# Single-cycle Processor

## (Computer Organization: Chapter 4)

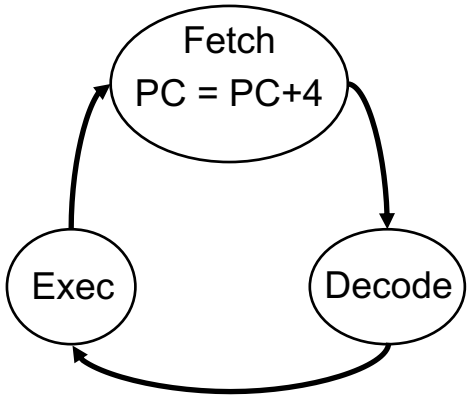
Yanyan Shen

Department of Computer Science and Engineering

# The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`

- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



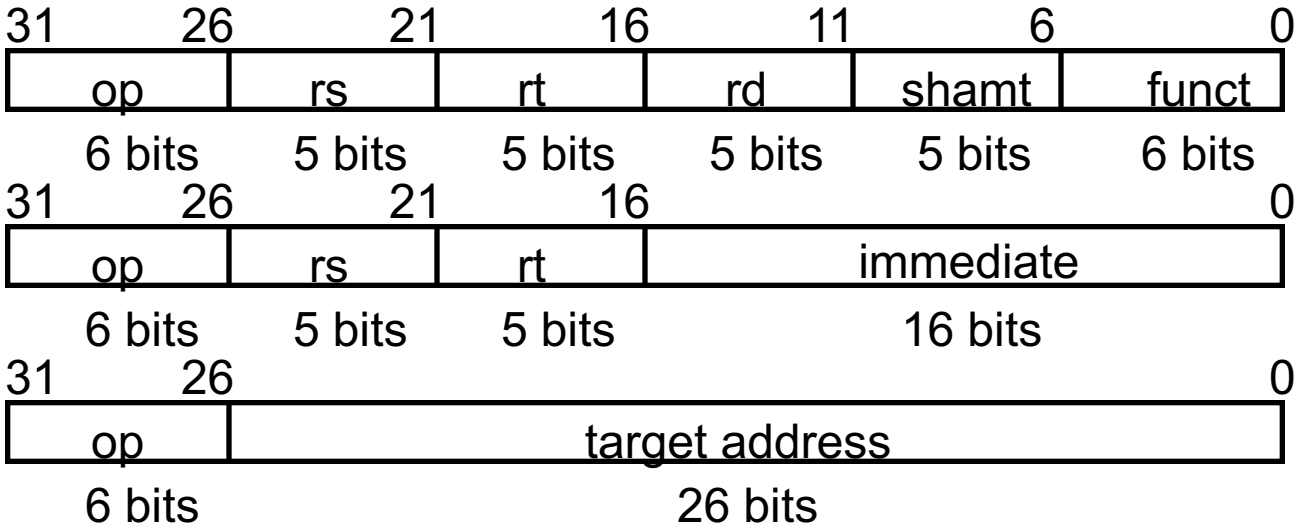

- All instructions (except `j`) use the ALU after reading the registers
  - How memory-reference? arithmetic? control flow?

# How to Design 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 element for ISA registers
    - possibly more
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

# The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type
  - I-type
  - J-type

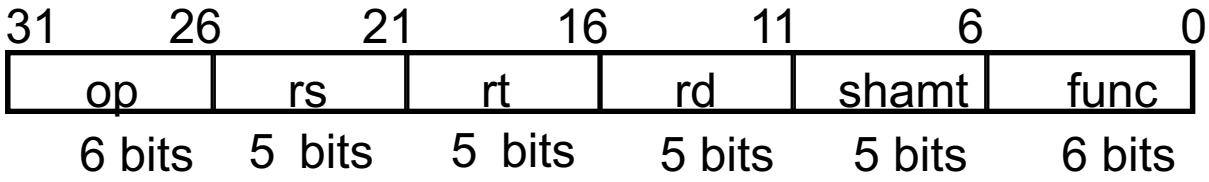


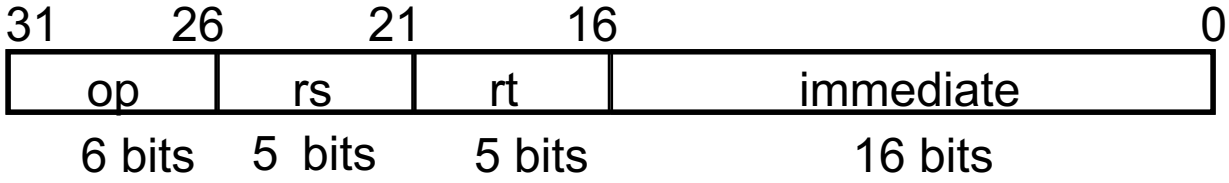

- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

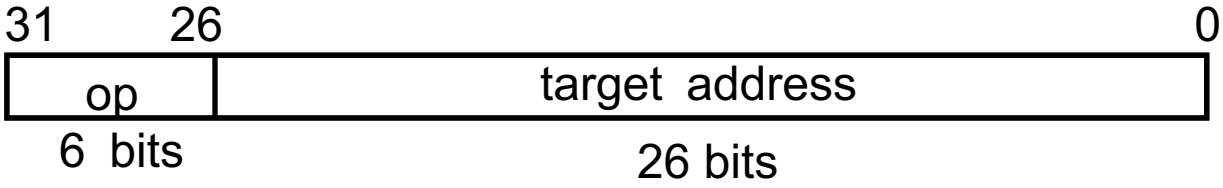
# Focus on a Subset of MIPS Instructions

7 Instructions

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



# Aside: Logical Register Transfers

- RTL gives the **meaning** of the instructions
- All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[ PC ]

op | rs | rt | Imm16 = MEM[ PC ]

| inst | Register Transfers | |
|---|---|---|
| ADDU | R[rd] <– R[rs] + R[rt]; | PC <– PC + 4 |
| SUBU | R[rd] <– R[rs] – R[rt]; | PC <– PC + 4 |
| ORI | R[rt] <– R[rs] \| zero_ext(Imm16); | PC <– PC + 4 |
| LOAD | R[rt] <– MEM[ R[rs] + sign_ext(Imm16)]; | PC <– PC + 4 |
| STORE | MEM[ R[rs] + sign_ext(Imm16) ] <– R[rt]; | PC <– PC + 4 |

BEQ if ( R[rs] == R[rt] ) then PC <– PC + 4 +sign_ext(Imm16)] || 00
else PC <– PC + 4

# Step 1: Requirements of the Instruction Set

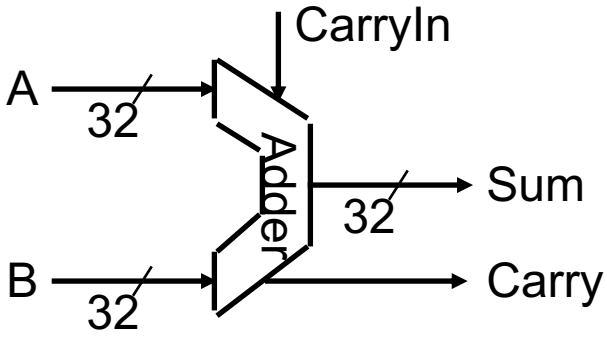
- Memory (MEM)
  - Instructions & data
- Registers (R: 32 x 32)
  - Read *rs*
  - Read *rt*
  - Write *rt* or *rd*
- PC
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
- Add 4 (+ maybe extended immediate) to PC

# Step 2: Components of the Datapath

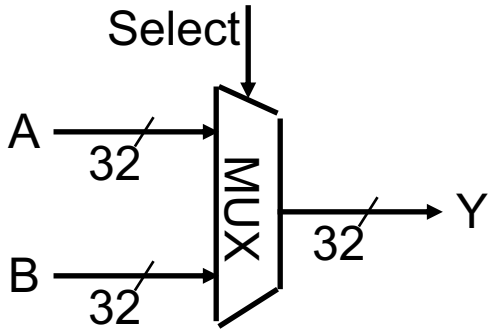
- Combinational Elements
- Storage Elements
  - Clocking methodology

# Combinational Logic Elements

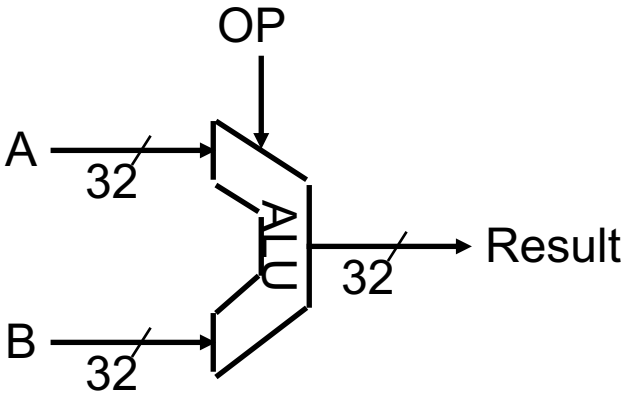
- Adder




- MUX

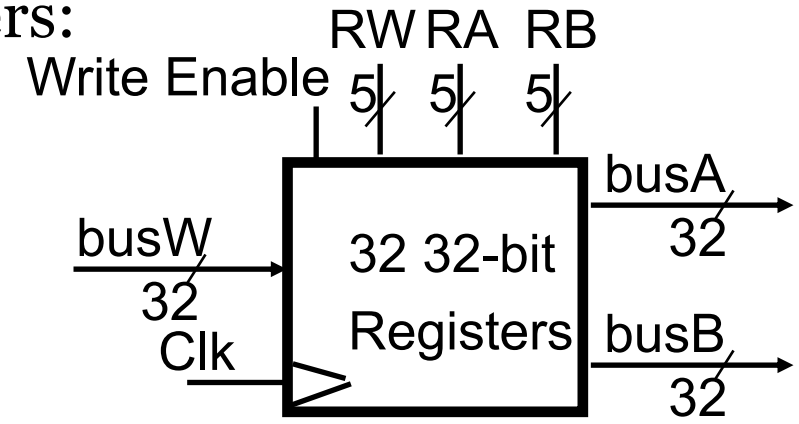



- ALU



# Storage Element: Register File

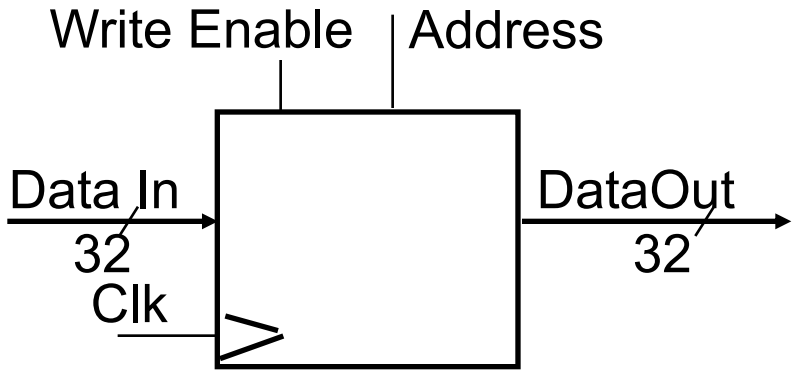
- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW




- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1

- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after "access time."

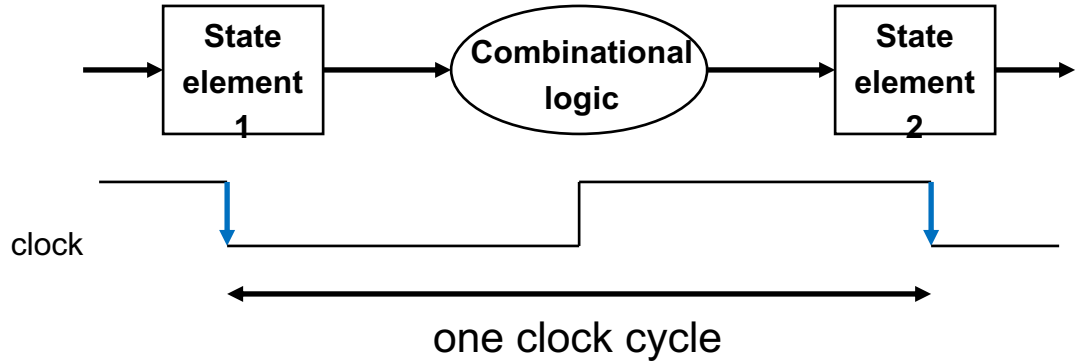
# Storage Element: Idealized Memory




- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

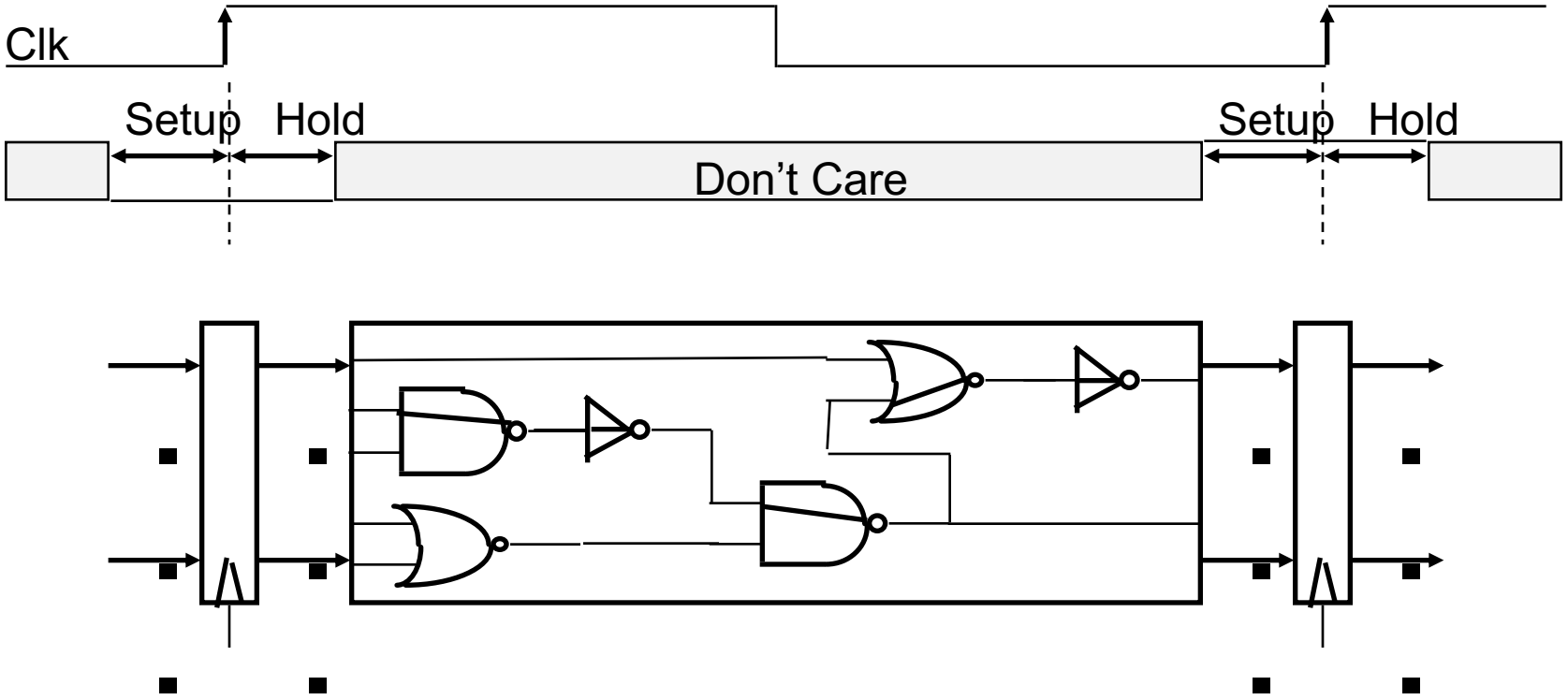
# Aside: Clocking Methodologies

- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements - a memory element such as a register
  - Edge-triggered – all state changes occur on a clock edge
- Typical execution
  - read contents of state elements -> send values through combinational logic -> write results to one or more state elements




- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

# Aside: Clocking Methodologies

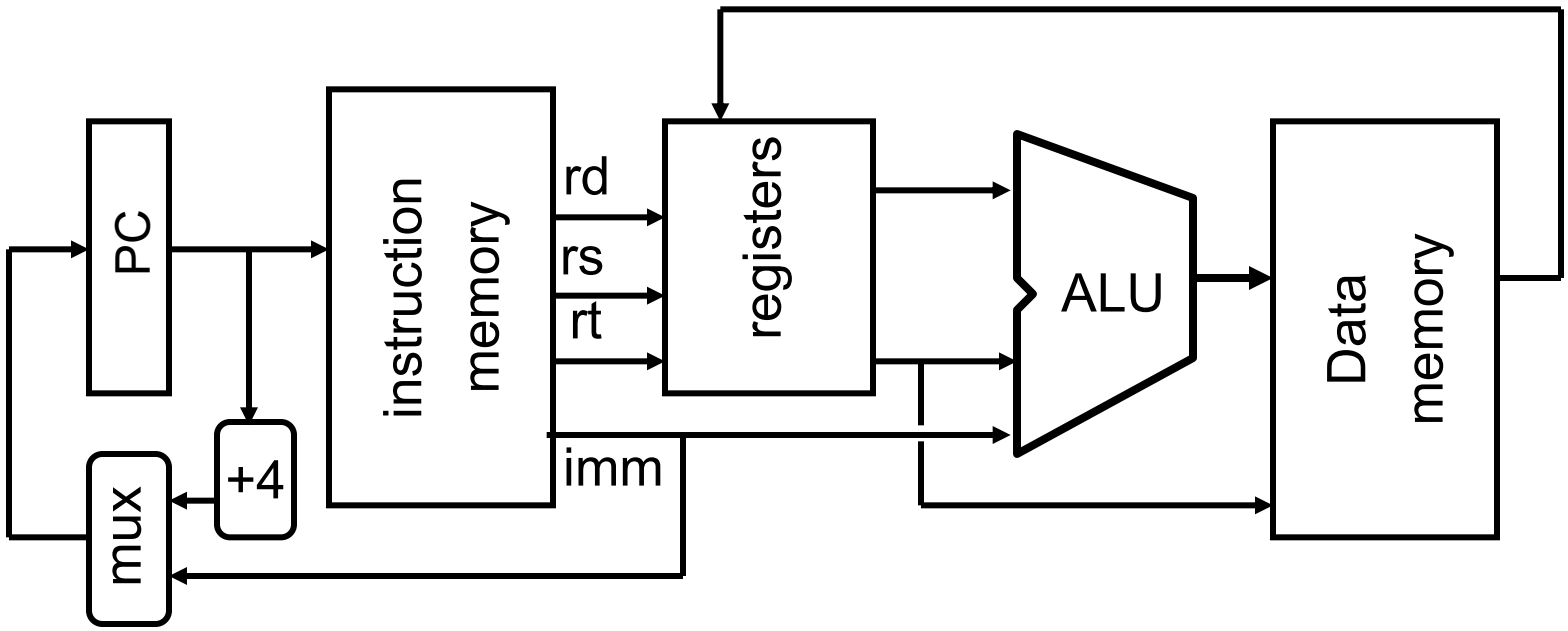



- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

# Step 3: Assemble DataPath meeting our requirements

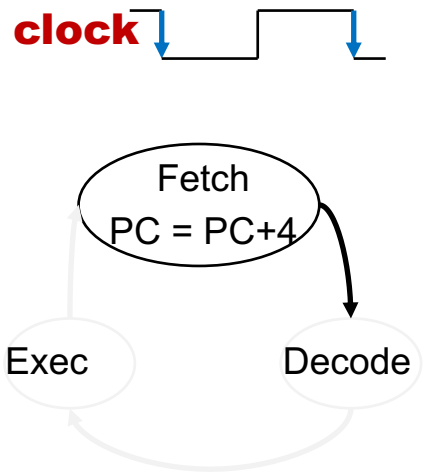
- Register Transfer Requirements
  ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

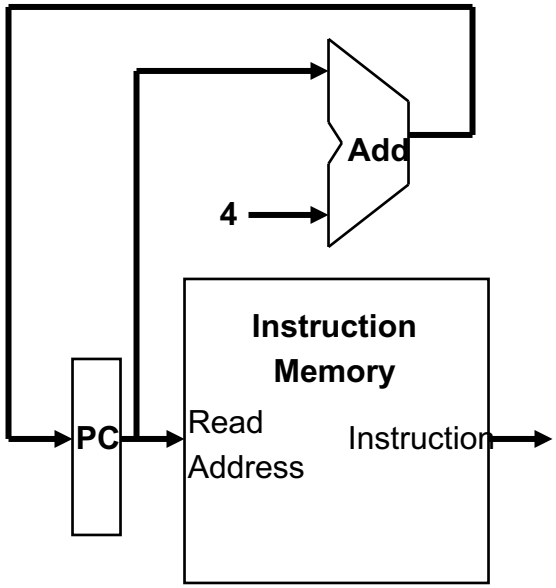
# Generic Steps of Datapath



# Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory M[PC]
  - updating the PC value to be the address of the next (sequential) instruction PC ← PC + 4

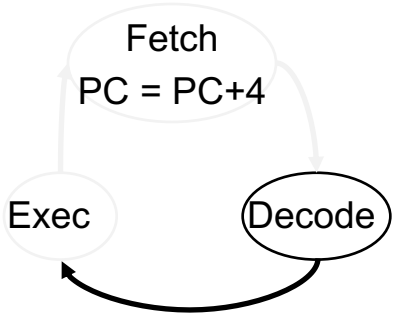


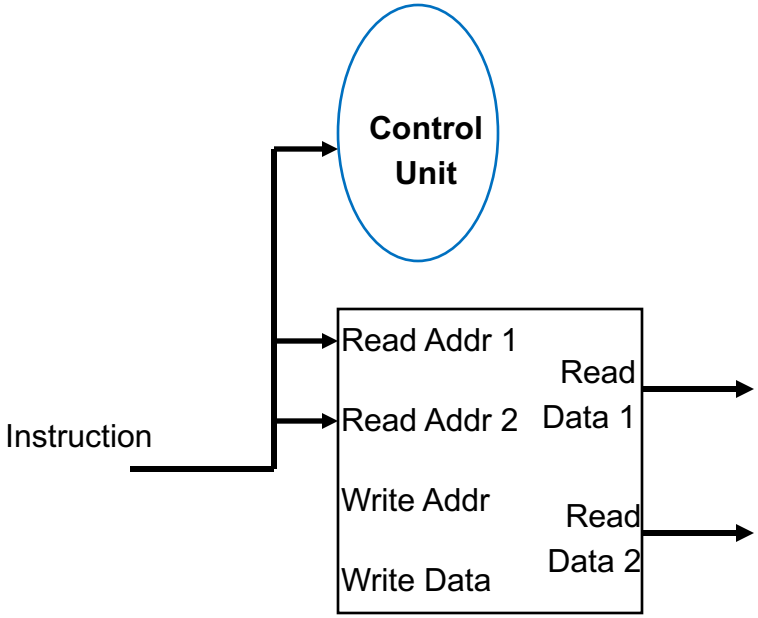




- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

# Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



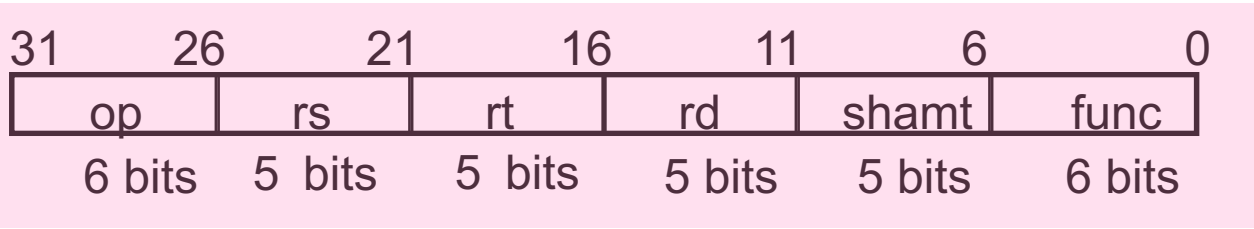

and

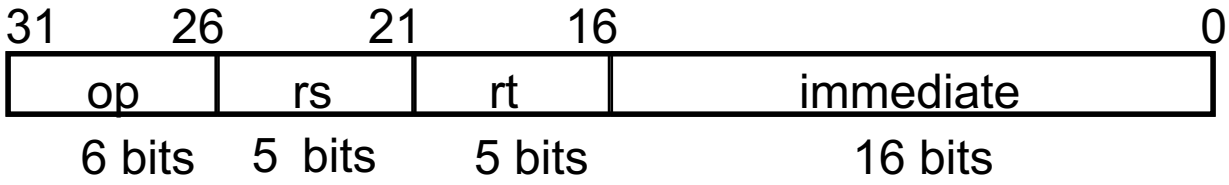
- reading two values from the Register File
  - Register File addresses are contained in the instruction

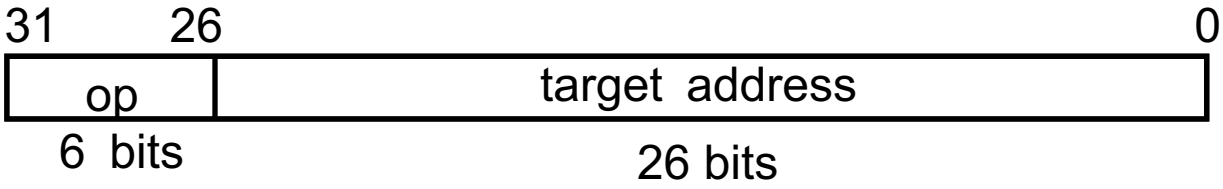
# Executing R-type Instructions

7 Instructions

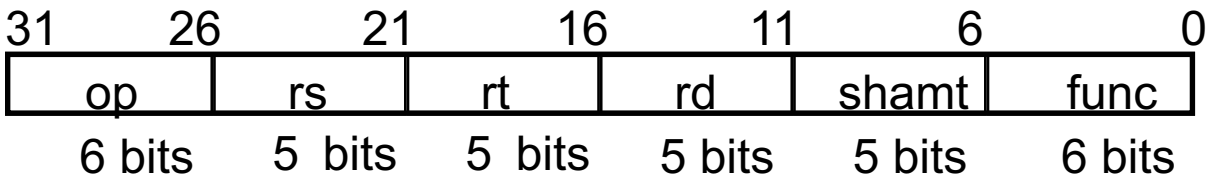
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



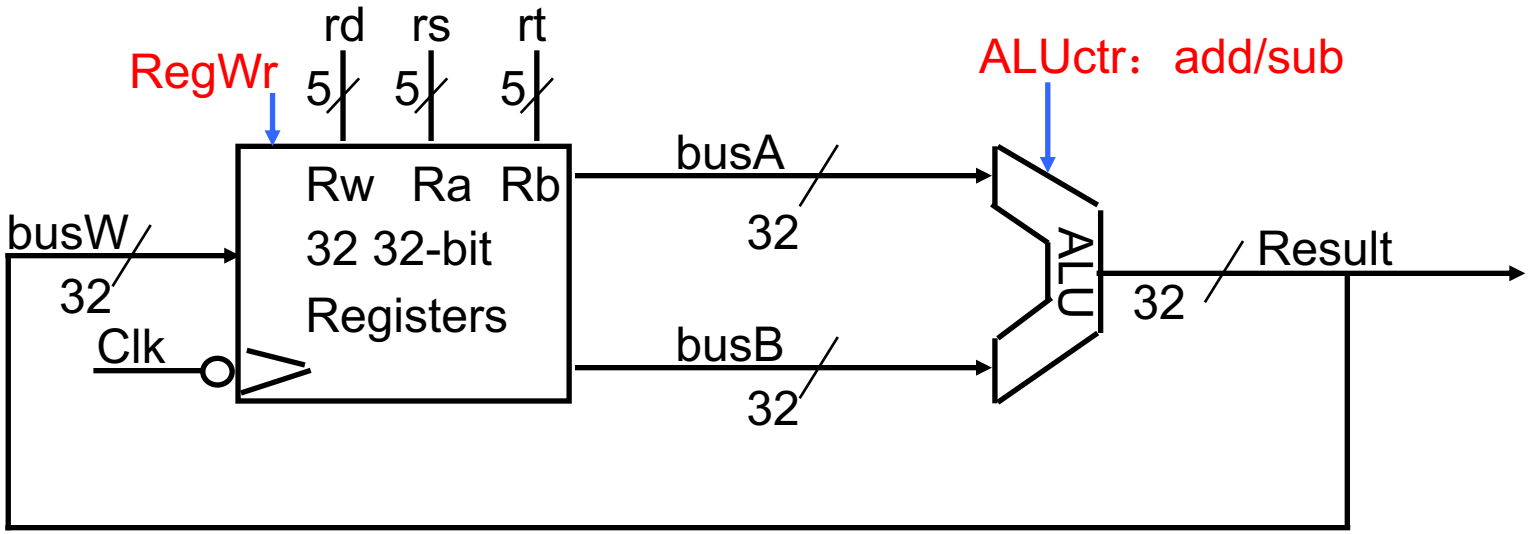



# Datapath of RR（R-type）




- RTL：R[rd] ← R[rs] op R[rt]　　Example: add　rd, rs, rt




Ra, Rb, Rw correspond to rs, rt, rd

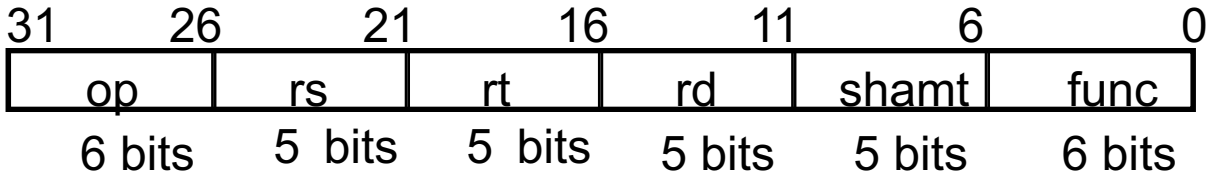
ALUctr,RegWr:  control signal

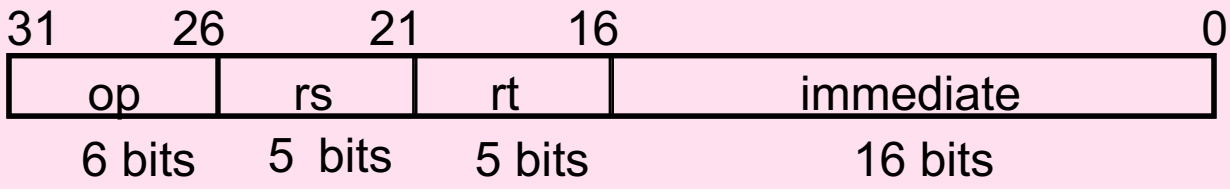
What are controls signals for “add rd, rs, rt” ?

