# Computer Architecture (CS F342)

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Upendra Singh, BITS Pilani

# Computer Architecture Lab


Instructor: Upendra Singh

p20170428@pilani.bits-Pilani.ac.in

TA1: ALARK FAJALIA

f20180296@pilani.bits-pilani.ac.in

TA2: AVADH RAJESH HARKISHANKA

f20180322@pilani.bits-pilani.ac.in

TA3: JVN Saketh Ram

h20200242@pilani.bits-pilani.ac.in


Section: P3 and P6

Day & Time: Tuesday, 10,11 hrs (P3)
Tuesday, 7,8 hrs (P6)

Reference Books: "**Verilog HDL: A guide to Digital Design and Synthesis**" by Samir Palnitkar.

# Course Objectives

- Getting started with HDL program using Icarus Simulator
- Understand basic Verilog language primitives (e.g. module, data types, identifiers, vectors, registers, keywords etc.)
- To understand the various types of modelling

innovate achieve lead

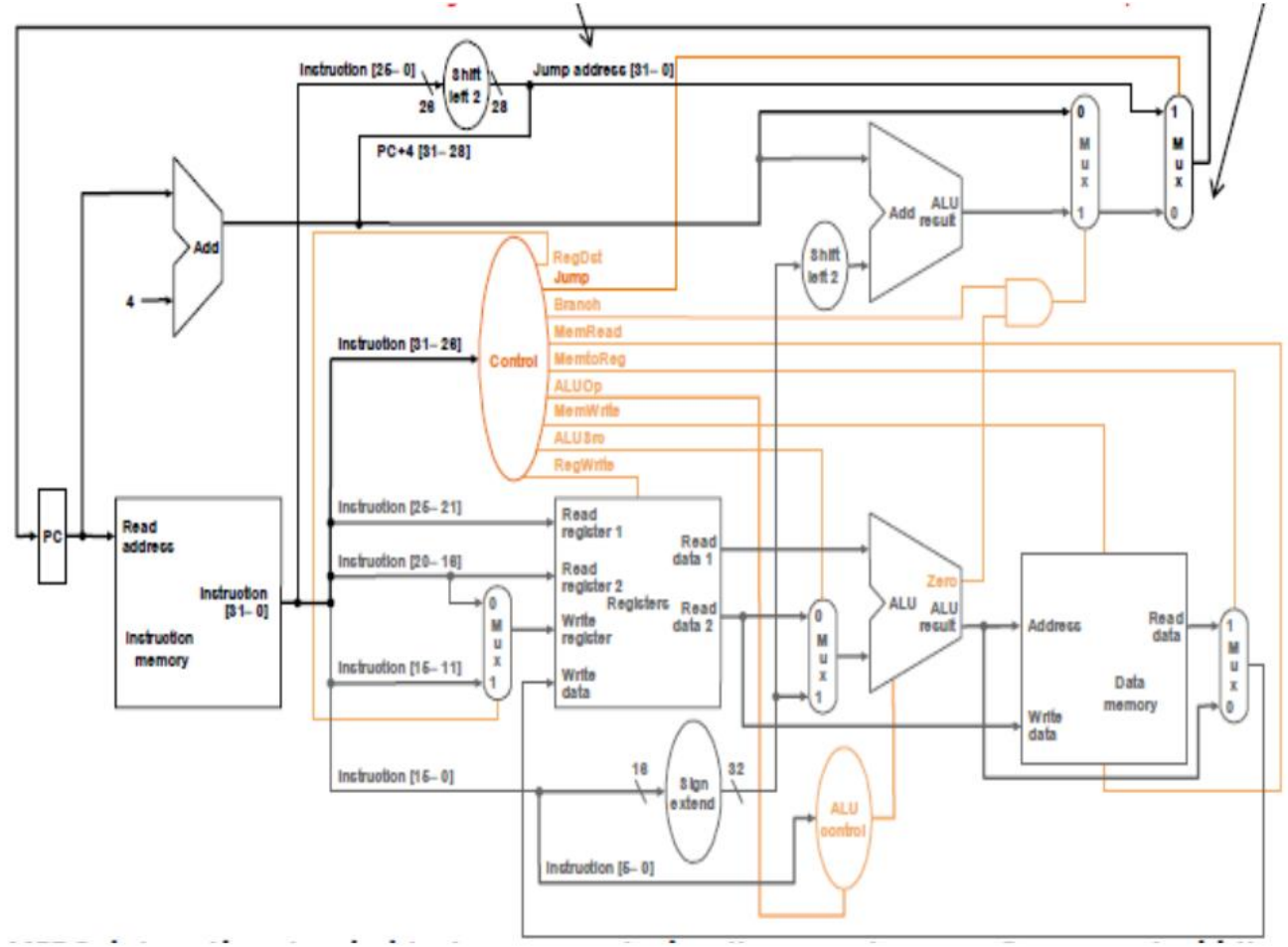
BITS Pilani

Pilani|Dubai|Goa|Hyderabad

# Lab – 6

- Single Cycle Datapath Design

# Single Cycle Datapath



# Single Cycle Datapath

- Datapath consists of the functional units of the processor.
  - Elements that hold data.
    - Program counter, register file, instruction memory, etc.
  - Elements that operate on data.
    - ALU, adders, etc.
  - Buses for transferring data between elements.
  - Control commands the Datapath regarding when and how to route and operate on data.

# Single Cycle Datapath

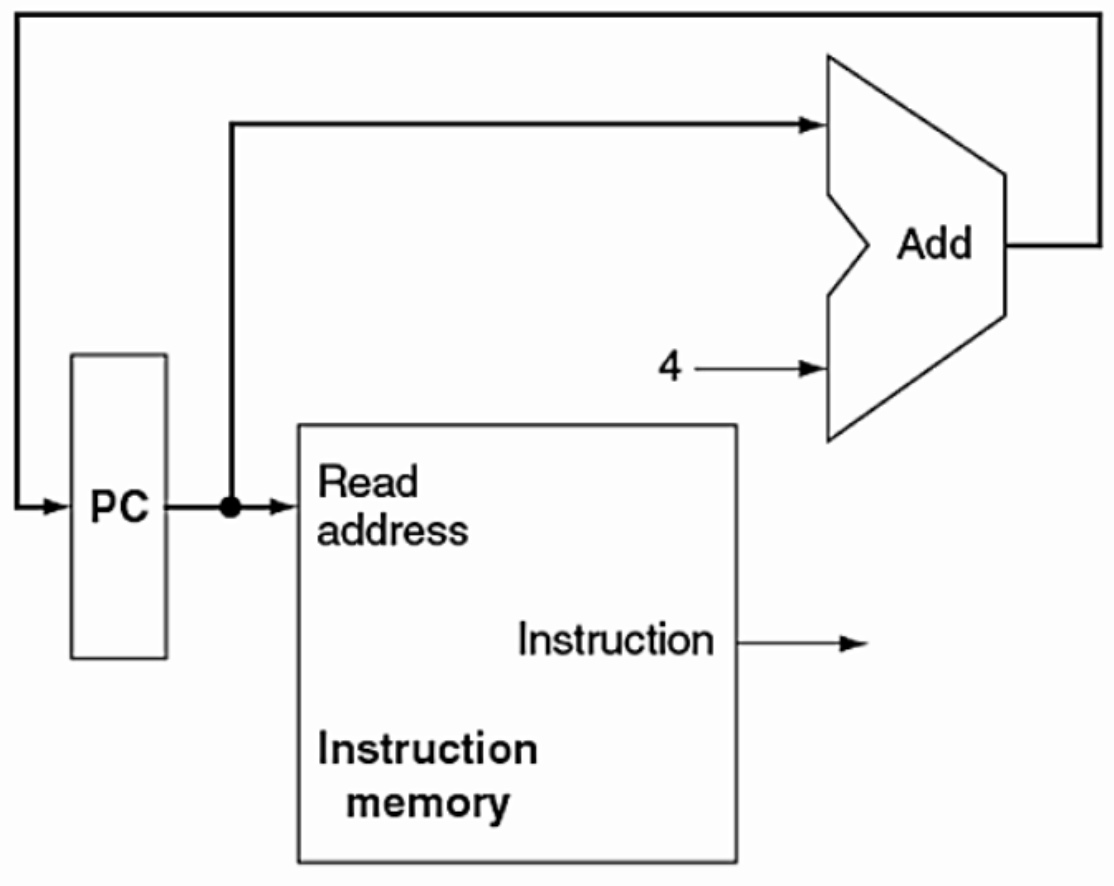
- To showcase the process of creating a datapath and designing a control, we will be using a subset of the MIPS instruction set. Our available instructions include:
  - add, sub, and, or, slt
  - lw, sw
  - beq, j

