Single Cycle Processor Design

Bölüm 4

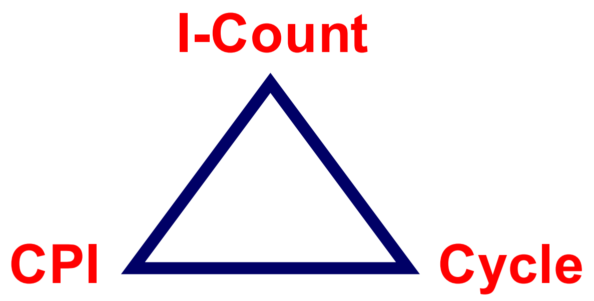
Bilgisayar Mimarisi

Presentation Outline

- Designing a Processor: Step-by-Step
- Datapath Components and Clocking
- Assembling an Adequate Datapath
- Controlling the Execution of Instructions
- The Main Controller and ALU Controller
- Drawback of the single-cycle processor design

The Performance Perspective

- Recall, performance is determined by:
  - Instruction count
  - Clock cycles per instruction (CPI)
  - Clock cycle time



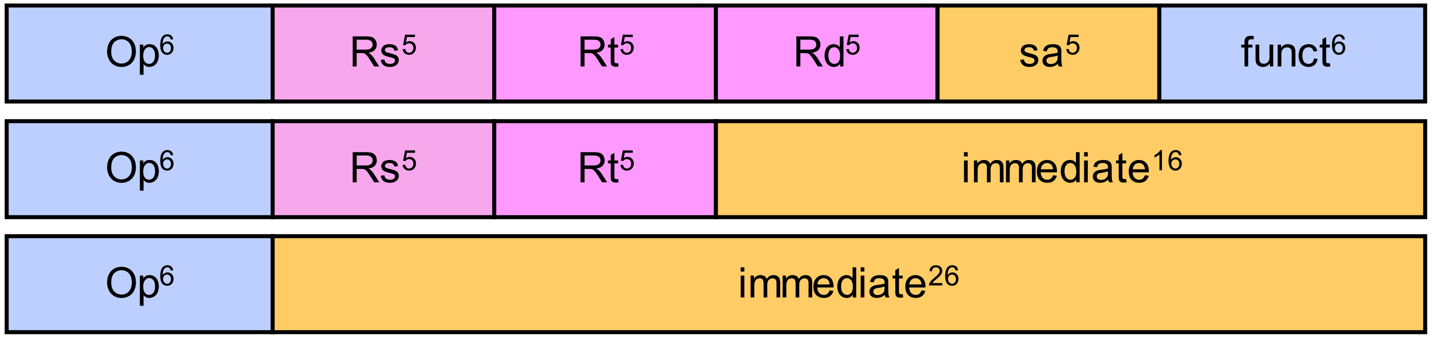
- Processor design will affect
  - Clock cycles per instruction
  - Clock cycle time
- Single cycle datapath and control design:
  - Advantage: One clock cycle per instruction
  - Disadvantage: long cycle time

Designing a Processor: Step-by-Step

- Analyze instruction set => datapath requirements
  - The meaning of each instruction is given by the register transfers
  - Datapath must include storage elements for ISA registers
  - Datapath must support each register transfer
- Select datapath components and clocking methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction
  - Determine the setting of control signals for register transfer
- Assemble the control logic

Review of MIPS Instruction Formats

- All instructions are 32-bit wide
- Three instruction formats: R-type, I-type, and J-type



- ✧ Op^6: 6-bit opcode of the instruction
- ✧ Rs^5, Rt^5, Rd^5: 5-bit source and destination register numbers
- ✧ sa^5: 5-bit shift amount used by shift instructions
- ✧ $funct^6$: 6-bit function field for R-type instructions
- ✧ $immediate^{16}$: 16-bit immediate value or address offset
- ✧ $immediate^{26}$: 26-bit target address of the jump instruction

MIPS Subset of Instructions

- Only a subset of the MIPS instructions are considered
  - ALU instructions (R-type): **add, sub, and, or, xor, slt**
  - Immediate instructions (I-type): **addi, slti, andi, ori, xori**
  - Load and Store (I-type): **lw, sw**
  - Branch (I-type): **beq, bne**
  - Jump (J-type): **j**
- This subset does not include all the integer instructions
- But sufficient to illustrate design of datapath and control
- Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers

Details of the MIPS Subset

Instruction	Meaning	Format					
add rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi rt, rs, im^{16}	add immediate	0x08	rs^5	rt^5	im^{16}		
slti rt, rs, im^{16}	slt immediate	0x0a	rs^5	rt^5	im^{16}		
andi rt, rs, im^{16}	and immediate	0x0c	rs^5	rt^5	im^{16}		
ori rt, rs, im^{16}	or immediate	0x0d	rs^5	rt^5	im^{16}		
xori rt, im^{16}	xor immediate	0x0e	rs^5	rt^5	im^{16}		
lw rt, im^{16}(rs)	load word	0x23	rs^5	rt^5	im^{16}		
sw rt, im^{16}(rs)	store word	0x2b	rs^5	rt^5	im^{16}		
beq rs, rt, im^{16}	branch if equal	0x04	rs^5	rt^5	im^{16}		
bne rs, rt, im^{16}	branch not equal	0x05	rs^5	rt^5	im^{16}		
j im^{26}	jump	0x02	im^{26}				

Register Transfer Level (RTL)

- ❖ RTL is a description of data flow between registers
- ❖ RTL gives a meaning to the instructions
- ❖ All instructions are fetched from memory at address PC

Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4
BEQ	if (Reg(Rs) == Reg(Rt))	
	PC ← PC + 4 + 4 × sign_extend(Im16)	
	else PC ← PC + 4	

Instructions are Executed in Steps

❖ R-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Execute operation:	ALU_result ← func(data1, data2)
	Write ALU result:	Reg(Rd) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ I-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Extend(imm16)
	Execute operation:	ALU_result ← op(data1, data2)
	Write ALU result:	Reg(Rt) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ BEQ	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Equality:	zero ← subtract(data1, data2)
	Branch:	if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
		else PC ← PC + 4

Instruction Execution – cont'd

❖ LW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch base register:	base ← Reg(Rs)
	Calculate address:	address ← base + sign_extend(imm16)
	Read memory:	data ← MEM[address]
	Write register Rt:	Reg(Rt) ← data
	Next PC address:	PC ← PC + 4
❖ SW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch registers:	base ← Reg(Rs), data ← Reg(Rt)
	Calculate address:	address ← base + sign_extend(imm16)
	Write memory:	MEM[address] ← data
	Next PC address:	PC ← PC + 4
❖ Jump	Fetch instruction:	Instruction ← MEM[PC]
	Target PC address:	target ← PC[31:28] || Imm26 || '00'
	Jump:	PC ← target

concatenation

Requirements of the Instruction Set

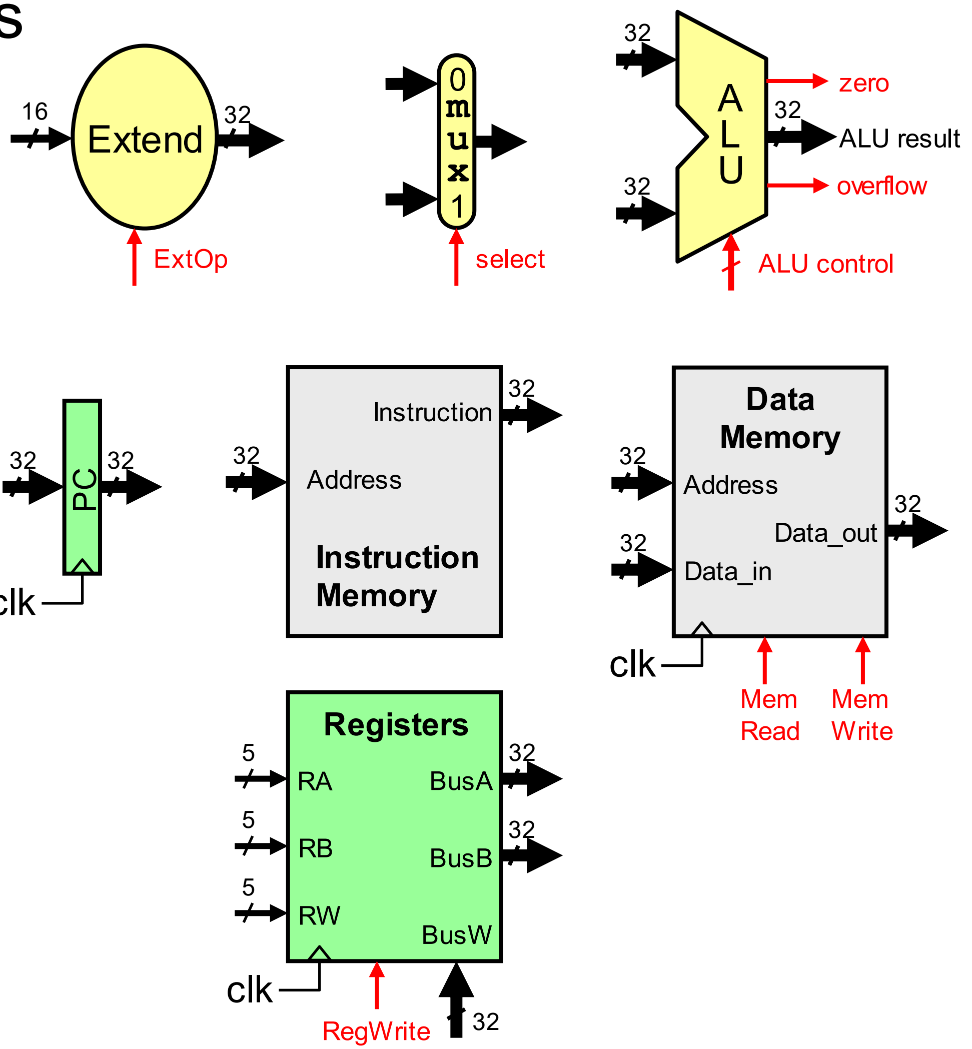
- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 31 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and Adder to increment PC
- Sign and Zero extender for immediate constant
- ALU for executing instructions

Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

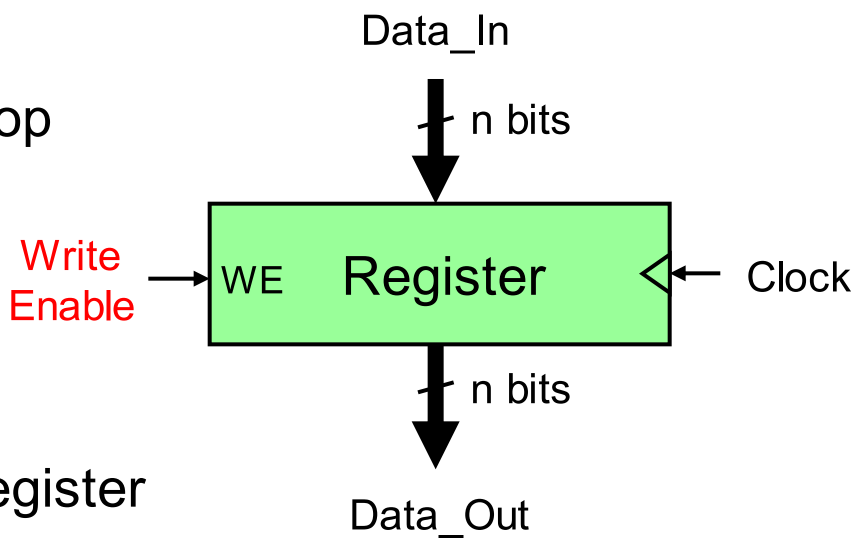
Components of the Datapath

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file
- Clocking methodology
  - Timing of writes



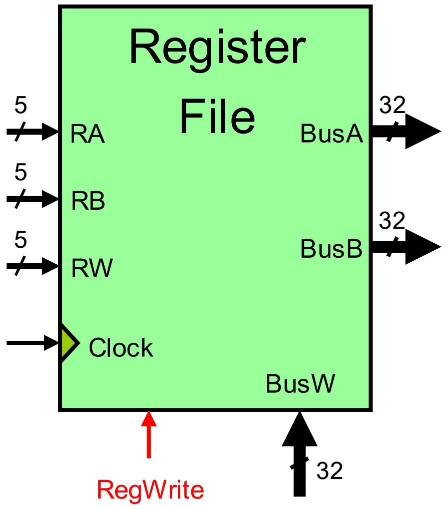
Register Element

- Register
  - Similar to the D-type Flip-Flop
- n-bit input and output
- Write Enable (WE):
  - Enable / disable writing of register
  - Negated (0): Data_Out will not change
  - Asserted (1): Data_Out will become Data_In after clock edge
- Edge triggered Clocking
  - Register output is modified at clock edge

