Source File

Design Sources:

1. Program Counter

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/04/2019 07:03:35 PM
// Design Name:
// Module Name: ProgramCounter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module ProgramCounter(input [31:0]Address,
                input Clk,
                input [26:0]Instruction,
                input Reset,
                input [31:0]startPC,
                output [31:0] PC_incby4,
                output [31:0] jumpaddress,
                output reg [31:0] PC);

assign PC_incby4 = PC + 32'd4;          //increment pc by 4 for next inst
assign jumpaddress={PC_incby4[31:28],({2'd0,Instruction}<<2)};  //jump address cal

always@(negedge Clk , negedge Reset)
begin
   if(Reset==1'b0)          //if reset 0 take startpc address
      PC<=startPC;
   else
      PC<=Address;        // else take the calculated address
end
endmodule
```

2. Instruction Memory

```
`timescale 1ns / 1ps
/*
 * Module: InstructionMemory
 *
 * Implements read-only instruction memory
 * Memory contents are initialized from the file "ImemInit.v"
 */
module InstructionMemory(Data, Address);
        parameter T_rd = 20;
        parameter MemSize = 40;

        output [31:0] Data;
        input [31:0] Address;
        reg [31:0] Data;

        /*
         * ECEN 651 Processor Test Functions
         * Texas A&M University
         */

        always @ (Address) begin
                case(Address)
                /*
                 * Test Program 1:
                 * Sums $a0 words starting at $a1.  Stores the sum at the end of the array
                 * Tests add, addi, lw, sw, beq
                 */

                /*
        main:
                li $t0, 50                              # Initialize the
array to (50, 40, 30)
                sw $t0, 0($0)                   # Store first value
                li $t0, 40
                sw $t0, 4($0)                   # Store Second Value
                li $t0, 30
                sw $t0, 8($0)                   # Store Third Value
                li $a0, 0                               # address of array
                li $a1, 3                               # 3 values to sum
        TestProg1:
                add $t0, $0, $0         # This is the sum
                add $t1, $0, $a0                # This is our array pointer
                add $t2, $0, $0         # This is our index counter
        P1Loop: beq $t2, $a1, P1Done # Our loop
                lw      $t3, 0($t1)                     # Load Array[i]
                add $t0, $t0, $t3               # Add it into the sum
                add $t1, $t1, 4         # Next address
                add $t2, $t2, 1         # Next index
                j P1Loop                                        # Jump to
loop
        P1Done: sw $t0, 0($t1)          # Store the sum at end of array
                lw $t0, 12($0)          # Load Final Value
                nop                                     # Complete
                add $0, $s0, $s0        # do nothing
        */
            32'h00: Data = 32'h34080032;
            32'h04: Data = 32'hac080000;
            32'h08: Data = 32'h34080028;
            32'h0C: Data = 32'hac080004;
            32'h10: Data = 32'h3408001e;
        32'h14: Data = 32'hac080008;
        32'h18: Data = 32'h34040000;
        32'h1C: Data = 32'h34050003;
        32'h20: Data = 32'h00004020;
        32'h24: Data = 32'h00044820;
        32'h28: Data = 32'h00005020;
        32'h2C: Data = 32'h11450005;
        32'h30: Data = 32'h8d2b0000;
        32'h34: Data = 32'h010b4020;
        32'h38: Data = 32'h21290004;
        32'h3C: Data = 32'h214a0001;
        32'h40: Data = 32'h0800000b;
        32'h44: Data = 32'had280000;
        32'h48: Data = 32'h8c08000c;
        32'h4C: Data = 32'h00000000;
        32'h50: Data = 32'h02100020;


    /*
     * Test Program 2:
     * Does some arithmetic computations and stores result in memory
     */


    /*
    main2:
            li      $a0, 32                     # Address of
memory to store result
    TestProg2:
            addi $2, $0, 1              # $2 = 1
            sub     $3, $0, $2          # $3 = -1
            slt     $5, $3, $0          # $5 = 1
            add     $6, $2, $5    # $6 = 2
```

```
			or	$7, $5, $6			# $7 = 3
			sub	$8, $5, $7		# $8 = -2
			and	$9, $8, $7		# $9 = 2
			sw	$9, 0($a0)		# Store $9 in DMem[8]
			lw  $9, 32($0)		# Load Final Value
			nop				# Complete
	*/
		32'h60: Data = 32'h34040020;
		32'h64: Data = 32'h20020001;
		32'h68: Data = 32'h00021822;
		32'h6C: Data = 32'h0060282a;
		32'h70: Data = 32'h00453020;
		32'h74: Data = 32'h00a63825;
		32'h78: Data = 32'h00a74022;
		32'h7C: Data = 32'h01074824;
		32'h80: Data = 32'hac890000;
		32'h84: Data = 32'h8c090020;
		32'h88: Data = 32'h00000000;