# Steps Taken to Execute a Program

- Now, we can talk about the general steps taken to execute a program.
  - Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory.
  - Instruction decoding: determine the fields within the instruction
  - Instruction execution: perform the operation indicated by the instruction.
  - Update the PC to hold the address of the next instruction.

# Steps Taken to Execute a Program

- Fetch the instruction at the address in PC.
- Decode the instruction.
- Execute the instruction.
- Update the PC to hold the address of the next instruction.




**Note: we perform PC+4 because MIPS instructions are word-aligned.**

# R-Format Instructions

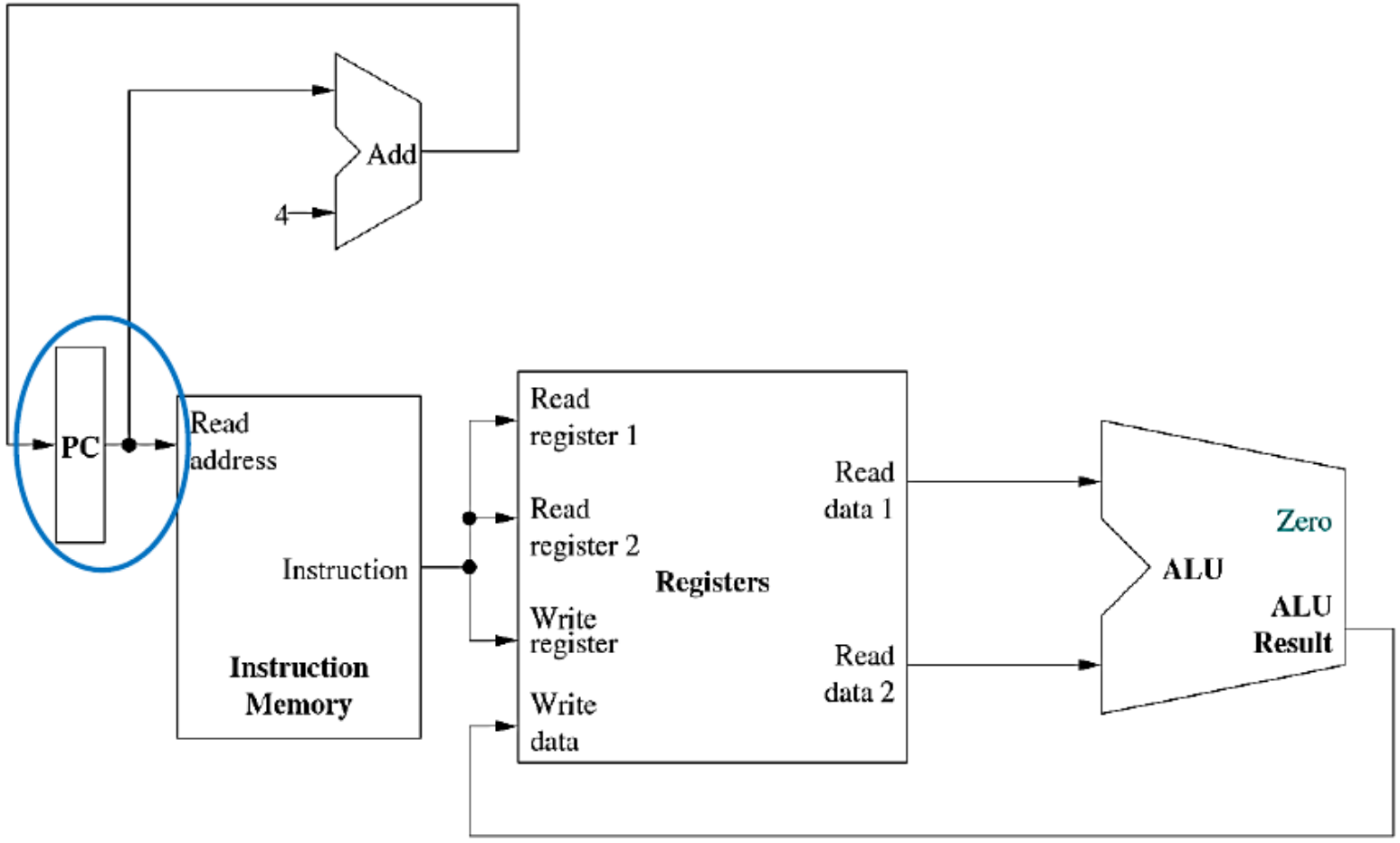
- All R-format instructions read two registers, rs and rt, and write to a register rd.

| Name | Fields | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| R format | op | rs | rt | rd | shamt | funct |

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

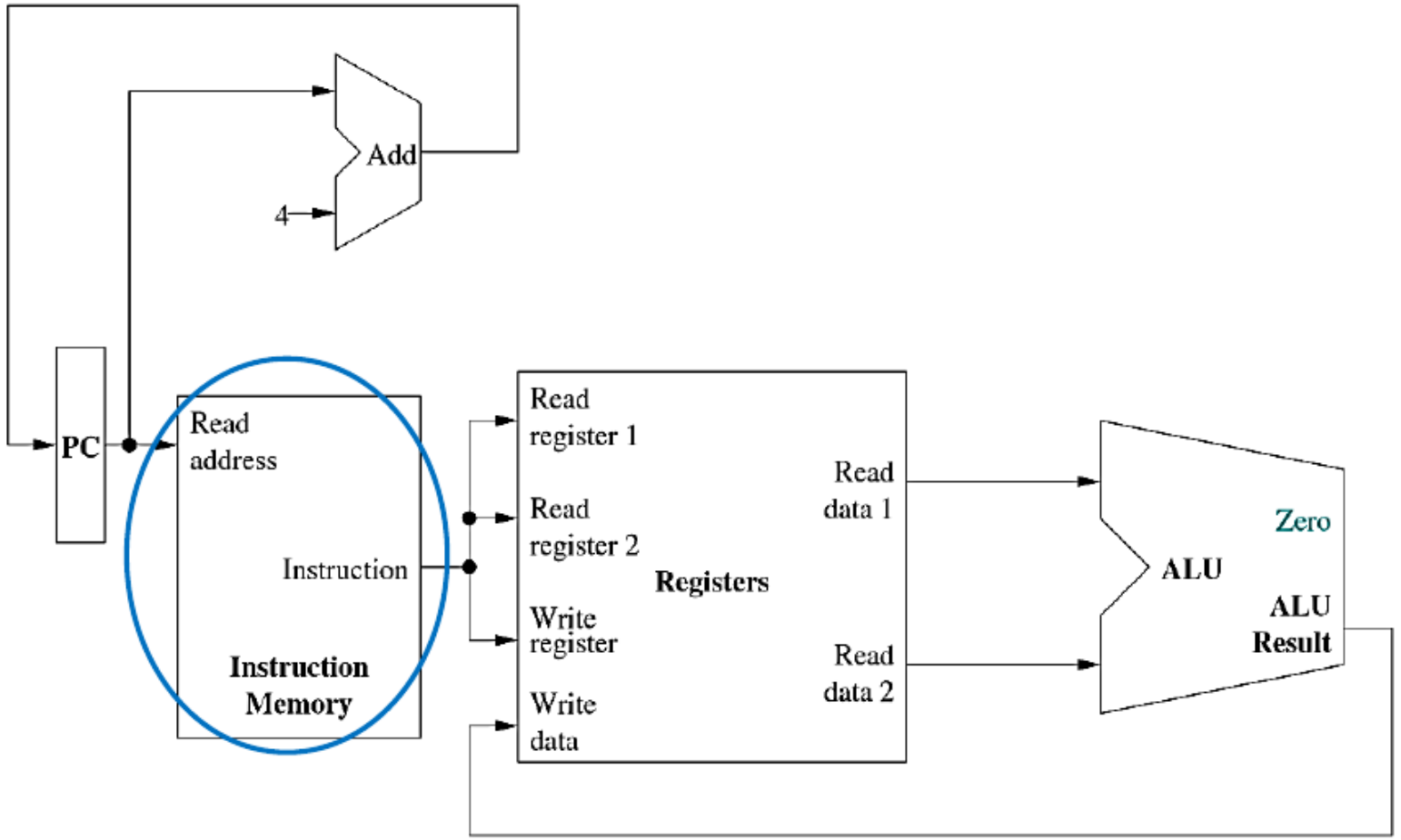
# Datapath for R-format Instructions.

