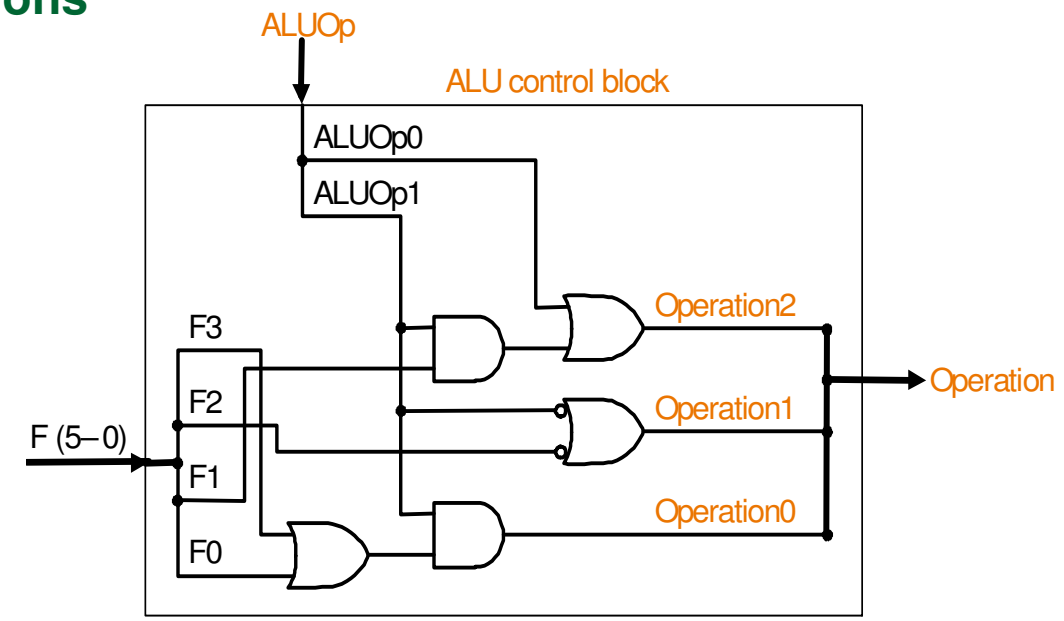
ALU Control Logic

Mapping of ALU Control Function to Gates



What is the mapping for the ALU control functions?

Mapping of ALU Control Functions



ALUop		Funct Field						Operation (Output)
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0011
1	X	X	X	1	0	1	0	0111

ALU Control Input	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than

Designing the Main Control Unit
Datapath

- Let's examine MIPS Instruction Fields & Control Lines:

R-Type Instruction

Field	op	rs	rt	rd	shamt	funct
Bit Position	31:26	25:21	20:16	15:11	10:16	5:0
# of bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Observations:

- **The op field** in bits 31:26 always set to **000000**
- **Source** Registers: Identified in bits **25:21** & bits **20:16** resp.
- **Destination** Register: Identified in bits **15:11**
- The shamt field is never used (ignore)
- The funct field is coded as per ALU design

Designing the Main Control Unit
Datapath

- *Load word, store word and branch-on-equal*:

I-Type Instruction

Field	op	rs	rt	address
Bit Position	31:26	25:21	20:16	15:0
# of bits	6 bits	5 bits	5 bits	16 bits

Observations:

- **The op field - same position as R-type** format
- **The base register (rs) and rt fields - same positions as R-type format**
- The address field in bits 15 - 0

Designing the Main Control Unit
Observations

- The Op field is always in bits 31 – 26
  - Labeled as op5, Op4, Op3, Op2, Op1, Op0
- Every instruction reads register specified by rs field
- Every instruction, except load word, reads the register specified by **rt** field
  - The load word writes to the **rt** field
- The base register for load and store word is always specified by rs field
- The destination register is one of two places:
  - For *load word*, it is **rt** field (bits 20 -16)
  - For R-Type instruction it is in rd field (bits 15 -11)
  - Thus a multiplexor needs to be added to the datapath to select the correct field for the write register input

