移动信息工程学院
School of Mobile Information Engineering

中山大學
SUN YAT-SEN UNIVERSITY

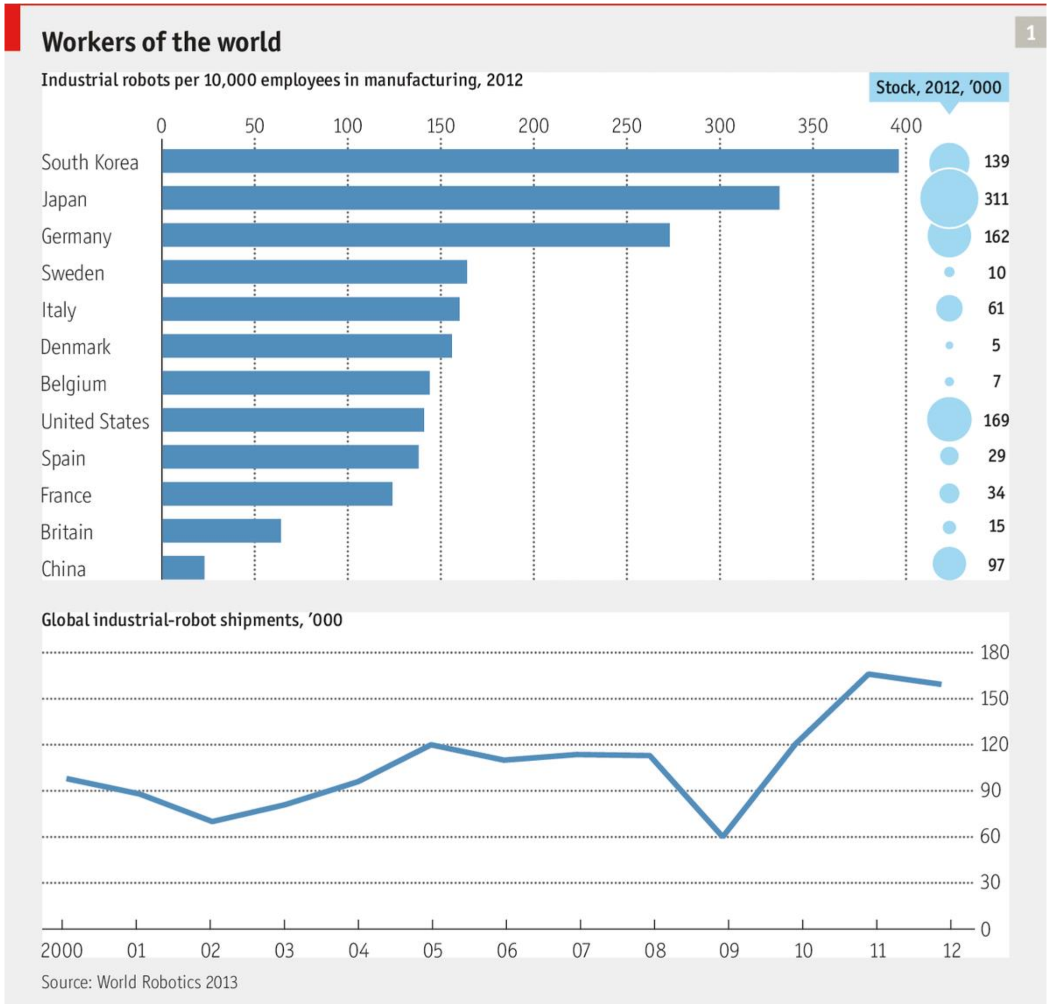
# Single Cycle Processor Design

Kai Huang

# Rise of the robots

The Economist

Mar 29th 2014



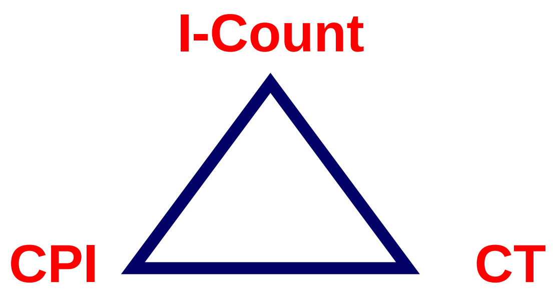
http://www.economist.com/printedition/2014-03-29

## 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 (CT)
- 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

## MIPS Instruction Formats

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

| | | | | | |
|---|---|---|---|---|---|
| $Op^6$ | $Rs^5$ | $Rt^5$ | $Rd^5$ | $sa^5$ | $funct^6$ |

| | | | |
|---|---|---|---|
| $Op^6$ | $Rs^5$ | $Rt^5$ | $immediate^{16}$ |

| | |
|---|---|
| $Op^6$ | $immediate^{26}$ |

  - $Op^6$: 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

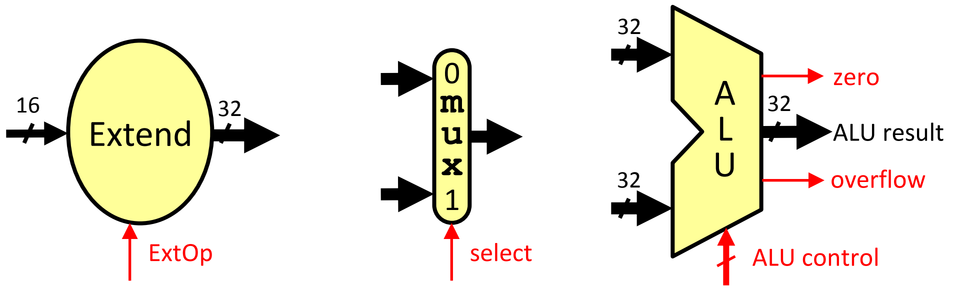
- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 32 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and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

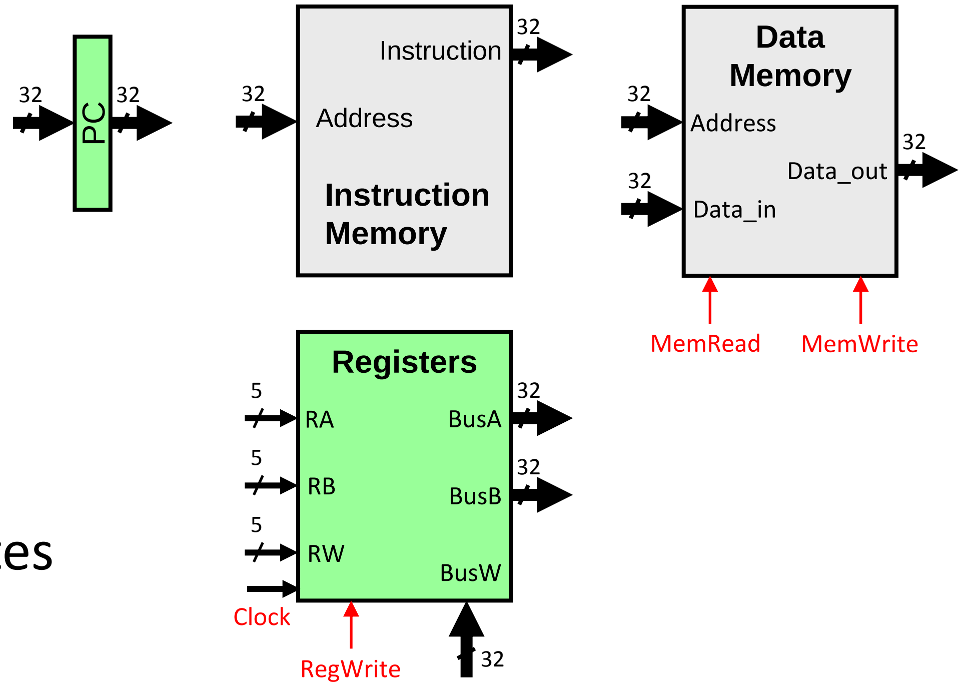
## Next . . .

- 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

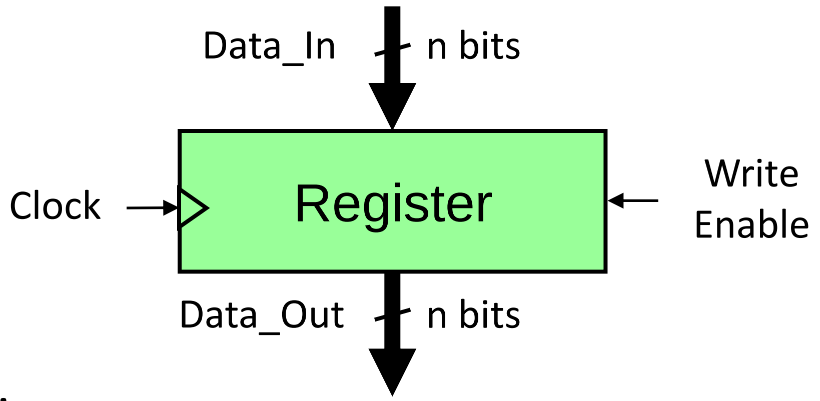
- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file
- Clocking methodology
  - Timing of reads and writes





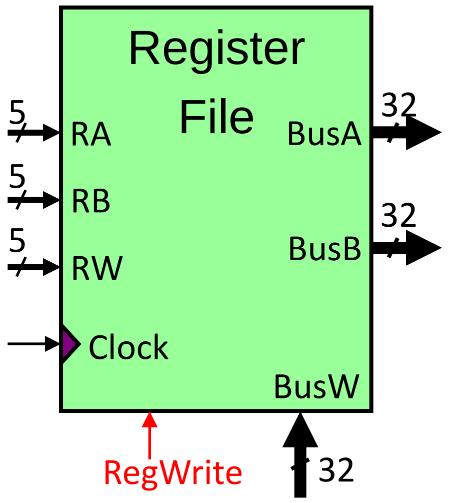
## Register Element

- Register
  - Similar to the D-type Flip-Flop
- n-bit input and output
- Write Enable:
  - 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



