Design an 8-bit Processor with Verilog at Behavioral Level (II)

/* increment and decrement register pair */
task inx_dcx;
  case(ir[5:3])
    0: {regb, regc} = {regb, regc} + 1; // INX B
    1: {regb, regc} = {regb, regc} - 1; // DCX B
    2: {regd, rege} = {regd, rege} + 1; // INX D
    3: {regd, rege} = {regd, rege} - 1; // DCX D
    4: {regh, regl} = {regh, regl} + 1; // INX H
    5: {regh, regl} = {regh, regl} - 1; // DCX H
    6: sp = sp + 1;                     // INX SP
    7: sp = sp - 1;                     // DCX SP
  endcase
endtask

/* load register pair immediate */
task lrpi;
  case(ir[5:4])
    0: adread({regb, regc}); // LXI B
    1: adread({regd, rege}); // LXI D
    2: adread({regh, regl}); // LXI H
    3: adread(sp);           // LXI SP
  endcase
endtask

/* fetch address from pc+1, pc+2 */
task adread;
  output[15:0] address;
  begin
    memread(address[7:0], pc);
    pc = pc + 1;
    if(!int) pc = pc + 1;
  end
endtask

/* memory read */
task memread;
  output[7:0] rdata;
  input[15:0] raddr;
  begin
    @(posedge clock) addr = raddr;
    s = 0;
    acontrol = 1;
    dcontrol = 1;
    iomff = int;
    s0ff = int;
    s1ff = 1;
    aeff = 1;
    @(posedge clock) aleff = 0;
    @posedge clock]
  end
endtask

/* memory write */
task memwrite;
  input[7:0] wdata;
  input[15:0] waddr;
  begin
    @(posedge clock) alleff = 1;
    s0ff = 1;
    s1ff = 0;
    acontrol = 1;
    dcontrol = 1;
    iomff = 0;
    addr = waddr;
    acontrol = 1;
    dcontrol = 1;
    bar
    @posedge clock) aleff = 0;
  end
endtask
```vhdl
/* add into regh, regl pair */
task addhl;
begin
    case(ir[5:4])
    0: (cc, regh, regl) = (1'b0, regh, regl) + (regb, regc); // DAD B
    1: (cc, regh, regl) = (1'b0, regh, regl) + (regd, rege); // DAD D
    2: (cc, regh, regl) = (1'b0, regh, regl) + (regh, regl); // DAD H
    3: (cc, regh, regl) = (1'b0, regh, regl) + sp;           // DAD SP
    endcase
    holdreq;
end
endtask

/* store and load instruction */
task sta_lda; reg[15:0] ra;
begin
    case(ir[5:3])
    0: memwrite(acc, {regb, regc}); // STAX B
    1: memread(acc, {regb, regc});  // LDAX B
    2: memwrite(acc, {regd, rege}); // STAX D
    3: memread(acc, {regd, rege});  // LDAX D
    4: // SHLD
        adread(ra);
        memwrite(regl, ra);
        memwrite(regh, ra + 1);
    end
    5: // LHLD
        adread(ra);
        memread(regl, ra);
        memread(regh, ra + 1);
    end
    6: // STA
        adread(ra);
        memwrite(acc, ra);
    end
    7: // LDA
        adread(ra);
        memread(acc, ra);
    end
endcase
endtask

/* reads from an i/o port */
task ioread;
input[7:0] sa;
begin
    @(posedge clock)
        aeff = 1;
        s0ff = 0;
        s1ff = 1;
        s = 0;
        iomff = 1;
        addr = {sa, sa};
        acontrol = 1;
        dcontrol = 1;
        checkint;
        @posedge clock
        intaff = 0;
        @(posedge clock)
        dcontrol = 0;
        if(int)
            intaff = 0;
        else
            read = 0;
    @(posedge clock)
        ready_hold;
        @(posedge clock)
        checkint;
        @posedge clock
        intaff = 1;
        read = 1;
        acc = ad; 
        if(dflags[2])
            $display("IN %h  data = %h", sa, acc);
    end
endtask

/* conditional jump, call and return instructions */
task condjcr;
reg branch;
begin
    case(ir[5:3])
    0: branch = !cz; // JNZ CNZ RNZ
    1: branch = cz;  // JZ  CZ  RZ
    2: branch = !cc; // JNC CNC RNC
    3: branch = cc;  // JC  CC  RC
    4: branch = !cp; // JPO CPO RPO
    5: branch = cp;  // JPE CPE RPE
    6: branch = !cs; // JP  CP  RP
    7: branch = cs;  // JM  CM  RM
    endcase
    if(branch)
        case(ir[2:0])
        0: // return
            pop2b(pc[15:8], pc[7:0]);
        2: // jump
            adread(pc);
        endcase
end
endtask
```
// call
begin :call
reg[15:0] newpc;
adread(newpc);
push2b(pc[15:8], pc[7:0]);
end