1. Grab instruction address from PC.



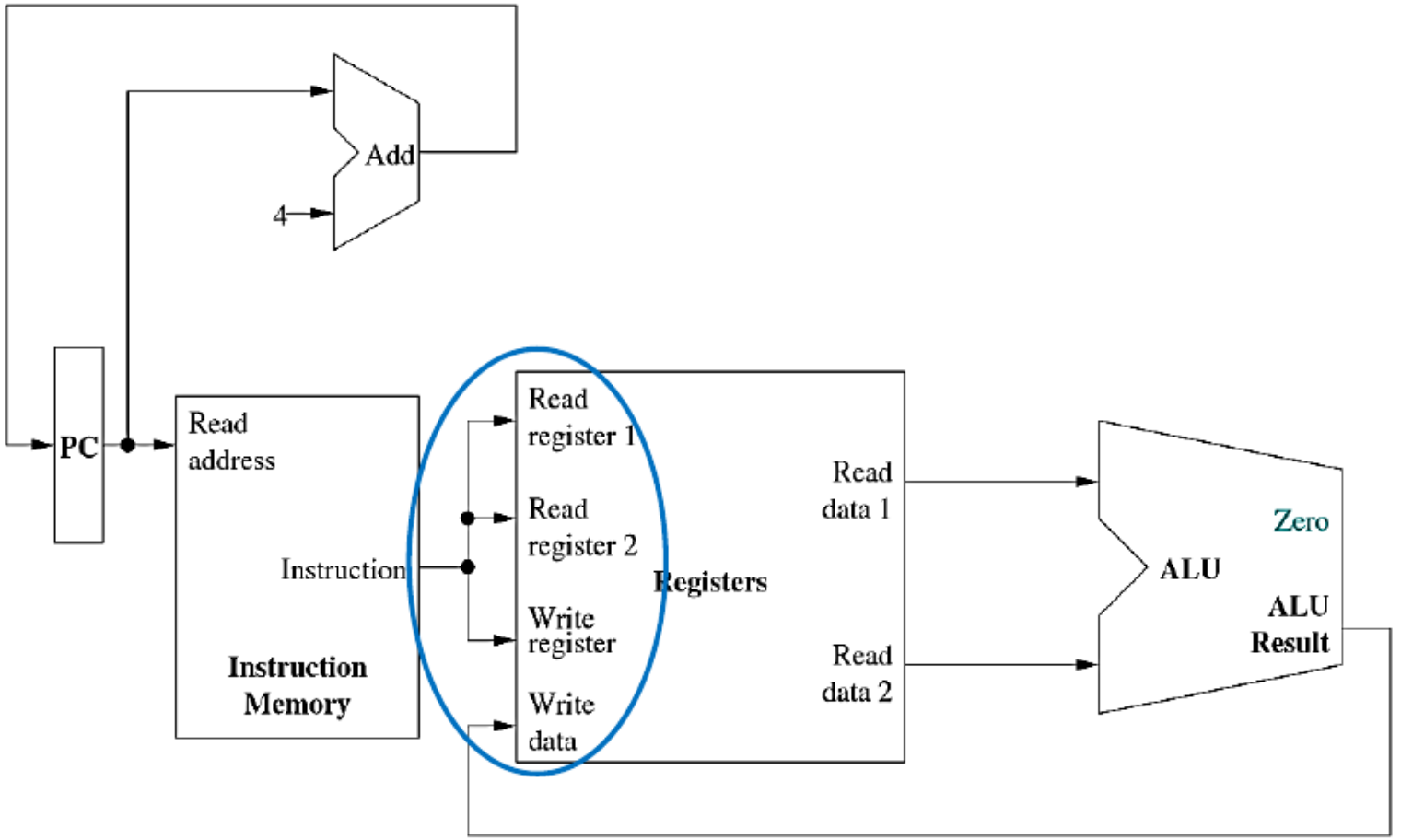
# Datapath for R-format Instructions.

2. Fetch instruction from instruction memory.
3. Decode instruction.



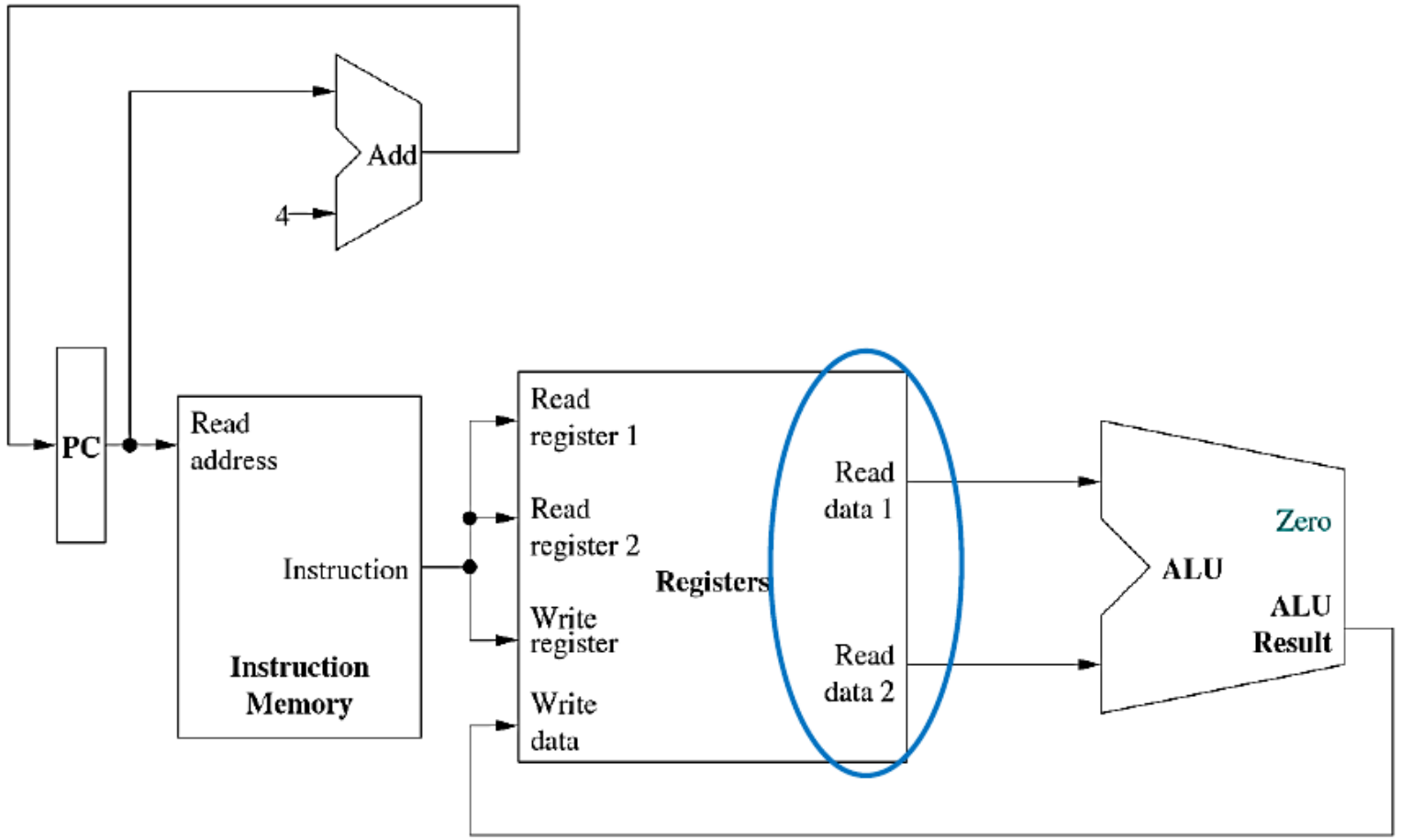
# Datapath for R-format Instructions.