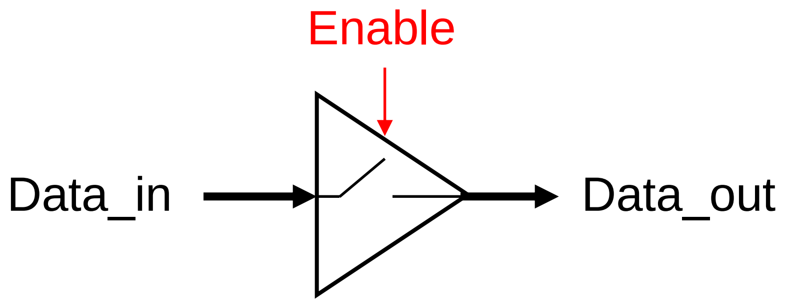
# MIPS Register File

- 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

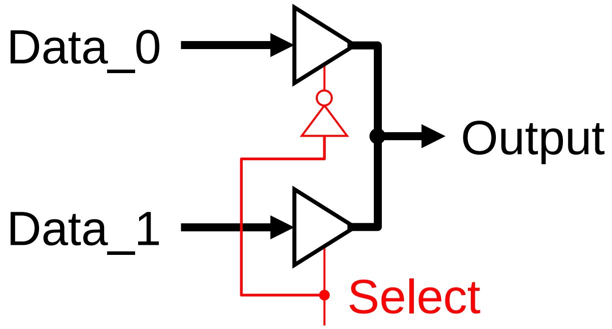


## Tri-State Buffers

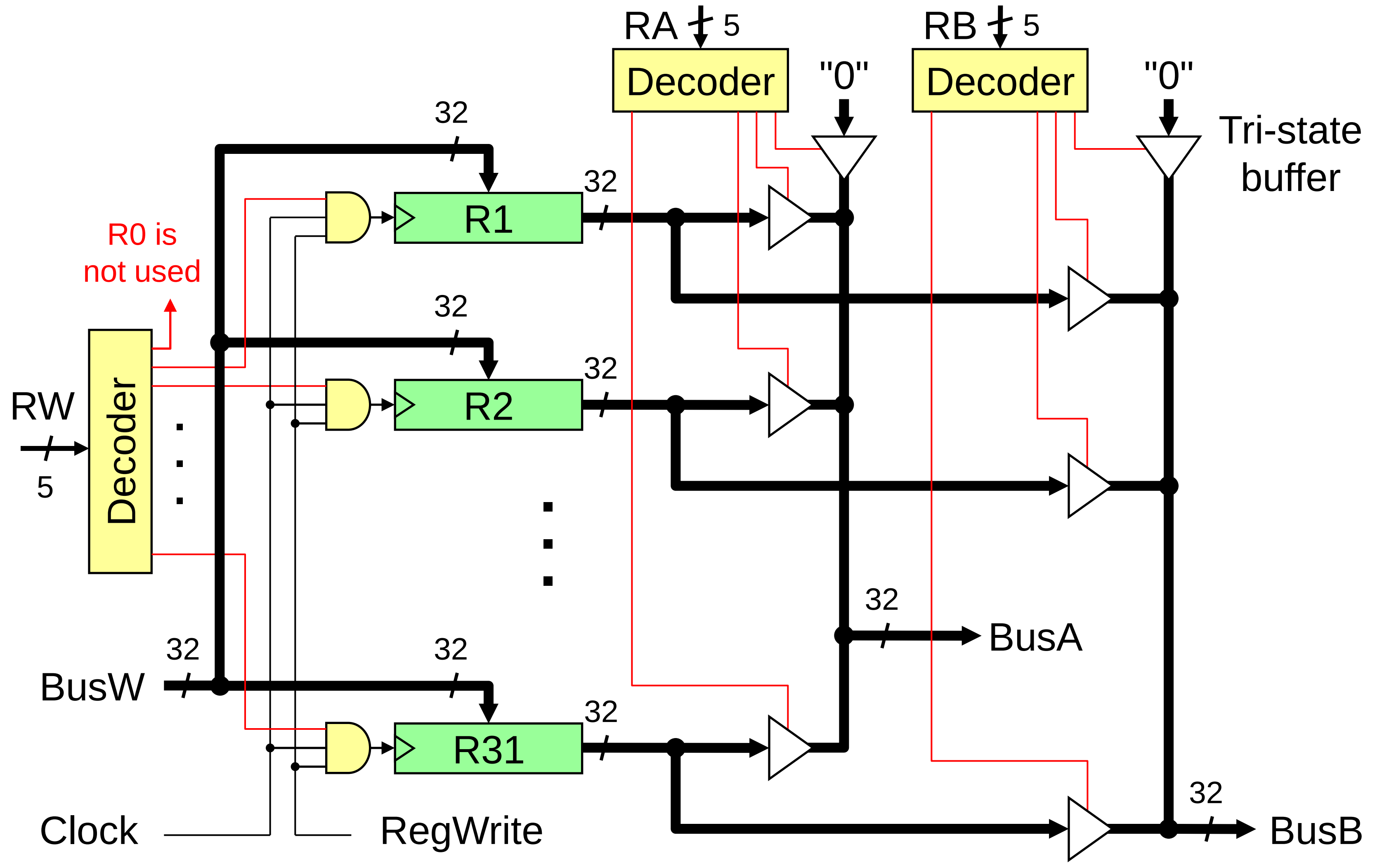
- Allow multiple sources to drive a single bus
- Two Inputs:
  - Data signal (data_in)
  - Output enable
- One Output (data_out):
  - If (Enable) Data_out = Data_in
    else Data_out = High Impedance state (output is disconnected)



- Tri-state buffers can be used to build multiplexors



## Details of the Register File



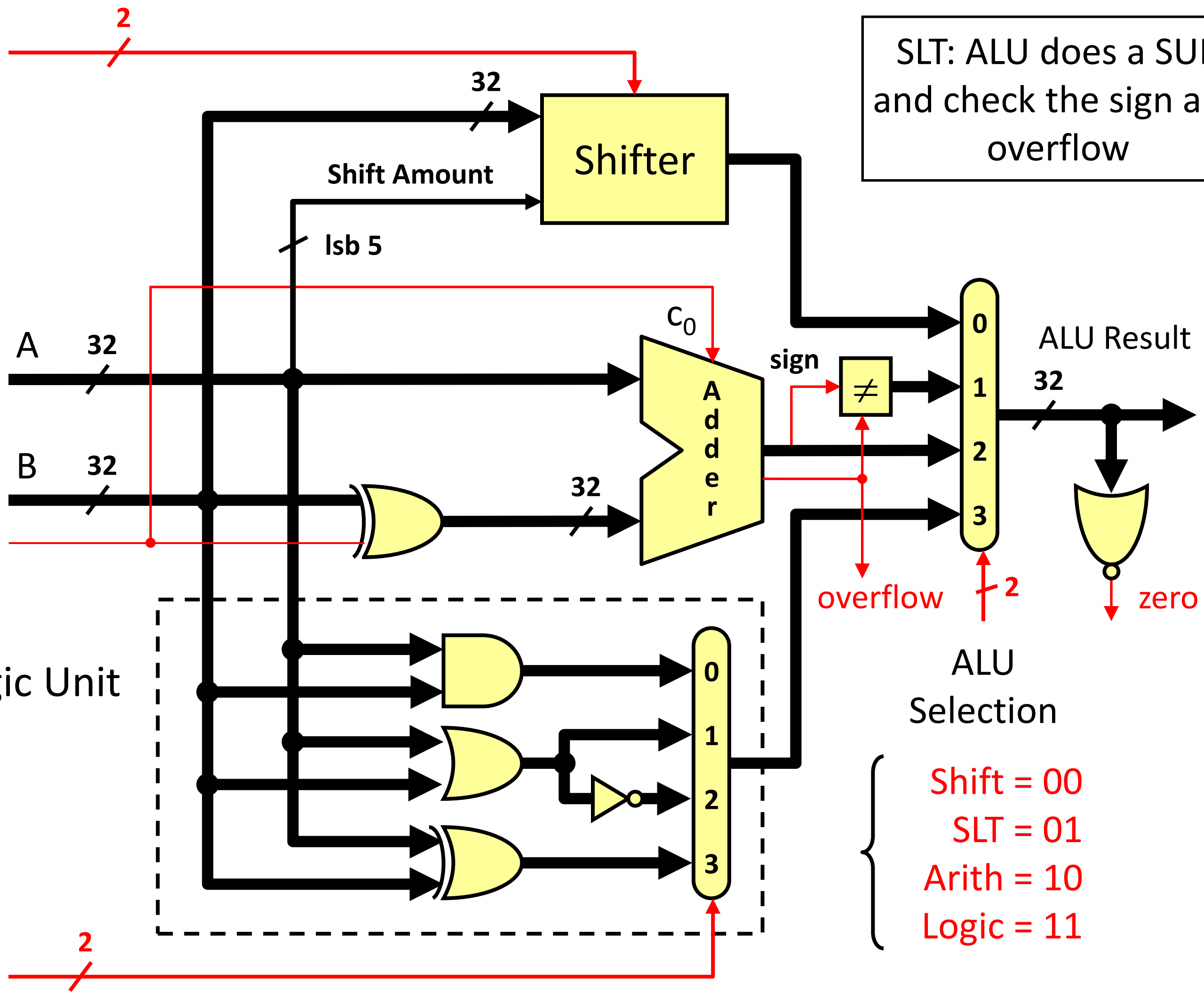
# Building a Multifunction ALU

Shift Operation
- None = 00
- SLL = 01
- SRL = 10
- SRA = 11

Arithmetic Operation
- ADD = 0
- SUB = 1

Logical Operation
- AND = 00
- OR = 01
- NOR = 10
- XOR = 11

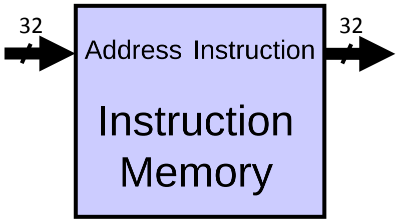
SLT: ALU does a SUB and check the sign and overflow