Designing the Main Control Unit

Effects of control signals

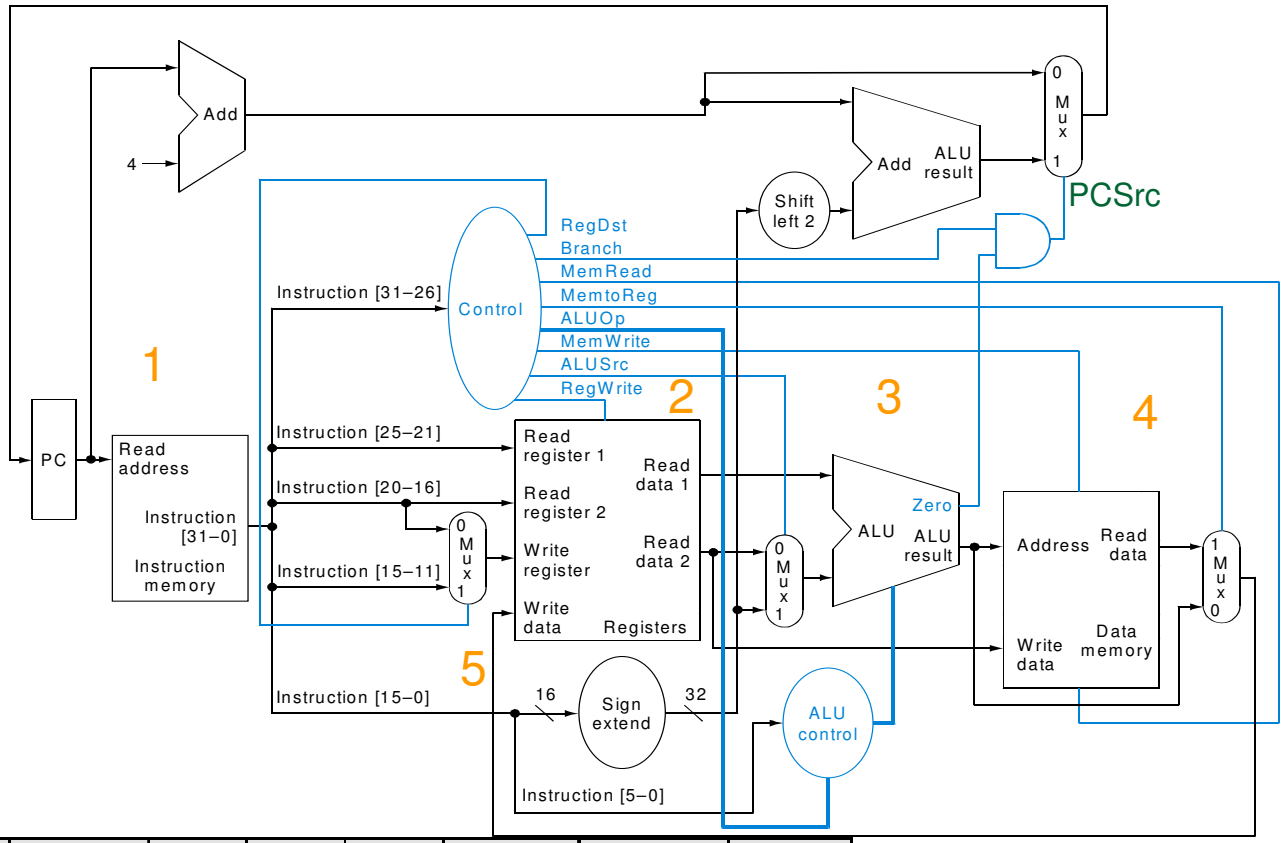
- Fig 5.17 shows the datapath with 7 control lines

