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:
   i. 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. Modify the instruction set architecture (ISA) of the CPU given above so that all the instructions in the instruction set can be executed on a single cycle datapath having a separate instruction memory and one data memory module and only one additional adder module over and above the ALU.
2. Implement a single cycle datapath for the modified ISA satisfying satisfying the aforementioned resource constraints.
3. Design the controller for the processor.

Submissibles

First round of submission (one lab day)

1. Modified ISA specs with clear indication and brief justification of each of the modifications (plain text)

Second round of submission (three lab days)

1. Datapath paper design with diagrams (PDF). Diagram should be neat and drawn using suitable software.
2. Verilog structural coding of the datapath
3. Verilog test bench for data path

Third round of submission (two lab days)

1. Verilog behavioural coding of the controller
2. Verilog test bench for the controller

Final round of submission (one lab day)

1. Overall test bench of CPU demonstration execution of a simple program involving all the instructions