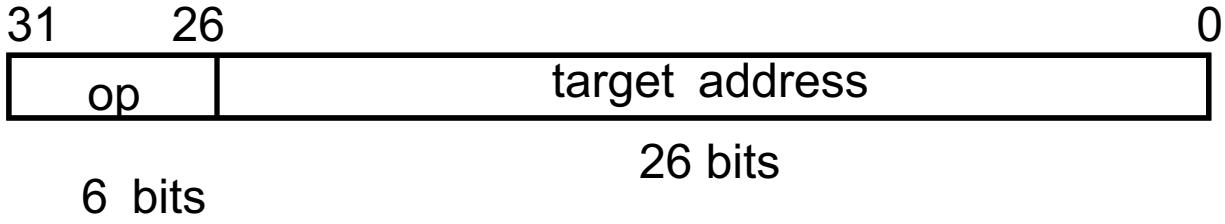
ALUctr=add，RegWr=1

# I-type instruction（ori）

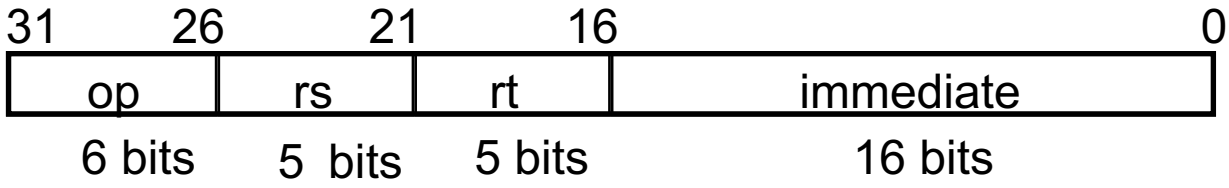
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



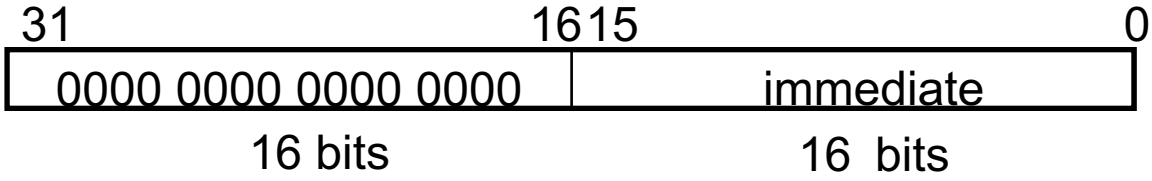



# RTL: The OR Immediate Instruction



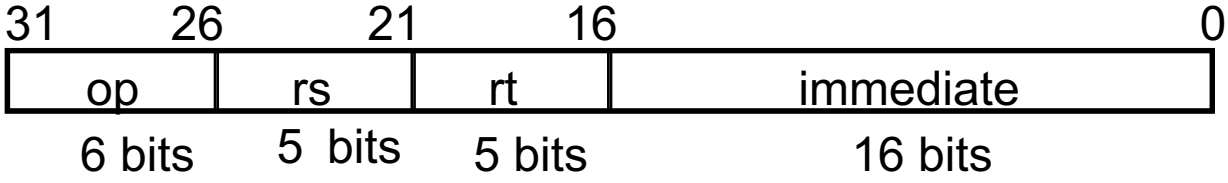

- ori rt, rs, imm16
  - M[PC] Instruction Fetech
  - R[rt] ← R[rs] or ZeroExt(imm16)
    zero extension of 16 bit constant or R[rs]
  - PC ← PC + 4 update PC

Zero extension ZeroExt(imm16)

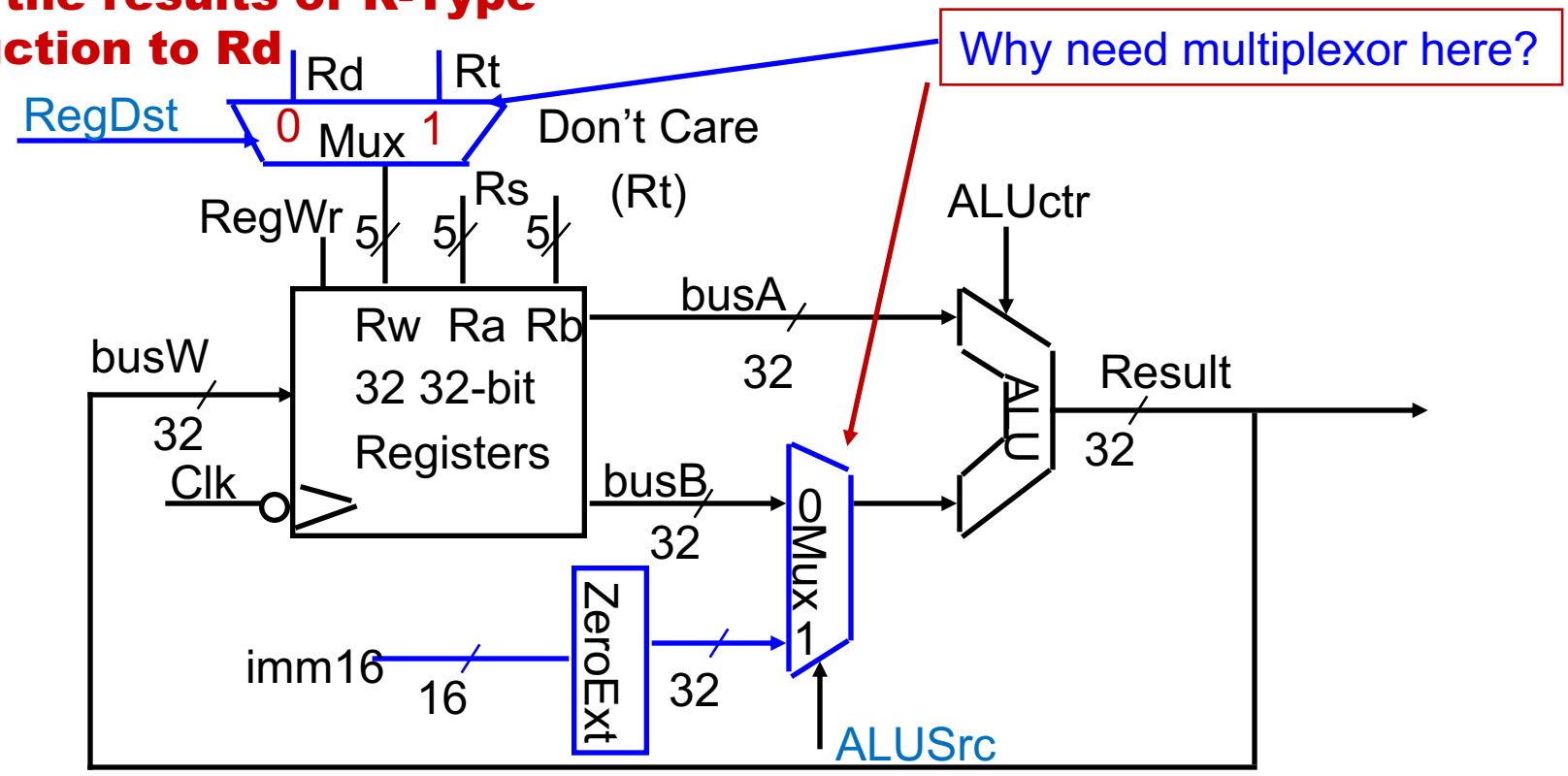


# Datapath of Immediate Instruction

- R[rt] ← R[rs] op ZeroExt[imm16]]  Example: ori rt, rs, imm16




**Write the results of R-Type instruction to Rd**

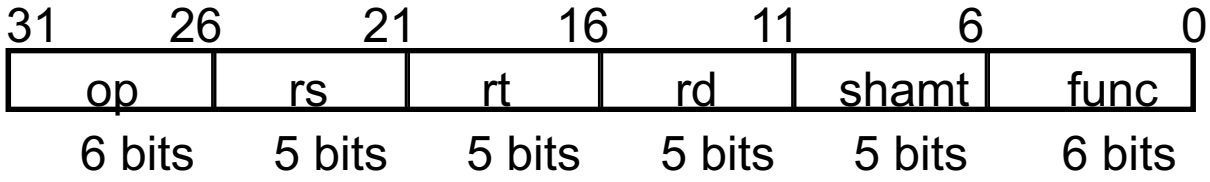


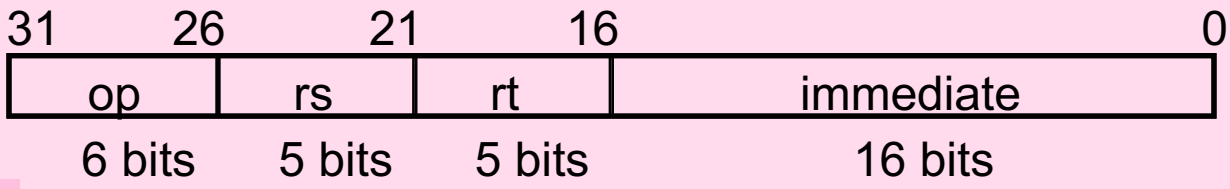

**Ori control signals: RegDst=? ; RegWr=? ; ALUctr=? ; ALUSrc=?**

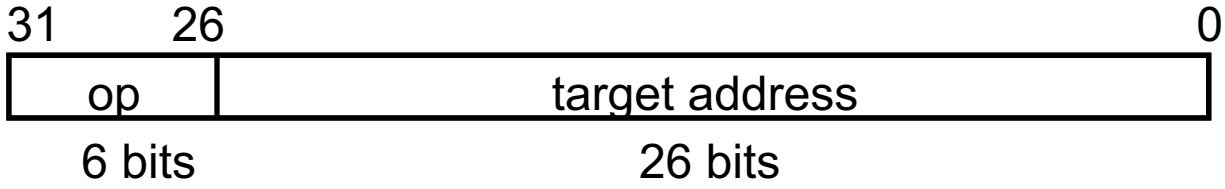
**Ori control signals: RegDst=1; RegWr=1; ALUctr=or; ALUSrc=1**

# Datapath for lw (memory access instruction)

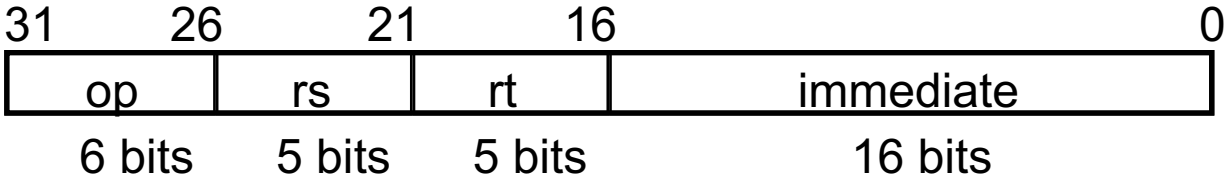
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



