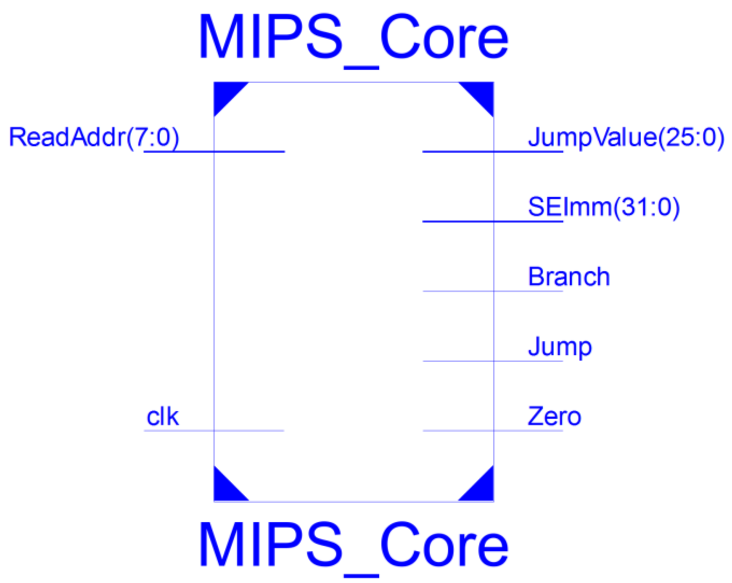
CECS 440
MIPS Core

OBJECTIVES:

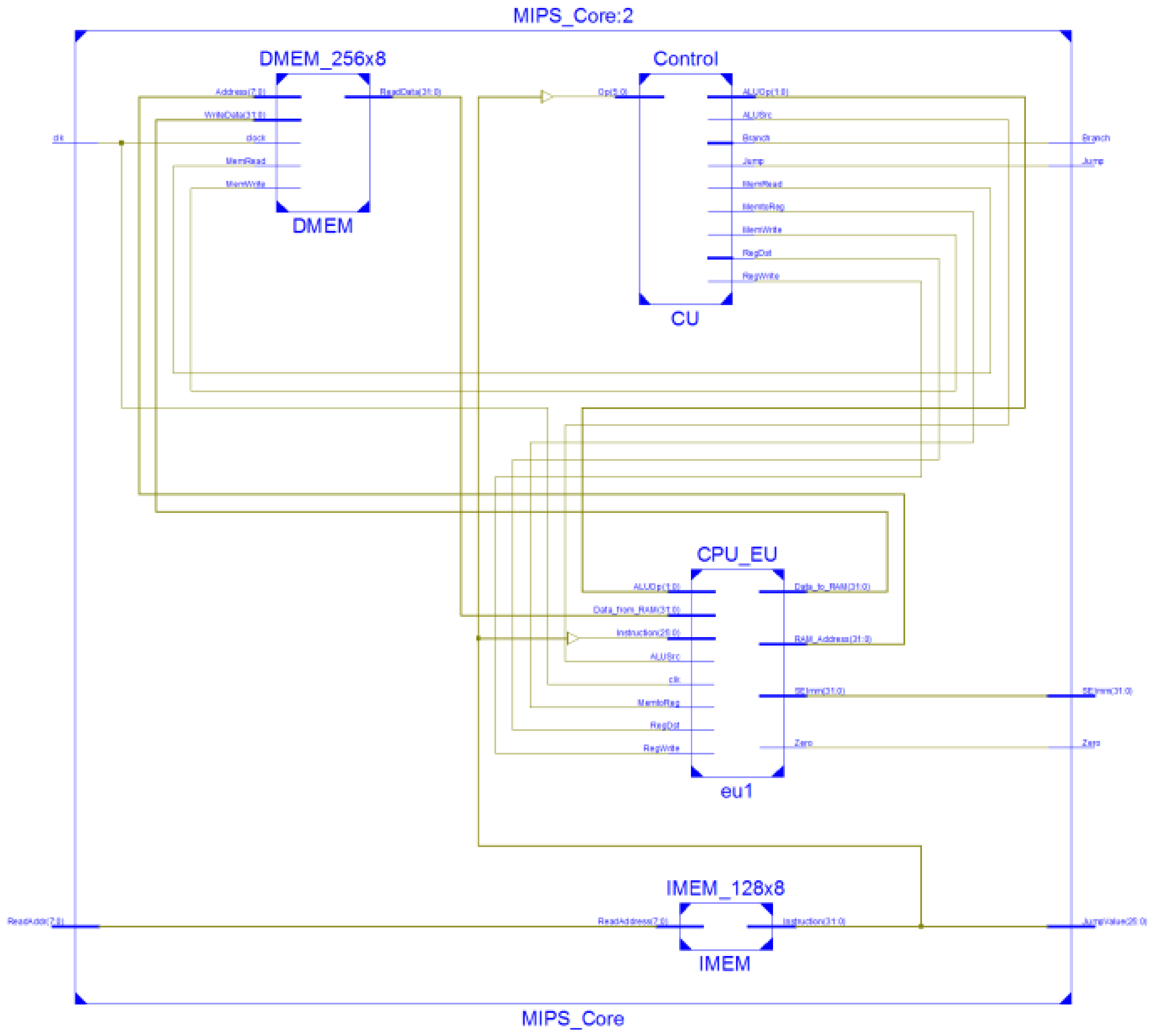
- Connect DataMemory, InstructionMemory, and ControlUnit to the MIPS CPU Execution Unit To form the Core of the MIPS processor.
- Test the MIPS Core for correct operation.