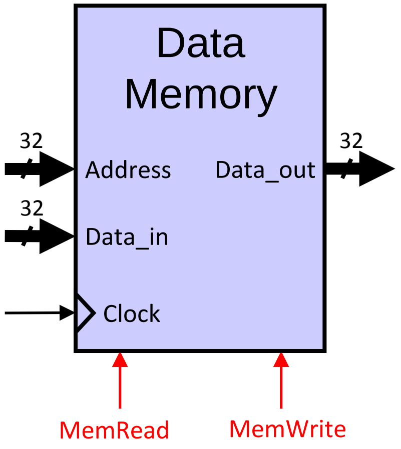


ALU Selection
- Shift = 00
- SLT = 01
- Arith = 10
- Logic = 11

## Instruction and Data Memories

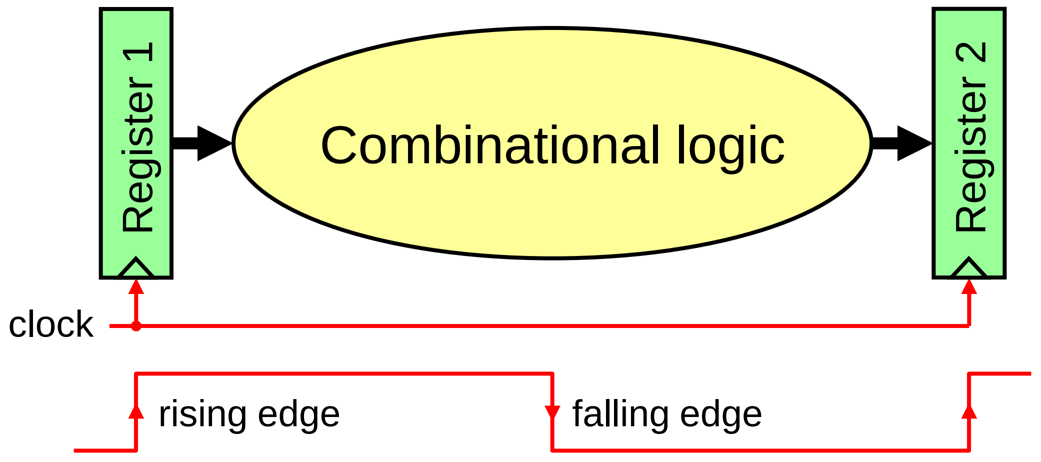
- Instruction memory needs only provide read access
  - Because datapath does not write instructions
  - Behaves as combinational logic for read
  - Address selects Instruction after access time
- Data Memory is used for load and store
  - MemRead: enables output on Data_out
    - Address selects the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

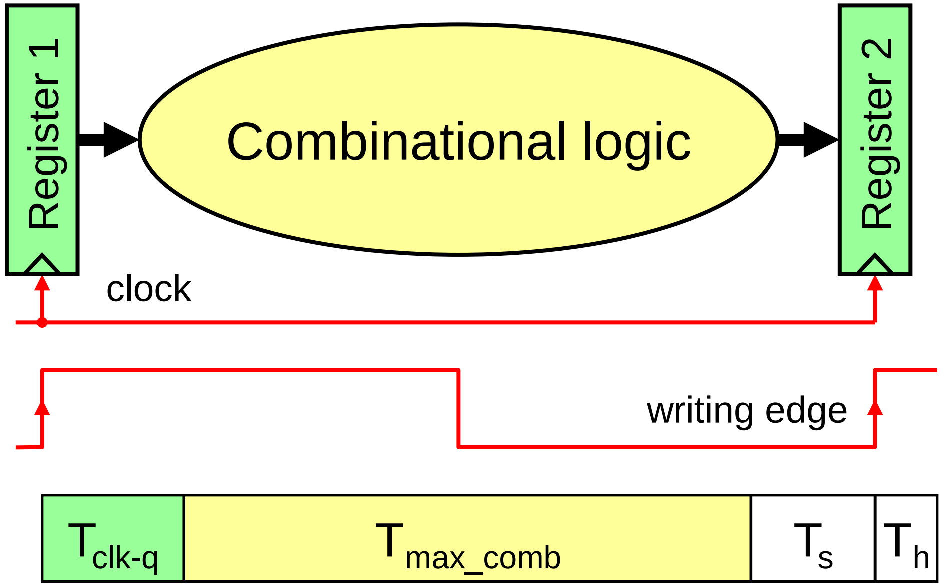
- Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- To ensure correctness, a clocking methodology defines when data can be written and read



- 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

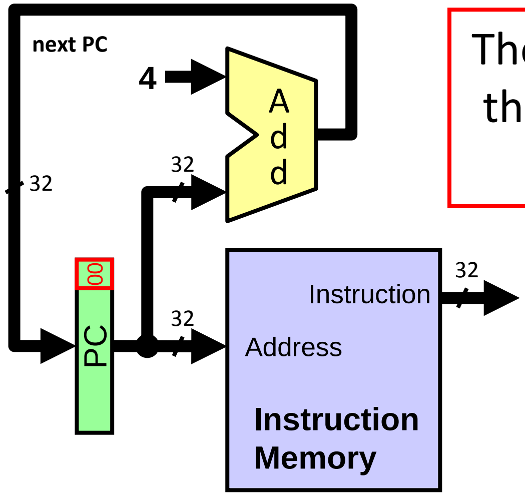
## Next . . .

- 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

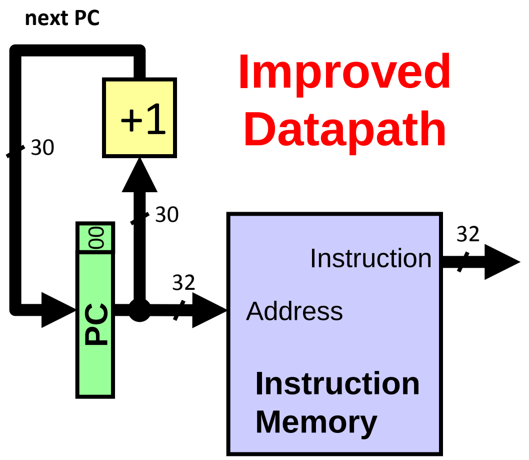
- We can now assemble the datapath from its components
- For instruction fetching, we need …
  - Program Counter (PC) register
  - Instruction Memory
  - Adder for incrementing PC

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

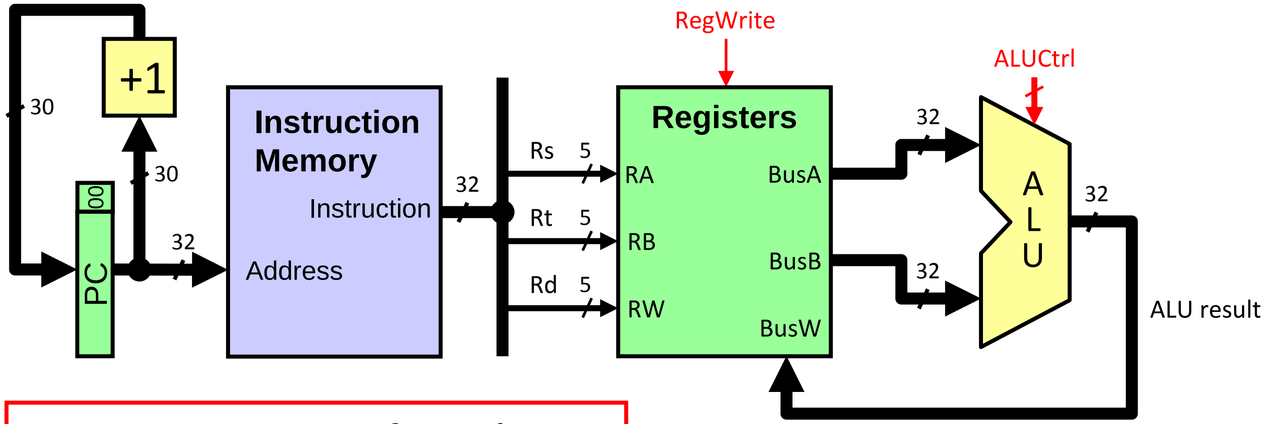
Datapath does not handle branch or jump instructions



**Improved Datapath**

# Datapath for R-type Instructions

| Op$^6$ | Rs$^5$ | Rt$^5$ | Rd$^5$ | sa$^5$ | funct$^6$ |
|---|---|---|---|---|---|