	/*
	 * Test Program 3
	 * Test Immediate Function
	 */

	/*
			TestProg3:
			li $a0, 0xfeedbeef		# $a0 = 0xfeedbeef
			sw $a0, 36($0)			# Store $a0 in DMem[9]
			addi $a1, $a0, -2656		# $a1 = 0xfeedb48f
			sw $a1, 40($0)			# Store $a1 in DMem[10]
```

```
addiu $a1, $a0, -2656 # $a1 = 0xfeeeb48f
sw $a1, 44($0)                          # Store $a1 in DMem[11]
andi $a1, $a0, 0xf5a0 # $a1 = 0xb4a0
sw $a1, 48($0)                          # Store $a1 in DMem[12]
sll $a1, $a0, 5                # $a1 = 0xddb7dde0
sw $a1, 52($0)                          # Store $a1 in DMem[13]
srl $a1, $a0, 5                # $a1 = 0x07f76df7
sw $a1, 56($0)                          # Store $a1 in DMem[14]
sra $a1, $a0, 5                # $a1 = 0xfff76df7
sw $a1, 60($0)                          # Store $a1 in DMem[15]
slti $a1, $a0, 1               # $a1 = 1
sw $a1, 64($0)                          # Store $a1 in DMem[16]
slti $a1, $a1, -1                       # $a1 = 0
sw $a1, 68($0)                          # Store $a1 in DMem[17]
sltiu $a1, $a0, 1                       # $a1 = 0
sw $a1, 72($0)                          # Store $a1 in DMem[18]
sltiu $a1, $a1, -1             # $a1 = 1
sw $a1, 76($0)                          # Store $a1 in DMem[19]
xori $a1, $a0, 0xf5a0  # $a1 = 0xfeed4b4f
sw $a1, 80($0)                          # Store $a1 in DMem[20]
lw $a0, 36($0)                          # Load Value to test
lw $a1, 40($0)                          # Load Value to test
lw $a1, 44($0)                          # Load Value to test
lw $a1, 48($0)                          # Load Value to test
lw $a1, 52($0)                          # Load Value to test
lw $a1, 56($0)                          # Load Value to test
lw $a1, 60($0)                          # Load Value to test
lw $a1, 64($0)                          # Load Value to test
lw $a1, 68($0)                          # Load Value to test
lw $a1, 72($0)                          # Load Value to test
```

```
                    lw $a1, 76($0)                          # Load Value to test
                    lw $a1, 80($0)                          # Load Value to test
                    nop                                          # Complete
        */
                32'hA0: Data = 32'h3c01feed;
                32'hA4: Data = 32'h3424beef;
                32'hA8: Data = 32'hac040024;
                32'hAC: Data = 32'h2085f5a0;
                32'hB0: Data = 32'hac050028;
                32'hB4: Data = 32'h2485f5a0;
                32'hB8: Data = 32'hac05002c;
                32'hBC: Data = 32'h3085f5a0;
                32'hC0: Data = 32'hac050030;
                32'hC4: Data = 32'h00042940;
                32'hC8: Data = 32'hac050034;
                32'hCC: Data = 32'h00042942;
                32'hD0: Data = 32'hac050038;
                32'hD4: Data = 32'h00042943;
                32'hD8: Data = 32'hac05003c;
                32'hDC: Data = 32'h28850001;
                32'hE0: Data = 32'hac050040;
                32'hE4: Data = 32'h28a5ffff;
                32'hE8: Data = 32'hac050044;
                32'hEC: Data = 32'h2c850001;
                32'hF0: Data = 32'hac050048;
                32'hF4: Data = 32'h2ca5ffff;
                32'hF8: Data = 32'hac05004c;
                32'hFC: Data = 32'h3885f5a0;
                32'h100: Data = 32'hac050050;
                32'h104: Data = 32'h8c040024;
                32'h108: Data = 32'h8c050028;
                32'h10C: Data = 32'h8c05002c;
                32'h110: Data = 32'h8c050030;
                32'h114: Data = 32'h8c050034;
                32'h118: Data = 32'h8c050038;
                32'h11C: Data = 32'h8c05003c;
                32'h120: Data = 32'h8c050040;
                32'h124: Data = 32'h8c050044;
                32'h128: Data = 32'h8c050048;
                32'h12C: Data = 32'h8c05004c;
                32'h130: Data = 32'h8c050050;
                32'h134: Data = 32'h00000000;


