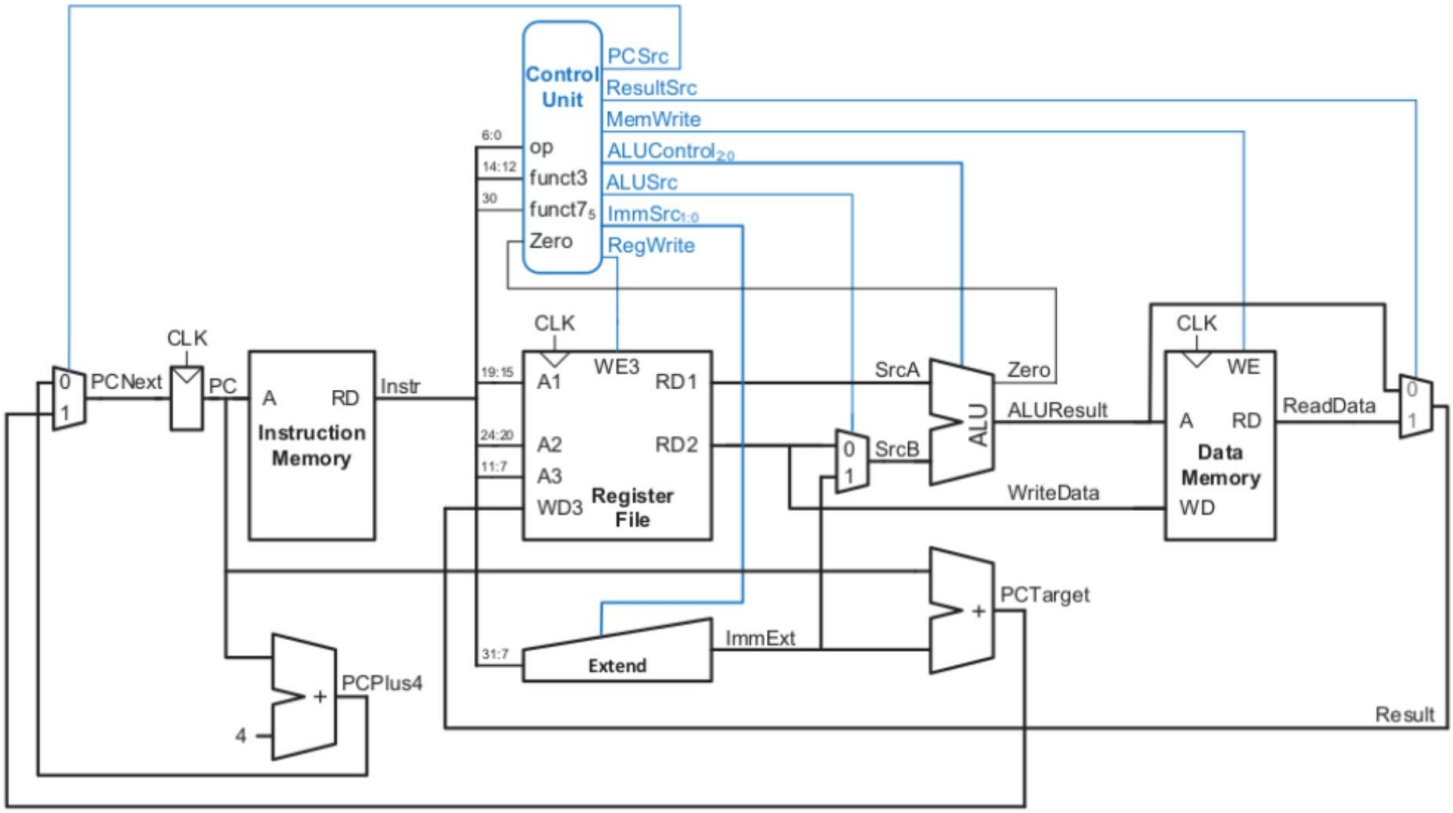
CA Innovative Assignment

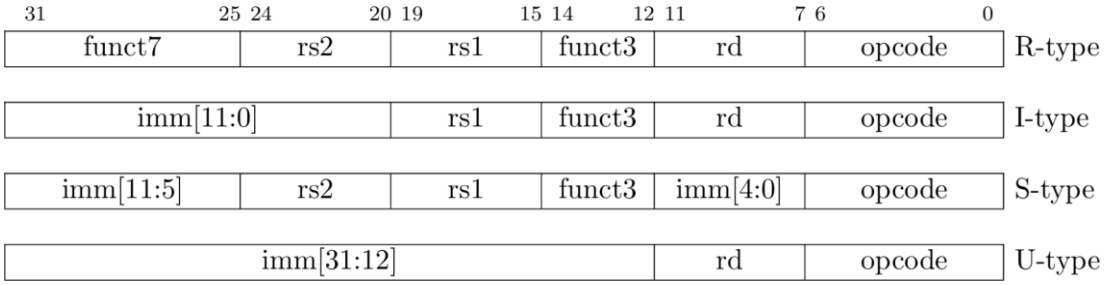
RISC-V based processor design in Verilog

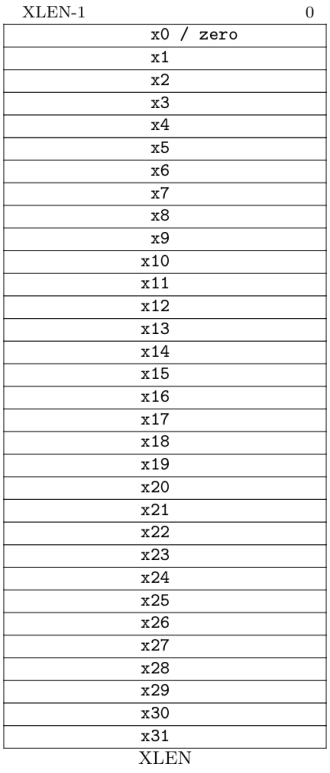
***CREDIT NOTE:** learnt, adapted from:
https://github.com/merldsu/RISCV_Single_Cycle_Core

addi x5,x0,0x5 #rs1 data. Moves it to x5 register. x0 reg is used for init
addi x6,x0,0x4 #rs2 data init to x6
or x7,x5,x6 #or operation on x5 and x6
and x8,x5,x6 #and operation



RISC-V





Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

ALU

```
module ALU(A,B,Result,ALUControl,OverFlow,Carry,Zero,Negative);

    input [31:0]A,B;
    input [2:0]ALUControl;
    output Carry,OverFlow,Zero,Negative;
    output [31:0]Result;

    wire Cout;
    wire [31:0]Sum;

    assign {Cout,Sum} = (ALUControl[0] == 1'b0) ? A + B :
                                          (A + ((~B)+1)) ;
    assign Result = (ALUControl == 3'b000) ? Sum :
                    (ALUControl == 3'b001) ? Sum :
                    (ALUControl == 3'b010) ? A & B :
                    (ALUControl == 3'b011) ? A | B :
                    (ALUControl == 3'b101) ? {{31{1'b0}},(Sum[31])} :
{32{1'b0}};

    assign OverFlow = ((Sum[31] ^ A[31]) &
                       (~(ALUControl[0] ^ B[31] ^ A[31])) &
                       (~ALUControl[1]));
    assign Carry = ((~ALUControl[1]) & Cout);
    assign Zero = &(~Result);
    assign Negative = Result[31];

endmodule
```

This is code for the Arithmetic Logic Unit (ALU) module. The ALU is responsible for performing arithmetic and logic operations on binary data. The ALU supports addition, subtraction, and logical operations like AND and OR. The operations are selected based on the ALUControl input.

