Single Cycle Processor Design

Presentation Outline

- ❖ **Designing a Processor: Step-by-Step**
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ Main, ALU, and PC Control

CPU Organization (Design)

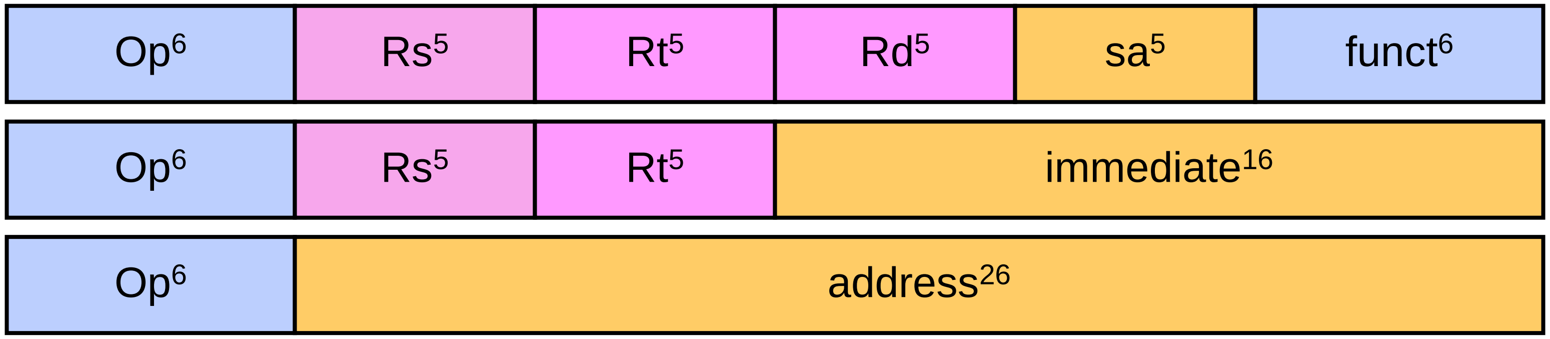
- Datapath Design:
  - Capabilities & performance characteristics of principal Functional Units (FUs) needed by ISA instructions
  - (e.g., Registers, ALU, Shifters, Logic Units, ...)
  - Ways in which these components are interconnected (buses connections, multiplexors, etc.).
  - How information flows between components.
- Control Unit Design:
  - Logic and means by which such information flow is controlled.
  - Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).
- Hardware description with a suitable language, possibly using Register Transfer Notation (RTN).

Designing a Processor: Step-by-Step

1. 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
2. Select datapath components and clocking methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction
   - ✧ Determine the setting of control signals for register transfer
5. 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 constant or PC-relative offset
- ✧ address^{26}: 26-bit target address of the jump instruction

MIPS Five Addressing Modes

1 Register Addressing:

Where the operand is a register (R-Type)

2 Immediate Addressing:

Where the operand is a constant in the instruction (I-Type, ALU)

3 Base or Displacement Addressing:

Where the operand is at the memory location whose address is the sum of a register and a constant in the instruction (I-Type, load/store)

4 PC-Relative Addressing:

Where the address is the sum of the PC and the 16-address field in the instruction shifted left 2 bits. (I-Type, branches)

5 Pseudodirect Addressing:

Where the jump address is the 26-bit jump target from the instruction shifted left 2 bits concatenated with the 4 upper bits of the PC (J-Type)

MIPS Addressing Modes/Instruction Formats

The image part with relationship ID rId3 was not found in the file.

MIPS Subset of Instructions

- Only a subset of the MIPS instructions is 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, imm^{16}	add immediate	0x08	rs^5	rt^5	imm^{16}		
slti rt, rs, imm^{16}	slt immediate	0x0a	rs^5	rt^5	imm^{16}		
andi rt, rs, imm^{16}	and immediate	0x0c	rs^5	rt^5	imm^{16}		
ori rt, rs, imm^{16}	or immediate	0x0d	rs^5	rt^5	imm^{16}		
xori rt, imm^{16}	xor immediate	0x0e	rs^5	rt^5	imm^{16}		
lw rt, imm^{16}(rs)	load word	0x23	rs^5	rt^5	imm^{16}		
sw rt, imm^{16}(rs)	store word	0x2b	rs^5	rt^5	imm^{16}		
beq rs, rt, offset^{16}	branch if equal	0x04	rs^5	rt^5	offset^{16}		
bne rs, rt, offset^{16}	branch not equal	0x05	rs^5	rt^5	offset^{16}		
j address^{26}	jump	0x02	address^{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(imm^{16});	PC ← PC + 4
LW	Reg(rt) ← MEM[Reg(rs) + sign_ext(imm^{16})];	PC ← PC + 4
SW	MEM[Reg(rs) + sign_ext(imm^{16})] ← Reg(rt);	PC ← PC + 4
BEQ	if (Reg(rs) == Reg(rt)) PC ← PC + 4 + 4 × sign_ext(offset^{16}) else PC ← PC + 4	

Instruction Fetch/Execute

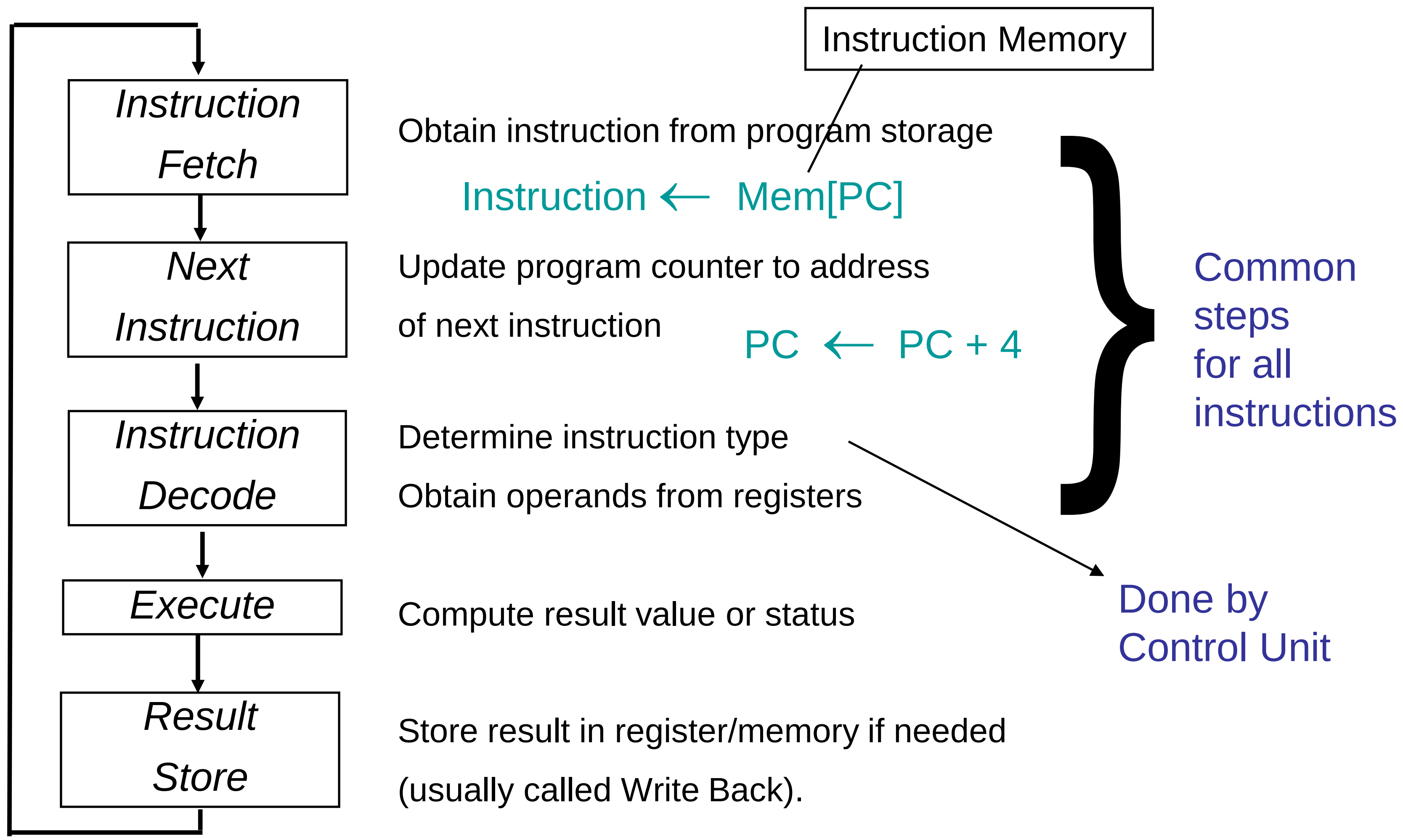
❖ R-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(rs), data2 ← Reg(rt)
	Execute operation:	ALU_result ← func(data1, data2)
	Write ALU result:	Reg(rd) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ I-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(rs), data2 ← Extend(imm^{16})
	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(offset^{16})
		else PC ← PC + 4

Instruction Fetch/Execute - cont'd

❖ LW	Fetch instruction:	Instruction ← MEM[PC]
	Fetch base register:	base ← Reg(rs)
	Calculate address:	address ← base + sign_extend(imm^{16})
	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(imm^{16})
	Write memory:	MEM[address] ← data
	Next PC address:	PC ← PC + 4
❖ Jump	Fetch instruction:	Instruction ← MEM[PC]
	Target PC address:	target ← PC[31:28] || address^{26} || '00'
	Jump:	PC ← target

concatenation

Basic MIPS Instruction Processing Steps



Instruction Fetch: Obtain instruction from program storage

Instruction ← Mem[PC]

Next Instruction: Update program counter to address of next instruction

PC ← PC + 4

Instruction Decode: Determine instruction type

Obtain operands from registers

Execute: Compute result value or status

Result Store: Store result in register/memory if needed (usually called Write Back).

Overview of MIPS Instruction Micro-operations

- All instructions go through these common steps:
  - Send program counter to instruction memory and fetch the instruction.
    (fetch) Instruction $\leftarrow$ Mem[PC]
  - Update the program counter to point to next instruction PC $\leftarrow$ PC + 4
  - Read one or two registers, using instruction fields. (*decode*)
    - Load reads one register only.
- Additional instruction execution actions (*execution*) depend on the instruction in question, but similarities exist:
  - All instruction classes use the ALU after reading the registers:
    - Memory reference instructions use it for address calculation.
    - Arithmetic and logic instructions (R-Type), use it for the specified operation.
    - Branches use it for comparison.
- Additional execution steps where instruction classes differ:
  - Memory reference instructions: Access memory for a load or store.
  - Arithmetic and logic instructions: Write ALU result back in register.
  - Branch instructions: Change next instruction address based on comparison.

Requirements of the Instruction Set

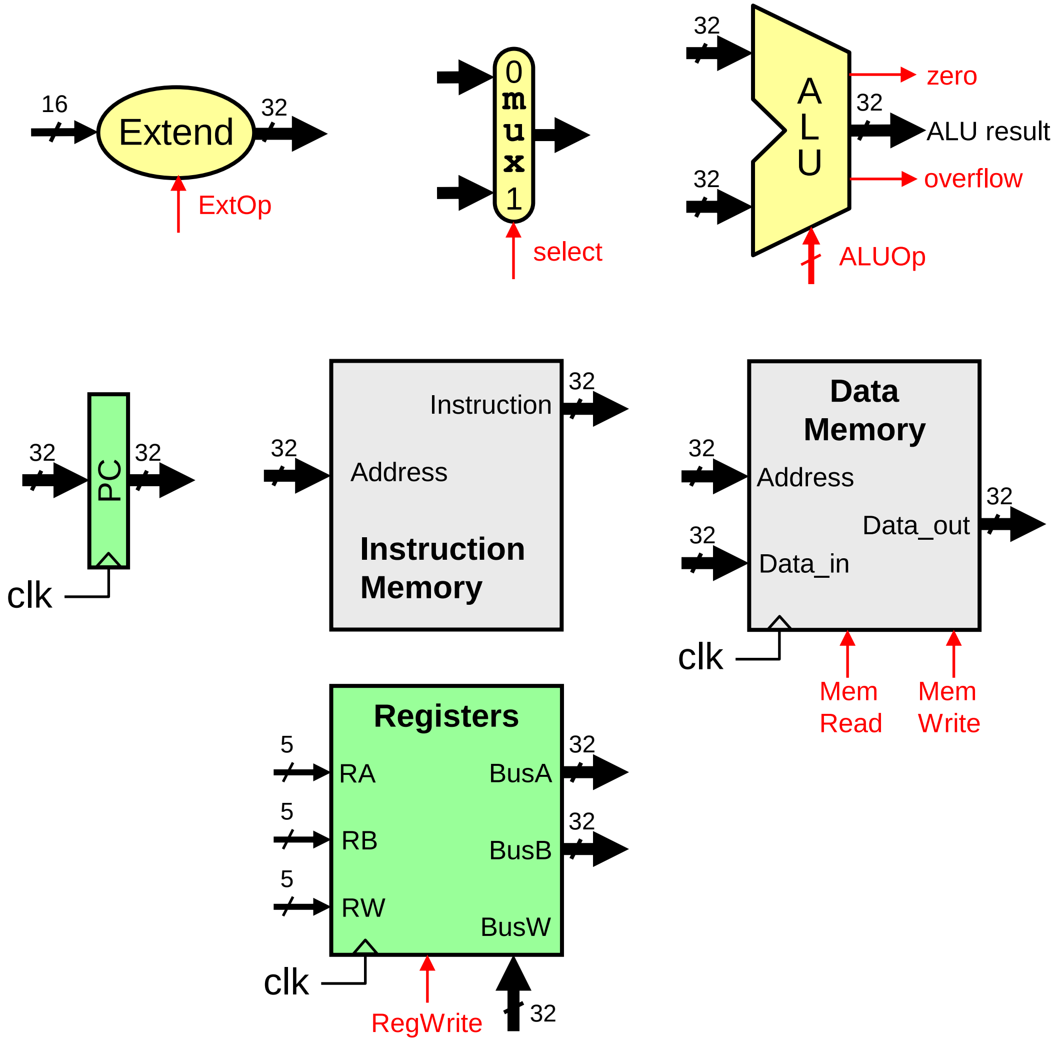
- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 31 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and 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
- ❖ Main, ALU, and PC Control