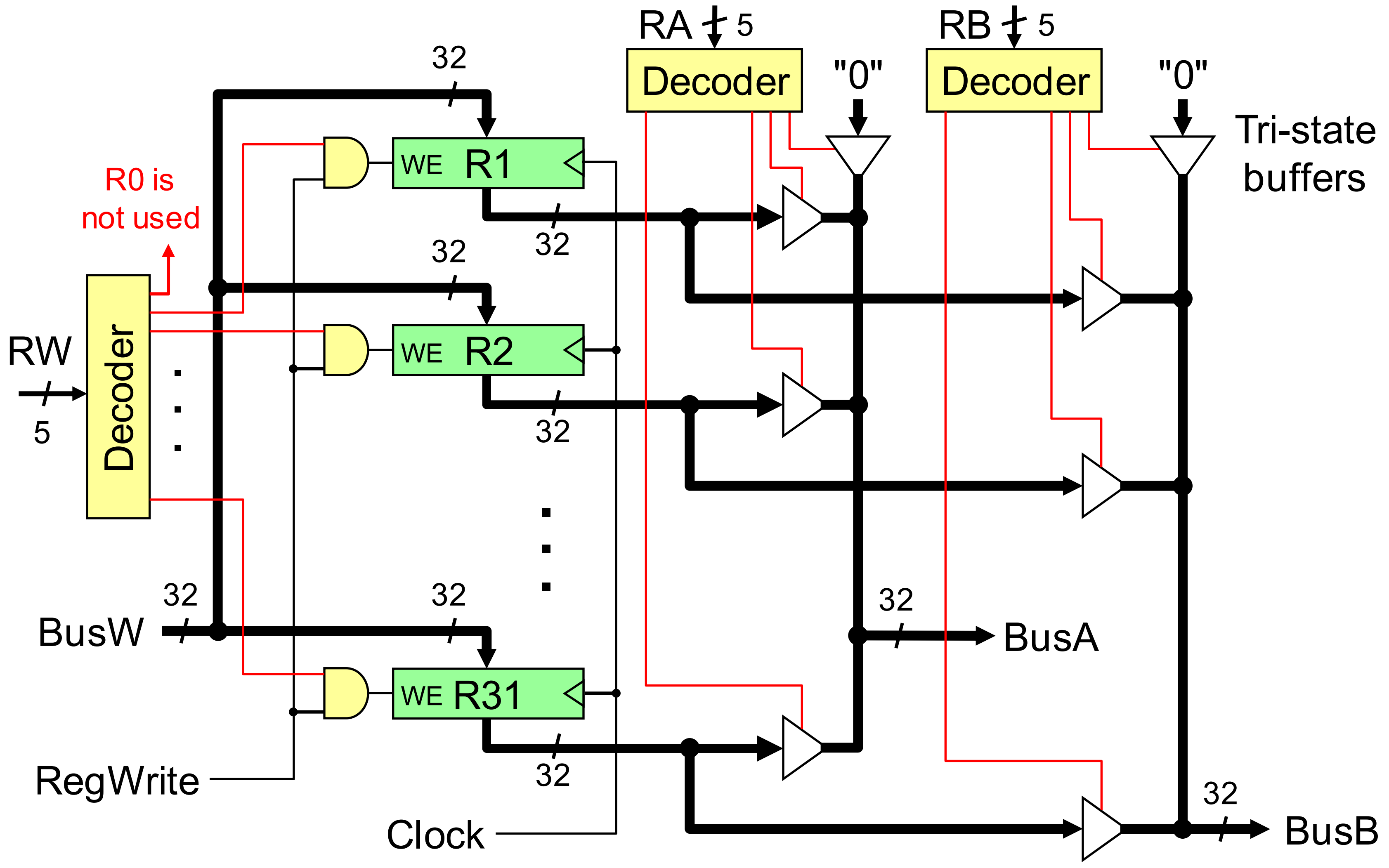


MIPS Register File

- Register File consists of 32 × 32-bit registers
  - BusA and BusB: 32-bit output busses for reading 2 registers
  - BusW: 32-bit input bus for writing a register when RegWrite is 1
  - Two registers read and one written in a cycle
- Registers are selected by:
  - RA selects register to be read on BusA
  - RB selects register to be read on BusB
  - RW selects the register to be written
- Clock input
  - The clock input is used ONLY during write operation
  - During read, register file behaves as a combinational logic block
    - RA or RB valid => BusA or BusB valid after access time

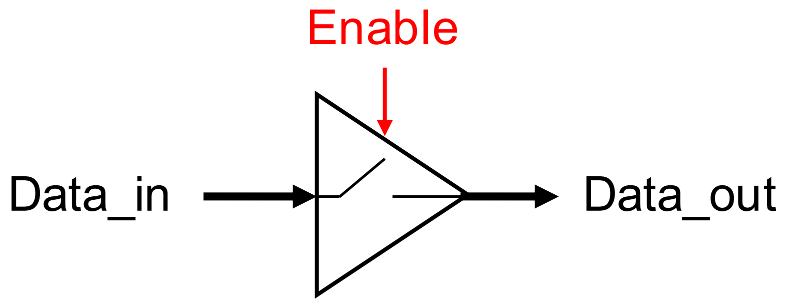


Details of the Register File



Tri-State Buffers

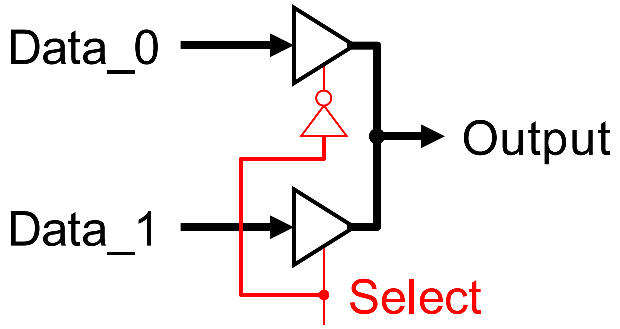
- Allow multiple sources to drive a single bus
- Two Inputs:
  - Data_in
  - Enable (to enable output)



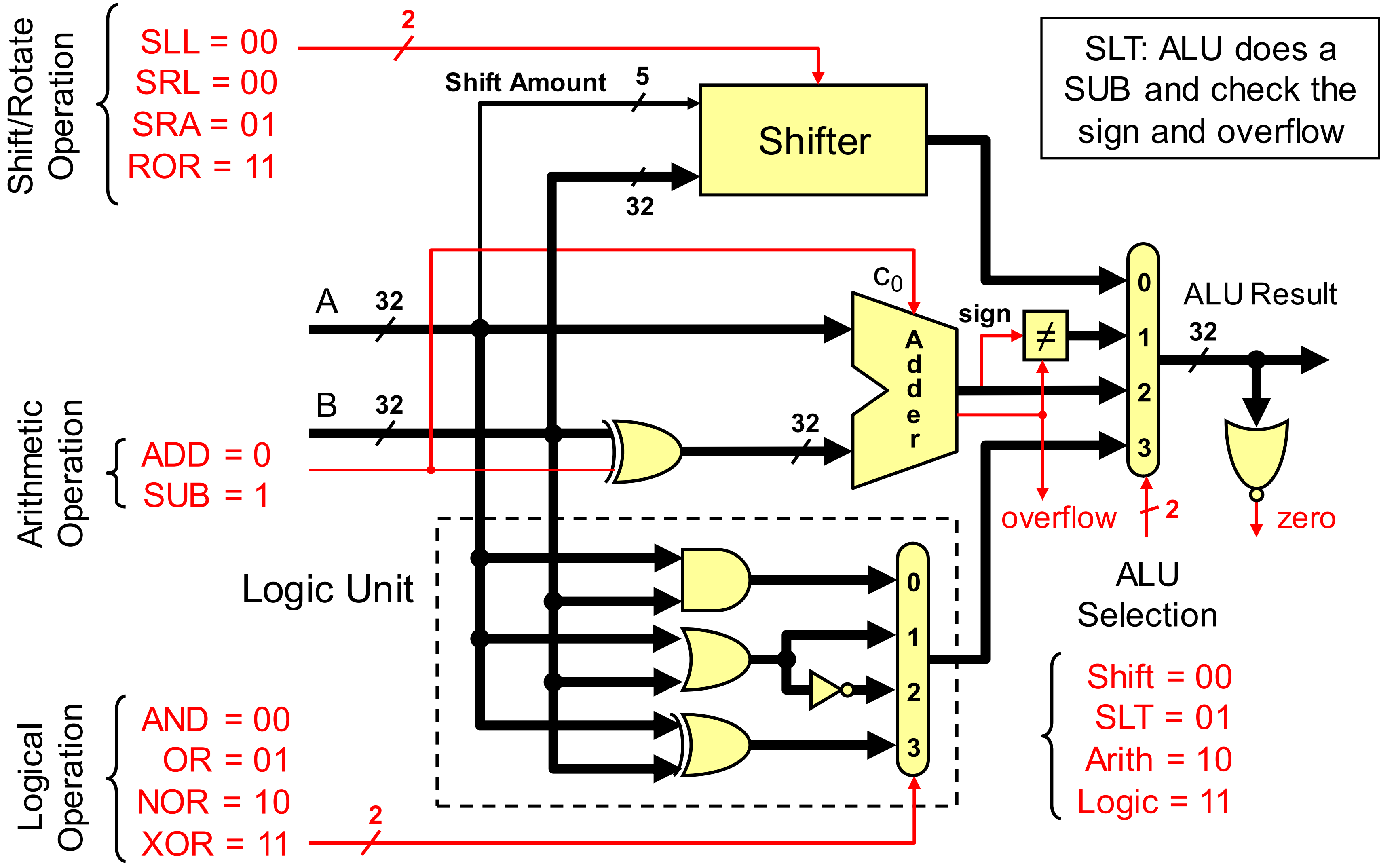
- One Output: Data_out
  - If (Enable) Data_out = Data_in

    else Data_out = High Impedance state (output is disconnected)

- Tri-state buffers can be used to build multiplexors

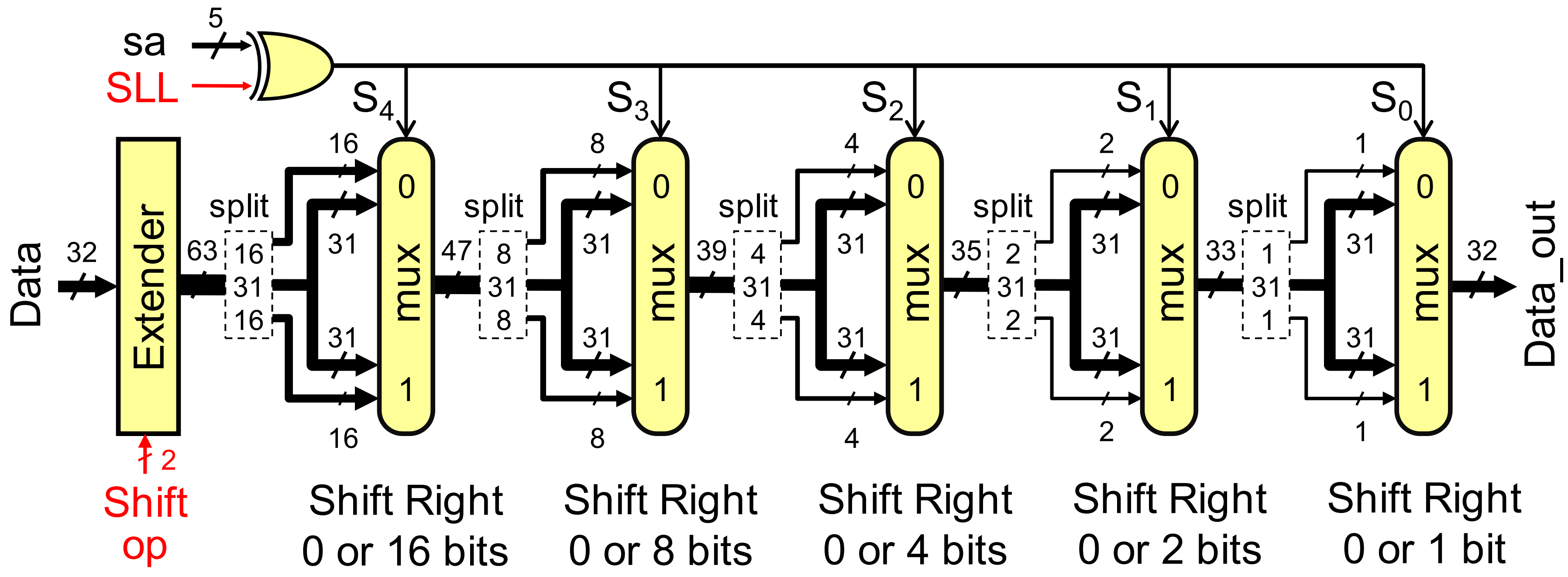


Building a Multifunction ALU



Details of the Shifter

- Implemented with multiplexers and wiring
- Shift Operation can be: SLL, SRL, SRA, or ROR
- Input Data is extended to 63 bits according to Shift Op
- The 63 bits are shifted right according to $S_4S_3S_2S_1S_0$



Details of the Shifter – cont'd

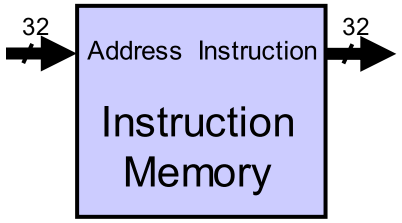
- Input data is extended from 32 to 63 bits as follows:
  - If shift op = SRL then ext_data[62:0] = 0^{31} || data[31:0]
  - If shift op = SRA then ext_data[62:0] = $\text{data}[31]^{31}$ || data[31:0]
  - If shift op = ROR then ext_data[62:0] = data[30:0] || data[31:0]
  - If shift op = SLL then ext_data[62:0] = data[31:0] || 0^{31}
- For SRL, the 32-bit input data is zero-extended to 63 bits
- For SRA, the 32-bit input data is sign-extended to 63 bits
- For ROR, 31-bit extension = lower 31 bits of data
- Then, shift right according to the shift amount
- As the extended data is shifted right, the upper bits will be: 0 (SRL), sign-bit (SRA), or lower bits of data (ROR)