        /*
         * Test Program 4
         * Test jal and jr
         */
        /*
        TestProg4:
                    li $t1, 0xfeed                  # $t1 = 0xfeed
                    li $t0, 0x190                   # Load address of P4jr
                    jr $t0                                  # Jump to P4jr
                    li $t1, 0                   # Check for failure to jump
        P4jr:   sw $t1, 84($0)              # $t1 should be 0xfeed if successful
                    li $t0, 0xcafe                  # $t0 = 0xcafe
                    jal P4Jal                               # Jump to P4Jal
                    li $t0, 0xbabe              # Check for failure to jump
        P4Jal:  sw $t0, 88($0)              # $t0 should be 0xcafe if successful
                    li $t2, 0xface                  # $t2 = 0xface
                    j P4Skip                    # Jump to P4Skip
                    li $t2, 0
        P4Skip:         sw $t2, 92($0)       # $t2 should be 0xface if successful
                    sw $ra, 96($0)                        # Store $ra
                    lw $t0, 84($0)                        # Load value for check
                    lw $t1, 88($0)                        # Load value for check
                    lw $t2, 92($0)                        # Load value for check
                    lw $ra, 96($0)                        # Load value for check

        */
                32'h180: Data = 32'h3409feed;
                32'h184: Data = 32'h34080190;
                32'h188: Data = 32'h01000008;
                32'h18C: Data = 32'h34090000;
                32'h190: Data = 32'hac090054;
                32'h194: Data = 32'h3408cafe;
                32'h198: Data = 32'h0c000068;
                32'h19C: Data = 32'h3408babe;
                32'h1A0: Data = 32'hac080058;
                32'h1A4: Data = 32'h340aface;
                32'h1A8: Data = 32'h0800006c;
                32'h1AC: Data = 32'h340a0000;
                32'h1B0: Data = 32'hac0a005c;
                32'h1B4: Data = 32'hac1f0060;
                32'h1B8: Data = 32'h8c080054;
                32'h1BC: Data = 32'h8c090058;
                32'h1C0: Data = 32'h8c0a005c;
                32'h1C4: Data = 32'h8c1f0060;
                32'h1C8: Data = 32'h00000000;
```

```
/*
 * Test Program 5
 * Tests Overflow Exceptions
 */

/*
Test5-1:
                    li $t0, -2147450880
                    add $t0, $t0, $t0
                    lw $t0, 4($0)       #incorrect if this instruction completes

Test5-2:
                    li $t0, 2147450879
                    add $t0, $t0, $t0
                    lw $t0, 4($0)       #incorrect if this instruction completes

Test 5-3:
                    lw $t0, 4($0)
                    li $t0, -2147483648
                    li $t1, 1
                    sub $t0, $t0, $t1
                    lw $t0, 4($0)

Test 5-4:
                    li $t0, 2147483647
                    mula $t0, $t0, $t0
                    lw $t0, 4($0)
*/
          32'h300: Data = 32'h3c018000;
          32'h304: Data = 32'h34288000;
```

```
/*

            32'h308: Data = 32'h01084020;
            32'h30C: Data = 32'h8c080004;

            32'h310: Data = 32'h3c017fff;
            32'h314: Data = 32'h34287fff;
            32'h318: Data = 32'h01084020;
            32'h31C: Data = 32'h8c080004;

            32'h320: Data = 32'h8c080004;
            32'h324: Data = 32'h3c088000;
            32'h328: Data = 32'h34090001;
            32'h32C: Data = 32'h01094022;
            32'h330: Data = 32'h8c080004;

            32'h334: Data = 32'h3c017FFF;
            32'h338: Data = 32'h3428FFFF;
            32'h33C: Data = 32'h01084038;
            32'h340: Data = 32'h8c080004;

        /*
         * Overflow Exception
         */
        /*
                    lw $t0, 0($0)
        */
            32'hF0000000: Data = 32'h8c080000;