RA & RB come from the instruction's Rs & Rt fields

ALU inputs come from BusA & BusB

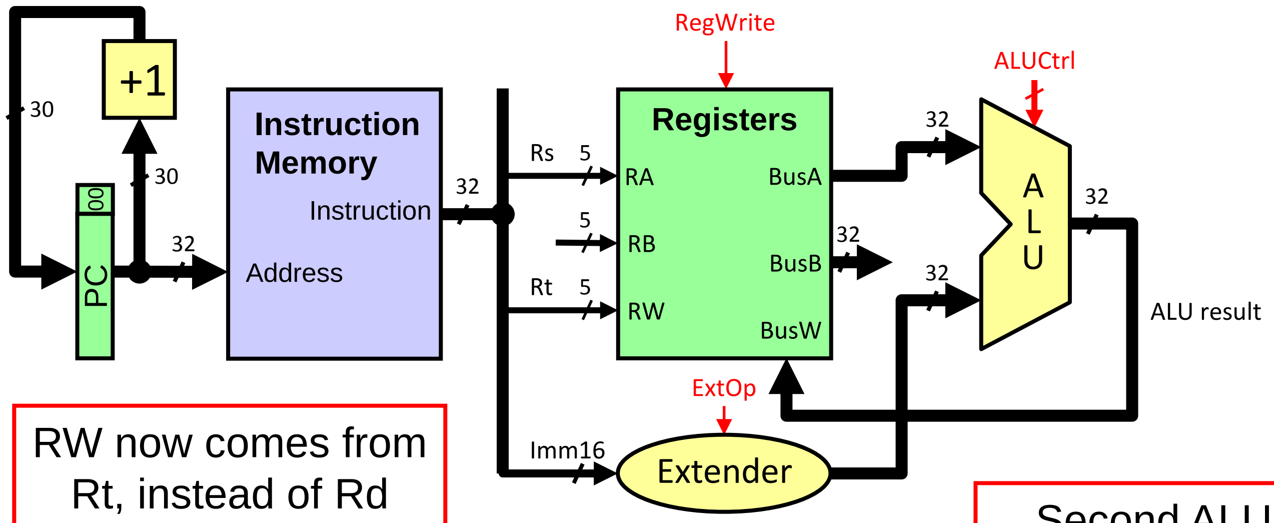
RW comes from the Rd field

ALU result is connected to BusW

- 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$^{16}$ |
|---|---|---|---|



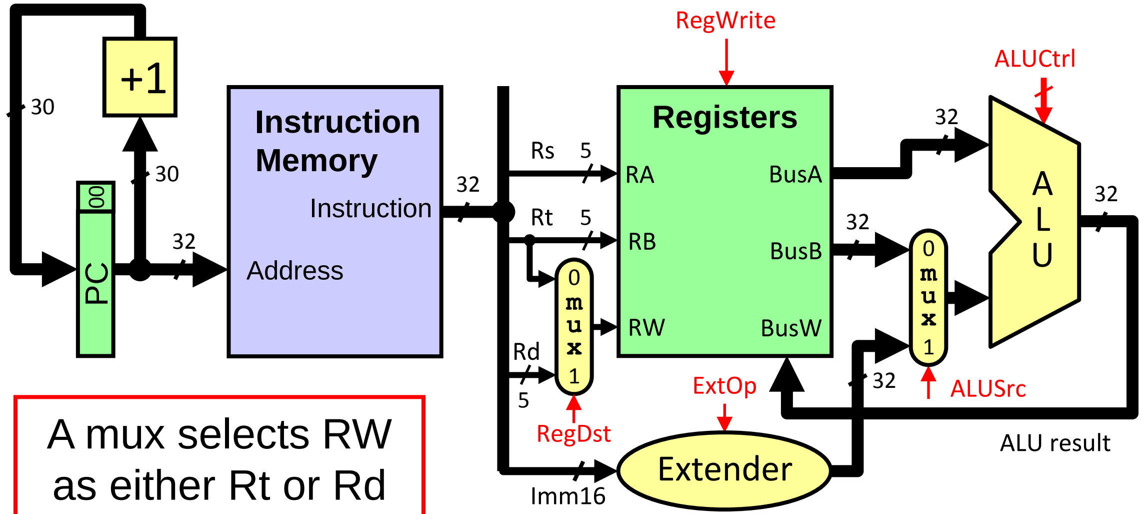
RW now comes from Rt, instead of Rd

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

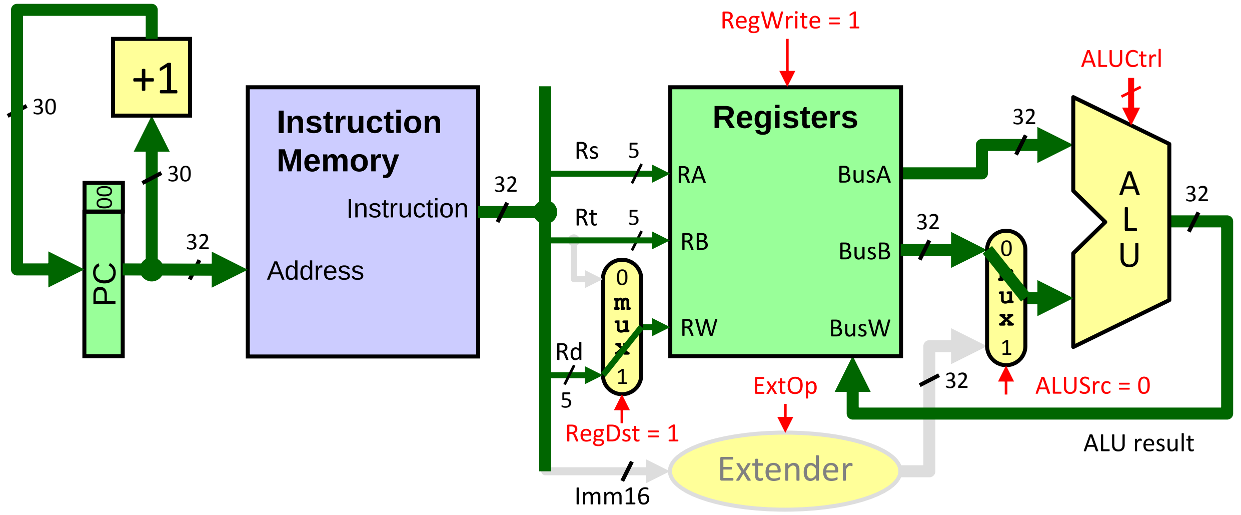


Another mux selects 2nd ALU input as either source register Rt data on BusB or the extended immediate

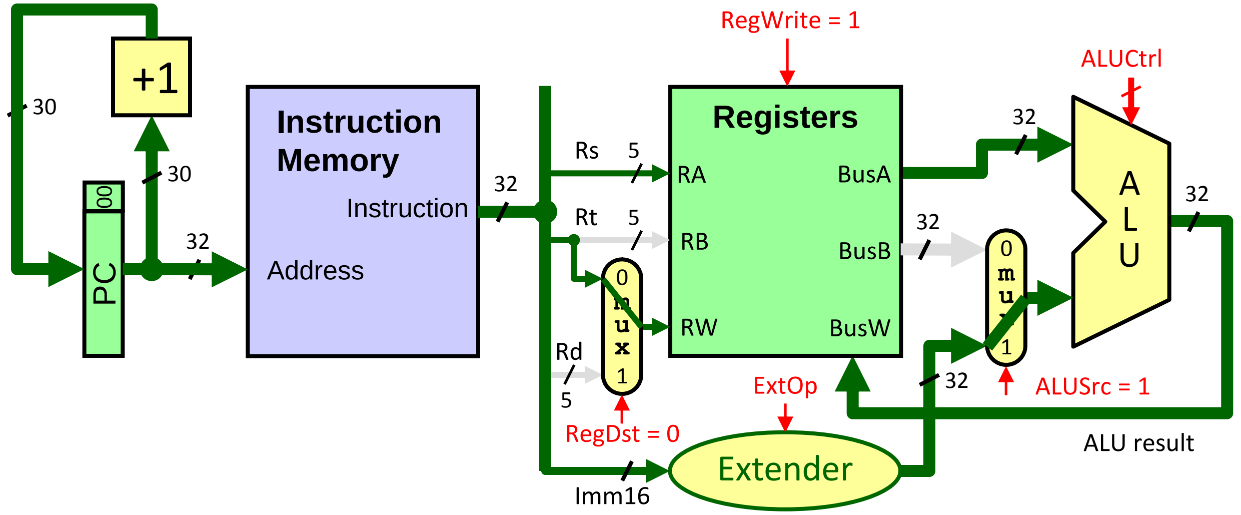
A mux selects RW as either Rt or Rd

- 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



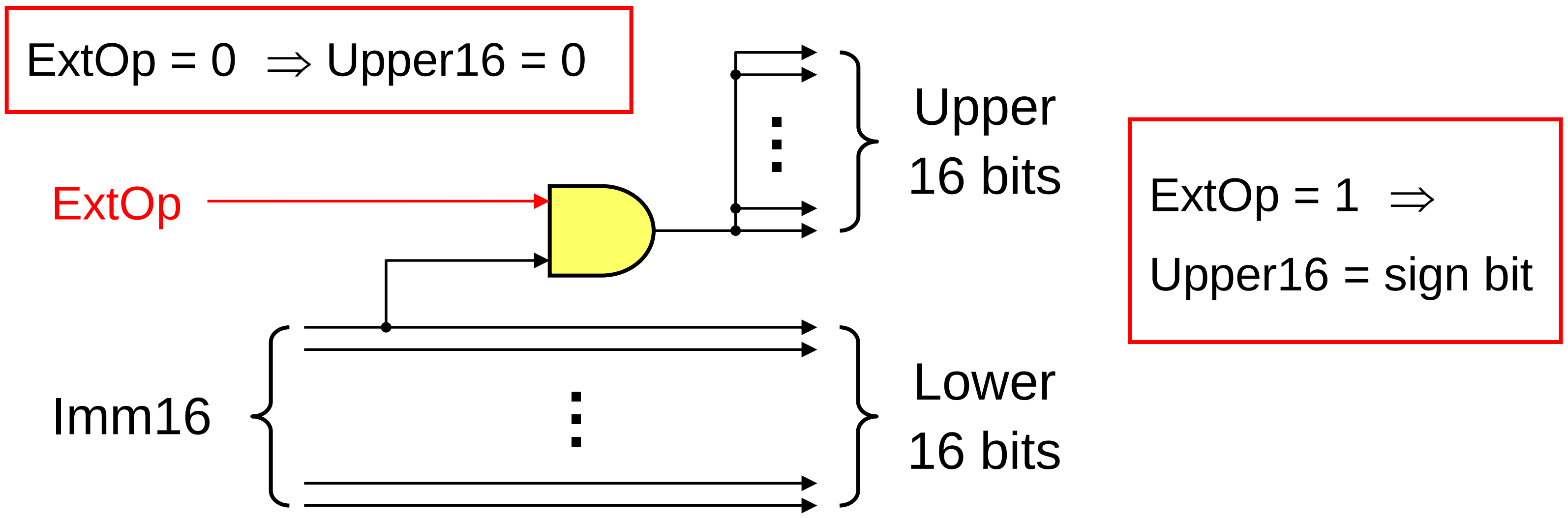
For R-type ALU instructions, RegDst is '1' to select Rd on RW and ALUSrc is '0' to select BusB as second ALU input. The active part of datapath is shown in **green**