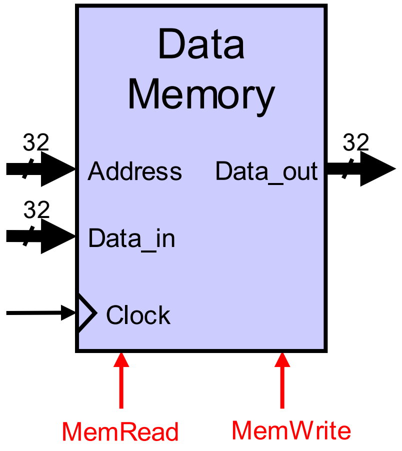
Implementing Shift Left Logical

- The wiring of the above shifter dictates a right shift
- However, we can convert a left shift into a right shift
- For SLL, 31 zeros are appended to the right of data
  - To shift left by 0 is equivalent to shifting right by 31
  - To shift left by 1 is equivalent to shifting right by 30
  - To shift left by 31 is equivalent to shifting right by 0
  - Therefore, for SLL use the 1's complement of the shift amount
- ROL is equivalent to ROR if we use (32 – rotate amount)
- ROL by 10 bits is equivalent to ROR by (32–10) = 22 bits
- Therefore, software can convert ROL to ROR

Instruction and Data Memories

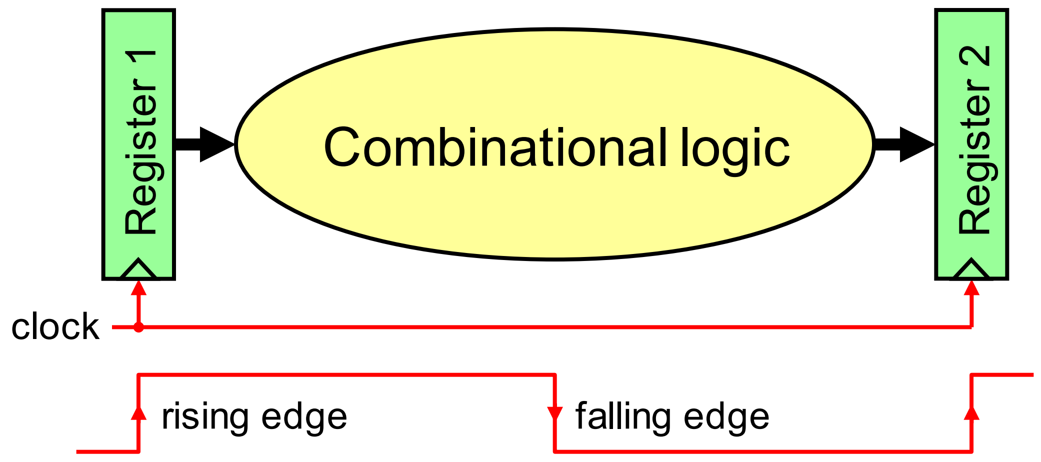
- Instruction memory needs only provide read access
  - Because datapath does not write instructions
  - Behaves as combinational logic for read
  - Address selects Instruction after access time
- Data Memory is used for load and store
  - MemRead: enables output on Data_out
    - Address selects the word to put on Data_out
  - MemWrite: enables writing of Data_in
    - Address selects the memory word to be written
    - The Clock synchronizes the write operation
- Separate instruction and data memories
  - Later, we will replace them with caches





Clocking Methodology

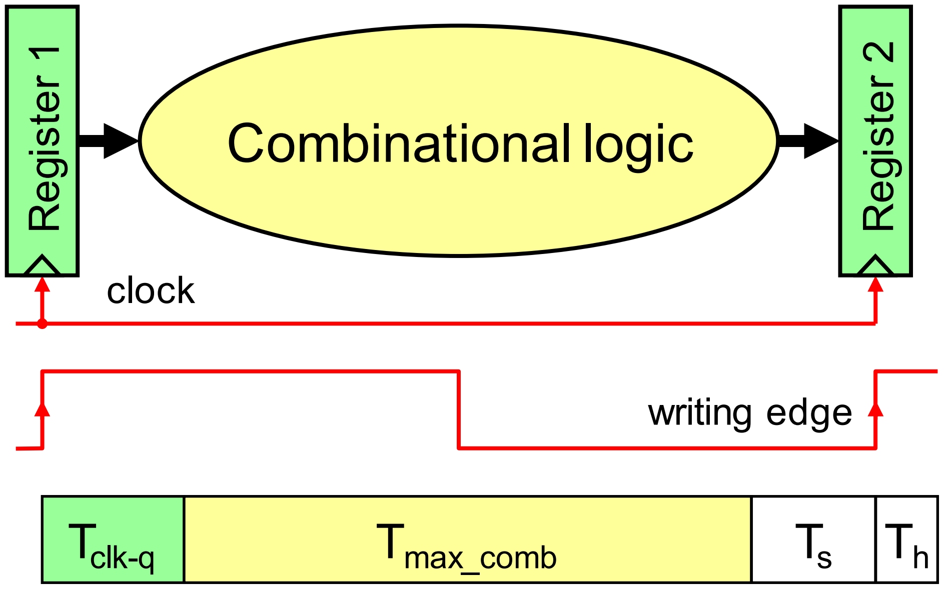
- Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- To ensure correctness, a clocking methodology defines when data can be written and read



- We assume edge-triggered clocking
- All state changes occur on the same clock edge
- Data must be valid and stable before arrival of clock edge
- Edge-triggered clocking allows a register to be read and written during same clock cycle

Determining the Clock Cycle

- With edge-triggered clocking, the clock cycle must be long enough to accommodate the path from one register through the combinational logic to another register



$$T_{cycle} \geq T_{clk\text{-}q} + T_{max\_comb} + T_s$$

- $T_{clk\text{-}q}$: clock to output delay through register
- T_{max_comb} : longest delay through combinational logic
- T_s : setup time that input to a register must be stable before arrival of clock edge
- T_h: hold time that input to a register must hold after arrival of clock edge
- Hold time (T_h) is normally satisfied since $T_{clk\text{-}q} > T_h$

Clock Skew

- Clock skew arises because the clock signal uses different paths with slightly different delays to reach state elements
- Clock skew is the difference in absolute time between when two storage elements see a clock edge
- With a clock skew, the clock cycle time is increased

$$\boxed{T_{cycle} \geq T_{clk\text{-}q} + T_{max\_combinational} + T_{setup} + T_{skew}}$$

- Clock skew is reduced by balancing the clock delays

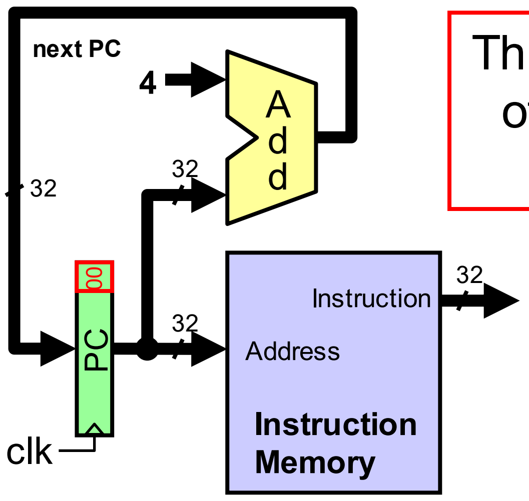
Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

Instruction Fetching Datapath

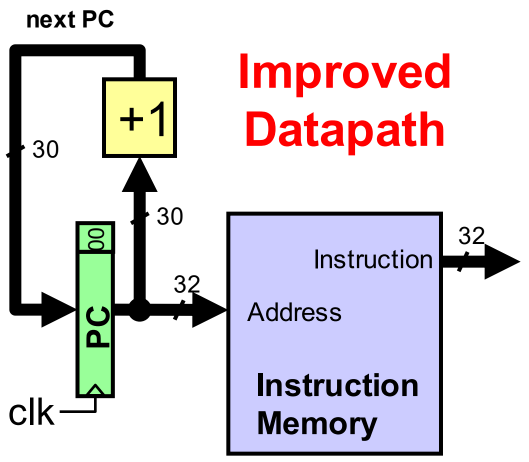
- We can now assemble the datapath from its components
- For instruction fetching, we need …
  - Program Counter (PC) register
  - Instruction Memory
  - Adder for incrementing PC

Improved datapath increments upper 30 bits of PC by 1



The least significant 2 bits of the PC are '00' since PC is a multiple of 4

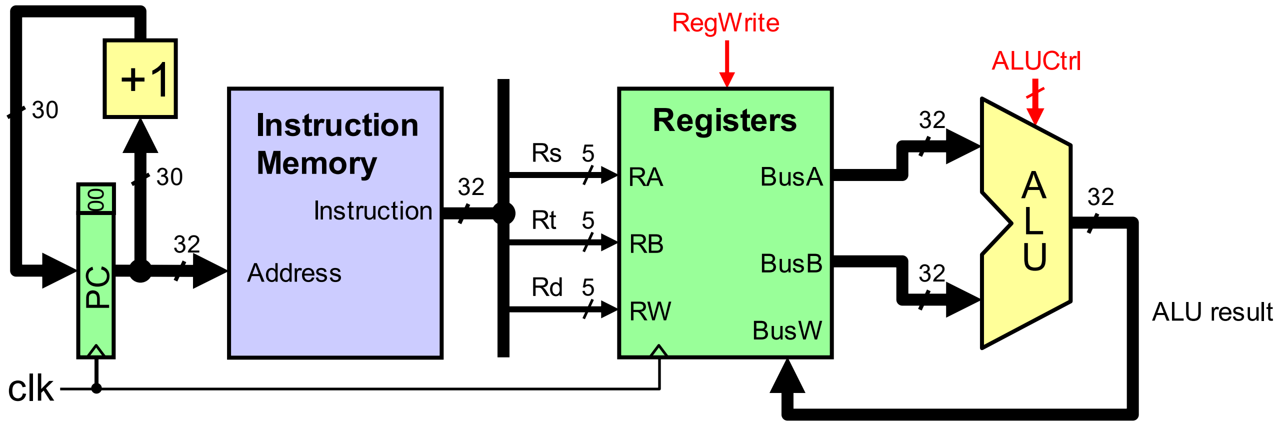
Datapath does not handle branch or jump instructions



Improved Datapath

Datapath for R-type Instructions

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6



Rs and Rt fields select two registers to read. Rd field selects register to write

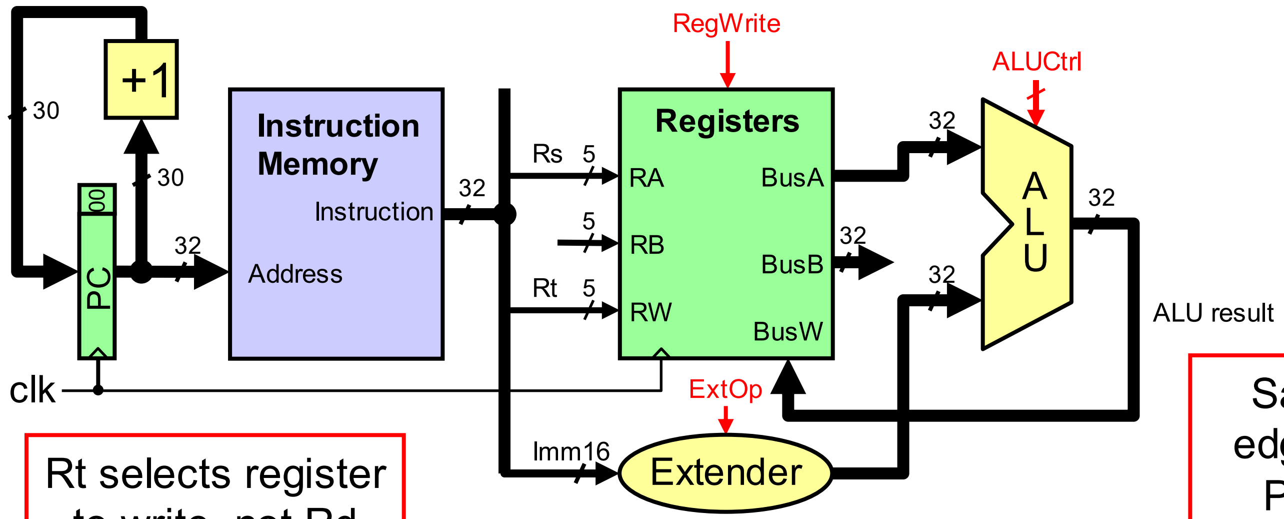
BusA & BusB provide data input to ALU. ALU result is connected to BusW

Same clock updates PC and Rd register

- Control signals
  - ALUCtrl is derived from the funct field because Op = 0 for R-type
  - RegWrite is used to enable the writing of the ALU result

Datapath for I-type ALU Instructions

Op^6	Rs^5	Rt^5	immediate^{16}



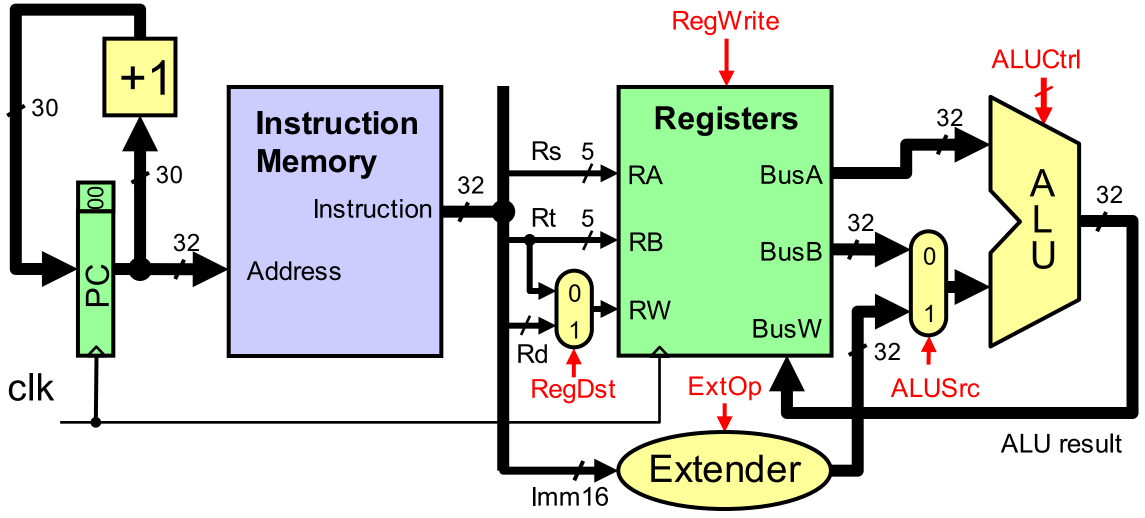
Rt selects register to write, not Rd

Same clock edge updates PC and Rt

Second ALU input comes from the extended immediate. RB and BusB are not used

- Control signals
  - ALUCtrl is derived from the Op field
  - RegWrite is used to enable the writing of the ALU result
  - ExtOp is used to control the extension of the 16-bit immediate