default no_instruction;
endcase
else
case(ir[2:0])
0: ;
2, 4:
begin
memread(data, pc);
end

default no_instruction;
endcase
end

/* push register pair from stack */
task push;
case(ir[5:4])
0: push2b(regb, regc); // PUSH B
1: push2b(regd, rege); // PUSH D
2: push2b(regh, regl); // PUSH H
3: push2b(acc, {cs,cz,1'b1,cac,1'b1,cp,1'b1,cc}); // PUSH PSW
end
default no_instruction;
endtask

/* pop register pair from stack */
task pop;
reg null1;
case(ir[5:4])
0: pop2b(regb, regc); // POP B
1: pop2b(regd, rege); // POP D
2: pop2b(regh, regl); // POP H
3: pop2b(acc, {cs,cz,1'b1,cac,1'b1,cp,1'b1,cc}); // POP PSW
end
default no_instruction;
endtask

/* pop 2 bytes from stack */
task pop2b;
output[7:0] highb, lowb;
begin
memread(lowb, sp);
sp = sp + 1;
memread(highb, sp);
sp = sp + 1;
end

/* ram module for 8085a */
module ram85a(ale, ad, a, write, read, iomout);
reg[15:0] address;
reg[7:0] ram[ramsize-1:0];
tri[7:0] ad = (read || iomout) ? 'bz : ad;
inout[7:0] ad;
input[7:0] a;
input ale, write, read, iomout;
reg[7:0] ad_reg;
tri[7:0] ad = (read || iomout) ? 'bz : ad_reg;
parameter ramsize = 'h7A;
reg[7:0] ram[ramsize-1:0];
reg[15:0] address;
always @(negedge ale) begin
    address = {a, ad};
    if(dflags[1])
        $display("Reading %h into ram address", address);
end
always @(negedge read) begin
    if(dflags[1])
        $display("Reading %h = ram[%h]", ram[address], address);
ad_reg = ram[address];
end
always @(negedge write) begin
    if(dflags[1])
        $display("Writing ram[%h] = %h", address, ad);
    ram[address] = ad;
end

//define contents of RAM
//sorting data: ram['h2f] contains no. of elements in the array,
//array begins at ram['h30]
initial begin
    ram['h0] = 'h06; ram['h1] = 'h00; ram['h2] = 'h21; ram['h3] = 'h2F;
    ram['h4] = 'h00; ram['h5] = 'h3F; ram['h6] = 'h0D; ram['h7] = 'h23;
    ram['h8] = 'h7E; ram['h9] = 'h23; ram['ha] = 'hBF; ram['hb] = 'hD2;
    ram['hc] = 'h15; ram['hd] = 'h00; ram['he] = 'h56; ram['hf] = 'h77;
    ram['h10] = 'h2B; ram['h11] = 'h72; ram['h12] = 'h23; ram['h13] = 'h06;
    ram['h14] = 'h01; ram['h15] = 'h00; ram['h16] = 'hC2; ram['h17] = 'h08;
    ram['h18] = 'h00; ram['h19] = 'h05; ram['h1a] = 'hCA; ram['h1b] = 'h00;
    ram['h1c] = 'h00; ram['h1d] = 'h7E; ram['h1e] = 'h00; ram['h1f] = 'hDB;
    ram['h20] = 'h0F; ram['h21] = 'h77; ram['h22] = 'hC9; ram['h23] = 'h7E;
    ram['h24] = 'h0F; ram['h25] = 'h0F; ram['h26] = 'h0F; ram['h27] = 'h0F;
    ram['h28] = 'hCD; ram['h29] = 'h30; ram['h2a] = 'h00; ram['h2b] = 'h7E;
    ram['h2c] = 'hCD; ram['h2d] = 'h30; ram['h2e] = 'h00; ram['h2f] = 'h09;
    /* last address of this program is 007AH (STACK) */
end

