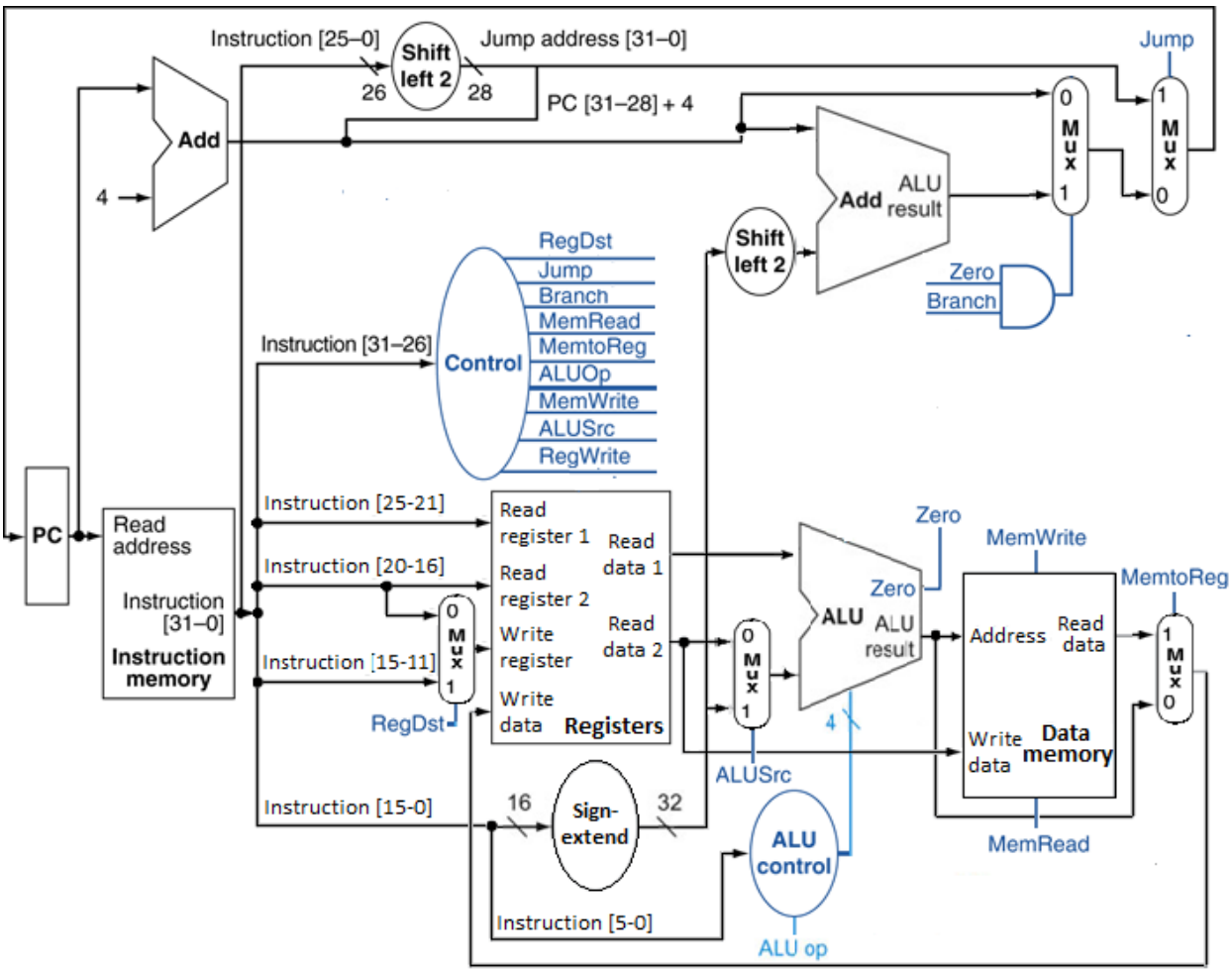
CECS 440
MIPS SINGLE CYLE PROCESSOR

OBJECTIVES:

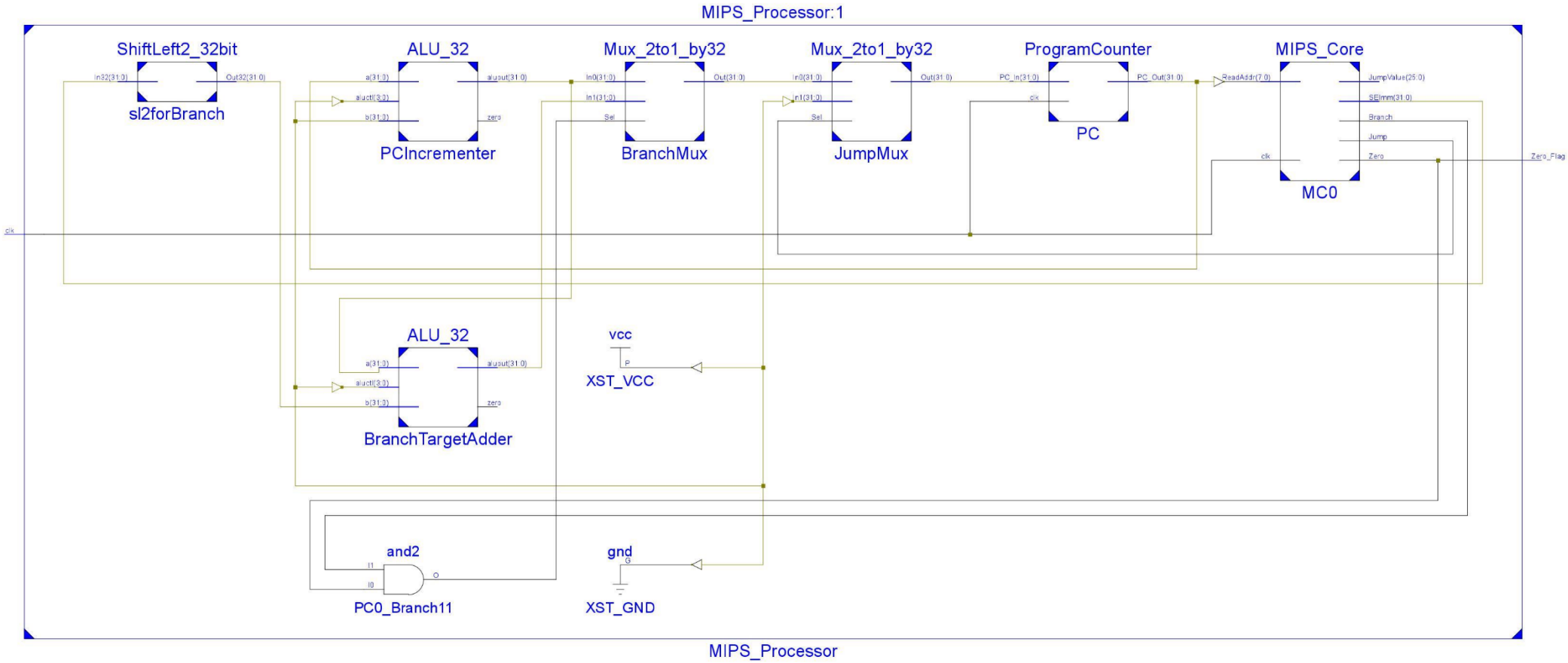
- Complete the MIPS Single Cycle Processor
- Test the MIPS Single Cycle Processor for correct operation by running a program.

All the components shown in the image below will be modeled and interconnected so a program can be loaded and run:



Another depiction of the processor produced by Xilinx is shown on the next page:

CECS 440
MIPS SINGLE CYLE PROCESSOR



CECS 440
MIPS SINGLE CYLE PROCESSOR

ACTIVITY 1

OBJECTIVE: Make he MIPS Single Cycle Processor Verilog module (MIPS_Processor.v)

Before proceeding, a couple of remaining components must be created.