4. Pass rs, rt, and rd into read register and write register arguments.



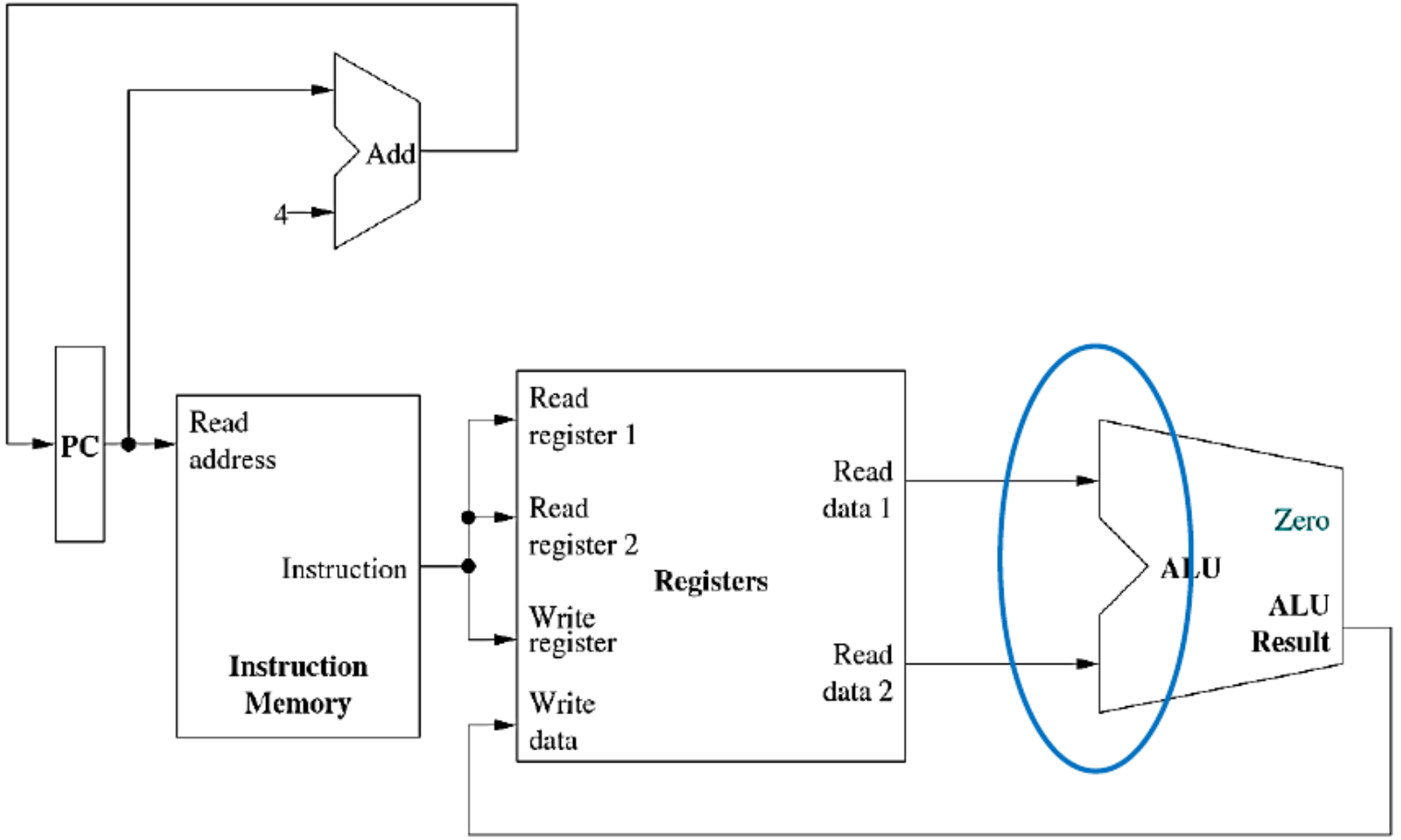
# Datapath for R-format Instructions.

5. Retrieve data from read register 1 and read register 2 (rs and rt).



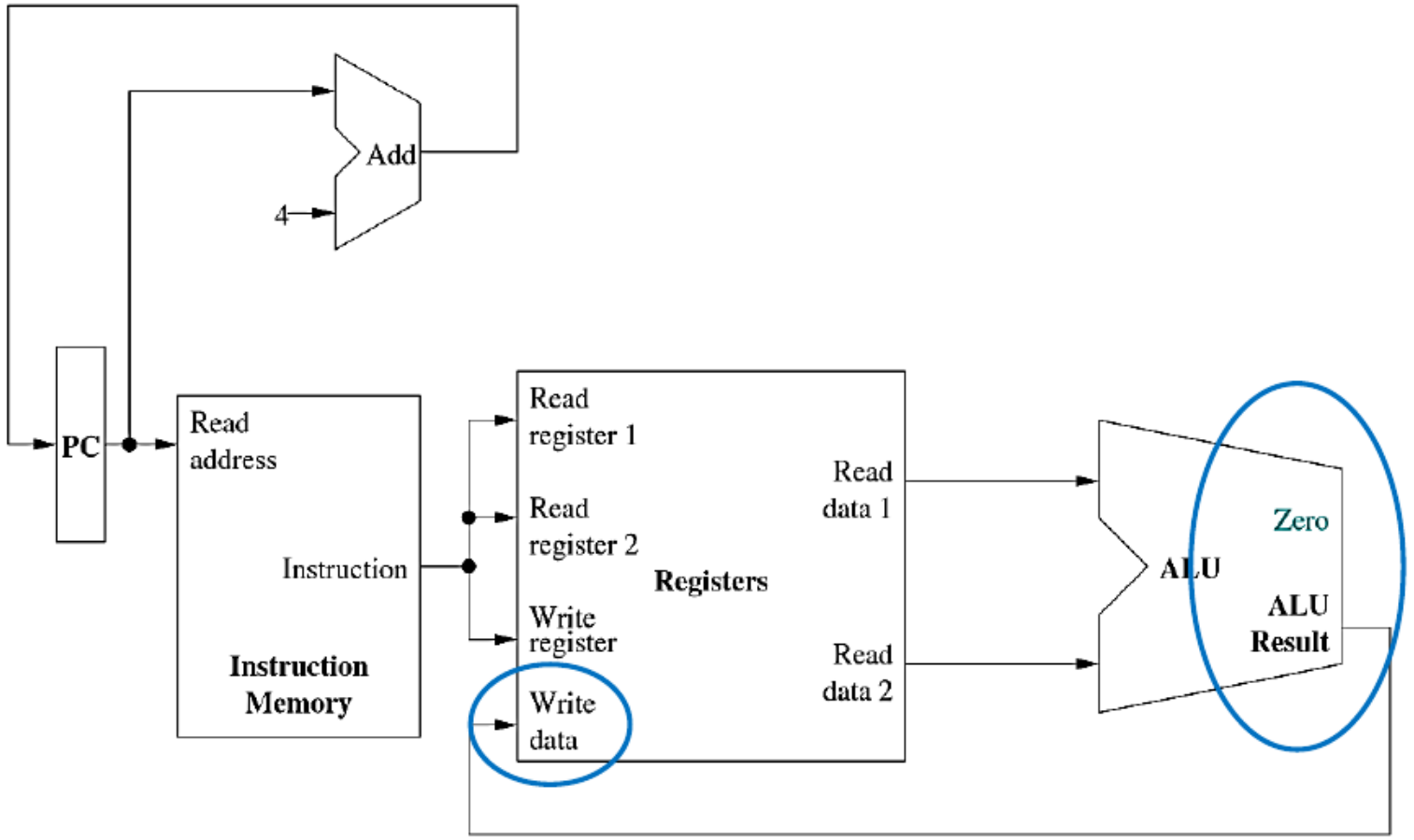
# Datapath for R-format Instructions.

