Practical Coursework

27/11/2020

Hardware Realisation of a Computer System - 3002CEM

Paul Johannes Aru

Table of Contents

Introduction

This report demonstrates the understanding and knowledge of microprocessor design alongside Hardware Description Language programming skills. In specific, the paper will focus on adding instructions to a single-cycle MIPS processor implemented in SystemVerilog. In order to verify the proposed solution, the document will also detail the steps taken to modify the assembly and machine code of the test program.

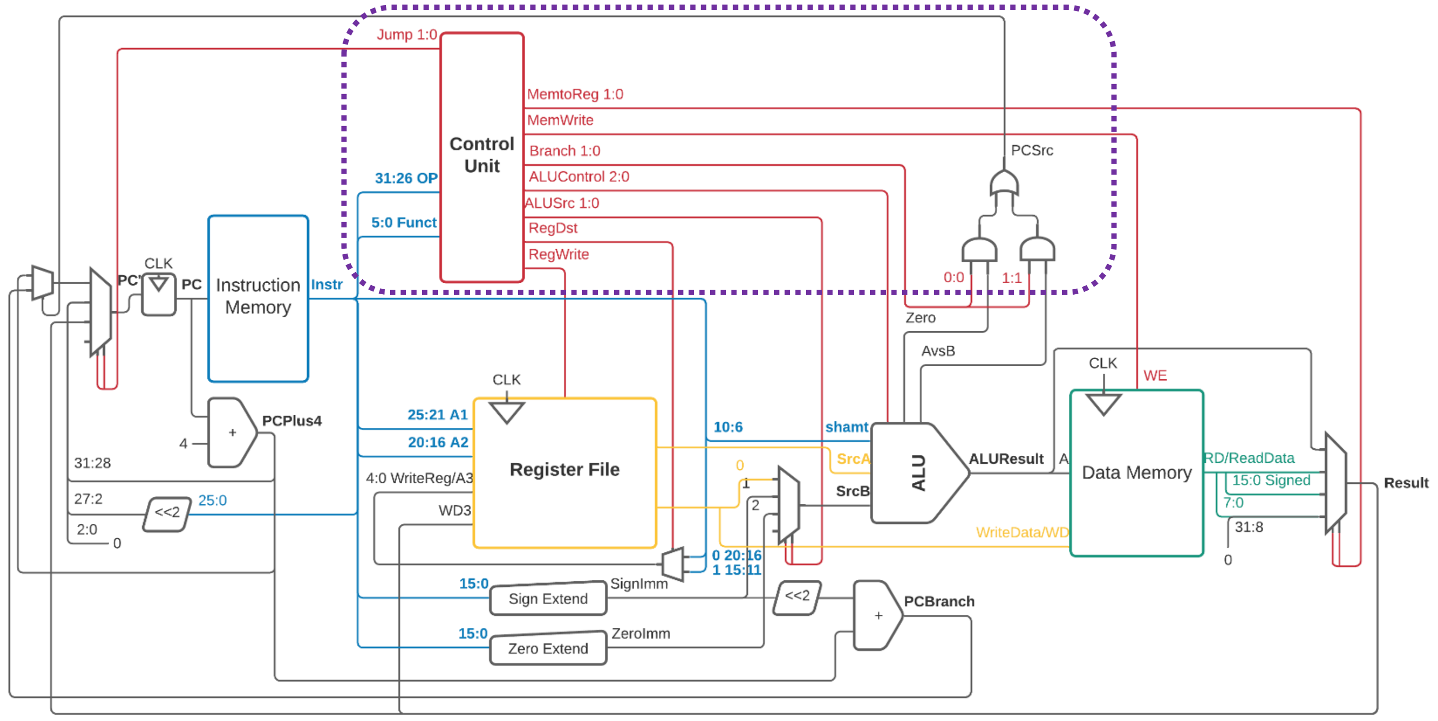
Microprocessor without Interlocked Pipelined Stages, or MIPS for short, is one of the earliest RISC Instruction Set Architecture. Like the name states, the key to a Reduced Instruction Set Computer, like ones with MIPS, is the smaller set of instructions. In total, Microprocessor without Interlocked Pipelined Stages has approximately 111 instructions (Chen et al., 2000). The following report starts off with a processor that supports 10 MIPS instructions and extends that to a total of 17 instructions.