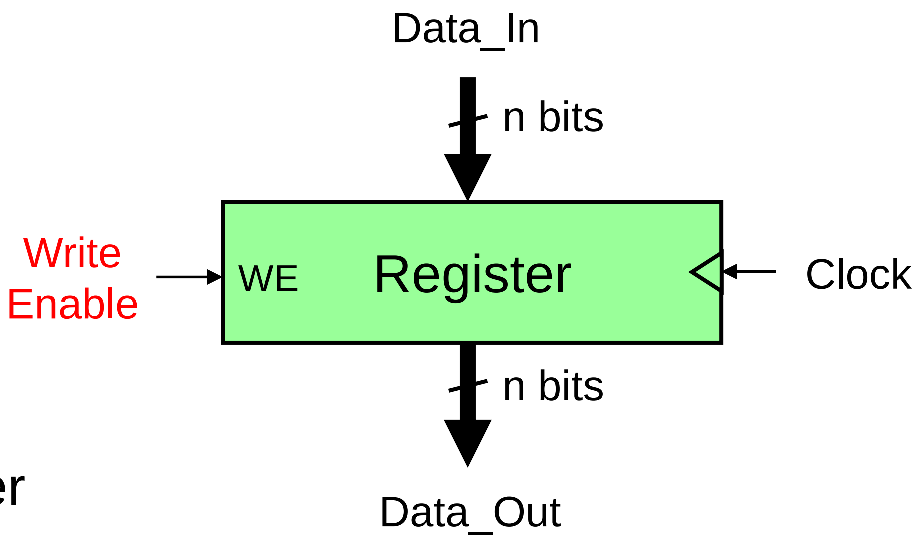
Components of the Datapath

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



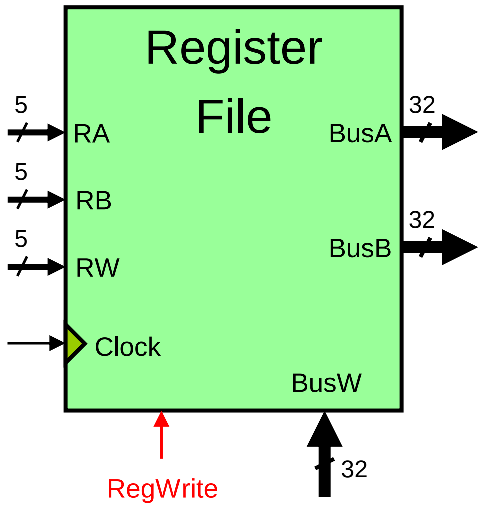
Register Element

- ❖ 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



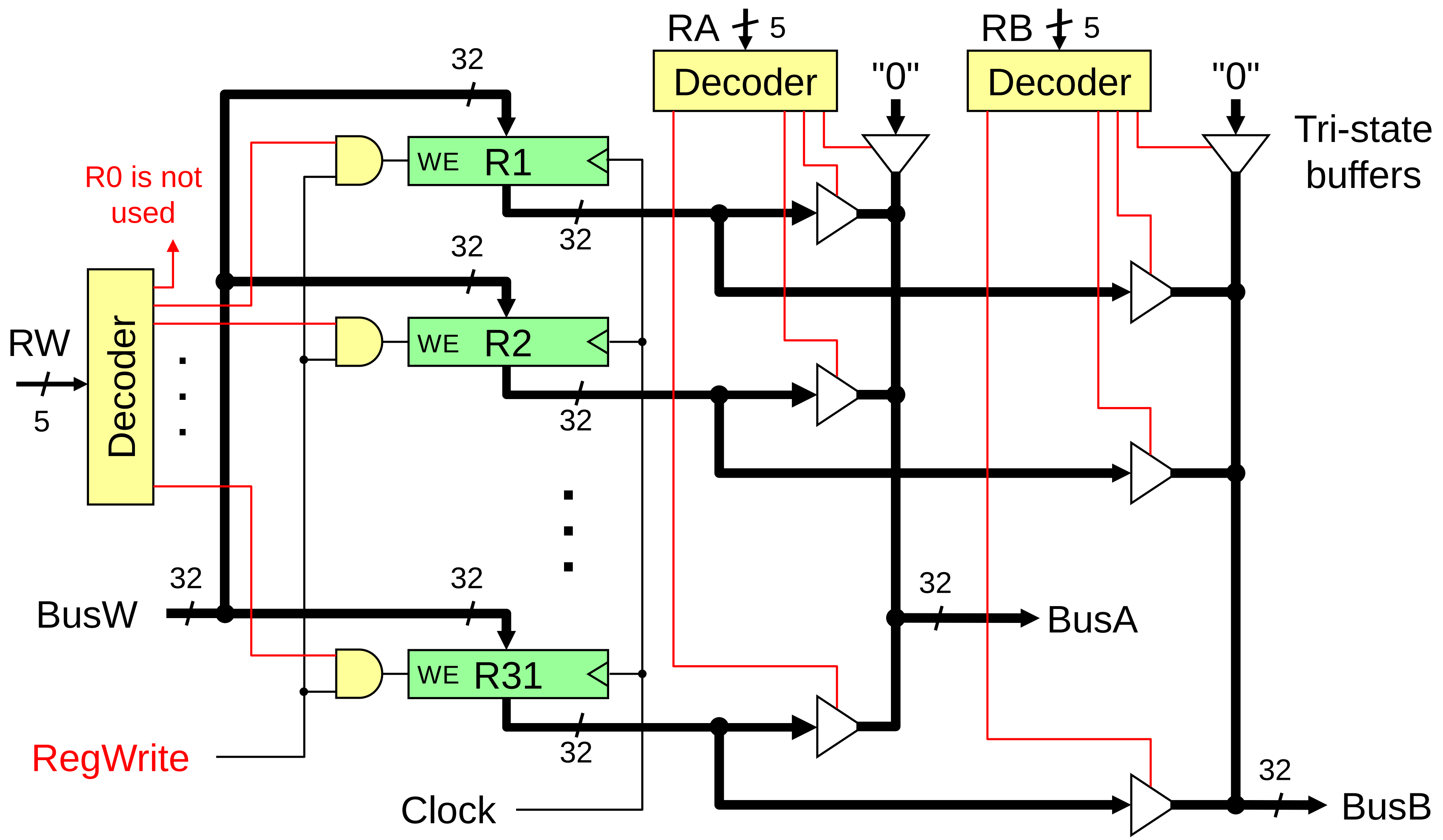
MIPS Register File

- Register File consists of 31 × 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

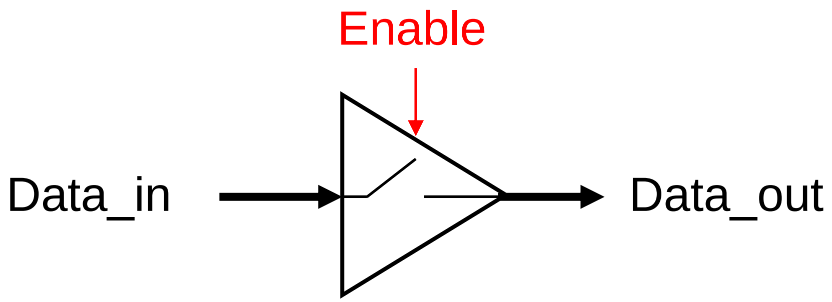
Details of the Register File



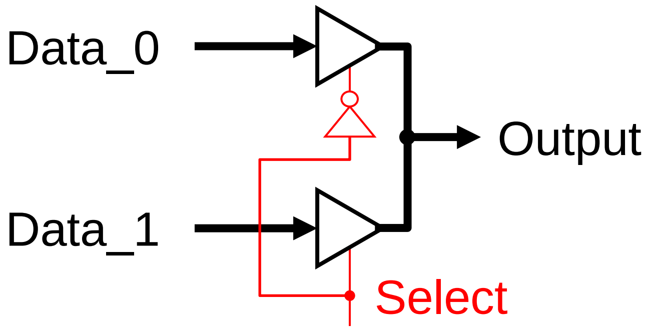
Tri-State Buffers

- 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

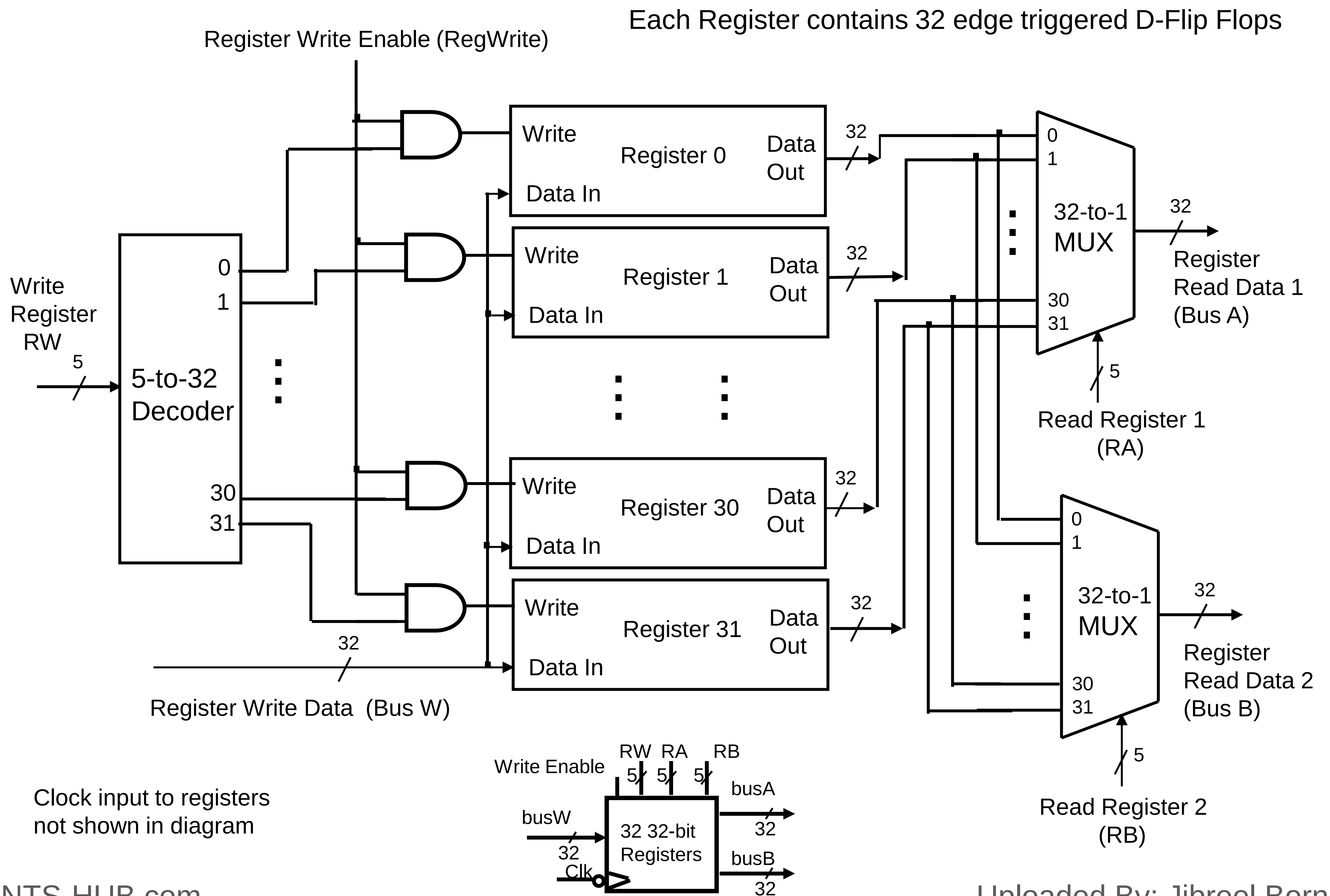
    else Data_out = High Impedance state (output is disconnected)



