SINGLE CYCLE PROCESSOR

Lecturer: Hui Annie Guo
h.guo@unsw.edu.au
K17-501F

Lecture overview

- Topics
  - Single-cycle processor
    - Datapath
    - Control
- Suggested reading
  - H&P Chapter 4.1-4.4

Typical steps of processor design

- **Assume ISA is given. To build a processor, we basically go through the following steps:**

For datapath

1. **Analyse instruction set to determine datapath requirements**
2. **Select a set of hardware components for the datapath and establish clocking methodology**
3. **Assemble datapath to meet the requirements**

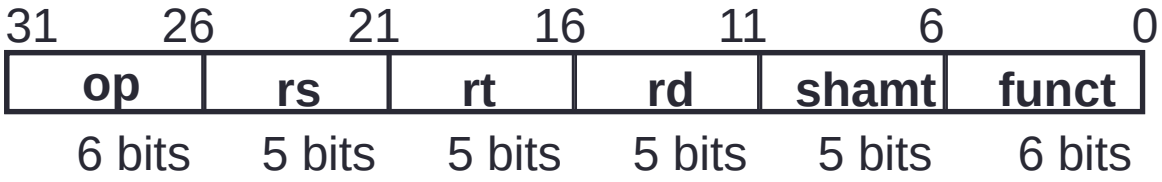
For control

4. **Analyse implementation of each instruction to determine control points**
5. **Assemble the control logic**

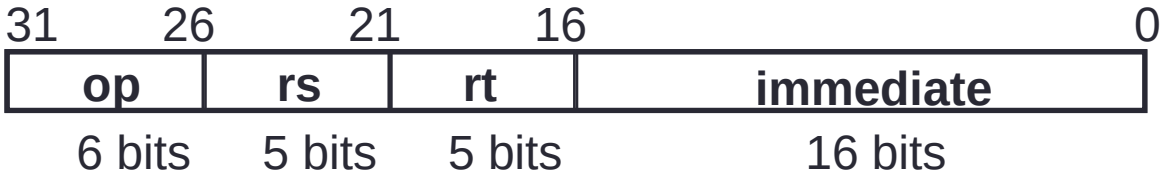
- **A demonstration with MIPS-Lite is given next**

MIPS-Lite (a sub set of MIPS ISA)

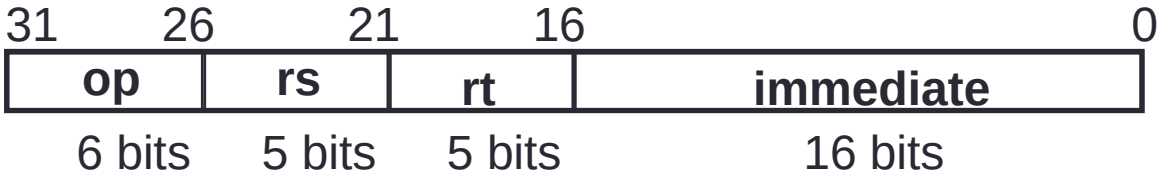
- **We demonstrate the datapath design for the following instructions**
  - **ADD and SUB**
    - **ADDU rd, rs, rt**
    - **SUBU rd, rs, rt**



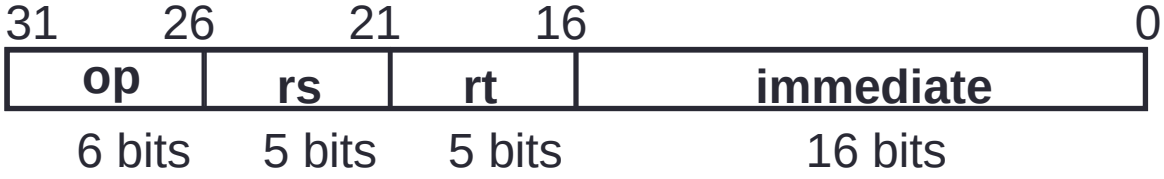
  - **OR Immediate:**
    - **ORi rt, rs, imm16**



  - **LOAD and STORE**
    - **LW rt, rs, imm16**
    - **SW rt, rs, imm16**



  - **BRANCH:**
    - **BEG rs, rt, imm16**



  - **JUMP**
    - **J imm26**
      - covered later

Step 1

- **Analyse instruction set to determine datapath requirements**
  - **For each instruction, its requirement can be identified as a set of register transfer operations**
    - **Data transferred from one storage location to another storage location may go through a combinational logic**
  - ***Register-level Transfer Language (RTL)* is used to describe the instruction execution**
    - **E.g. $3 ← $1+$2, for** ADDU $3,$1,$2
      - Values in register $1 and $2 are added and the result is saved in register $3
  - **Datapath must support each transfer operation**

MIPS-Lite RTL specifications

Step 1

- **All instructions start by fetching instruction,**

MEM[PC] = op | rs | rt | rd | shamt | funct or
op | rs | rt | Imm16

← instructions in one of the two instruction formats

- **Then followed by different operations**

Instr.	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;
LW	R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4;
SW	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;

Requirements of the instruction set

Step 1

- Memory
  - instruction & data
- PC
- Registers (let's say 32 x 32)
  - read rs
  - read rt
  - write rt or rd
- Add/Sub/Or
  - on registers, or
  - extended immediate
- Bit extension
- Add
  - PC + 4
  - PC + 4 + extended immediate

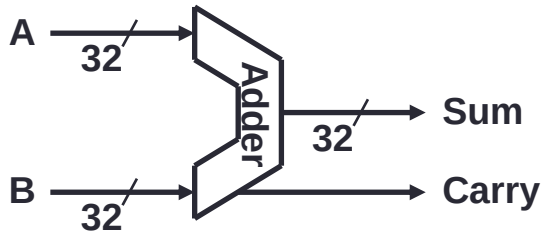
Step 2:

- **Select a set of components for the datapath and establish clocking methodology**
  - **Combinational elements**
  - **Storage elements**
    - **Clocking methodology**

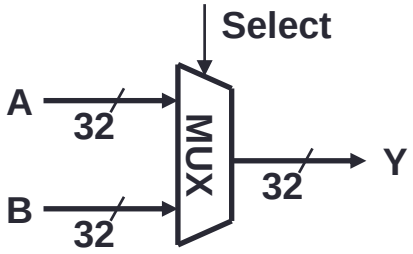
Combinational logic elements

Step 2

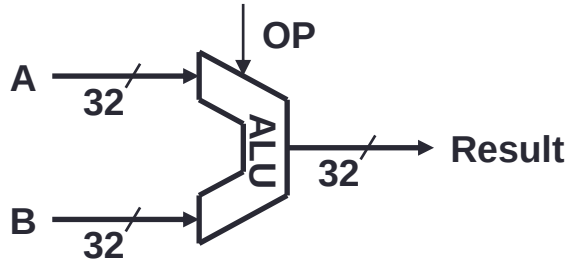
- **adder**



- **MUX**



- **ALU**

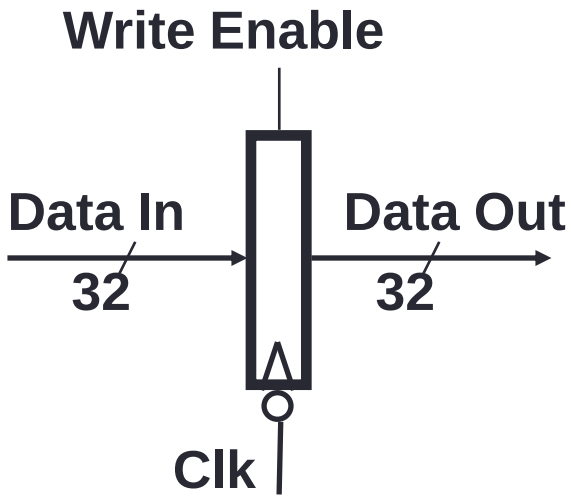


- **Other components**
  - **Added as we go**

Step 2

Storage elements: registers

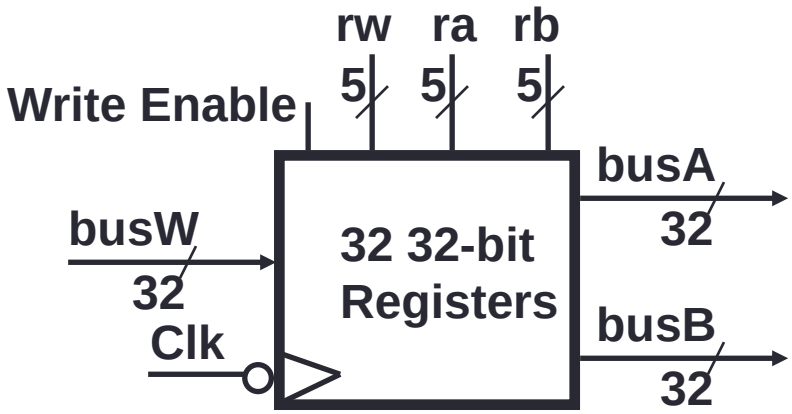
- A register consists of a set of Flip Flops
  - 32-bit input and output
  - Write Enable input
    - 0: disable
    - 1: enable