SystemVerilog is a Hardware Description (and Verification) Language that extends the older Verilog language. Since the first standardisation in 2005, SystemVerilog has been adopted by both Intel (previously Altera) and Xilinx (IEEE, 2005). The rationale for preferring SystemVerilog over VHDL, not to mention Verilog, for our project lies within its simplicity. For example, importing a text file that contains our test program instructions would be considerably more complex in other languages.

Designing a Microprocessor

Processor Block Diagram

The following figure illustrates the entirety of the proposed single-cycle MIPS processor alongside two external memories. The design can be broken down to four main components. First of all, we have the Controller and Datapath that make up the processor. The Controller, that is mainly made up of *Main Decoder* and *ALU Decoder* can be seen in a purple dotted line below. Like the component names suggest, the controller handles the decoding of instructions. The Datapath contains everything else that a CPU requires. This includes, but is not limited to, the *register* and *Arithmetic Logic Unit*. In addition to the two processor components, we can also see an Instruction Memory and a Data Memory. The instruction memory stores our test program and the data memory can store data for our program.



The subsequent sections will elaborate on the functionality of the Control Unit and ALU.

Control Unit

The Control Unit is made up of the Main Decoder and ALU Decoder. Both of these components are essentially multiplexers that decide how the processor should proceed depending on the operation code and/or function extension of an instruction.

The table below is an overview of how the Main Decoder will interact with the 17 MIPS instructions that our processor supports. The rows that are highlighted in a shade of green illustrate the operations that were not supported by the provided example. The column headers that are highlighted in yellow are Control Unit outputs that had to be extended in order to support the 7 additional instructions.

While `xori` and `andi` are not R-type instructions, the proposed solution has extended the ALU Decoder to support Immediate Functions. In contrast, `jr` that is an R-Type instruction differs in its operation significantly from other R-Type instructions and as such is treated as a separate instruction by the Main Decoder. Lastly, it should be mentioned that while the table has a dedicated row for `srl`, the Main Decoder sees (and treats) it as a regular R-Type instruction.

Instruction	OP 5:0	RegWrite	RegDst	ALUSrc 1:0	Branch 1:0	MemWrite	MemToReg 1:0	Jump 1:0	ALUOP 1:0
R-Type	000000	1	1	00	00	0	00	00	10
lw	100011	1	0	01	00	0	01	00	00
sw	101011	0	X	01	00	1	XX	00	00
beq	000100	0	X	00	01	0	XX	00	01
addi	001000	1	0	01	00	0	00	00	00
j	000010	0	X	XX	XX	0	XX	01	XX
bgtz	000111	0	X	00	10	0	XX	00	00
lh	100001	1	0	01	00	0	10	00	00
xori (*Not R-type*)	001110	1	0	10	00	0	00	00	11
lbu	100100	1	0	01	00	0	11	00	00
andi (*Not R-type*)	001100	1	0	10	00	0	00	00	11
srl (*R-type*)	000000	1	1	00	00	0	00	00	10
jr (*Special R-type*)	000000	1	X	00	00	0	01	10	00

