CS 21 Project 2 Documentation

Carlos Sandoval

April 2020

Contents

1 Introduction

This document details the design and implementation of a single cycle processor capable of the instructions described below.
The format of this documentation will be as follows per instruction:

Instruction - Modifications

The modifications section will contain snippets and descriptions of the modules that were altered, while the datapath section will contain the datapath of the instruction

Because the code of modules is the code that is capable of all 7 instructions, changes will be visible per module that might not be important to the current instruction, thus there will be a summary at the end per module that explains the total changes to each module.

The changes per module that are important will be in **boldface font** for easier visibility

2 andi

2.1 Functionality

R[rt] = R[rs] ZeroExtImm
The andi instruction sets R[rt] to the result of R[rs] and a Zero extended 16 bit immediate, to extend our datapath to do this, signext was extended to handle zero extending. Decoding logic was also added to aludec and maindec.

2.2 Modifications

2.2.1 signext

Code Block 1 signext module

```
module signext(input  logic [15:0] a, input logic [5:0] b,
               output logic [31:0] y);
always_comb
    case(b)
        6'b001100:  y <= 160, a; // andi
        default:   y <= {{16{a[15]}}, a};
    endcase
endmodule
```

As can be seen in the above snippet, the module was modified such that it expects the opcode of the instruction, it then switches the output (y) to its

correct value if the opcode of andi is detected. This "correct" value of y is similar to the original extention except that it always extends with 0, it doesnt sign extend like the original module.

2.2.2 aludec

Code Block 2 aludec module

```
                ////////////
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [3:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      2'b01: alucontrol <= 4'b1010;  // sub (for beq)
      2'b11:  alucontrol <= 4'b0000; // and (for andi)
      default: case(funct)          // R-type instructions
          #decoding for R-type instructions#
        endcase
    endcase
endmodule
```

As seen above, the alucontrol signal was extended to 4 bits, this will be explained later in the srl section, the relevant change for the andi instruction is the added case (aluop = 11) that invokes the already present functionality of ALU to do the "and" operation

2.2.3 maindec

Code Block 3 maindec module

```
                `timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

logic [8:0] controls;
assign {regwrite, regdst, alusrc, branch, memwrite,
        memtoreg, jump, aluop} = controls;

always_comb
    case(op)
    #decoding logic for other instructions#
    6'b001100:  controls <= 9'b101000011; // ANDI
    endcase
endmodule
```

The only addition to maindec was a line for decoding the opcode the andi instruction, asserting the correct control signals.
The control signals and the explanations are as follows:

1. regwrite = 1
   The andi instruction stores the result in a register.
2. regdst = 0
   This selects instr[20:16] as address of the register to be written to, this is correct as this is an I-type instruction.
3. alusrc = 1
   This selects the output of the signext module as the correct 2nd source of the ALU.
4. branch = 0
   This is not asserted as andi is not a branch instruction
5. memwrite = 0
   This is not asserted as andi does not write to memory
6. memtoreg = 0
   This Is not asserted as andi does not write from memory
7. jump = 0
   This is not asserted as andi is not a jump instruction

8. aluop = 11
   This is set to induce the and operation from the ALU as shown earlier in the aludec snippet

2.2.4 datapath

Code Block 4 datapath module

```
                ////////////
                `timescale 1ns / 1ps
                module datapath(input  logic        clk, reset,
                                input  logic
memtoreg, pcsrc,
                                input  logic        alusrc, regdst,
                                input  logic        regwrite, jump,
                                input  logic [3:0]  alucontrol,
                                output logic        zero,
                                output logic [31:0] pc,
                                input  logic [31:0] instr,
                                output logic [31:0] aluout, writedata,
                                input  logic [31:0] readdata);
                  logic jrcontrol1, jrcontrol2;
                  logic [4:0]  writereg;
                  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch,jrnext;
                  logic [31:0] signimm, signimmsh;
                  logic [31:0] srca, srcb, srcd;
                  logic [31:0] result;
                  // next PC logic
                  #other parts of the datapath#
                  signext     se(instr[15:0],instr[31:26], signimm);
                  #other parts of the datapath#
                endmodule
```

The only change to the datapath module is the addition to the signext parameters. The opcode (instr[31:26]) is passed to the module to facilitate its new functionality as described previously

3 bne

3.1 Functionality

if(R[rs]!=R[rt]):
   PC=PC+4+BranchAddr

The bne instruction branches to PC + 4 + 16 bit immediate if R[rs] != R[rt]

To implement this, we remember that beq is already an existing part of the instruction set. Going over the modules it can be noticed that the "zero" output of the ALU is only used for beq. Therefore, it can be also be interpreted as a branch or not signal if beq is the only instruction that uses it.

