EE304 – Digital Design with HDL (II)

Lecture 15

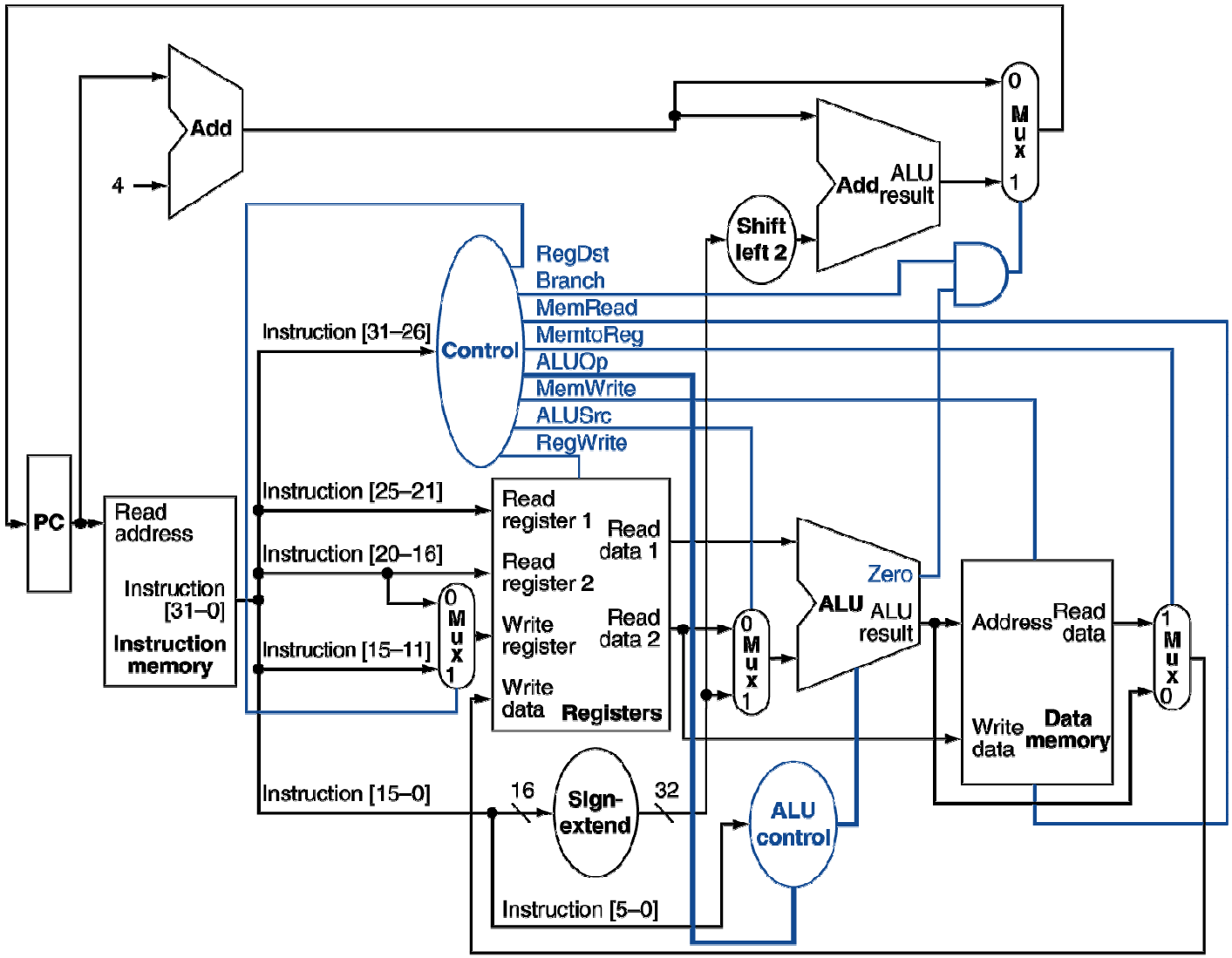
Verilog for MIPS Processors

Dr. Ming Xu

Dept of Electrical & Electronic Engineering

XJTLU

Single-Cycle Processor



```
// MIPS single Cycle processor originaly developed for simulation by Patterson and Hennesy
// Modified for synthesis using the Quartus II package by Dr. S. Ami-Nejad. Feb. 2009

// Register File
module RegisterFile (Read1,Read2,Writereg,WriteData,RegWrite, Data1, Data2,clock,reset);

    input  [4:0] Read1,Read2,Writereg; // the registers numbers to read or write
    input  [31:0] WriteData;     // data to write
    input  RegWrite;  // The write control
    input  clock, reset; // The clock to trigger writes
    output [31:0] Data1, Data2; // the register values read;
    reg    [31:0] RF[31:0]; // 32 registers each 32 bits long
    integer k;

    // Read from registers independent of clock
    assign Data1 = RF[Read1];
    assign Data2 = RF[Read2];

    always @(posedge clock or posedge reset)
        if (reset) for(k=0;k<32;k=k+1) RF[k]<=32'h00000000;
        // Register 0 is a read only register with the content of 0
        else       if (RegWrite & (Writereg!=0)) RF[Writereg] <= WriteData;
endmodule
```

```
//ALU Control
module ALUControl (ALUOp, FuncCode, ALUCtl);

    input  [1:0]      ALUOp;
    input  [5:0]      FuncCode;
    output[3:0]       ALUCtl;
    reg               [3:0]       ALUCtl;

    always@( ALUOp, FuncCode)
    begin
    case(ALUOp)
    2'b00: ALUCtl = 4'b0010;
    2'b01: ALUCtl = 4'b0110;
    2'b10: case(FuncCode)
                      6'b 100000: ALUCtl = 4'b 0010;
                      6'b 100010: ALUCtl = 4'b 0110;
                      6'b 100100: ALUCtl = 4'b 0000;
                      6'b 100101: ALUCtl = 4'b 0001;
                      6'b 101010: ALUCtl = 4'b 0111;
                      default:    ALUCtl = 4'b xxxx;
                      endcase
    default:          ALUCtl = 4'b xxxx;
    endcase
    end
endmodule
```

```
//ALU
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
    input  [3:0]       ALUctl;
    input  [31:0]      A,B;
    output[31:0]       ALUOut;
    output             Zero;
    reg                [31:0] ALUOut;

    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
    always @(ALUctl, A, B) begin //reevaluate if these change
    case (ALUctl)
        0: ALUOut <= A & B;
        1: ALUOut <= A | B;
        2: ALUOut <= A + B;
        6: ALUOut <= A - B;
        7: ALUOut <= A < B ? 1:0;
        // .... Add more ALU operations here
        default: ALUOut <= A;
        endcase
    end
endmodule
```

```
// Data Memory
module DataMemory(Address, DWriteData, MemRead, MemWrite, clock, reset,
DReadData);
input       [31:0]      Address, DWriteData;
input       MemRead, MemWrite, clock, reset;
output      [31:0]      DReadData;
reg         [31:0]      DMem[7:0];

assign  DReadData = DMem[Address[2:0]];
always @(posedge clock or posedge reset)begin
            if (reset) begin
                        DMem[0]=32'h00000005;
                        DMem[1]=32'h0000000A;
                        DMem[2]=32'h00000055;
                        DMem[3]=32'h000000AA;
                        DMem[4]=32'h00005555;
                        DMem[5]=32'h00008888;
                        DMem[6]=32'h00550000;
                        DMem[7]=32'h00004444;