//print out the contents of the RAM
task printmem;
integer i;
reg[7:0] data;
begin
    $write("dump of ram");
    for(i = 0; i < ramsize; i = i + 1) begin
        data = ram[i];
        if((i % 4) == 0) $write(" ");
        if((i % 16) == 0) $write("%h: ", i);
        $write("%h", data);
        $write(" ");
    end
    $write("n");
end
task
module intel_8085a
  (clock, x2, resetff, sodff, sid, trap,
   rst7p5, rst6p5, rst5p5, intr, intaff,
   ad, a, s0, alleff, writeout, readout, s1,
   iomout, ready, nreset,
   clockff, hldaff, hold);

always begin:run_processor
#1 reset_sequence;
fork
  execute_instructions; // Instructions executed
  wait(!nreset)         // in parallel with reset
  @(posedge clock) disable run_processor; // control. Reset will
join                                  // and all tasks and
end // functions enabled from
     // It when nreset set to 0.

module s85; // simulation control module
....
....
....

// instantiate the clock
osc timebase(clock);

// instantiate the RAM module
ram85a r0(ale, ad, a, write, read, iomout);

// instantiate the 8085a processor module
intel_8085a i85(clock, , , , , trap,
                 rst7p5, rst6p5, rst5p5, intr, ,
                 ad, a, s0, ale, write, read, ,
                 iomout, ready, nreset,
                 , , hold);

initial begin
  @(posedge i85.haltff) @(posedge timebase.clock);
  r0.printmem;
  $finish;//disable clockwave;
  //disable pclockwave;
  disable i85.run_processor;
end

Project 4 is to design a 16-bit processor with HDL Verilog
at Behavioral Level.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

This is the instruction format of a 32-bit MIPS processor

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Instruction set of the 16-bit processor (Project 4)
A word has two bytes in project 4. It has 16 registers.

r15 is the program counter. r14 is the stack pointer.

Idb (LoaD Byte)

Idb rd c
Immediate: The 8-bit unsigned constant c is loaded into
the least significant byte of register rd;
the most significant byte of register rd is set to zero.
Example: Idb r12 29

Idb rd rb d
Indexed: The byte in memory addressed by adding constant d to
the base register rb is loaded into the least significant byte of register rd;
the most significant byte of register rd is set to zero.
d has to be in the range 0..15.
Example: Idb r12 r15 9
ldw (Load Word)
ldw rd c
*Immediate:* The 16-bit unsigned constant c is loaded into the register rd.
Example: ldw r12 2893
        ldw r12 62

ldw rd rb d
*Indexed:* The word in memory addressed by adding constant d to the base register rb is loaded into the register rd.
d has to be in the range 0..15.
Example: ldw r12 r15 9

stb (Store Byte)
stb rs rb d
*Indexed:* The least significant byte of register rs is written to the byte in memory addressed by adding constant d to the base register rb.
d has to in the range 0..15.  
Example: stb r12 r15 9

stw (Store Word)
stw rs rb d
*Indexed:* The value of the source register rs is written to the word at the memory addressed by adding constant d to the base register rb. d has to be in the range 0..15.
Example: stw r12 r15 9

bra (Branch Absolute)
bra rx op ry c
*Register/Register:* The value of the first register rx is compared to the value of the second register ry using the compare operation op. If the result of the comparison is true, program control flows to memory location c (a 16-bit unsigned constant), otherwise program flow continues to the following instruction.
Example: bra r5 < r6 27936
        bra r5 < r6 label

bra rx op d c
*Register/Constant:* The value of the register rx is compared to the constant d (which has to be in the range 0..15) using the compare operation op. If the result of the comparison is true, program control flows to memory location c (a 16-bit unsigned constant), otherwise program flow continues to the following instruction.
Example: bra r5 < 6 27936
        bra r5 < 6 label.