We have implemented only the ADD, SUB, SLT, OR,AND R-type instructions

Module Ports and Signals

Inputs:
- **A, B**: These are the 32-bit input operands on which the ALU performs operations.
- **ALUControl**: A 3-bit signal used to determine the operation the ALU should perform.

Outputs:
- **Result:** A 32-bit output that holds the result of the ALU operation.
- **Carry**: A single-bit output indicating if there was a carry-out from the most significant bit in the last arithmetic operation.
- **OverFlow**: A single bit that indicates if an overflow occurred during the operation.
- **Zero**: A single bit that is set if the result of the operation is zero.
- **Negative**: A single bit indicating if the result is negative (i.e., the most significant bit is set).

Internal Wires:
- **Cout**: Indicates carry out from the addition or subtraction.
- **Sum**: A 32-bit internal signal used to hold intermediate sums or results of addition and subtraction.

Operation Logic

1. Addition and Subtraction:
   - The ALU performs addition or subtraction based on the lsb (least significant bit) of the ALUControl. If ALUControl[0] is 0, the operands A and B are added. If it is 1, B is negated and then added to A to perform subtraction.

2. Result Assignment:
   - The actual operation to execute is chosen based on the value of ALUControl:
     - 3'b000 and 3'b001: Perform addition/subtraction and set the Result to Sum.
     - 3'b010: Perform bitwise AND operation between A and B.
     - 3'b011: Perform bitwise OR operation.
     - 3'b101: Extracts and sign-extends the most significant bit of Sum to form the Result. This is used to check the result's sign in operations affecting the sign bit.
     - All other values of ALUControl default to a zero Result.

3. Flags/Status Bits:

- Carry: Set if there is a carry out from the most significant bit (Cout) during addition/subtraction and if the ALUControl does not direct a bitwise operation (~ALUControl[1]).
- Overflow: Detected by XOR-ing the sign bit of Sum and A, and checking conditions with B and ALUControl bits. It indicates an overflow condition for addition or subtraction.
- Zero: Set if all bits of Result are zero.
- Sign: the sign of the Result,

Execution of LW:
Instr address fetched from Instruction memory, then it goes to A1 of reg file, instn is returned from rd1 port. Then offset is added to the instn in the main alu, using extend hardware. Thats why alucontrol is 000 for LW. And then The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle, in port 3 of the register file.
read the source register containing the base address. This register is specified in the rs1 field of the instruction,

ALU decoder

```
module ALU_Decoder(ALUOp,funct3,funct7,op,ALUControl);

    input [1:0]ALUOp;
    input [2:0]funct3;
    input [6:0]funct7,op;
    output [2:0]ALUControl;

    assign ALUControl = (ALUOp == 2'b00) ? 3'b000 :
                        (ALUOp == 2'b01) ? 3'b001 :
                        ((ALUOp == 2'b10) & (funct3 == 3'b000) &
({op[5],funct7[5]} == 2'b11)) ? 3'b001 :
                        ((ALUOp == 2'b10) & (funct3 == 3'b000) &
({op[5],funct7[5]} != 2'b11)) ? 3'b000 :
                        ((ALUOp == 2'b10) & (funct3 == 3'b010)) ? 3'b101 :
                        ((ALUOp == 2'b10) & (funct3 == 3'b110)) ? 3'b011 :
                        ((ALUOp == 2'b10) & (funct3 == 3'b111)) ? 3'b010 :
                                                                  3'b000 ;
endmodule
```

The ALU decoder determines the operation that the ALU should perform.

The ALU Decoder accepts the following inputs:
- ALUOp: A 2-bit control signal that categorizes the type of operation
- funct3 and funct7: Bits extracted from the instruction that, along with ALUOp, refine what operation should be performed.
- op: a portion of the opcode, helps in further decoding especially when operations share funct3 and funct7 patterns.

The output of the decoder is:
- ALUControl: A 3-bit signal that directly controls the operation of the ALU.

