Ayden Dauenhauer and Gigi Patmore
Prof. Yang
ELEN 122L
27 February 2024

Postlab 6

1. State Machine Code

```
module l6_SM(input clk,
                        input reset,
                        input [3:0] operation,
                        output reg _Extern,
                        output reg Gout,
                        output reg Ain,
                        output reg Gin,
                        output reg DPin,
                        output reg RdX,
                        output reg RdY,
                        output reg WrX,
                        output reg add_sub,
                        output reg pc_en,
                        output reg ILin,
                        output reg rf_sel,
                        output reg sw_sel,
                        output reg MemWr,
                        output reg AddrSel,
                        output reg bz,  /* new */
                        output reg bnz, /* new */
                        output [3:0] cur_state);

   // state definitions
        parameter FETCH       =  4'b0000;
        parameter DECODE      =  4'b0001;
```

```
parameter LOAD        =  4'b0010;
parameter READ_Y      =  4'b0011;
parameter READ_X      =  4'b0100;
parameter ADD         =  4'b0101;
parameter SUB         =  4'b0110;
parameter MV          =  4'b0111;
parameter WRITE_X     =  4'b1000;
parameter ADDI        =  4'b1001;
parameter SUBI        =  4'b1010;
parameter DISP        =  4'b1011;
parameter HALT        =  4'b1100;
parameter STORE       =  4'b1101;
/* TODO #3-1: add two states for bz and bnz */
parameter BNZ         =  4'b1110;
parameter BZ          =  4'b1111;

reg [3:0] state = FETCH;
assign cur_state = state;


always@(*)
begin
  case(state)
     FETCH:
        begin
           _Extern = 1'b0;
           Gout = 1'b0;
           //Iout = 1'b0;
           Ain = 1'b0;
           Gin = 1'b0;
           DPin = 1'b0;
```

```
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b1;
        ILin = 1'b1;
        rf_sel = 1'b0;
                sw_sel = 1'b0;
                MemWr = 1'b0;
                AddrSel = 1'b0;
                /* TODO #3-3: output handling for bz and bnz */
                bz = 1'b0;
                bnz = 1'b0;
    end
DECODE:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                sw_sel = 1'b0;
                MemWr = 1'b0;
```

```
                AddrSel = 1'b0;
                 /* TODO #3-3: output handling for bz and bnz */
                bz = 1'b0;
                bnz = 1'b0;
    end
LOAD:
    begin
        _Extern = 1'b1;
        Gout = 1'b0;
        //Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b1;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                sw_sel = 1'b0;
                MemWr = 1'b0;
                AddrSel = 1'b1;
                 /* TODO #3-3: output handling for bz and bnz */
                bz = 1'b0;
                bnz = 1'b0;
    end
READ_Y:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
```

```
        //Iout = 1'b0;
        Ain = 1'b1;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b1;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
READ_X:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b0;
        Ain = 1'b1;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
```

```
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
ADD:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
        DPin = 1'b0;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b1;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
```

```
SUB:
   begin
      _Extern = 1'b0;
      Gout = 1'b0;
      //Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b1;
      DPin = 1'b0;
      RdX = 1'b0;
      RdY = 1'b1;
      WrX = 1'b0;
      add_sub = 1'b1;
      pc_en = 1'b0;
      ILin = 1'b0;
      rf_sel = 1'b1;
                  sw_sel = 1'b0;
                  MemWr = 1'b0;
                  AddrSel = 1'b0;
                   /* TODO #3-3: output handling for bz and bnz */
                  bz = 1'b0;
                  bnz = 1'b0;
   end
MV:
   begin
      _Extern = 1'b0;
      Gout = 1'b0;
      //Iout = 1'b0;
      Ain = 1'b0;
      Gin = 1'b1;
      DPin = 1'b0;
      RdX = 1'b0;
```

```
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b1;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
WRITE_X:
    begin
        _Extern = 1'b0;
        Gout = 1'b1;
        //Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b1;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
```

```
                    /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
HALT:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                    /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
DISP:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b0;
```

```
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b1;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
ADDI:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b1;
        Ain = 1'b0;
        Gin = 1'b1;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
```

```
        rf_sel = 1'b0;
                    sw_sel = 1'b1;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
SUBI:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b1;
        Ain = 1'b0;
        Gin = 1'b1;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b1;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b1;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
STORE:
```

```
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b1;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b1;
                    AddrSel = 1'b1;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b0;
    end
BNZ:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b1;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
```

```
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
                    bz = 1'b0;
                    bnz = 1'b1;
    end
BZ:
    begin
        _Extern = 1'b0;
        Gout = 1'b0;
        //Iout = 1'b1;
        Ain = 1'b0;
        Gin = 1'b0;
        DPin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
        pc_en = 1'b0;
        ILin = 1'b0;
        rf_sel = 1'b0;
                    sw_sel = 1'b0;
                    MemWr = 1'b0;
                    AddrSel = 1'b0;
                     /* TODO #3-3: output handling for bz and bnz */
```

```
                        bz = 1'b1;
                        bnz = 1'b0;
                end
            endcase
        end



/*
opcode encodings
0000 - load
0001 - move
0010 - subtract
0011 - add
0100 - disp
0101 - HALT
0110 - subi
0111 - addi
1000 - store
1001 - bz
1010 - bnz
*/

        always@(posedge clk or posedge reset)
        begin

                if (reset==1) state <= FETCH;
                else
                case(state)
                        FETCH:
                                state <= DECODE;
```

```
DECODE:
                if(operation == 4'b0000) state <= LOAD;
                else if(operation == 4'b0001) state <= READ_Y;
                else if(operation == 4'b0010) state <= READ_X;
                else if(operation == 4'b0011) state <= READ_Y;
                else if(operation == 4'b0100) state <= DISP;
                else if(operation == 4'b0101) state <= HALT;
                else if(operation == 4'b0110) state <= READ_X;
                else if(operation == 4'b0111) state <= READ_X;
                else if(operation == 4'b1000) state <= STORE;
                /* TODO #3-2: state transitions for bz and bnz */
                else if(operation == 4'b1001) state <= BZ;
                else if(operation == 4'b1010) state <= BNZ;
                else state <= FETCH;
LOAD:
                //state <= DONE;
                state <= FETCH;
READ_Y:
                        if(operation == 4'b0001) state <= MV;
                        else if(operation == 4'b0011) state <= ADD;
                        else state <= READ_Y;
READ_X:
                        if(operation == 4'b0010) state <= SUB;
                        else if(operation == 4'b0110) state <= SUBI;
                        else if(operation == 4'b0111) state <= ADDI;
                        else state <= READ_X;
ADD:
                state <= WRITE_X;
SUB:
                state <= WRITE_X;
MV:
```

```
                    state <= WRITE_X;
            WRITE_X:
                        state <= FETCH;
            DISP:
                    state <= FETCH;
            ADDI:
                    state <= WRITE_X;
            SUBI:
                state <= WRITE_X;
            STORE:
                state <= FETCH;
        /* TODO #3-2: state transitions for bz and bnz */
            BNZ:
                state <= FETCH;
            BZ:
                state <= FETCH;
            HALT:
                state <= HALT;
            default: state <= FETCH;

        endcase

    end //end always


endmodule
```