- Tri-state buffers can be used to build multiplexors

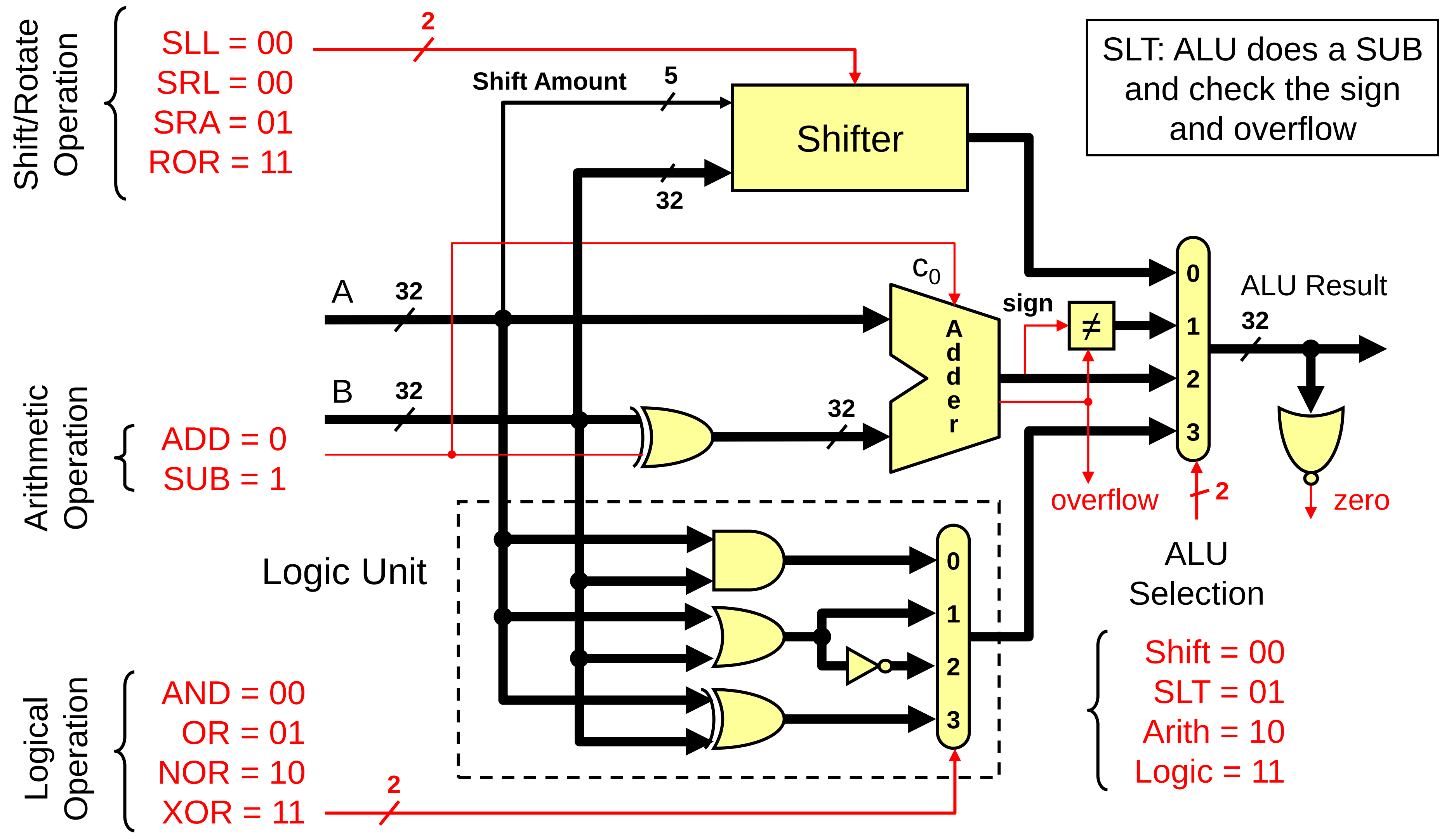


Another Implementation



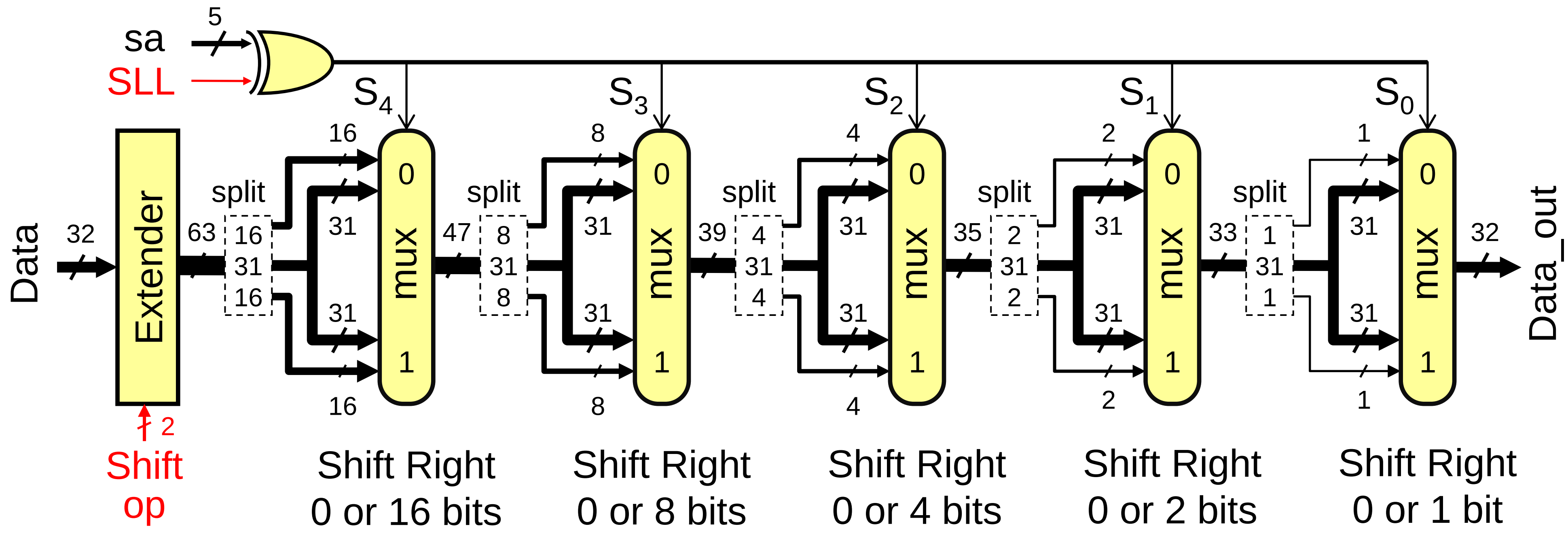
Clock input to registers not shown in diagram

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

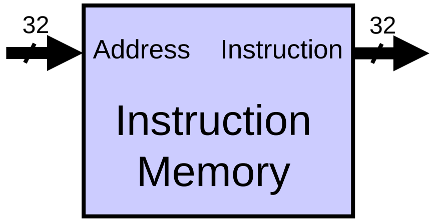
- Input data is extended from 32 to 63 bits as follows:
  - If shift op = SRL then ext_data[62:0] = 0^{31} || data[31:0]
  - If shift op = SRA then ext_data[62:0] = $\text{data}[31]^{31}$ || data[31:0]
  - If shift op = ROR then ext_data[62:0] = data[30:0] || data[31:0]
  - If shift op = SLL then ext_data[62:0] = 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)

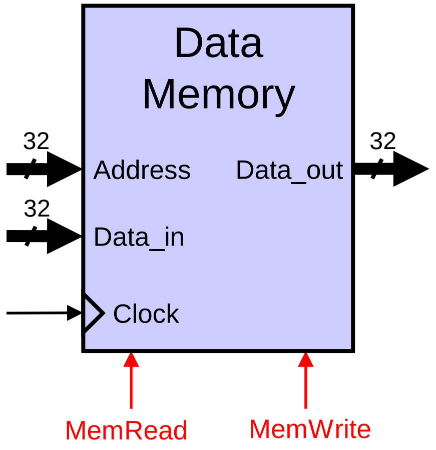
Implementing Shift Left Logical

- 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

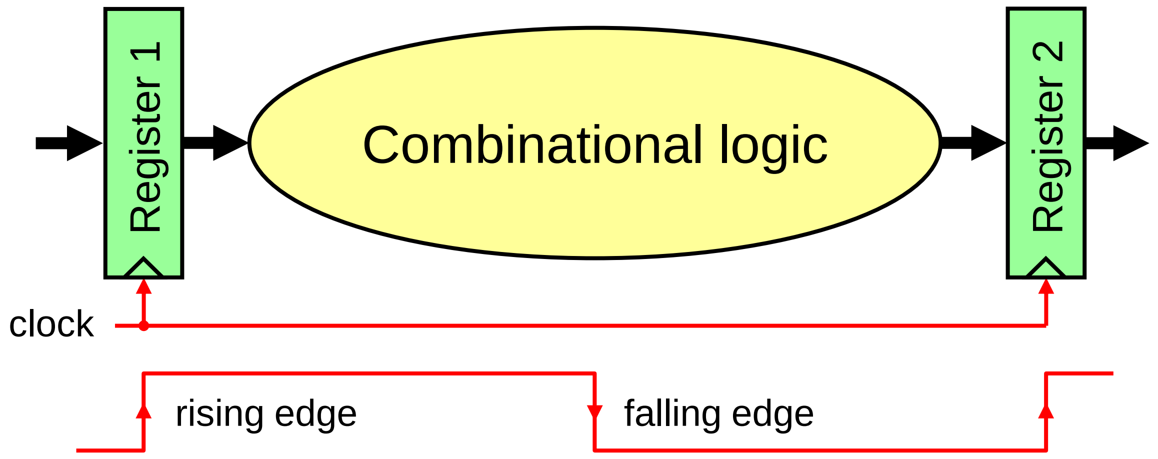
- Instruction memory needs only provide read access
  - Because datapath does not write instructions
  - Behaves as combinational logic for read
  - Address selects Instruction after access time
- Data Memory is used for load and store
  - MemRead: enables output on Data_out
    - Address selects the 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

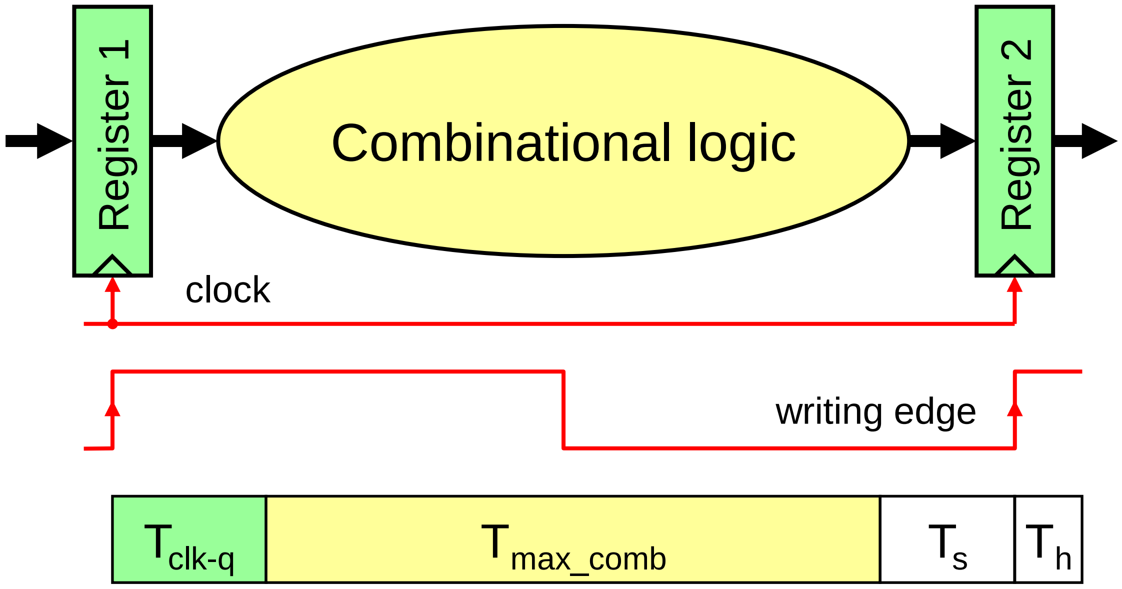
- Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- To ensure correctness, a clocking methodology defines when data can be written and read



- 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

Clocking Methodologies

- State element design choices
  - Latches:
    - Output responds to input changes only when the clock is asserted
    - Assert means "logically true“(could mean electrically low)
  - Flip-flop:
    - State changes only on a clock edge
    - (edge-triggered methodology)

State Elements

- Level sensitive D latch

The image part with relationship ID rId2 was not found in the file.

- D flip flop

The image part with relationship ID rId2 was not found in the file.

Overview: Processor Implementation Styles

- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: each instruction uses only as many cycles as it needs
- Pipelined
  - execute each instruction in multiple steps
  - perform 1 step / instruction in each clock cycle
  - process multiple instructions in parallel – assembly line

Single Cycle, Multiple Cycle, vs. Pipeline

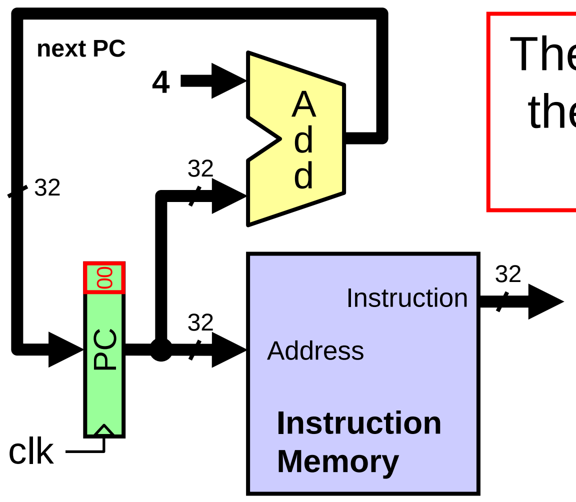
The image part with

Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ **Assembling an Adequate Datapath**
- ❖ **Controlling the Execution of Instructions**
- ❖ Main, ALU, and PC Control

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

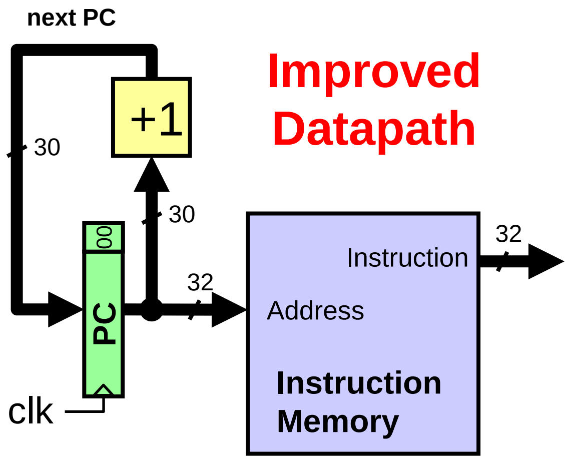


The least significant 2 bits of the PC are '00' since PC is a multiple of 4

Datapath does not handle branch or jump instructions

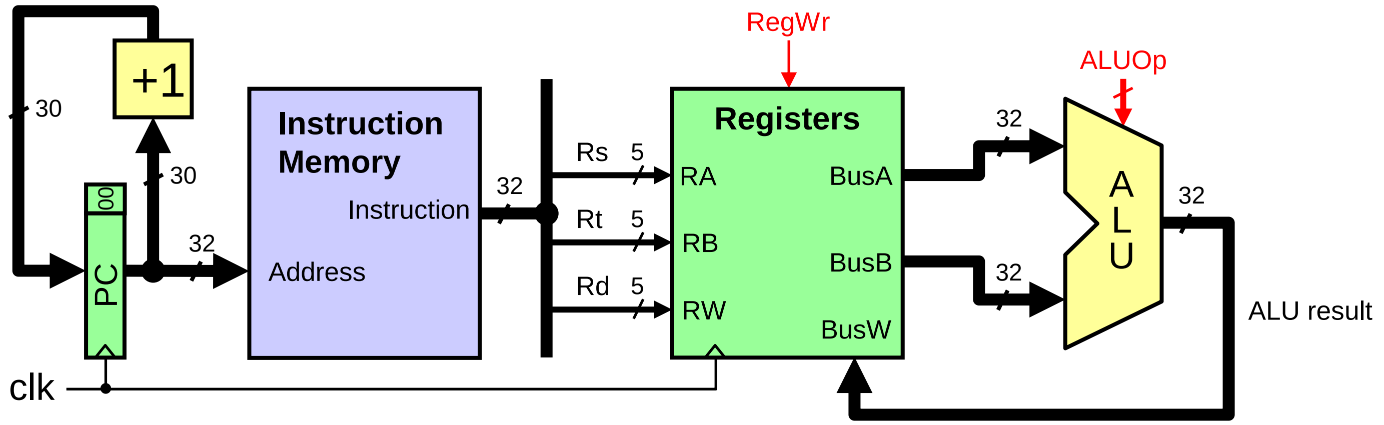
Improved datapath increments upper 30 bits of PC by 1