Combining R-type & I-type Datapaths

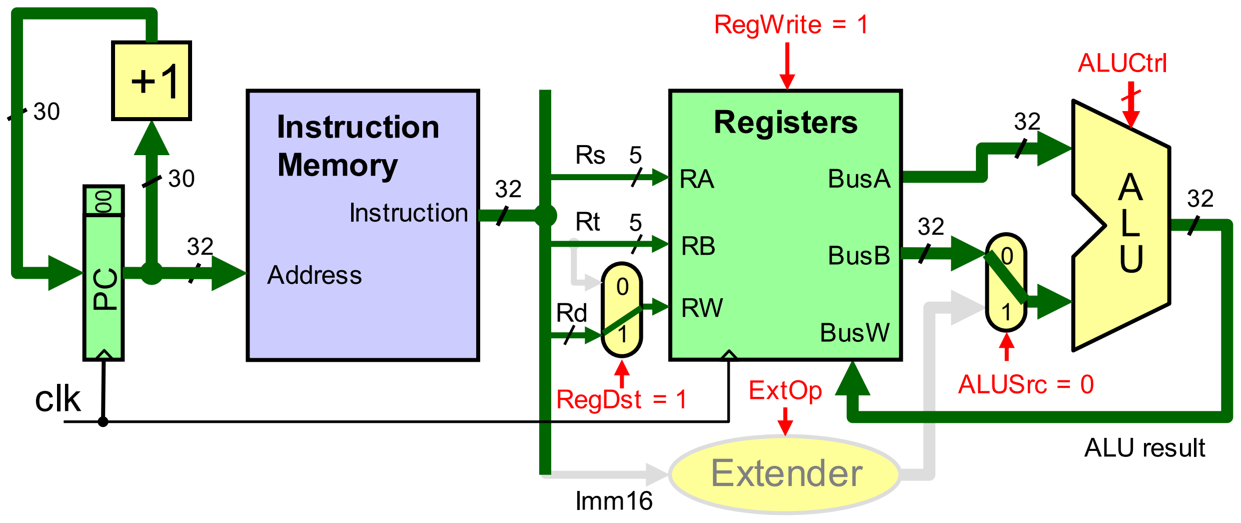


A mux selects RW as either Rt or Rd

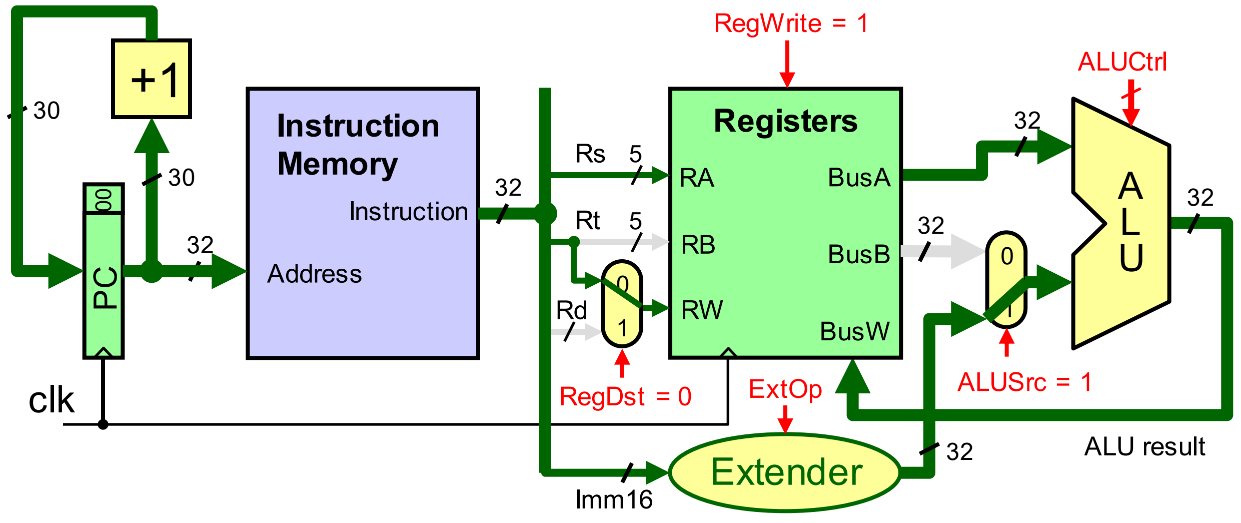
Another mux selects 2nd ALU input as either data on BusB or the extended immediate

- Control signals
  - ALUCtrl is derived from either the Op or the funct field
  - RegWrite enables the writing of the ALU result
  - ExtOp controls the extension of the 16-bit immediate
  - RegDst selects the register destination as either Rt or Rd
  - ALUSrc selects the 2nd ALU source as BusB or extended immediate

Controlling ALU Instructions



For R-type ALU instructions, RegDst is '1' to select Rd on RW and ALUSrc is '0' to select BusB as second ALU input. The active part of datapath is shown in **green**

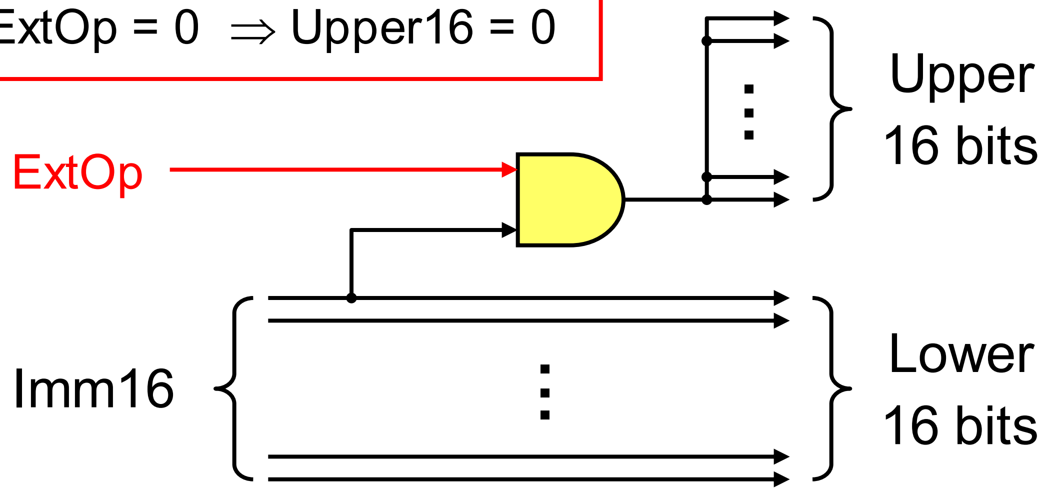


For I-type ALU instructions, RegDst is '0' to select Rt on RW and ALUSrc is '1' to select Extended immediate as second ALU input. The active part of datapath is shown in **green**

Details of the Extender

- Two types of extensions
  - Zero-extension for unsigned constants
  - Sign-extension for signed constants
- Control signal ExtOp indicates type of extension
- Extender Implementation: wiring and one AND gate

ExtOp = 0 ⇒ Upper16 = 0

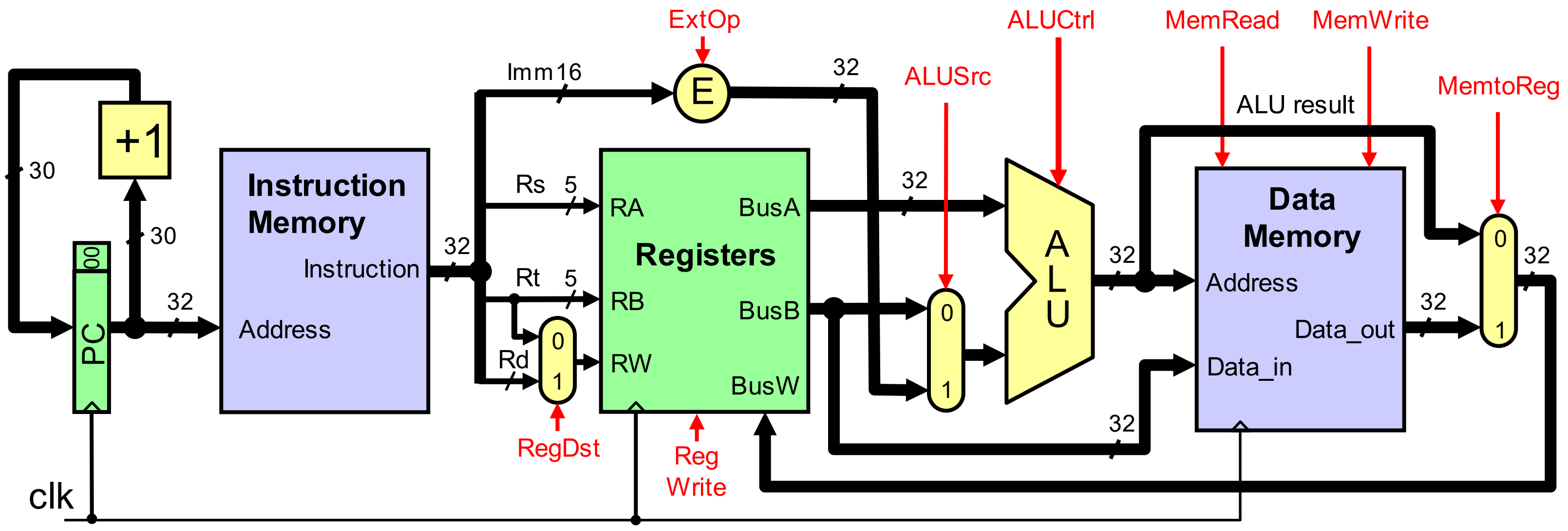


ExtOp = 1 ⇒

Upper16 = sign bit

Adding Data Memory to Datapath

❖ A data memory is added for load and store instructions



ALU calculates data memory address

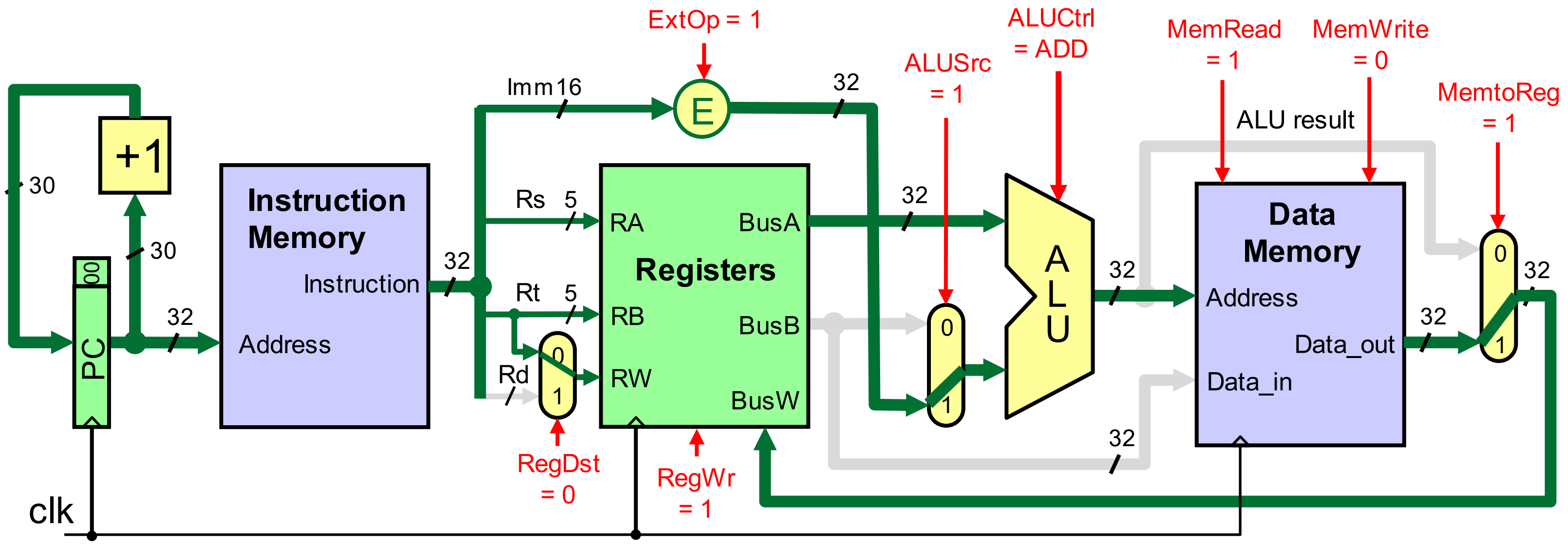
A 3rd mux selects data on BusW as either ALU result or memory data_out

