Jack Vincent Nicolas

CS 21 Project: MIPS Single Cycle Processor Extension

Video Documentation:

https://drive.google.com/file/d/1GKEDQeuFRxMr-8Q2W3YCq_26_PHu1NTW/view?usp=sharing

Documentation:

Pseudo-Instructions

1. Load Immediate (li)

- opcode is 0x11
- target register is in bits 20-16 (rt)
- rs (bits 25-21) are Xs (Don't cares - not necessarily zeroes)
- immediate field (bits 15-0) contain values to be stored to rt; when used in assembly language, assume value to be stored always no longer than 16 bits; logically- or zero-extend the 16-bit value to 32 when storing to register

The first module that had a change was the mips module. In this module, I simply changed the number of bits that the alucontrol will take from 3 bits ([2:0]) to 4 bits ([3:0]). The mips module contains the inputs clk, reset, a 32-bit instr and readdata. It has the outputs 32-bit pc, aluout, write data and an output of memwrite. It has local variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, zero and a 4-bit alu control. Under this are the controller and data path modules.

```
`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
```

The second module that I changed was the controller module. In this module, I changed the number of bits for the alucontrol from 3 bits tp 4 bits. I also added the aluop to the input of aludec. It has the inputs 6-bit op and funct, and an input zero. It outputs memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump and the 4-bit alucontrol. It has local variables 2-bit aluop and a branch. Under this are the modules maindec and aludec.

```
`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, op, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

The third module that I manipulated to add some instructions was the maindec module. The maindec module takes in as an input the opcode of the instruction. From the opcode, we will get the control signals that the instruction needs. We have the variable controls which contains 9 bits, wherein each bit denote a control signal. It has the arrangement regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump and aluop.

```
`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;
```

It is followed by an instruction case which checks the op and assigns the corresponding control signals for that instruction.

```
  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'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      6'b010001: controls <= 9'b101000000; // li (addi)
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

In this module, I added another case for the load immediate instruction. It checks the op for the case when it is 0x11 and assigns the control 9'b101000000 which is the same as in addi.

The fourth module that was changed was the datapath module. This module handles the logic which occurs inside a mips single cycle datapath. It takes in as input the clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump, alucontrol, instr and readdata. Using the input, it outputs zero, pc, aluout and writedata.

It has local variables writetoreg, pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, srca, srcb and result which are used for the operations inside the 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 [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;
```

Under this, we have the next PC logic, register file logic and the ALU logic. The next PC logic is responsible for the address of the instructions being executed. The register file logic is responsible for the storage of the values that the instructions are executing. Finally, the alu logic is responsible for executing the instructions and returning the desired values.

```
  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder #(32) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust this to use
the more complex adder; wmt-modification
  sl2         immsh(signimm, signimmsh);
  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);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb, alucontrol, aluout, zero);
endmodule
```

In this module, I simply changed the number of bits that the alucontrol will take from 3 bits to 4 bits, which is highlighted above.

The fifth module that I changed and added some instructions was the alu module. This is where the main operation happens. We have 32-bit inputs a and b, and a 4-bit input called alucontrol. This

then outputs, a 32-bit output result and an output zero. We have some local variables here, a 32-bit condinvb, sum and mask. The condinvb checks alucontrol[3] (leftmost bit) if it will turn input b to negative. The sum adds input a and b. The mask is assigned a hex value of ‘h0000FFFF. Under this we have cases for alucontrol[2:0] (the remaining bits after the leftmost bit is dsicarded), it performs the operations depending on the last 3 bits of the alucotnrol. If it is ‘b000, it performs an AND operation, when it is ‘b001, it performs an OR operation, when it is ‘b010, it returns the sum, when it is ‘b011, it returns sum[31] and finally if it is ‘b100, it performs the AND operation between mask and b. The and between mask and b returns the zero-extended immediate field of the MIPS instruction.