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

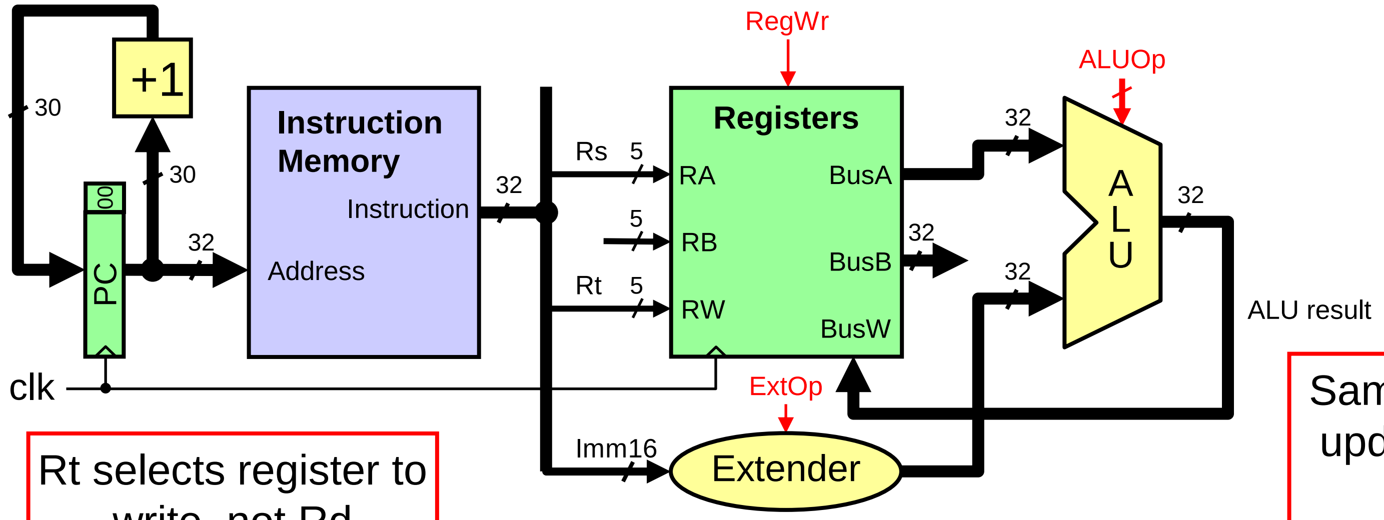
❖ Control signals

✧ ALUOp is the ALU operation as defined in the funct field for R-type

✧ RegWr is used to enable the writing of the ALU result

Datapath for I-type ALU Instructions

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



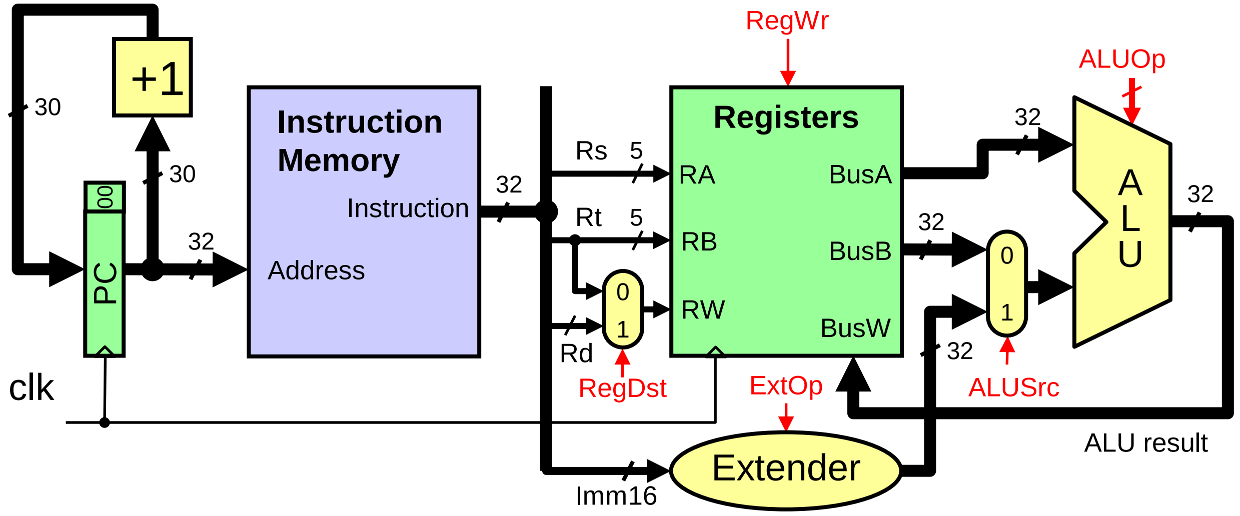
Rt selects register to write, not Rd

Same clock edge updates PC and Rt

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

- Control signals
  - ALUOp is derived from the Op field for I-type instructions
  - RegWr 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

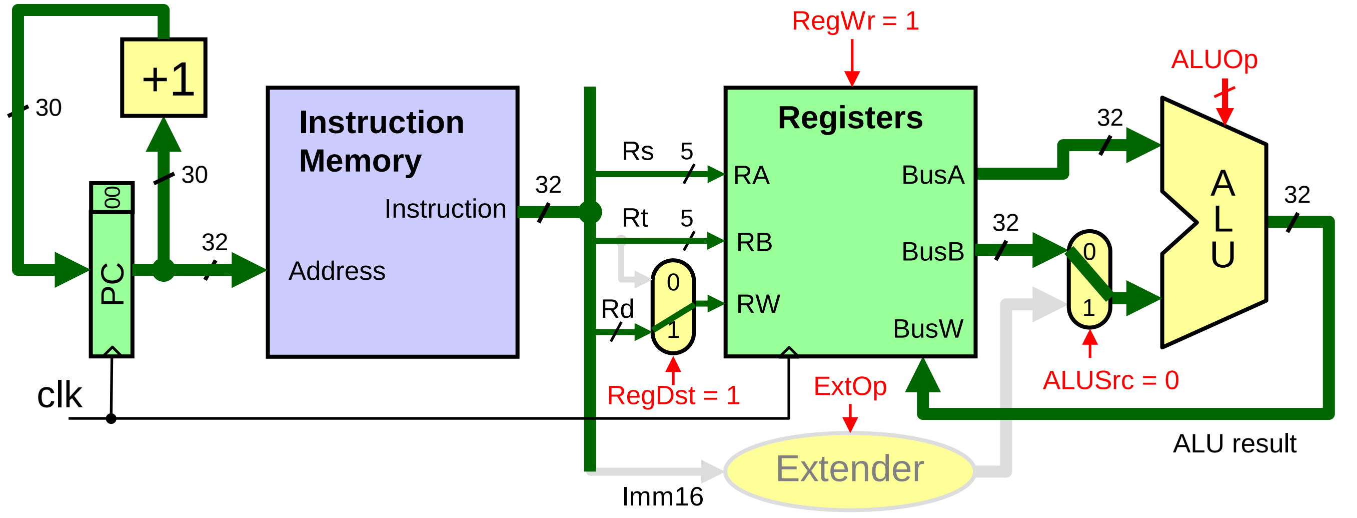


A mux selects RW as either Rt or Rd

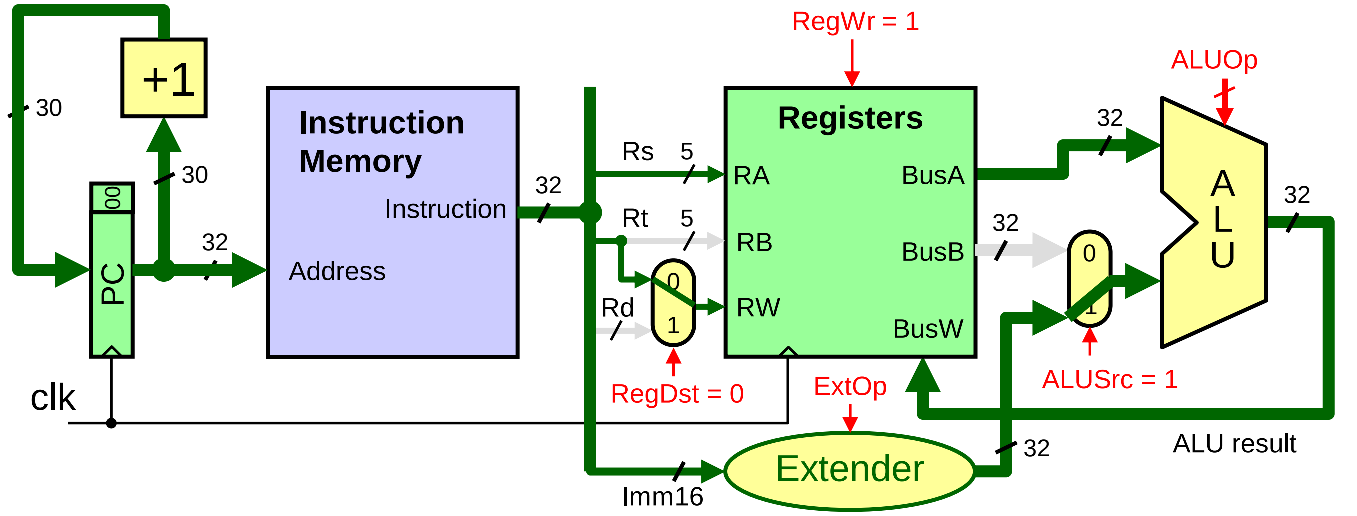
Another mux selects 2nd ALU input as either data on BusB or the extended immediate

- Control signals
  - ALUOp is derived from either the Op or the funct field
  - RegWr 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



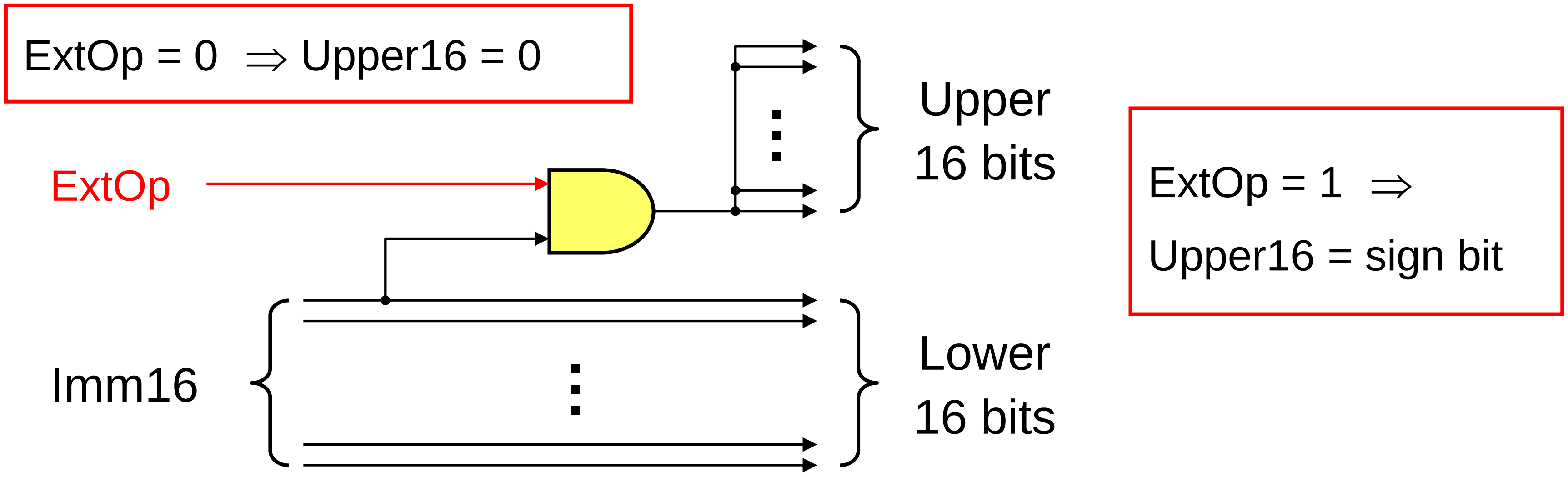
For R-type ALU instructions, RegDst is '1' to select Rd on RW and ALUSrc is '0' to select BusB as second ALU input. The active part of datapath is shown in **green**