/*
         * Test Program 6
         * Test Branch Prediction performance
```

```
    */
                /*
            li $t5, 0           # initialize data to 0
            li $t0, 100         # initialize exit value
            li $t1, 0           # initialize outer loop index to 0
        outer_loop:
            addi $t1, $t1, 1 #increment outer loop index
            li $t2, 0       #initialize inner loop index to 0
        inner_loop:
            addi $t2, $t2, 1 #increment inner loop index
            addi $t5, $t5, 1 #increment data
            bne $t2, $t0, inner_loop #go back to top of inner loop
            bne $t1, $t0, outer_loop #go back to top of outer loop
            sw $t5, 12($0) #store data into memory
            lw $t5, 12($0) #load data back out of memory
    */
32'h500: Data = 32'h240d0000;
32'h504: Data = 32'h24080064;
32'h508: Data = 32'h24090000;
32'h50C: Data = 32'h21290001;
32'h510: Data = 32'h240a0000;
32'h514: Data = 32'h214a0001;
32'h518: Data = 32'h21ad0001;
32'h51C: Data = 32'h1548fffd;
32'h520: Data = 32'h1528fffa;
32'h524: Data = 32'hac0d000c;
32'h528: Data = 32'h8c0d000c;


    /*
```

```
 * Test Program 7
 * Test Branch Prediction performance again
 */
/*
        li $t5, 0           # initialize data to 0
        li $t0, 100         # initialize exit value
        li $t1, 0           # initialize outer loop index to 0
outer_loop:
        addi $t1, $t1, 1 #increment outer loop index
        li $t2, 0       #initialize inner loop index to 0
inner_loop:
        addi $t2, $t2, 1 #increment inner loop index
        andi $t3, $t2, 2 #mask inner loop index
        li $t4, 1       #set $t4 to 1
        beq $t3, $0, skip1
        li $t4, 0      #set $t4 to 0
skip1:
        beq $t4, $0, skip2
        addi $t5, $t5, 1 #increment data
skip2:
        beq $t2, $t1, exit_inner
        j inner_loop #go back to top of loop
exit_inner:
        beq $t1, $t0, exit_outer
        j outer_loop
exit_outer:
        sw $t5, 12($0) #store data into memory
        lw $t5, 12($0) #load data back out of memory
  */
```

```
    32'h400: Data = 32'h240d0000;
    32'h404: Data = 32'h24080064;
    32'h408: Data = 32'h24090000;
    32'h40C: Data = 32'h21290001;
    32'h410: Data = 32'h240a0000;
    32'h414: Data = 32'h214a0001;
    32'h418: Data = 32'h314b0002;
    32'h41C: Data = 32'h240c0001;
    32'h420: Data = 32'h11600001;
    32'h424: Data = 32'h240c0000;
    32'h428: Data = 32'h11800001;
    32'h42C: Data = 32'h21ad0001;
    32'h430: Data = 32'h11490001;
    32'h434: Data = 32'h08000105;
    32'h438: Data = 32'h11280001;
    32'h43C: Data = 32'h08000103;
    32'h440: Data = 32'hac0d000c;
    32'h444: Data = 32'h8c0d000c;