                        end else
                        if (MemWrite) DMem[Address[2:0]] <= DWriteData;
            end
endmodule
```

```
// Main Controller
module Control
(opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite);
input       [5:0]       opcode;
output      [1:0]       ALUOp;
output      RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;
reg         [1:0]       ALUOp;
reg         RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

parameter R_Format = 6'b000000, LW = 6'b100011, SW = 6'b101011,
BEQ=6'b000100;
always @(opcode)begin
     case(opcode)
          R_Format:{RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp}= 9'b 100100010;
          LW:         {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp}= 9'b 011110000;
          SW:         {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp}= 9'b x1x001000;
          BEQ:        {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp}= 9'b x0x000101;
          default: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,
Branch,ALUOp}= 9'b xxxxxxxxx;
          endcase
     end
endmodule
```

```
// Datapath (Part 1)
module DataPath(RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite,
ALUSrc, RegWrite, clock, reset, opcode,/* RF1,  RF2,
RF3,*/ALUResultOut ,DReadData);

input       RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,clock,
reset;
input       [1:0]       ALUOp;
output      [5:0]       opcode;
output      [31:0]      /*RF1,  RF2, RF3,*/ ALUResultOut ,DReadData;

reg         [31:0] PC, IMemory[0:31];
wire        [31:0] SignExtendOffset, PCOffset, PCValue, ALUResultOut,
                   IAddress, DAddress, IMemOut, DmemOut, DWriteData, Instruction,
                   RWriteData, DReadData, ALUAin, ALUBin;
wire        [3:0] ALUctl;
wire        Zero;
wire        [4:0] WriteReg;

//Instruction fields, to improve code readability
wire [5:0]  funct;
wire [4:0]  rs, rt, rd, shamt;
wire [15:0] offset;
```

```
// Datapath (Part 2)

//Instantiate local ALU controller
ALUControl alucontroller(ALUOp,funct,ALUctl);

// Instantiate ALU
MIPSALU ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);

// Instantiate Register File
RegisterFile REG(rs, rt, WriteReg, RWriteData, RegWrite, ALUAin,
DWriteData,clock,reset);

// Instantiate Data Memory
DataMemory datamemory(ALUResultOut, DWriteData, MemRead, MemWrite, clock,
reset, DReadData);

// Instantiate Instruction Memory
IMemory  IMemory_inst (
      .address ( PC[6:2] ),
      .q ( Instruction )
      );
```

```
// Datapath (Part 3)
// Synthesize multiplexers
assign      WriteReg = (RegDst) ? rd : rt;
assign      ALUBin = (ALUSrc) ? SignExtendOffset : DWriteData;
assign      PCValue  = (Branch & Zero)    ? PC+4+PCOffset    : PC+4;
assign      RWriteData = (MemtoReg) ? DReadData : ALUResultOut;

// Acquire the fields of the R_Format Instruction for clarity
assign {opcode, rs, rt, rd, shamt, funct} = Instruction;
// Acquire the immediate field of the I_Format instructions
assign offset = Instruction[15:0];
//sign-extend lower 16 bits
assign SignExtendOffset = { {16{offset[15]}} , offset[15:0]};
// Multiply by 4 the PC offset
assign PCOffset = SignExtendOffset << 2;
// Write the address of the next instruction into the program counter
always @(posedge clock ) begin
if (reset) PC<=32'h00000000; else
      PC <= PCValue;
end
endmodule
```