The MIPS Core functional block is depicted below:



The MIPS Core module contains all the functional blocks needed to process data and and provide values used in control flow instructions. The internals of the MIPS Core are depicted on the following page.

CECS 440
MIPS Core



ACTIVITY 1

Make the MIPS Core.

First make the Data Cache according to the following verilog module definition:

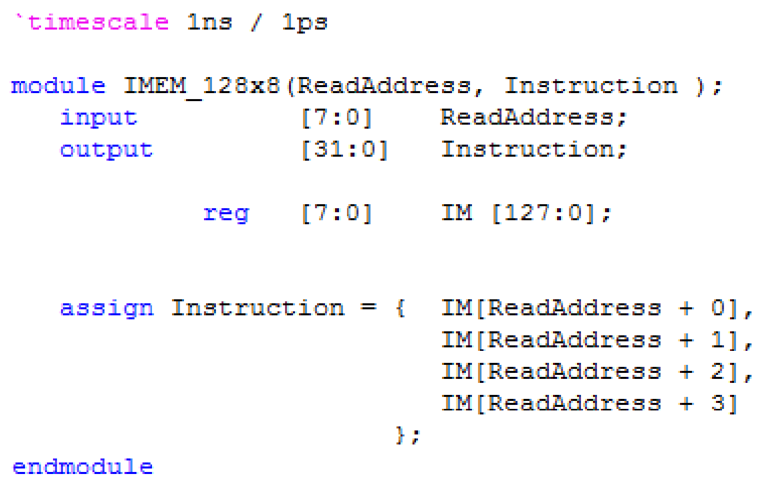
```
`timescale 1ns / 1ps

module DMEM_256x8(          clock, MemWrite, Address,
                            WriteData, MemRead, ReadData );

   input                    clock, MemWrite, MemRead;
   input           [7:0]    Address;
   input           [31:0]   WriteData;
   output    reg   [31:0]   ReadData;

             reg   [7:0]    DM [0:255];

   always    @(posedge clock)
   begin
      if(MemWrite == 1)
      begin
         DM[Address + 0]   <= WriteData[31:24];
         DM[Address + 1]   <= WriteData[23:16];
         DM[Address + 2]   <= WriteData[15:8];
         DM[Address + 3]   <= WriteData[7:0];
      end
   end


   always    @(MemRead, Address, DM)
   begin
      if(MemRead == 1)
      begin
         ReadData <= {  DM[Address + 0],
                        DM[Address + 1],
                        DM[Address + 2],
                        DM[Address + 3]   };
      end
   end

endmodule
```

Data cache is the DMEM functional block referred to in lecture, the Data portion of the L1 Split cache. The size of the Data cache has been reduced to work correctly in Xilinx.