Storage elements: register file

Step 2

- **Register File (RF) consists of 32 registers:**
  - **Two 32-bit output ports/buses:**
    - **busA and busB**
  - **One 32-bit input port/bus: busW**



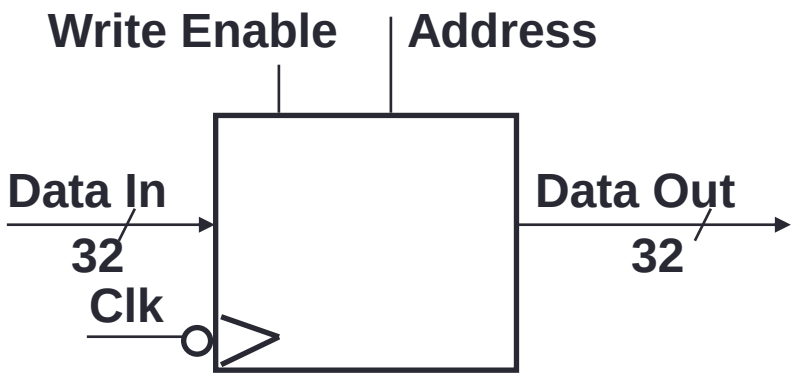
- **Register is selected by ra, rb, rw:**
  - **(ra) → busA**
  - **(rb) → busB**
  - **(rw) ← busW if Write Enable is 1**
- **Clock input (Clk)**
  - **The Clk input is often used for write operation**

Note: (*) represents the content of a storage location.

Storage elements: memory

Step 2

- **Memory (idealized)**
  - One input port/bus: Data In
  - One output port/bus: Data Out

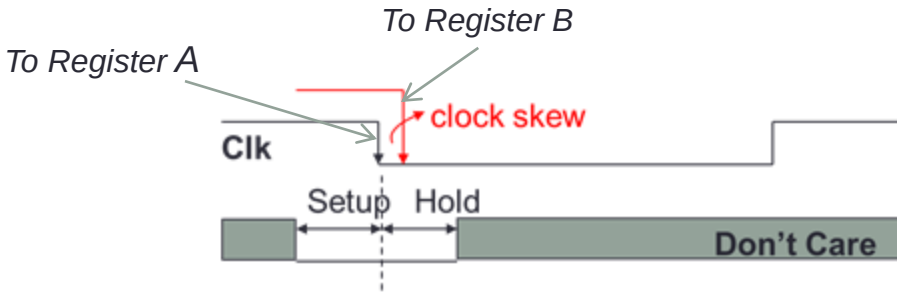


- **Memory word is selected by Address**
  - (Address) →Data Out
  - (Address) ← Data In if Write Enable = 1
- **Clock input (Clk)**
  - The Clk input is used for write operation

Typical clocking methodology

Step 2

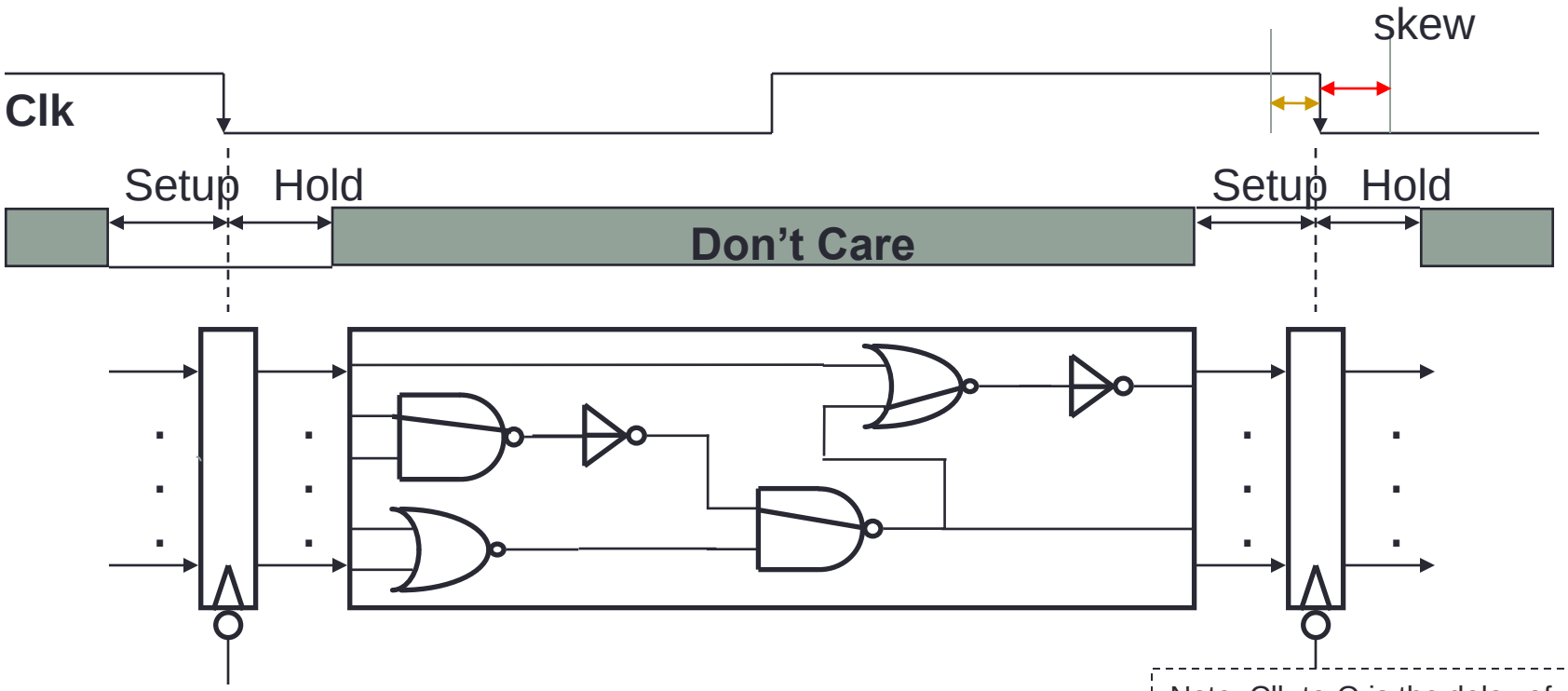
- **Edge triggered clocking**
  - **All storage elements are clocked by the same clock edge.**
    - **Simple and robust**
- **Issues need to be considered**
  - **Clock skew**
    - **difference in clock arrival time at different storage components**
  - **Setup time**
    - **Period of stable input for the register to read**
  - **Hold time**
    - **Period of stable input for the register output**



Typical clocking methodology (cont.)

Step 2

- **Two requirements for saving a new data into a register:**
  - **Cycle Time ≥ Clk-to-Q + Longest Delay Path + Setup + Clock Skew**
  - **(Clk-to-Q + Shortest Delay Path - Clock Skew) > Hold Time**



Note: Clk-to-Q is the delay of register

Step 3

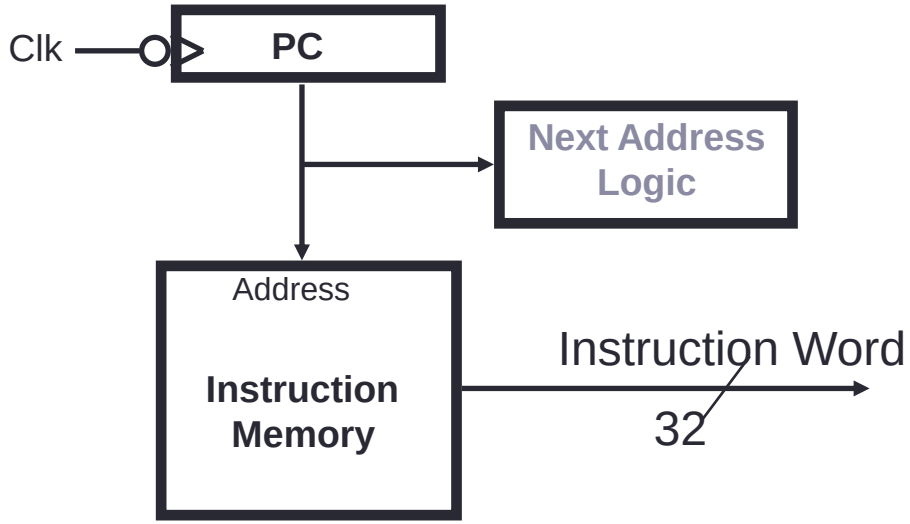
- **Assemble datapath to meet the requirements**
  - **Put the selected components together based on the register transfer requirements**
  - **It can start with each step of the instruction execution cycle**
    - **Instruction Fetch**
    - **Read Operands**
    - **Execute Operation**
    - **Next Instruction**
  - **See next a few slides for details**

For all instructions

Step 3

- **Instruction fetch unit:**
  - **At the start of clock cycle, fetch instruction:**