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Decoding Logic

The ALU Decoder translates the input signals into a specific control signal using a series of conditional checks:

- When ALUOp is 00, the decoder sets ALUControl to 000, representing addition.
- When ALUOp is 01, it sets ALUControl to 001, used for subtraction.

ALUOp = 10:

- If funct3 is 000, it further checks the combination of op[5] and funct7[5]. If this combination is 11, it sets ALUControl to 001, which means subtraction, otherwise, it sets to 000 for addition.
- For funct3 set to 010, 110, or 111, the control signals are set to 101, 011, and 010 respectively, directing the ALU to perform set-less-than(slt), XOR, and AND operations. Op[5] and funct7[5] are set to don't care.

Main decoder

```
module
Main_Decoder(Op,RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,Branch,ALUOp);
    input [6:0]Op;
    output RegWrite,ALUSrc,MemWrite,ResultSrc,Branch;
    output [1:0]ImmSrc,ALUOp;

    assign RegWrite = (Op == 7'b0000011 | Op == 7'b0110011) ? 1'b1 :
                                                              1'b0 ;
    assign ImmSrc = (Op == 7'b0100011) ? 2'b01 :
                    (Op == 7'b1100011) ? 2'b10 :
                                         2'b00 ;
    assign ALUSrc = (Op == 7'b0000011 | Op == 7'b0100011) ? 1'b1 :
                                                            1'b0 ;
    assign MemWrite = (Op == 7'b0100011) ? 1'b1 :
                                           1'b0 ;
    assign ResultSrc = (Op == 7'b0000011) ? 1'b1 :
                                            1'b0 ;
    assign Branch = (Op == 7'b1100011) ? 1'b1 :
                                         1'b0 ;
    assign ALUOp = (Op == 7'b0110011) ? 2'b10 :
                   (Op == 7'b1100011) ? 2'b01 :
                                        2'b00 ;

endmodule
```

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01

Inputs and Outputs:

Input:
- Op: A 7-bit input that determines the type of operation based on the instruction's opcode.

Outputs:
- RegWrite: if a register should be written. Set for load (0000011) and arithmetic/logical instructions (0110011).
- ImmSrc: Determines the type of immediate data sourcing. Set to 01 for store instructions (0100011), 10 for branch instructions (1100011), and 00 for others.
- ALUSrc: Specifies whether the ALU should use an immediate value or a register value. Set for load (0000011) and store (0100011) instructions.
- MemWrite: Controls memory write operations, active for store instructions (0100011).
- ResultSrc: Indicates the source of the result for register write-back, set for load instructions (0000011) indicating the result comes from memory.
- Branch: Signals whether a branch operation should be taken, active for branch instructions (1100011).
- ALUOp: Provides specific operation codes to the ALU. Set to 10 for arithmetic/logical instructions (0110011) and 01 for branch instructions (1100011).

Decoding Logic Explained:
- The decoder evaluates the Op input to generate specific control signals that direct other components of the CPU on how to execute the current instruction.
- RegWrite is activated for operations that involve writing to a register, such as load and arithmetic/logical operations.
- ImmSrc specifies how immediate values are utilized, differentiating among addressing modes for store operations, and providing distinctions for branch decisions.
- ALUSrc is set high when an immediate value is needed directly for the operation, distinctively for load and store operations.
- MemWrite is enabled specifically for store operations to permit data storage into memory.

- ResultSrc determines if the result written back to the register comes from the ALU or memory, mainly activated for load operations.
- Branch is used to manage the execution flow change, particularly for conditional branches.
- ALUOp assigns more detailed codes for the ALU, enabling it to understand whether it should perform complex calculations or simpler comparison tasks.

Instruction Memory

```
module Instruction_Memory(rst,A,RD);
  input rst;
  input [31:0]A;
  output [31:0]RD;

  reg [31:0] mem [1023:0];

  assign RD = (~rst) ? {32{1'b0}} : mem[A[31:2]];

  initial begin
    $readmemh("memfile.hex",mem,0,1023);
  end

endmodule
```

Instruction_Memory, is responsible for storing and providing access to the instructions that the CPU will execute.