Thus, It was decided that zero would be reappropriated as a branch signal for the remainder of this project. Now that the zero signal controls whether or not the branch is taken (assuming the correct control signals are also asserted), we can add cases to the already existing logic inside the alu.

This means that zero will be set to 1 not only when the result is 0, but in the case of the bne instruction, also when the result isnt 0. This also means that the opcode needs to also be passed to the ALU.

3.2 Modifications

3.2.1 ALU

Code Block 5 alu module

```
            ////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b, d,
           input  logic [3:0]  alucontrol,
           input logic [5:0] opcode,
           input logic [4:0] shamt,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, diffshift, diff;
  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];
  always_comb
    case (alucontrol[2:0])
      #other alu decoding l
      3'b010: result = sum;
      #other alu decoding logic#
    endcase
  always_comb
    case(opcode)
      6'b000101:  zero <= (result != 32'b0); // bne
      default:   zero <= (result == 32'b0); // beq
    endcase
endmodule
```

As seen in the snippet, the ALU was modified to accomodate the opcode, this allows for decoding logic that determines what conditions will assert zero.

It can also be seen that alucontrol was extended to 4 bits, this will be discussed later in the srl section and is not important for bne.

3.2.2 maindec

Code Block 6 maindec module

```
                ‘timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

logic [8:0] controls;
assign {regwrite, regdst, alusrc, branch, memwrite,
        memtoreg, jump, aluop} = controls;

always_comb
    case(op)
    #decoding logic for other instructions#
     6'b000101:  controls <= 9'b000100001; // BNE
    endcase
endmodule
```

As can be seen, the control signals for bne are exactly the same as beq
The control signals and the explanations are as follows:

1. regwrite = 0
   bne doesn't write to a register
2. regdst = 0
   bne doesn't write to a register
3. alusrc = 0
   This selects the register instead of the sign extended immediate as the correct source of the ALU
4. branch = 1
   This is asserted as bne is a branch instruction
5. memwrite = 0
   This is not asserted as bne does not write to memory
6. memtoreg = 0
   This Is not asserted as bne does not write from memory
7. jump = 0
   This is not asserted as bne is not a jump instruction
8. aluop = 10
   This is set to induce the sub operation from the ALU, same as the other branch instructions

4 jr

4.1 Functionality

PC=R[rs]

The jr instruction causes the program counter to take the value at R[rs] instead of pc + 4

The approach taken to extend the datapath to accomodate this instruction was to modify the flopr module to check if the instruction is jr. If this is the case, it will set the pc to the corresponding register value instead of pc+4.

4.2 Modifications

4.2.1 flopr

Code Block 7 flopr module

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

                 logic [31:0] pcf = d;
                 always_comb
                 case(instr[31:26],instr[5:0])
                 12'b000000001000:  pcf <= srca;
                 default:   pcf <= d;
                 endcase
    always_ff (posedge clk, posedge reset)if (reset) q <= 0;else q <=
pcf;endmodule
```

As can be seen in the code block above, instr is passed to the module so that it can use the opcode and funct fields to determine if the instruction is jr.

A new logic was created to handle this called pcf that either retains d or is set to srca (R[rs]) in the case of the instruction being jr.

The logic that sets the state of q at posedge of clk remains the same, although with pcf instead of d.

4.2.2 datapath

The only change to the datapath module is the addition to the flopr parameters. The instr is passed to the module to facilitate its new functionality as described

Code Block 8 datapath module

```
                ////////////
                `timescale 1ns / 1ps
                module datapath(input  logic        clk, reset,
                                input  logic
memtoreg, pcsrc,
                                input  logic        alusrc, regdst,
                                input  logic        regwrite, jump,
                                input  logic [3:0]  alucontrol,
                                output logic        zero,
                                output logic [31:0] pc,
                                input  logic [31:0] instr,
                                output logic [31:0] aluout, writedata,
                                input  logic [31:0] readdata);
                  logic jrcontrol1, jrcontrol2;
                  logic [4:0]  writereg;
                  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch,jrnext;
                  logic [31:0] signimm, signimmsh;
                  logic [31:0] srca, srcb, srcd;
                  logic [31:0] result;
                  // next PC logic
                  #other parts of the datapath#
                  flopr (32) pcreg(clk, reset, pcnext,instr,srca, pc);;
                  #other parts of the datapath#
                endmodule
```

previously

5 srl

5.1 Functionality

R[rd] = R[rt] >> shamt

The srl is an R type instruction that sets the register rd to R[rt] extended to the right by the value contained in shamt

The datapath was extended to accomodate this instruction by adding decoding logic to aludec and extending alucontrol by one bit to accomodate a new operation, shift by shamt. This change also means that shamt will be passed to the ALU as well. To ensure that the already pre existing instructions arent affected, their decoding logic is updated to use the extended alucontrol.

Code Block 9 aludec module