                default: Data = 32'hXXXXXXXX;
            endcase
     end
endmodule
```

3. <u>Control unit</u>

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:23:34 03/10/2009
// Design Name:
// Module Name:    SingleCycleControl
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
`define RTYPEOPCODE 6'b000000
`define LWOPCODE        6'b100011
`define SWOPCODE        6'b101011
`define BEQOPCODE       6'b000100
`define JOPCODE     6'b000010
`define ORIOPCODE       6'b001101
`define ADDIOPCODE  6'b001000
`define ADDIUOPCODE 6'b001001
`define ANDIOPCODE  6'b001100
`define LUIOPCODE       6'b001111
`define SLTIOPCODE  6'b001010
`define SLTIUOPCODE 6'b001011
`define XORIOPCODE  6'b001110

`define AND     4'b0000
`define OR      4'b0001
`define ADD     4'b0010
`define SLL     4'b0011
`define SRL     4'b0100
`define SUB     4'b0110
`define SLT     4'b0111
`define ADDU    4'b1000
`define SUBU    4'b1001
`define XOR     4'b1010
`define SLTU    4'b1011
`define NOR     4'b1100
`define SRA     4'b1101
`define LUI     4'b1110
`define FUNC    4'b1111

module SingleCycleControl(RegDst, ALUSrc, MemToReg, RegWrite, MemRead,
MemWrite, Branch, Jump, SignExtend, ALUOp, Opcode);
   input [5:0] Opcode;
   output RegDst;
   output ALUSrc;
   output MemToReg;
   output RegWrite;
   output MemRead;
   output MemWrite;
   output Branch;
   output Jump;
   output SignExtend;
   output [3:0] ALUOp;

    reg RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch,
Jump, SignExtend;
    reg  [3:0] ALUOp;
    always @ (Opcode) begin
        case(Opcode)
            `RTYPEOPCODE: begin   // for R type
                RegDst <=  1'b1;
                ALUSrc <=  1'b0;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b0;
                ALUOp <= `FUNC;
            end
            `LWOPCODE: begin            //For Load instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b1;
                RegWrite <=  1'b1;
                MemRead <=  1'b1;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b1;
                ALUOp <= `ADD;
            end
            `SWOPCODE: begin               //For Store instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b1;
                RegWrite <=  1'b0;
                MemRead <=  1'b0;
                MemWrite <= 1'b1;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b1;
                ALUOp <= `ADD;
            end
            `BEQOPCODE: begin        //For Branch instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b0;
                MemToReg <= 1'b0;
                RegWrite <=  1'b0;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b1;
                Jump <= 1'b0;
                SignExtend <= 1'b1;
                ALUOp <= `SUB;
            end
            `JOPCODE: begin                  //For Jump instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b0;
                MemToReg <= 1'b0;
                RegWrite <=  1'b0;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b1;
                SignExtend <= 1'b1;
                ALUOp <= `AND;
            end
            `ORIOPCODE: begin               //For OR instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b0;
                ALUOp <= `OR;
            end
            `ADDIOPCODE: begin            //For ADDI instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b1;
                ALUOp <= `ADD;
            end
            `ADDIUOPCODE: begin        //For ADDI Unsigned instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b0;
                ALUOp <= `ADDU;
            end
            `ANDIOPCODE: begin          // For ANDI instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b0;
                ALUOp <= `AND;
            end
            `LUIOPCODE: begin                 // For Load Upper Immediate
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
                MemToReg <= 1'b0;
                RegWrite <=  1'b1;
                MemRead <=  1'b0;
                MemWrite <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
                SignExtend <= 1'b0;
                ALUOp <= `LUI;
            end
            `SLTIOPCODE: begin          //For Set less than instruction
                RegDst <=  1'b0;
                ALUSrc <=  1'b1;
```

```
        MemToReg <= 1'b0;
        RegWrite <=  1'b1;
        MemRead <=  1'b0;
        MemWrite <= 1'b0;
        Branch <= 1'b0;
        Jump <= 1'b0;
        SignExtend <= 1'b1;
        ALUOp <= `SLT;
    end
    `SLTIUOPCODE: begin       //For Set Less Than Unsigned instruction
        RegDst <=  1'b0;
        ALUSrc <=  1'b1;
        MemToReg <= 1'b0;
        RegWrite <=  1'b1;
        MemRead <=  1'b0;
        MemWrite <= 1'b0;
        Branch <= 1'b0;
        Jump <= 1'b0;
        SignExtend <= 1'b1;
        ALUOp <= `SLTU;
    end
    `XORIOPCODE: begin     //For xor instruction
        RegDst <=  1'b0;
        ALUSrc <=  1'b1;
        MemToReg <= 1'b0;
        RegWrite <=  1'b1;
        MemRead <=  1'b0;
        MemWrite <= 1'b0;
        Branch <= 1'b0;
        Jump <= 1'b0;
        SignExtend <= 1'b0;
        ALUOp <= `XOR;
    end
    default: begin
        RegDst <= 1'bx;
        ALUSrc <= 1'bx;
        MemToReg <= 1'bx;
        RegWrite <= 1'bx;
        MemRead <= 1'bx;
        MemWrite <= 1'bx;
        Branch <= 1'bx;
        Jump <= 1'bx;
        SignExtend <= 1'bx;
        ALUOp <= 4'bxxxx;
    end
endcase
```

```
    end
endmodule
```

4. <u>Register File</u>

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 09/18/2019 08:40:41 AM
// Design Name:
// Module Name: RegisterFile
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module RegisterFile(
    output  [31:0] BusA,
    output  [31:0] BusB,
    input [31:0] BusW,
    input [4:0] RA,
    input [4:0] RB,
    input [4:0] RW,
    input RegWr,
    input Clk
    );
    reg [31:0]registers[31:0];


   assign BusA = (RA==0)? 32'd0 :registers[RA] ;     //Asynchronous read of RA.
   assign BusB = (RB==0)? 32'd0 :registers[RB];      //Asynchronous read of RB.

    always@(negedge Clk)                //Storage at neg egde of clk.
    begin
        if (RegWr && RW!=5'd0)
        begin
            registers[RW] <= BusW;
        end
    end

endmodule
```

5. Sign Extension

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/04/2019 07:33:25 PM
// Design Name:
// Module Name: SignExtender
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module SignExtender(
    output [31:0] signex,
    input SignExtend,
    input [15:0] Instruction
    );
wire signbit;
assign signbit = SignExtend==1'b1 ? Instruction[15] : 0;  // extend 16 bits by 1 if

                                                                MSB is 1
assign signex={{16{signbit}},Instruction[15:0]};   // Sign extend MSB

endmodule
```