```
module alu(input  logic [31:0] a, b,
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, mask;

  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];
  assign mask = 'h0000FFFF;

  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'b100: result = mask & b;
    endcase

  assign zero = (result == 32'b0);
endmodule
```

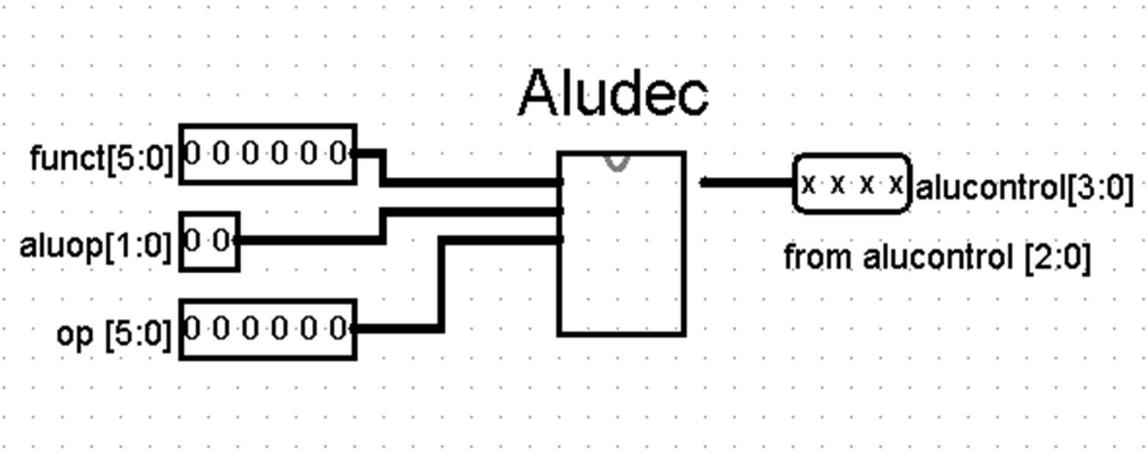
The last module that we changed and added some instructions to is the aludec module. In this module, we have the inputs 6-bit funct, 2-bit aluop and we added the 6-bit op as a change from the original module, then we have the 4-bit alucontrol for the output.

Under this, we have the case for the aluop. If the aluop is ‘b00, I placed another case inside wherein if it has an opcode of 6’b010001, then it has an alucontrol of 4’b0100 which is used for the load immediate instruction, else it has an alucontrol of 4’b0010. If it has an opcode 2’b01, then it has an alucontrol of 4’b0010. In the case of R-type instructions, it will be assigned to the default case wherein it checks the funct value of the instruction. If it has the funct 6'b100000, then it has an alucontrol 4’b0010 for add. If it has the funct 6'b100010, then it has an alucontrol 4'b1010 for sub. If it has the funct 6'b100100, then it has an alucontrol 4'b0000 for and. If it has the funct 6'b100101, then it has an alucontrol 4'b0001 for or. If it has the funct 6'b101010, then it has an alucontrol 4'b1011 for slt. And in default, it has alucontrol 4’bxxxx.

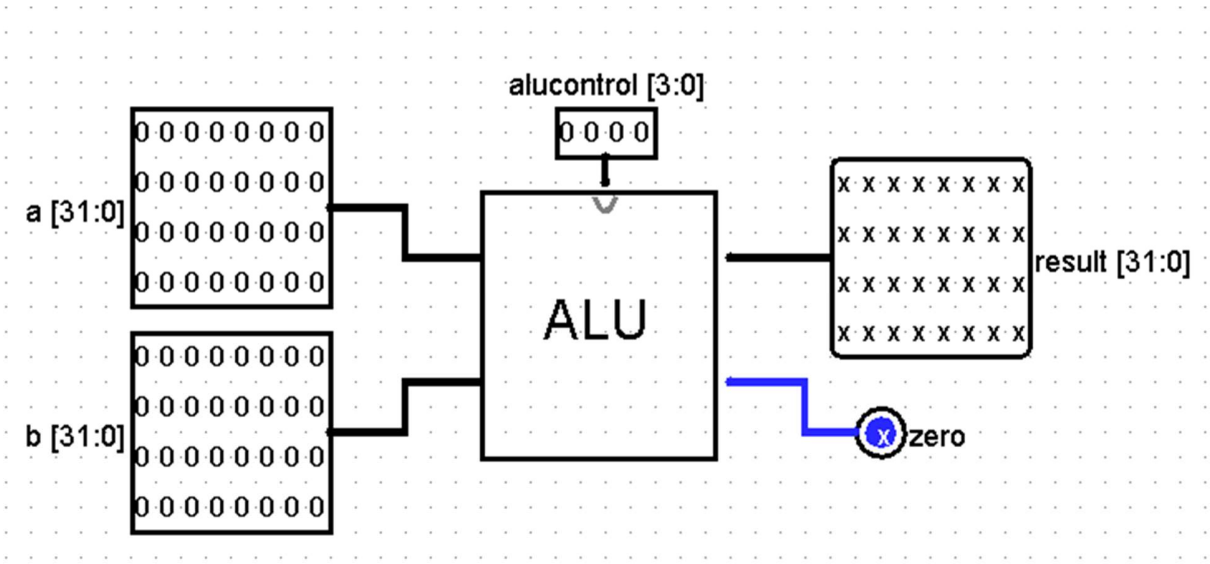
```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              input  logic [5:0] op,
              output logic [3:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: case(op)
        6'b010001: alucontrol <= 4'b0100; // li
        default: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      endcase
      2'b01: alucontrol <= 4'b0110;  // sub (for beq)
      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
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