Inputs:
- rst: A reset input that typically clears or initializes the system state when active.

- A: A 32-bit input address which specifies the memory location to be accessed. This address directly determines which instruction is read from memory.

Output:
- RD: A 32-bit output which holds the instruction read from the specified address in memory.

Memory Array:
- mem: This is a register array with 1024 slots (indexed from 0 to 1023), each capable of holding a 32-bit instruction. This represents the actual storage area for the instructions that the CPU will execute.

Operation Logic

Memory Access:
- The module maps the input address A to the corresponding instruction in the mem array. To align the 32-bit address to the word boundary required for 32-bit instructions (since RISC-V is a word-addressable architecture), only the higher bits of A (A[31:2]) are used. This effectively ignores the lower 2 bits, which aligns the address to 4-byte boundaries (since 2^2 = 4 bytes per word).

Reset Behavior:
- The output RD is conditional based on the rst signal. If rst is not low, RD outputs zero (represented by {32{1'b0}}, a 32-bit wide bus of zeros), effectively disabling output when the system is in a reset state. If rst is high, the memory content at the specified address is outputted through RD.

Memory Initialization:
- The initial block with $readmemh("memfile.hex", mem) initializes the mem array with values from a file named memfile.hex. This file contains the hexadecimal representation of the instructions and is loaded into the memory at simulation start-up, preparing the instruction set that the CPU will process.

Data memory

```
module Data_Memory(clk,rst,WE,WD,A,RD);


    input clk,rst,WE;
    input [31:0]A,WD;
    output [31:0]RD;


    reg [31:0] mem [1023:0];


    always @ (posedge clk)
    begin
        if(WE)
            mem[A] <= WD;
    end


    assign RD = (~rst) ? 32'd0 : mem[A];


    initial begin
        mem[28] = 32'h00000020;
    end


endmodule
```

Key Components:

The inputs are:

- clk: Clock signal, used to synchronize the operations within the module.
- rst: Reset signal, used to initialize or reset the memory.
- WE: Write Enable signal, determines whether data should be written to the memory.
- A: Address input, specifies the memory location to be accessed.
- WD: Write Data input, contains the data to be written to the memory.

The output is:

- RD: Read Data output, contains the data read from the memory.

2. Memory Array: The module contains a 1024-word (32-bit each) memory array named mem. This array is indexed from 0 to 1023, allowing for a total of 4096 bytes of memory.
3. Write Operation: The write operation is performed on the positive edge of the clock signal (posedge clk). If the Write Enable (WE) signal is high, the data from WD is written to the memory location specified by A.
4. Read Operation: The read operation is performed by assigning the value from the memory location specified by A to RD. However, if the reset (rst) signal is high, RD is set to 0. This ensures that the memory can be reset to a known state.
5. Initialization: The initial block is used to initialize the memory. In this example, the memory location at index 28 is set to 32'h00000020. This could be a specific address used by the CPU for a particular purpose, such as storing the stack pointer or a program counter.

Control Unit

```
`include "ALU_Decoder.v"
`include "Main_Decoder.v"

module
Control_Unit_Top(Op,RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,Branch,funct
3,funct7,ALUControl);

    input [6:0]Op,funct7;
    input [2:0]funct3;
    output RegWrite,ALUSrc,MemWrite,ResultSrc,Branch;
    output [1:0]ImmSrc;
    output [2:0]ALUControl;

    wire [1:0]ALUOp;

    Main_Decoder Main_Decoder(
                .Op(Op),
                .RegWrite(RegWrite),
                .ImmSrc(ImmSrc),
                .MemWrite(MemWrite),
                .ResultSrc(ResultSrc),
                .Branch(Branch),
                .ALUSrc(ALUSrc),
                .ALUOp(ALUOp)
    );

    ALU_Decoder ALU_Decoder(
                            .ALUOp(ALUOp),
                            .funct3(funct3),
                            .funct7(funct7),
                            .op(Op),
```

```
                                .ALUControl(ALUControl)
    );
endmodule
```