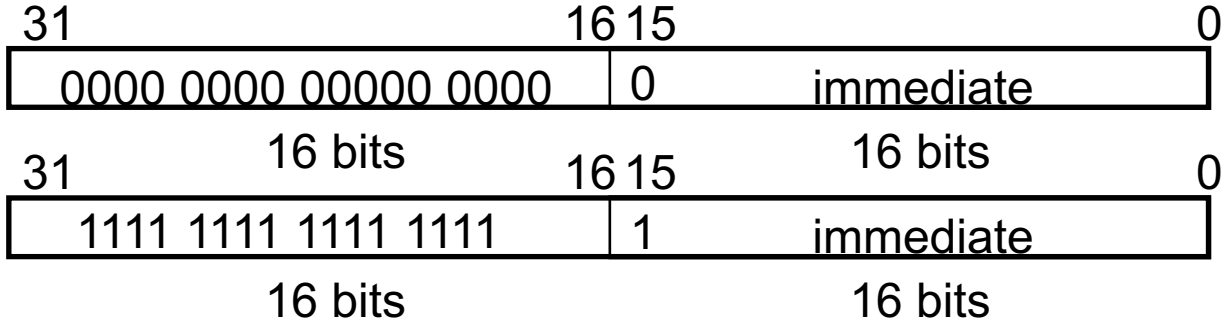



# RTL: The Load Instruction



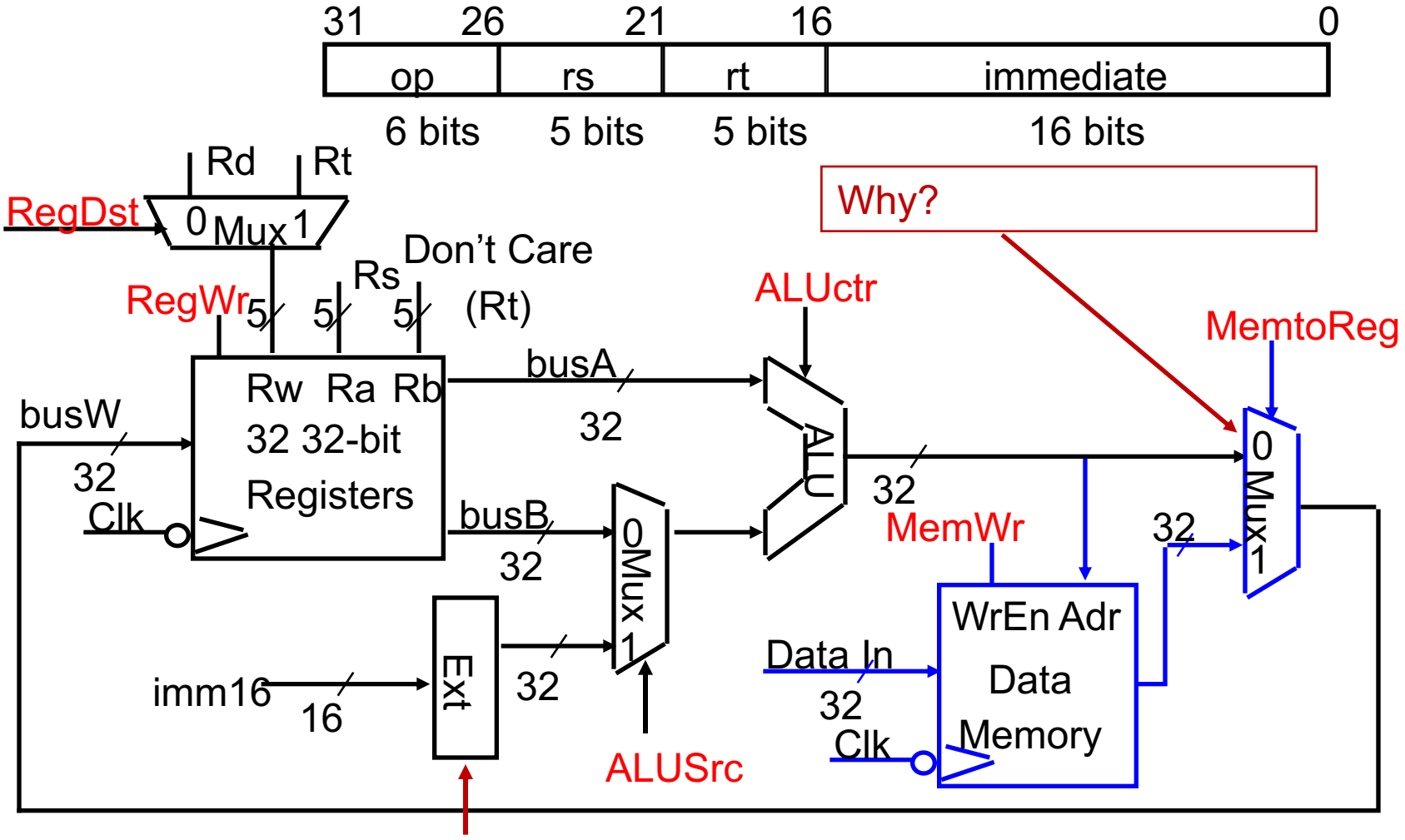

- lw rt, rs, imm16
  - M[PC] Instruction Fetch
  - Addr ← R[rs] + SignExt(imm16) Compute the address
  - R[rt] ← M [Addr] Load Data to rt
  - PC ← PC + 4 Update PC

Why using signed extension rather than zero extension?



# Datapath for Load Instruction

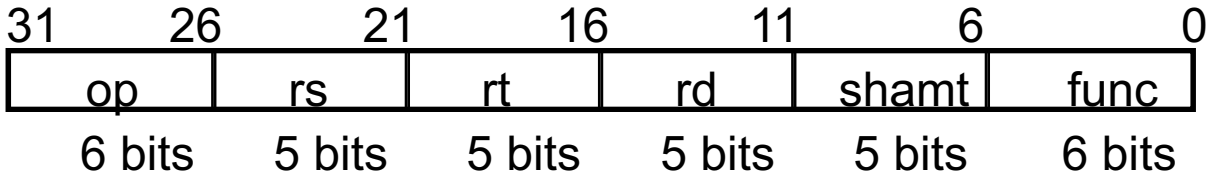
- R[rt] ← M[ R[rs] + SignExt[imm16] ]  Example: lw rt, rs, imm16

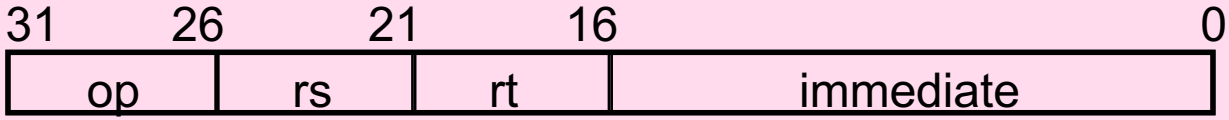


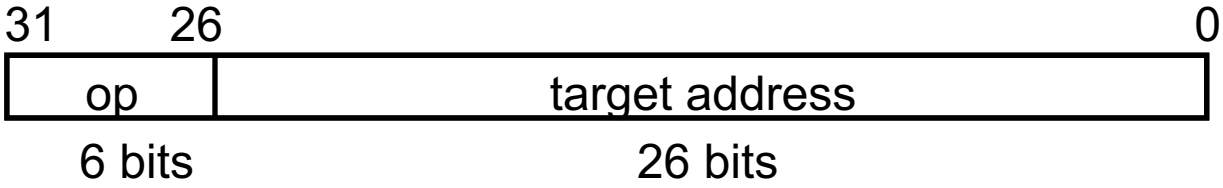

RegDst=1, RegWr=1, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=0, MemtoReg=1

# SW instruction

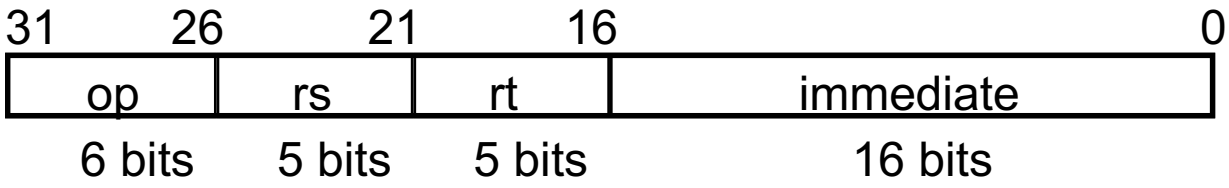
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



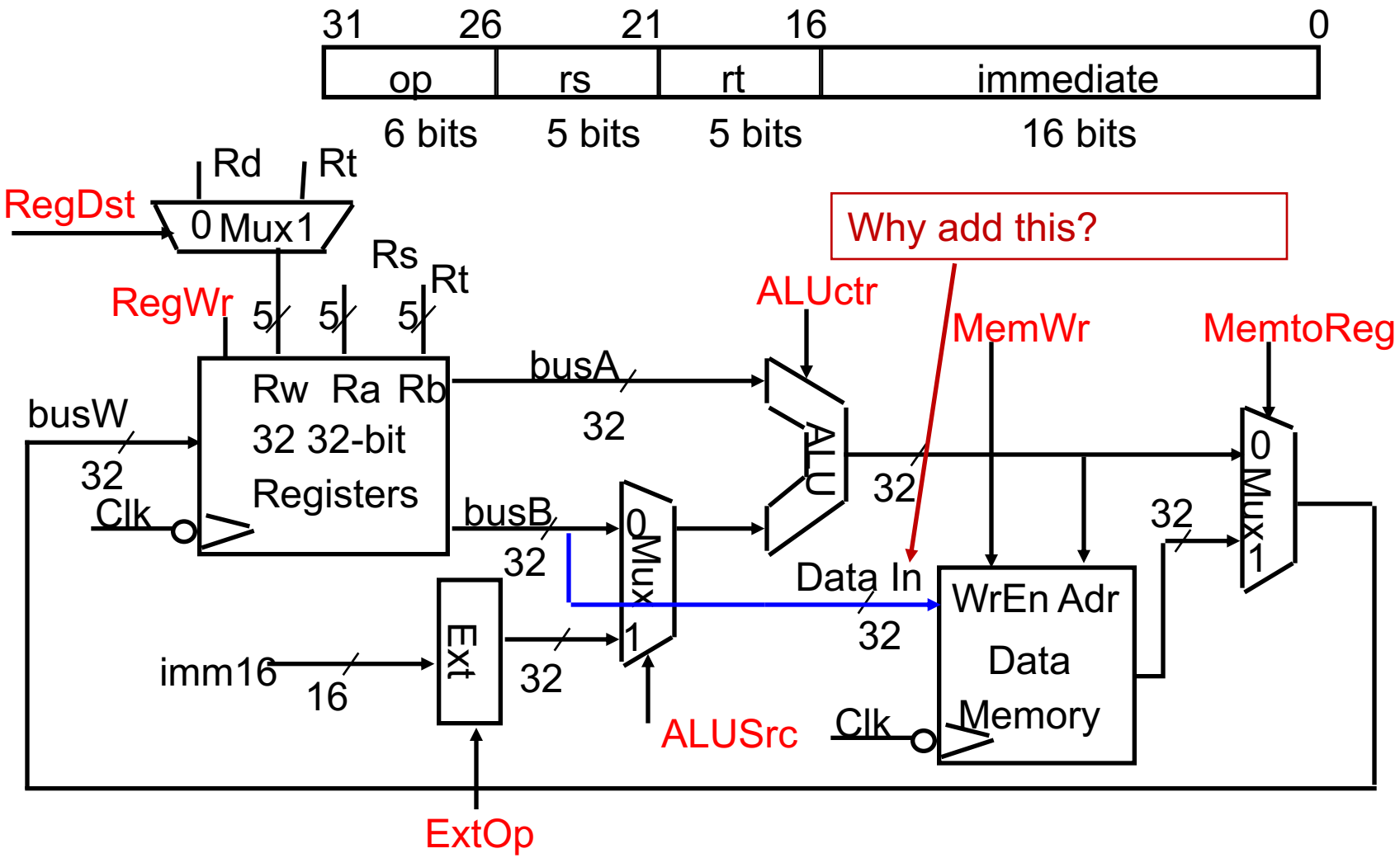
# RTL: The Store Instruction




- sw rt, rs, imm16
  - M[PC]
  - Addr ← R[rs] + SignExt(imm16)
  - Mem[Addr] ← R[rt]
  - PC ← PC + 4

# Datapath for SW

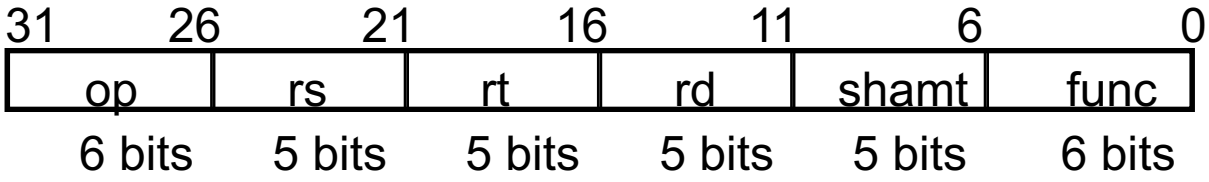
- M[ R[rs] + SignExt[imm16] ] ← R[rt]  Example: sw rt, rs, imm16

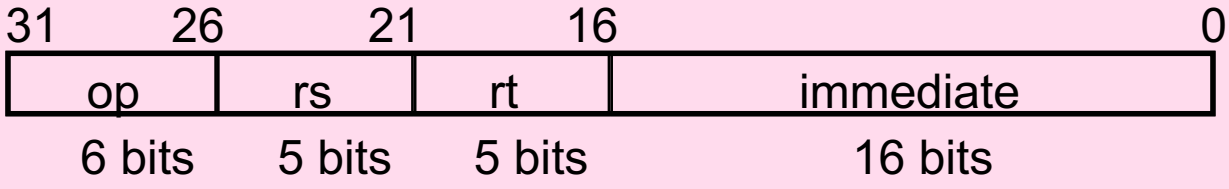


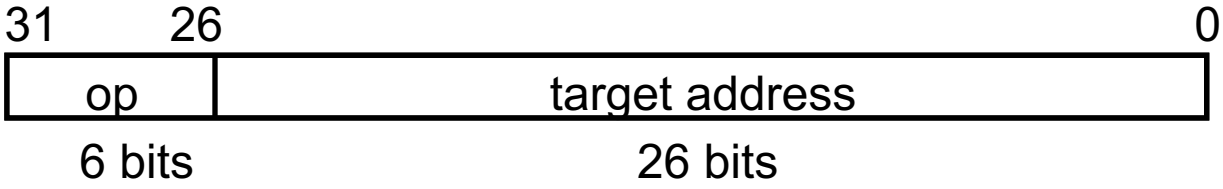

RegDst=x, **RegWr=0**, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=1, MemtoReg=x

# Beq

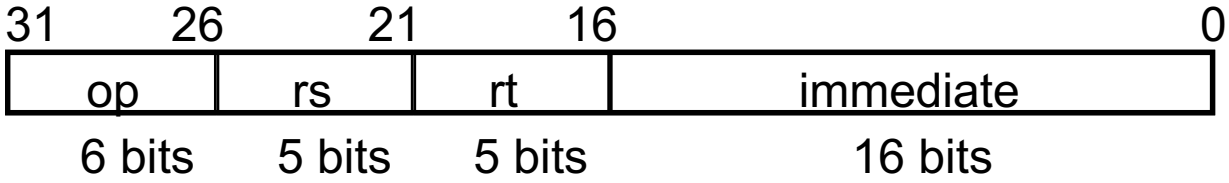
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



# RTL: The Branch Instruction




- beq rs, rt, imm16
  - M[PC]
  - Cond ← R[rs] - R[rt] Compare rs and rt
  - if (COND eq 0) Calculate the next instruction's address

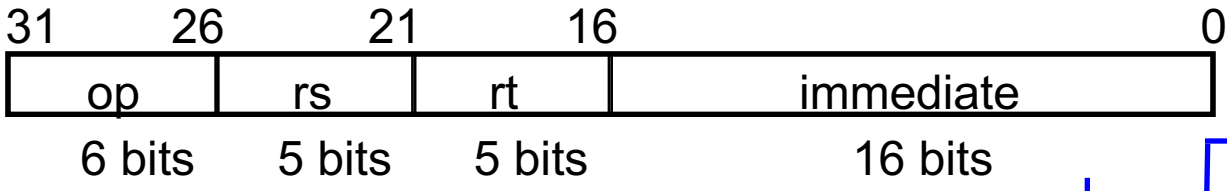
    PC ← PC + 4 + ( SignExt(imm16) x 4 )