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**

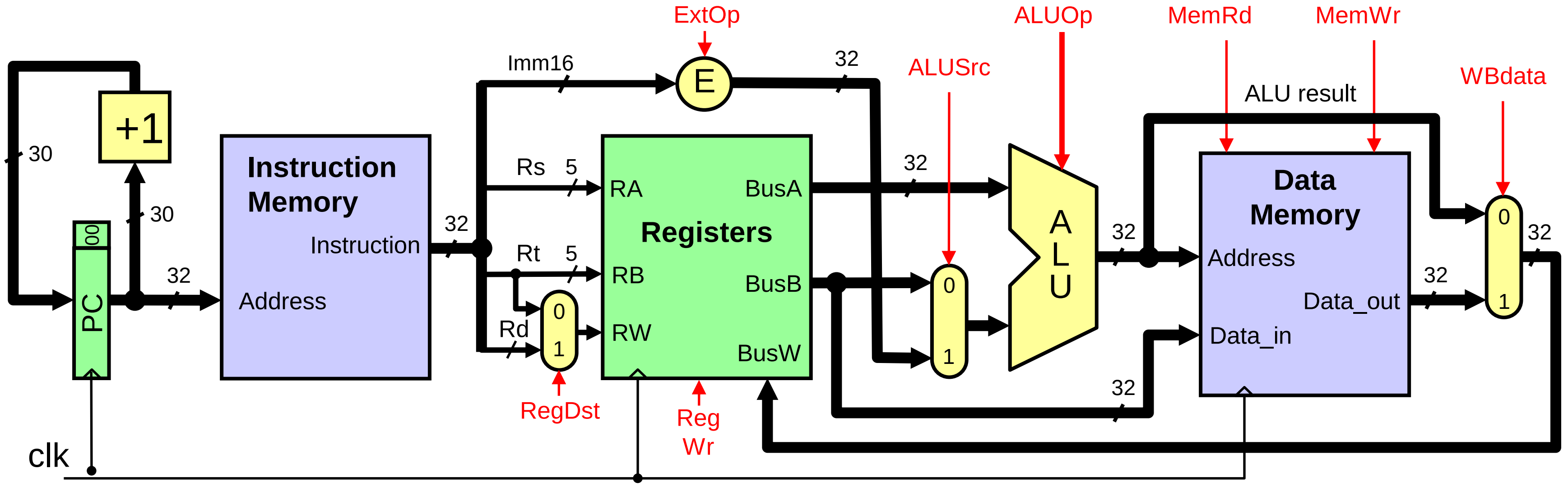
Details of the Extender

- ❖ 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



Adding Data Memory to Datapath

- A data memory is added for load and store instructions



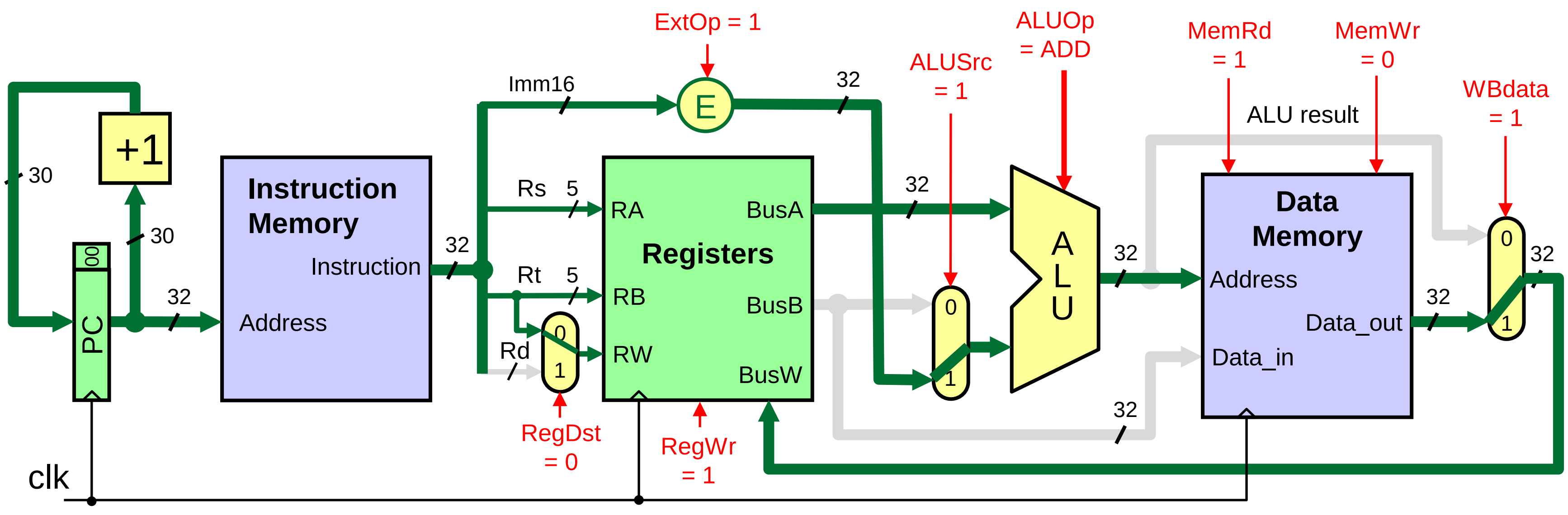
ALU calculates data memory address

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
  - MemRd for load instructions
  - MemWr for store instructions
  - WBdata selects data on BusW as ALU result or Memory Data_out

Controlling the Execution of Load



RegDst = '0' selects Rt as destination register

RegWr = '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

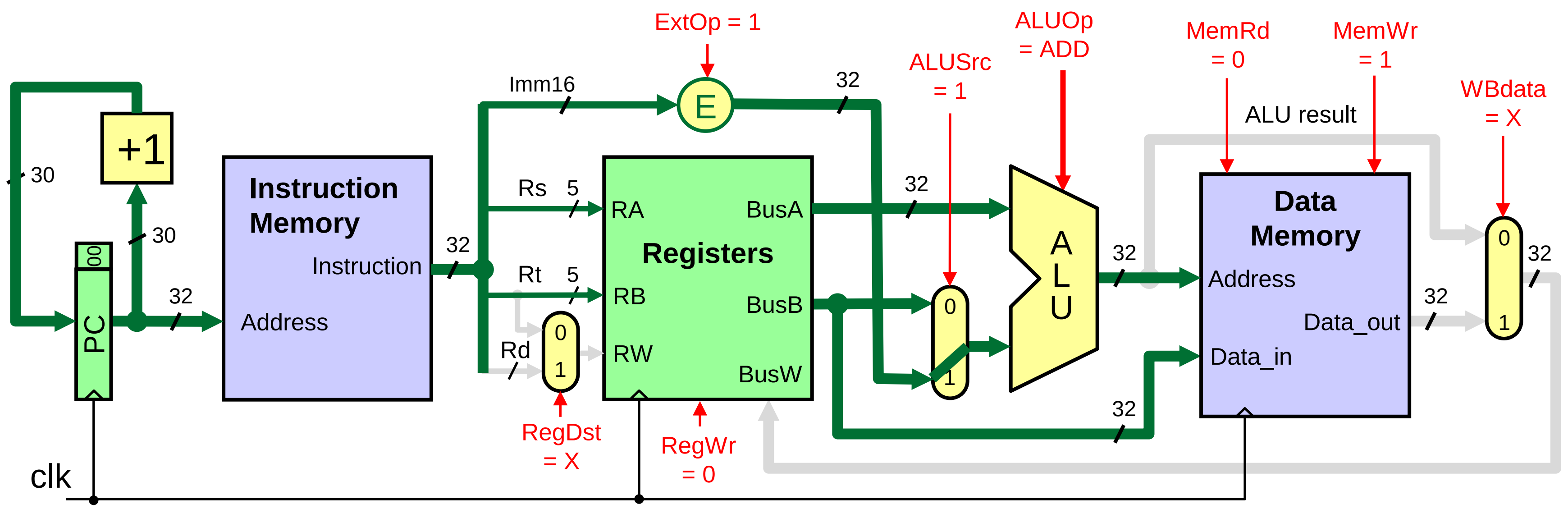
ALUOp = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemRd = '1' to read data memory

WBdata = '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

RegWr = '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

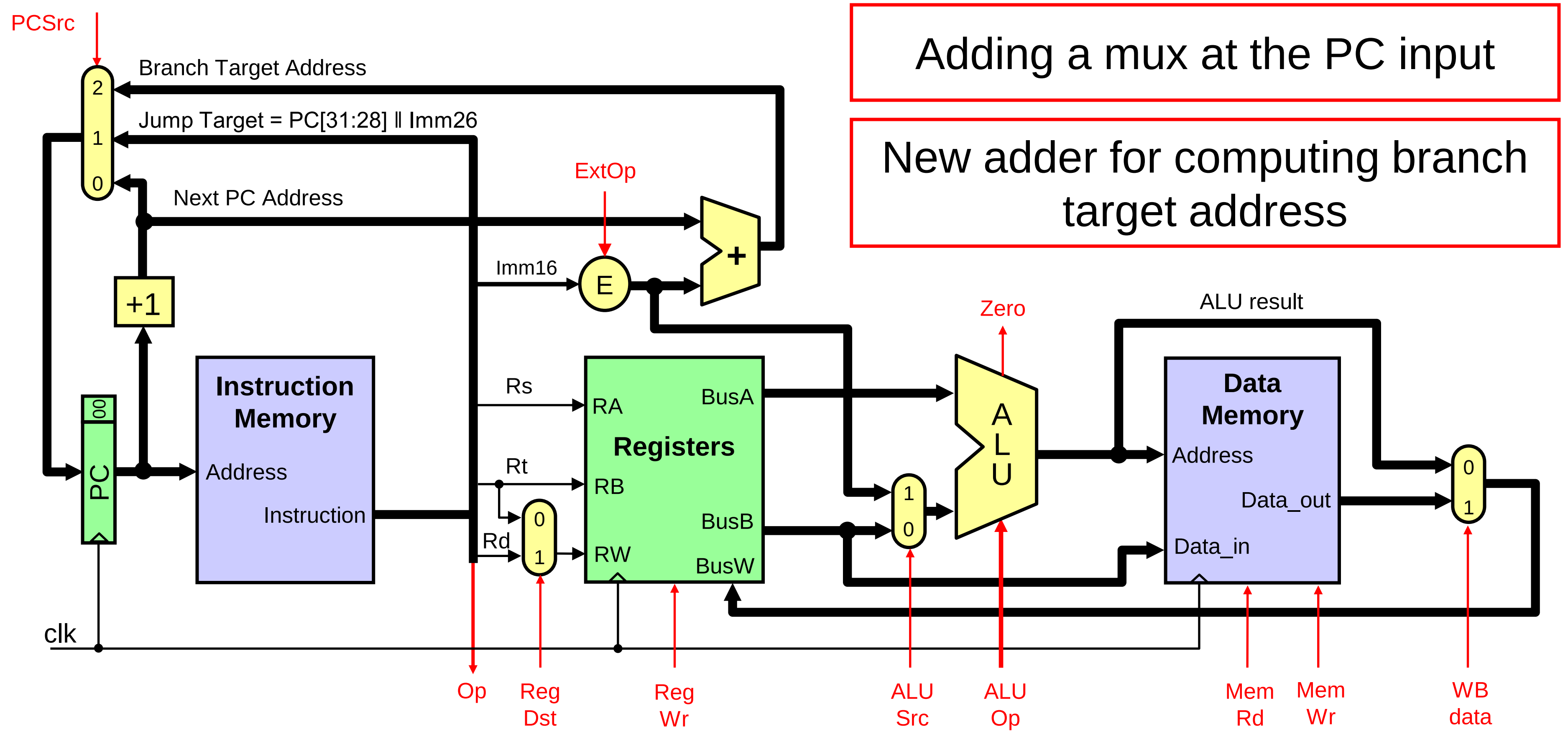
ALUOp = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemWr = '1' to write data memory

WBdata = 'X' because don't care what data is put on BusW

Clock edge updates PC and Data Memory

Adding Jump and Branch to Datapath



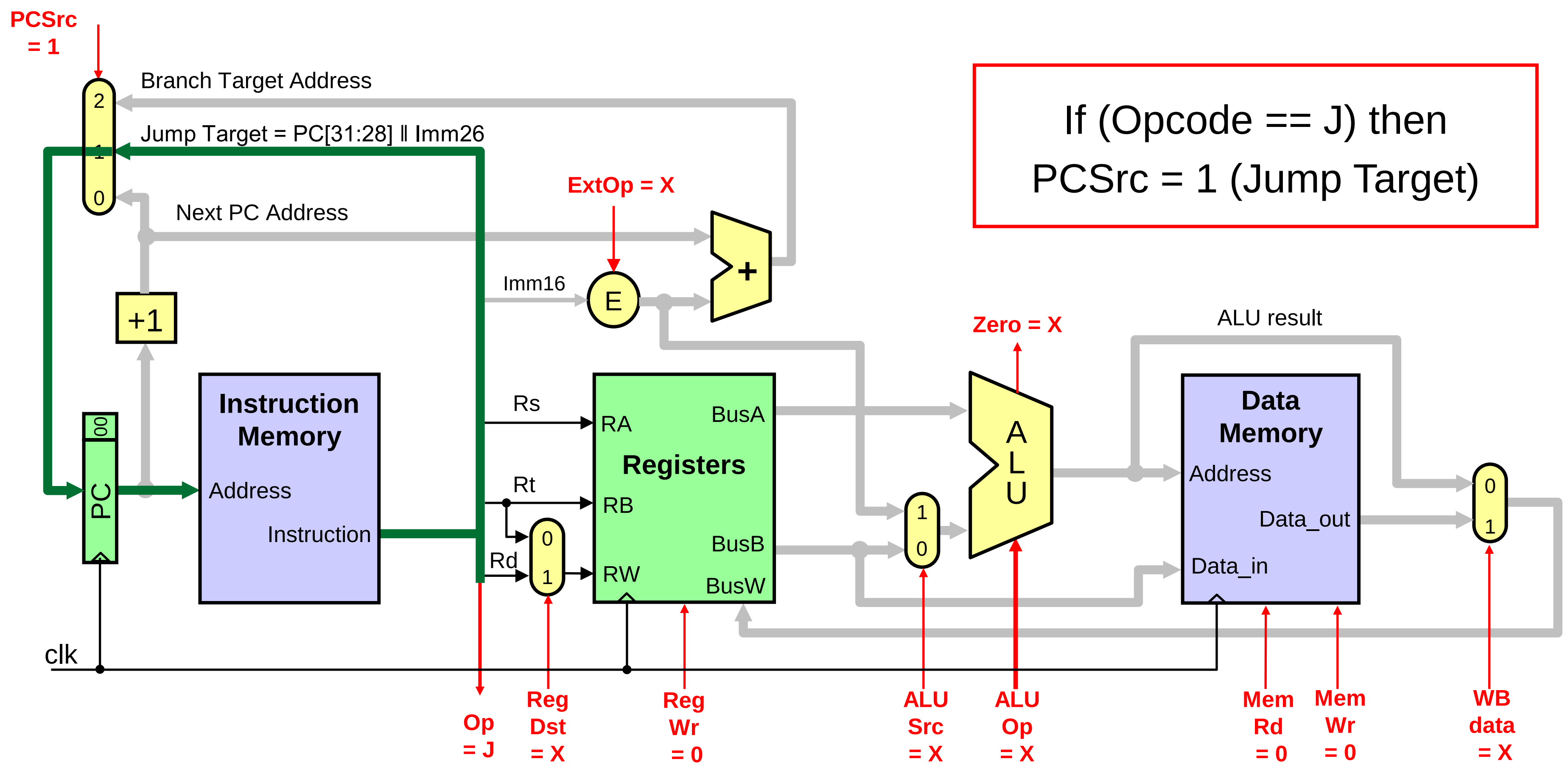
Adding a mux at the PC input

New adder for computing branch target address

❖ Additional Control Signals

- ✧ PCSrc for PC control: **1** for a jump and **2** for a taken branch
- ✧ Zero flag for branch control: whether branch is taken or not