BusB is connected to Data_in of Data Memory for store instructions

❖ Additional Control signals

- ✧ MemRead for load instructions
- ✧ MemWrite for store instructions
- ✧ MemtoReg selects data on BusW as ALU result or Memory Data_out

Controlling the Execution of Load



RegDst = '0' selects Rt as destination register

RegWrite = '1' to enable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = '1' selects extended immediate as second ALU input

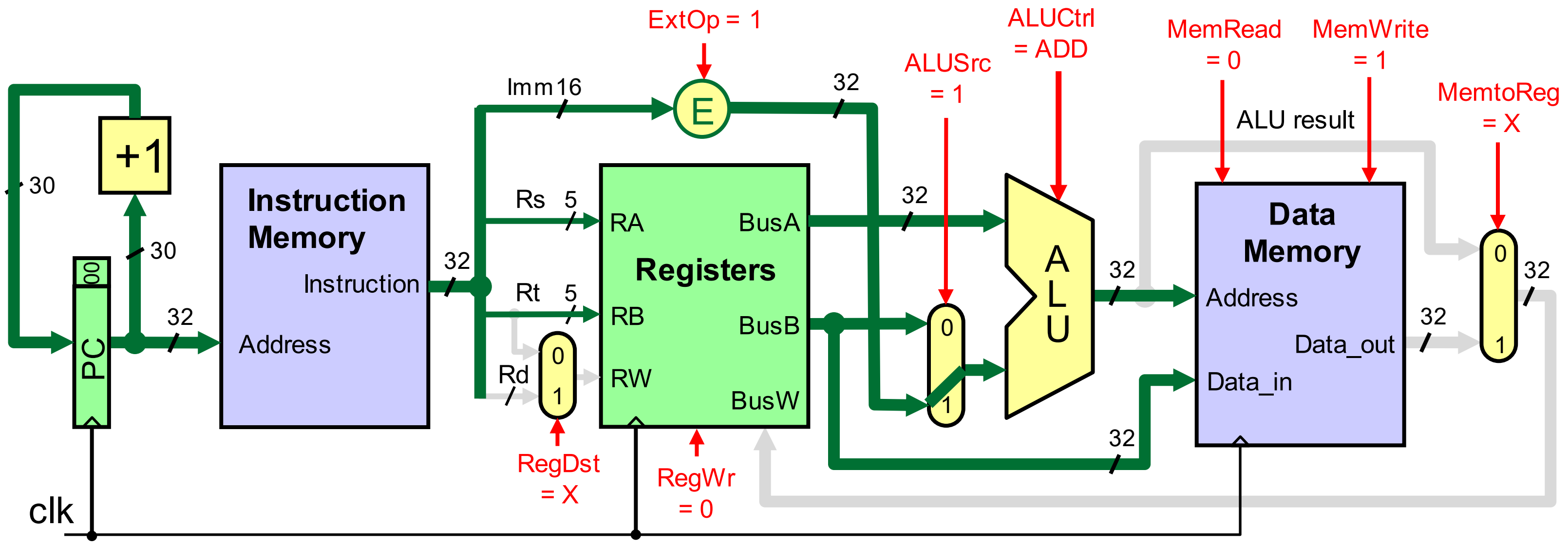
ALUCtrl = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemRead = '1' to read data memory

MemtoReg = '1' places the data read from memory on BusW

Clock edge updates PC and Register Rt

Controlling the Execution of Store



RegDst = ‘X’ because no register is written

RegWrite = ‘0’ to disable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = ‘1’ selects extended immediate as second ALU input

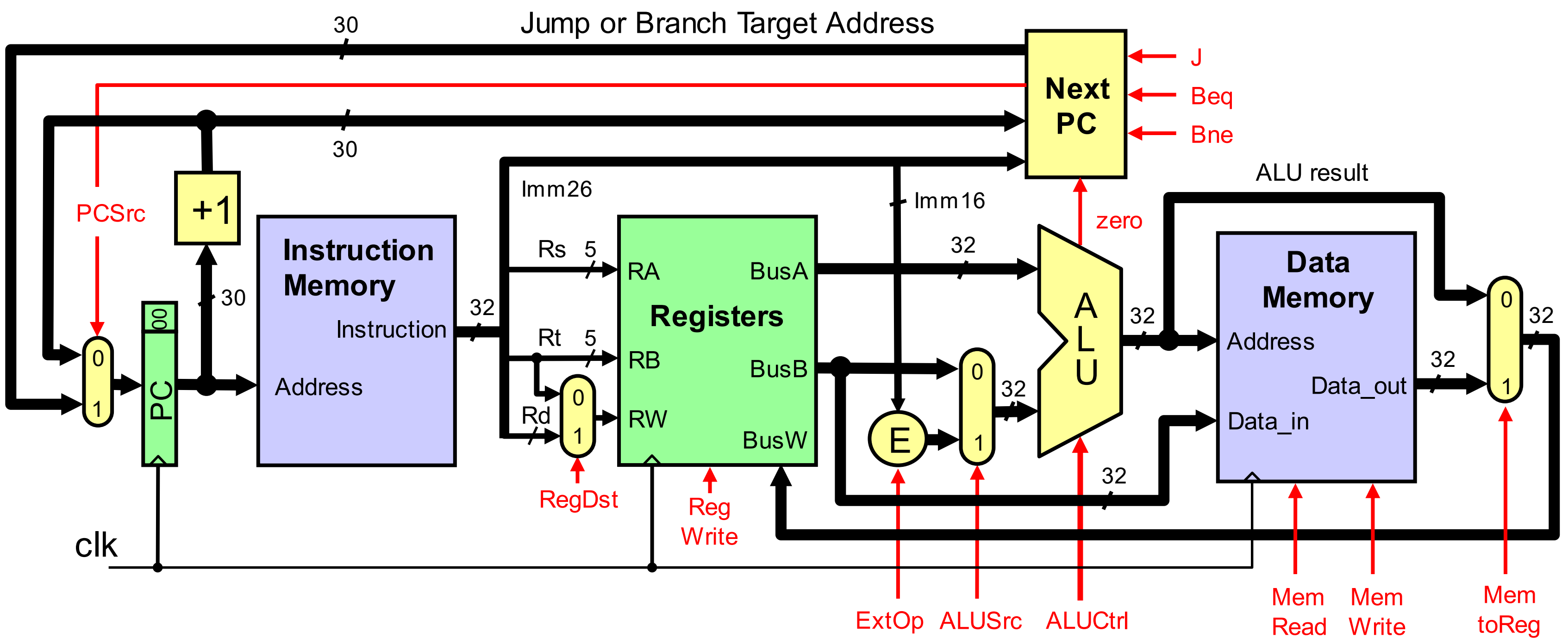
ALUCtrl = ‘ADD’ to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemWrite = ‘1’ to write data memory

MemtoReg = ‘X’ because don’t care what data is put on BusW

Clock edge updates PC and Data Memory

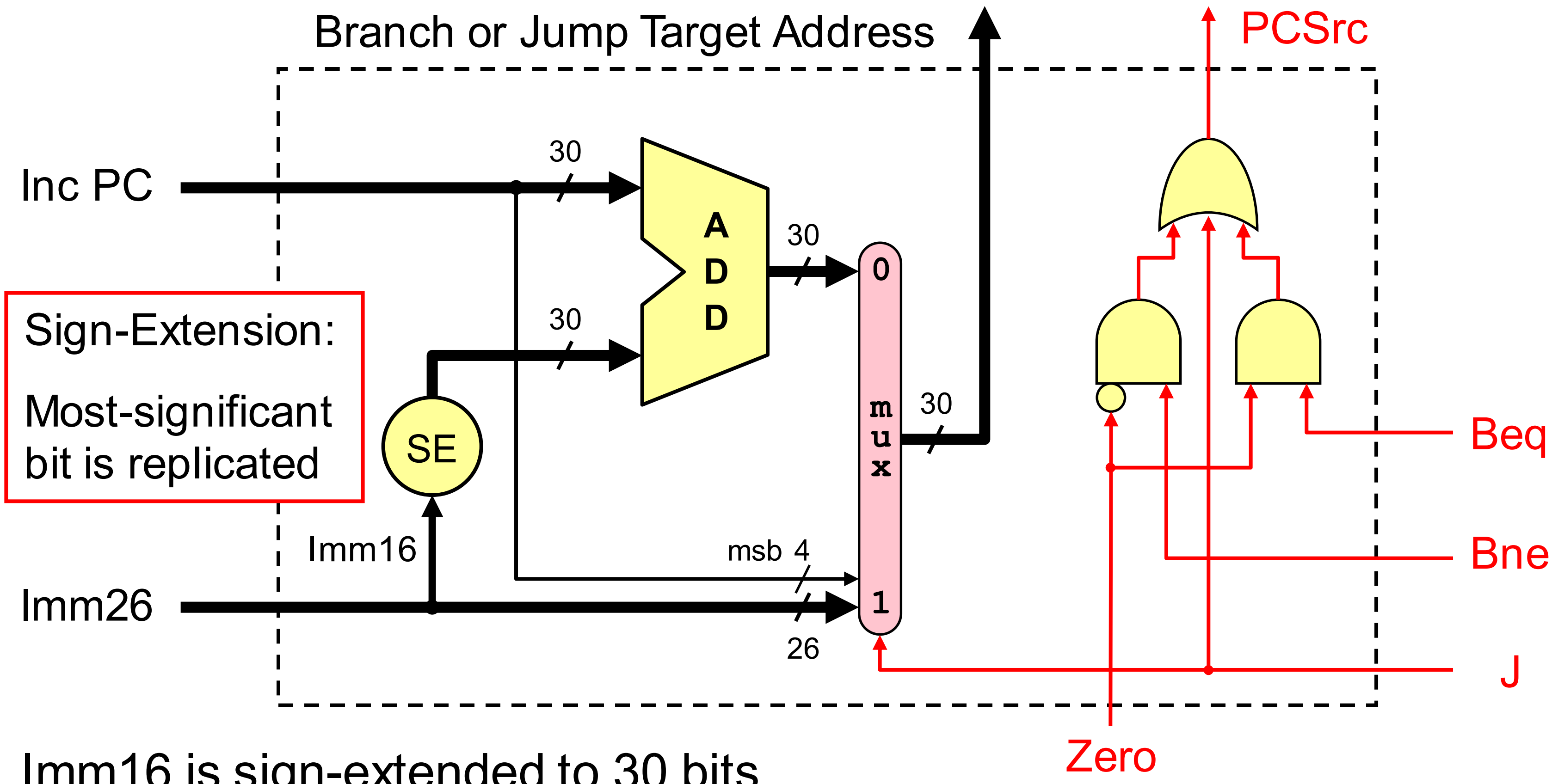
Adding Jump and Branch to Datapath



- Additional Control Signals
  - J, Beq, Bne for jump and branch instructions
  - Zero flag of the ALU is examined
  - PCSrc = 1 for jump & taken branch

Next PC logic computes jump or branch target instruction address

Details of Next PC



Sign-Extension:

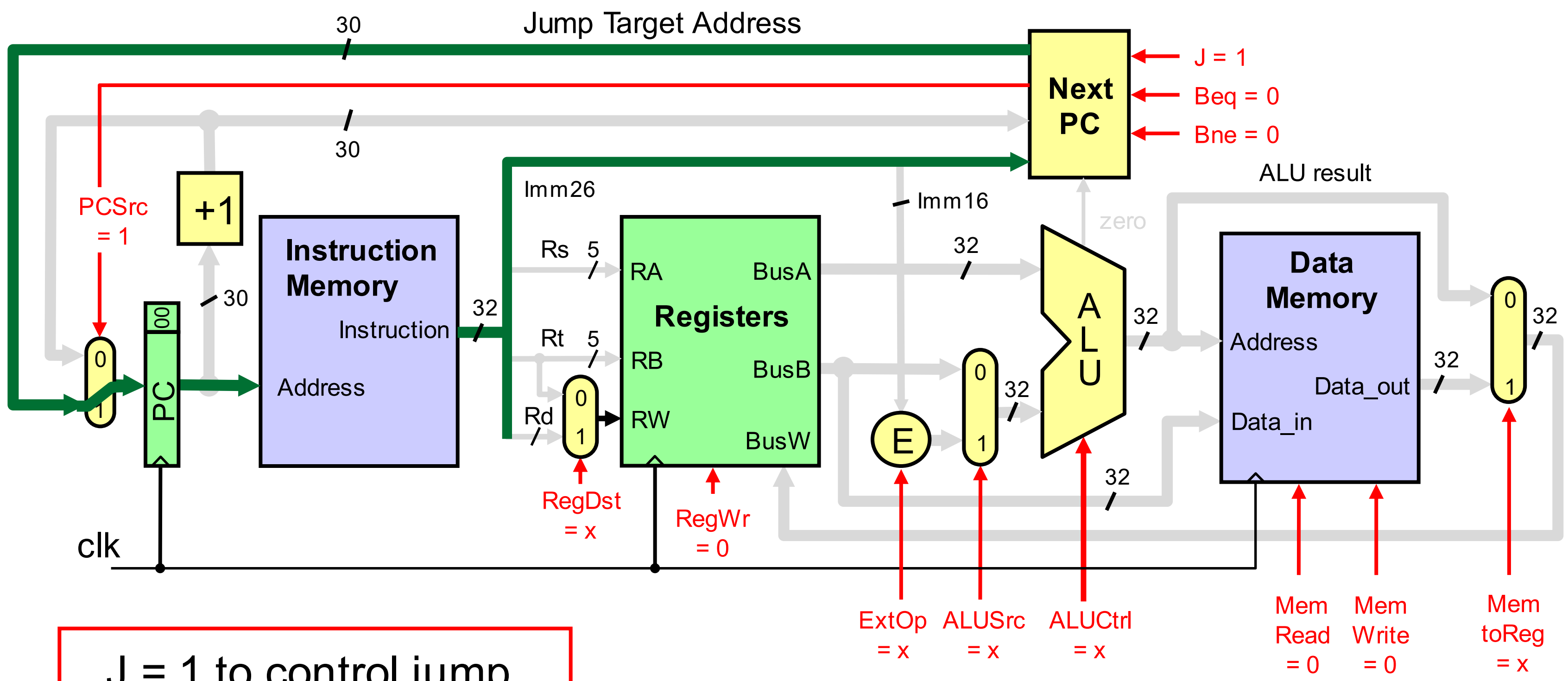
Most-significant bit is replicated

Imm16 is sign-extended to 30 bits

Jump target address: upper 4 bits of PC are concatenated with Imm26

PCSrc = J + (Beq . Zero) + (Bne . $\overline{\text{Zero}}$)