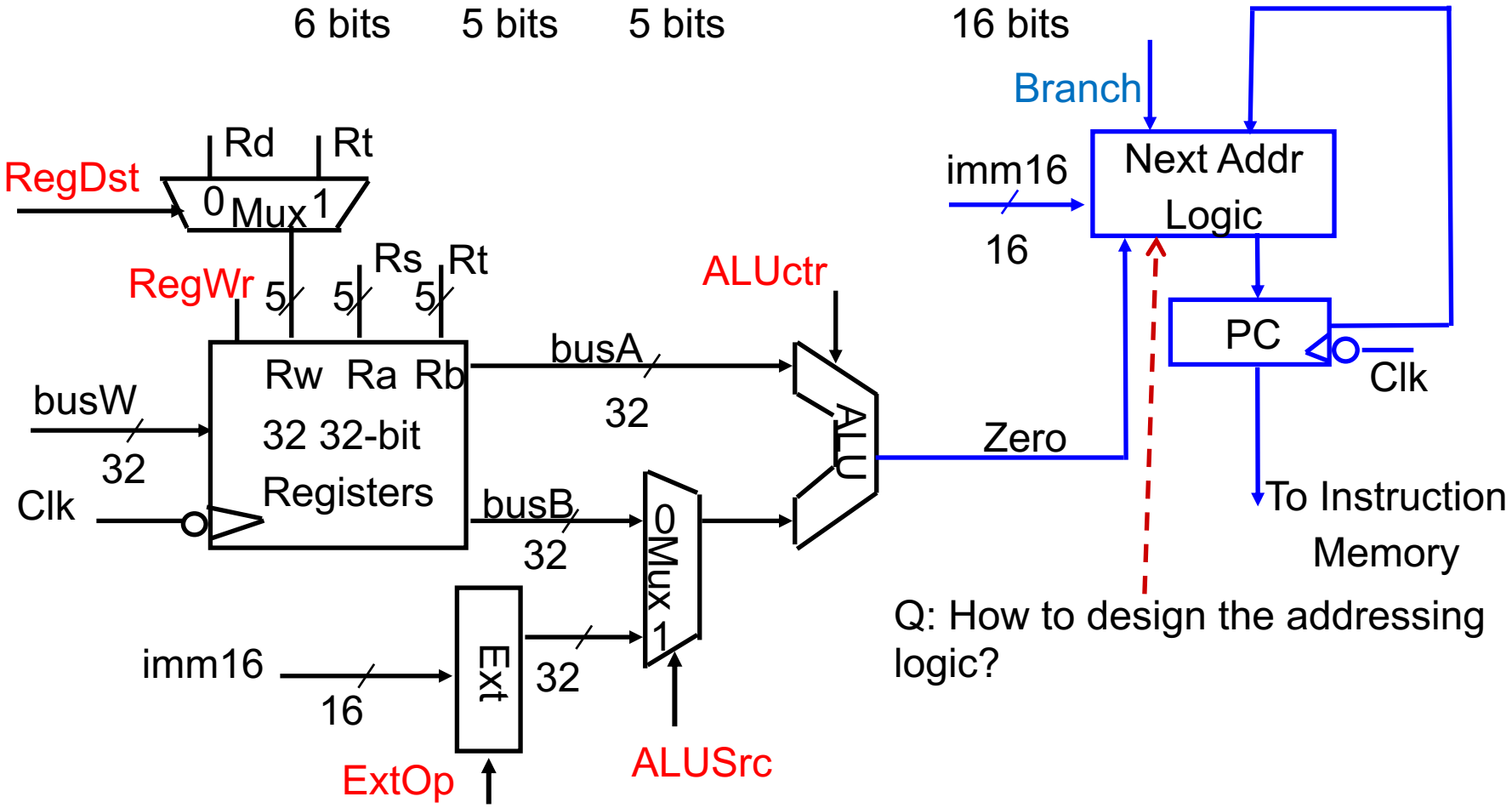
    else

    PC ← PC + 4

# Datapath for beq

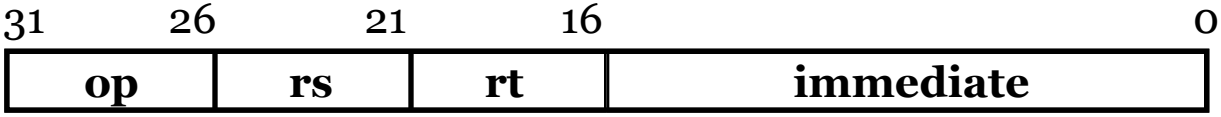
- beq rs, rt, imm16 — We need to compare Rs and Rt !



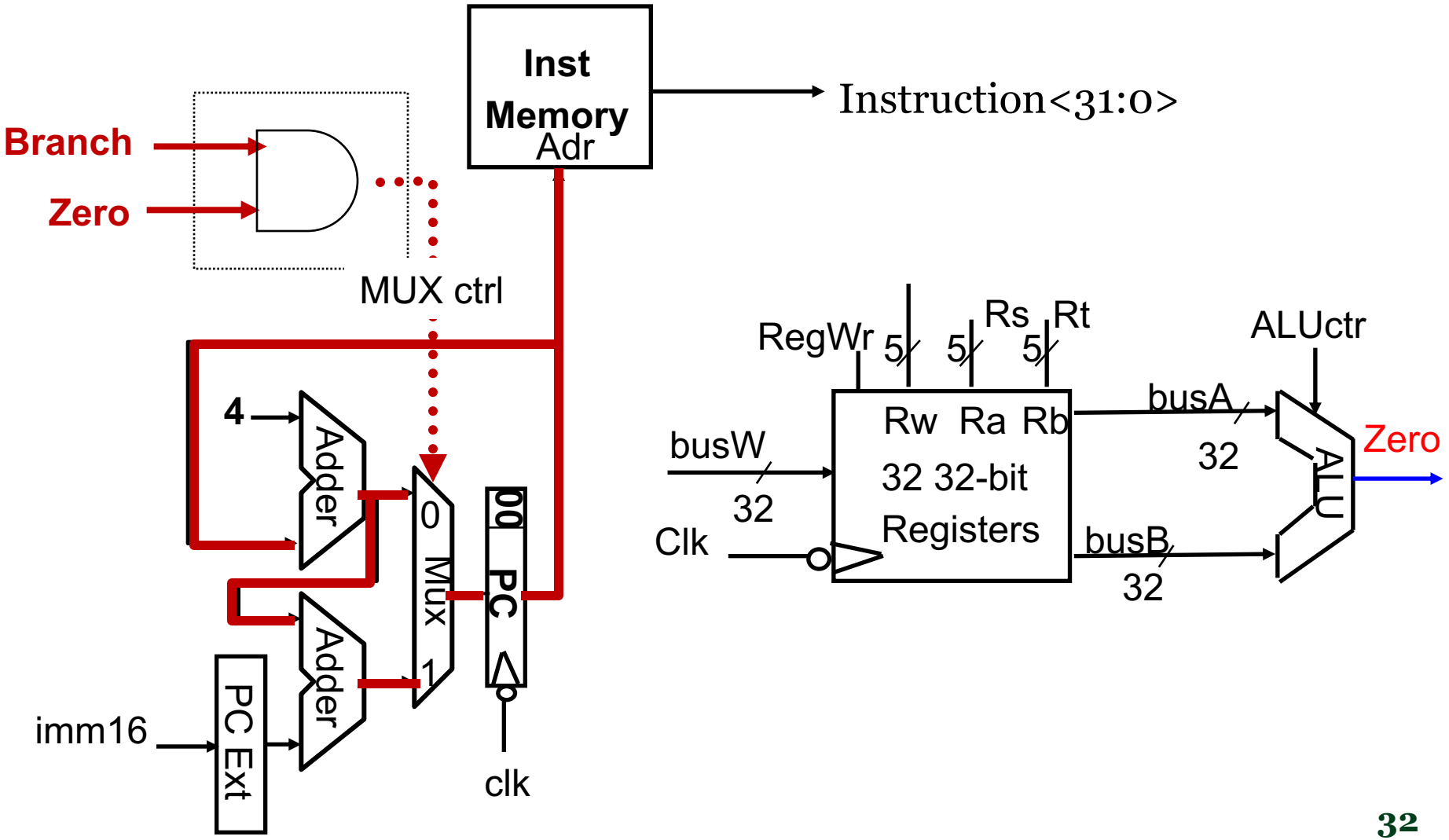




RegDst=x, **RegWr=0**, ALUctr=sub, ExtOp=x, ALUSrc=0, MemWr=0, MemtoReg=x, Branch=1