Signal Name	Effect if 0	Effect if 1
RegDst	Destination number of write register is in **rt** field	Destination write register is in **rd** field
RegWrite	None	Store Write data input into Destination register
ALUSrc	Send rt register to ALU	Send sign-extended lower 16 bits of instruction to ALU
PCSrc	Send PC+4 to PC	Send branch target to PC
MemRead	None	Read contents of addressed memory word
MemWrite	None	Store Write data into addressed memory word
MemtoReg	Send ALU result to register file	Send word to register file

Designing the Main Control Unit

Control unit computes settings of control lines

1. **Instruction Memory**
2. **Register Read**
3. **ALU Operation**
4. **Data Memory**
5. **Register Write**



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Designing the Main Control Unit
Effects of control signals

- Fig 5.17 shows two additional control line control lines
  - ALUOp1 and ALUOp0 (ALUOp)
- The states of all control lines (except PCSrc) are determined by Op code field
- The state of PCSrc is determined by
  - AND gate with inputs: Zero output of ALU and Branch control line
  - The Branch control line is asserted if Op field equals 000100 (beq)

Designing the Main Control Unit

Control unit computes settings of control lines

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

R-format Instructions: add, sub, slt

Source Register fields ~ rs and rt

Destination register field ~ rd

add **$t1**, **$t2**, **$s1**

The datapath in operation

ALUop		Funct Field						Operation (Output)
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0011
1	X	X	X	1	0	1	0	0111

Signal Name	Effect if 0	Effect if 1
RegDst	Destination number of write register is in **rt** field	Destination write register is in **rd** field
RegWrite	None	Store Write data input into Destination register
ALUSrc	Send rt register to ALU	Send sign-extended lower 16 bits of instruction to ALU
PCSrc	Send PC+4 to PC	Send branch target to PC
MemRead	None	Read contents of addressed memory word
MemWrite	None	Store Write data into addressed memory word
MemtoReg	Send ALU result to register file	Send word to register file

Instruction	RegDst	ALUSrc	Memto-Reg	RegWrite	MemRead	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

The datapath in operation

Truth Table

Input/ Output	Signal Name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	AlUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The datapath in operation

Input/ Output	Signal Name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	AlUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Signal Name	Effect if 0	Effect if 1
RegDst	Destination number of write register is in **rt** field	Destination write register is in **rd** field
RegWrite	None	Store Write data input into Destination register
ALUSrc	Send rt register to ALU	Send sign-extended lower 16 bits of instruction to ALU
PCSrc	Send PC+4 to PC	Send branch target to PC
MemRead	None	Read contents of addressed memory word
MemWrite	None	Store Write data into addressed memory word
MemtoReg	Send ALU result to register file	Send word to register file

Instruction	RegDst	ALUSrc	Memto-Reg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Single-cycle implementation

Let's summarize

Instruction Class	Major Functional Units used by Instruction class/Single Clock cycle				
	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write
R-type	Fetch Instruction	Access Register	ALU		Register access
Load word	Fetch Instruction	Access Register	ALU	Memory access	Register access
Store word	Fetch Instruction	Access Register	ALU	Memory access	
Branch	Fetch Instruction	Access Register	ALU		

Other Units: PC, Control, Multiplexors

Single-cycle implementation

Let's summarize

Assume Critical Machine Operation Times: Memory Unit ~ 200 ps, ALU and adders ~ 100ps, Register file (read/write) ~ 50 ps. Estimate **clock cycle** for the machine

Instruction Class	Major Functional Units used by Instruction Operation Times					Period
	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	**600 ps**
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps

Single Clock cycle determined by longest instruction period = **600 ps**

A Multicycle Implementation
Datapath

- Basic Idea ("*Divide and Conquer*")
  - Break MIPS Instructions into independent and "*manageable*" steps
    - Balance workload across steps
  - Each step takes one clock cycle
  - Re-use functional units in different steps
    - Pack as much work into each step
    - At most one ALU operation, one register file access or one memory access per step