Next create the Instruciton Cache.

CECS 440
MIPS Core

Make the Instruction Cache according to the following verilog module definition:

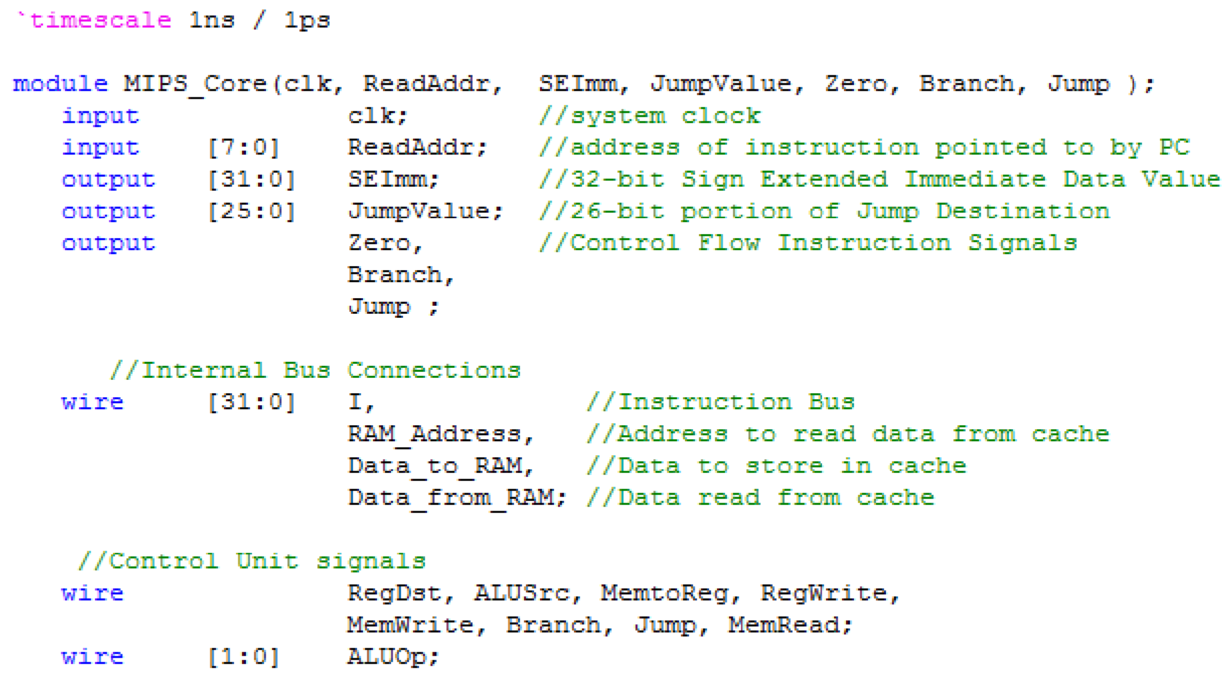


```
`timescale 1ns / 1ps

module IMEM_128x8(ReadAddress, Instruction );
   input          [7:0]    ReadAddress;
   output         [31:0]   Instruction;

         reg      [7:0]    IM [127:0];


   assign Instruction = {  IM[ReadAddress + 0],
                           IM[ReadAddress + 1],
                           IM[ReadAddress + 2],
                           IM[ReadAddress + 3]
                        };
endmodule
```

The Instruction Cache size has been reduced to work correctly in Xilinx. Also IMEM does not need to be very large because the programs being run on our simulated MIPS processor will not be so elaborate.