    MEM[PC]
  - **At the end of the cycle, update the program counter:**
    - **For sequential code execution: PC ←PC + 4**
    - **For branch: PC ← "something else"**
      - Will be covered later

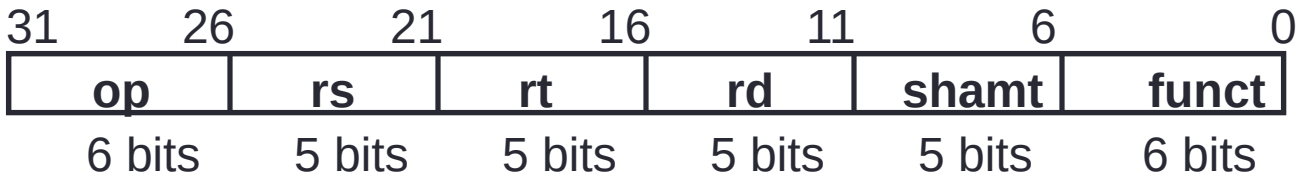


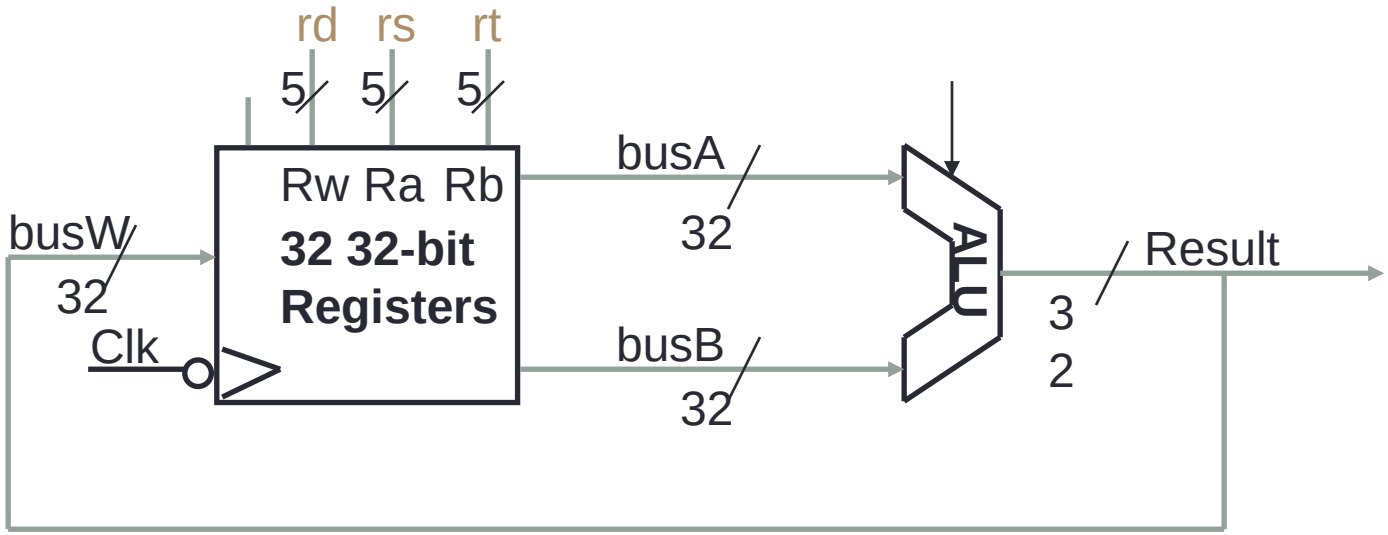
For *ADDU* and *SUBU*

Step 3

R[rd] ← R[rs] op R[rt] ***e.g. ADDU rd, rs, rt***

- **Ra, Rb, and Rw come from instruction's rs, rt, and rd fields**

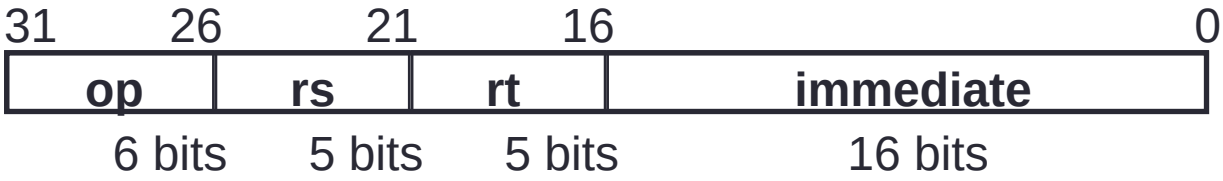


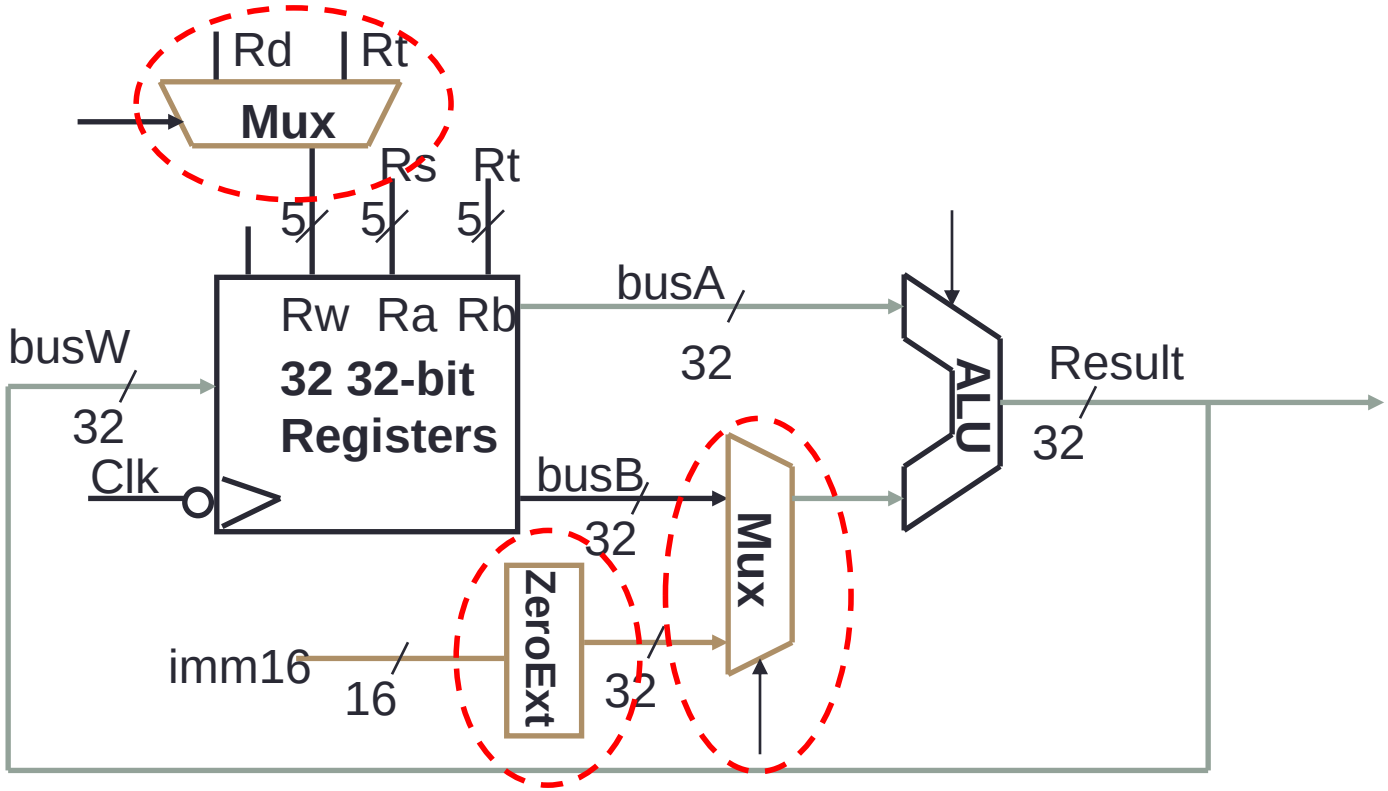


For *ORi*

Step 3

R[rt] ← R[rs] op ZeroExt[imm16] ***e.g. ORi rt, rs, imm16***

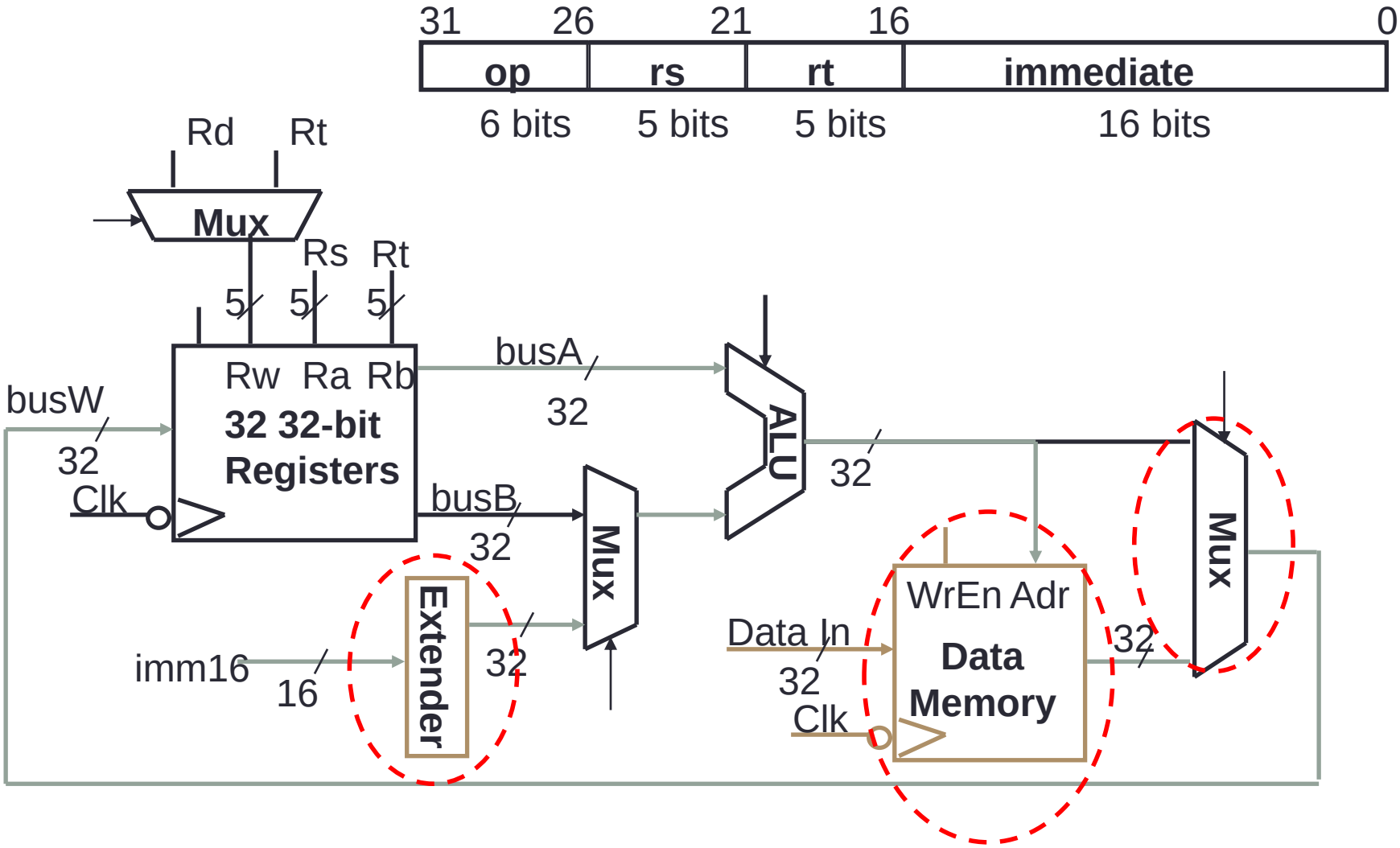




For *LW*

Step 3

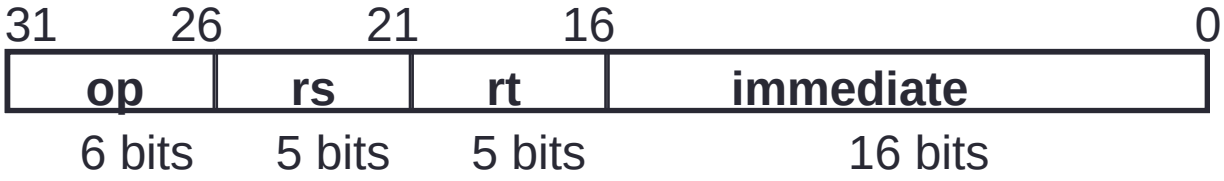
R[rt] ← MEM[R[rs] + SignExt[imm16]] ***e.g. LW rt, rs, imm16***