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**

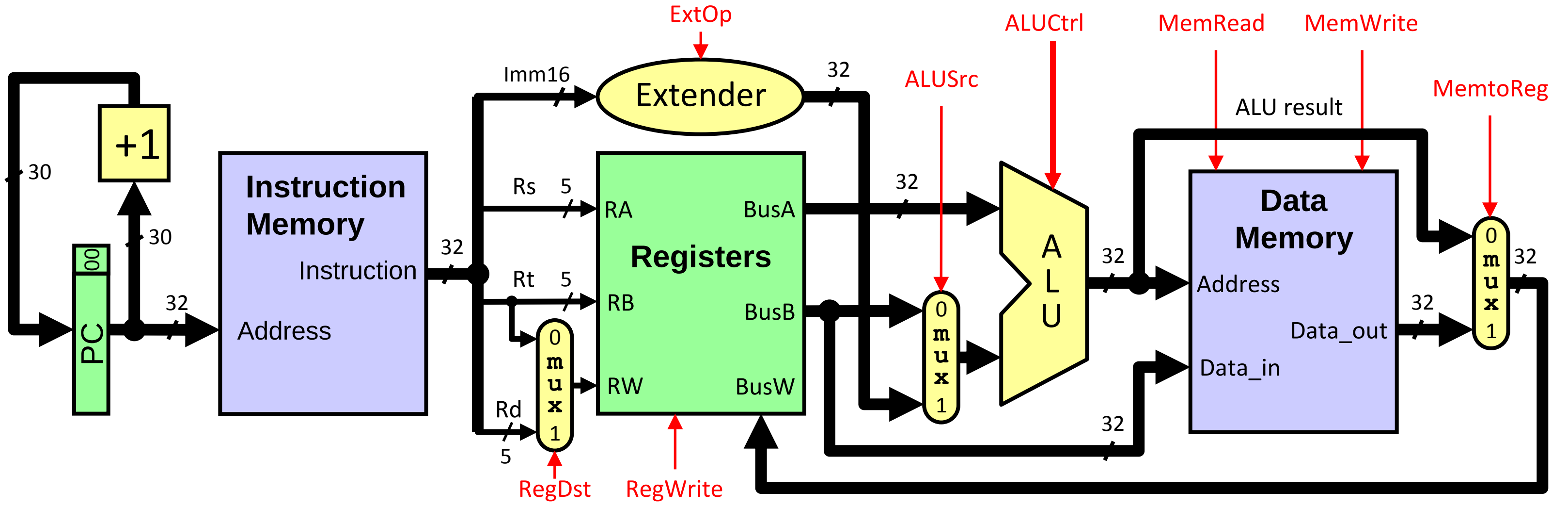
## Details of the Extender

- 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



# Adding Data Memory to Datapath

- A data memory is added for load and store instructions



ALU calculates data memory address

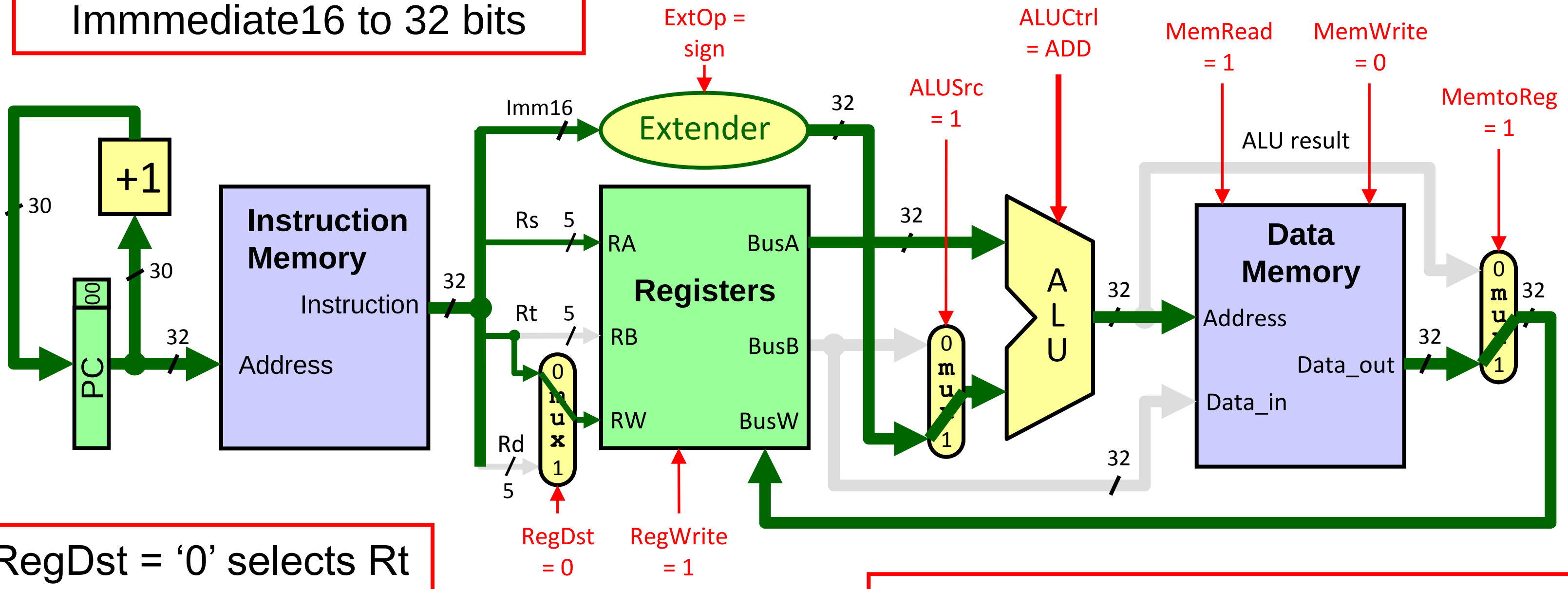
A 3rd mux selects data on BusW as either ALU result or memory data_out

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

ExtOp = ‘sign’ to sign-extend Immmediate16 to 32 bits



RegDst = ‘0’ selects Rt as destination register

MemRead = ‘1’ to read data memory

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

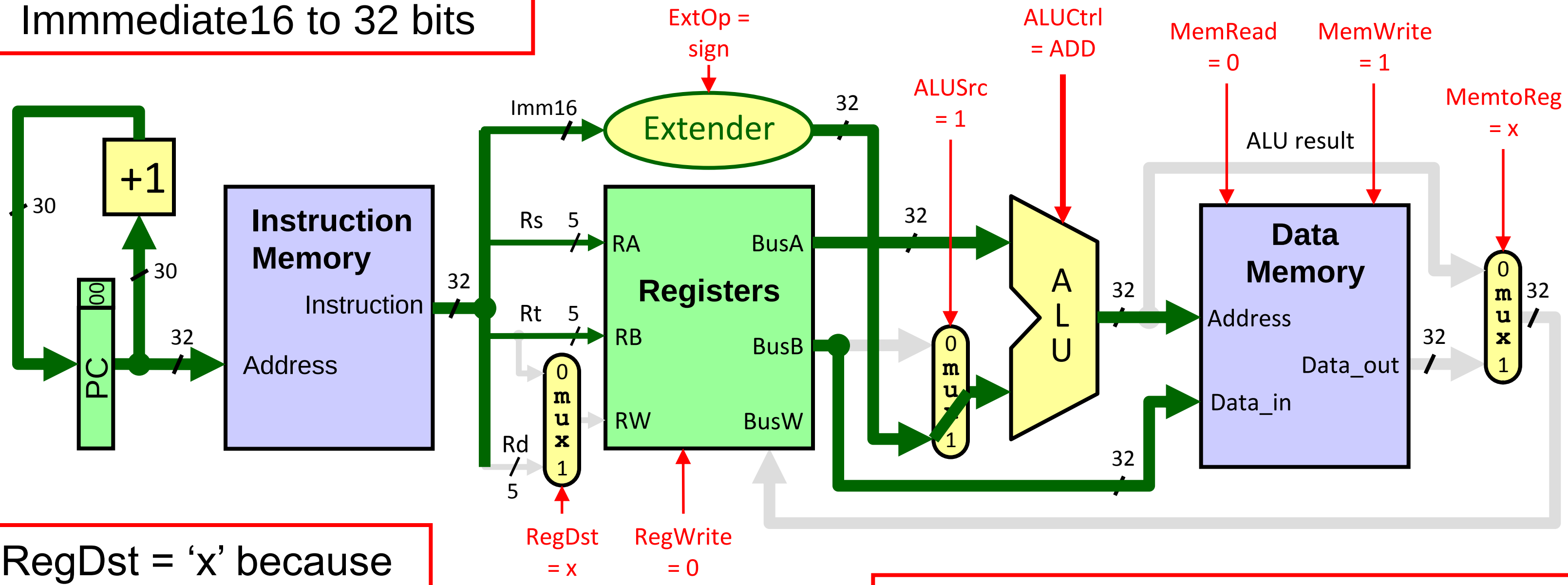
MemtoReg = ‘1’ places the data read from memory on BusW