With those Cache Memory modules created, they can be connected to the Execution Unit to make the Core of our MIPS processor.

The following screenshot depicts what you will use for the MIPS_Core module declaration, I/O Port declarations, and internal bus/wire declarations (you may need more or less internal buses):



```
`timescale 1ns / 1ps

module MIPS_Core(clk, ReadAddr,  SEImm, JumpValue, Zero, Branch, Jump );
   input                 clk;        //system clock
   input      [7:0]      ReadAddr;   //address of instruction pointed to by PC
   output     [31:0]     SEImm;      //32-bit Sign Extended Immediate Data Value
   output     [25:0]     JumpValue;  //26-bit portion of Jump Destination
   output                Zero,       //Control Flow Instruction Signals
                         Branch,
                         Jump ;

      //Internal Bus Connections
   wire       [31:0]     I,             //Instruction Bus
                         RAM_Address,   //Address to read data from cache
                         Data_to_RAM,   //Data to store in cache
                         Data_from_RAM; //Data read from cache

    //Control Unit signals
   wire                  RegDst, ALUSrc, MemtoReg, RegWrite,
                         MemWrite, Branch, Jump, MemRead;
   wire       [1:0]      ALUOp;
```

ACTIVITY 2

Test the MIPS Core. Here are the portions of the verilog test fixture to create. The first section automatically generated by Xilinx and should appear as follows:

```
`timescale 1ns / 1ps

module MIPS_Core_Tester;

   // Inputs
   reg              clk;
   reg    [7:0]     ReadAddr;

   // Outputs
   wire   [31:0]    SEImm;
   wire   [25:0]    JumpValue;
   wire             Zero;
   wire             Branch;
   wire             Jump;

   // Instantiate the Unit Under Test (UUT)
   MIPS_Core uut (
      .clk(        clk         ),
      .ReadAddr(   ReadAddr    ),
      .SEImm(      SEImm       ),
      .JumpValue(  JumpValue   ),
      .Zero(       Zero        ),
      .Branch(     Branch      ),
      .Jump(       Jump        )
   );
```