A Multicycle Datapath Implementation
Approach

- A single memory replaces the instruction memory and data memory associated with single-cycle approach
- A single ALU performs the addition operations (thereby eliminating the two adders for single-cycle operation)
- A 32-bit Instruction Register (IR) is added to store instruction fetched from memory
- A 32-bit Memory Data Register (MDR) is added to hold data fetched from memory
- Two 32-bit registers, A and B are added to store values of source registers *rs* and *rt* respectively
- A 32-bit register, ALUOut, is added to temporarily hold the results of the ALU
- Use FSM to determine control signals (instead of instructions)

A Multicycle Datapath Implementation

Actions of the 1bit Control Signals

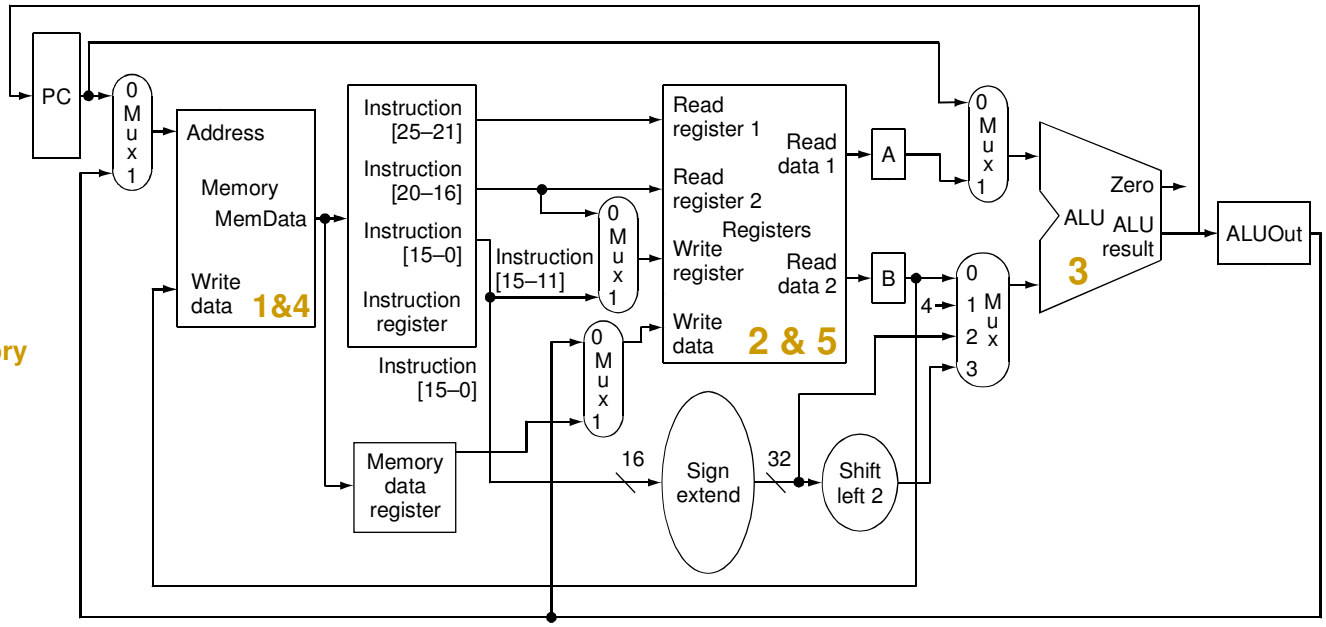
Signal Name	Effect if 0	Effect if 1
RegDst	Destination number of write register is in **rt** field	Destination write register is in **rd** field
RegWrite	None	Store Write data input into Destination register
ALUSrcA	First ALU Operand = PC	First ALU Operand = A register
MemRead	None	Read contents of addressed memory word
MemWrite	None	Store Write data into addressed memory word
MemtoReg	Send ALUOut result to register file	Send MDR to register file
IorD	Memory address is PC	Memory address is ALUOut
IRWrite	None	Store memory word into the IR
PCWrite	None	Store a new value into the PC
PCWriteCond	None	Store a new value into PC if ALU result is 0