1. Create the Program Counter:

```
`timescale 1ns / 1ps

module ProgramCounter(clk, PC_In, PC_Out);
   input                    clk;
   input          [31:0]    PC_In;
   output   reg   [31:0]    PC_Out;

   always   @(posedge clk)
      PC_Out   =  PC_In;

endmodule
```

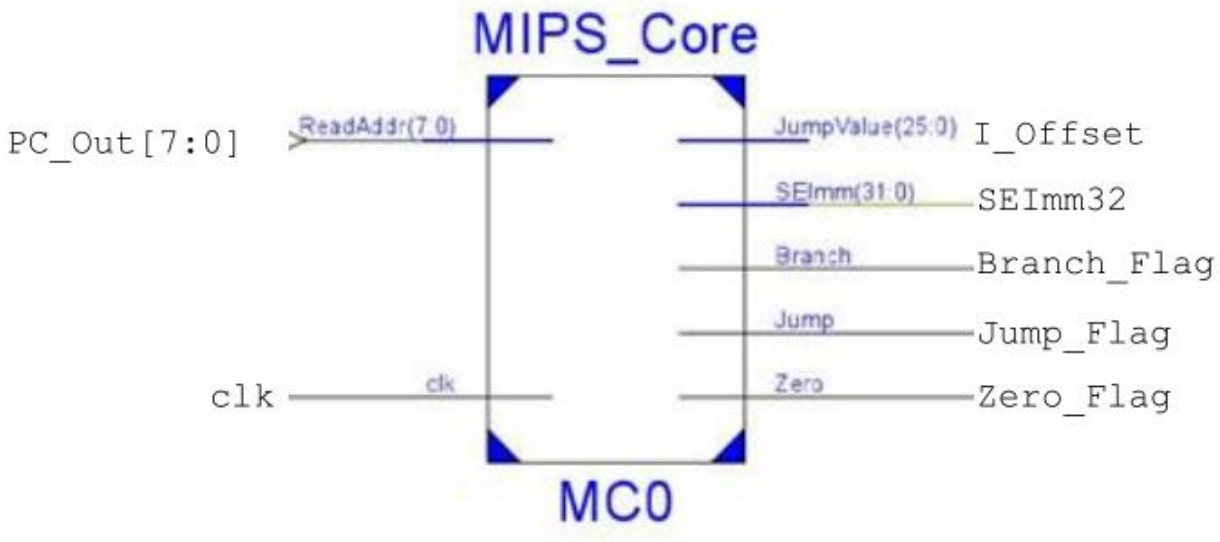
2. Create a left shifter which will receive a 32-bit input and output a the 2 positional logically left shifted version of the input. This module will be used for calculating Jump target address and the branch target address.

3. A port list definition with internal wire/bus declarations is given, the provided verilog test fixture will refer to port names and bus names specified below:

```
module MIPS_Processor(  clk, Zero_Flag    );
    input       clk;
    output  Zero_Flag;

    wire    [31:0]  PC_Out,
                    PC_Out_plus4,
                    PC_Next,
                    SEImm32,
                    SL2SEImm32,
                    JumpAddress,
                    BranchTarget,
                    BranchMuxOut;
    wire    [25:0]  I_Offset;
    wire            Zero_Flag,
                    Branch_Flag,
                    Jump_Flag,
                    PC0_Branch1;
```

4. Recommendations on how to create other modules is given as follows
    a. An instance of the ALU can serve as the PC incrementer
        i. One of the PC incrementer data inputs should be the output of the Program Counter
        ii. The other PC incrementer data input shold be hard coded to the value 4
        iii. The ALU Control input will be hard coded with the value to perform addition
    b. An instance of the ALU can also serve as the Branch Target Adder.
        i. Use a similar strategy with respect to the PC incrementer description
5. MIPS core can be instantiated and interconnected as follows:



6. And then include other multiplexers where appropriate

ACTIVITY 2

Creata a verilog test fixture named MIPS_Processor_Tester to test your MIPS Single Cycle Processor for correct operation. A program will be loaded into IMEM and instructions will execute once per clock cycle.