Controlling the Execution of a Jump

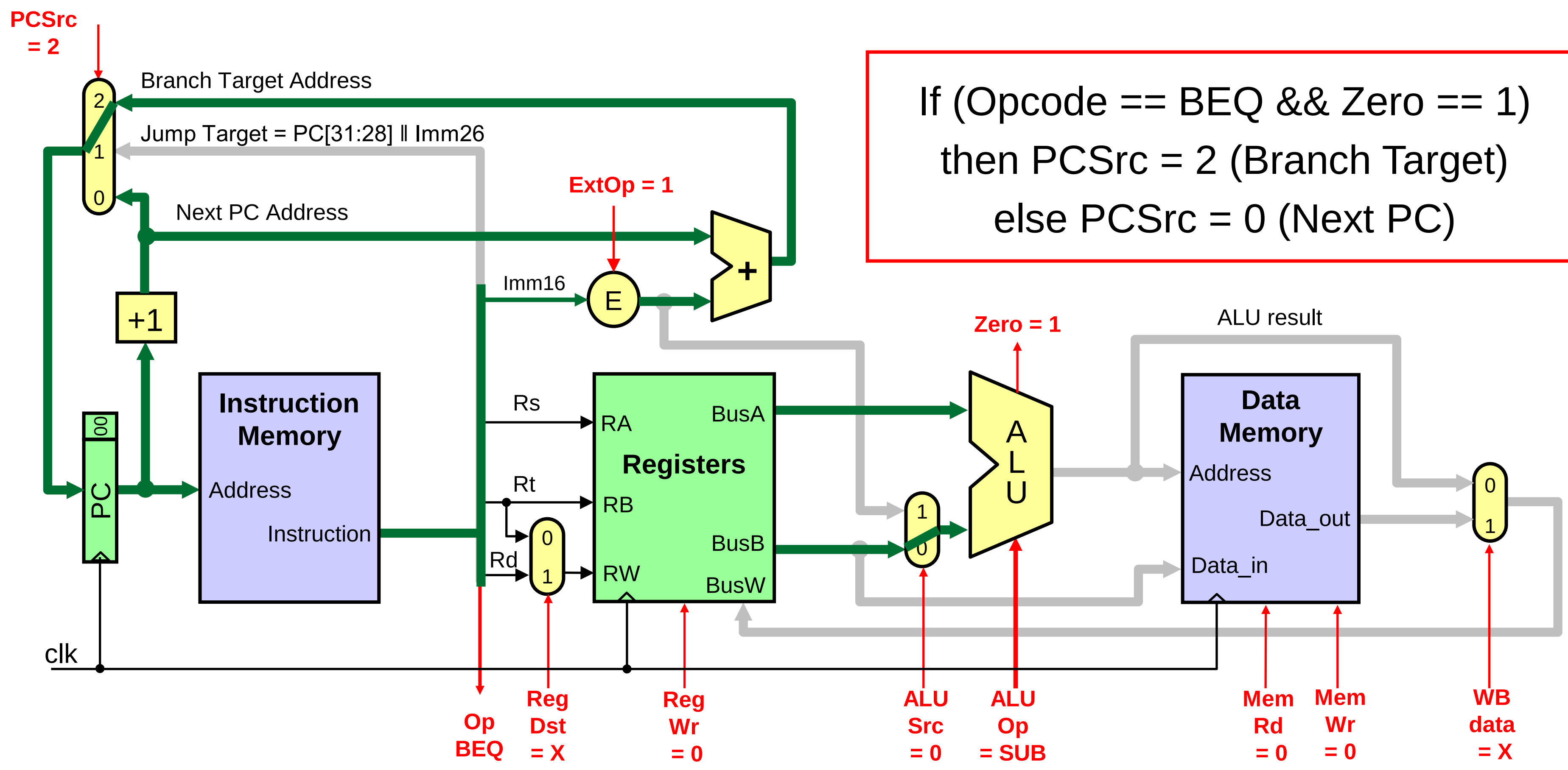


If (Opcode == J) then
PCSrc = 1 (Jump Target)

MemRd = MemWr = RegWr = 0, Don't care about other control signals

Clock edge updates PC register only

Controlling the Execution of a Branch



If (Opcode == BEQ && Zero == 1)
then PCSrc = 2 (Branch Target)
else PCSrc = 0 (Next PC)

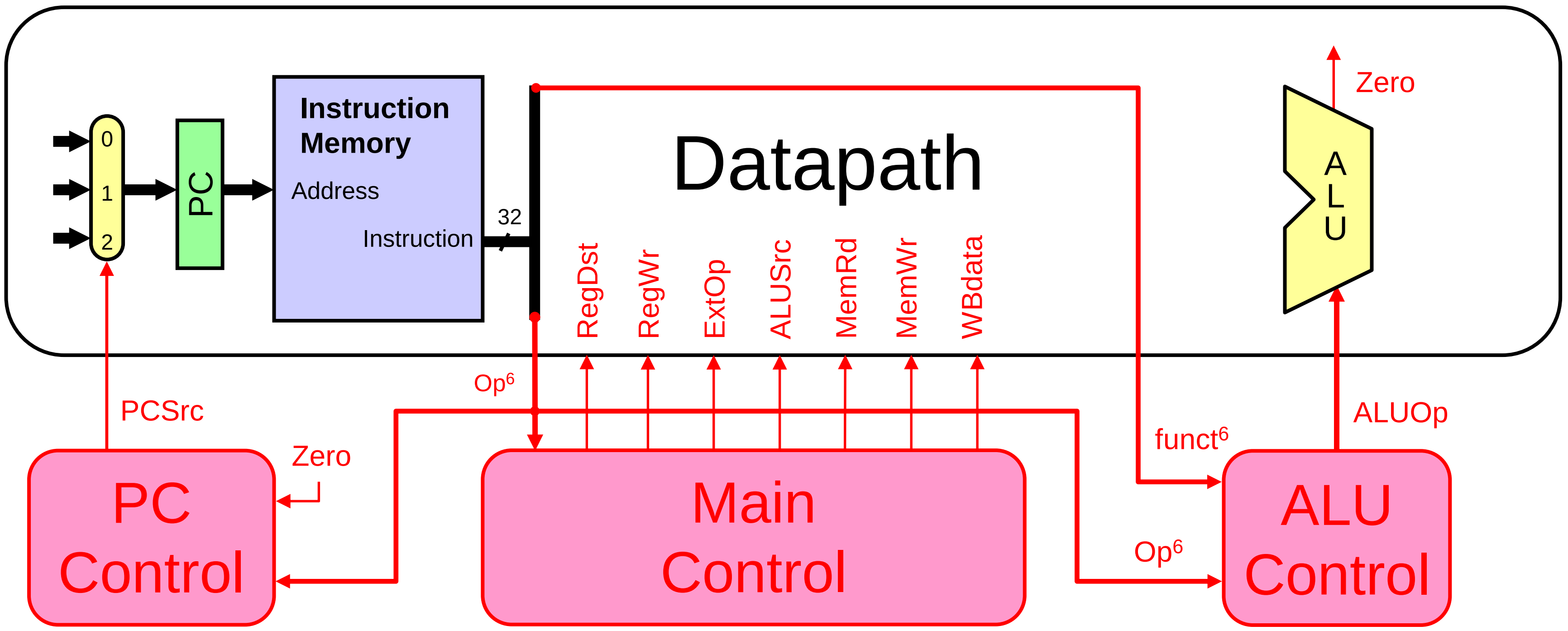
ALUSrc = 0, ALUOp = SUB, ExtOp = 1, MemRd = MemWr = RegWr = 0

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
- ❖ **Main, ALU, and PC Control**

Main, ALU, and PC Control



PC Control Input

- ✧ 6-bit opcode
- ✧ ALU zero flag

PC Control Output

- ✧ PCSrc signal

Main Control Input

- ✧ 6-bit opcode field

Main Control Output

- ✧ Main control signals

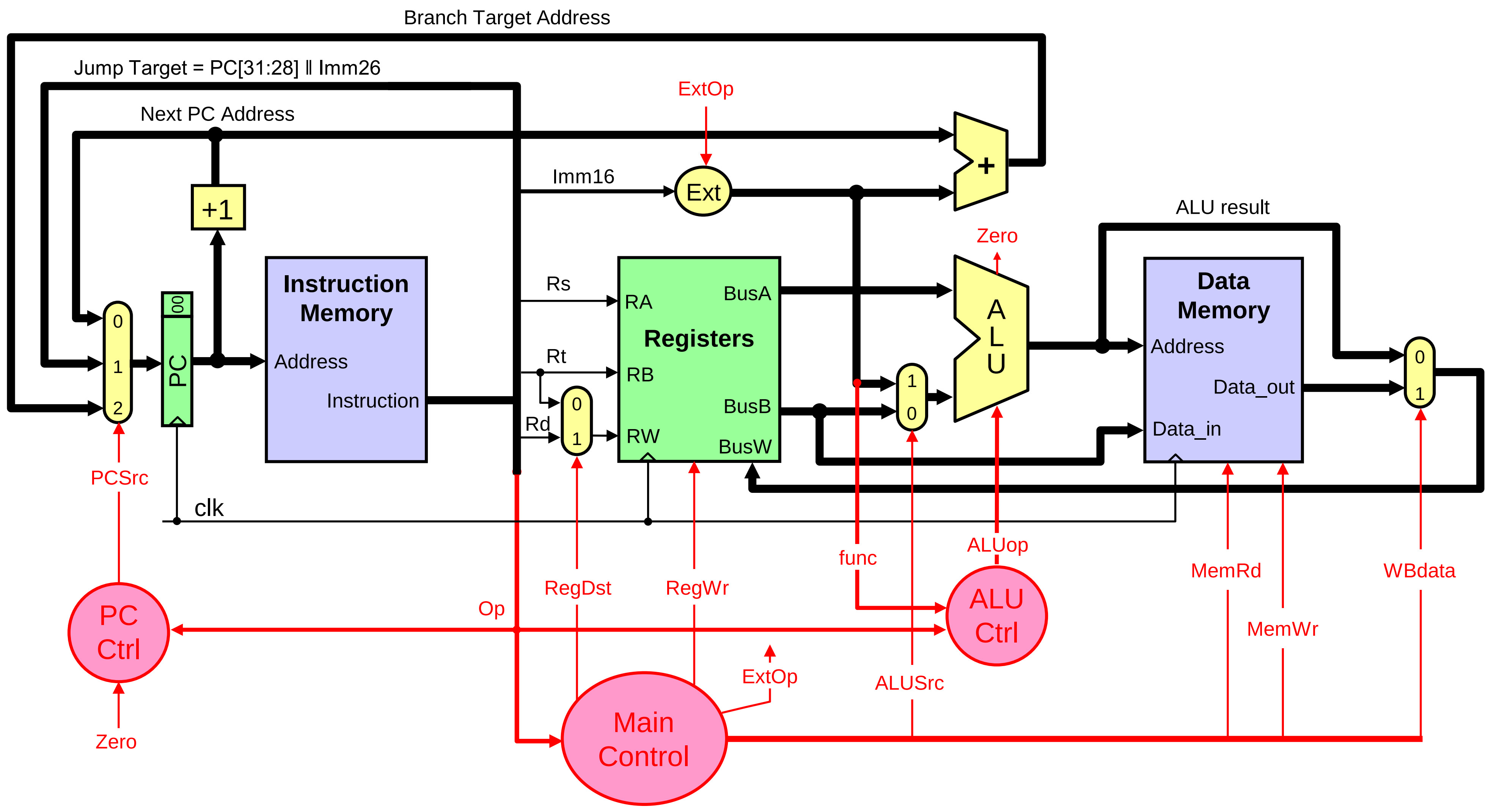
ALU Control Input

- ✧ 6-bit opcode field
- ✧ 6-bit function field

ALU Control Output

- ✧ ALUOp signal for ALU

Single-Cycle Datapath + Control



Main Control Signals

Signal	Effect when '0'	Effect when '1'
RegDst	Destination register = Rt	Destination register = Rd
RegWr	No register is written	Destination register (Rt or Rd) is written with the data on BusW
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
ALUSrc	Second ALU operand is the value of register Rt that appears on BusB	Second ALU operand is the value of the extended 16-bit immediate
MemRd	Data memory is NOT read	Data memory is read Data_out ← Memory[address]
MemWr	Data Memory is NOT written	Data memory is written Memory[address] ← Data_in
WBdata	BusW = ALU result	BusW = Data_out from Memory

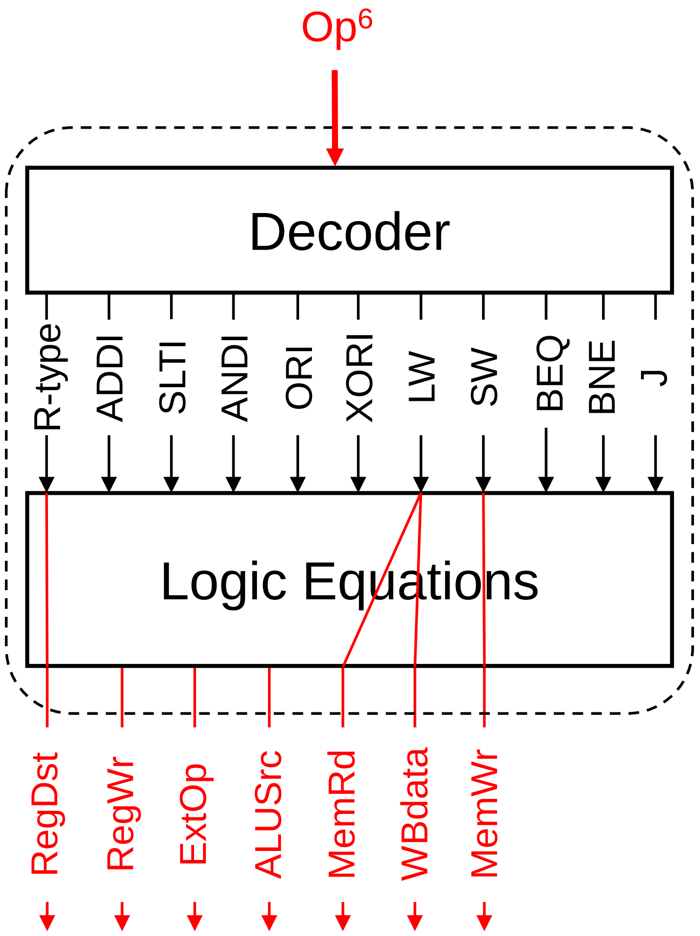
Main Control Truth Table

Op	RegDst	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata
R-type	1 = Rd	1	X	0 = BusB	0	0	0 = ALU
ADDI	0 = Rt	1	1 = sign	1 = Imm	0	0	0 = ALU
SLTI	0 = Rt	1	1 = sign	1 = Imm	0	0	0 = ALU
ANDI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
ORI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
XORI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
LW	0 = Rt	1	1 = sign	1 = Imm	1	0	1 = Mem
SW	X	0	1 = sign	1 = Imm	0	1	X
BEQ	X	0	1 = sign	0 = BusB	0	0	X
BNE	X	0	1 = sign	0 = BusB	0	0	X
J	X	0	X	X	0	0	X

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

Logic Equations for Main Control Signals

RegDst	=	R-type
RegWrite	=	$\overline{(SW + BEQ + BNE + J)}$
ExtOp	=	$\overline{(ANDI + ORI + XORI)}$
ALUSrc	=	$\overline{(R\text{-}type + BEQ + BNE)}$
MemRd	=	LW
MemWr	=	SW
WBdata	=	LW



ALU Control Truth Table

Op	funct	ALUOp	4-bit Coding
R-type	AND	AND	0001
R-type	OR	OR	0010
R-type	XOR	XOR	0011
R-type	ADD	ADD	0100
R-type	SUB	SUB	0101
R-type	SLT	SLT	0110
ADDI	X	ADD	0100
SLTI	X	SLT	0110
ANDI	X	AND	0001
ORI	X	OR	0010
XORI	X	XOR	0011
LW	X	ADD	0100
SW	X	ADD	0100
BEQ	X	SUB	0101
BNE	X	SUB	0101
J	X	X	X

The 4-bit Coding defines the binary ALU operations.