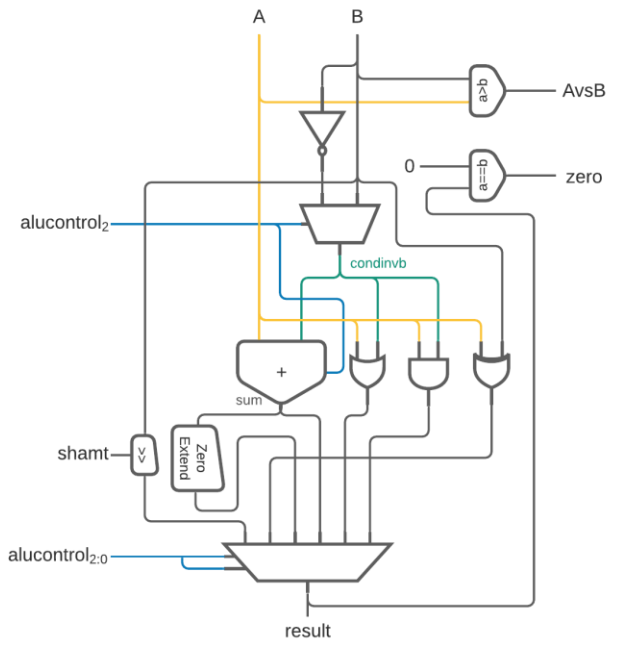
This next table illustrates the operation of the ALU Decoder. Once again, the rows highlighted in green indicate the newly supported instructions. This time the column header that is highlighted in yellow marks an additional input to the ALU Decoder.

In order for the ALU Decoder to be able to identify immediate functions, we take the last 3 bits of the operation code, as they hint which instruction we are dealing with. Since the decoder already has a control signal for the `and` logical operation, we could simply reuse it for `andi`.

Instruction	ALUOP $_{1:0}$	Function (Instr$_{5:0}$)	Immediate Function (Instr$_{28:26}$)	ALUControl $_{2:0}$
add	00	XXXXXX	XXX	010
subtract	01	XXXXXX	XXX	110
add (func.)	10	100000	XXX	010
subtract (func.)	10	100010	XXX	110
and (func.)	10	100100	XXX	000
or (func.)	10	100101	XXX	001
SLT (func.)	10	101010	XXX	111
SRL (func.)	10	000010	XXX	011
andi (im. func.)	11	XXXXXX	100	000
xori (im. func.)	11	XXXXXX	110	101

Arithmetic Logic Unit

The figure on the left offers a more detailed overview of the proposed <u>ALU</u> module. Aside from the addition of new operations, there is also a new output called *AvsB*. When we take a look at the earlier overview block diagram, we can see that this serves the `bgtz` instruction.



Operation	ALUControl $_{2:0}$
$\boldsymbol{A}$ AND $\boldsymbol{B}$	000
$\boldsymbol{A}$ OR $\boldsymbol{B}$	001
$\boldsymbol{A}+\boldsymbol{B}$	010
$\boldsymbol{B} \gg \boldsymbol{shamt}$	011
$\boldsymbol{A}$ AND $\overline{\boldsymbol{B}}$	100
$\boldsymbol{A}$ XOR $\boldsymbol{B}$	101
$\boldsymbol{A}-\boldsymbol{B}$	110
SLT	111

Near the end of the ALU, we have a multiplexer that based on the *ALUControl* signal decides what operation the *result* should be. The rows in green indicate new ALU operations and the row in orange depicts the only *ALUControl* signal that the ALU Decoder does not call and is available for modification.

While the ALU design would support A AND $\bar{B}$ & A OR $\bar{B}$; the proposed processor does not support instructions that would call on such operations. As a result, a decision was made to not extend the *ALUControl* bus at this time. Instead, the expected *ALUControl* signals would be used for other operations like *XOR*.

Coding and simulation

New Code Modules

For the instructions `xori` and `andi`, we need to create a Zero Extend module. The module will take the 16-bit immediate value from an instruction and extend it with zeroes to create a 32-bit item.

```
module zeroext(input logic [15:0] a,
               output logic [31:0] y);
    assign y = {16'b0000000000000000, a};
endmodule
```

Since a majority of the added instructions required extending 2-input multiplexers to 4-input multiplexers, we also had to create a new module to handle that. When compared to the old 2-input multiplexers, these also require 2 select-inputs.

```
module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [WIDTH-1:0] y);
always_comb
    case (s)
        2'b00: y <= d0;
        2'b01: y <= d1;
        2'b10: y <= d2;
        2'b11: y <= d3;
        default: y <= d0;
    endcase
endmodule
```

Noteworthy Code Modifications

The first important change that you will come across in our SystemVerilog code is regarding the branch decision logic inside the *controller* module. This alteration enables the `bgtz` instruction to call a branch.

```
assign pcsrc = (branch[1] & AvsB) | (branch[0] & zero);
```

The next interesting modification can be found inside the *maindec* module. In specific, we will be taking a look at the *case* statement. Aside from having to extend the 9-bit output controls to 13-bits, there are two notable changes. As mentioned earlier in the design stage, `jr` will not be treated like a regular *R-Type* instruction due to the complexity of its operation.