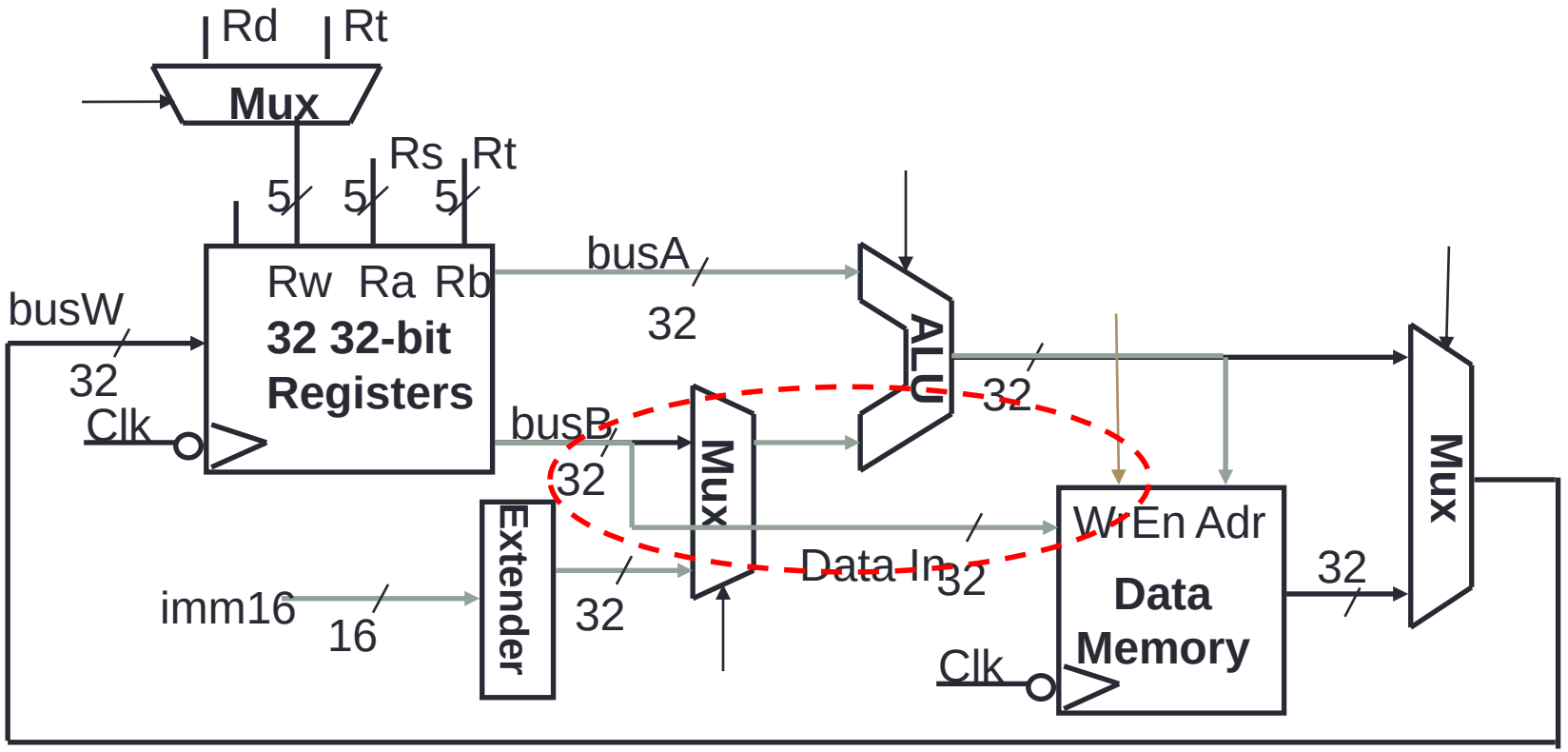


For *SW*

Step 3

MEM[R[rs] + SignExt[imm16]] ← R[rt] e.g. SW rt,rs,imm16

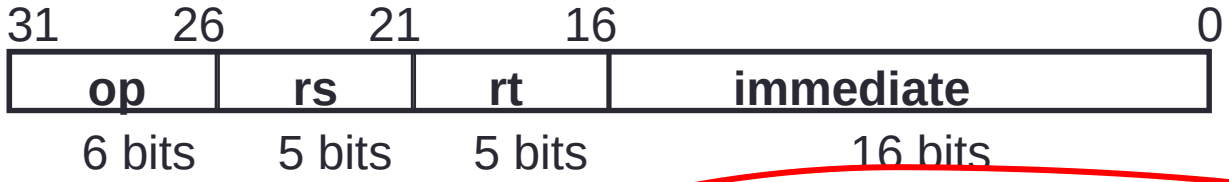


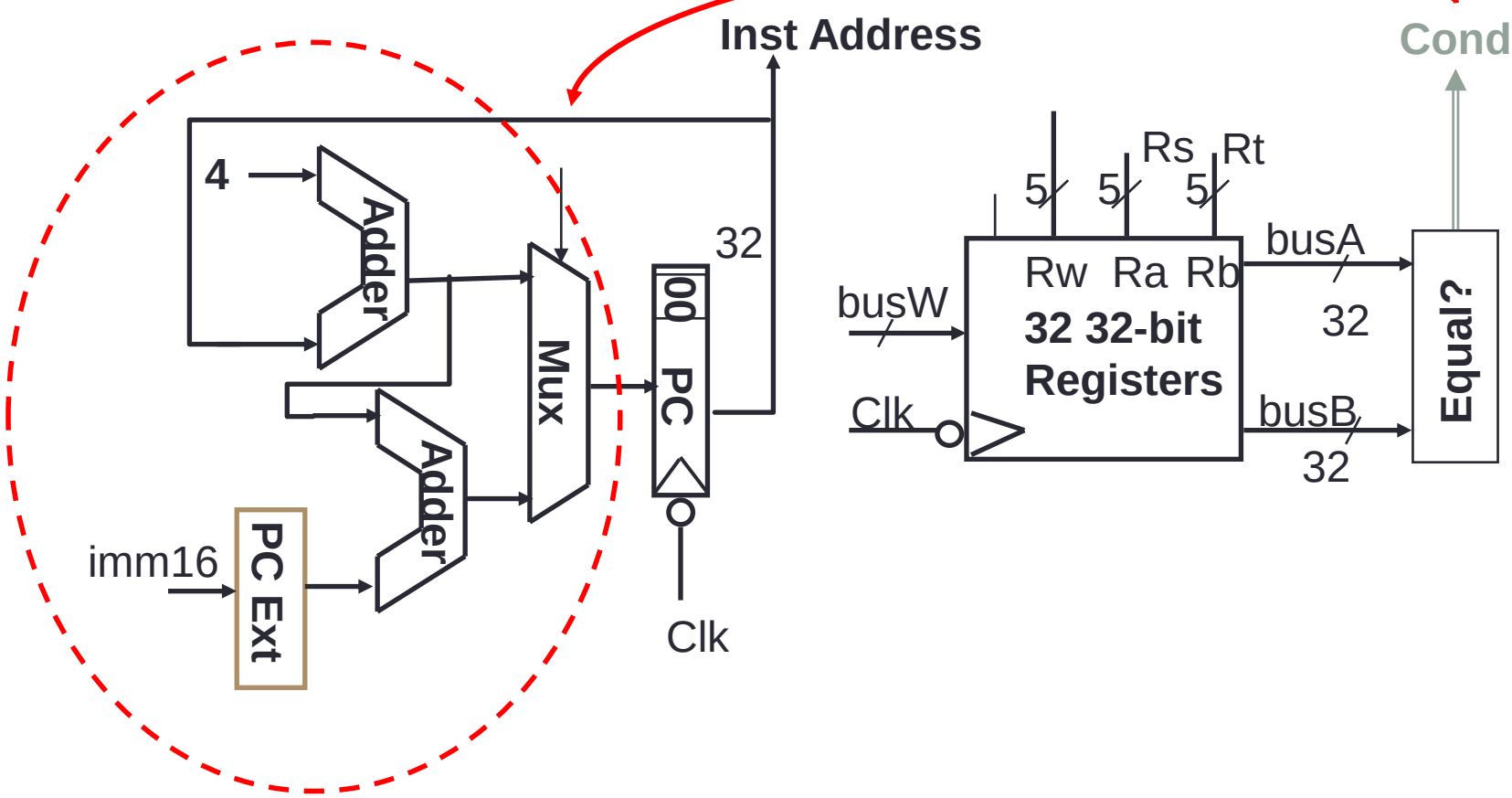


For *BEG*

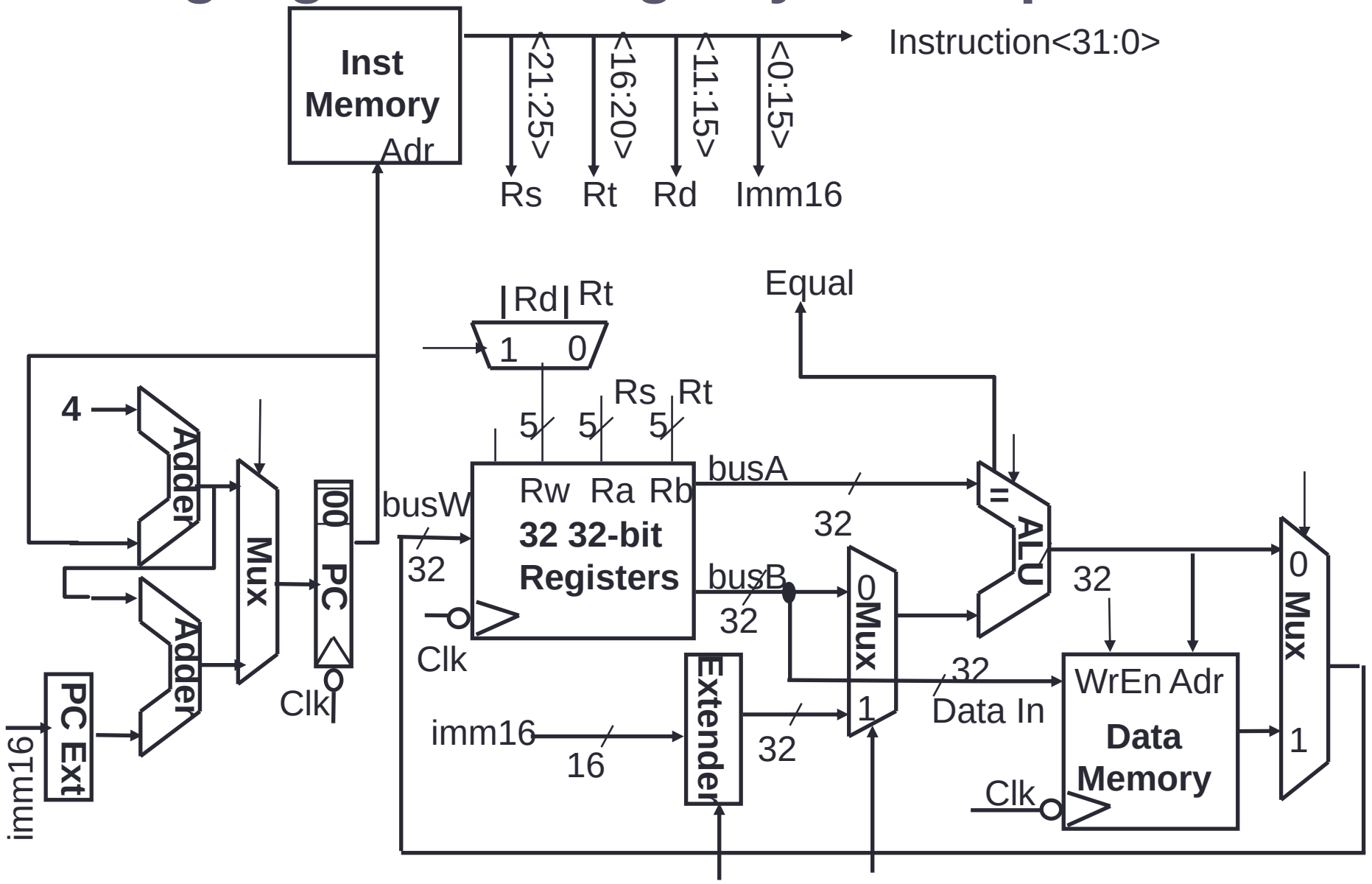
Step 3

BEG rs, rt, imm16 (Datapath generates the condition, equal)





Putting together: a single cycle datapath



Instruction encoding

- **Instruction encoding uses binary code to represent operations and operands.**
  - **See MIPS reference data sheet in the textbook for MIPS instruction encoding**
  - **Example**

Instr.	R-type	ori	lw	sw	beq	jump
op	000000	001101	100011	101011	000100	000010

R-type Instr.	add	sub	and	or
funct	100000	100010	100100	100101

- **Control unit design is closely related to instruction encoding**
  - **Here we only discuss how to design the control unit given the MIPS encoding**

Recall: Typical steps of processor design (this lecture)

- **Assume ISA is given. To build a processor, we basically go through the following steps:**

For datapath

1. **Analyse instruction set to determine datapath requirements**
2. **Select a set of components for the datapath and establish clocking methodology**
3. **Assemble datapath to meet the requirements**

For control

4. **Analyse implementation of each instruction to determine control points**
5. **Assemble the control logic**