6. ALU Control

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
```

```
// Create Date: 10/02/2019 08:12:01 AM
// Design Name:
// Module Name: ALUControl
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

`define SLLFunc  6'b000000
`define SRLFunc  6'b000010
`define SRAFunc  6'b000011
`define ADDFunc  6'b100000
`define ADDUFunc 6'b100001
`define SUBFunc  6'b100010
`define SUBUFunc 6'b100011
`define ANDFunc  6'b100100
`define ORFunc   6'b100101
`define XORFunc  6'b100110
`define NORFunc  6'b100111
`define SLTFunc  6'b101010
`define SLTUFunc 6'b101011

`define AND 4'b0000
`define OR 4'b0001
`define ADD 4'b0010
`define SLL 4'b0011
`define SRL 4'b0100
`define SUB 4'b0110
`define SLT 4'b0111
`define ADDU 4'b1000
`define SUBU 4'b1001
`define XOR 4'b1010
`define SLTU 4'b1011
`define NOR 4'b1100
`define SRA 4'b1101
`define LUI 4'b1110

module ALUControl(ALUCtrl, ALUop, FuncCode);
```

```
output reg [3:0]ALUCtrl;
input [3:0]ALUop;
input [5:0]FuncCode;

always@(*)
begin
   case({ALUop,FuncCode})
   {4'b1111,`SLLFunc}: ALUCtrl<= `SLL;
   {4'b1111,`SRLFunc}: ALUCtrl<= `SRL;
   {4'b1111,`SRAFunc}: ALUCtrl<= `SRA;
   {4'b1111,`ADDFunc}: ALUCtrl<= `ADD;
   {4'b1111,`ADDUFunc}: ALUCtrl<= `ADDU;
   {4'b1111,`SUBFunc}: ALUCtrl<= `SUB;
   {4'b1111,`SUBUFunc}: ALUCtrl<= `SUBU;
   {4'b1111,`ANDFunc}: ALUCtrl<= `AND;
   {4'b1111,`ORFunc}: ALUCtrl<= `OR;
   {4'b1111,`XORFunc}: ALUCtrl<= `XOR;
   {4'b1111,`NORFunc}: ALUCtrl<= `NOR;
   {4'b1111,`SLTFunc}: ALUCtrl<= `SLT;
   {4'b1111,`SLTUFunc}: ALUCtrl<= `SLTU;
   default: ALUCtrl <= ALUop;
   endcase
end

endmodule
```

7. <u>ALU</u>

```
`timescale 1ns / 1ps
`define AND 4'b0000
`define OR 4'b0001
`define ADD 4'b0010
`define SLL 4'b0011
`define SRL 4'b0100
`define SUB 4'b0110
`define SLT 4'b0111
`define ADDU 4'b1000
`define SUBU 4'b1001
`define XOR 4'b1010
`define SLTU 4'b1011
`define NOR 4'b1100
`define SRA 4'b1101
`define LUI 4'b1110
module ALU(BusW, Zero, BusA, BusB, ALUCtrl
   );
input wire [31:0] BusA, BusB;
output reg [31:0] BusW;
input wire [3:0] ALUCtrl ;
```

```
output wire Zero ;

wire less;
wire [63:0] Bus64;
assign Zero = (BusW==32'd0 ? 1'b1 : 1'b0);
assign less = ({1'b0,BusA} < {1'b0,BusB}  ? 1'b1 : 1'b0);
assign Bus64 = {32'd0,BusW};
always@(*)begin
            case (ALUCtrl)                    // Assign according to ALU arithmetic operations
            `AND:   BusW <= BusA & BusB;
            `OR:    BusW <= BusA | BusB;
            `ADD:   BusW <= BusA + BusB;
            `ADDU:  BusW <= BusA + BusB;
            `SLL:   BusW <= BusA << BusB;
            `SRL:   BusW <= BusA >> BusB;
            `SUB:   BusW <= BusA - BusB;
            `SUBU:  BusW <= BusA - BusB;
            `XOR:   BusW <= BusA ^ BusB;
            `NOR:   BusW <= ~(BusA|BusB);
    `SLT:   if(BusA[31] != BusB[31])
                    begin
                      if(BusA[31] > BusB[31])begin
                       BusW <= 1;
                       end else begin
                       BusW <= 0;
                       end
                       end else begin
                                    if (BusA < BusB)
                                    begin
                                        BusW <= 1;
                                    end
                                    else
                                    begin
                                        BusW <= 0;
                                    end
                                end