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

RegWrite = ‘1’ to write the memory data on BusW to register Rt

# Controlling the Execution of Store

ExtOp = ‘sign’ to sign-extend Immmediate16 to 32 bits



RegDst = ‘x’ because no destination register

MemWrite = ‘1’ to write data memory

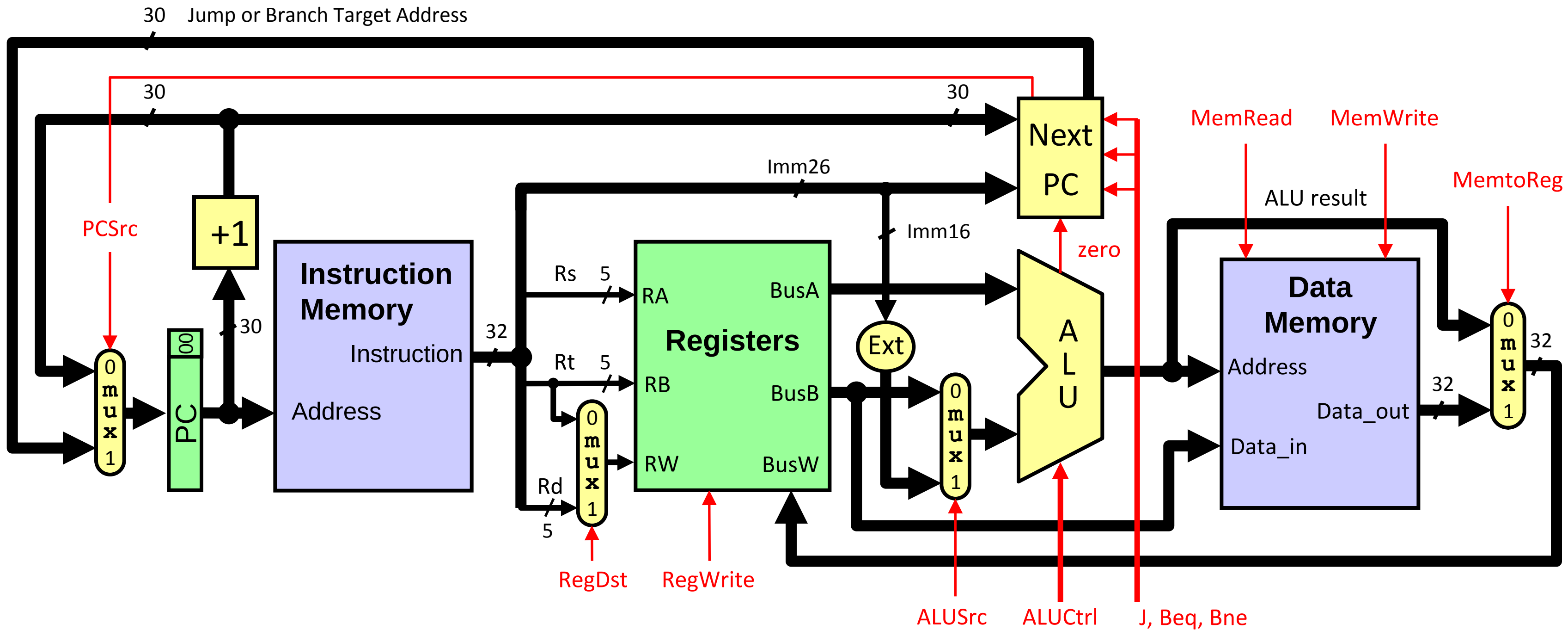
ALUSrc = ‘1’ to select the extended immediate as second ALU input

MemtoReg = ‘x’ because we don’t care what data is placed on BusW

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

RegWrite = ‘0’ because no register is written by the store instruction

# Adding Jump and Branch to Datapath

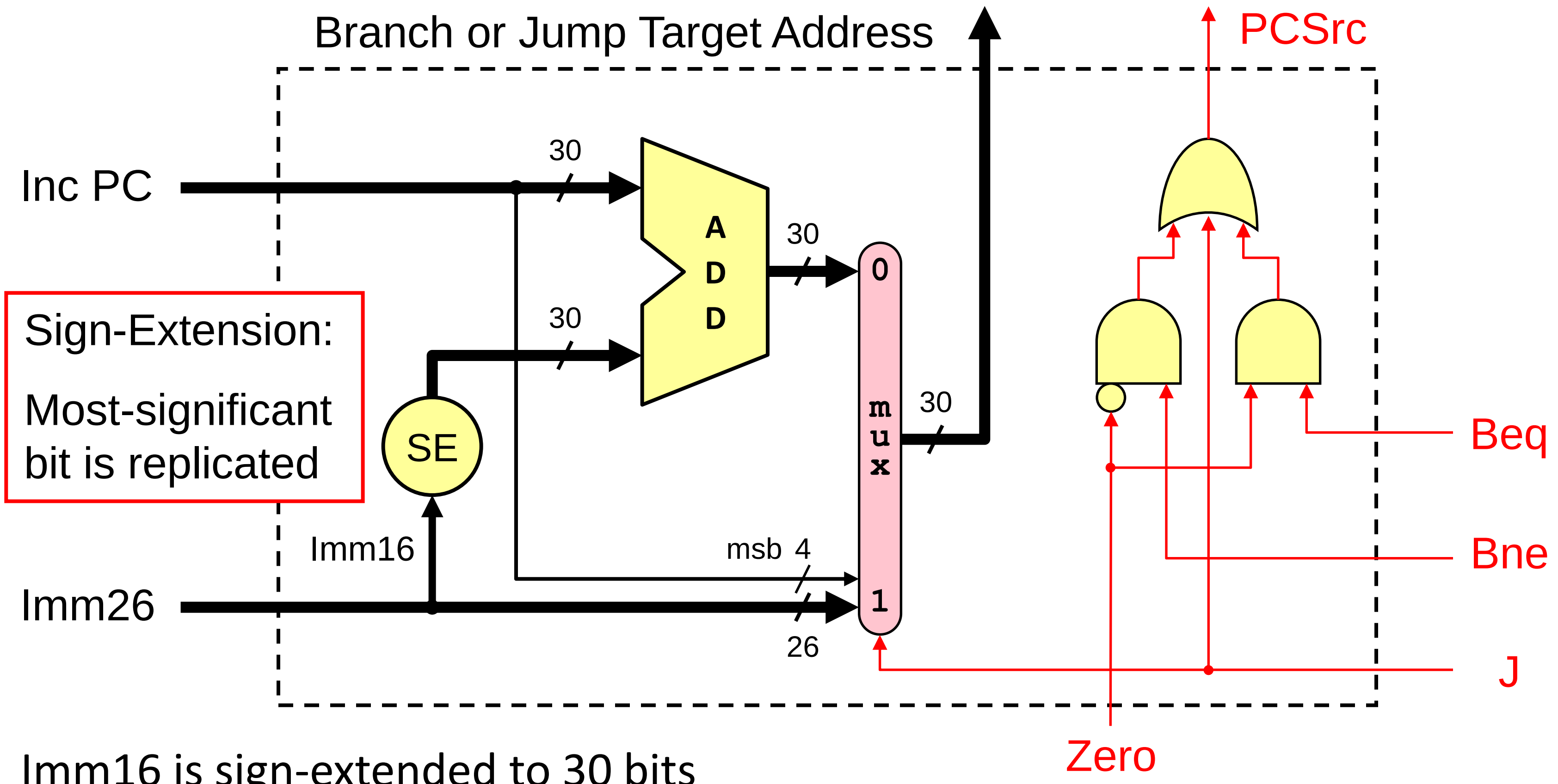


- Additional Control Signals
  - J, Beq, Bne for jump and branch instructions
  - Zero condition of the ALU is examined
  - PCSrc = 1 for Jump & taken Branch