To ensure that it would not happen, we have added an *if* function that also checks the function of an *R-Type* operation in the Main Decoder. The second alteration is the addition of new operations and their decoded controls.

```
case(op)
  6'b000000: begin
    if (funct == 5'b001000)
        controls <= 13'b0000000001000; // JR
    else
        controls <= 13'b1100000000010; // RTYPE
    end
  6'b100011: controls <= 13'b1001000010000; // LW
  6'b101011: controls <= 13'b0001001000000; // SW
  6'b000100: controls <= 13'b0000010000001; // BEQ
  6'b001000: controls <= 13'b1001000000000; // ADDI
  6'b000010: controls <= 13'b0000000000100; // J
  6'b000111: controls <= 13'b0000100000000; // BGTZ
  6'b100001: controls <= 13'b1001000100000; // LH
  6'b001110: controls <= 13'b1010000000011; // XORI
  6'b100100: controls <= 13'b1001000110000; // LBU
  6'b001100: controls <= 13'b1010000000011; // ANDI
  default:   controls <= 13'bxxxxxxxxxxxxx; // illegal op
endcase
```

Moving along to the *aludec* module, we can find the ALU Decoder *case* statement. In order for the decoder to support the immediate functions, we have extended the *aluop case* statement with another *case* statement for them.

```
case(aluop)
  2'b00: alucontrol <= 3'b010;  // add (for lw/sw/addi)
  2'b01: alucontrol <= 3'b110;  // sub (for beq)
  2'b10: case(funct)           // Functions (R-type instructions)
      6'b100000: alucontrol <= 3'b010; // add
      6'b100010: alucontrol <= 3'b110; // sub
      6'b100100: alucontrol <= 3'b000; // and
      6'b100101: alucontrol <= 3'b001; // or
      6'b101010: alucontrol <= 3'b111; // slt
      6'b000010: alucontrol <= 3'b011; // srl
      default:   alucontrol <= 3'bxxx; // ???
    endcase
  2'b11: case(imFunct)          // Immediate Functions
      3'b100:   alucontrol <= 3'b000; // and immediate
      3'b110:   alucontrol <= 3'b101; // xor immediate
      default:  alucontrol <= 3'bxxx; // ???
    endcase
  default: alucontrol <= 3'bxxx; // ???
endcase
```

The *datapath* module contains several changes, as might be expected. Under next Program Counter logic, we have switched the *pcmux mux2* to a *mux4* in order to add the option of using *result* as our next instruction address. This is required to introduce `jr` operation support. One last thing to note about this line of code is that since a *mux4* expects 4 inputs, we have had to create an empty *logic* for the last input option.

```
mux4 #(32)  pcmux(pcnextbr, {pcplus4[31:28],instr[25:0],2'b00},
result, EMPTY, jump, pcnext);
```

Under register file logic in the *datapath* module, we find the values for `lh` and `lbu` and once again extend *resmux* to include these as result options.

```
  assign readHalf = 32'(signed'(readdata[15:0]));
  assign readByte = {24'b000000000000000000000000, readdata[7:0]};
  mux4 #(32)  resmux(aluout, readdata, readHalf, readByte, memtoreg,
result);
```

The last significant change in the *datapath* module is regarding the source b selector. We create a zero-extended immediate value and add that as an option to the *srcbmux*. As a result, this multiplexer will have to be changed to a *mux4* as well.

```
  zeroext     ze(instr[15:0], zeroimm);

  // ALU logic
  mux4 #(32)  srcbmux(writedata, signimm, zeroimm, EMPTY, alusrc,
srcb);
```