            `SLTU: if (BusA < BusB)
                    begin
                      BusW <= 1;
                     end
                      else
                    begin
                    BusW <= 0;
                     end
```

```
        `SRA:   BusW <= $signed(BusA)>>>$signed(BusB) ;
        `LUI:   BusW <= BusB<<16;
        default:BusW <= 32'd0;
        endcase
end
endmodule
```

8. Data Memory

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 09/18/2019 09:08:30 AM
// Design Name:
// Module Name: DataMemory
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Data_memory(ReadData, Address, WriteData, MemoryRead, MemoryWrite,
Clock);


   output reg [31:0] ReadData;
   input wire [31:0] WriteData;
   input wire [5:0] Address ;

   input wire MemoryRead, MemoryWrite ;
   input wire Clock;

   reg [31:0] regs [63:0];

   integer i;
     initial begin
     for (i = 0; i < 32; i = i + 1) begin
```

```
                    regs[i] <= 0;
                 end
                 end

  always @ (negedge Clock) begin
    if(MemoryWrite) begin                //Memory write on the neg edge of clock
      regs[Address] <= WriteData;
      end
   end

   always @ (posedge Clock) begin          //Memory read on the pos edge of clock
      if(MemoryRead) begin
      ReadData = regs[Address];
        //regs[Address] <= WriteData;
        end
     end


endmodule
```

9. <u>Single Cycle Processor</u>

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/04/2019 04:45:52 PM
// Design Name:
// Module Name: SingleCycleProc
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module SingleCycleProc(CLK, Reset_L, startPC, dMemOut);

input CLK,Reset_L;
input [31:0] startPC;
```

```
output [31:0] dMemOut;

wire [31:0] Instruction,ALUresult,branch_addr;
// Local Signals for Single cycle control
wire [31:0]
PC,ReadData1,ReadData2,WriteData,PC_incby4,Address,jumpaddress,PC_imm_add
ed,signex,ALUIP2,ReadDataMem;

wire RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump,
SignExtend,BEQ_indicator,Zero;

wire [3:0] ALUop,ALUCtrl;
wire [4:0] reg_write_addr;

wire [31:0] final_bus_B;
wire [31:0] final_bus_A;

assign dMemOut=ReadDataMem;
InstructionMemory I1(.Data(Instruction),.Address(PC));
//DEfining PC
ProgramCounter Progcnt(.Address(Address),
            .Clk(CLK),
            .Instruction(Instruction[26:0]),
            .PC_incby4(PC_incby4),
            .jumpaddress(jumpaddress),
            .Reset(Reset_L),
            .startPC(startPC),
            .PC(PC));
//Defining Data Memory
Data_memory Memory_module(
          .ReadData(ReadDataMem),
          .Address(ALUresult),
          .WriteData(ReadData2),
          .MemoryRead(MemRead),
          .MemoryWrite(MemWrite),
          .Clock(CLK)
                   );
//Defining Main ALU
ALU ALU_actual(.BusW(ALUresult),
           .Zero(Zero),
           .BusA(final_bus_A),
          .BusB(final_bus_B),
          .ALUCtrl(ALUCtrl)
          );
//Defining Register File
RegisterFile RegFile1(.RA(Instruction[25:21]),
                        .RB(Instruction[20:16]),
                          .RW(reg_write_addr),
                            .BusA(ReadData1),
                          .BusB(ReadData2),
                          .BusW(WriteData),
                           .RegWr(RegWrite),
                           .Clk(CLK)
                                                  );
//Defining sign extention module
SignExtender Signextender1(
             .signex(signex),
         .SignExtend(SignExtend),
       .Instruction(Instruction[15:0])
          );
//Defining alu control
ALUControl AlUController(.ALUCtrl(ALUCtrl),
                              .ALUop(ALUop),
               .FuncCode(Instruction[5:0]));
//Defining control unit
SingleCycleControl ControlUnit1(.RegDst(RegDst),
                             .ALUSrc(ALUSrc),
                            .MemToReg(MemToReg),
                           .RegWrite(RegWrite),
                             .MemRead(MemRead),
                           .MemWrite(MemWrite),
                            .Branch(Branch),
                            .Jump(Jump),
                            .SignExtend(SignExtend),
                            .ALUOp(ALUop),
                            .Opcode(Instruction[31:26]));
//For branch add cal
assign PC_imm_added=PC_incby4+(signex<<2);
//Branch address calculation
assign branch_addr= BEQ_indicator==1'b1 ? PC_imm_added : PC_incby4;
//Jump Address calculation
assign Address = Jump==1'b1 ? jumpaddress : branch_addr;
//To inticate branch is taken
assign BEQ_indicator=Branch && Zero;
 // Mux for destination register in instruction before register file
assign reg_write_addr= RegDst==1'b1 ? Instruction[15:11] : Instruction[20:16];
//Sign extend if ALUSrc is high
assign ALUIP2 = ALUSrc==1'b1 ? signex : ReadData2;
//For shift operation
assign final_bus_B =( (ALUCtrl == 4'b0011) ? Instruction[10:6]  : (ALUCtrl ==
4'b0100)? Instruction[10:6] :(ALUCtrl == 4'b1101)? Instruction[10:6]:ALUIP2 );
```