For basc testing, the program will be used:
Given initial register contents in decimal: $4 = 4; $12 = 12;
DMEM[0] = 3; DMEM[4] = 5; DMEM[8] = 11; DMEM[12] = 23
All other registers and DMEM address contain the value 0

MIPS ASSEMBLY	PSEUDOCODE

`.data`
`arr1: .word 3, 5, 11, 23`
`arr2: .word 0, 0, 0`
 	
`.text`
`testloop:`
`beq $5, $12, endloop`
`lw $10, 0($5)`
`lw $11, 4($5)`
`sub $9, $11, $10`
`sw $9, 16($5)`
`add $5, $5, $4`
`j testloop`
`endloop:`
`add $0, $0, $0`
	int arr1[] = {3, 5, 11, 23};
	int arr2[] = {0, 0, 0};
	$4 = 4;
	$12 = 12;
	$5 = 0;
	while($5 != $12){
	arr2[$5] = arr1[$5+1] - arr1[$5];
	$5++;
	}
	Nop;

Initial DMEM and Register values are loaded from .dat files. The program machine code is loaded from a .dat file as well. The .dat files are attached to the beachboard dropbox. The Verilog test fixture source code below provides all the functionality needed to initialize all memory elements and show instructio execution details in the console.

CECS 440
MIPS SINGLE CYLE PROCESSOR

```
`timescale 1ns / 1ps
module MIPS_Processor_Tester;
    reg clk; wire Zero_Flag;
    MIPS_Processor uut (.clk(clk), .Zero_Flag(Zero_Flag) );
    integer i;
    parameter   RTYPE   = 6'b000000,    LW      = 6'b100011,
                SW      = 6'b101011,    BEQ     = 6'b000100,
                J       = 6'b000010,    ADD     = 6'b100000,
                SUB     = 6'b100010,    AND     = 6'b100100,
                OR      = 6'b100101,    SLT     = 6'b101010;
    wire [4:0] rd, rs, rt; wire [15:0] imm; wire [25:0] addr;
    assign {rd, rs, rt} = {uut.MC0.I[15:11], uut.MC0.I[25:21], uut.MC0.I[20:16]},
           imm = uut.MC0.I[15:0], addr = uut.MC0.I[25:0];
    always #5 clk = ~clk;   //clock pulse generation
    always  @(posedge clk) begin
            $timeformat(    -9, 1,  " ns", 8);
            $display("At time %t", $time);
            #1;
            case(   uut.MC0.I[31:26]    )
                RTYPE:
                    case( uut.MC0.I[5:0] )
                        ADD: $display("add $%0d, $%0d, $%0d", rd, rs, rt);
                        SUB: $display("sub $%0d, $%0d, $%0d", rd, rs, rt);
                        AND: $display("and $%0d, $%0d, $%0d", rd, rs, rt);
                        OR: $display("or $%0d, $%0d, $%0d", rd, rs, rt);
                        SLT: $display("slt $%0d, $%0d, $%0d", rd, rs, rt);
                        default begin $display("ERROR: invalid R-Type Instruction");
                                      i = 1; end
                    endcase
                LW:     $display("lw $%0d, 0x%h( $%0d )", rt, imm, rs);
                SW:     $display("sw $%0d, 0x%h( $%0d )", rt, imm, rs);
                BEQ:    $display("beq $%0d, $%0d, 0x%h", rs, rt, imm);
                J:      $display("j 0x%h", {addr, 2'b00});
                default: begin  $display("ERROR: Invalid Instruction");
                                i = 1; end
            endcase
            $display("");
    end

    /*************************************************************************************/
    initial begin
        $display("SIMULATION BEGINS!");
        uut.PC.PC_Out = 0;
        i = 0;
        $readmemh("DMEM.dat",uut.MC0.DMEM.DM);
        $readmemh("IMEM.dat",uut.MC0.IMEM.IM);
        $readmemh("REGFILE.dat",uut.MC0.eu1.RF32.RF);
        clk = 0;
        while(!i)   #1 i = i;   // do nothing while i == 0
        $display("DUMPING REGISTERS");
        for(i = 0; i < 32; i = i + 1)
            $display("$%0d = 0x%h",i, uut.MC0.eu1.RF32.RF[i]);
        $display("\nDUMPING DATA MEMORY");
        for(i = 0; i < 28; i = i + 1)
            $display("DMEM[0x%h] = 0x%h", i, uut.MC0.DMEM.DM[i]);
        $display("SIMULATION ENDS!");
        $finish;
    end

endmodule
```