Control_Unit_Top module is the modelling of the control unit

Key Components and Structure:

Includes:
- include "ALU_Decoder.v"
- include "Main_Decoder.v"
These include statements that incorporate the ALU Decoder and Main Decoder modules into the current design, enabling the Control Unit to utilize the functionality defined in these separate files.

Inputs:
- Op: A 7-bit input representing the opcode part of an instruction, specifying the type of operation.
- funct3: A 3-bit function code part of the instruction, providing finer control and differentiation between operations that share opcodes or other parameters.
- funct7: Another part of the instruction used for detailed operation decoding, often in conjunction with funct3.

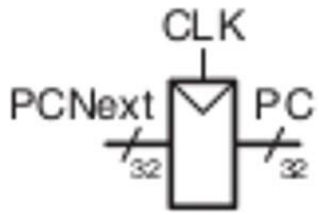
Outputs:
- RegWrite, ALUSrc, MemWrite, ResultSrc, Branch: Control signals generated by the Main Decoder to manage different aspects of the CPU's operation.
- ImmSrc: A 2-bit output from the Main Decoder that specifies the type of immediate data sourcing required by the instruction.
- ALUControl: A 3-bit signal from the ALU Decoder that tells the ALU exactly what operation to perform.

Internal Wire:
- ALUOp: A 2-bit signal used internally between the Main Decoder and the ALU Decoder to further specify ALU operations.

Operation:
- The Control_Unit_Top module serves as the central decision-making hub for the CPU, interpreting instruction codes and directing activities across various CPU components. It ensures that instructions are correctly understood and that appropriate actions are executed in the ALU, register file, and memory based on the decoded outputs.

PC

```
module PC_Module(clk,rst,PC,PC_Next);
    input clk,rst;
    input [31:0]PC_Next;
    output [31:0]PC;
    reg [31:0]PC;

    //For every positive edge of the clock pulse
    always @(posedge clk)
    begin
        if(~rst)
            PC <= {32{1'b0}};
        else
            PC <= PC_Next;
    end
endmodule
```



The PC_Module represents the Program Counter (PC) in a processor's design. It's responsible for maintaining the address of the current instruction that the CPU is executing and updating it to the next instruction's address.

Inputs:
- clk: This is the clock signal that helps synchronize the updates to the PC with the rest of the CPU operations.
- rst: This is the reset signal. When active, it sets the PC to zero, effectively resetting the program counter.

- PC_Next: This 32-bit input specifies what the next value of the PC should be, usually the address of the next instruction.

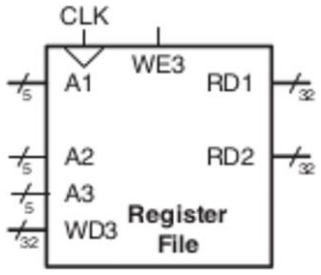
Output:
- PC: This 32-bit output holds the current value of the program counter.

Functionality:
- The PC value is updated with each positive edge of the clock signal.
- If the reset signal is active, the PC is set to zero.
- If the reset signal is not active, the PC updates to the value given by PC_Next.

Operation:
- The module updates the program counter each time the clock ticks, ensuring that the processor fetches and executes the correct instruction sequentially.