```
                ////////////
                `timescale 1ns / 1ps
                module aludec(input  logic [5:0] funct,
                              input  logic [1:0] aluop,
                              output logic [3:0] alucontrol);

                always_comb
                    case(aluop)
                    2'b00: alucontrol <= 4'b0010;
// add (for lw/sw/addi)
                    2'b01: alucontrol <= 4'b1010;  // sub (for beq)
                    2'b11: alucontrol <= 4'b0000;
// and (for andi)
                    default: case(funct)
// R-type instructions
                        #other aludec decoding logic#
                        6'b000010:    alucontrol  <= 4'b0111; //srl
                        default:   alucontrol <= 4'bxxxx; // ???
                        endcase
                    endcase
                endmodule
```

5.2 Modifications

5.2.1 aludec

As can be seen in the code block above, the aludec decoding logic was extended to include the srl instruction, the alucontrol is set to 111 and will be explained in the next section.

5.2.2 ALU

Code Block 10 ALU module

```
////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [3:0]  alucontrol,
           input logic [5:0] opcode,
           input logic [4:0] shamt,
           output logic [31:0] result,
           output logic        zero);
  logic [31:0] condinvb, sum;
  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];
  always_comb
    case (alucontrol[2:0])
      #other alu logic#
      3'b111:  result = b >> shamt;
    endcase
  always_comb
    case(opcode)
      6'b000101: zero <= (result != 32'b0); // bne
      default:   zero <= (result == 32'b0); // beq
    endcase
endmodule
```

As can be seen above, The ALU logic was extended to accept a 4 bit alucontrol. Shamt was also included in the parameters so the ALU would know by what value to shift right by in case that was the instruction.

The first bit was retained to determine if the operation is subtraction, the remaining three were used to determine the already existing functionality and the new shift right by shamt function.

5.2.3 datapath

Code Block 11 datapath module

```
                //////////////
                `timescale 1ns / 1ps
                module datapath(input  logic        clk, reset,
                                input  logic
memtoreg, pcsrc,
                                input  logic        alusrc, regdst,
                                input  logic        regwrite, jump,
                                input logic [3:0] alucontrol,
                                output logic        zero,
                                output logic [31:0] pc,
                                input  logic [31:0] instr,
                                output logic [31:0] aluout, writedata,
                                input  logic [31:0] readdata);
                  logic jrcontrol1, jrcontrol2;
                  logic [4:0]  writereg;
                  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch,jrnext;
                  logic [31:0] signimm, signimmsh;
                  logic [31:0] srca, srcb;
                  logic [31:0] result;
                  // next PC logic
                  #other parts of the datapath#
                   alu alu(srca, srcb,srcd, alucontrol,instr[31:26],instr[10:6],
aluout, zero);;
                  #other parts of the datapath#
                endmodule
```

The changes to the datapath module are the additions of shamt to ALU's parameters and the extension of alucontrol to 4 bits The instr is passed to the module to facilitate its new functionality as described previously

5.2.4 controller

Code Block 12 controller module

```
          `timescale 1ns / 1ps
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [3:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

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

  assign pcsrc = branch & zero;
endmodule
```

5.2.5 mips

Code Block 13 mips module

```
‘timescale 1ns / 1ps
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata);

  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero;
  logic [3:0] alucontrol;

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

Similar to the controller module, the only change to the mips module was the adjustment of alucontrols bit width.

6 blt

6.1 Functionality

if(R[rs]< R[rt]):
    PC=PC+4+BranchAddr

The blt instruction branches to PC + 4 + 16 bit immediate if R[rs] < R[rt]

To implement this, we remember the changes we made to beq to implement bne. Recalling the earlier section, this includes using zero as our branch or not signal. We use the same control signals as the other two branch instructions, this means that ALU will subtract.

This can be used since the first bit of result will be 1 if ever the result is negative. In fact, we can just set zero to that bit and it will only assert when R[rs] ¡ R[rt].

6.2 Modifications

6.2.1 ALU

Code Block 14 alu module

```
        ////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b, d,
       input  logic [3:0]  alucontrol,
       input logic [5:0] opcode,
       input logic [4:0] shamt,
       output logic [31:0] result,
       output logic        zero);

logic [31:0] condinvb, sum, diffshift, diff;
assign condinvb = alucontrol[3] ? ~b : b;
assign sum = a + condinvb + alucontrol[3];
always_comb
case (alucontrol[2:0])
  #other alu decoding l
  3'b010: result = sum;
  #other alu decoding logic#
endcase
always_comb
case(opcode)
  6'b000001:  zero <= (result[31]); // blt
  6'b000101: zero <= (result != 32'b0); // bne
  default:   zero <= (result == 32'b0); // beq
endcase
endmodule
```

As seen in the snippet, the ALU was modified to accomodate the opcode, this allows for decoding logic that determines what conditions will assert zero.