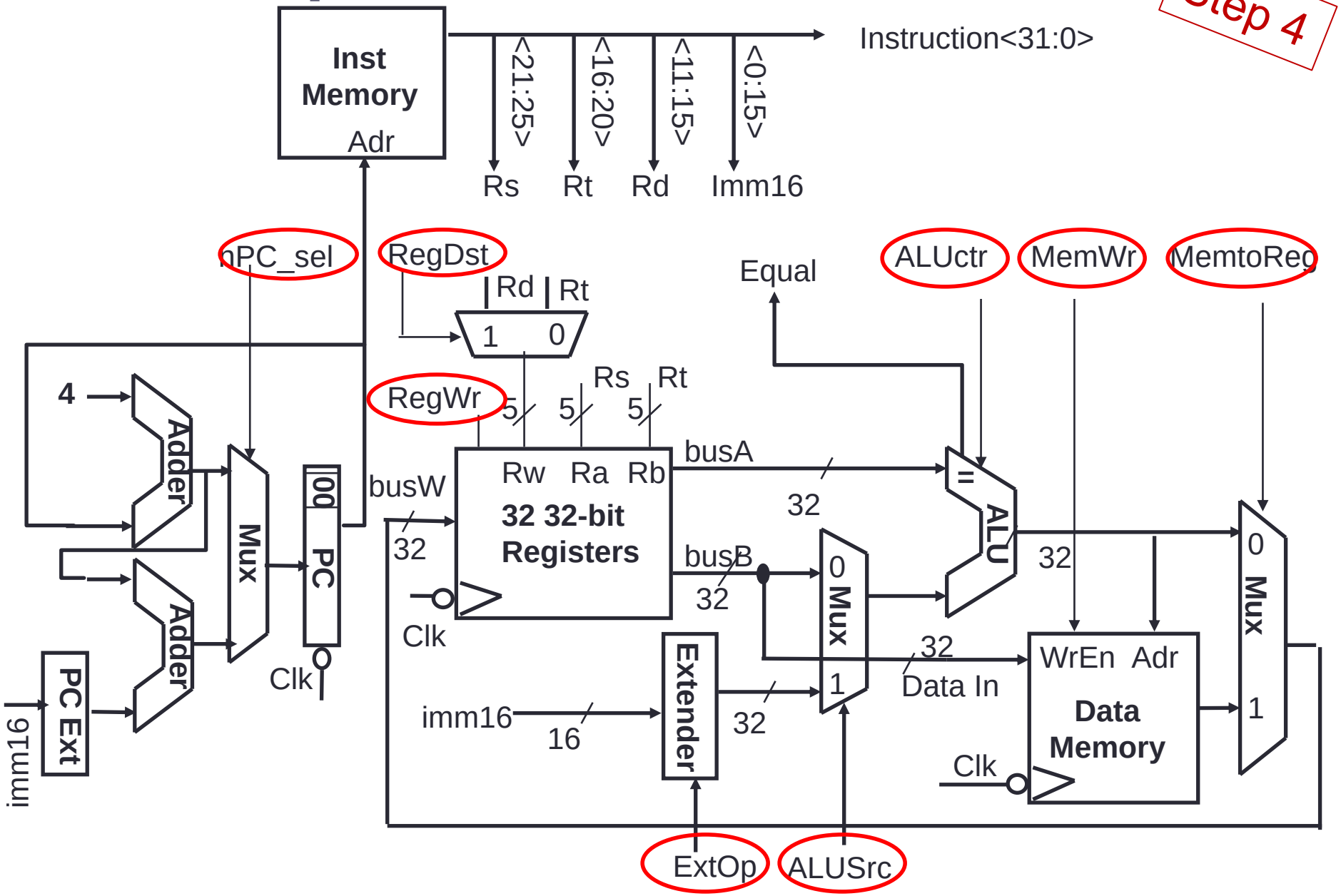
- **A demonstration is given below.**

Step 4

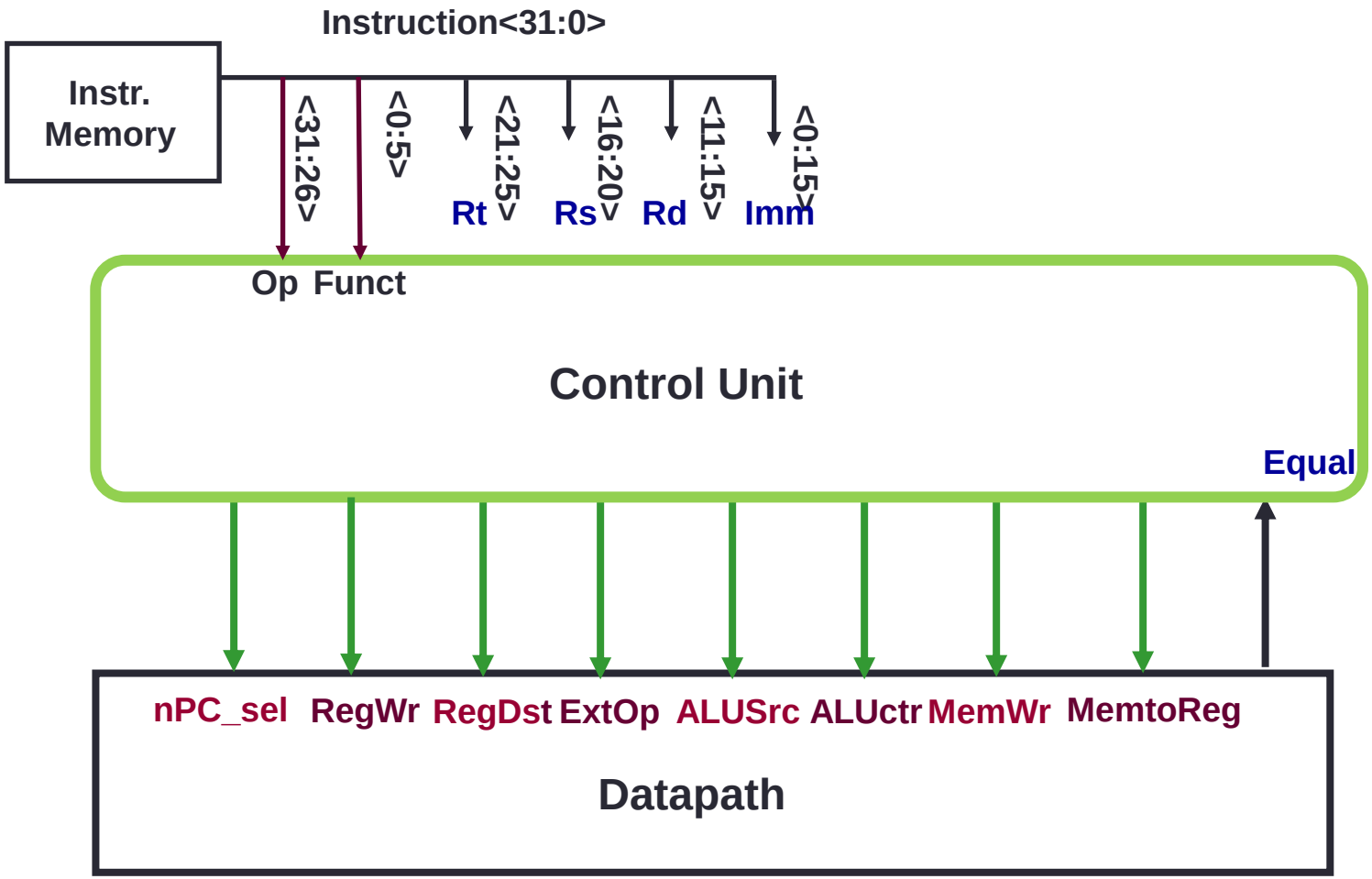
- **Analyse implementation of each instruction to determine control points**
  - **Here we consider single cycle datapath**
    - **Control needs to make sure each instruction to be completed in one clock cycle**

Control points

Step 4

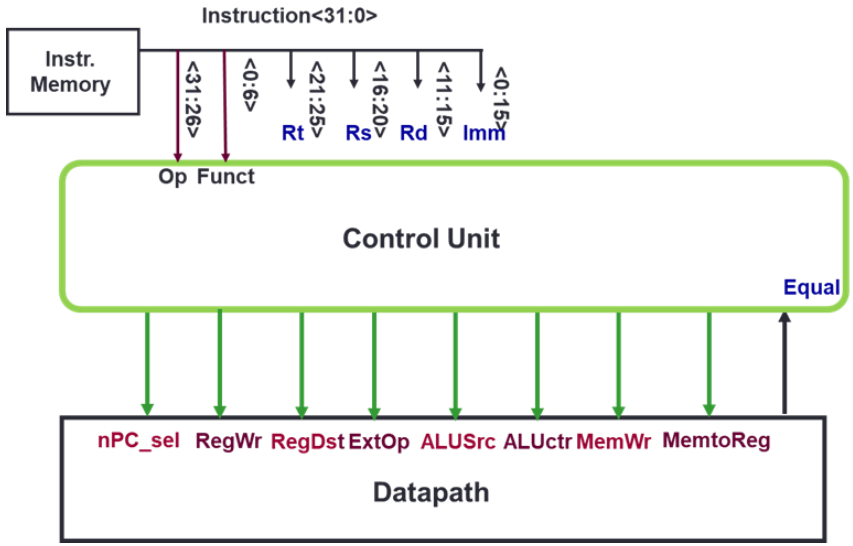


What do we need to do next?