Controlling the Execution of Jump



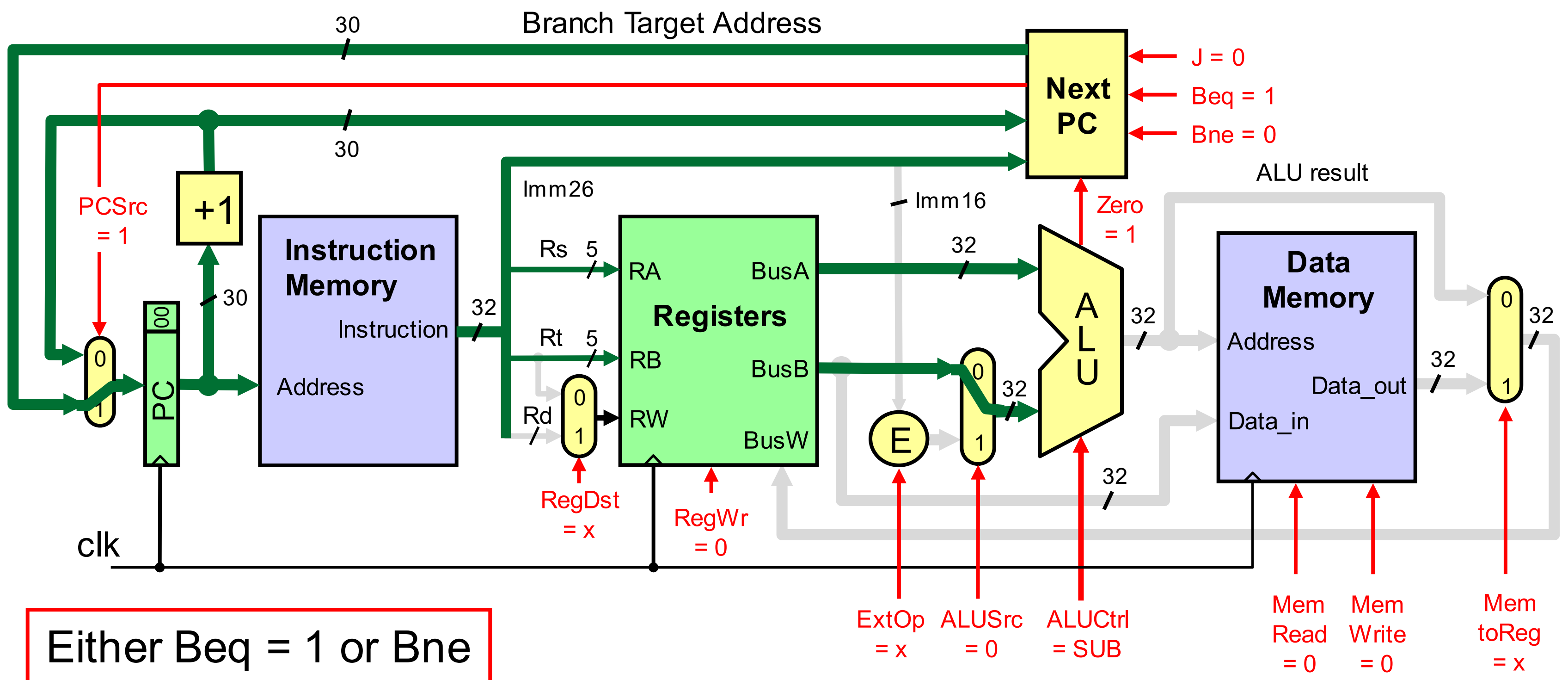
J = 1 to control jump. Next PC outputs Jump Target Address

MemRead, MemWrite, and RegWrite are 0

We don’t care about RegDst, ExtOp, ALUSrc, ALUCtrl, and MemtoReg

Clock edge updates PC register only

Controlling the Execution of Branch



Either Beq = 1 or Bne depending on opcode

ALUSrc = 0 to select value on BusB

ALUCtrl = SUB to generate Zero Flag

Next PC outputs branch target address
PCSrc = 1 if branch is taken

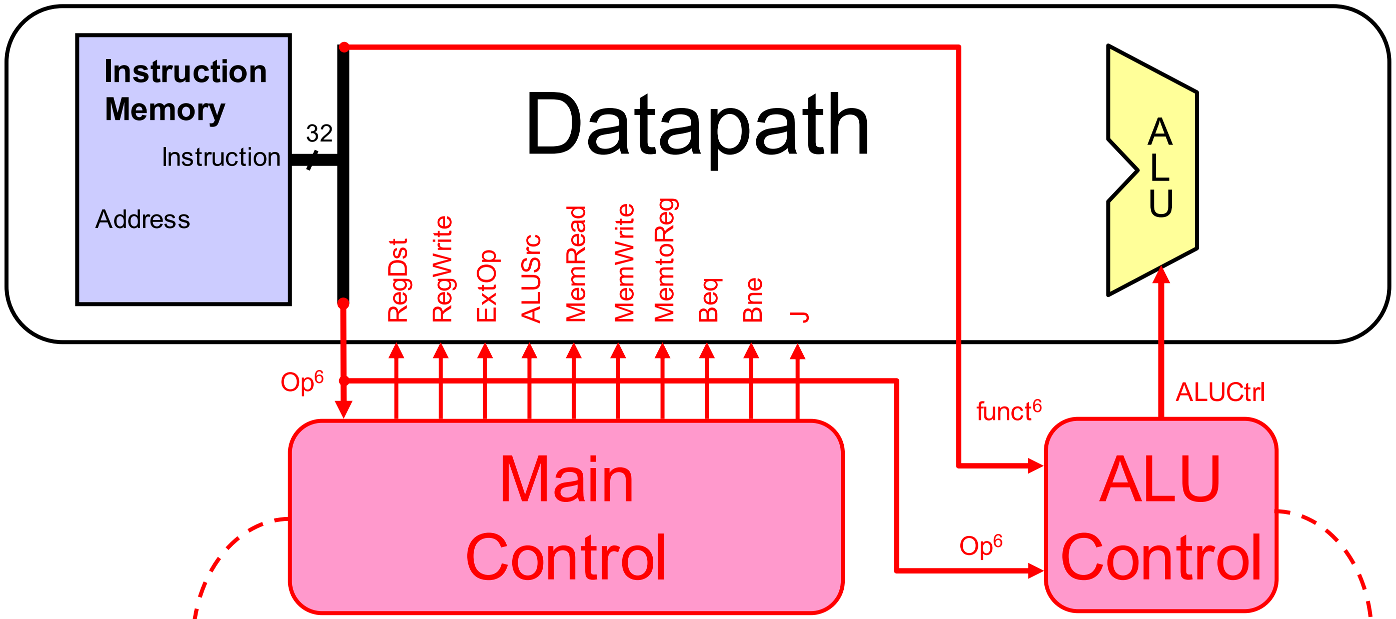
RegWrite, MemRead, and MemWrite are 0

Clock edge updates PC register only

Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

Main Control and ALU Control



Main Control Input:

- ✧ 6-bit opcode field from instruction

Main Control Output:

- ✧ 10 control signals for the Datapath

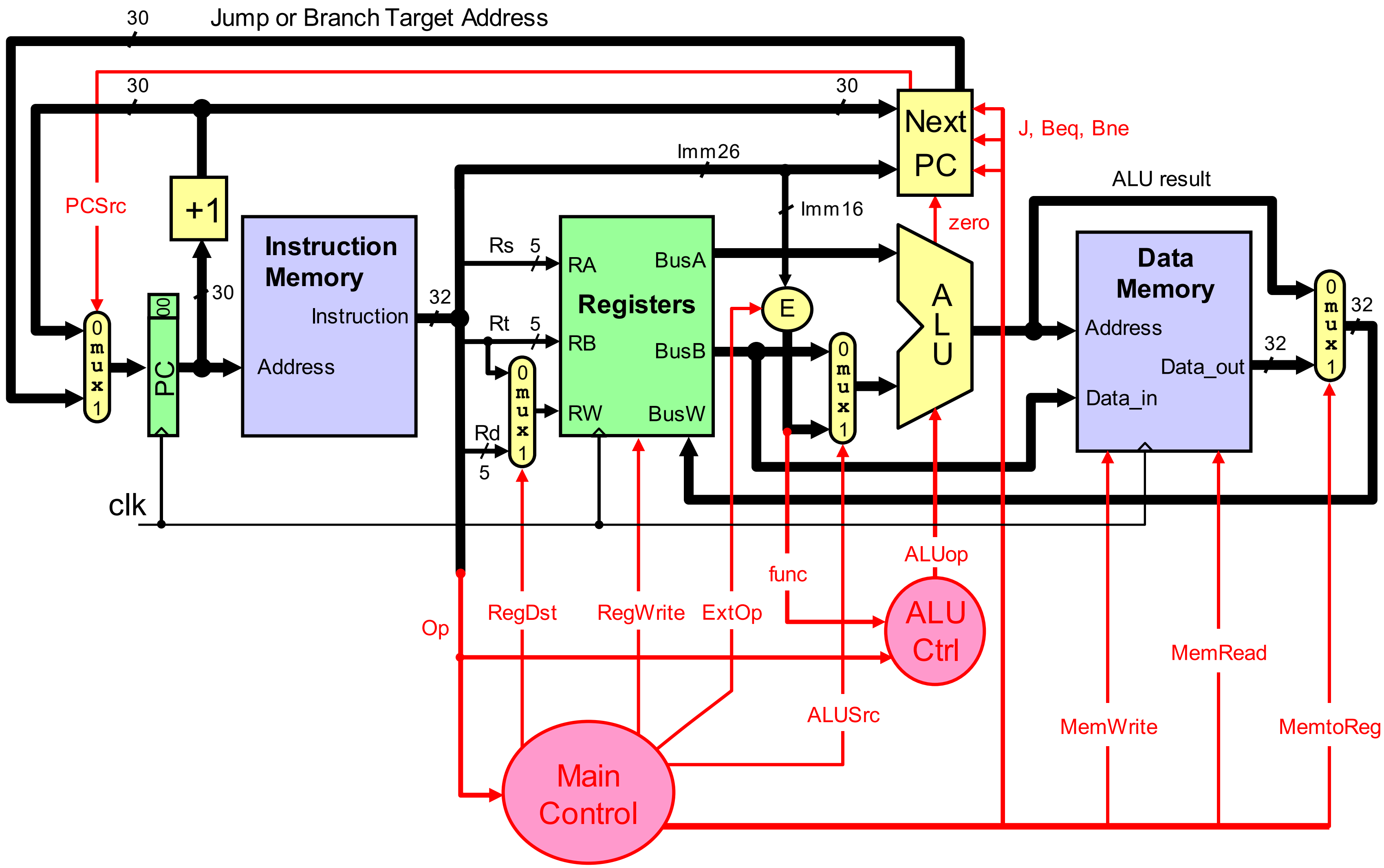
ALU Control Input:

- ✧ 6-bit opcode field from instruction
- ✧ 6-bit function field from instruction

ALU Control Output:

- ✧ ALUCtrl signal for ALU

Single-Cycle Datapath + Control



Main Control Signals

Signal	Effect when ‘0’	Effect when ‘1’
RegDst	Destination register = Rt	Destination register = Rd
RegWrite	None	Destination register is written with the data value on BusW
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
ALUSrc	Second ALU operand comes from the second register file output (BusB)	Second ALU operand comes from the extended 16-bit immediate
MemRead	None	Data memory is read Data_out ← Memory[address]
MemWrite	None	Data memory is written Memory[address] ← Data_in
MemtoReg	BusW = ALU result	BusW = Data_out from Memory
Beq, Bne	PC ← PC + 4	PC ← Branch target address If branch is taken
J	PC ← PC + 4	PC ← Jump target address

Main Control Signal Values

Op	Reg Dst	Reg Write	Ext Op	ALU Src	Beq	Bne	J	Mem Read	Mem Write	Mem toReg
R-type	1 = Rd	1	x	0=BusB	0	0	0	0	0	0
addi	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
slti	0 = Rt	1	1=sign	1=Imm	0	0	0	0	0	0
andi	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
ori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
xori	0 = Rt	1	0=zero	1=Imm	0	0	0	0	0	0
lw	0 = Rt	1	1=sign	1=Imm	0	0	0	1	0	1
sw	x	0	1=sign	1=Imm	0	0	0	0	1	x
beq	x	0	x	0=BusB	1	0	0	0	0	x
bne	x	0	x	0=BusB	0	1	0	0	0	x
j	x	0	x	x	0	0	1	0	0	x