Logic equations are derived for the 4-bit coding.

Other bit-coding can be used. The goal is to simplify the ALU control.

PC Control Truth Table

Op	Zero flag	PCSrc
R-type	X	0 = Increment PC
J	X	1 = Jump Target Address
BEQ	0	0 = Increment PC
BEQ	1	2 = Branch Target Address
BNE	0	2 = Branch Target Address
BNE	1	0 = Increment PC
Other than Jump or Branch	X	0 = Increment PC

The ALU Zero flag is used by BEQ and BNE instructions

PC Control Logic

❖ The PC control logic can be described as follows:

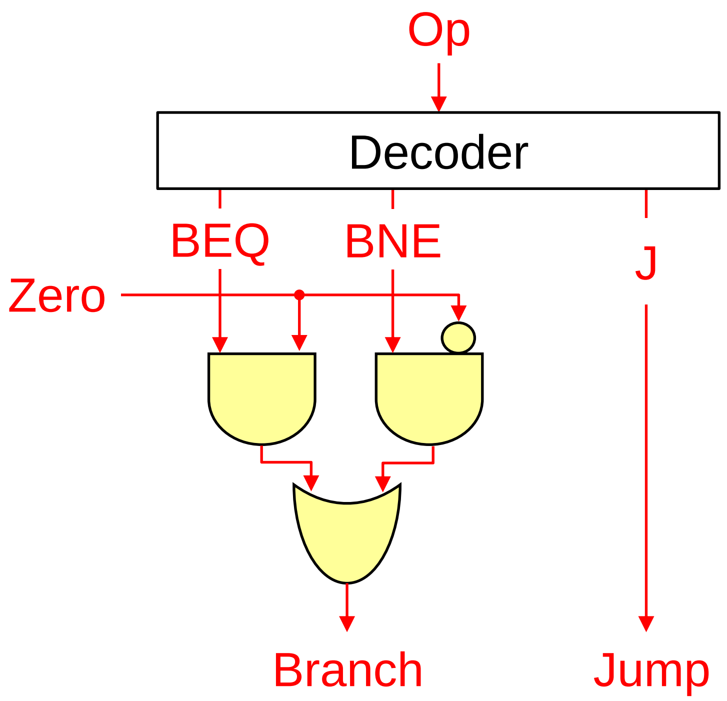
```
if (Op == J) PCSrc = 1;
else if ((Op == BEQ && Zero == 1) ||
         (Op == BNE && Zero == 0)) PCSrc = 2;
else PCSrc = 0;
```

$\text{Branch} = (\text{BEQ} . \text{Zero}) + (\text{BNE} . \overline{\text{Zero}})$

Branch = 1, Jump = 0 ➔ PCSrc = 2

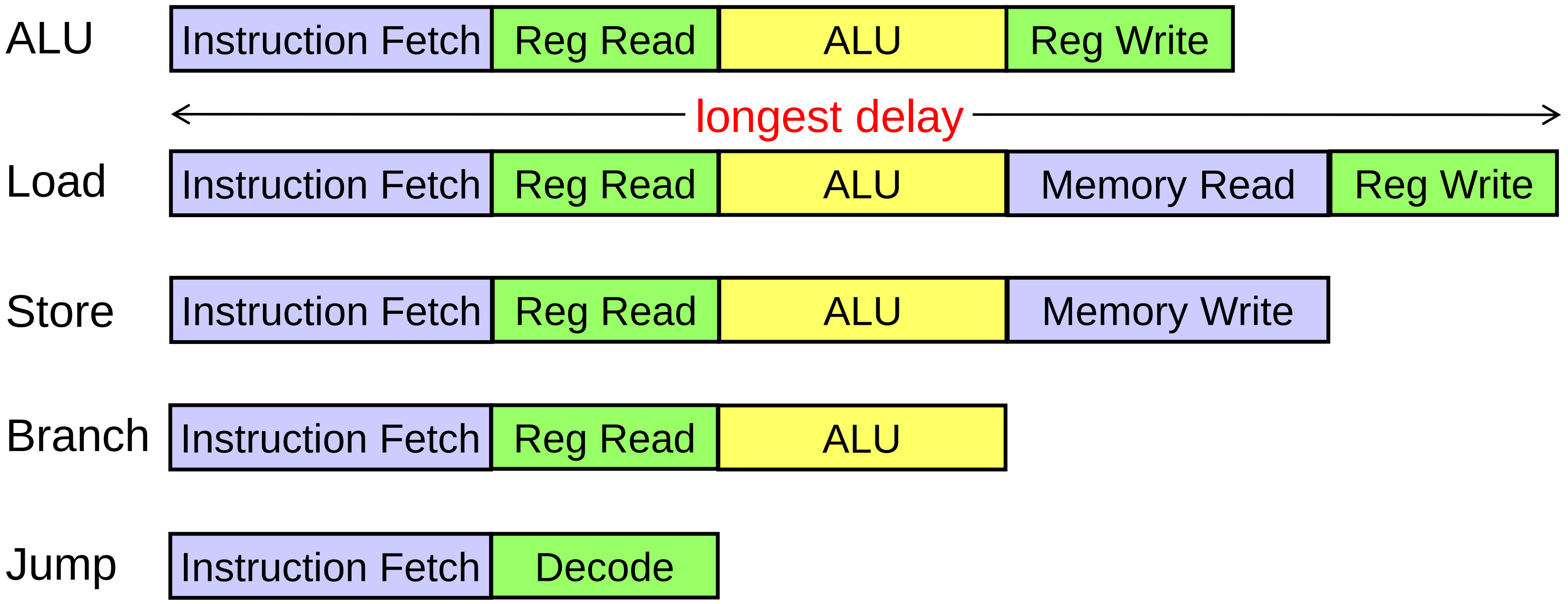
Branch = 0, Jump = 1 ➔ PCSrc = 1

Branch = 0, Jump = 0 ➔ PCSrc = 0



Drawbacks of Single Cycle Processor

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



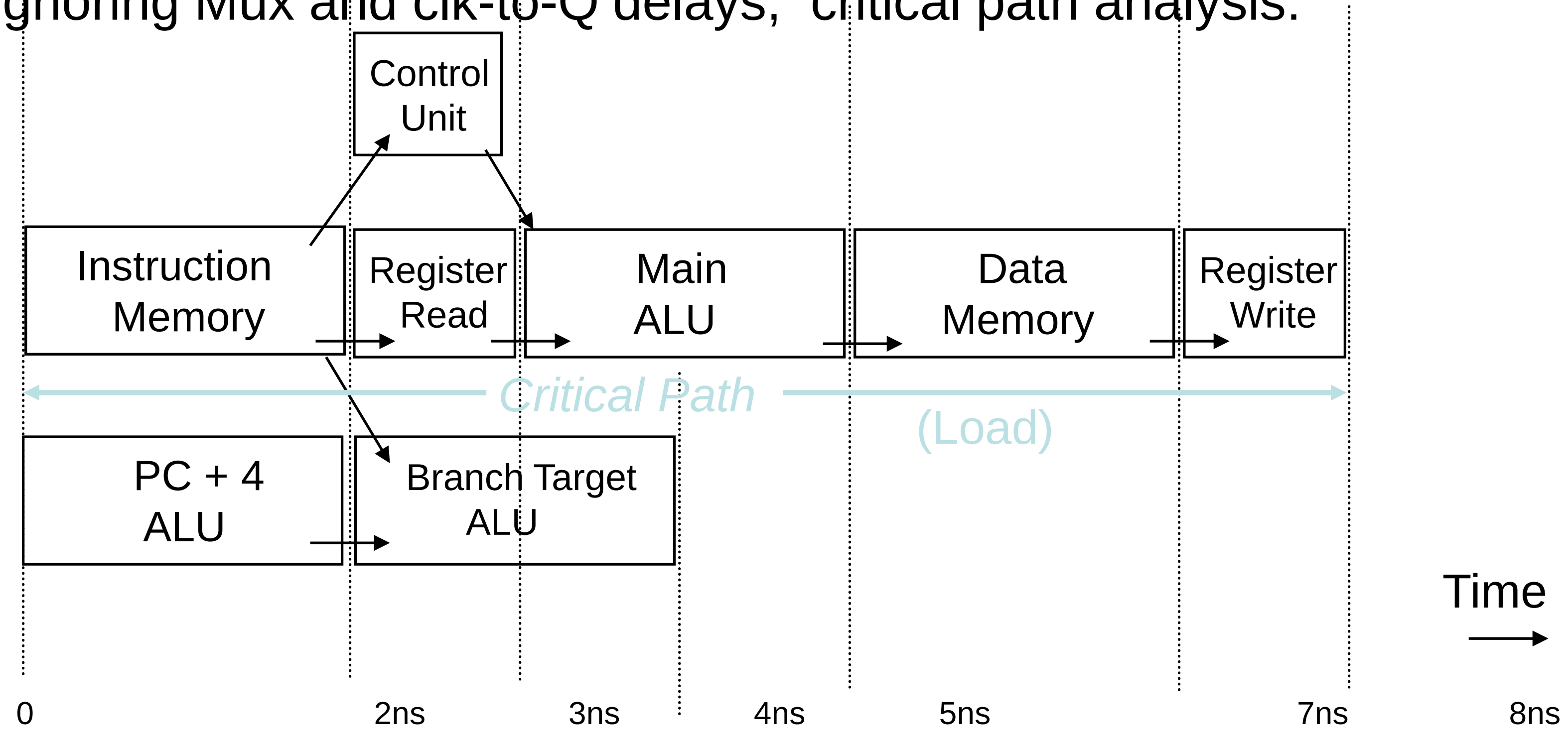
- Alternative Solution: Multicycle implementation
  - Break down instruction execution into multiple cycles

Simplified Single Cycle Datapath Timing

- Assuming the following datapath/control hardware components delays:
  - Memory Units: 2 ns
  - ALU and adders: 2 ns
  - Register File: 1 ns
  - Control Unit < 1 ns

} Obtained from low-level target VLSI implementation technology of components

- Ignoring Mux and clk-to-Q delays, critical path analysis:



125 MHz

ns = nanosecond = 10^{-9} second

Drawbacks of Single Cycle Processor

1. Long cycle time:
   - ✧ All instructions must take as much time as the slowest
     - ▪ Here, cycle time for load is longer than needed for all other instructions.
       - – Cycle time must be long enough for the load instruction:

         PC's Clock -to-Q + Instruction Memory Access Time +
         Register File Access Time + ALU Delay (address calculation) +
         Data Memory Access Time + Register File Setup Time + Clock Skew
   - ✧ Real memory is not as well-behaved as idealized memory
     - ▪ Cannot always complete data access in one (short) cycle.
2. Impossible to implement complex, variable-length instructions and complex addressing modes in a single cycle.
   - ✧ e.g indirect memory addressing.
3. High and duplicate hardware resource requirements
   - ✧ Any hardware functional unit cannot be used more than once in a single cycle (e.g. ALUs).
4. Does not allow overlap of instruction processing