Step 5

- **Assemble control logic**
  - **Determine the logic function of each control signal**
  - **Design to implement those functions**
- **The logic can be very large**
  - **Should be carefully designed**



Logic for each control signal

Step 5

nPC_sel: if (OP == BEQ) then Equal else 0

ALUsrc: if ((OP == “000000”)|(OP == BEQ)) then “regB”
else “Imm”

ALUctr: if (OP == “000000”) then Funct
elseif (OP == ORi) then “or”
elseif (OP == BEQ) then “sub”
else “add”

ExtOp: if (OP == ORi) then “zero” else “sign”

MemWr: if (OP == SW)

MemtoReg: if (OP == LW)

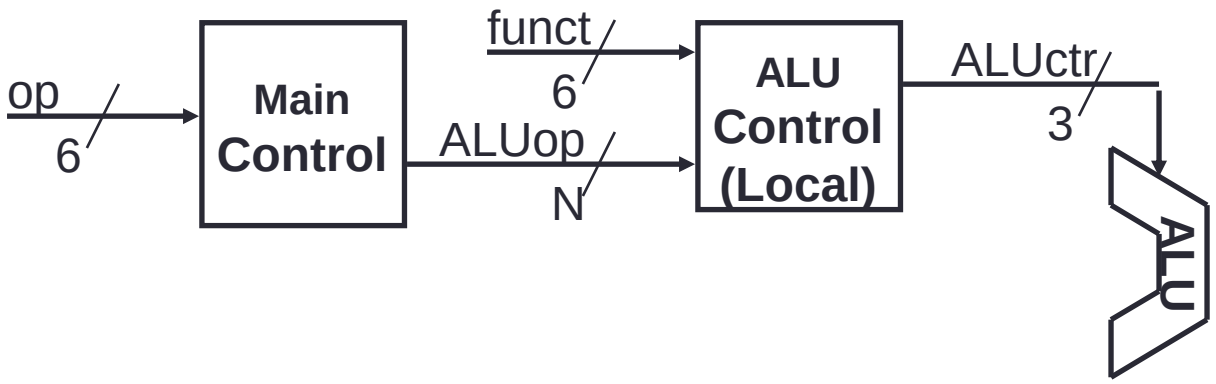
RegWr: if ((OP == SW) || (OP == BEQ)) then 0 else 1

RegDst: if ((OP == LW) || (OP == ORi)) then 0 else 1

Step 5

Control logic with two levels

- **A single level of control logic may be costly**
- **Local decoding for ALU operations makes design simpler, smaller and faster.**



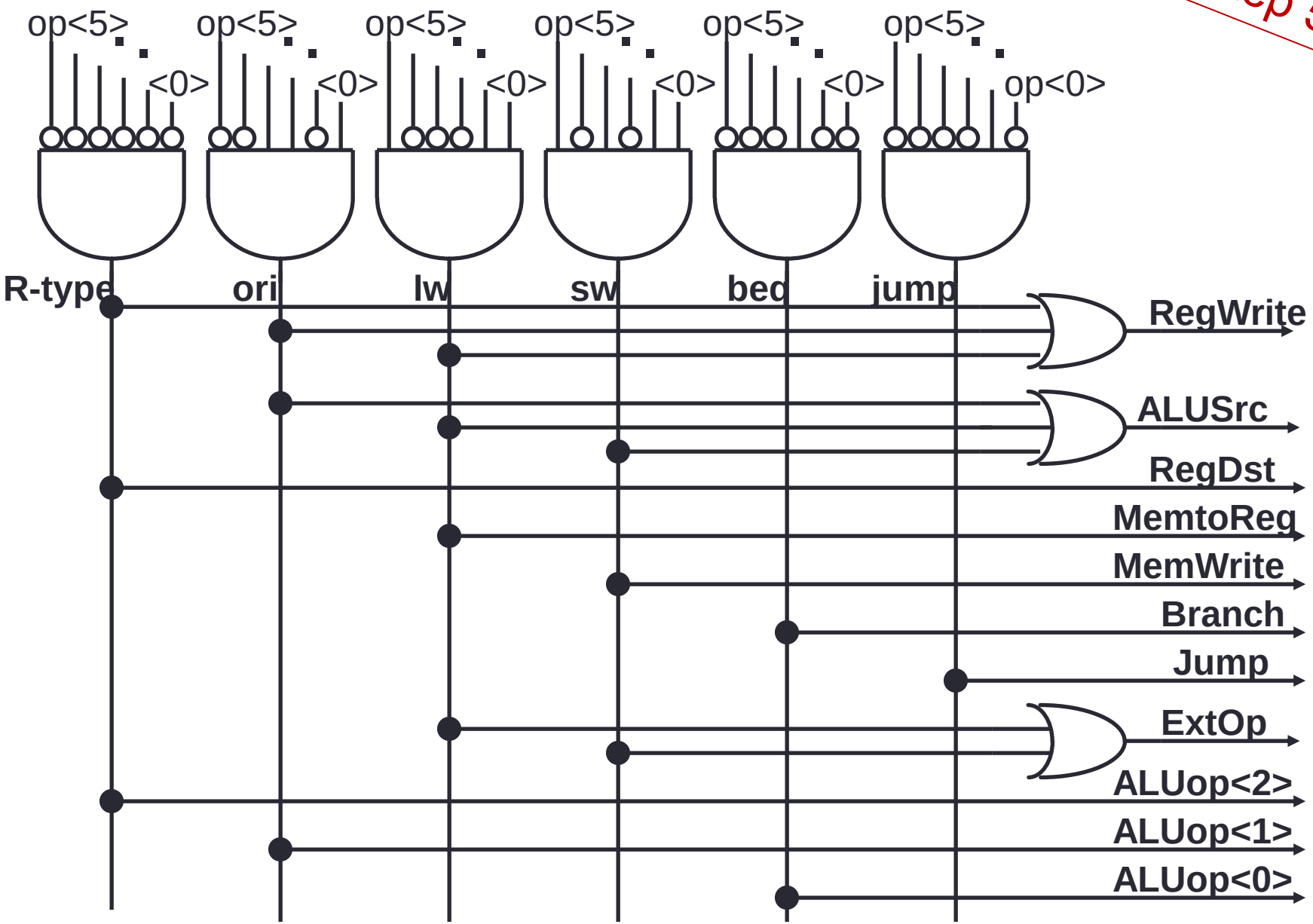
Truth table for main control

Step 5

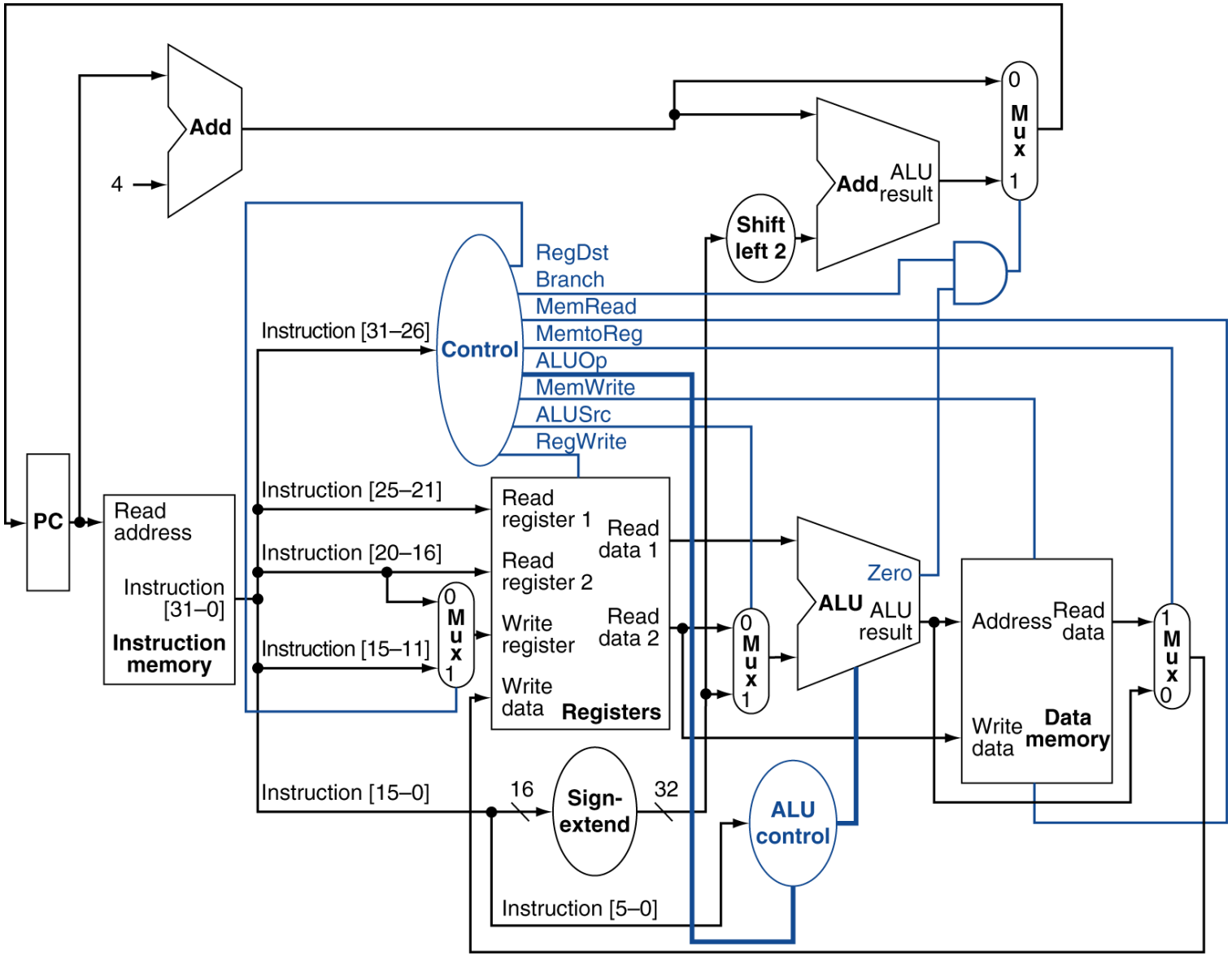
op	000000	001101	100011	101011	000100	000010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop(Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