6. Pass contents of rs and rt into the ALU as operands of the operation to be performed.



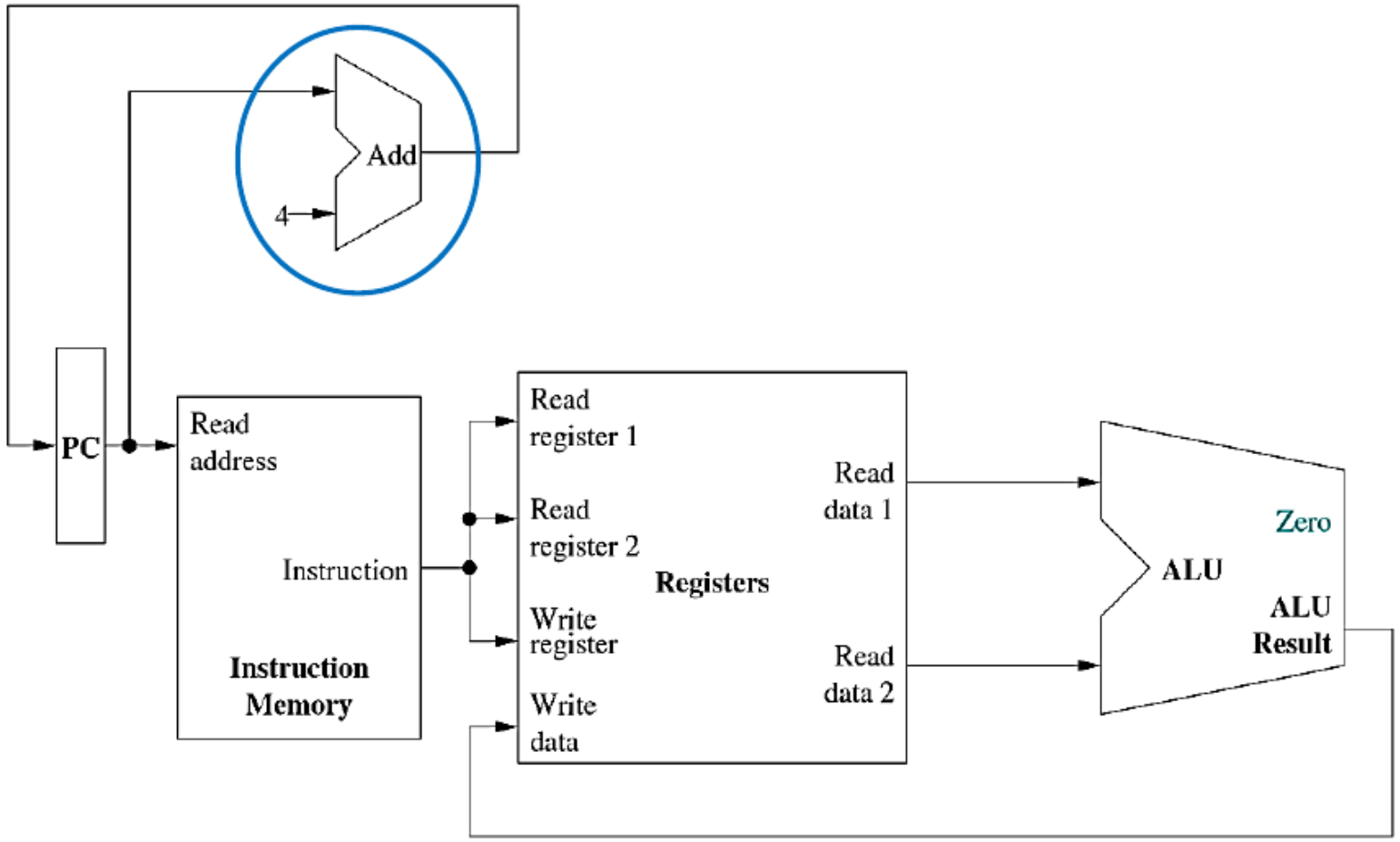
# Datapath for R-format Instructions.

7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the RegWrite bit set).



# Datapath for R-format Instructions.

8. Add 4 bytes to the PC value to obtain the word-aligned address of the next instruction.



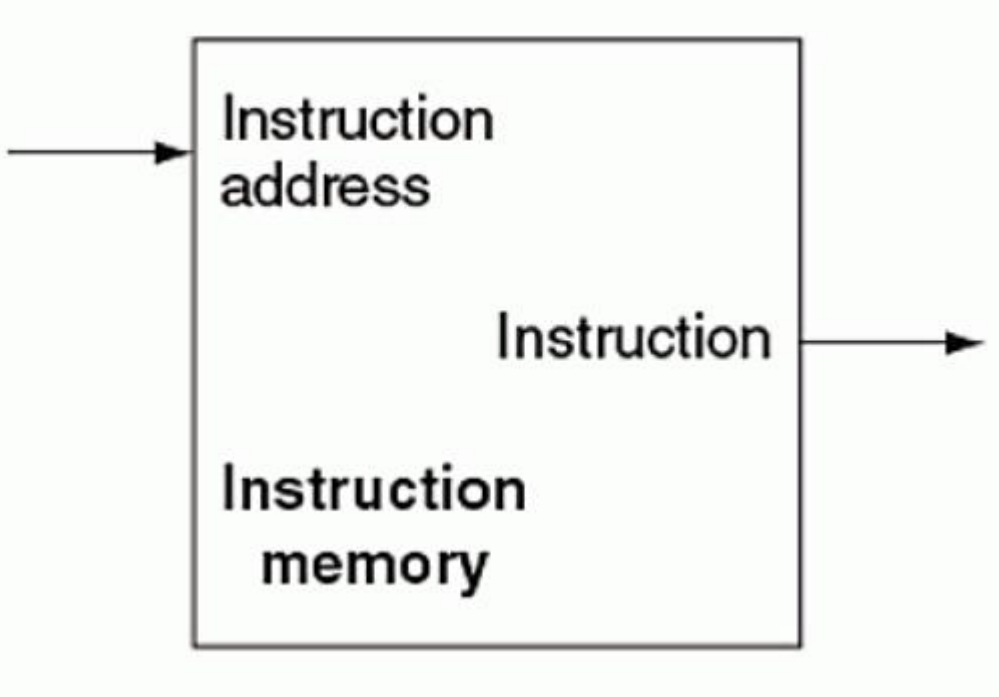
# Component list in SCD

**Following are the hardware components**

1. **Instructio**n Memory
2. **Register File** (Lab 5)
3. **ALU** (Lab 4)
4. **Control Unit and ALU Control** (Lab 4)
5. **Data Memory**
6. **Sign Extender**
7. **Shifter** (Lab 3)
8. **Program Counter**
9. **Adders (required for computing next** PC) (Lab 2)
10. **Multiplexers** (Lab 1)

# Instruction Memory

- First, we have instruction memory.
  - Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.