Register File

```
module Register_File(clk,rst,WE3,WD3,A1,A2,A3,RD1,RD2);

    input clk,rst,WE3;
    input [4:0]A1,A2,A3;
    input [31:0]WD3;
    output [31:0]RD1,RD2;

    reg [31:0] Register [31:0];

    always @ (posedge clk)
    begin
        if(WE3)
            Register[A3] <= WD3;
    end

    assign RD1 = (~rst) ? 32'd0 : Register[A1];
    assign RD2 = (~rst) ? 32'd0 : Register[A2];

    initial begin
        Register[5] = 32'h00000005;
        Register[6] = 32'h00000004;

    end

endmodule
```



Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

Inputs:

- clk: clock input signal for the register file module.
- rst: reset input signal, which is used to initialize the register file.
- WE3: write enable signal for register A3. When WE3 is 1, the data on WD3 will be written to the register specified by the address A3.
- A1, A2, A3: These are 5-bit input signals that specify the addresses of the registers to be read from (A1 and A2) or written to (A3).
- WD3: This is a 32-bit input signal that carries the data to be written to the register specified by A3, when WE3 is high.

Outputs:

- RD1: This is a 32-bit output signal that carries the data read from the register specified by the address A1.
- RD2: This is a 32-bit output signal that carries the data read from the register specified by the address A2.

Wires and Functionality:

- Register [31:0]: This is a 32-entry register array, where each entry is 32 bits wide. It represents the actual register file storage.
- always @ (posedge clk): This is a procedural block that is triggered on the positive edge of the clock signal clk. It performs the write operation to the register file.
    - if(WE3): If the write enable signal WE3 is high, the data on WD3 is written to the register.
    - The line Register[A3] <= WD3; performs this write operation.
- assign RD1 = (~rst) ? 32'd0 : Register[A1];: This continuous assignment statement handles the read operation for RD1. If the reset signal rst is low (active low), RD1 is set to zero. Otherwise, RD1 is assigned the value of the register specified by the address A1.
- assign RD2 = (~rst) ? 32'd0 : Register[A2];: same as for RD1
- initial begin ... end: sets the initial values of registers 5 and 6 to 0x00000005 and 0x00000004, respectively. This block is executed only once during the simulation or synthesis process.

Main code

```
`include "PC.v"
`include "Instruction_Memory.v"
`include "Register_File.v"
`include "Sign_Extend.v"
`include "ALU.v"
`include "Control_Unit_Top.v"
`include "Data_Memory.v"
`include "PC_Adder.v"
`include "Mux.v"

module Single_Cycle_Top(clk,rst);

    input clk,rst;

    wire [31:0]
PC_Top,RD_Instr,RD1_Top,Imm_Ext_Top,ALUResult,ReadData,PCPlus4,RD2_Top,Src
B,Result;
    wire RegWrite,MemWrite,ALUSrc,ResultSrc;
    wire [1:0]ImmSrc;
    wire [2:0]ALUControl_Top;

    PC_Module PC(
        .clk(clk),
        .rst(rst),
        .PC(PC_Top),
        .PC_Next(PCPlus4)
    );

    PC_Adder PC_Adder(
                    .a(PC_Top),
                    .b(32'd4),
                    .c(PCPlus4)
```

```
    );

    Instruction_Memory Instruction_Memory(
                            .rst(rst),
                            .A(PC_Top),
                            .RD(RD_Instr)
    );

    Register_File Register_File(
                            .clk(clk),
                            .rst(rst),
                            .WE3(RegWrite),
                            .WD3(Result),
                            .A1(RD_Instr[19:15]),
                            .A2(RD_Instr[24:20]),
                            .A3(RD_Instr[11:7]),
                            .RD1(RD1_Top),
                            .RD2(RD2_Top)
    );

    Sign_Extend Sign_Extend(
                        .In(RD_Instr),
                        .ImmSrc(ImmSrc[0]),
                        .Imm_Ext(Imm_Ext_Top)
    );

    Mux Mux_Register_to_ALU(
                            .a(RD2_Top),
                            .b(Imm_Ext_Top),
                            .s(ALUSrc),
                            .c(SrcB)
    );
```

```
ALU ALU(
        .A(RD1_Top),
        .B(SrcB),
        .Result(ALUResult),
        .ALUControl(ALUControl_Top),
        .OverFlow(),
        .Carry(),
        .Zero(),
        .Negative()
);

Control_Unit_Top Control_Unit_Top(
                        .Op(RD_Instr[6:0]),
                        .RegWrite(RegWrite),
                        .ImmSrc(ImmSrc),
                        .ALUSrc(ALUSrc),
                        .MemWrite(MemWrite),
                        .ResultSrc(ResultSrc),
                        .Branch(),
                        .funct3(RD_Instr[14:12]),
                        .funct7(RD_Instr[6:0]),
                        .ALUControl(ALUControl_Top)
);

Data_Memory Data_Memory(
                    .clk(clk),
                    .rst(rst),
                    .WE(MemWrite),
                    .WD(RD2_Top),
                    .A(ALUResult),
                    .RD(ReadData)
);
```

```
    Mux Mux_DataMemory_to_Register(
                                .a(ALUResult),
                                .b(ReadData),
                                .s(ResultSrc),
                                .c(Result)
    );

endmodule
```

