	T_0	T_1	T_2	T_3	T_4	T_5
mv	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rY or IR, rX$_{in}$, Done*		
mvt	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select IR, rX$_{in}$, Done*		
add	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or IR, G$_{in}$*	*Select G, rX$_{in}$, Done*
sub	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or IR, AddSub, G$_{in}$*	*Select G, rX$_{in}$, Done*
and	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or IR, ALU_and, G$_{in}$*	*Select G, rX$_{in}$, Done*
ld	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rY, ADDR$_{in}$*		*Select DIN, rX$_{in}$, Done*
st	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select rY, ADDR$_{in}$*	*Select rX, DOUT$_{in}$, W_D, ~~Done~~*	

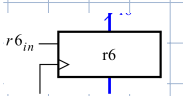
	T_0	T_1	T_2	T_3	T_4	T_5
b{cond}	*Select pc, ADDR$_{in}$, pc_incr*		*IR$_{in}$*	*Select pc, A$_{in}$, if ~~cond=1~~, Done*	*Select IR, G$_{in}$*	*Select G, pc$_{in}$, Done*

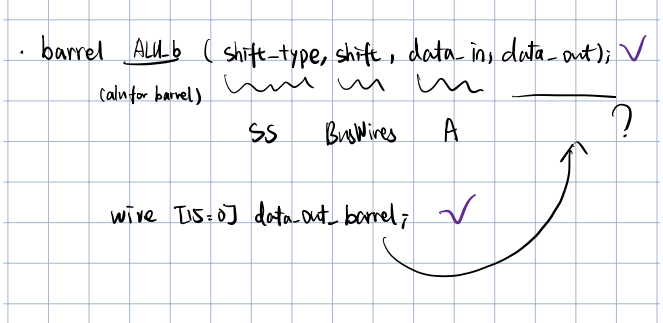
	T_0	T_1	T_2	T_3	T_4	T_5
push	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*sp_decr*	*Select rY, ADDR$_{in}$*	*Select rX, DOUT$_{in}$, W_D, Done*
pop	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rY, ADDR$_{in}$, sp_incr*	*Wait*	*Select DIN, rX$_{in}$, Done*
bl	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select pc, A$_{in}$, r6$_{in}$*	*Select #D, G$_{in}$*	*Select G, pc$_{in}$, Done*
cmp	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or #D, AddSub, F$_{in}$, Done*	
lsl, lsr asr, ror	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or #D, do_shift, G$_{in}$, F$_{in}$*	*Select G, rX$_{in}$, Done*

The processor will have eight new instructions, which are listed in Table 5. The `push rX` instruction is used to store the contents of a register, *rX*, onto the stack. This instruction first decrements the *sp* (register $r5$), and then stores *rX* into memory at the address in *sp*. The `pop rX` instruction is used to load data into a register *rX* from memory at the address in *sp*. After loading this data, *sp* is then incremented.

Recall from Lab 8 that the `b{cond}` instruction uses the `XXX` field to encode a *condition*, where `XXX` = 000 (*none*), 001 (*eq*), 010 (*ne*), and so on. Implement the `bl` instruction by using the previously-unassigned code `XXX` = 111.

You should encode the `cmp` instruction in the same way as the `add`, `sub`, and `and` instructions. Use the previously-unassigned code `III = 111`; if *Op2* is a register, then `cmp` is encoded as `1110XXX000000YYY`, and if *Op2* is *#D*, then `cmp` is encoded as `1111XXXDDDDDDDDD`. For the shift/rotate instructions you should also use the code `III = 111`, as follows. When *Op2* is a register, encode these instructions as `1110XXX10SS00YYY`, and when *Op2* is *#D* encode them as `1110XXX1SSDDDDD`. In these encodings `SS` specifies the type of shift/rotate, where `SS` = 00 (`lsl`), 01 (`lsr`), 10 (`asr`), or 11 (`ror`). Note that the instruction `cmp rX, rY` and the various shift/rotate instructions share the most-significant digits of their encodings (bits 15-9) which are `1110XXX`. However, for this `cmp` instruction the next six digits (bits 8-3) are `000000`, whereas for the shift/rotate instructions these bits are never all zeros. To differentiate between `cmp rX, rY` and the shift/rotate instructions it is sufficient to examine the digit in bit-position 8.