❖ X is a don’t care (can be 0 or 1), used to minimize logic

Logic Equations for Control Signals

$\text{RegDst} = \text{R-type}$

$\text{RegWrite} = \overline{(\text{sw} + \text{beq} + \text{bne} + \text{j})}$

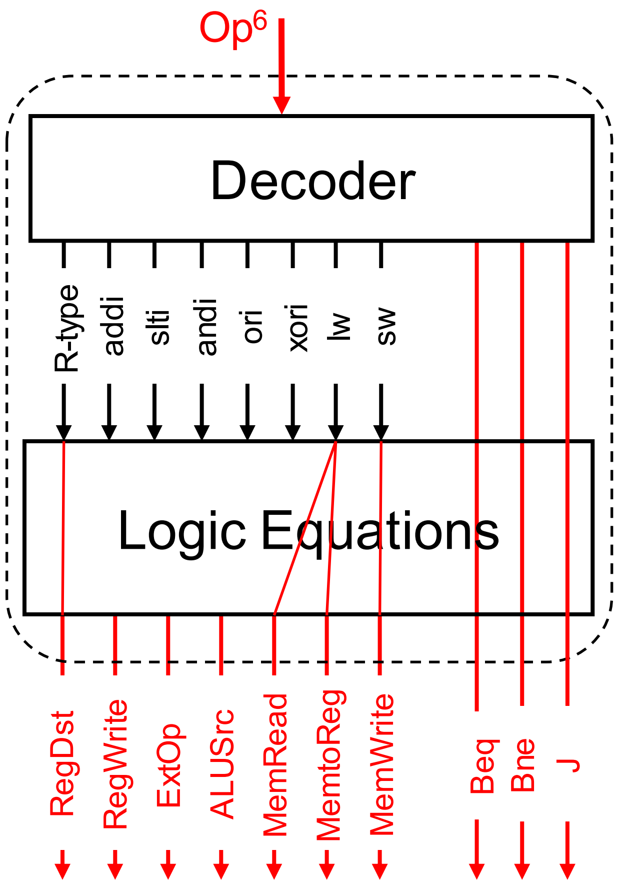
$\text{ExtOp} = \overline{(\text{andi} + \text{ori} + \text{xori})}$

$\text{ALUSrc} = \overline{(\text{R-type} + \text{beq} + \text{bne})}$

$\text{MemRead} = \text{lw}$

$\text{MemtoReg} = \text{lw}$

$\text{MemWrite} = \text{sw}$



ALU Control Truth Table

Input		Output	4-bit
Op⁶	funct⁶	ALUCtrl	Encoding
R-type	add	ADD	0000
R-type	sub	SUB	0010
R-type	and	AND	0100
R-type	or	OR	0101
R-type	xor	XOR	0110
R-type	slt	SLT	1010
addi	x	ADD	0000
slti	x	SLT	1010
andi	x	AND	0100
ori	x	OR	0101
xori	x	XOR	0110
lw	x	ADD	0000
sw	x	ADD	0000
beq	x	SUB	0010
bne	x	SUB	0010
j	x	x	x

The 4-bit ALUCtrl is encoded according to the ALU implementation

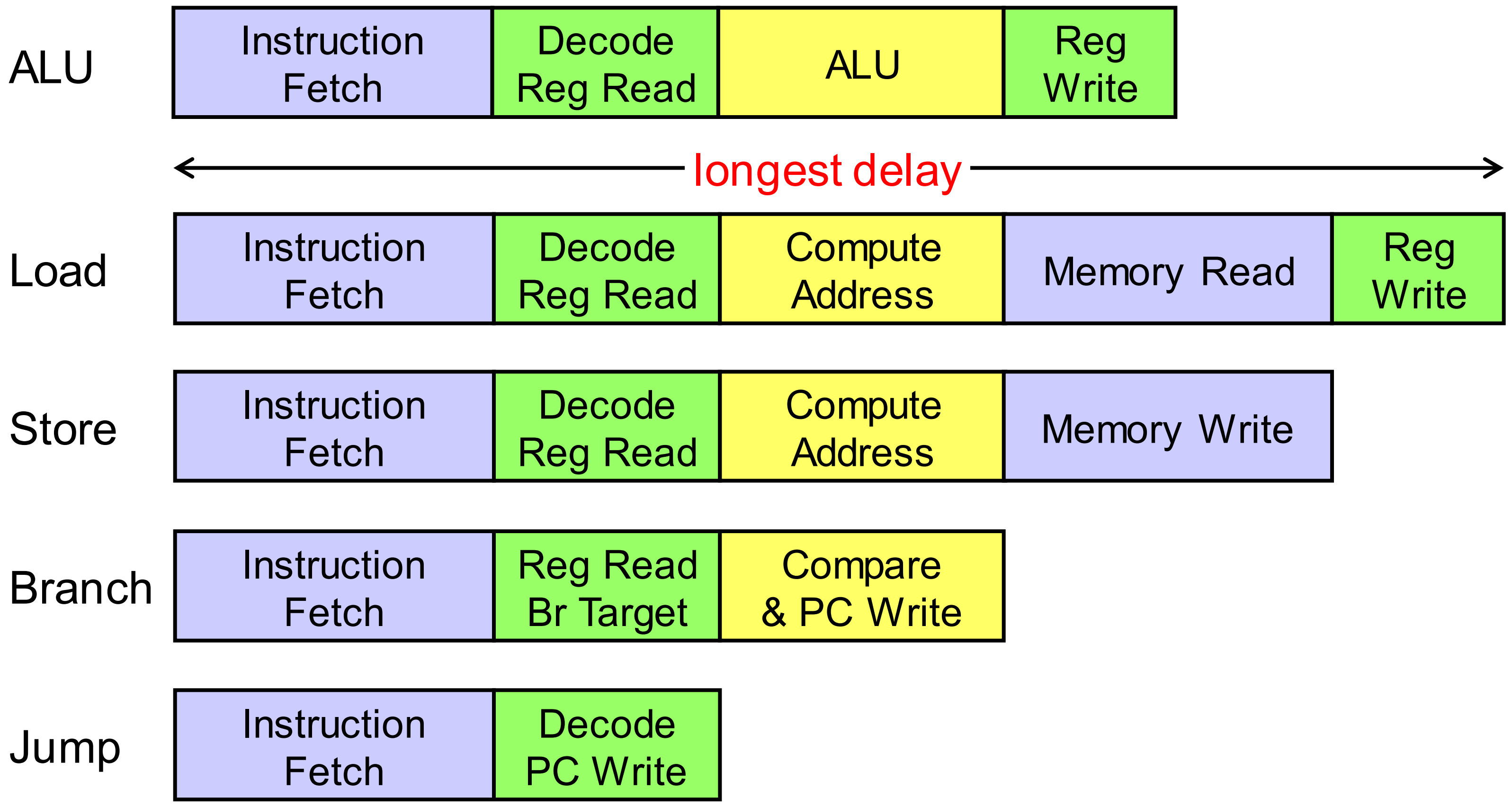
Other ALU control encodings are also possible. The idea is to choose a binary encoding that will simplify the logic

Next . . .

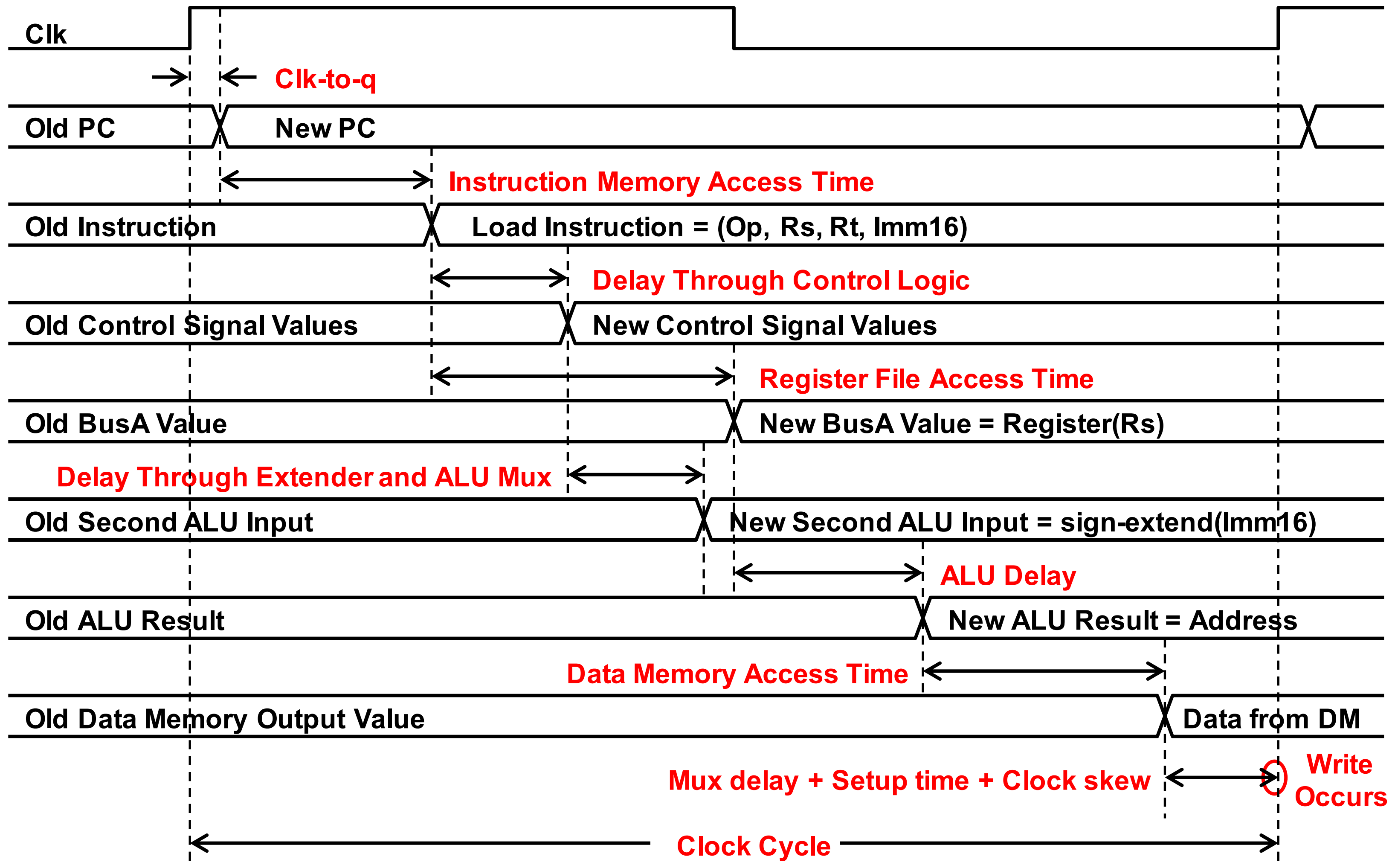
- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

Drawbacks of Single Cycle Processor

- Long cycle time
  - All instructions take as much time as the slowest instruction



Timing of a Load Instruction


Clk
Clk-to-q
Old PC
New PC
Instruction Memory Access Time
Old Instruction
Load Instruction = (Op, Rs, Rt, Imm16)
Delay Through Control Logic
Old Control Signal Values
New Control Signal Values
Register File Access Time
Old BusA Value
New BusA Value = Register(Rs)
Delay Through Extender and ALU Mux
Old Second ALU Input
New Second ALU Input = sign-extend(Imm16)
ALU Delay
Old ALU Result
New ALU Result = Address
Data Memory Access Time
Old Data Memory Output Value
Data from DM
Mux delay + Setup time + Clock skew
Write Occurs
Clock Cycle

Worst Case Timing – Cont'd

- Long cycle time: long enough for Slowest instruction

  PC Clk-to-Q delay

  + Instruction Memory Access Time

  + Maximum of (

    Register File Access Time,

    Delay through control logic + extender + ALU mux)

  + ALU to Perform a 32-bit Add

  + Data Memory Access Time

  + Delay through MemtoReg Mux

  + Setup Time for Register File Write + Clock Skew

- Cycle time is longer than needed for other instructions
  - Therefore, single cycle processor design is not used in practice

Alternative: Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode, register read, target address for jump/branch
  - Execution, memory address calculation, or branch outcome
  - Memory access or ALU instruction completion
  - Load instruction completion
- One clock cycle per step (clock cycle is reduced)
  - First 2 steps are the same for all instructions

Instruction	# cycles	Instruction	# cycles
ALU & Store	4	Branch	3
Load	5	Jump	2

Performance Example

- Assume the following operation times for components:
  - Instruction and data memories: 200 ps
  - ALU and adders: 180 ps
  - Decode and Register file access (read or write): 150 ps
  - Ignore the delays in PC, mux, extender, and wires
- Which of the following would be faster and by how much?
  - Single-cycle implementation for all instructions
  - Multicycle implementation optimized for every class of instructions
- Assume the following instruction mix:
  - 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps

Solution

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180 ← Compare and write PC			530 ps
Jump	200	150 ← Decode and write PC				350 ps

- For fixed single-cycle implementation:
  - Clock cycle = 880 ps determined by longest delay (load instruction)
- For multi-cycle implementation:
  - Clock cycle = max (200, 150, 180) = 200 ps (maximum delay at any step)
  - Average CPI = 0.4×4 + 0.2×5 + 0.1×4+ 0.2×3 + 0.1×2 = 3.8
- Speedup = 880 ps / (3.8 × 200 ps) = 880 / 760 = 1.16

Summary

- 5 steps to design a processor
  - Analyze instruction set => datapath requirements
  - Select datapath components & establish clocking methodology
  - Assemble datapath meeting the requirements
  - Analyze implementation of each instruction to determine control signals
  - Assemble the control logic
- MIPS makes Control easier
  - Instructions are of same size
  - Source registers always in same place
  - Immediates are of same size and same location
  - Operations are always on registers/immediates
- Single cycle datapath => CPI=1, but Long Clock Cycle

I-Mem	Add	Mux	ALU	Registers	D-Mem	Sign-extend	Shift-left 2	Control
400ps	100ps	35ps	120ps	230ps	350ps	30ps	3ps	90ps

(a) [4pt] What is the minimum clock cycle time of this processor if only R-type instructions are supported? Show your work.

