CPU design from ISA

You need to be alert to (usually minor) changes that may be made to the assignment statement or to the guidelines after the assignment is first put up. Refresh this frame and re-read the assignment carefully before you make your final submission.

ISA specification

1. There are 8 general-purpose registers, r0 to r7.
2. All the registers are 16-bit in size.
3. The memory address is 16-bit; memory width is 16 bit.
4. The instructions can be either 16-bit or 32-bit in size.
5. There are four status flags, zero (Z), carry (C), overflow (V) and sign (S) which are affected by the arithmetic and logical instructions and used by jump instructions.
6. Each instruction has at most two operands.

   General assembly instruction format:

op-code	*operand 1*	*operand 2*

7. The addressing modes (for operand 2 only) are:
   - Immediate (with 16-bit operand)
   - Register
   - Base-Indexed addressing (with 16-bit displacement)
   - Base addressing (with 16-bit displacement)
   - (Memory) Indirect (with 16-bit displacement)
   - PC relative (with 16 bit displacement)
8. The instruction set is as follows:
   1. Load and store: ld, st with a 16-bit displacement (depending upon the addressing mode).

      The *dst* operand for the load instruction and the *src* operand for the store instruction is always in register mode.

```
Load -- General assembly instruction format:
ld   dst src
li   r5, #100         // r5 ← 100 i.e., r5 ← M[PC]; PC ← PC+2, where M[PC] is
                      // the second word of the instruction
                      // (since PC is incremented in the fetch phase)
                      // -- load immediate
lr   r5, r7           // r5 ← r7 -- load register
lx   r5, 10(r1, r7)   // r5 ← M[r1+ r7 + 10] i.e.,
                      // r5 ← M[r1 + r7 +M[PC]]; PC ← PC+2 -- load indexed
ldn r5, @10(r1, r7)   // r5 ← M[M[r1+ r7 + 10]] i.e.,
                      // r5 ← M[M[r1+ r7 + M[PC]]]; PC ← PC+2
                      // -- load indirect
Store -- General assembly instruction format:
st   dst src          // only two addressing modes are needed
                      // -- Indexed and Indirect
stx -5(r2), r3        // M[r2-5] ← R3 -- base addressing
stn @-5(r2), r3       // M[M[r2-5]] ← R3 i.e.,
                      // M[M[r2 + M[PC]]]; PC ← PC+2 -- indirect
```

ii. Arithmetic and logical instructions: add, sub, and, or, mns, cmp using two's complement arithmetic

The *dst* operand for these instructions is always in register mode.

General assembly instruction format: *op-code* | *dst/src 1* | *src 2*

All addressing modes supported for *src 2* for two operand instructions.

```
addi  r1, #43             // r1 ← r1 + 43 add immediate
subn  r4, @-120(r2, r6)   // r4 ← r4 - M(M(r2 + r6 -120)) -- subtract indirect
mnsr  r2, r6              // just compute r2 - r6 -- result not saved
cmp   r3                  // only one register operand instruction
                          // src2 is don't care
                          // r3 ← 2's complement of r3
```

Comparison is accomplished by subtraction. All these instructions set the status flags.

iii. Jump instructions:

- j (jump unconditionally),
- jz (jump on zero (flag set), jnz (jump on not zero),
- jc (jump on carry), jnc (jump on not carry),
- jv (jump on overflow), jnv (jump on not overflow),
- jm (jump on minus (sign flag set)), jnm (jump on not minus),

General assembly instruction format: *op-code* | *dst*

It's a two word instruction where the second word contains a signed displacement relative to the PC. (So only one addressing mode allowed.) Interpretation:

```
j    addr // if (true) PC ← PC + M[PC]
jnm  addr // if (not S) PC ← PC + M[PC], where "not S"
          // indicates that the result of last ALU
          // operation was not negative.
```

The rest are similar.

iv. Subroutine call and return: jal (jump and link), jr (jump to return)

Assembly instruction format: jal | *link register* | *subroutine address*

Interpretation:

```
jal r5, sub // r5 ← PC+2; PC ← PC + M[PC], where M[PC] (the
            // second word) contains the address of the first
            // instruction of the subroutine "sub" relative
            // to the PC -- better is to permit it to be base
            // register relative providing for longer jumps --
            // but for simplicity let's have PC relative.
```

Return instruction: Single operand

Assembly instruction format: jr | *link register*

Interpretation:

```
jr r5 // PC ← r5
```

Assignment statement

1. Design a suitable instruction format for each of the instructions.

   ### Guidelines

   1. identify the fields in the (first) instruction word -- (the second word, if any, is always a flat signed 16 bit data/address offset) -- may consider the following fields:
      - op code (for broad instruction category)
        - data movement (ld, st),
        - alu (add, sub, and, or, mns, cmp),
        - control transfer (all variations of jump)
        - subroutine call and return
      - operand 1 -- only a GPR needs to be specified
      - operand 2 -- with subfields:

        mode/cc -- addressing mode/status condition code
        base reg --
        index reg --

      (Thus, note that although for uniformity of assembly level specification of the load and store instruction, the first operand has been kept as the destination, in the m/c instruction format, this information may be provided through the "operand2" field, for uniformity of design.)

      Suggested time: half a lab day

2. Design the data path structurally and give the interpretation of each instruction over the data path by RTL micro-operations (microinstructions); encode the design in structural Verilog as a module.