brr (Branch Relative)
brr rx op ry c
*Register/Register:* The value of the first register rx is compared to the value of the second register ry using the compare operation op. If the result of the comparison is true, program control flows to memory location c (a 16-bit unsigned constant), otherwise program flow continues to the following instruction. The difference between the value of the immediate constant and the value of the Program Counter must be in the range -128..+127.
Example: brr r5 < r6 27936
        brr r5 < r6 label
add (ADD)  
add rx ry rd  
**Register/Register:** The value of the first register rx is added to the value of the second register ry and the result stored in the destination register rd. The register rx can only be in the range r0-r7.
The instruction works correctly even when some or all the registers are the same.  
Example: add r7 r10 r11  
add r7 r10 r10

add d rs rd  
**Register/Constant:** The value of the constant d (in the range 1..8) is added to the value of the source register rs and the result stored in the destination register rd. The instruction works correctly even when the source and destination registers are the same.  
Example: add 8 r10 r11  
add 8 r10 r10

sub (SUBtract)  
sub rx ry rd  
**Register/Register:** The value of the first register rx is subtracted from the value of the second register ry and the result stored in the destination register rd. The register can only be in the range r0-r7. The instruction works correctly even when some or all the registers are the same.  
Example: sub r7 r10 r11  
sub r7 r10 r10

sub d rs rd  
**Register/Constant:** The value of the constant d (in the range 1..8) is subtracted from the value of the source register rs and the result stored in the destination register rd. The instruction works correctly even when the source and destination registers are the same.  
Example: sub 8 r10 r11  
sub 8 r10 r10

mul (MULTiply)  
mul rx ry rd  
**Register/Register:** The value of the first register rx is multiplied with the value of the second register ry and the result stored in the destination register rd. The register rx can only be in the range r0-r7. The instruction works correctly even when some or all the registers are the same.  
Example: mul r7 r10 r11  
mul r7 r10 r10

mul d rs rd  
**Register/Constant:** The value of the constant d (in the range 1..8) is multiplied with the value of the source register rs and the result stored in the destination register rd. The instruction works correctly even when the source and destination registers are the same.  
Example: mul 8 r10 r11  
mul 8 r10 r10

div (Divide)  
div rx ry rd  
**Register/Register:** The value of the second register ry is divided by the value of the first register rx and the quotient stored in the destination register rd. The remainder is ignored. The register rx can only be in the range r0-r7. The instruction works correctly even when some or all the registers are the same.  
If the value of the first register rx is zero, the processor should display an appropriate error message and halt.  
Example: div r7 r10 r11  
div r7 r10 r10

div d rs rd  
**Register/Constant:** The value of the source register rs is divided by the value of the constant d (in the range 1..8) and the quotient stored in the destination register rd. The remainder is ignored. The instruction works correctly even when the source and destination registers are the same.  
Example: div 8 r10 r11  
div 8 r10 r10
nop (No OPeration)
nop
nop does not have any operands. The values of all 16 registers are displayed. Example: nop

hlt (HaLT) hlt
hlt does not have any operands. The processor halts. Example: hlt.
'Use the "$stop" Verilog command to halt the processor.

<table>
<thead>
<tr>
<th>comparisons</th>
<th>symbols</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than (signed)</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal (signed)</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal (signed)</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than (signed)</td>
<td></td>
</tr>
<tr>
<td>&lt;!</td>
<td>less than (unsigned)</td>
<td></td>
</tr>
<tr>
<td>&gt;=!</td>
<td>greater than or equal (unsigned)</td>
<td></td>
</tr>
<tr>
<td>&lt;=!</td>
<td>less than or equal (unsigned)</td>
<td></td>
</tr>
<tr>
<td>&gt;!</td>
<td>greater than (unsigned)</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>and is zero (first &amp; second) == 0</td>
<td></td>
</tr>
<tr>
<td>&amp;!=</td>
<td>and is non-zero (first &amp; second) &lt;&gt;0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>or is zero</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>or is non-zero</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>signs are equal    sign(first) == sign(second)</td>
<td></td>
</tr>
<tr>
<td>^!=</td>
<td>signs are not equal</td>
<td></td>
</tr>
</tbody>
</table>