Solution:

If only R-type instructions are supported, the critical path would be

I-Mem - Registers - Mux - ALU - Mux - Registers .

So the clock cycle time is 400+230+35+120+35+230 ps = 1050 ps

(b) [6pt] Determine the critical (longest) path for the single-cycle MIPS processor. What is the latency of the critical path? Please show all work.

Solution:

The longest path is I-Mem - Registers - Mux - ALU - D-Mem - Mux - Registers .
The `lw` instruction goes through all these components.

So the latency is 400+230+35+120+350+35 + 230 ps = 1400 ps

(c) In the following table, parts of control signals are given for each instruction. Please select all possible instructions from the given instructions set for these five instructions. ("x" means don't care.)

Instruction set :

`{add, lw, ori, slt, beq, sw}`

Instr.	RegDst	ALUSrc	MemtoReg	RegWrite	Branch
A	1	0	0	1	0
B	0	1	0	1	0
C	0	1	1	1	0
D	x	0	x	0	1

Possible instructions for A:

`add, slt`

Possible instructions for B:

`ori`

Possible instructions for C:

`lw`

Possible instructions for D:

`beq`

MIPS Komut Setindeki tüm load ve store komutlarının formatını değiştirmek istediğimizi düşünün.

load ve store komutları R türü komutlar haline dönüştürülmek isteniyor. Offset (0) olarak alınacak yani 'immediate' operand gerekmeyecektir.
Yani bellek adresi hesaplamasında ALU ya ihtiyaç duyulmaması öngörülmüştür.

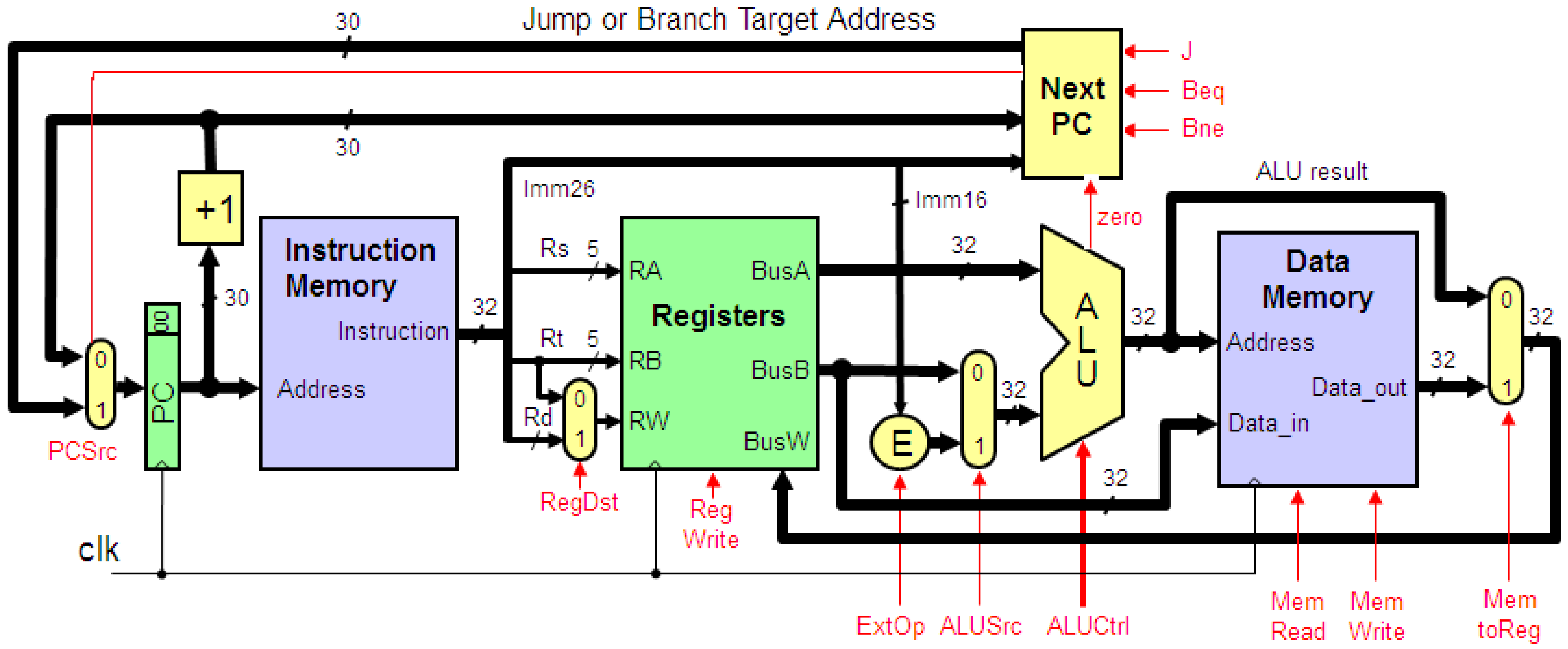
Yeni load ve store komutları formatı :

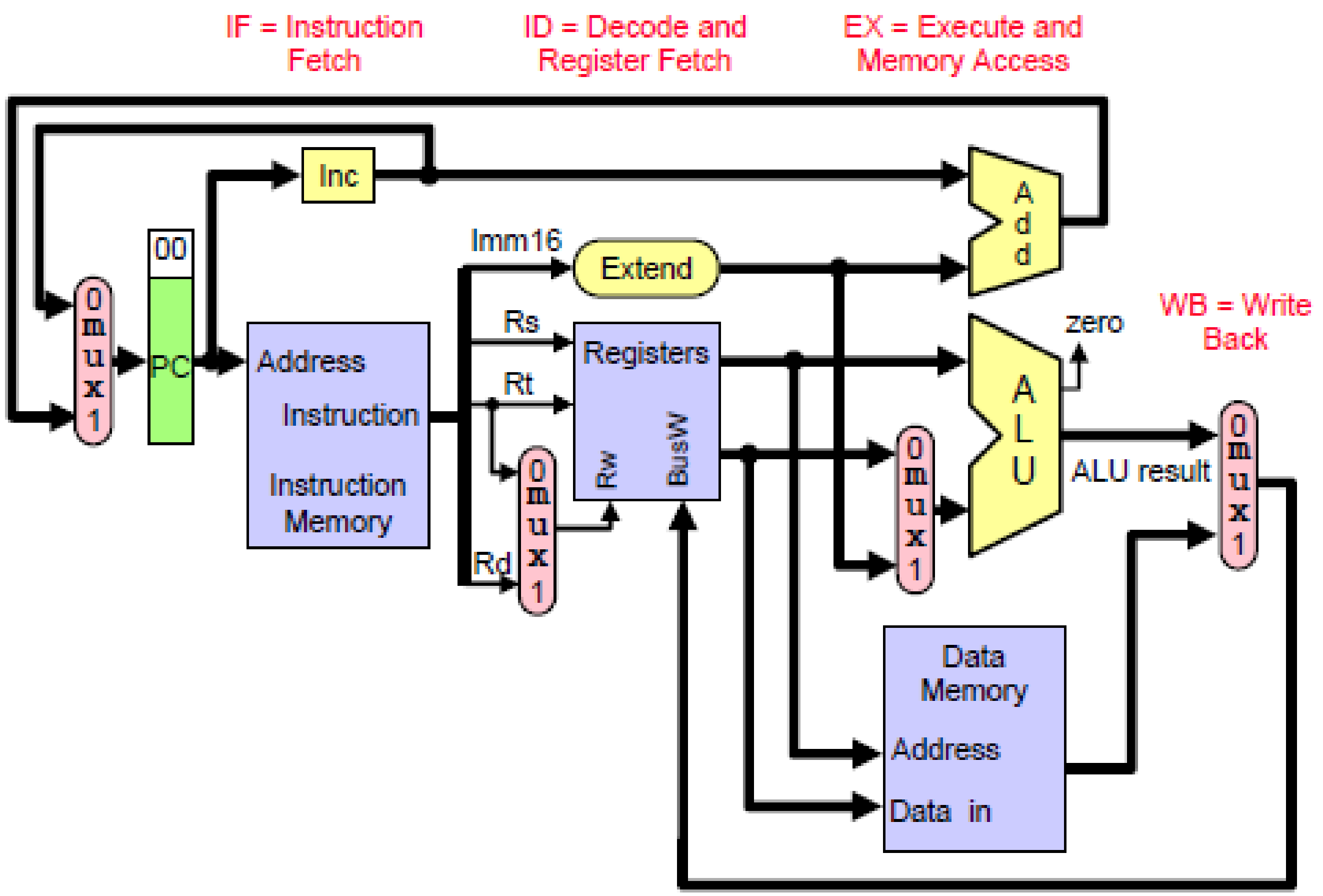
LW Rt, (Rs)
SW Rt, (Rs)

Burada Rs bellek adresini içeren register'i göstermektedir.

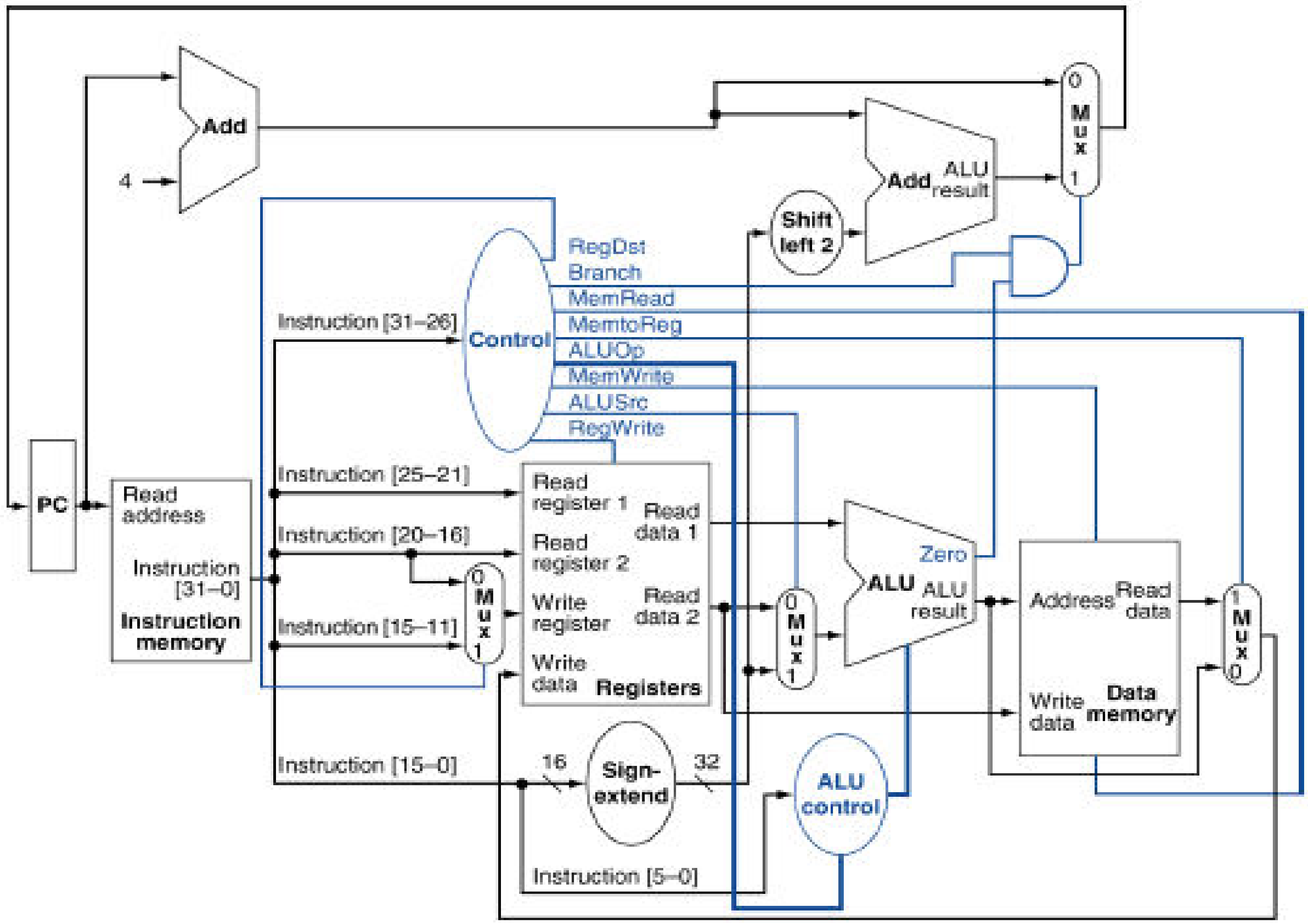
.

Single cycle datapath üzerinde gerekli donanımsal değişimi gösteriniz..?




IF = Instruction Fetch
ID = Decode and Register Fetch
EX = Execute and Memory Access
WB = Write Back
Inc
00
PC
0 mux 1
Address
Instruction
Instruction Memory
Imm16
Extend
Rs
Rt
Rd
Registers
Rw
BusW
Add
zero
ALU
ALU result
Data Memory
Address
Data in

The block diagram for a single cycle MIPS processor is shown in the figure below. And the latencies of major blocks are listed in the table.


Add
4
Add
ALU result
Shift left 2
M u x
0
1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
32
Sign-extend
ALU control
Instruction [5–0]