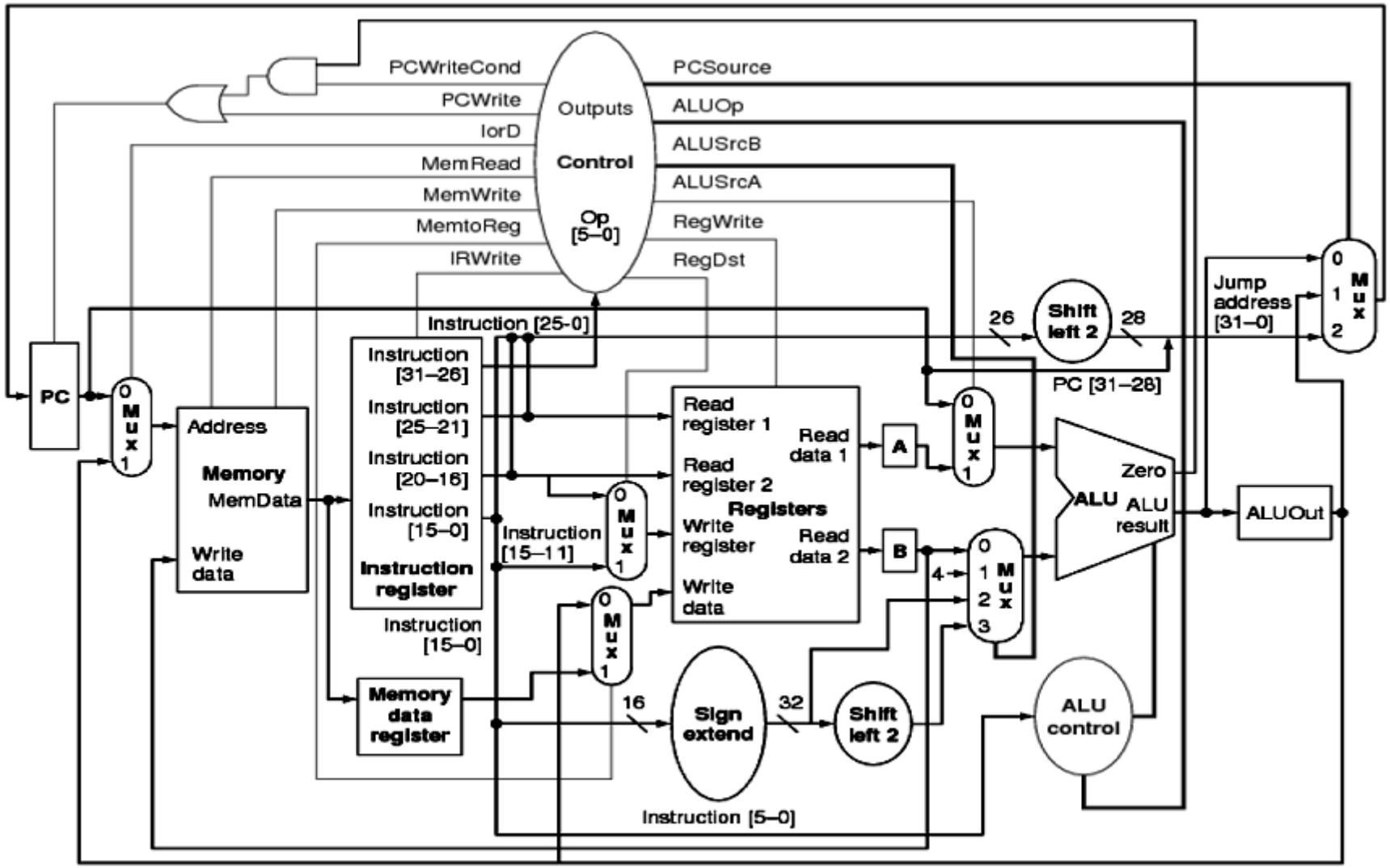
```
// Processor
module MIPS1CYCLE(clock, reset,opcode, ALUResultOut ,DReadData);
    input  clock,      reset;
    output[5:0]        opcode;
    output[31:0]       ALUResultOut ,DReadData; // For simulation purposes

    wire [1:0] ALUOp;
    wire [5:0] opcode;
    wire [31:0] SignExtend,ALUResultOut ,DReadData;
    wire RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

    // Instantiate the Datapath
    DataPath MIPSDP (RegDst,Branch,MemRead,MemtoReg,ALUOp,
    MemWrite,ALUSrc,RegWrite,clock, reset, opcode, ALUResultOut ,DReadData);

    //Instantiate the combinational control unit
    Control MIPSControl (opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,
MemWrite,ALUSrc,RegWrite);
endmodule
```