As discussed previously, alucontrol was extended to 4 bits, but alucontrol for blt stays the same from bne, beq, and later on, bge. The added logic can be seen in the decoding logic for the opcode.

When the opcode for blt is detected, it sets zero to result[31].

This was decided because since the ALU was set to subtract, the first bit of result would be 1 if it is a negative number.

6.2.2 maindec

Code Block 15 maindec module

```
                ‘timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

logic [8:0] controls;
assign {regwrite, regdst, alusrc, branch, memwrite,
        memtoreg, jump, aluop} = controls;

always_comb
    case(op)
    #decoding logic for other instructions#
    6'b000001:  controls <= 9'b000100001; // BLT
    endcase
endmodule
```

As can be seen, the control signals for blt are exactly the same as the other branch instructions
The control signals and the explanations are as follows:

1. regwrite = 0
   blt doesn't write to a register
2. regdst = 0
   blt doesn't write to a register
3. alusrc = 0
   This selects the register instead of the sign extended immediate as the correct source of the ALU
4. branch = 1
   This is asserted as blt is a branch instruction
5. memwrite = 0
   This is not asserted as blt does not write to memory
6. memtoreg = 0
   This Is not asserted as blt does not write from memory
7. jump = 0
   This is not asserted as blt is not a jump instruction
8. aluop = 10
   This is set to induce the sub operation from the ALU, same as the other branch instructions

7 bge

7.1 Functionality

if(R[rs]>= R[rt]):
    PC=PC+4+BranchAddr

The bge instruction branches to PC + 4 + 16 bit immediate if R[rs] >= R[rt]

To implement this, we remember the changes we made to beq to implement bne. Recalling the earlier section, this includes using zero as our branch or not signal. We use the same control signals as the other two branch instructions, this means that ALU will subtract.

This can be used since the first bit of result will be 1 if ever the result is negative. In fact, we can just set zero to result[31] and it will only assert when R[rs] >= R[rt].

7.2 Modifications

7.2.1 ALU

Code Block 16 alu module

```
        ////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b, d,
       input  logic [3:0]  alucontrol,
       input logic [5:0] opcode,
       input logic [4:0] shamt,
       output logic [31:0] result,
       output logic        zero);

logic [31:0] condinvb, sum, diffshift, diff;
assign condinvb = alucontrol[3] ? ~b : b;
assign sum = a + condinvb + alucontrol[3];
always_comb
case (alucontrol[2:0])
  #other alu decoding l
  3'b010: result = sum;
  #other alu decoding logic#
endcase
always_comb
case(opcode)
  6'b100110:  zero <= ( result[31]); // bge
  6'b000001: zero <= (result[31]); // blt
  6'b000101: zero <= (result != 32'b0); // bne
  default:   zero <= (result == 32'b0); // beq
endcase
endmodule
```

As seen in the snippet, the ALU was modified to accomodate the opcode, this allows for decoding logic that determines what conditions will assert zero.

As discussed previously, alucontrol was extended to 4 bits, but alucontrol for bge stays the same from bne, beq, and blt. The added logic can be seen in the decoding logic for the opcode.

When the opcode for blt is detected, it sets zero to result[31].

This was decided because since the ALU was set to subtract, the first bit of result would be 1 if it is a negative number.

7.2.2 maindec

As can be seen, the control signals for bge are exactly the same as the other branch instructions
The control signals and the explanations are as follows:

Code Block 17 maindec module

```
                ‘timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

logic [8:0] controls;
assign {regwrite, regdst, alusrc, branch, memwrite,
        memtoreg, jump, aluop} = controls;

always_comb
    case(op)
    #decoding logic for other instructions#
     6'b100110:  controls <= 9'b000100001;// BGE
    endcase
endmodule
```

1. regwrite = 0
   bge doesn't write to a register

2. regdst = 0
   bge doesn't write to a register

3. alusrc = 0
   This selects the register instead of the sign extended immediate as the correct source of the ALU

4. branch = 1
   This is asserted as bge is a branch instruction

5. memwrite = 0
   This is not asserted as bge does not write to memory

6. memtoreg = 0
   This Is not asserted as bge does not write from memory

7. jump = 0
   This is not asserted as bge is not a jump instruction

8. aluop = 10
   This is set to induce the sub operation from the ALU, same as the other branch instructions

8 diffshift

8.1 Functionality

As described in the project specifications, diffshift stores 0xc0debabe in R[rd] if R[rs] ¿ R[rt], and stores R[rd] << R[rs]-R[rt] otherwise.

The approach taken was to take the value at rd from the regfile, pass it to the alu, add decoding logic in aludec, and add decoding logic in the ALU.