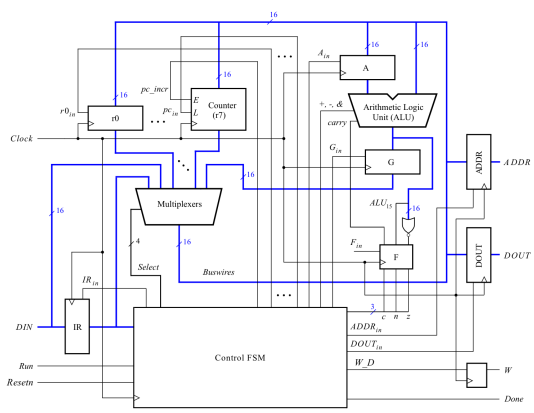


In assembly-language code register $r5$ can be referred to as the *stack pointer* register, *sp*. It is used as an *address* for pushing and popping data on the stack. Since it is an up/down counter, the *sp* can easily be *decremented* before a register is *pushed* onto the stack, and *incremented* after a register has been *popped* off of the stack. The processor's control unit decrements *sp* by using the *sp_decr* signal shown in Figure 24, and increments this register by using the *sp_incr* signal. These signals are just the *up/down* control inputs for the counter. Arbitrary data can also be loaded into register $r5$ (*sp*) in the same way as in Lab 8, by using the *r5$_{in}$* signal.

```
reg sp_decr;
reg sp_incr;

push = 3'b101;
pob = 3'b100;

parameter bl=3'b111;

parameter cmp = 3'b111;

wire [15:0] check_inst;
assign check_inst = IR [8:3];

wire [1:0] ss;
assign ss = IR [0:5]

wire shift_check
assign shift_check = IR[7];

reg R6_in;
```

Internal processor Bus:

```
check_inst : BusWires = {12'b0, IR[8:0]};
```

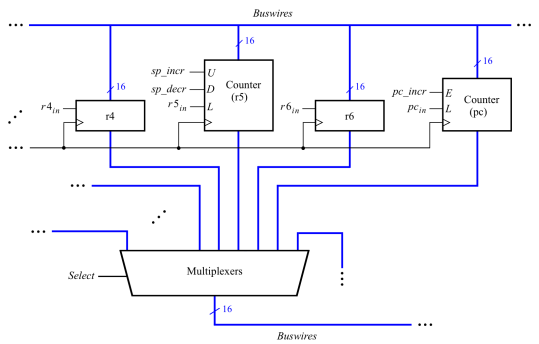


Selection for Buswires Multiplexer:

parameter _shift = 4'b1100;
(bits ss = 00 (lsl))

Control FSM outputs:

sp_incr = 1'b0; ✓
sp_decr = 1'b0; ✓
Rb_in = R_in[6]; ✓



and 1010XXX000000YYY, respectively. You should encode the pop instruction similarly to ld, with the encoding 1001XXX000000101. Also, encode push similarly to st, using the code 1011XXX000000101. Notice that for both push and pop the YYY field is hard-coded to correspond to the stack pointer register, *r*5.

To implement each of the new instructions, you will need to augment the finite state machine for your processor. Table 6 indicates how the required signals may be asserted in each time step to implement the instructions in Table 5. Following the style used in Labs 7 and 8, in this table *Select pc* means "put the program counter onto the *Buswires*," *Select #D* means "put the sign-extended immediate data that is in the instruction register (*IR*) onto the *Buswires*," *W_D* means "assert the input to the flip-flop that provides the *write* signal for the memory," and *do_shift* means "set the control signal on the ALU such that its output will be provided by the barrel shifter."

	T_0	T_1	T_2	T_3	T_4	T_5
push	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*sp_decr*	*Select rY, ADDR$_{in}$*	*Select rX, DOUT$_{in}$, W_D, Done*
pop	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rY, ADDR$_{in}$, sp_incr*	*Wait*	*Select DIN, rX$_{in}$, Done*
bl	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select pc, A$_{in}$, r6$_{in}$*	*Select #D, G$_{in}$*	*Select G, pc$_{in}$, Done*
cmp	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or #D, AddSub, F$_{in}$, Done*	
lsl, lsr asr, ror	*Select pc, ADDR$_{in}$, pc_incr*	*Wait*	*IR$_{in}$*	*Select rX, A$_{in}$*	*Select rY or #D, do_shift, G$_{in}$, F$_{in}$*	*Select G, rX$_{in}$, Done*

The processor will have eight new instructions, which are listed in Table 5. The push rX instruction is used to store the contents of a register, *rX*, onto the stack. This instruction first decrements the *sp* (register *r*5), and then stores *rX* into memory at the address in *sp*. The pop rX instruction is used to load data into a register *rX* from memory at the address in *sp*. After loading this data, *sp* is then incremented.