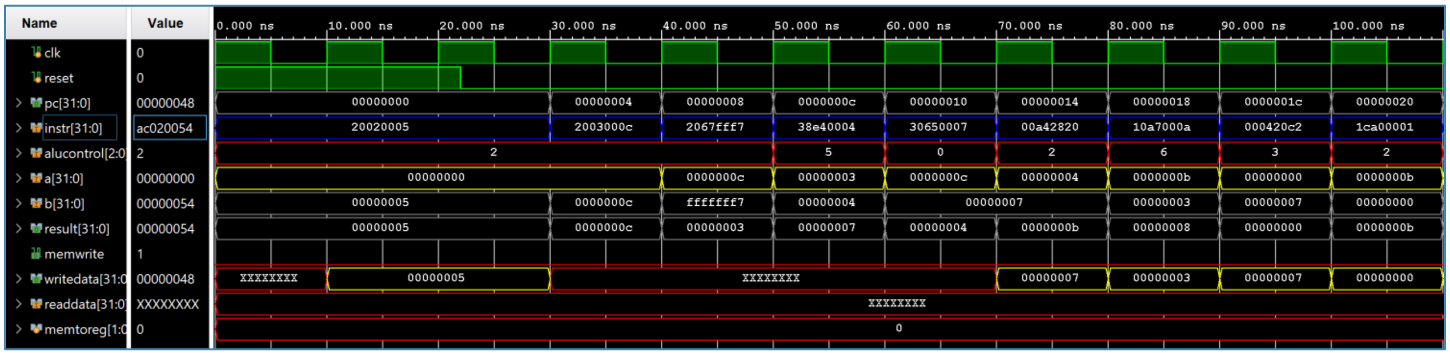
This brings us to the last modified module and changes. In the *alu* module, the main noticeable change takes place with the *result* selecter. The *case* switch has been replaced with a *casez* and additional operations have been added. In addition, we will also find the *AvsB* check in this module, used by the bgtz instruction.

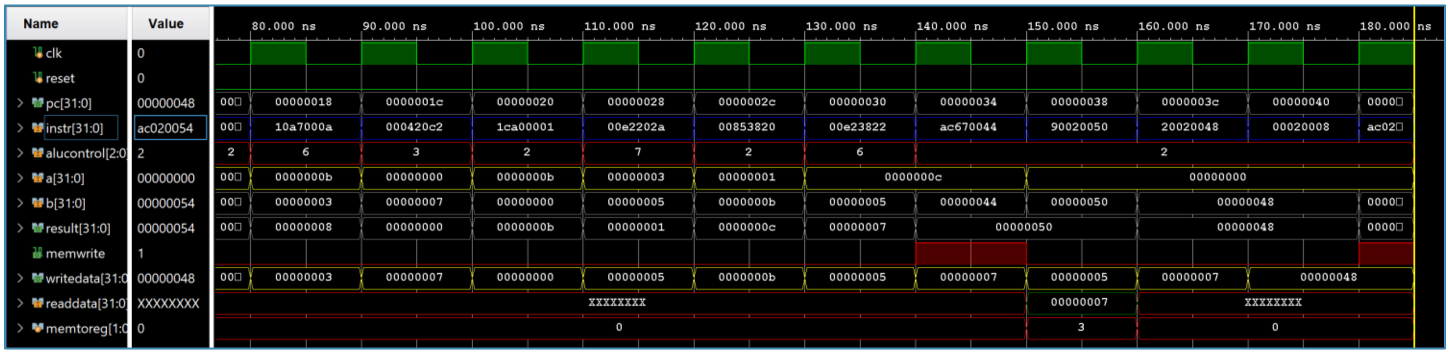
```
    casez (alucontrol)
      3'bz00: result <= a & condinvb;
      3'b001: result <= a | condinvb;
      3'b101: result <= a ^ b;
      3'bz10: result <= sum;
      3'b011: result <= b >> shamt;
      3'b111: result <= sum[31];
    endcase

  assign zero = (result == 32'b0);
  assign AvsB = a>b;
```

Simulating a Test Program

To ensure the functionality of the proposed MIPS processor, a custom test program was executed as a part of the testbench in *Xilinx Vivado*. The following table lays out the program instructions both in Assembly and Machine Code. The rows highlighted in green illustrate the instructions designed to verify our proposed MIPS processor. Below the table are captures of selected waveforms illustrating the successful execution of our test program.