# Instruction Fetch Unit at the End of Branch

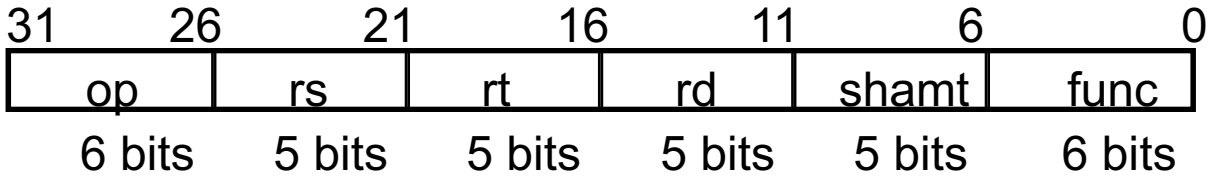


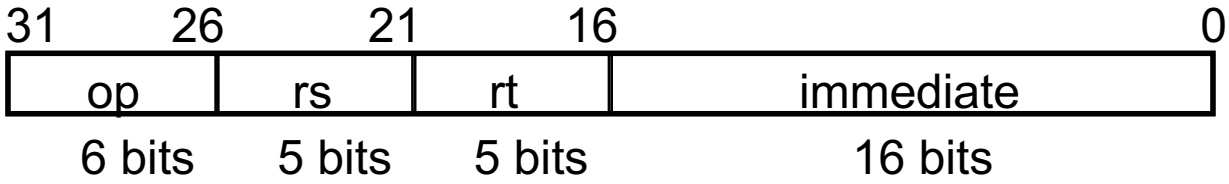

- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

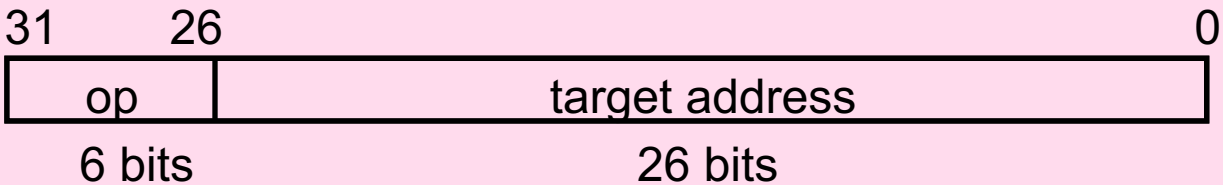


# Jump Operation

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

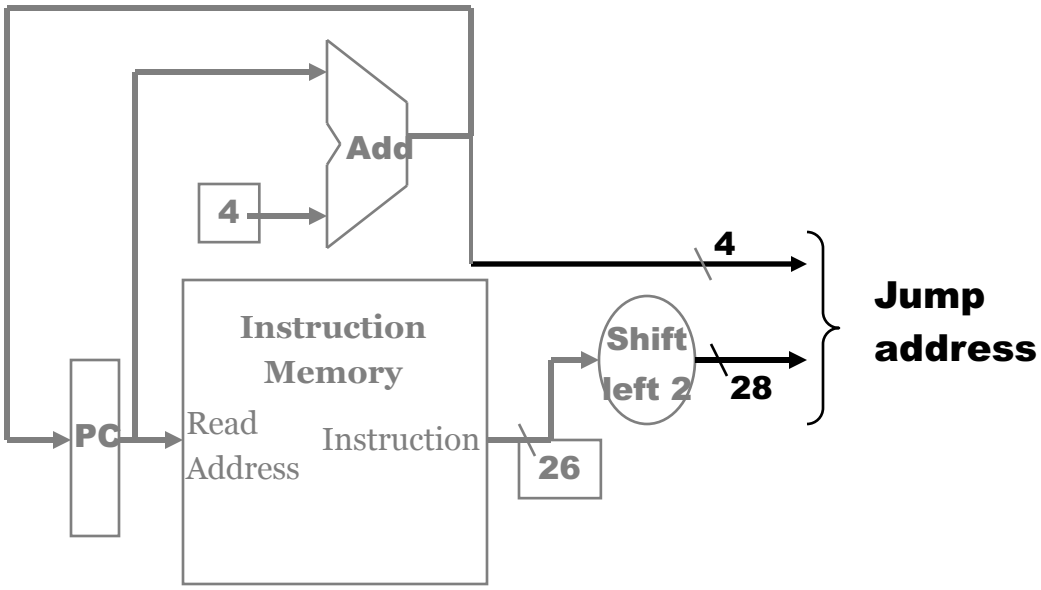


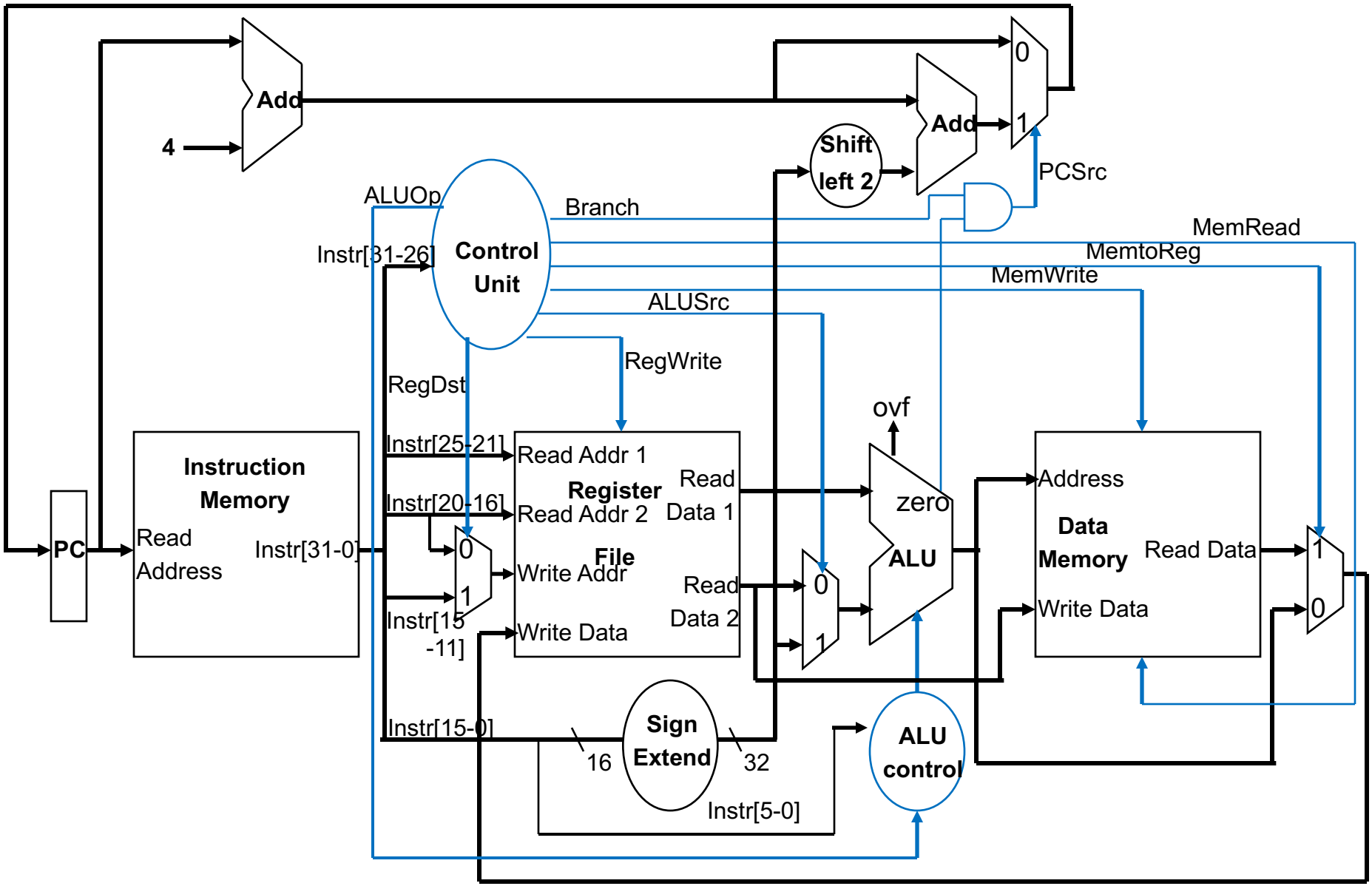



# Executing Jump Operations

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

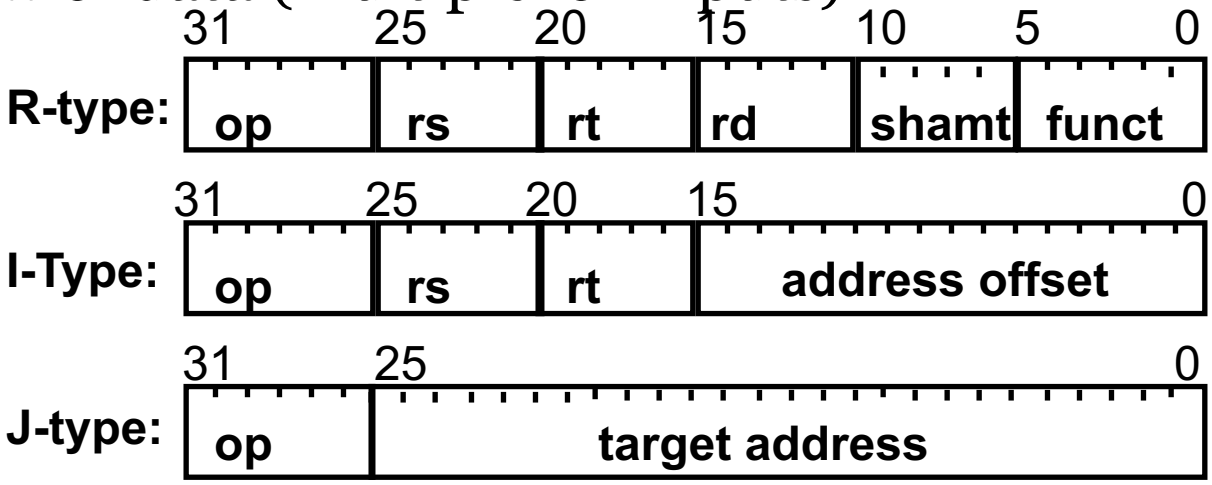


# Single Cycle Datapath ( Without Jump )



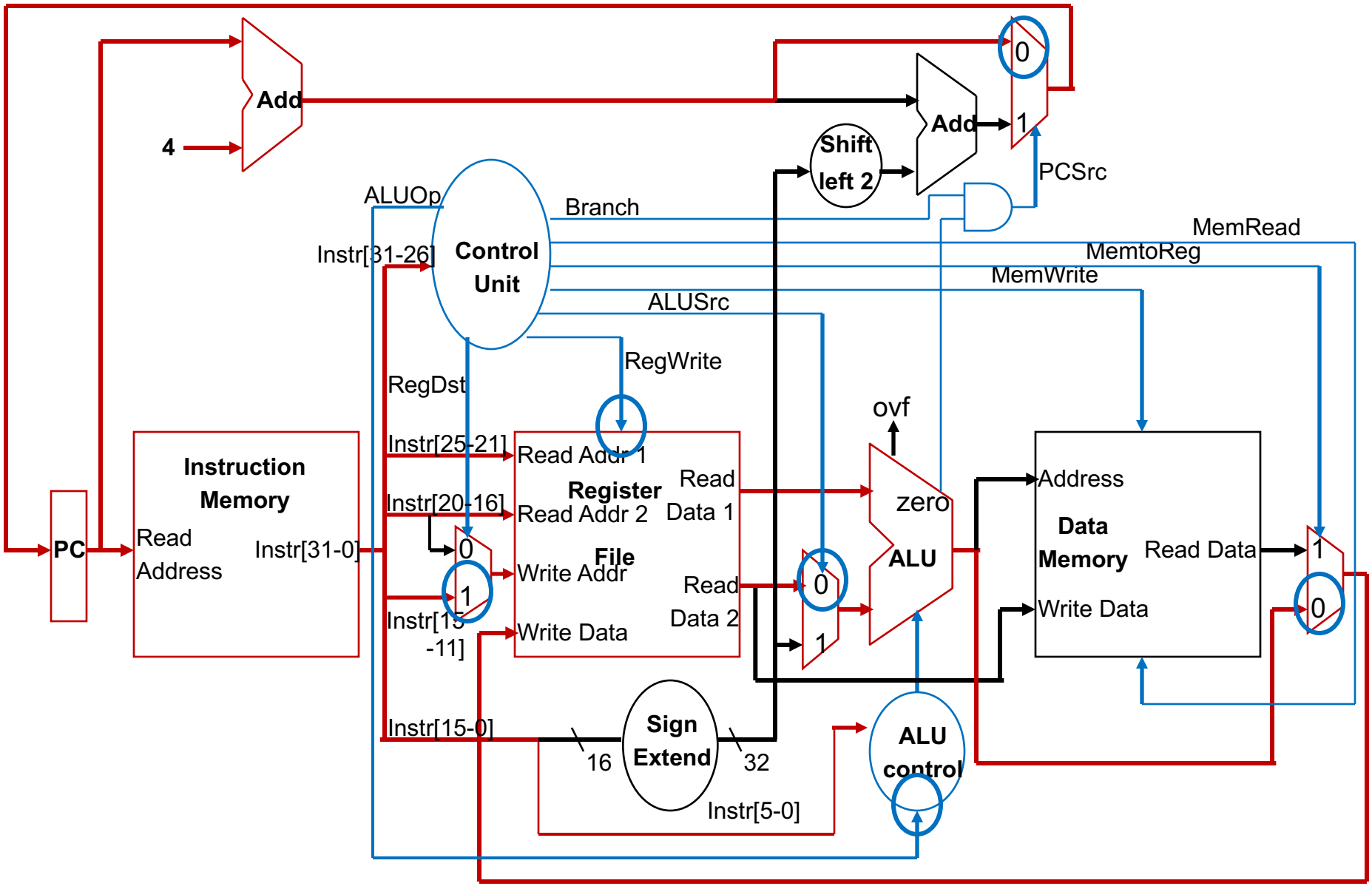
# Step 4: Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)