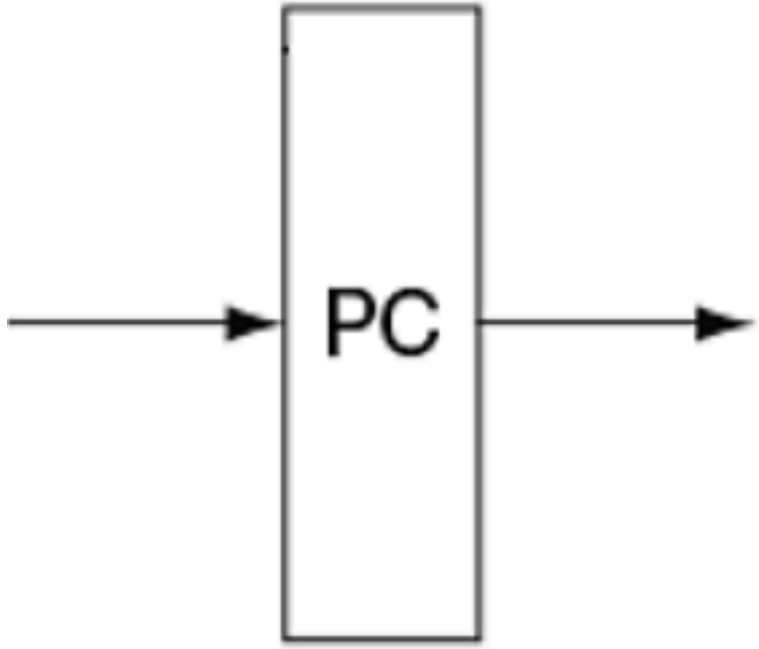
# Instruction Memory

```
module Instruction_Memory(Inst, PC, clock);
    input[31:0] PC;
    input clock;
    output[31:0] Inst;
    reg [31:0] memory [0:31];
    reg [31:0] Inst;
  integer addr;
  initial begin
    memory[0] = 32'b00000000000000000000000000000000;  // nop
    memory[1] = 32'b00000000000000000000000000000000;  // nop
    memory[2] = 32'b00000000000000000000000000000000;  // nop
    memory[3] = 32'b10001100000100010000000000001000;  // lw  $s1($17), 8($0)
    memory[4] = 32'b10001100000100100000000000000100;  // lw  $s2($18), 4($0)
    memory[5] = 32'b00000010001100100100000000100000;  // add $t0($8), $s1($17), $s2($18)
```

```
    always @(posedge clock) begin
      addr = PC[31:0];
      Inst = memory[addr/4];
      end
```

# Program Counter

- Next, we have the program counter or PC.
  - The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.
    - Normally PC increments sequentially except for branch instructions

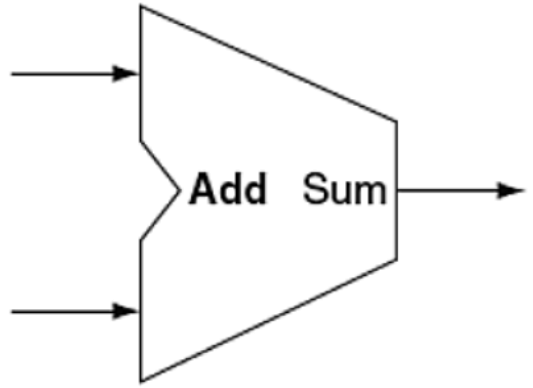


# Program Counter

```
module PC(count, clock, reset);
   input       clock, reset;
   output      [31:0]  count;
   reg    [31:0]  count;
   always @(posedge clock)
      if(!reset)
         count = count + 1;
      else
         count = 0;
endmodule
```

# FADDER32 (2 modules)

- The adder is responsible for incrementing the PC to hold the address of the next instruction.
- It takes two input values, adds them together and outputs the result.

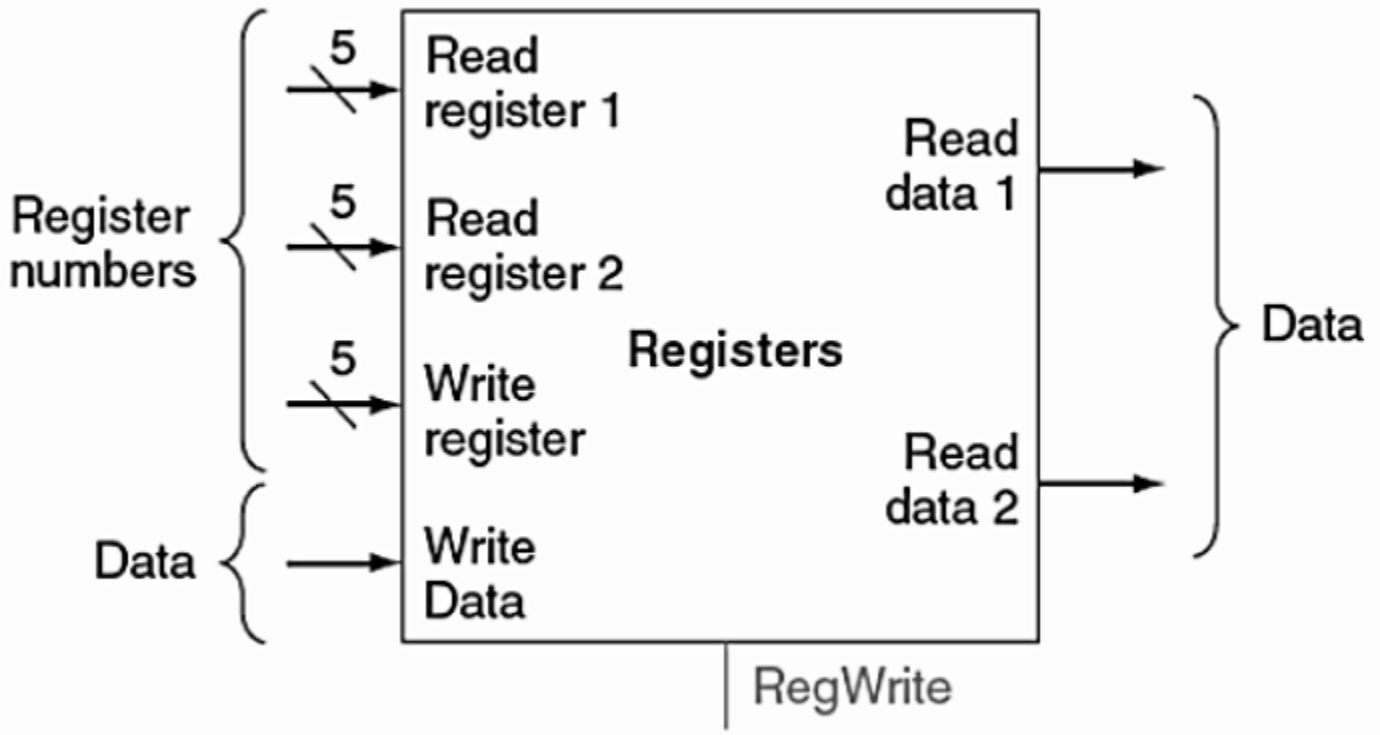



```
module FADDER32(carry, sum, A, B, CarryIn);
  input [31:0] A, B;
  input CarryIn;
  output  [31:0]sum;
  output  carry;
  wire  c1, c2, c3;
  FADDER8  mod1(c1, sum[7:0], A[7:0], B[7:0], CarryIn);
  FADDER8  mod2(c2, sum[15:8], A[15:8], B[15:8], c1);
  FADDER8  mod3(c3, sum[23:16], A[23:16], B[23:16], c2);
  FADDER8  mod4(carry, sum[31:24], A[31:24], B[31:24], c3);
endmodule
```