8.2 Modifications

8.2.1 regfile

Code Block 18 regfile module

```
                // regfile.v
// Register file for the single-cycle and multicycle processors

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2,rd3);

  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 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;\textbf{assign rd3 = rf[wa3];}wa3 is rdendmodule
```

The only changes to regfile were the addition of R[rd] as output logic as seen in the snippet.

8.2.2 ALU

The modifications to ALU start off with the addition of R[rd] as a parameter, this value is taken from the register because it is needed to execute diffshift. The diff logic is introduced as a copy of the subtraction operation, it is then used to determine the value that will be stored in rd if ever diffshift is the instruction. As will be seen later, a unique alucontrol signal was alotted for this instruction.

```
                ////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b, d,
           input  logic [3:0]  alucontrol,
           input logic [5:0] opcode,
           input logic [4:0] shamt,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, diffshift, diff;
  assign diff = a +  b + alucontrol[3];
  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];

  always_comb
        begin
        if (diff > 32)
            diffshift <=32'hc0debabe;
        else
            diffshift <= (d << diff);
        end
  always_comb
    case (alucontrol[2:0])
      #other alu logic#
      3'b100: result <= diffshift;
    endcase
  #branch alu logic#
endmodule
```

8.2.3 aludec

Code Block 19 aludec module

```
                ////////////
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [3:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      2'b01: alucontrol <= 4'b1010;  // sub (for beq/bne/blt/bge)
      2'b11: alucontrol <= 4'b0000;  // and (for andi)
      default: case(funct)          // R-type instructions
          #other aludec logic#
          6'b000100: alucontrol <= 4'b0100; //diffshift
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

When the funct of diffshift is detected alucontrol is set to a previously unused alucontrol signal 0100, this will indicate to the ALU that the operation required is the diffshift operation.

8.2.4 datapath

```
            \caption{datapath module}
            ////////////
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [3:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);
  logic jrcontrol1, jrcontrol2;
  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch,jrnext;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb, @\textbf{srcd}@;
  logic [31:0] result;
  // register file logic
  regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata,srcd);
  #other datapath logic#
  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb,srcd, alucontrol,instr[31:26],instr[10:6], aluout, zero);
endmodule
```

The only change to datapath was the adding of parameters for both regfile and alu.

9 Testing

For testing of instructions, the following programs were used.

9.1 andi

Code Block 20 andi test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        addi $3, $0, 12     #initialize $3 = 12
        andi $4, $3, 0xc0de
```

9.2 bne

Code Block 21 bne test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        addi $3, $0, 12     #initialize $3 = 12
        bne $2, $3, end
        j bend
end:    sw $2, 0($2)
bend:   addi $2, $0, 0
```

9.3 srl

Code Block 22 slt test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        addi $3, $0, 12     #initialize $3 = 12
        srl  $3, $3, 2
```

9.4 jr

Code Block 23 jr test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        jr $0
```

9.5 blt

Code Block 24 blt test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        addi $3, $0, 12     #initialize $3 = 12
        blt $2, $3, end
        j bend
end:    sw $2, 0($2)
bend:   addi $2, $0, 0
```

9.6 bge

Code Block 25 bge test program

```
main:   addi $2, $0, 5      #initialize $2 = 5
        addi $3, $0, 12     #initialize $3 = 12
        bne $3, $2, end
        j bend
end:    sw $2, 0($2)
bend:   addi $2, $0, 0
```

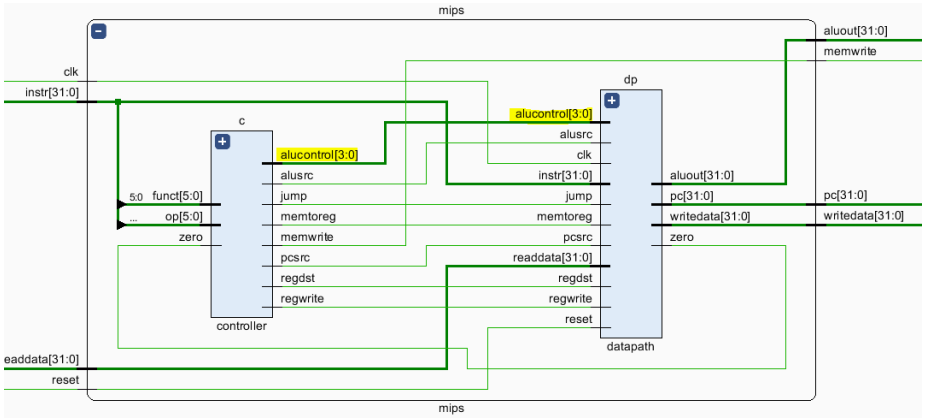
9.7 diffshift

Code Block 26 diffshift test program

```
main:   addi $2, $0, 33     #initialize $2 = 33
        addi $3, $0, 1    #initialize $3 = 1
        diffshift $4, $2, $3
        diffshift $5, $2, $0
```