A Multicycle Datapath Implementation
Actions of the 2 bit **Control Signals**

Signal Name	Value	Effect if 1
ALUOp	00	ALU performs an ADD operation
	01	ALU performs a subtract operation
	10	ALU performs operation specified by *funct* field of the IR
ALUSrcB	00	Second ALU Operand is the B register
	01	Second ALU Operand is the constant 4
	10	Second ALU Operand is sign-extended lower 16 bits of the IR
	11	Second ALU Operand is sign-extended lower 16 bits of the IR shifted left 2 places
PCSource	00	New PC value comes from ALU (PC+4)
	01	New PC value comes from branch target address in ALUOut
	10	New PC value is jump target address

A Multicycle Datapath Implementation
Approach

1. **Instruction Memory**
2. **Register Read**
3. **ALU Operation**
4. **Data Memory**
5. **Register Write**



Breaking the Instructions

Example: add $t0, $t1, $t2

- The add instruction changes value of a register (Reg).
  - Register specified by bits 15:11 of Instruction.
- Instruction specified by the PC.
- New value is the sum (“op”) of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=
Reg[Memory[PC][25:21]] op
                  Reg[Memory[PC][20:16]]
```

Breaking the Instructions

Example: add $t0, $t1, $t2

```
Reg[Memory[PC][15:11]] <=
Reg[Memory[PC][25:21]] op
                  Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[15:11]] <= ALUOut`

A Multicycle Datapath Implementation

Five Execution steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. Execution, Memory Address Computation, or Branch Completion
4. Memory Access or R-type instruction completion
5. Memory Read Completion

1. Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.

```
IR <= Memory[PC];
PC <= PC + 4;
```

Why update the PC now?

Signal Settings:

Signal Name	Set to 0	Set to 1
ALUSrcA	**First ALU Operand = PC**	
MemRead		**Read contents of addressed memory word**
IorD	**Memory address is PC**	
IRWrite		**Store memory word into the IR**
PCWrite		**Store a new value into the PC**

ALUSrcB	01	Second ALU Operand is the constant 4
ALUOp	00	New PC value comes from ALU (PC+4)

2. Instruction Decode and Register Fetch

- Read registers *rs* and *rt* in case we need them
- Compute the branch address in case the instruction is a branch

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0])
<< 2);
```

What is the signal setting for step 2?

3. Execution, Memory Address Computation, or Branch Completion

- ALU performs one of three functions, based on instruction type
  - Memory Reference:
    ```
    ALUOut <= A + sign-extend(IR[15:0]);
    ```
  - R-type:
    ```
    ALUOut <= A op B;
    ```
  - Branch:
    ```
    if (A==B) PC <= ALUOut;
    ```

Determine the signal settings for each instruction type

4. Memory Access or R-type instruction completion

- Loads and stores access memory

```
MDR <= Memory[ALUOut];
    or
Memory[ALUOut] <= B;
```

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

Memory Read Completion

- `Reg[IR[20:16]] <= MDR;`

Only the *load word* instruction executes this step

A Multicycle Datapath Implementation

Let's summarize

Step Name	Action for R-type Instructions	Action for memory-reference instruction	Action for branches
Instruction fetch	IR <= Memory[PC] PC <= PC + 4		
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)		
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B	
Memory read completion		Load: Reg[IR[20:16]] <= MDR	

Multicycle Datapath
Example

- How many cycles will it take to execute the following MIPS code

```
lw  $s2, 0($s3)
lw  $s3, 4($s3)
beq $s2, $s3, Label          #assume not
add $s5, $s2, $s3
sw $s5, 8($s3)
```

Finite State Machine (FSM)

Overview

- Subsystems characterized by:
  - Set of **well defined** states (States: 1, 2, 3,4, 5)
  - State function determined typically by
    - Current state
    - Input values
  - Output function determined typically by
    - Current state
    - Input values