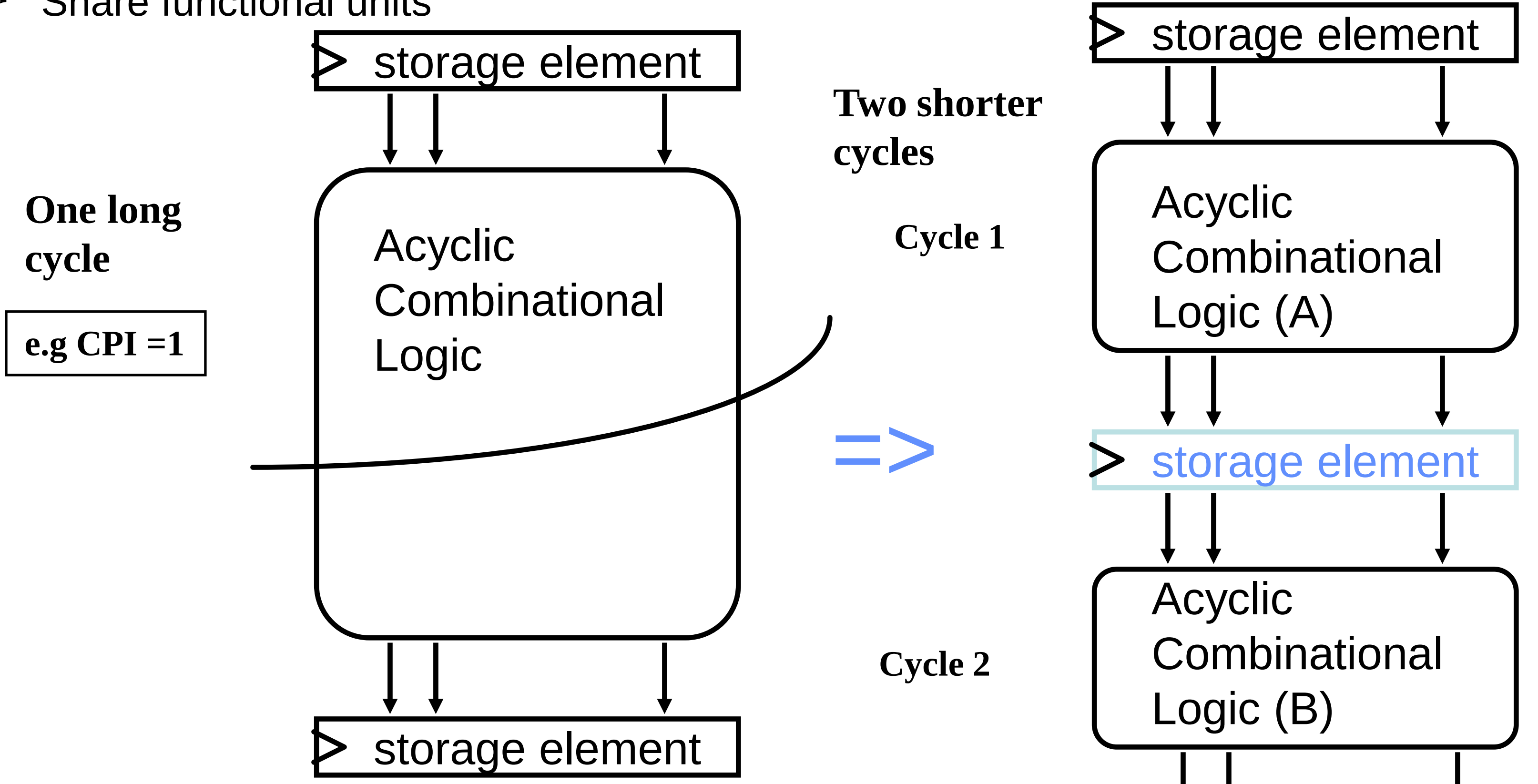
Alternative: Multicycle Implementation

- 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

Reducing Cycle Time: Multi-Cycle Design

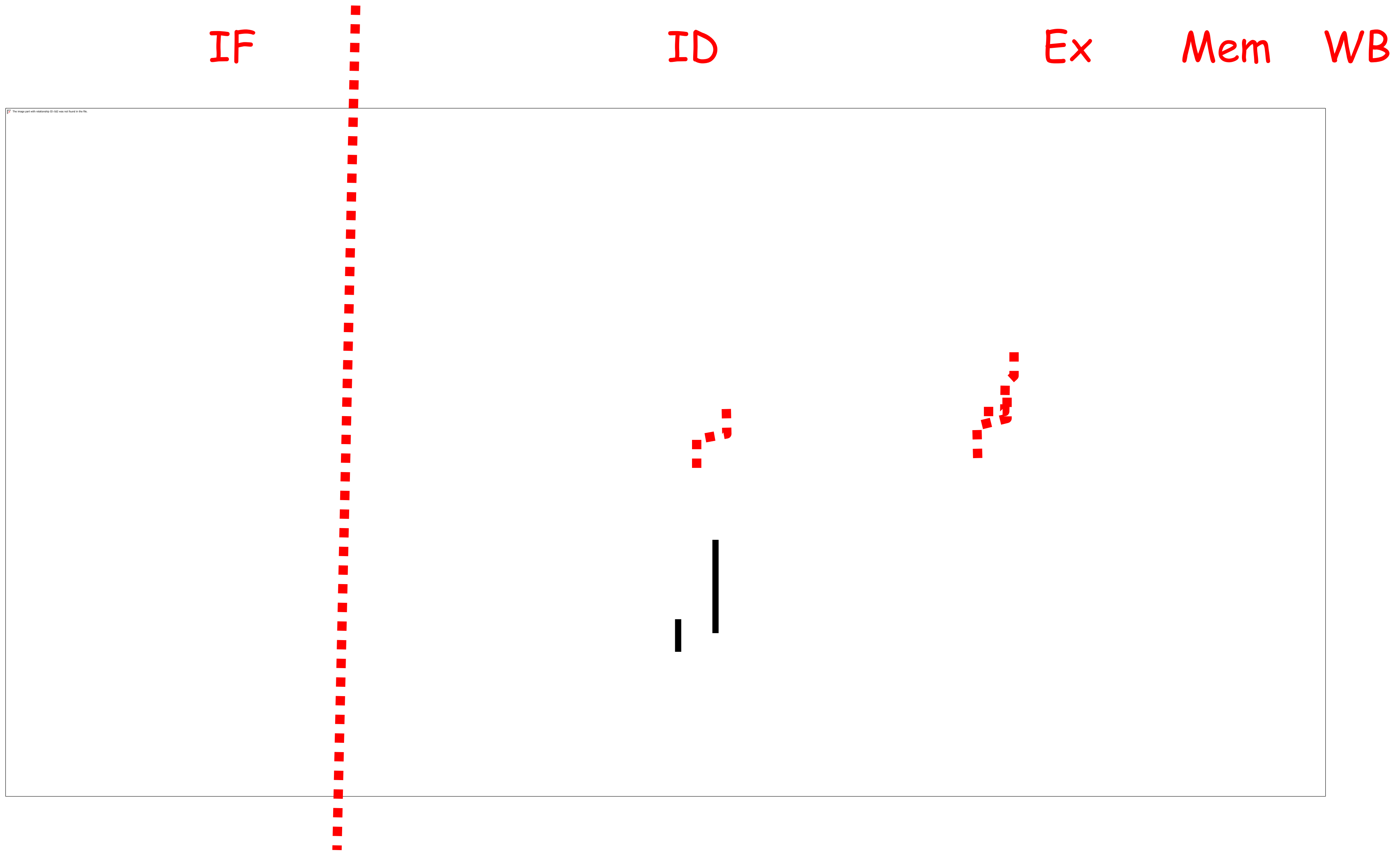
- Cut combinational dependency graph by <u>inserting registers / latches.</u>
- The same work is done in two or more shorter cycles, rather than one long cycle.
  - Different CPI
  - Share functional units



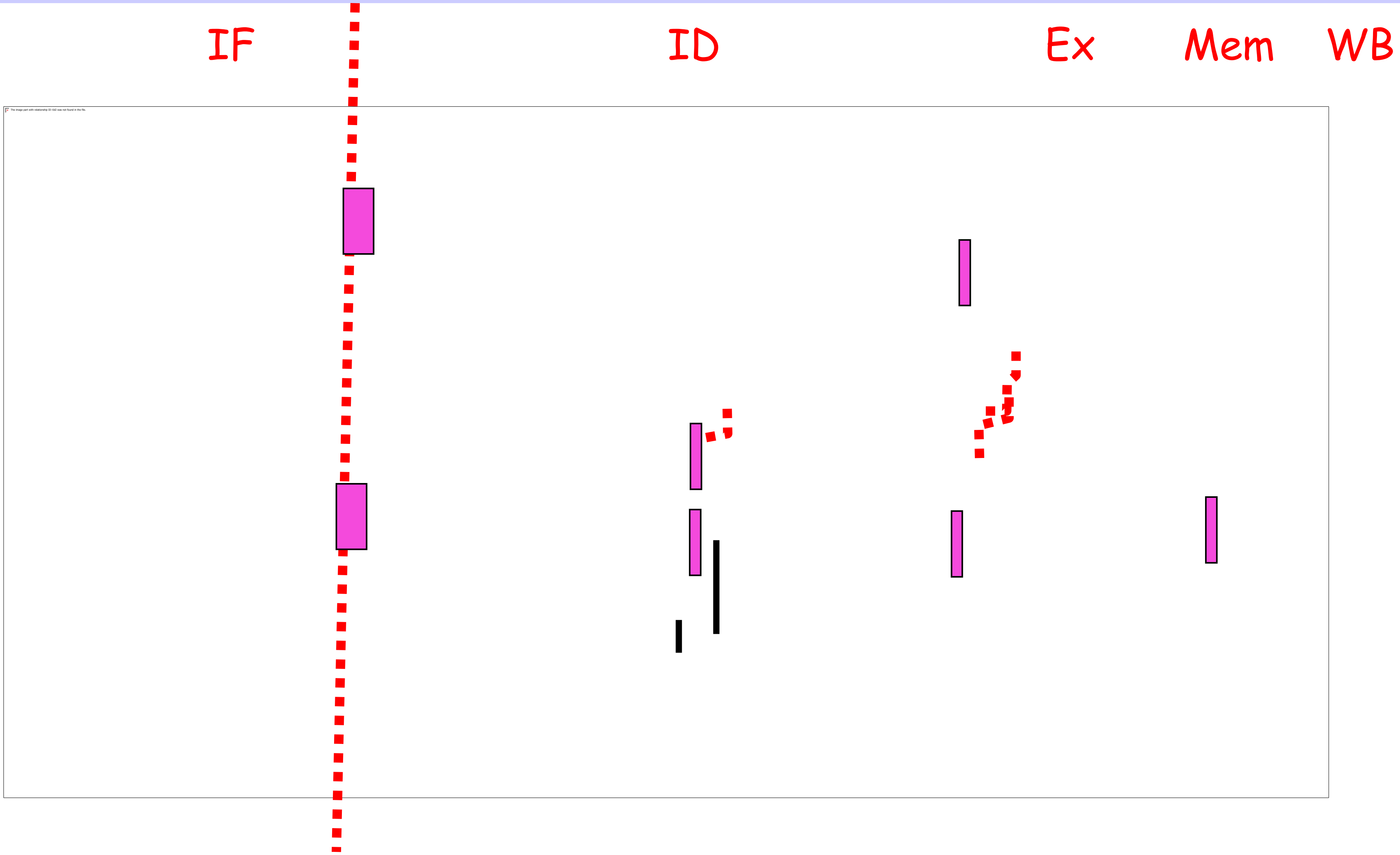
Place registers to:

- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

Partitioning the Single-Cycle Design


IF
ID
Ex
Mem
WB
The image part with relationship ID rId2 was not found in the file.

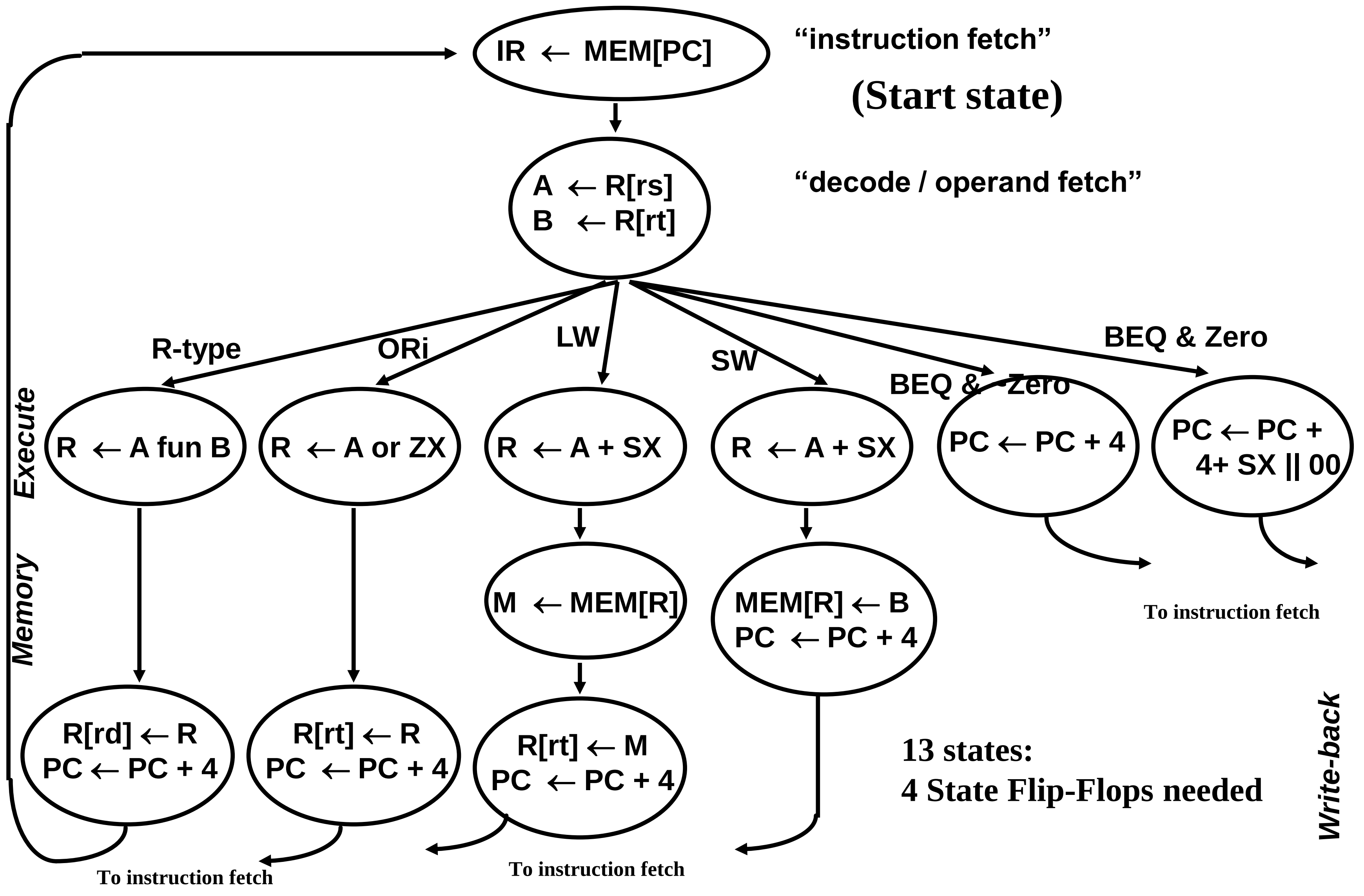
Where to add registers



Place registers to:

- **Get a balanced clock cycle length**
- **Save any results needed for the remaining cycles**

Control Specification For Multi-cycle CPU
Finite State Machine (FSM) - State Transition Diagram



13 states:
4 State Flip-Flops needed

Multi-cycle Datapath Instruction CPI

- R-Type/Immediate: Require four cycles, <u>CPI = 4</u>
  - IF, ID, EX, WB
- Loads: Require five cycles, <u>CPI = 5</u>
  - IF, ID, EX, MEM, WB
- Stores: Require four cycles, <u>CPI = 4</u>
  - IF, ID, EX, MEM
- Branches/Jumps: Require three cycles, <u>CPI = 3</u>
  - IF, ID, EX

- Average or effective program CPI: $3 \le \text{CPI} \le 5$ depending on program profile (instruction mix).

Performance Example

- ❖ Assume the following operation times for components:
  - ✧ Access time for Instruction and data memories: 200 ps
  - ✧ Delay in ALU and adders: 180 ps
  - ✧ Delay in Decode and Register file access (read or write): 150 ps
  - ✧ Ignore the other 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 update PC		530 ps
Jump	200	150	← Decode and update 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 = $0.4 \times 4 + 0.2 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.1 \times 2 = 3.8$
- 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 the same size
  - Source registers always in the same place
  - Immediate constants are of same size and same location
  - Operations are always on registers/immediates