#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# $2=0+5 *($2=5)*	0	20020005
	addi $3, $0, 12	# $3=0+12 *($3=12)*	4	2003000c
	addi $7, $3, -9	# $7=12+(-9) *($7=3)*	8	2067fff7
	xori $4, $7, 4	# $4=3|4 *($4=7)*	c	38e40004
	andi $5, $3, 7	# $5=12&7 *($5=4)*	10	30650007
	add $5, $5, $4	# $5=4+7 *($5=11)*	14	00a42820
	beq $5, $7, end	# 11==3?, ->end *(SKIP)*	18	10a7000a
	srl $4, $4, 3	# $4=7>>3 *($4=0)*	1c	000420c2
	bgtz $5, around	# 11>0?, ->around *(GO TO 28)*	20	1ca00001
	addi $5, $0, 0	# $5=0+0 *(SKIP)*	24	20050000
around:	slt $4, $7, $2	# $4=3<5 *($4=1)*	28	00e23822
	add $7, $4, $5	# $7=1+11 *($7=12)*	2c	00853820
	sub $7, $7, $2	# $7=12-5 *($7=7)*	30	00e23822
	sw $7, 68($3)	# [80]=7	34	ac670044
	lbu $2, 80($0)	# $2=[80] *($2=7)*	38	90020050
	addi $2, $0, 48	# $2=0+48 *($2=48)*	3c	20020048
	jr $2	# ->$2 *(GO TO 48)*	40	00020008
	addi $2, $0, 1	# $2=0+1 *(SKIP)*	44	20020001
end:	sw $2, 84($0)	# [84]=7	48	ac020054





References

Chen, C., Novick, G., & Shimano, K. (2000). *MIPS*. Stanford University. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html

Institute of Electrical and Electronics Engineers. (2005). *IEEE 1800-2005 - IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language*. https://standards.ieee.org/standard/1800-2005.html