PLA implementation of the main control

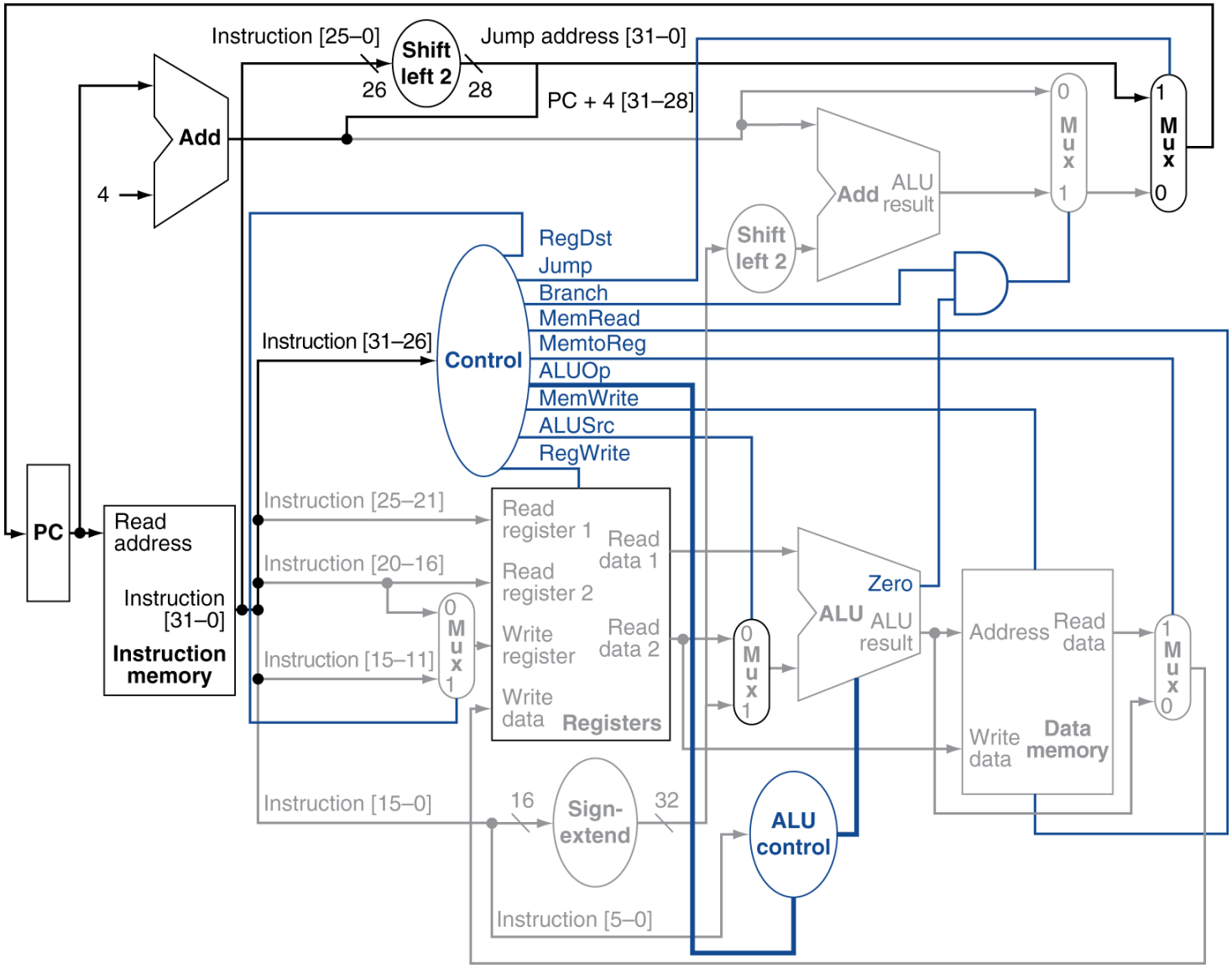
Step 5



Putting it all together: a single cycle processor

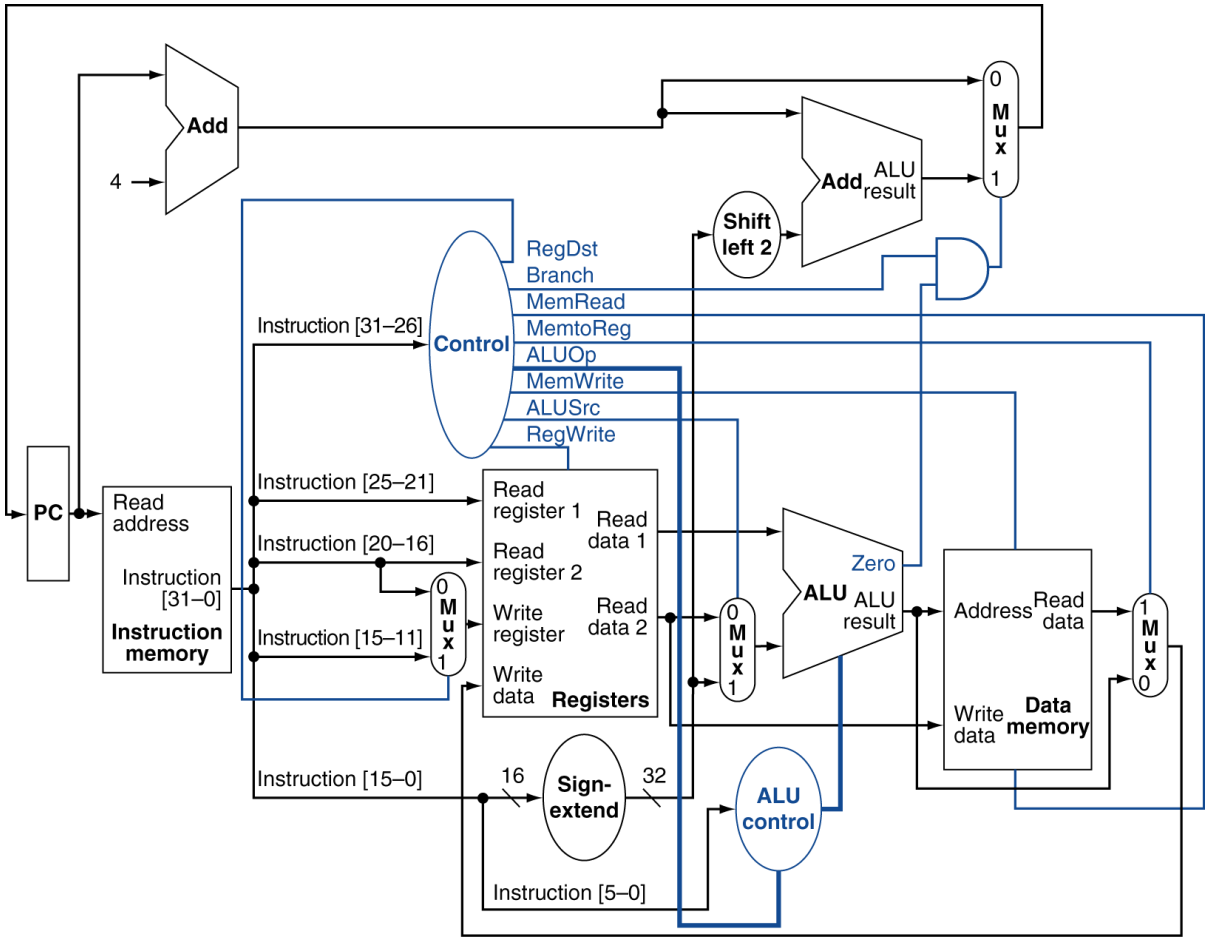


Single Cycle Processor with Jump Added



In-class exercise 1

- **Given the processor design below, identify the datapath components for R-type instructions and BEQ instruction.**



In-class exercise 2

- **Can we use the MIPS-Lite processor we just built to solve the following problem? Why? Assume x and y are unsigned integers and can stored in registers.**

```
if (x > y)
        y++
else
        y--
```