When simulation is run the following results should be produced:

```
At time   5.0 ns
beq $5, $12, 0x0006

At time  15.0 ns
lw $10, 0x0000( $5 )

At time  25.0 ns
lw $11, 0x0004( $5 )

At time  35.0 ns
sub $9, $11, $10

At time  45.0 ns
sw $9, 0x0010( $5 )

At time  55.0 ns
add $5, $5, $4

At time  65.0 ns
j 0x0000000

At time  75.0 ns
add $0, $0, $0

At time  85.0 ns
beq $5, $12, 0x0006

At time  95.0 ns
lw $10, 0x0000( $5 )

At time 105.0 ns
lw $11, 0x0004( $5 )

At time 115.0 ns
sub $9, $11, $10

At time 125.0 ns
sw $9, 0x0010( $5 )

At time 135.0 ns
add $5, $5, $4

At time 145.0 ns
j 0x0000000

At time 155.0 ns
add $0, $0, $0

At time 165.0 ns
beq $5, $12, 0x0006

At time 175.0 ns
lw $10, 0x0000( $5 )

At time 185.0 ns
lw $11, 0x0004( $5 )

At time 195.0 ns
sub $9, $11, $10

At time 205.0 ns
sw $9, 0x0010( $5 )

At time 215.0 ns
add $5, $5, $4

At time 225.0 ns
j 0x0000000

At time 235.0 ns
add $0, $0, $0

At time 245.0 ns
beq $5, $12, 0x0006

At time 255.0 ns
add $0, $0, $0

At time 265.0 ns
ERROR: Invalid Instruction
```

```
DUMPING REGISTERS
$0 = 0x00000000
$1 = 0x00000000
$2 = 0x00000000
$3 = 0x00000000
$4 = 0x00000004
$5 = 0x0000000c
$6 = 0x00000000
$7 = 0x00000000
$8 = 0x00000000
$9 = 0x0000000c
$10 = 0x0000000b
$11 = 0x00000017
$12 = 0x0000000c
$13 = 0x00000000
$14 = 0x00000000
$15 = 0x00000000
$16 = 0x00000000
$17 = 0x00000000
$18 = 0x00000000
$19 = 0x00000000
$20 = 0x00000000
$21 = 0x00000000
$22 = 0x00000000
$23 = 0x00000000
$24 = 0x00000000
$25 = 0x00000000
$26 = 0x00000000
$27 = 0x00000000
$28 = 0x00000000
$29 = 0x00000000
$30 = 0x00000000
$31 = 0x00000000
```

```
DUMPING DATA MEMORY
DMEM[0x00000000] = 0x00
DMEM[0x00000001] = 0x00
DMEM[0x00000002] = 0x00
DMEM[0x00000003] = 0x03
DMEM[0x00000004] = 0x00
DMEM[0x00000005] = 0x00
DMEM[0x00000006] = 0x00
DMEM[0x00000007] = 0x05
DMEM[0x00000008] = 0x00
DMEM[0x00000009] = 0x00
DMEM[0x0000000a] = 0x00
DMEM[0x0000000b] = 0x0b
DMEM[0x0000000c] = 0x00
DMEM[0x0000000d] = 0x00
DMEM[0x0000000e] = 0x00
DMEM[0x0000000f] = 0x17
DMEM[0x00000010] = 0x00
DMEM[0x00000011] = 0x00
DMEM[0x00000012] = 0x00
DMEM[0x00000013] = 0x02
DMEM[0x00000014] = 0x00
DMEM[0x00000015] = 0x00
DMEM[0x00000016] = 0x00
DMEM[0x00000017] = 0x06
DMEM[0x00000018] = 0x00
DMEM[0x00000019] = 0x00
DMEM[0x0000001a] = 0x00
DMEM[0x0000001b] = 0x0c
SIMULATION ENDS!
```

The contents of DMEM[0x13], DMEM[0x17], and DMEM[0x1B] are of particular importance since these are the memory locations overwritten by the program that is run.

While this test program is not an exhaustive test of the MIPS instruction set, it does serve to validate proper operation of most instructions.

Deliverables:

Submit a single report as a PDF which contains the following:

1. Fully commented **MIPS_Processor.v** from Activity 1
2. Fully commented Verilog Test Fixture source code from Activity 2
3. Console output text showing the execution of each instruction followed by the register dump and data memory dump