This module integrates various components essential for CPU operations, simulating the complete data path of a single-cycle CPU.

Included Modules:

- It includes external modules such as PC.v (Program Counter), Instruction_Memory.v, Register_File.v, Sign_Extend.v, ALU.v, Control_Unit_Top.v, Data_Memory.v, PC_Adder.v, and Mux.v. Each of these modules performs specific tasks in the CPU's data path, such as fetching instructions, executing them, handling data storage, and controlling the flow of execution.

Inputs:

- clk (clock): Drives the timing of operations in the processor, synchronizing actions.
- rst (reset): Used to initialize or reset the processor, setting it to a known state.

Internal Wires:

- The module defines multiple 32-bit internal wires such as PC_Top, RD_Instr (Read Data from Instruction Memory), RD1_Top, Imm_Ext_Top (Immediate Extended), ALUResult, ReadData, PCPlus4, RD2_Top, SrcB, and Result. These are used to connect different modules within the processor and carry data throughout the cycle.
- PC_Top wire connects the PC module to the instruction memory module.

- Control signals like RegWrite, MemWrite, ALUSrc, ResultSrc, along with a 2-bit ImmSrc and a 3-bit ALUControl_Top, are used to control the behavior of various modules based on the decoded instruction.

Module Connections and Functions:

- PC_Module: Manages the program counter that holds the address of the current instruction.

- PC_Adder: Responsible for incrementing the program counter by 4 to point to the next instruction.
- Instruction_Memory: Fetches the instruction from memory based on the current program counter.
- Register_File: Manages CPU registers, reading operands for the ALU and writing back results.
- Sign_Extend: Extends the immediate values from the instruction to full word size for operations that require them.
- Mux_Register_to_ALU and Mux_DataMemory_to_Register: These multiplexers control data paths based on control signals, directing either register values or immediate values to the ALU and choosing between ALU results or memory data for register write-back.
- ALU: Performs arithmetic and logical operations based on inputs from the register file and control signals defining the operation.
- Control_Unit_Top: Decodes the opcode, funct3, and funct7 fields from the instruction to generate control signals that dictate the operation of other modules.
- Data_Memory: Handles data storage and retrieval operations during load and store instructions.

Operation:

- The Single_Cycle_Top module simulates the full operation of a single-cycle processor. On each clock cycle, an instruction is fetched, decoded, and executed, with results written back as necessary, all within a single cycle. This setup demonstrates the interaction between different components of a CPU and how they contribute to the execution of instructions.

PC_Module PC(….) means that we are calling/imporitng the PC_Module from PC.v file, and giving it the name PC.

always block is used to model sequential or combinational circuits on specific behaviour. In sequential circuits, the always block executes at a clock edge specified. In combinational circuit, it executes when specified elements change. always(*) generates dependency list, i.e, it will execute when either of the dependent variables change.

The register file is 32x32, as it is specified by the RISC-V ISA