10 summary

10.1 mips

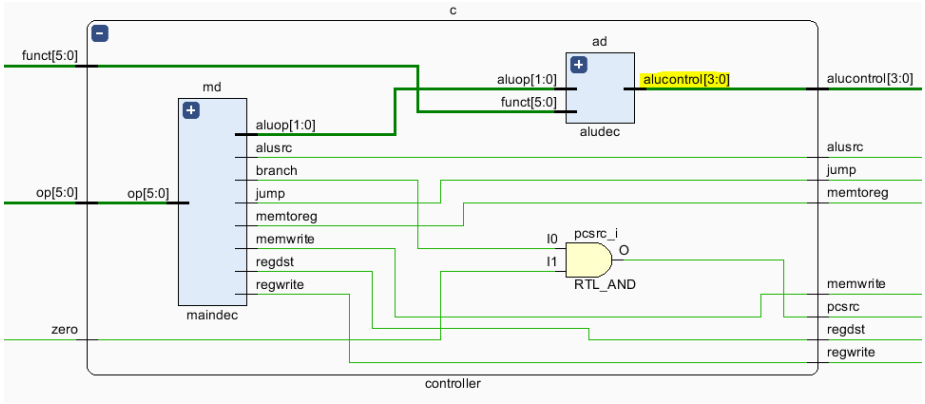


1. alucontrol was updated to 4 bits to accomodate srl

Code Block 27 mips module

```
        `timescale 1ns / 1ps
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata);

  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero;
  logic [3:0] alucontrol;

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

10.2 controller



1. alucontrol was updated to 4 bits to accomodate srl

Code Block 28 controller module

```
        `timescale 1ns / 1ps
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [3:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

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

  assign pcsrc = branch & zero;
endmodule
```

10.3 maindec

1. decoding logic was added for bne
2. decoding logic was added for blt
3. decoding logic was added for bge
4. decoding logic was added for andi

Code Block 29 maindec module

```
        `timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

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

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b000101: controls <= 9'b000100001; // BNE
      6'b000001: controls <= 9'b000100001; // BLT
      6'b100110: controls <= 9'b000100001; // BGE
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b001100: controls <= 9'b101000011; // ANDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

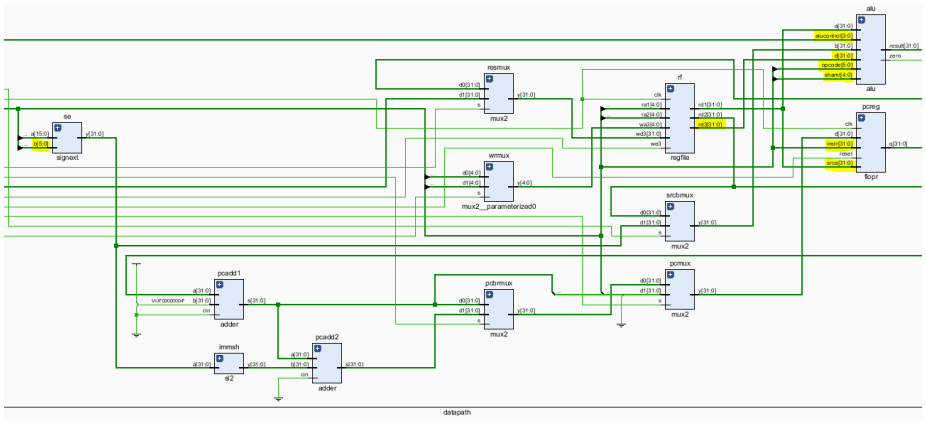
10.4 aludec

Code Block 30 aludec module

```
        ////////////
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [3:0] alucontrol);
  always_comb
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      2'b01: alucontrol <= 4'b1010;  // sub (for beq/bne/blt/bge)
      2'b11: alucontrol <= 4'b0000;  // and (for andi)
      default: case(funct)          // R-type instructions
          6'b100000: alucontrol <= 4'b0010; // add
          6'b100010: alucontrol <= 4'b1010; // sub
          6'b100100: alucontrol <= 4'b0000; // and
          6'b100101: alucontrol <= 4'b0001; // or
          6'b101010: alucontrol <= 4'b1011; // slt
          6'b000010: alucontrol <= 4'b0111; //srl
          6'b000100: alucontrol <= 4'b0100; //diffshift
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

1. In accordance with the earlier changes, all alucontrol signals were updated to 4 bits
2. Added alucontrol signals were added for srl and diffshift

10.5 datapath

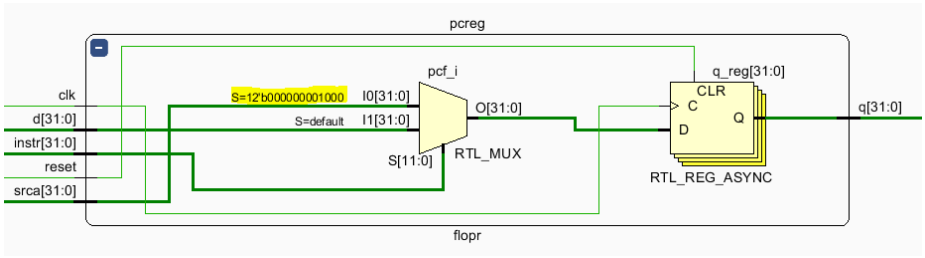


1. alucontrol was updated to 4 bits
2. R[rd], opcode, and shamt are all passed to the ALU
3. The opcode is passed to signext
4. R[rd] was added as an output of regfile
5. Regfile also outputs R[rd]
6. R[rs] and instr are passed to flopr

Code Block 31 datapath module