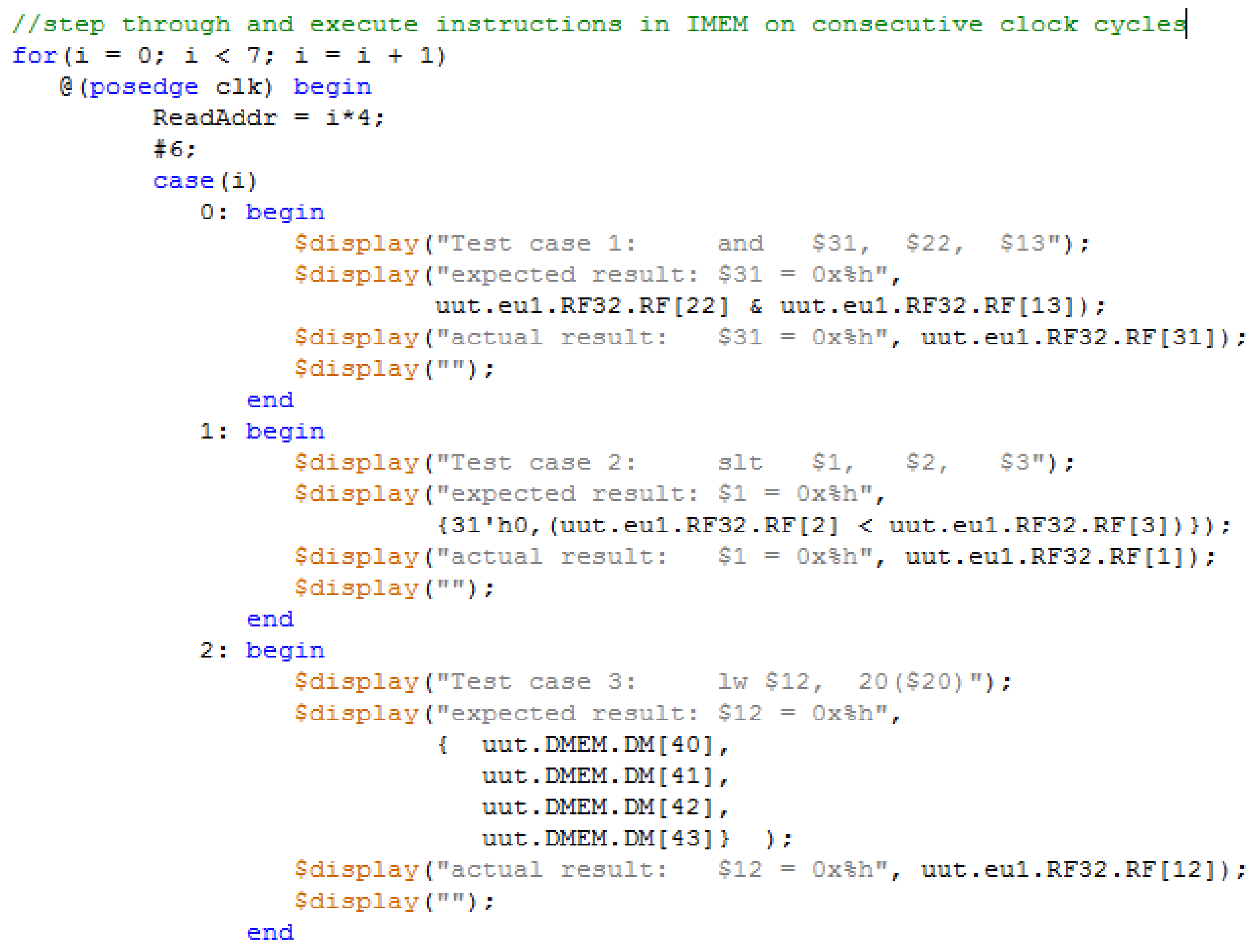
Next declare an integer, make a clock pulse generator, and use loops to initialize data memories:

```
 integer i;

 always   #5 clk   =  ~clk; //clock pulse generator

 initial begin
    //Initialize Inputs
    clk      =  0;
    ReadAddr =  0;

    //initialize register values
    for(i = 1; i < 32; i = i + 1)
          uut.eu1.RF32.RF[i] = i;

    //initialize DMEM
    for(i = 0; i < 256; i = i + 1)
          uut.DMEM.DM[i] = $random;
```

Next within the initial block the instruction memory will be assigned values according to the machine code for 6 different instructions. The assignment statements have been set up but for the right hand side of the assignment you must determine the 32-bit hex machine code for each instruction in the comment.

```
//-+--+--+--+- initialize IMEM with instructions   -+--+--+--+-//
   //instruction 0:  and   $31,  $22,  $13
{uut.IMEM.IM[0],uut.IMEM.IM[1],uut.IMEM.IM[2],uut.IMEM.IM[3]}     =

   //instruction 1:  slt   $1,   $2,   $3
{uut.IMEM.IM[4],uut.IMEM.IM[5],uut.IMEM.IM[6],uut.IMEM.IM[7]}     =

   //instruction 2:  lw $12,  20($20)
{uut.IMEM.IM[8],uut.IMEM.IM[9],uut.IMEM.IM[10],uut.IMEM.IM[11]}   =

   //instruction 3:  sw $4,   0($0)
{uut.IMEM.IM[12],uut.IMEM.IM[13],uut.IMEM.IM[14],uut.IMEM.IM[15]} =

   //instruction 4:  beq   $5,   $6,   0xffffffff
{uut.IMEM.IM[16],uut.IMEM.IM[17],uut.IMEM.IM[18],uut.IMEM.IM[19]} =

   //instruction 5:  sub   $7,   $8,   $9
{uut.IMEM.IM[20],uut.IMEM.IM[21],uut.IMEM.IM[22],uut.IMEM.IM[23]} =

  //instruction 6:  jump to I-MEM location  0x00400010
{uut.IMEM.IM[24],uut.IMEM.IM[25],uut.IMEM.IM[26],uut.IMEM.IM[27]} =

//-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+-//
```