```
        assign final_bus_A = ( (ALUCtrl == 4'b0011) ? ReadData2  : (ALUCtrl == 4'b0100)?
        ReadData2 :(ALUCtrl == 4'b1101)? ReadData2 :ReadData1 );
        //Write data to memory when Memtoreg is high
        assign WriteData = MemToReg == 1'b1 ? ReadDataMem : ALUresult ;
        endmodule
```

TESTBENCH:

```
`timescale 1ns / 1ps


////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   19:56:09 10/13/2017
// Design Name:   SingleCycleProc
// Module Name:   /home/grads/y/yc2704/ecen651/newlab5/tb_top.v
// Project Name:  newlab5
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: SingleCycleProc
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
`define STRLEN 32
`define HalfClockPeriod 60
```

```
`define ClockPeriod `HalfClockPeriod * 2
module SingleCycleProcTest_v;


        task passTest;
                input [31:0] actualOut, expectedOut;
                input [`STRLEN*8:0] testType;
                inout [7:0] passed;


                if(actualOut == expectedOut) begin $display ("%s passed", testType); passed
= passed + 1; end
                else $display ("%s failed: 0x%x should be 0x%x", testType, actualOut,
expectedOut);
        endtask


        task allPassed;
                input [7:0] passed;
                input [7:0] numTests;


                if(passed == numTests) $display ("All tests passed");
                else $display("Some tests failed: %d of %d passed", passed, numTests);
        endtask


        // Inputs
        reg CLK;
        reg Reset_L;
        reg [31:0] startPC;
        reg [7:0] passed;


        // Outputs
        wire [31:0] dMemOut;
```

```
// Instantiate the Unit Under Test (UUT)
SingleCycleProc uut (
        .CLK(CLK),
        .Reset_L(Reset_L),
        .startPC(startPC),
        .dMemOut(dMemOut)
);

initial begin
        // Initialize Inputs
        Reset_L = 1;
        startPC = 0;
        passed = 0;

        // Wait for global reset
        #(1 * `ClockPeriod);

        // Program 1
        #1
        Reset_L = 0; startPC = 0;
        #(1 * `ClockPeriod);
        Reset_L = 1;
        #(33 * `ClockPeriod);
        passTest(dMemOut, 120, "Results of Program 1", passed);

        // Program 2
        #(1 * `ClockPeriod)
        Reset_L = 0; startPC = 32'h60;
        #(1 * `ClockPeriod);
        Reset_L = 1;
```

```
#(11 * `ClockPeriod);
passTest(dMemOut, 2, "Results of Program 2", passed);

// Program 3
#(1 * `ClockPeriod)
Reset_L = 0; startPC = 32'hA0;
#(1 * `ClockPeriod);
Reset_L = 1;
#(26 * `ClockPeriod);
passTest(dMemOut, 32'hfeedbeef, "Result 1 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hfeedb48f, "Result 2 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hfeeeb48f, "Result 3 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'h0000b4a0, "Result 4 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hddb7dde0, "Result 5 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'h07f76df7, "Result 6 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 32'hfff76df7, "Result 7 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 1, "Result 8 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 0, "Result 9 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 0, "Result 10 of Program 3", passed);
#(1 * `ClockPeriod);
passTest(dMemOut, 1, "Result 11 of Program 3", passed);
```

```
                #(1 * `ClockPeriod);
                passTest(dMemOut, 32'hfeed4b4f, "Result 12 of Program 3", passed);

                // Done
                allPassed(passed, 14);
                $stop;
        end

  initial begin
    CLK = 0;
  end

  // The following is correct if clock starts at LOW level at StartTime //
  always begin
    #`HalfClockPeriod CLK = ~CLK;
    #`HalfClockPeriod CLK = ~CLK;
  end

endmodule
```