```
        ////////////
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [3:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);
  logic jrcontrol1, jrcontrol2;
  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch,jrnext;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb, srcd;
  logic [31:0] result;
  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext,instr,srca, pc);
//  adder       pcadd1(pc, 32'b100, pcplus4); //This is what is wrong, this assumes a "simpler
  adder #(32) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust to use the more complex adder
  sl2         immsh(signimm, signimmsh);
//  adder       pcadd2(pcplus4, signimmsh, pcbranch); //See comment above
  adder #(32) pcadd2(pcplus4, signimmsh, 'b0, pcbranch); //See comment above
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);
  // register file logic
  regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata,srcd);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0],instr[31:26], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb,srcd, alucontrol,instr[31:26],instr[10:6], aluout, zero);
endmodule
```

10.6 flopr



Code Block 32 flopr module

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

         logic [31:0] pcf = d;
         always_comb
         case({instr[31:26],instr[5:0]})
         12'b000000001000: pcf <= srca;
         default:   pcf <= d;
         endcase
always_ff @(posedge clk, posedge reset)
if (reset) q <= 0;
else       q <= pcf;
endmodule
```

1. New logic pcf was added to be the final next program counter
2. Decoding logic was added to choose R[rs] as the next pc in case of the jr instruction
3. The instr was added as a parameter for decoding of the jr instruction

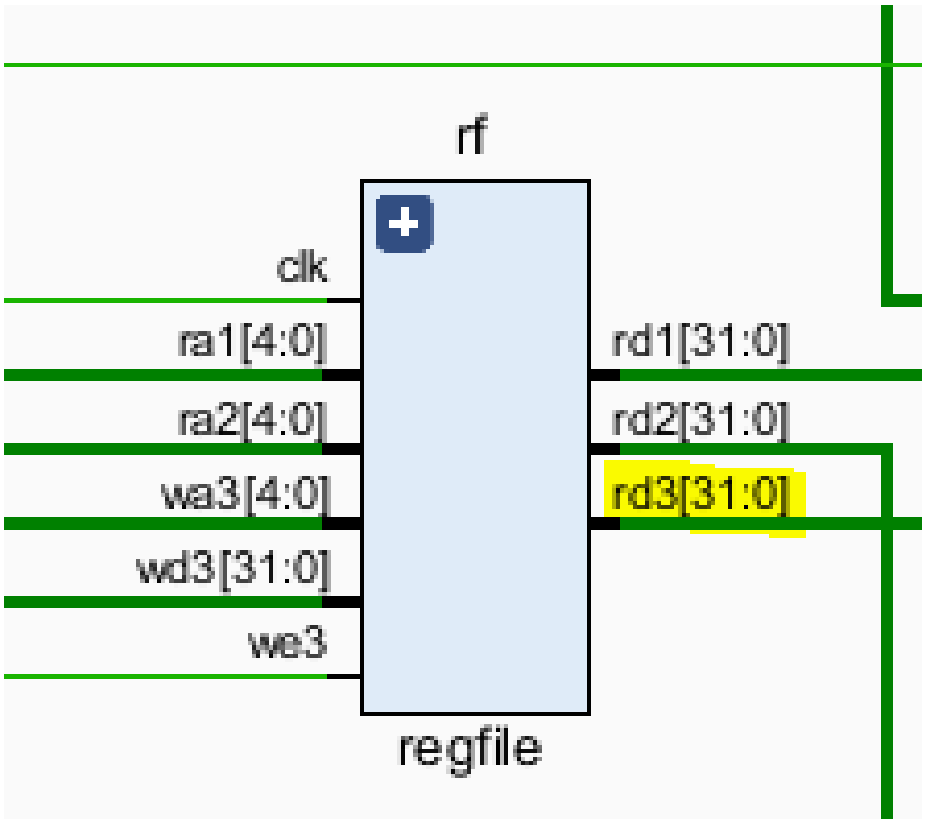
10.7 adder

No changes were made to the adder.