# Register File

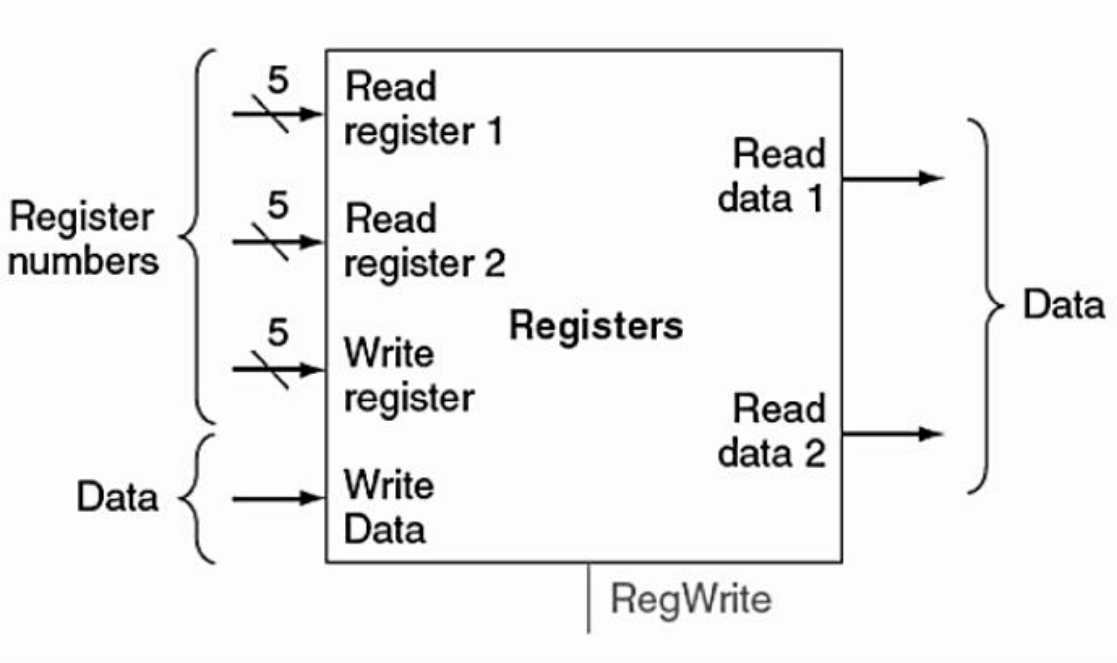
- To support R-format instructions, we'll need to add a state element called a register file. A register file is a collection readable/writeable registers.

- `Read register 1` – first source register. 5 bits wide.
- `Read register 2` – second source register. 5 bits wide.
- `Write register` – destination register. 5 bits wide.
- `Write data` – data to be written to a register. 32 bits wide.



# Register File

- At the bottom, we have the RegWrite input. A writing operation only occurs when this bit is set.
- The two output ports are:
  - Read data 1 – contents of source register 1.
  - Read data 2 – contents of source register 2.



# Register File

```
module d_ff(q, d, clock, reset);
  input d, clock, reset;
  output  q;
  reg q;
  always @ (posedge clock or negedge reset)
  if(~reset)
    q = 1'b0;
  else
    q = d;
endmodule
```

```
module reg_32bit(q, d, clock, reset);
  input [31:0]  d;
  input clock, reset;
  output  [31:0]  q;
  genvar j;
  generate
    for(j = 0; j < 32; j = j + 1) begin:  d_loop
      d_ff ff(q[j], d[j], clock, reset);
    end
  endgenerate
endmodule
```

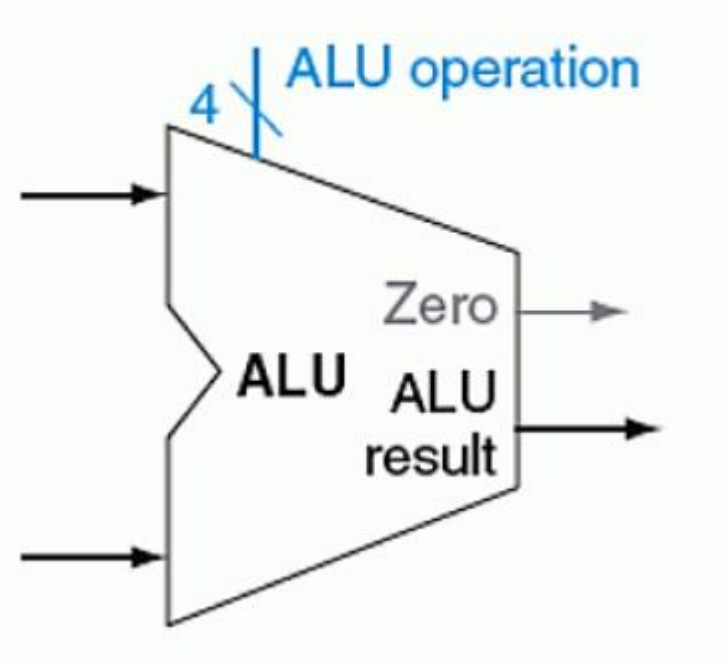
# Register File

- Thirty two registers of 32-bit wide (32 instances)

```
reg_32bit r0(w0, WriteData, c0, Reset);
reg_32bit r1(w1, WriteData, c1, Reset);
reg_32bit r2(w2, WriteData, c2, Reset);
reg_32bit r3(w3, WriteData, c3, Reset);
```

# ALU Unit

- To actually execute R-format instructions, we need to include the ALU element.
  - The ALU performs the operation indicated by the instruction.
  - It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value.