Schematic Diagram

In the aludec, the number of bits for the alucontrol was changed, so this was depicted in the diagram below.



In the ALU, the input of alucontrol was changed from 3 bits top 4 bits, so this was depicted in the diagram below.

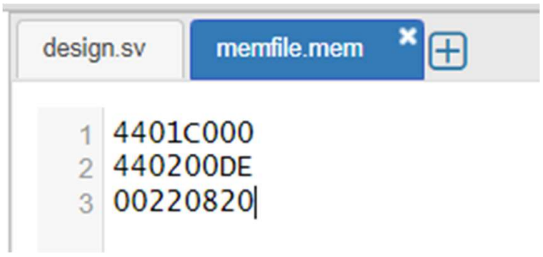


How the instructions were tested?

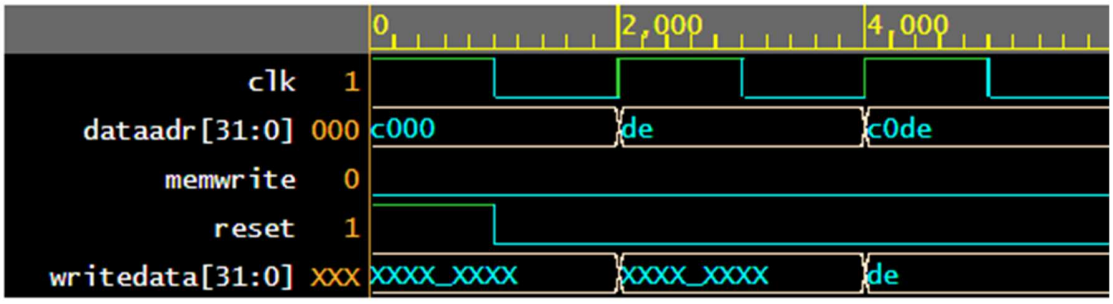
First Testcase: Checking if the li works as intended.