Appendices

A – SystemVerilog Design (a.k.a. `mipssingle.sv`)

```
// 3002CEM - Hardware Realisation of a Computer System
// 2020 Coursework
// Single-cycle MIPS processor
// Paul Johannes Aru

module mips(input logic clk, reset, input logic [31:0] instr, readdata, output logic memwrite, output logic [31:0] pc,
aluout, writedata);
  logic       regdst, regwrite, pcsrc, zero, AvsB;
  logic [1:0] alusrc, memtoreg, jump;
  logic [2:0] alucontrol;

  controller c(zero, AvsB, instr[28:26], instr[31:26], instr[5:0], memwrite, pcsrc, regdst, regwrite, memtoreg, alusrc,
jump, alucontrol);
  datapath dp(clk, reset, pcsrc, regdst, regwrite, memtoreg, alusrc, jump, alucontrol, instr, readdata, zero, AvsB, pc,
aluout, writedata);
endmodule

module controller(input  logic zero, AvsB, input logic [2:0] imFunct, input  logic [5:0] op, funct, output logic
memwrite, pcsrc, regdst, regwrite, output logic [1:0] memtoreg, alusrc, jump, output logic [2:0] alucontrol);
    logic [1:0] aluop, branch;

    maindec md(op, funct, memwrite, regdst, regwrite, memtoreg, branch, alusrc, jump, aluop);
    aludec  ad(aluop, imFunct, funct, alucontrol);

    assign pcsrc = (branch[1] & AvsB) | (branch[0] & zero);
endmodule

module maindec(input logic [5:0] op, funct, output logic memwrite, regdst, regwrite, output logic [1:0] memtoreg,
branch, alusrc, jump, aluop);
    logic [12:0] controls;

    assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop} = controls;

    always_comb
        case(op)
            6'b000000: begin
                if (funct == 5'b001000)
                    controls <= 13'b0000000001000; // JR
                else
                    controls <= 13'b1100000000010; // RTYPE
            end
            6'b100011: controls <= 13'b1001000010000; // LW
            6'b101011: controls <= 13'b0001001000000; // SW
            6'b000100: controls <= 13'b0000010000001; // BEQ
            6'b001000: controls <= 13'b1001000000000; // ADDI
            6'b000010: controls <= 13'b0000000000100; // J
            6'b000111: controls <= 13'b0000100000000; // BGTZ
            6'b100001: controls <= 13'b1001000100000; // LH
            6'b001110: controls <= 13'b1010000000011; // XORI
            6'b100100: controls <= 13'b1001000110000; // LBU
            6'b001100: controls <= 13'b1010000000011; // ANDI
            default:   controls <= 13'bxxxxxxxxxxxxx; // illegal op
        endcase
endmodule
```

```
module aludec(input logic [1:0] aluop, input logic [2:0] imFunct, input logic [5:0] funct, output logic [2:0]
alucontrol);

    always_comb
        case(aluop)
            2'b00: alucontrol <= 3'b010;  // add (for lw/sw/addi)
            2'b01: alucontrol <= 3'b110;  // sub (for beq)
            2'b10: case(funct)          // Functions (R-type instructions)
                6'b100000: alucontrol <= 3'b010; // add
                6'b100010: alucontrol <= 3'b110; // sub
                6'b100100: alucontrol <= 3'b000; // and
                6'b100101: alucontrol <= 3'b001; // or
                6'b101010: alucontrol <= 3'b111; // slt
                6'b000010: alucontrol <= 3'b011; // srl
                default:   alucontrol <= 3'bxxx; // ???
            endcase
            2'b11: case(imFunct)          // Immediate Functions
                3'b100:   alucontrol <= 3'b000; // and immediate
                3'b110:   alucontrol <= 3'b101; // xor immediate
                default:  alucontrol <= 3'bxxx; // ???
            endcase
            default: alucontrol <= 3'bxxx; // ???
        endcase
endmodule

module datapath(input logic clk, reset, pcsrc, regdst, regwrite, input logic [1:0]  memtoreg, alusrc, jump, input logic
[2:0] alucontrol, input logic [31:0] instr, readdata, output logic zero, AvsB, output logic [31:0] pc, aluout,
writedata);
    logic [4:0]  writereg;
    logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
    logic [31:0] signimm, signimmsh, zeroimm;
    logic [31:0] srca, srcb;
    logic [31:0] result, readHalf, readByte;
    logic [31:0] EMPTY;

    // next PC logic
    flopr #(32) pcreg(clk, reset, pcnext, pc);
    adder       pcadd1(pc, 32'b100, pcplus4);
    sl2         immsh(signimm, signimmsh);
    adder       pcadd2(pcplus4, signimmsh, pcbranch);
    mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
    mux4 #(32)  pcmux(pcnextbr, {pcplus4[31:28],instr[25:0],2'b00}, result, EMPTY, jump, pcnext);

    // register file logic
    regfile     rf(clk, regwrite, instr[25:21], instr[20:16], writereg, result, srca, writedata);
    mux2 #(5)   wrmux(instr[20:16], instr[15:11], regdst, writereg);
    assign readHalf = 32'(signed'(readdata[15:0])); //Inspiration: https://stackoverflow.com/questions/35763633/how-do-
i-sign-extend-in-systemverilog
    assign readByte = {24'b000000000000000000000000, readdata[7:0]};
    mux4 #(32)  resmux(aluout, readdata, readHalf, readByte, memtoreg, result);
    signext     se(instr[15:0], signimm);
    zeroext     ze(instr[15:0], zeroimm);

    // ALU logic
    mux4 #(32)  srcbmux(writedata, signimm, zeroimm, EMPTY, alusrc, srcb);
    alu         alu(alucontrol, instr[10:6], srca, srcb, zero, AvsB, aluout);
endmodule
```

```
module regfile(input logic clk, we3, input logic [4:0] ra1, ra2, wa3, input logic [31:0] wd3, output logic [31:0] rd1,
rd2);
    logic [31:0] rf[31:0];

    // three ported register file
    // read two ports combinationally
    // write third port on rising edge of clk
    // register 0 hardwired to 0
    // note: for pipelined processor, write third port
    // on falling edge of clk

    always_ff @(posedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

module adder(input  logic [31:0] a, b, output logic [31:0] y);

    assign y = a + b;
endmodule

module sl2(input  logic [31:0] a, output logic [31:0] y);

    // shift left by 2
    assign y = {a[29:0], 2'b00};
endmodule

module signext(input  logic [15:0] a, output logic [31:0] y);

    assign y = {{16{a[15]}}, a};
endmodule

module zeroext(input logic [15:0] a, output logic [31:0] y);

    assign y = {16'b0000000000000000, a};
endmodule

module flopr #(parameter WIDTH = 8) (input  logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8) (input  logic [WIDTH-1:0] d0, d1, input logic s, output logic [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