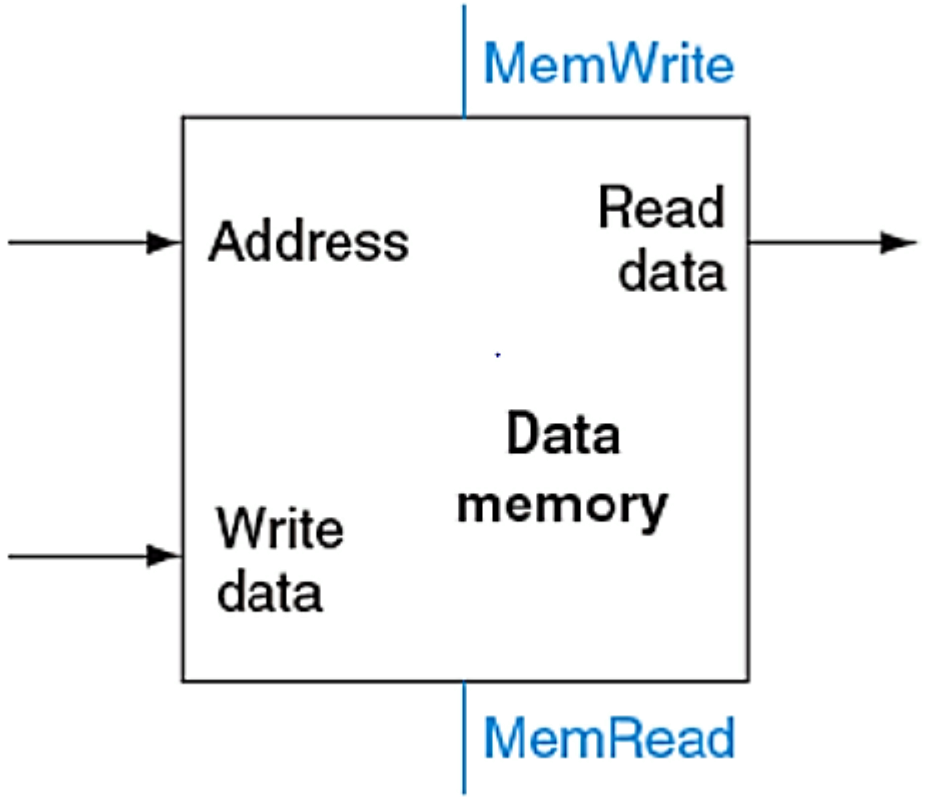


# ALU Unit

```
module  ALU32Bit(Zero, CarryOut, Result, A, B, Op);
  input [2:0] Op;
  input [31:0] A, B;
  output [31:0] Result;
  reg [31:0] Result;
  output CarryOut, Zero;
  reg CarryOut;
  assign Zero = (({Result} == 0)) ? 1 : 0;
  always @ (Op, A, B) begin
    case(Op)
      0:  Result <= A & B;
      1:  Result <= A | B;
      2:  {CarryOut, Result[31:0]} <= A + B;
      6:  {CarryOut, Result[31:0]} <= A - B;
      7:  Result <= A < B ? 1 : 0;
      default: Result <= 0;
    endcase
  end
endmodule
```

# Data Memory

- There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.
- The output is the data read from the memory location accessed, if applicable.



# Data Memory

- **Input**:- Read address, write address, write data
- **Output**: Read data

```
input MemRead, MemWrite, Clock;
input [31:0] ReadAddress, WriteAddress;
input [31:0] WriteData;
output reg [31:0] ReadData;
reg [31:0] memory [0:31];
integer raddr, waddr;
```

```
initial begin
  memory[0] = 32'b00000000000000000000000000000000;  // nop
memory[1] = 32'b00000000000000000000000001010100;  // Value of 84
memory[2] = 32'b00000000000000000000000000001011;  // Value of 11
```

# Data Memory

```
always @(posedge Clock) begin
    raddr = ReadAddress;
    waddr = WriteAddress;
    if(MemRead)
        ReadData = memory[raddr/4];
    else if(MemWrite)
        memory[waddr/4] = WriteData;
end
```

# Multiplexers

- Multiplexers 32bit-2x1 (4 modules)
  - Generating the new address after the branch MUX
  - Generating the new address after the Jump MUX
  - Writing back to register file
  - Selecting the second ALU Source
- Multiplexers 5bit-2x1 (1 module)
  - Selecting the writing register

# Multiplexers 5bit-2x1

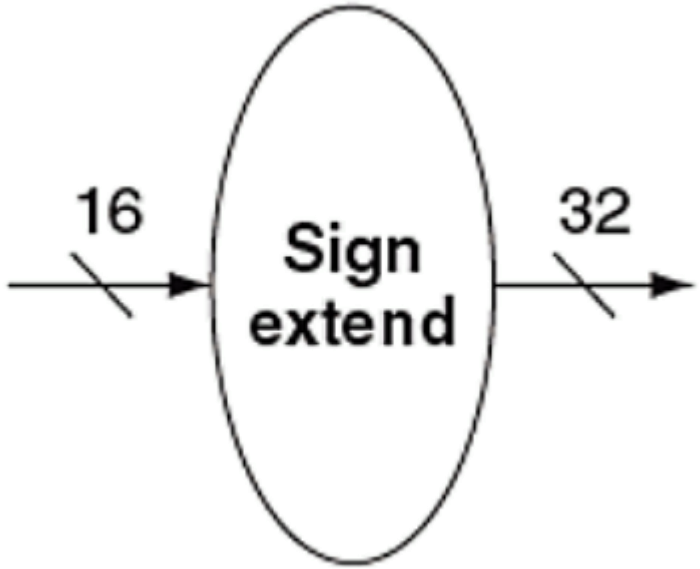
```
module MUX5Bit_2To1(out, select, q1, q2);
    input  [4:0] q1, q2;
    input select;
    output [4:0] out;
    genvar  j;
  generate  for(j = 0; j < 5; j = j + 1)
    begin:  mux_loop
      Mux2To1 Mux(out[j], select, q1[j], q2[j]);
    end
  endgenerate
endmodule
```

# Multiplexers 32bit-2x1

```
module  Mux32Bit_2To1(out, select, in1, in2);
  input [31:0] in1, in2;
  input select;
  output  [31:0] out;
  Mux8Bit_2To1_generate Mux1(out[7:0], select, in1[7:0], in2[7:0]);
  Mux8Bit_2To1_generate Mux2(out[15:8], select, in1[15:8], in2[15:8]);
  Mux8Bit_2To1_generate Mux3(out[23:16], select, in1[23:16], in2[23:16]);
  Mux8Bit_2To1_generate Mux4(out[31:24], select, in1[31:24], in2[31:24]);
endmodule
```

# Sign Extender

- To perform sign-extending, we can add a sign extension element.
  - The sign extension element takes as input a 16-bit wide value to be extended to 32-bits.
  - To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.



# Sign Extender

- Sign Extend the Branch Offset

```
module Sign_Extender(out, in);
    input [15:0] in;
    output [31:0] out;
    assign out = { {16{in[15]}}, in};
endmodule
```

# Shifter (2 Modules)

- Shift left by 2 bits the Sign Extended Branch Offset
- Shift left by 2 bits the Jump Offset.

```
module Shift_Left(out, in);
    input [31:0] in;
    output [31:0] out;
    assign out = {in[29:0],1'b0,1'b0};
endmodule
```

# Concatenate PC and Jump Offset

- Concatenate PC and Left Shifted Jump Offset

```
module concatJuPC(out, J, PC);
  input [31:0] J, PC;
  output [31:0] out;
  assign {out} = {{PC[31:28]}, {J[27:0]}};
endmodule
```

# ALU Control Unit

```
module  ALUControlUnit(Op, Func, ALUOp);
   input [5:0] Func;
   input [1:0] ALUOp;
   output  [2:0] Op;
   assign  Op[0] = ALUOp[1] & (Func[3] | Func[0]);
   assign  Op[1] = (~ALUOp[1]) | (~Func[2]);
   assign  Op[2] = ALUOp[0] | (ALUOp[1] & Func[1]);
endmodule
```

# Main Control Unit

- Generate the Control Signals

```
output  RegDst, Jump, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite,
        Branch, ALUOp0, ALUOp1;
input [5:0] Op;
wire  RFormat, LW, SW, BEQ, J;
assign  RFormat = (~Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(~Op[1])&(~Op[0]);
assign  LW = (Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(Op[1])&(Op[0]);
assign  SW = (Op[5])&(~Op[4])&(Op[3])&(~Op[2])&(Op[1])&(Op[0]);
assign  BEQ = (~Op[5])&(~Op[4])&(~Op[3])&(Op[2])&(~Op[1])&(~Op[0]);
assign  J = (~Op[5])&(~Op[4])&(~Op[3])&(~Op[2])&(Op[1])&(~Op[0]);
```

# Final Task

- Integrate all Modules.
- Test on given testbench.

# Thank You