Next test cases will be written to the Xilinx Simulation console. Only important signal values will be monitored in the console outputs. This type of functionality checking is sufficient for basic testing but troubleshooting will likely require analysis of waveform signals. The test case definitions are given on the following page:



```
//step through and execute instructions in IMEM on consecutive clock cycles
for(i = 0; i < 7; i = i + 1)
   @(posedge clk) begin
         ReadAddr = i*4;
         #6;
         case(i)
            0: begin
                  $display("Test case 1:     and   $31,  $22,  $13");
                  $display("expected result: $31 = 0x%h",
                           uut.eu1.RF32.RF[22] & uut.eu1.RF32.RF[13]);
                  $display("actual result:   $31 = 0x%h", uut.eu1.RF32.RF[31]);
                  $display("");
               end
            1: begin
                  $display("Test case 2:     slt   $1,   $2,   $3");
                  $display("expected result: $1 = 0x%h",
                           {31'h0,(uut.eu1.RF32.RF[2] < uut.eu1.RF32.RF[3])});
                  $display("actual result:   $1 = 0x%h", uut.eu1.RF32.RF[1]);
                  $display("");
               end
            2: begin
                  $display("Test case 3:     lw $12,  20($20)");
                  $display("expected result: $12 = 0x%h",
                           {  uut.DMEM.DM[40],
                              uut.DMEM.DM[41],
                              uut.DMEM.DM[42],
                              uut.DMEM.DM[43]}  );
                  $display("actual result:   $12 = 0x%h", uut.eu1.RF32.RF[12]);
                  $display("");
               end
```

Continuation of the test cases occur on the following page:

```
3: begin
      $display("Test case 4:     sw $4,   0($0)");
      $display("expected result: DMEM[0] = 0x%h", uut.eu1.RF32.RF[4]);
      #5;   //delay to wait for store to occur on next positive clock edge
      $display("actual result:   DMEM[0] = 0x%h",
                  {  uut.DMEM.DM[0],
                     uut.DMEM.DM[1],
                     uut.DMEM.DM[2],
                     uut.DMEM.DM[3]}    );
      $display("");
   end
4: begin
      $display("Test case 5:     beq   $5,   $6,   0xffff");
      $display("expected results:");
      $display("  Zero   = 0");
      $display("  Branch   = 1");
      $display("  SeIMM = 0xffffffff");
      $display("actual results:");
      $display("  Zero   = %b",   Zero);
      $display("  Branch   = %b", Branch);
      $display("  SeIMM = 0x%h", SEImm);
      $display("");
   end
5: begin
      $display("Test case 6:     sub   $7,   $8,   $9");
      $display("expected result: $7 = 0x%h",
                 uut.eu1.RF32.RF[8] - uut.eu1.RF32.RF[9]);
      $display("actual result:   $7 = 0x%h", uut.eu1.RF32.RF[7]);
      $display("");
   end
   6: begin
         $display("Test case 7:     jump to IMEM address 0x00400010");
         $display("expected result: JumpValue = 0x0100004");
         $display("actual result:   JumpValue = 0x%h", JumpValue);
         $display("");
      end
   default: begin
         $display("INVALID TEST CASE");
      end
endcase
end         //end of @(posedge clk) within the for loop
#20 $stop;
end         //end of initial block
endmodule
```

Run your test fixture and observe the results in the console. Do a manual verification of the test results to ensure correctness.

Deliverables:

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

1. A general description of what a Processor Core is
2. Fully commented **MIPS_Core.v** from Activity 1
3. Fully commented Verilog Test Fixture source code from Activity 2
4. Console output text showing the results of each test case