Next PC computes jump or branch target instruction address

For Branch, ALU does a subtraction

## Details of Next PC

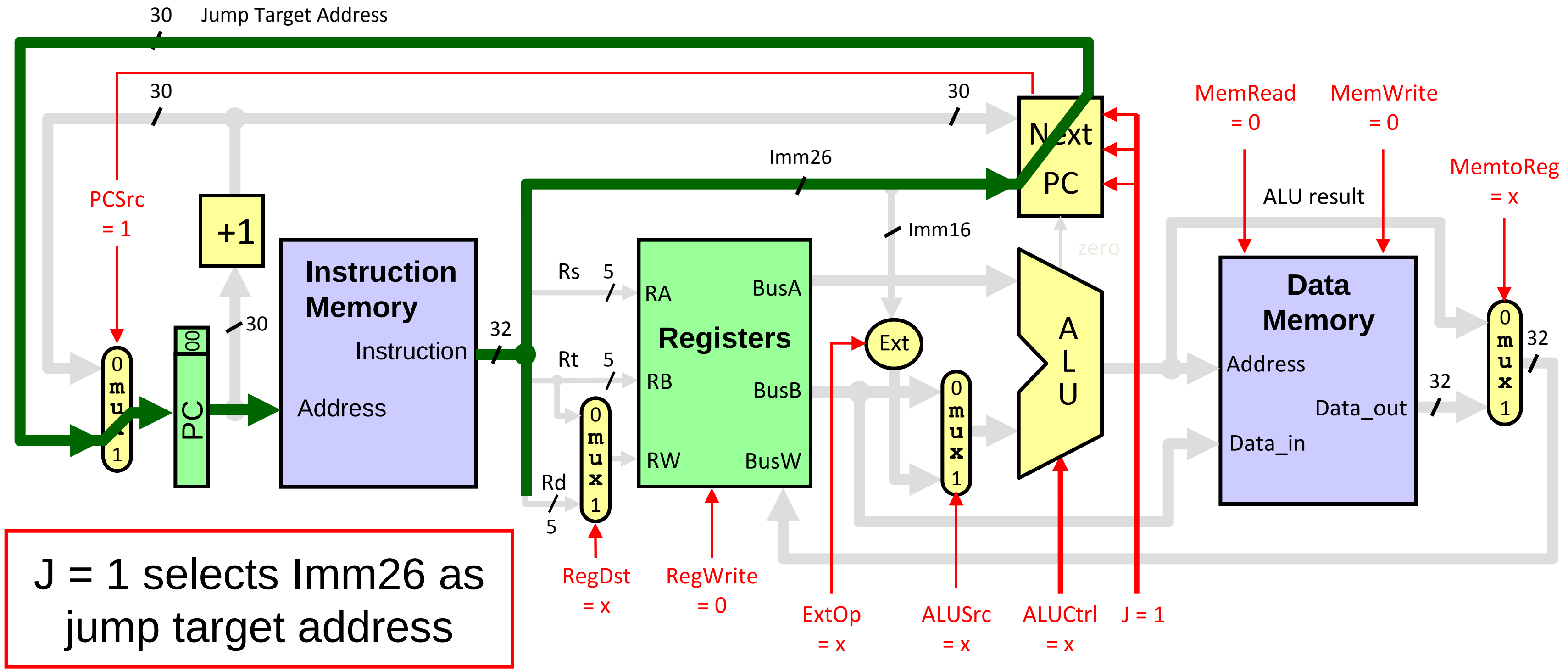


Imm16 is sign-extended to 30 bits

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 selects Imm26 as jump target address

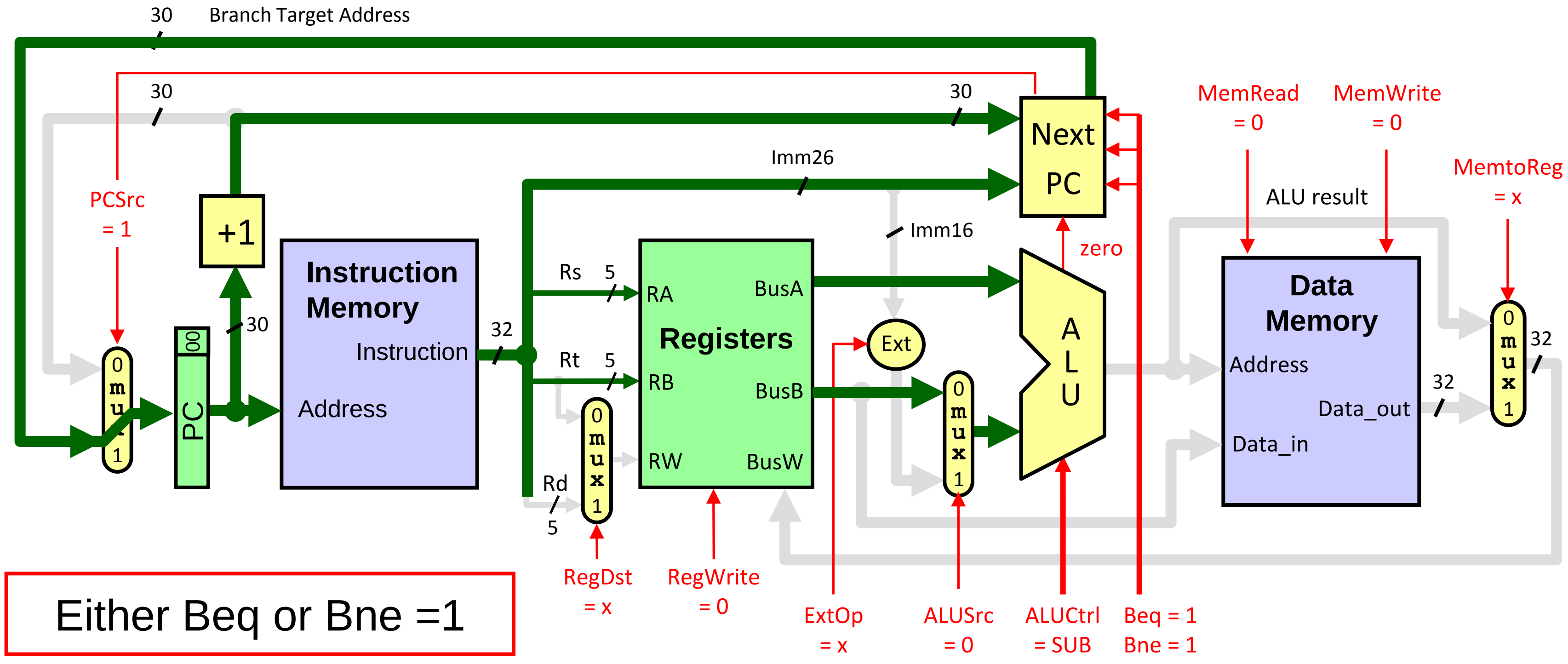
Upper 4 bits are from the incremented PC

PCSrc = 1 to select jump target address

MemRead, MemWrite & RegWrite are 0

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

# Controlling the Execution of Branch



Either Beq or Bne =1

Next PC outputs branch target address

ALUSrc = ‘0’ (2nd ALU input is BusB)
ALUCtrl = ‘SUB’ produces zero flag

Next PC logic determines PCSrc according to zero flag

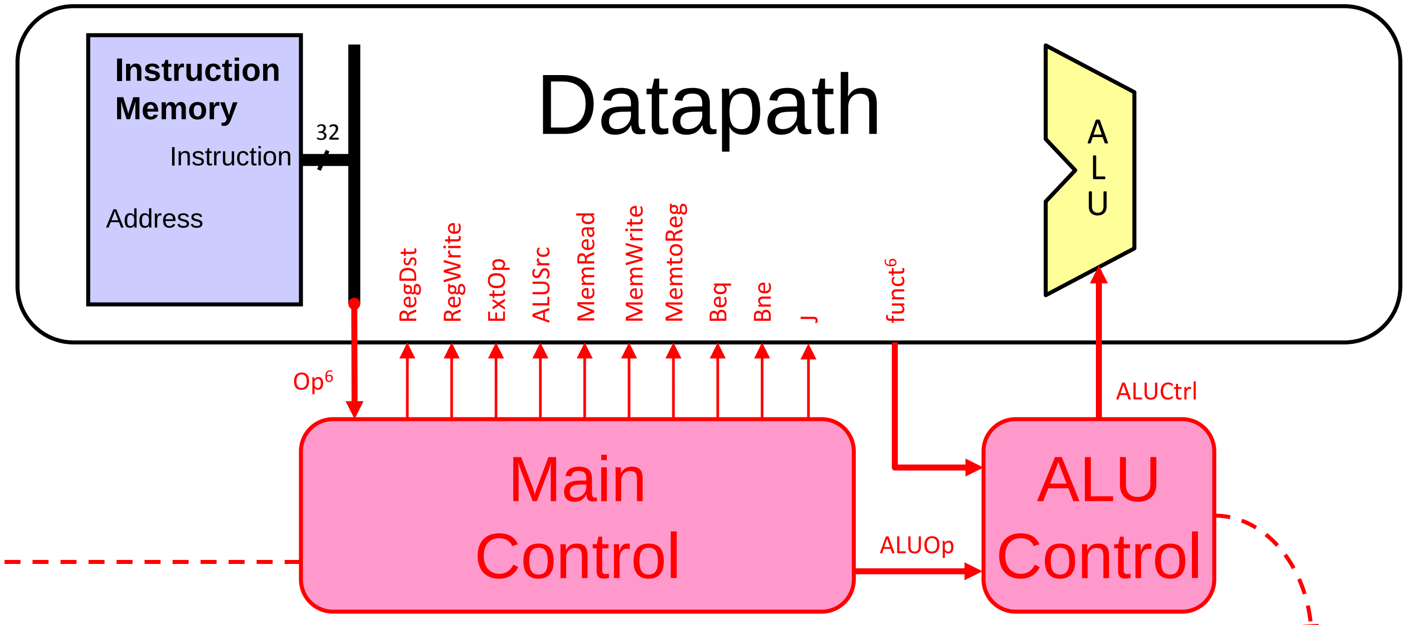
MemRead = MemWrite = RegWrite = 0

RegDst = ExtOp = MemtoReg = x

## 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



Input:

- 6-bit opcode field from instruction

Output:

- 10 control signals for datapath
- ALUOp for ALU Control

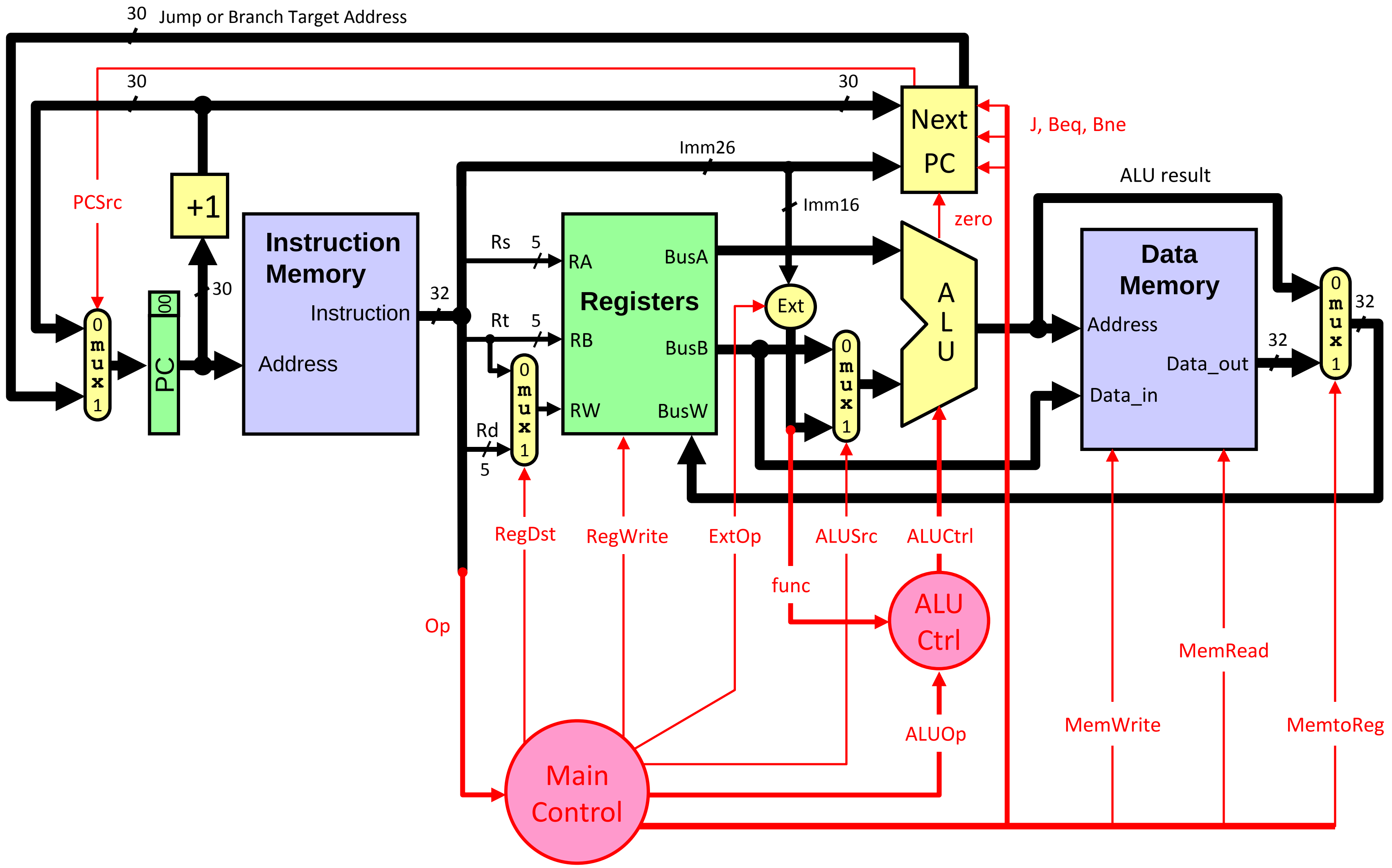
Input:

- ✧ 6-bit function field from instruction
- ✧ ALUOp from main 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<br>Data_out ← Memory[address] |
| MemWrite | None | Data memory is written<br>Memory[address] ← Data_in |
| MemtoReg | BusW = ALU result | BusW = Data_out from Memory |
| Beq, Bne | PC ← PC + 4 | PC ← Branch target address<br>If branch is taken |
| J | PC ← PC + 4 | PC ← Jump target address |
| ALUOp | This multi-bit signal specifies the ALU operation as a function of the opcode | |

# Main Control Signal Values

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

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

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

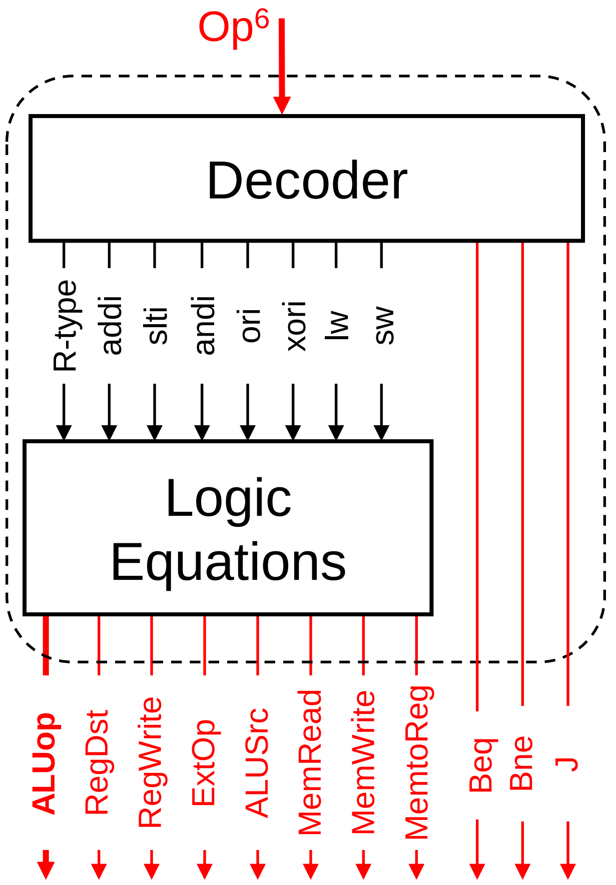
$\text{ExtOp} \Leftarrow \overline{(\text{andi} + \text{ori} + \text{xori})}$

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

$\text{MemRead} \Leftarrow \text{lw}$

$\text{MemWrite} \Leftarrow \text{sw}$

$\text{MemtoReg} \Leftarrow \text{lw}$



# ALU Control Truth Table

| Op$^6$ | ALU Control | | | 4-bit Encoding |
|---|---|---|---|---|
| | ALUOp | funct$^6$ | ALUCtrl | |
| R-type | R-type | add | ADD | 0000 |
| R-type | R-type | sub | SUB | 0010 |
| R-type | R-type | and | AND | 0100 |
| R-type | R-type | or | OR | 0101 |
| R-type | R-type | xor | XOR | 0110 |
| R-type | R-type | slt | SLT | 1010 |
| addi | ADD | x | ADD | 0000 |
| slti | SLT | x | SLT | 1010 |
| andi | AND | x | AND | 0100 |
| ori | OR | x | OR | 0101 |
| xori | XOR | x | XOR | 0110 |
| lw | ADD | x | ADD | 0000 |
| sw | ADD | x | ADD | 0000 |
| beq | SUB | x | SUB | 0010 |
| bne | SUB | x | SUB | 0010 |
| j | x | x | x | x |

The 4-bit encoding for ALUctrl is chosen here to be equal to the last 4 bits of the function field

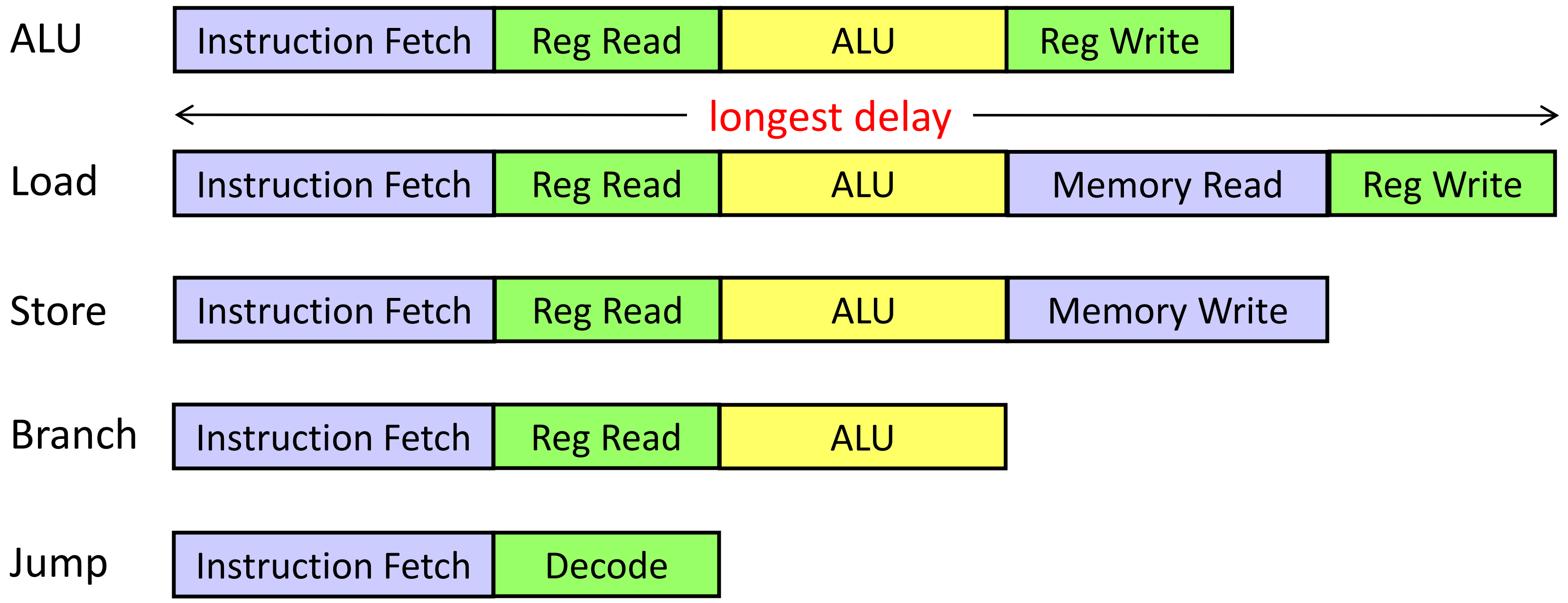
Other binary encodings are also possible. The idea is to choose a binary encoding that will minimize the logic for ALU Control

## 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



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