Multi-Cycle Processor



```
// Part 1
// The behavioural specification of a MIPS multicycle processor originaly developed for simulation by Patterson and Hennesy
// Modified by Dr. M. Xu in June 2010

module CPU (clock, reset);
parameter R_Type = 6’b000000, LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6’000010;
input clock, reset; //external inputs
// The architecturally visible registers and scratch registers for implementation
reg [31:0] PC, Regs[0:31], Memory [0:31], IR, ALUOut, MDR, A, B;
reg [2:0] state; // processor state
wire [5:0] opcode; //use to get opcode easily
wire [31:0] SignExtend,PCOffset; //used to get sign extended offset field
assign opcode = IR[31:26]; //opcode is upper 6 bits
assign SignExtend = {{16{IR[15]}},IR[15:0]}; //sign extension of lower 16-bits of instruction
assign PCOffset = SignExtend << 2; //PC offset is shifted
// set the PC to 0 and start the control in state 1
initial begin PC = 0; state = 1; end
```

```
// Part 2
//The state machine--triggered on a rising clock
always @(posedge clock or posedge reset) begin
Regs[0] = 0; // to make sure R0 is always 0

if (reset) begin
      PC = 0;
      state = 1;
      Memory[16]=32'h00000005;   // words 16-31 used for data memory
      Memory[17]=32'h0000000B;  //  words 0-15 used for instruction memory
      Memory[18]=32'h00000055;
      Memory[19]=32'h000000BB;
      Memory[20]=32'h00005555;
      Memory[21]=32'h0000BBBB;
      Memory[22]=32'h55555555;
      Memory[23]=32'hBBBBBBBB;
      //  add the machine code of your instructions here
end

case (state) //action depends on the state
1: begin // first step: fetch the instruction, increment PC, go to next state
IR <= Memory[PC>>2];
PC <= PC + 4;
state=2; //next state
end
```

```
// Part 3
2: begin // second step: Instruction decode, register fetch, also branch address
A <= Regs[IR[25:21]];
B <= Regs[IR[20:16]];
state= 3;
ALUOut <= PC + PCOffset; // compute PC-relative branch target
end

3: begin // third step: Load/store execution, ALU execution, Branch completion
state =4; // default next state
if ((opcode==LW) |(opcode==SW)) ALUOut <= A + SignExtend; // memory address
else if (opcode == R_Type) case (IR[5:0]) //case for the various R-type instructions
32: ALUOut= A + B; //add operation
default: ALUOut= A; //other R-type operations: subtract, SLT, etc.
endcase
else if (opcode == BEQ) begin
if (A==B) PC <= ALUOut; // branch taken--update PC
state = 1;
end
else if (opcode == J) begin
PC = {PC[31:28], IR[25:0],2'b00}; // the jump target PC
state = 1;
end //Jumps
else ; // other opcodes or exception for undefined instruction would go here
end
```

```
// Part 4
4: begin
if (opcode == R_Type) begin //ALU Operation
Regs[IR[15:11]] <= ALUOut; // write the result
state =1;
end //R-type finishes
else if (opcode == LW) begin // load instruction
MDR <= Memory[ALUOut>>2]; // read the memory
state =5; // next state
end
else if (opcode == SW) begin
Memory[ALUOut>>2] <= B; // write the memory
state = 1; // return to state 1
end //store finishes
else ; // other instructions go here
end

5: begin // LW is the only instruction still in execution
Regs[IR[20:16]] = MDR; // write the MDR to the register
state = 1;
end //complete a LW instruction
endcase
end
endmodule
```