Operation	Function performed
push rX	$sp \leftarrow sp - 1$, $[sp] \leftarrow rX$
pop rX	$rX \leftarrow [sp]$, $sp \leftarrow sp + 1$
bl Label	$r6 \leftarrow pc$, $pc \leftarrow Label$
cmp rX, Op2	performs $rX - Op2$, sets flags
lsl rX, Op2	$rX \leftarrow rX \ll Op2$
lsr rX, Op2	$rX \leftarrow rX \gg Op2$
asr rX, Op2	$rX \leftarrow rX \ggg Op2$
ror rX, Op2	$rX \leftarrow rX \ll\gg Op2$

T3:
```
Cmp: begin ✓
     Select = rX; ✓
     A_in = 1'b1; ✓
     end ✓
st, push: begin ✓
     if (Imm==1'b0) begin ✓
          < store instructions; >
     end ✓
     else begin ✓
          sp_decr = 1'b1; ✓
     end ✓
ld, pop: begin ✓
     < store instruction >
     if (Imm==1'b1)  sp_incr = 1'b1; ✓
mvt, b_: case (cond)
          bl: Rb_in = 1'b1; ✓
          < other instructions >
```

You should encode the cmp instruction in the same way as the add, sub, and and instructions. Use the previously-unassigned code III = 111; if *Op2* is a register, then cmp is encoded as 1110XXX000000YYY, and if *Op2* is #*D*, then cmp is encoded as 1111XXXDDDDDDDDD. For the shift/rotate instructions you should also use the code III = 111, as follows. When *Op2* is a register, encode these instructions as 1110XXX10SS00YYY, and when *Op2* is #*D* encode them as 1110XXX11SS0DDDD. In these encodings SS specifies the type of shift/rotate, where SS = 00 (lsl), 01 (lsr), 10 (asr), or 11 (ror). Note that the instruction cmp rX, rY and

T4:
```
Cmp: begin
     if (Imm==1'b1) begin          (#D) ✓
          Select = _IR8_IR8_0; ✓
          AddSub = 1'b1; ✓
          F_in = 1'b1; ✓
          Done = 1'b1; ✓
     end ✓
     else begin ✓
          if(check_inst = 6'b000000) begin  (cmp) ✓
               Select = rY; ✓
               AddSub = 1'b1; ✓
               F_in = 1'b1; ✓
               Done = 1'b1; ✓
     end ✓
```

```
else begin          (shift and rotate) ✓
     if (shift_check == 0) ✓
          Select = rY; ✓
     else  Select = _shift; ✓
     G_in = 1'b1; ✓
     F_in = 1'b1; ✓
end end end ✓
st, push: if (Imm==0) ... ✓
          else begin ✓
          Select = rY; ADDR_in = 1'b1; ✓
     end. ✓
ld, pop:  ; ✓
```

```
T5: st, push: begin
    if (Imm == 0) begin (just st, move to next cycle)
        Done = 1'b1;
    end
    else begin
        Select = rX;
        DOUT_in = 1'b1;
        W_D = 1'b1;
        Done = 1'b1;
    end
end
```

T_5

Select rX, $DOUT_{in}$, W_D, Done

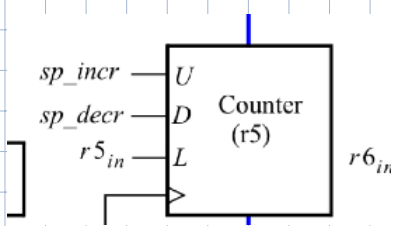
```
ld, pop: begin
    Select = _DIN;
    rX_in = 1'b1;
    Done = 1'b1;
end
cmp: begin (for shift)
    Select = _G;
    rX_in = 1'b1;
    Done = 1'b1;
end
b_ : b1: PC_in = 1'b1;
```

Select DIN, rX_{in}, Done

Select G, rX_{in}, Done

```
// r7 is program counter
// module pc_count(R, Resetn, Clock, E, L, Q);
pc_count reg_pc (BusWires, Resetn, Clock, pc_incr, pc_in, pc);
```

```
// r5 is stack pointer
sp_count reg_sp (BusWires, Resetn, Clock, sp_incr, sp_decr, R_in[5], r5);
```




```
module sp_count (R, Resetn, U, D, L, Q)
    input [15:0] R;
    input Resetn, Clock, U, D, L;
    output [15:0] Q;
    reg [15:0] Q;
    always @(posedge Clock)
        if (!Resetn)
            Q <= 16'b0;
        else if (L)
            Q <= R;
        else if (U)
            Q <= Q+1'b1;
        else if (D)
            Q <= Q-1'b1;
endmodule
```

```
module pc_count(R, Resetn, Clock, E, L, Q);
    input [15:0] R;
    input Resetn, Clock, E, L;
    output [15:0] Q;
    reg [15:0] Q;

    always @(posedge Clock)
        if (!Resetn)
            Q <= 16'b0;
        else if (L)
            Q <= R;
        else if (E)
            Q <= Q + 1'b1;
endmodule
```

```
reg Shift_Rotate;
Shift_rotate = 1'b0;        (initialization)

in T4: after do_shift:  Shift_rotate = 1'b1;

alu: if (!shift_rotate)
```