   ### Guidelines

   i. Try to keep information transparent to the controller as much as possible -- for example, the specific alu operation, specific register operands (including those in the base/index field), specific condition code for the jump instructions, etc., should be tried to be kept transparent.
   ii. Follow a top-down approach -- structure should comprise a register file having read port(s) and write port(s), ALU, condition detectors with flag flip-flops, condition multiplexers (to keep the exact condition code transparent to the controller), other registers for memory operations, PC, etc.; interconnections of various fields of the instruction (register) with the read-write ports of the register file, ALU operation control inputs, the condition selection inputs of the condition multiplexer (and the op-code decoder inputs -- this last one can also be in the controller itself)
   iii. the internal bus organization of the CPU -- for this consider that the buses carry data to and from the ALU and the registers -- accordingly decide how many data buses you would keep -- one or two. The number of buses would, in turn, decide whether additional buffer registers at the ALU input(s) and output are required or not; e.g., a three bus organization would not need any buffer. (More the number of buses, faster is the execution of the micro-operations but more complication in routing and more area.) Note that your decision regarding the number of internal buses in the CPU datapath will decide the number of read / write ports of the register file (in the earlier step).
   iv. write the interpretations of the instructions, that is, RTL micro-operations of fetch phase and execute phase of EACH instruction (with all the addressing modes) to ensure that the datapath indeed supports all the instruction interpretation -- if not, enhance the datapath design as needed,
   v. Identify the (control) inputs to the datapath -- list them as bits of a "control word",
   vi. Identify the outputs of the datapath,

   Suggested time: two lab days and a half

3. Design the controller for the processor by giving a state transition diagram. Encode the controller FSM in behavioural Verilog as a module. It should be possible to reset the CPU through the controller.

   ### Guidelines

   Make your design modular, that is, the FSM should have submachines such as those
   i. for memory read which can be shared among the fetch phase and instruction operand/displacement read phase in the execution cycle of each instruction that needs memory referencing,
   ii. for memory write which can be shared among all the instructions that need to write in the memory, etc
   iii. provide for slower memory than CPU clock. The state transition diagrams of all the submachines should be reported.

   Suggested time: one lab day

4. Give a state based behavioural encoding of the controller FSM having functional modules for the submachines for:
    i. memory read which can be shared among the fetch phase and instruction operand/displacement read phase in the execution cycle of each instruction that needs memory referencing,
    ii. for memory write which can be shared among all the instructions that need to write in the memory, etc.

    Suggested time: one lab day

5. Design a testbench for the controller FSM -- it should be able to issue as inputs to the controller

    ⟨instruction op-code (only), status bit(s), mode bits⟩

    The outputs of the controller can be examined for correctness by way of display from simulator and comparison (within the testbench) with the expected output.

    If the testing is also done on the FPGA kit, then the sequence of control words generated by the controller "finally", may be recored in the memory of the FPGA kit and that sequence can be later examined manually (from the FPGA kit memory).

    Through a testbench, test the entire design, step by step, for each instruction and then for a programme.

    Suggested time: one lab day

6. Give a Verilog STRUCTURAL encoding of the datapath having modules such as "register file", "ALU" (adder/subtractor built from the CLA designed by you, the two's complementor using the subtractor, and separate modules for AND, and OR) "condition detectors" and "condition flag registers", PC, memory interface, instruction decoder, etc.

    Suggested time: two lab days

7. Through a test bench, test the entire design step by step for each instruction and then for a programme. For this you can use the FPGA kit memory to store the instruction or program containing the intructions to be tested into the FPGA kit memory which can be examined manually "after the program is over" --how to realize this condition under quote -- do we not need "halt" instruction for this?

    Suggested time: two lab days

Total lab days -- 10 (5 weeks)

Submissibles

1. A PDF file to the site depicting:
    i. all the design steps 1 -- 3 (in the above sequence) with complete schematic/logic diagrams at suitable places,
    ii. the instruction decoder,
    iii. the internal schematic diagram of the register file showing the internal blocks, the circuit for selecting the register(s) specified by the input/output port(s) and the datapath from the register file input to the data input lines of the selected register(s) and from the selected register's data output lines to the register file's data output lines (step 6),
    iv. the internal schematic of the ALU showing the blocks used in step 6,
    v. the condition detector logic circuits,
    vi. the condition flag register loading logic,
    vii. the instruction decoder
2. The Verilog behavioural encoding of the controller along with its test bench (in PDF on to the site),
3. The Verilog structural encoding of the datapath (in PDF on to the site)
4. The test bench for the entire processor

First round of submission

1. Instruction format paper design (plain text/PDF)

Second round of submission

1. Datapath paper design with diagrams (PDF). Diagram should be neat, if necessary, use fig file of class room design as starting point.
2. Interpretation of the instructions over the datapath (in plain text).

Third round of submission

1. Refined datapath diagram where each datapath component is shown with associated control signal and register bank design is depicted structurally (pdf).
2. Verilog coding of the refined datapath (each individual register, decoder, multiplexer and demultiplexer may be defined behaviourally as separate components and instantiated within the structural verilog specification) (verilog text with division of work for module design between group members)
3. Specification of the controller corresponding to the refined datapath clearly indicating the steps of instruction execution (fetch, decode, operand fetch, alu operation, etc.), where common sequences of (concurrent) microperations between instructions are shared. (plain text)

Final round of submission

1. Verilog of CPU controller
2. Verilog of CPU datapath testbench
3. Verilog of CPU controller testbench
4. Verilog of overall CPU
5. Verilog of overall CPU testbench