10.8 mux2

No changes were made to the mux2.

10.9 regfile

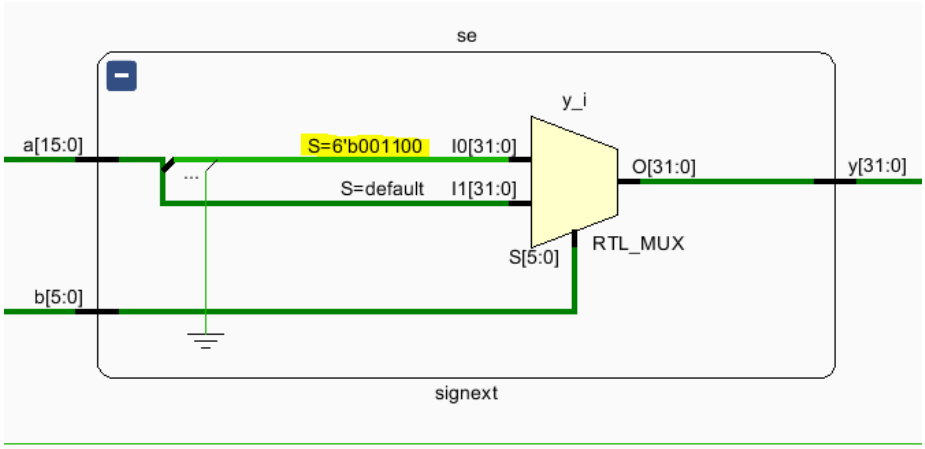


1. R[rd] was added as output logic for the diffshift instruction

Code Block 33 regfile module

```
        // regfile.v
// Register file for the single-cycle and multicycle processors

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2,rd3);

  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 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;
  assign rd3 = rf[wa3];
endmodule
```

10.10 signext

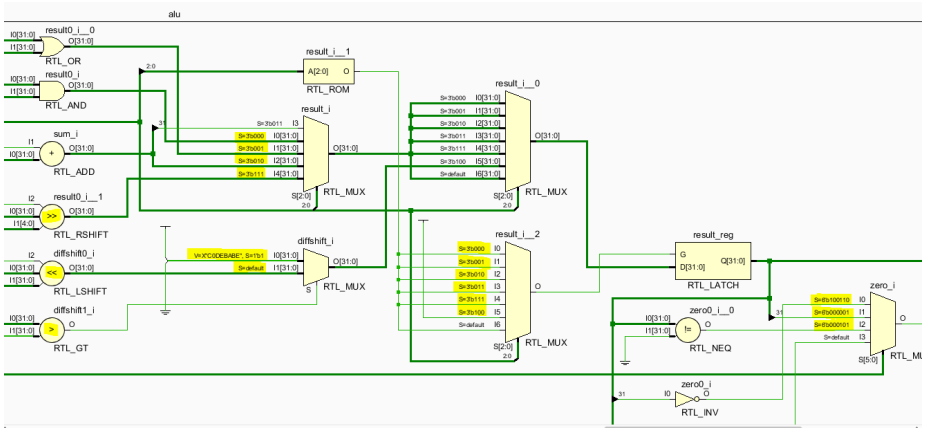


1. The opcode was added to signext's parameters in case of the andi instruction

Code Block 34 signext module

```
        module signext(input  logic [15:0] a, input logic [5:0] b,
        output logic [31:0] y);
always_comb
case(b)
6'b001100: y <= {{16{0}}, a}; // RTYPE
default:   y <= {{16{a[15]}}, a}; // illegal op
endcase
endmodule
```

10.11 ALU



1. diff was added for logic involving the diffshift instruction
2. Decoding logic was also added for the diffshift instruction

Code Block 35 ALU module

```
        ////////////
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b, d,
           input  logic [3:0]  alucontrol,
           input logic [5:0] opcode,
           input logic [4:0] shamt,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, diffshift, diff;
  assign diff =  a + ~b + alucontrol[3];
  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];

  always_comb
        begin
        if (diff > 32)
            diffshift <=32'hc0debabe;
        else
            diffshift <= (d << diff);
        end
  always_comb
    case (alucontrol[2:0])
      3'b000: result = a & b;
      3'b001: result = a | b;
      3'b010: result = sum;
      3'b011: result = sum[31];
      3'b111: result = b >> shamt;
      3'b100: result <= diffshift;
    endcase
  always_comb
    case(opcode)
      6'b100110: zero <= (~result[31]); // bge
      6'b000001: zero <= (result[31]); // blt
      6'b000101: zero <= (result != 32'b0); // bne
      default:   zero <= (result == 32'b0); // beq
    endcase
endmodule
```