```
module mux4 #(parameter WIDTH = 8) (input logic [WIDTH-1:0] d0, d1, d2, d3, input logic [1:0] s, output logic [WIDTH-
1:0] y);

always_comb
    case (s)
        2'b00: y <= d0;
        2'b01: y <= d1;
        2'b10: y <= d2;
        2'b11: y <= d3;
        default: y <= d0;
    endcase
endmodule

module alu(input logic [2:0] alucontrol, input logic [4:0] shamt, input logic [31:0] a, b, output logic zero, AvsB,
output logic [31:0] result);
    logic [31:0] condinvb, sum;

    assign condinvb = alucontrol[2] ? ~b : b;
    assign sum = a + condinvb + alucontrol[2];

    always_comb
        casez (alucontrol)
            3'bz00: result <= a & condinvb;
            3'b001: result <= a | condinvb;
            3'b101: result <= a ^ b;
            3'bz10: result <= sum;
            3'b011: result <= b >> shamt;
            3'b111: result <= sum[31];
        endcase

    assign zero = (result == 32'b0);
    assign AvsB = a>b;
endmodule
```

mipssingle.sv

B – SystemVerilog Simulation (a.k.a. `mipstest.sv`)

```
// 3002CEM - Hardware Realisation of a Computer System
// 2020 Coursework
// Testbench for MIPS processor
// Paul Johannes Aru

module testbench();
    logic        clk, reset, memwrite;
    logic [31:0] writedata, dataadr;

    // instantiate device to be tested
    top dut(clk, reset, writedata, dataadr, memwrite);

    // initialize test
    initial
        begin
            reset <= 1; # 22; reset <= 0;
        end

    // generate clock to sequence tests
    always
        begin
            clk <= 1; # 5; clk <= 0; # 5;
        end

    // check results
    always @(negedge clk)
        begin
            if(memwrite) begin
                if(dataadr === 84 & writedata === 72) begin
                    $display("Simulation succeeded");
                    $stop;
                end else if (dataadr !== 80) begin
                    $display("Simulation failed");
                    $stop;
                end
            end
        end
endmodule

module top(input logic clk, reset, output logic [31:0] writedata, dataadr, output logic memwrite);
    logic [31:0] pc, instr, readdata;

    // instantiate processor and memories
    mips mips(clk, reset, instr, readdata, memwrite, pc, dataadr, writedata);
    imem imem(pc[7:2], instr);
    dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule

module dmem(input logic clk, we, input logic [31:0] a, wd, output logic [31:0] rd);
    logic [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // word aligned

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule

module imem(input  logic [5:0] a, output logic [31:0] rd);
    logic [31:0] RAM[63:0];

    initial
        $readmemh("C:/Vivado_Projects/MIPS/memfile.dat",RAM);

    assign rd = RAM[a]; // word aligned
endmodule
```

mipstest.sv

C – Test Program (a.k.a. `memfile.dat`*)*

```
20020005
2003000c
2067fff7
38e40004
30650007
00a42820
10a7000a
000420c2
1ca00001
20050000
00e2202a
00853820
00e23822
ac670044
90020050
20020048
00020008
20020001
ac020054
```

memfile.dat