HEX CODE	MIPS Instruction
4401C000	li $1 C000
440200DE	li $2 00DE
00220820	add $1 $1 $2

The result should be 0xC0DE



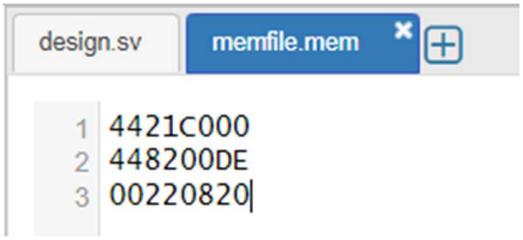
We managed to get the right output.



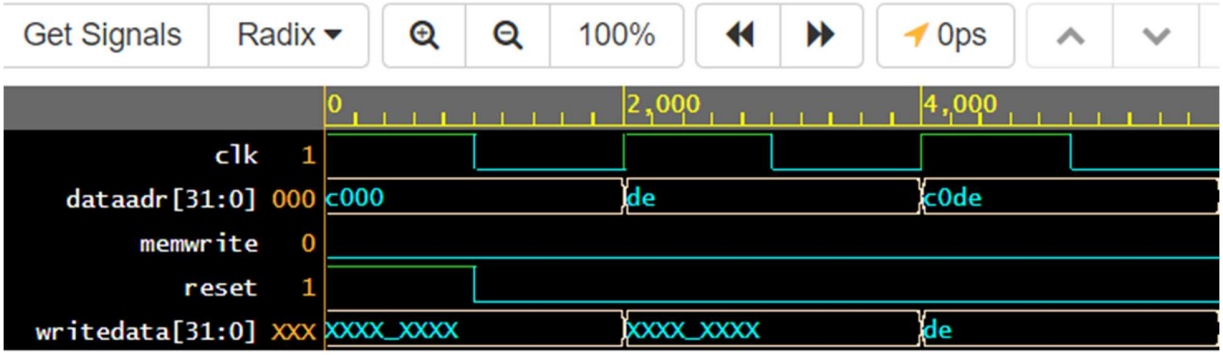
Second Testcase: Checking if the li works as intended even if the rs part of the instruction isn't 0. (Simply put, rs = XXXXX)

HEX CODE	MIPS Instruction
4421C000	li $1 0xC000
448200DE	li $2 0x00DE
00220820	add $1 $1 $2

Again, the result must be 0xC0DE



We still managed to get the right output.



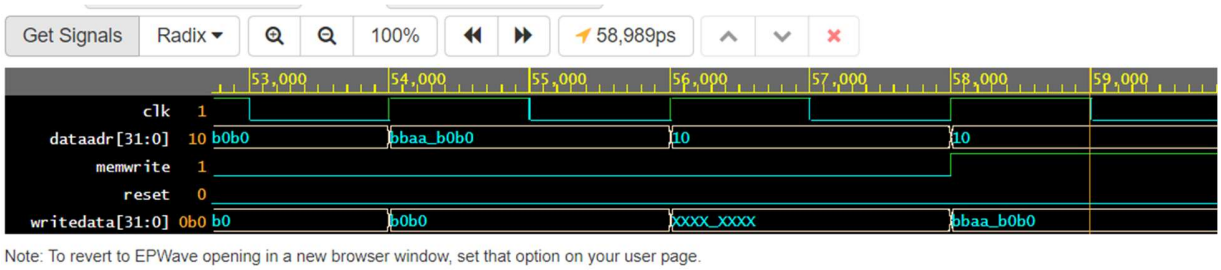
Third testcase: I utilized the memfile for lab 12, replacing the addi $register $0 0xXXXX to li $register 0xXXXX. This shows that it works with other instructions as well as show that it works with the previous outputs.

HEX CODE	MIPS Instructions
4401BBAA	li $1 0xBBAA
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1

00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
00210820	add $1 $1 $1
440200B0	li $2 0x00B0
00421820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00631820	add $2 $2 $2
00232020	add $4 $1 $3
44050010	li $5 0x0010
ACA40000	sw $4 ($5)

The stored output should be 0xBBAAB0B0

We managed to get the right output.



Note: To revert to EPWave opening in a new browser window, set that option on your user page.