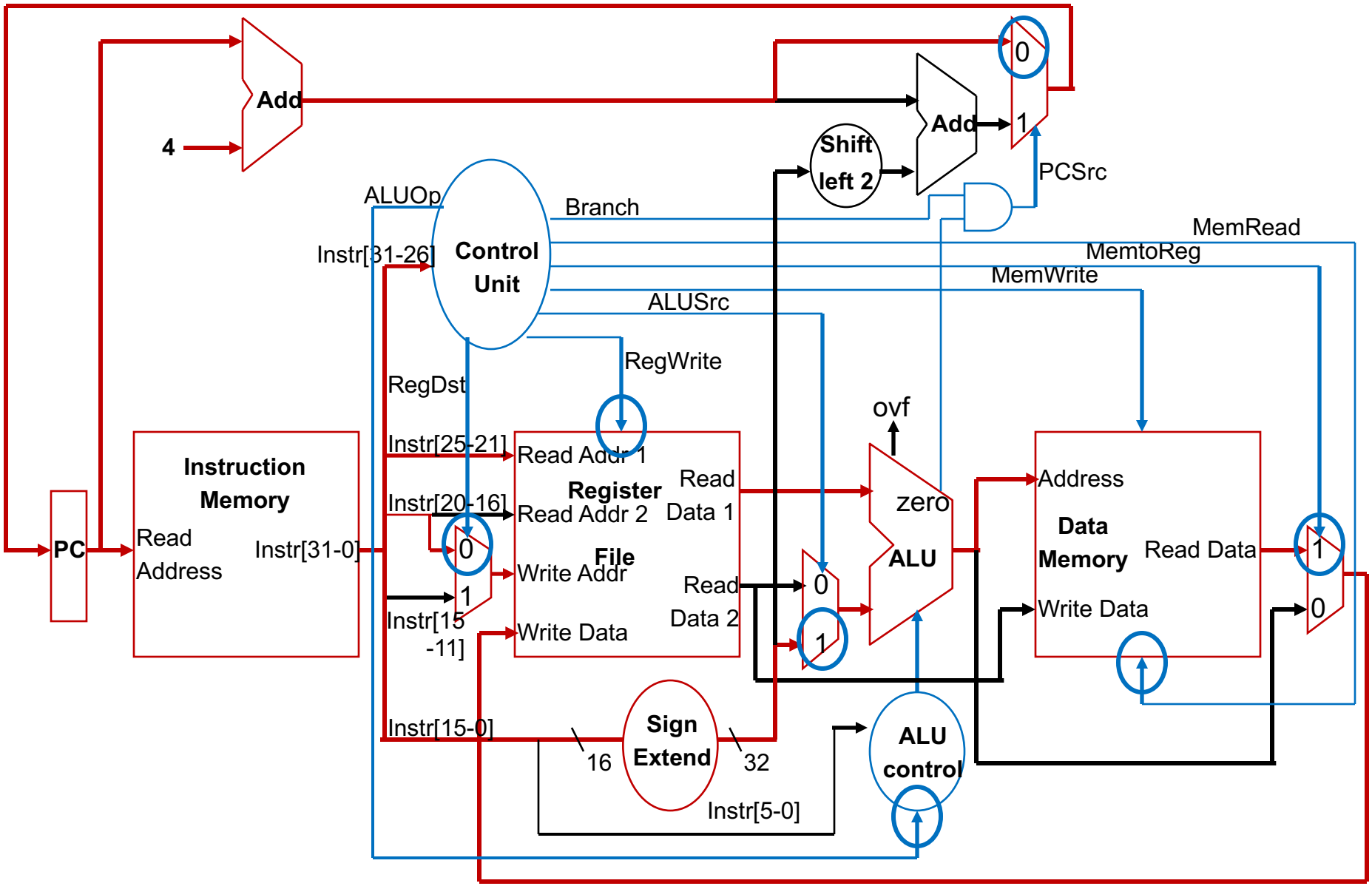



- **Observations**
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

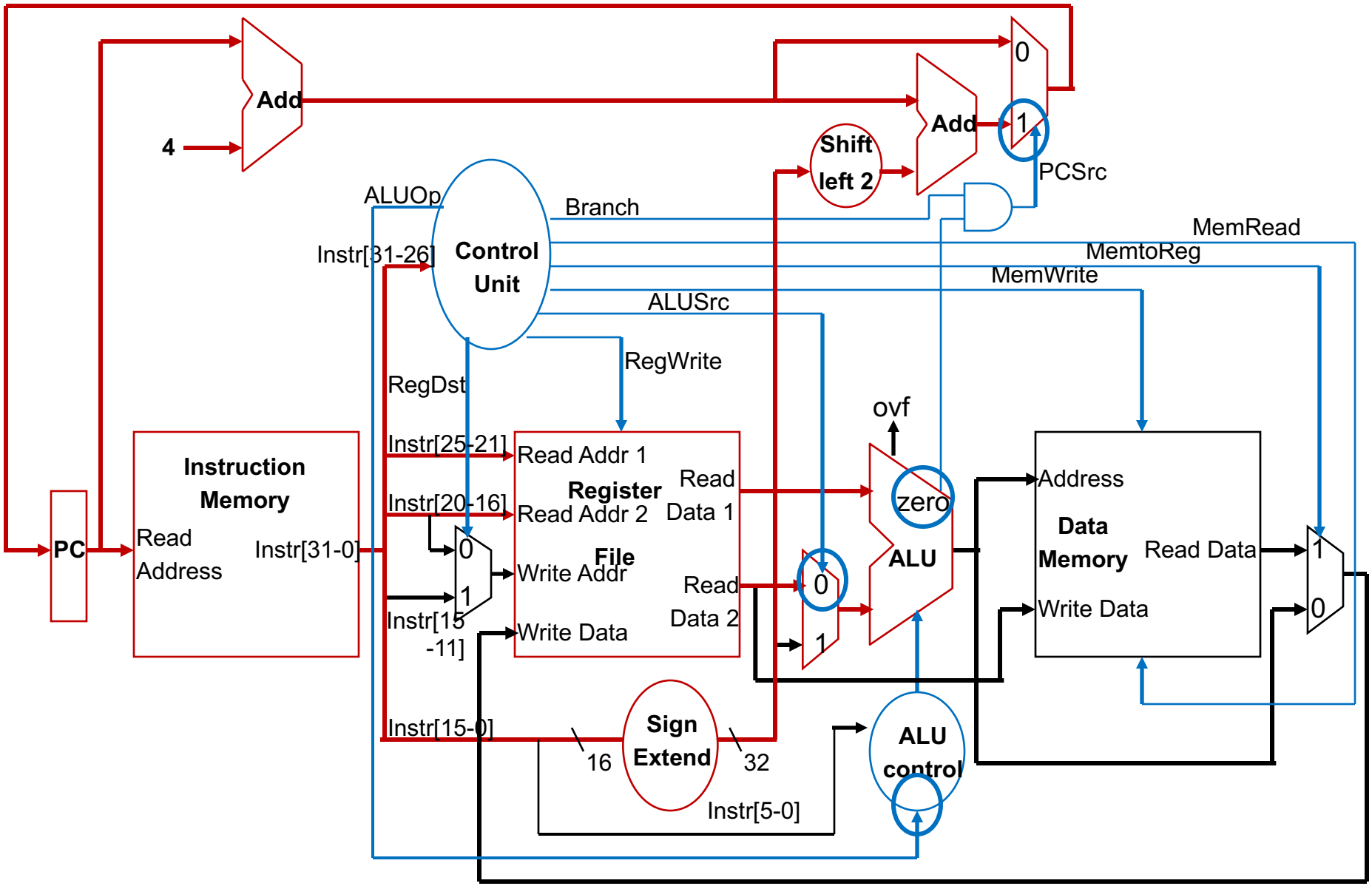
# R-type Instruction Data/Control Flow



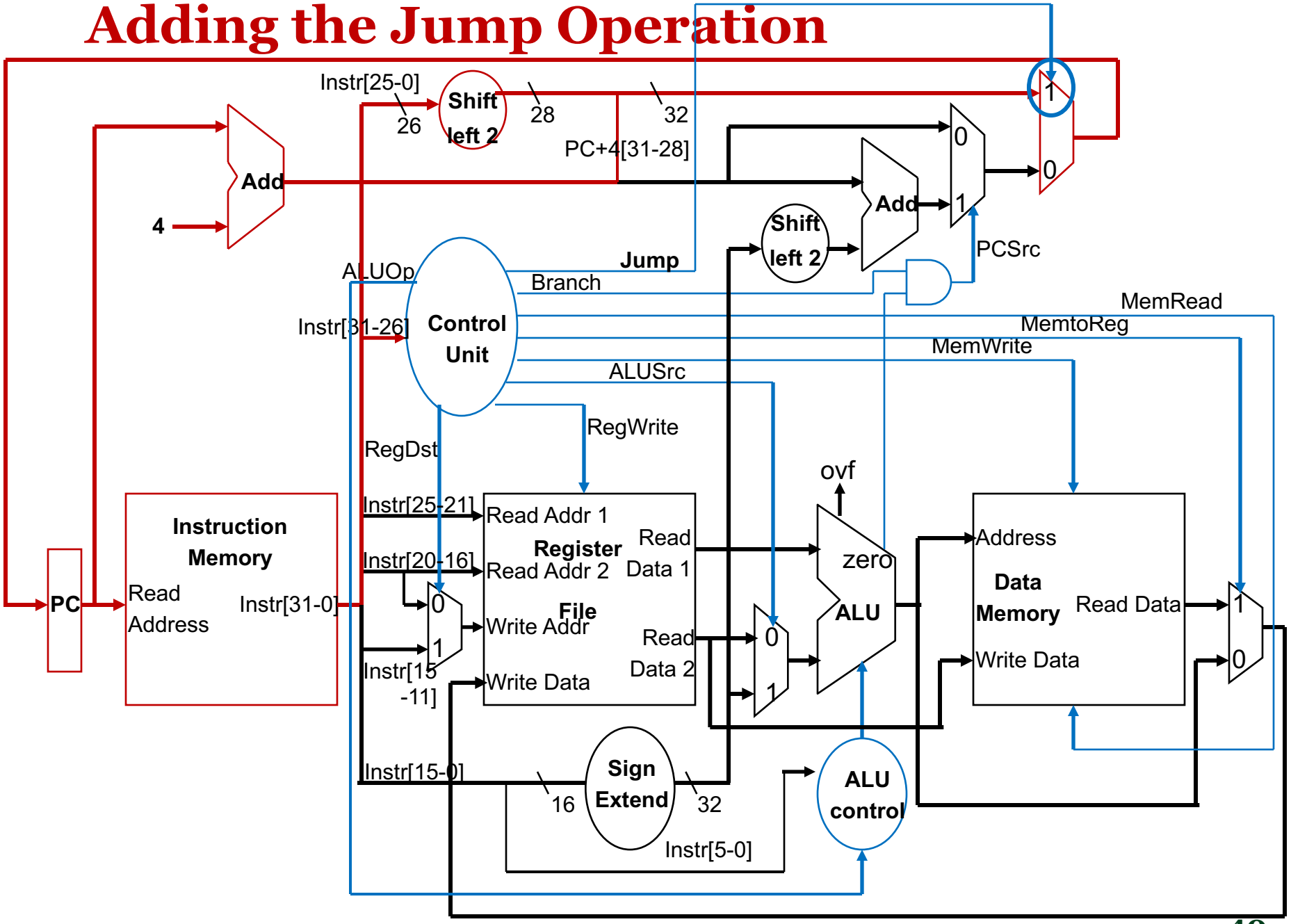
# Load Word Instruction Data/Control Flow



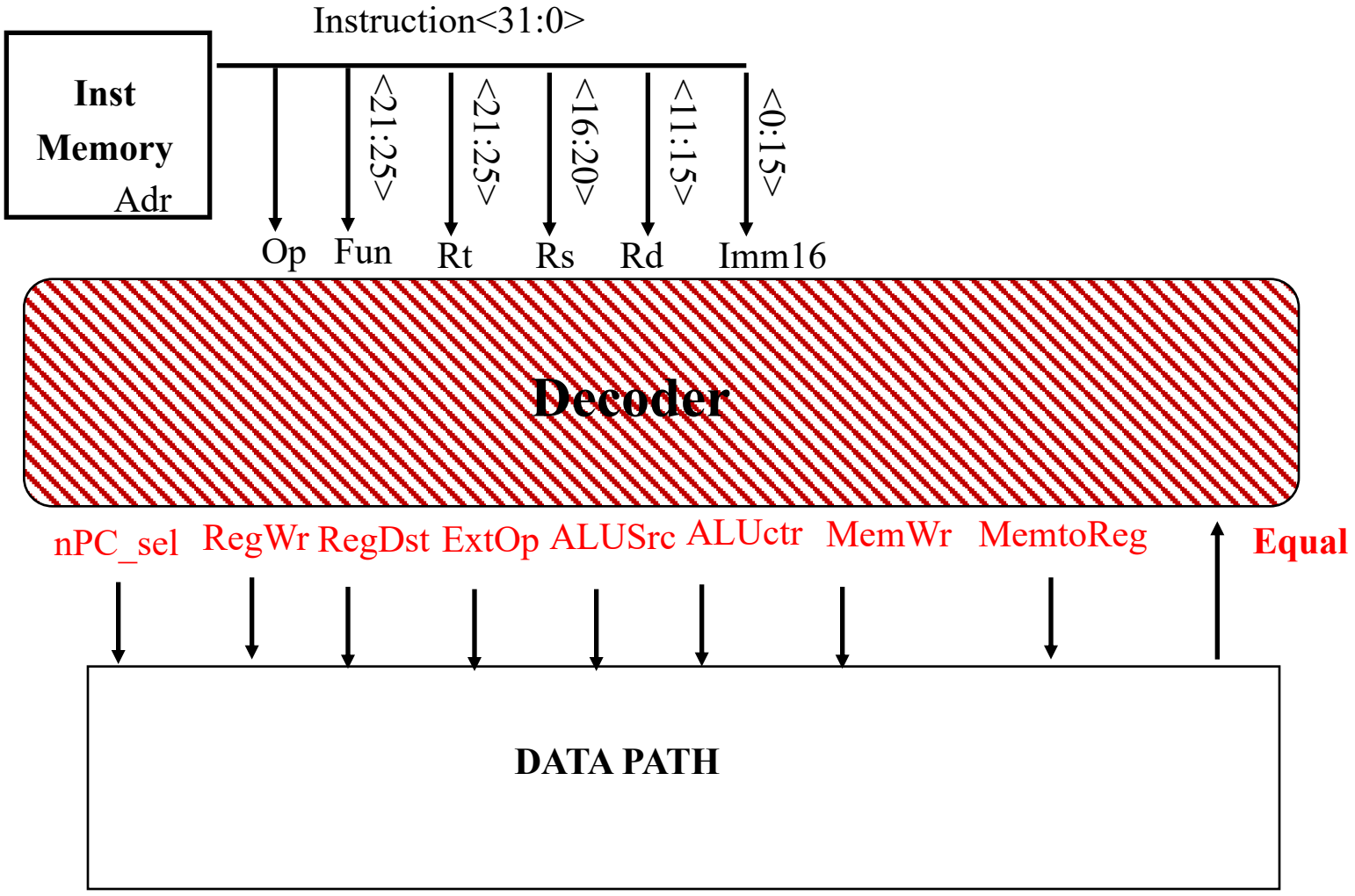
# Branch Instruction Data/Control Flow



# Adding the Jump Operation

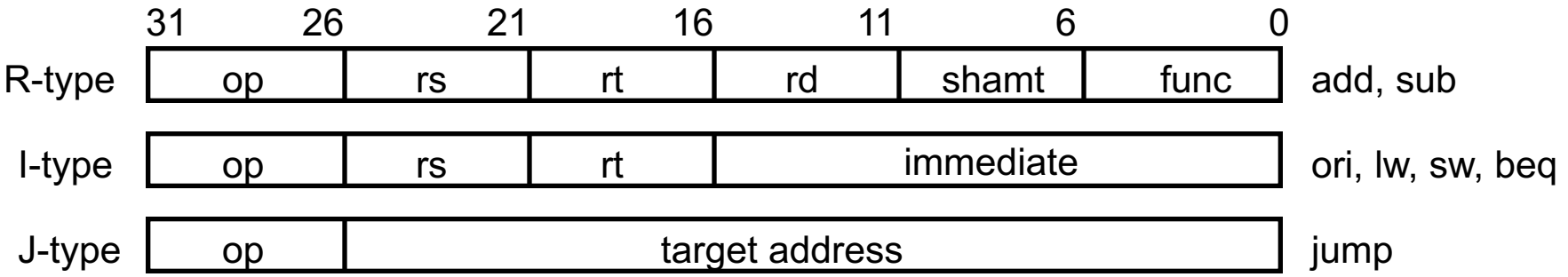


# Assemble Control Logic



# A Summary of the Control Signals

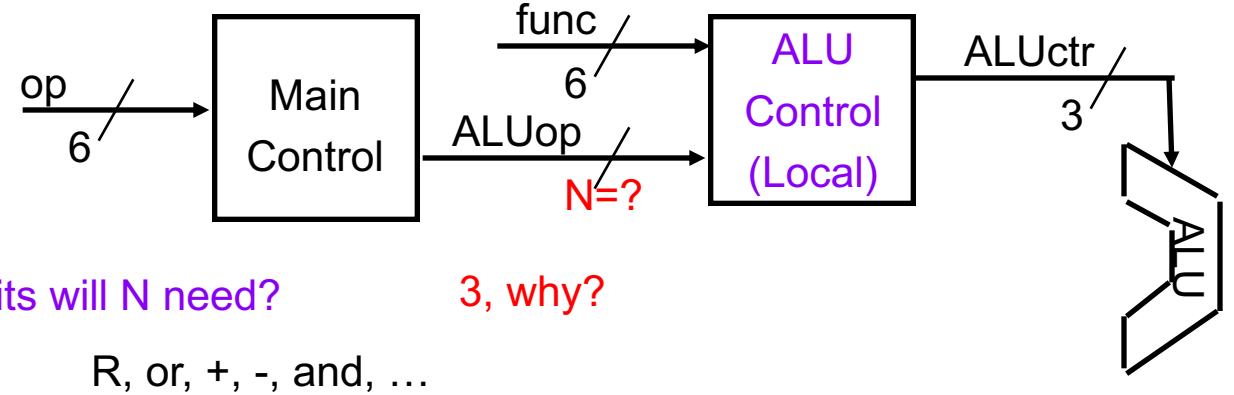
| func / op | 10 0000 / 00 0000 | 10 0010 / 00 0000 | We Don't Care :-) | | | | |
|---|---|---|---|---|---|---|---|
| | add | sub | ori | lw | sw | beq | jump |
| | 00 0000 | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |
| RegDst | 1 | 1 | 0 | 0 | x | x | x |
| ALUSrc | 0 | 0 | 1 | 1 | 1 | 0 | x |
| MemtoReg | 0 | 0 | 0 | 1 | x | x | x |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Branch | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Jump | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| ExtOp | x | x | 0 | 1 | 1 | x | x |
| ALUctr<2:0> | Add | Subtr | Or | Add | Add | Subtr | xxx |



# The Concept of Local Decoding

**Two levels of decoding: Main Control and ALU Control**

| op | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |
|---|---|---|---|---|---|---|
| | R-type | ori | lw | sw | beq | jump |
| RegDst | 1 | 0 | 0 | x | x | x |
| ALUSrc | 0 | 1 | 1 | 1 | 0 | x |
| MemtoReg | 0 | 0 | 1 | x | x | x |
| RegWrite | 1 | 1 | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 1 | 0 | 0 |
| Branch | 0 | 0 | 0 | 0 | 1 | 0 |
| Jump | 0 | 0 | 0 | 0 | 0 | 1 |
| ExtOp | x | 0 | 1 | 1 | x | x |
| ALUctr | Add/Subtr | Or | Add | Add | Subtr | xxx |



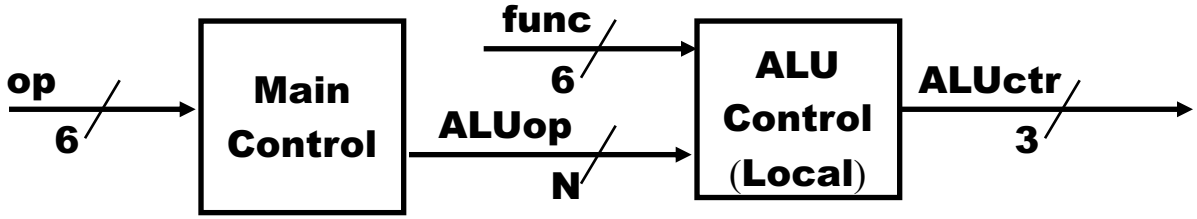

ALUctr is determined by ALUop and func, while other control signals are determined by op

How many bits will N need?

3, why?