2. Branch Logic Code

```
module l6_branch(
    input bnz, /* asserted if the instruction is bnz */
    input bz,  /* asserted if the instruction is bz */
```

```
        input [15:0] rf_data_in,
        output reg branch  /* output: assert if the branch condition is fulfilled */
);

 /* TODO #2: complete the following */
 always@(*)
 begin
  if(bnz==1)  /* if the instruction is bnz? */
     if(rf_data_in==0) branch=0;
     else branch=1;
  else if (bz==1) /* if the instruction is bz */
     if(rf_data_in==0) branch=1;
     else branch=0;
  else /* if the instruction is not a branch-related one? */
     branch=0;

 end

endmodule


module l6_branchPC(
        input [7:0] currPC,
        input [3:0] offset,
        output [7:0] adjustedPC
);

     /* TODO #1: calculate the target branch address */
     /* currPC + sign-extended offset */
        assign adjustedPC = currPC + {{4{offset[3]}}, {offset}};
endmodule
```

What problems did you encounter while implementing and testing your system?

The largest problems we encountered were when creating our mem.txt code. We struggled at first with creating the correct instruction to branch backwards at the end of our while loop, however after utilizing the signed number convention within our hex instruction, we were easily able to branch backwards allowing our code to work well.

Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn't think of this yourself.

No issues arose when demonstrating for the TA after we rectified our mem.txt file code. We were asking to demo the correct display value, showing that our loop had completed correctly. Additionally, we talked about the difference in do while versus while loops and how they can be implemented within the hex format we used in this lab. This was interesting to think about and created questions about how the type of loop impacts the makeup of our code.

Pick another CTI operation that could have been implemented. Describe what changes to the datapath would have been necessary to support this operation, and how your state machine would need to be modified to correctly sequence the relevant control signals.

We could've also implemented the NOP instruction. It requires no changes to the datapath because it does not perform any data transfer, computation, or control flow modification. It still goes through the standard instruction cycle stages: Fetch, Decode, Execute. However, during each of these stages, it does not trigger any meaningful actions. Still is done so that a stall can be implemented so the pipelining can avoid any hazards if the compiler doesn't take care of that itself.