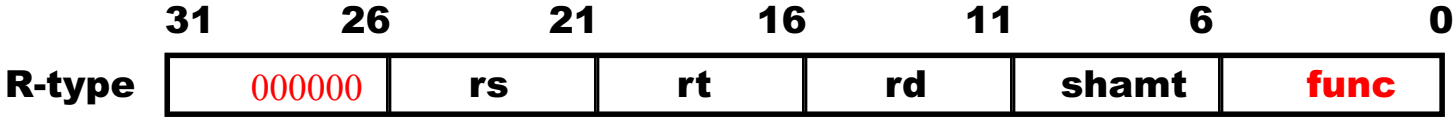
R, or, +, -, and, …

# The Decoding of the "func" Field

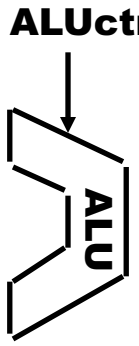



**Encoding ALUop as follows**

| | R-type | ori | lw | sw | beq | jump |
|---|---|---|---|---|---|---|
| ALUop (Symbolic) | "R-type" | Or | Add | Add | Subtr | xxx |
| ALUop<2:0> | 1 00 | 0 10 | 0 00 | 0 00 | 001 | xxx |




| func<5:0> | Instruction Operation |
|---|---|
| 10 0000 | add |
| 10 0010 | subtract |
| 10 0100 | and |
| 10 0101 | or |
| 10 1010 | set-on-less-than |




| ALUctr<2:0> | ALU Operation |
|---|---|
| 000 | Add |
| 001 | Subtract |
| 010 | And |
| 110 | Or |
| 001 | Subtract |

# The Truth Table for ALUctr

R-type Instructions determined by funct

Non-R-type Instructions determined by ALUop

| ALUop (Symbolic) | R-type | ori | lw | sw | beq |
|---|---|---|---|---|---|
| | "R-type" | Or | Add | Add | Subtr |
| ALUop<2:0> | 1 00 | 0 10 | 0 00 | 0 00 | 001 |

| funct<3:0> | Instruction Op. |
|---|---|
| 0000 | add |
| 0010 | subtract |
| 0100 | and |
| 0101 | or |
| 1010 | set-on-less-than |

| ALUop | | | func | | | | ALU | ALUctr | | |
|---|---|---|---|---|---|---|---|---|---|---|
| bit2 | bit1 | bit0 | bit<3> | bit<2> | bit<1> | bit<0> | Operation | bit<2> | bit<1> | bit<0> |
| 0 | 0 | 0 | x | x | x | x | Add | 0 | 0 | 0 |
| 0 | x | 1 | x | x | x | x | Subtract | 0 | 0 | 1 |
| 0 | 1 | 0 | x | x | x | x | Or | 1 | 1 | 0 |
| 1 | x | x | 0 | 0 | 0 | 0 | Add | 0 | 0 | 0 |
| 1 | x | x | 0 | 0 | 1 | 0 | Subtract | 0 | 0 | 1 |
| 1 | x | x | 0 | 1 | 0 | 0 | And | 0 | 1 | 0 |
| 1 | x | x | 0 | 1 | 0 | 1 | Or | 1 | 1 | 0 |
| 1 | x | x | 1 | 0 | 1 | 0 | Subtract | 0 | 0 | 1 |

# The Logic Equation for ALUctr<0>

## Choose the rows with ALUctr[0]=1

| ALUop | | | func | | | | |
|---|---|---|---|---|---|---|---|
| bit<2> | bit<1> | bit<0> | bit<3> | bit<2> | bit<1> | bit<0> | ALUctr<0> |
| 0 | x | 1 | x | x | x | x | 1 |
| 1 | x | x | 0 | 0 | 1 | 0 | 1 |
| 1 | x | x | 1 | 0 | 1 | 0 | 1 |

This makes func<3> a don't care

- ALUctr<0> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>

# The Logic Equation for ALUctr<1>

**Choose the rows with ALUctr[1]=1**

| ALUop | | | func | | | | |
|---|---|---|---|---|---|---|---|
| bit<2> | bit<1> | bit<0> | bit<3> | bit<2> | bit<1> | bit<0> | ALUctr<1> |
| 0 | 1 | 0 | x | x | x | x | 1 |
| | | | | | | | |
| 1 | x | x | 0 | 1 | 0 | 0 | 1 |
| 1 | x | x | 0 | 1 | 0 | 1 | 1 |
| | | | | | | | |

- ALUctr<1> = !ALUop<2> & ALUop<1> & ! ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1>

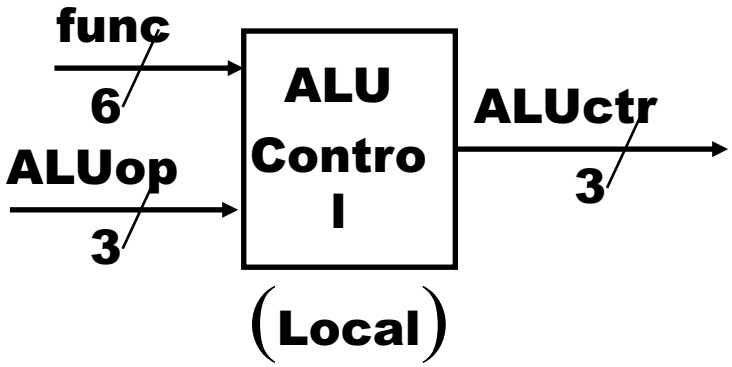
# The Logic Equation for ALUctr<2>

**Choose the rows with ALUctr[2]=1**

| ALUop | | | func | | | | |
|---|---|---|---|---|---|---|---|
| bit<2> | bit<1> | bit<0> | bit<3> | bit<2> | bit<1> | bit<0> | ALUctr<2> |
| 0 | 1 | 0 | x | x | x | x | 1 |
| 1 | x | x | 0 | 1 | 0 | 1 | 1 |
| | | | | | | | |

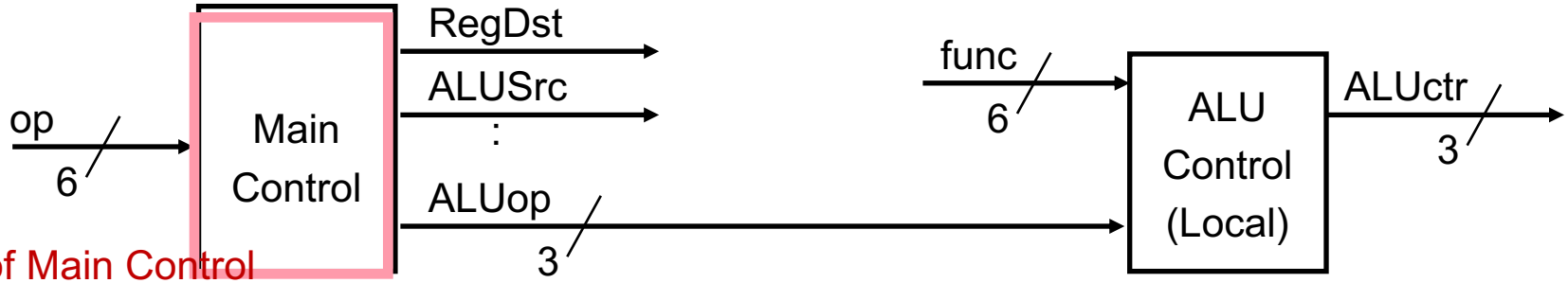
- ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0>
  + ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

# Summary of Control Logic of Local Control




- ALUctr<0> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>
- ALUctr<1> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1>
- ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

# The "Truth Table" for the Main Control




Output of Main Control

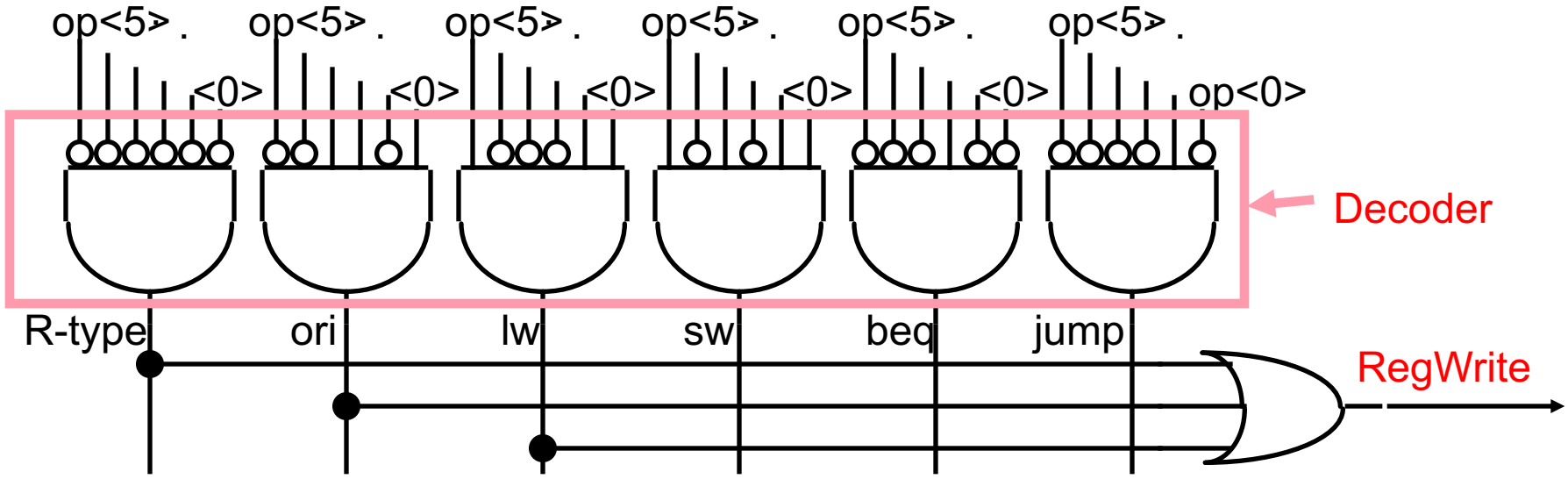
Input of main control

| op | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |
|---|---|---|---|---|---|---|
| | R-type | ori | lw | sw | beq | jump |
| RegDst | 1 | 0 | 0 | x | x | x |
| ALUSrc | 0 | 1 | 1 | 1 | 0 | x |
| MemtoReg | 0 | 0 | 1 | x | x | x |
| RegWrite | 1 | 1 | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 1 | 0 | 0 |
| Branch | 0 | 0 | 0 | 0 | 1 | 0 |
| Jump | 0 | 0 | 0 | 0 | 0 | 1 |
| ExtOp | x | 0 | 1 | 1 | x | x |
| ALUop (Symbolic) | "R-type" | Or | Add | Add | Subtr | xxx |
| ALUop <2> | 1 | 0 | 0 | 0 | 0 | x |
| ALUop <1> | x | 1 | 0 | 0 | x | x |
| ALUop <0> | x | 0 | 0 | 0 | 1 | x |

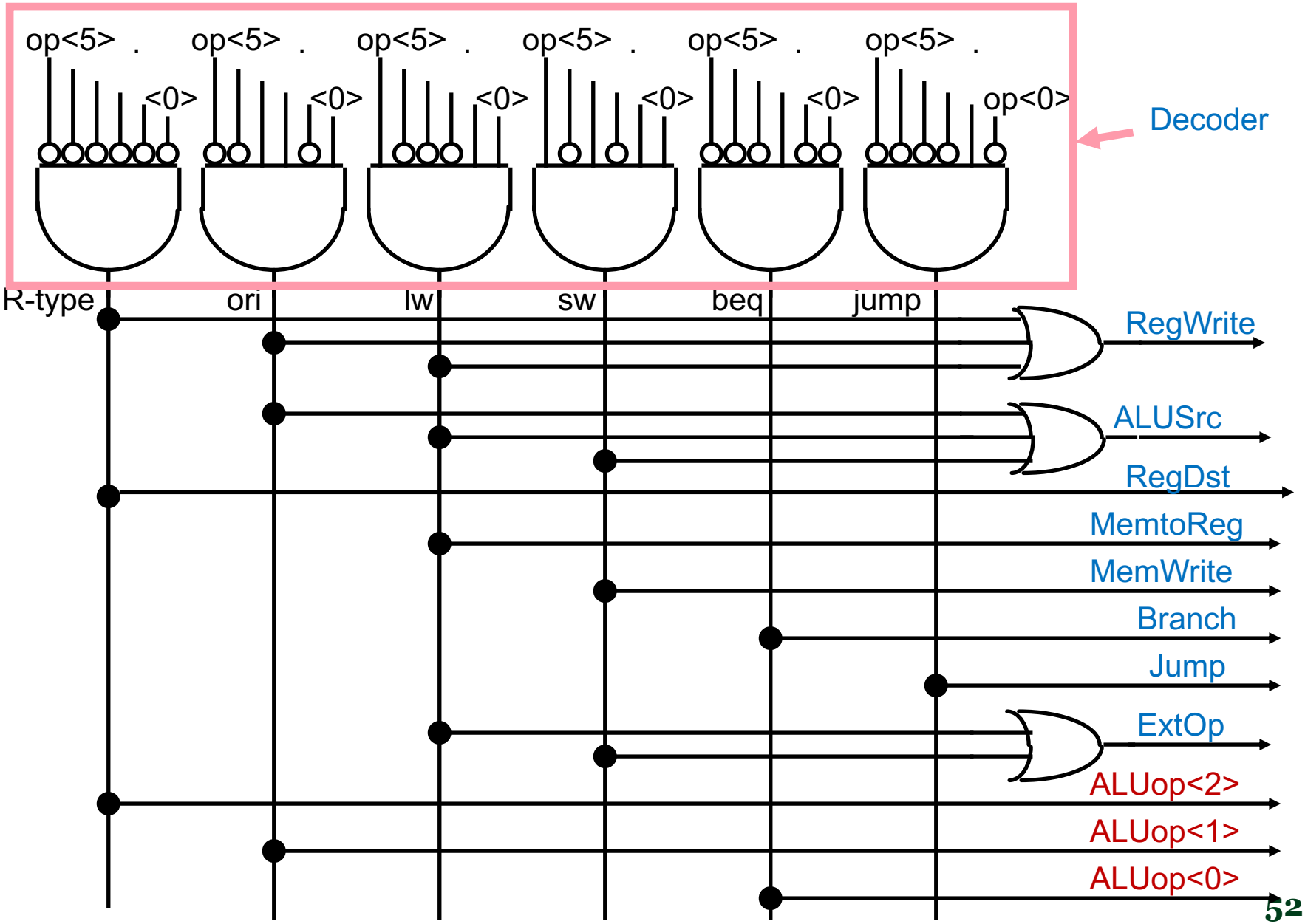
# The "Truth Table" for RegWrite

| op | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |
|---|---|---|---|---|---|---|
| | R-type | ori | lw | sw | beq | jump |
| RegWrite | 1 | 1 | 1 | 0 | 0 | 0 |

- RegWrite = R-type + ori + lw
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



# Assemble Main Control (PLA)



# The Complete Single Cycle Data Path with Control