COL216 Computer Architecture **Minor Test** **17.02.2022**

1. Consider the ARM assembly function "Freq" given below that finds the frequency of occurrence of each character in a given null terminated string (solution to Q3 of Quiz1). Characters in the string consist of 7-bit code and a parity bit. Address of the string and the result array are passed on to the function in registers r0 and r1.

```
    .global Freq
    .text
Freq:
    mov r2, #1             // index for frequency array
    mov r3, #0             // constant 0 to initialize the frequency array
Loop0:
    str r3, [r1, r2, LSL #2]     // initialization
    add r2, r2, #1               // next index
    cmp r2, #127
    blt Loop0
Loop1:
    ldrb r2, [r0], #1            // load byte from string
    ands r2, r2, #0x7f           // ignore parity bit
    beq Done                     // check for null
    ldr r3, [r1, r2, LSL #2]
    add r3, r3, #1               // update frequency
    str r3, [r1, r2, LSL #2]
    b Loop1
Done:   mov pc, lr               // return
    .end
```

Write a function "Maxfreq" by augmenting/modifying this function in the following ways.

a) After finding frequencies of occurrence of various characters, a function "Max" is called to find which character has the maximum frequency.
b) Argument passed to "Maxfreq" is starting address of the string in register r0. Code of the character with maximum frequency is returned in register r0.
c) Frequencies of occurrence are stored in an array allocated on stack by Maxfreq. This array is disposed off before returning from the function.

Write function "Max" assuming that it is in a separate file. Arguments passed to "Max" are - starting address of frequency array in register r0 and size of this array in register r1. Code of the character with maximum frequency is returned in register r0.

[6]

Solution:

```
     .global Maxfreq         // changes are shown in blue
     .extern Max
     .text
 Maxfreq:
     str lr, [sp, #-4]!      // save lr on stack
     mov r1, r0
     sub sp, sp, #512        // allocate array on stack
     mov r2, #1              // index for frequency array
     mov r3, #0              // constant 0 to initialize the frequency array
 Loop0:
     str r3, [sp, r2, LSL #2]      // initialization
     add r2, r2, #1                // next index
     cmp r2, #127
     blt Loop0
 Loop1:
     ldrb r2, [r0], #1             // load byte from string
     ands r2, r2, #0x7f            // ignore parity bit
     beq Done                      // check for null
     ldr r3, [sp, r2, LSL #2]      // sp is the base address of array now
     add r3, r3, #1                // update frequency
     str r3, [sp, r2, LSL #2]
     b Loop1
 Done:
     mov r0, sp
     mov r1, #128
     bl Max                        // call to Max function
     add sp, sp, #512              // de-allocate array
     ldr pc, [sp], #4              // return
     .end
```

```
     .global Max
     .text
Max:
     mov r2, #1
     mov r3, #0                    // max freq so far
Loop2:
     ldr r4, [r0, r2, LSL #2]      // load next frequency value
     cmp r4, r3                    // compare it with max so far
     movgt r5, r2                  // update index of max value
     movgt r3, r4                  // update max value
     add r2, r2, #1
     cmp r2, r1                    // loop termination check
     blt Loop2
     mov pc, lr                    // return
     .end
```

Marking scheme:

Changes to function Freq (3 marks)

Global and extern declaration (0.5), saving lr on stack and correct return (0.5), array allocation/de-allocation on stack (1), call to Max with correct arguments (1).

Max function (3 marks)

Global declaration (0.5), correct loop structure including initialization and termination (1), correct loop body including comparison and update (1.5).

2. Write VHDL description of a simple computer which can execute a subset of 9 ARM instructions {`add, sub, cmp, mov, ldr, str, b, beq, bne`} with limited variants/features as described below.

- Instructions {`add, sub, cmp`} have both operands as registers, whereas instruction {`mov`} has immediate operand. Shift / rotate features are not available.
- Instructions {`ldr, str`} have immediate offset only (byte offset). Only word transfers, no write back, only pre-indexing. Addresses generated by adding/subtracting contents of `Rn` and `Offset` are byte addresses, but are word aligned.
- There is only one Flag - `Z`. It can only be modified by {`cmp`} instruction and can only be tested by {`beq, bne`} instructions.
- Assume that the Program Counter (`PC`) is a separate register and has nothing to do with `R15` of register file.
- The offsets in instructions {`b, beq, bne`} are word offsets, with respect to `PC` + 8.
- Assume that Program Memory and Data Memory have independent address spaces. That is, both have byte addresses 0 to 255 (or word addresses 0 to 63).

Formats of the instructions are shown below. Bits shown as `'x'` are don't care. Assume that the program memory contains only valid instructions as per these formats.

`add Rd, Rn, Rm`

xxxx	00	x	01xx	x	Rn	Rd	xxxxxxxx	Rm
4	2	1	4	1	4	4	8	4

`sub Rd, Rn, Rm`

xxxx	00	x	00xx	x	Rn	Rd	xxxxxxxx	Rm
4	2	1	4	1	4	4	8	4

`cmp Rn, Rm`

xxxx	00	x	10xx	x	Rn	xxxx	xxxxxxxx	Rm
4	2	1	4	1	4	4	8	4

`mov Rd, #Imm`

xxxx	00	x	11xx	x	xxxx	Rd	xxxx	Imm
4	2	1	4	1	4	4	4	8

`ldr Rd, Rn, #Offset` `{add offset if U=1, else subtract}`

xxxx	01	xxUxx1	Rn	Rd	Offset
4	2	6	4	4	12

`str Rd, Rn, #Offset` `{add offset if U=1, else subtract}`

xxxx	01	xxUxx0	Rn	Rd	Offset
4	2	6	4	4	12

`b #Signed_Offset`

xx10	10	xx	Signed_Offset
4	2	2	24

`beq #Signed_Offset`

xx00	10	xx	Signed_Offset
4	2	2	24

`bne #Signed_Offset`

xx01	10	xx	Signed_Offset
4	2	2	24

Each instruction is to be executed in a single clock cycle. Make a provision for innitializing program counter zero on an external reset signal. Unlike Lab assignment 2, describe the design as a single entity-architecture pair, with no external components. Model the register file and memories as arrays local to the architecture. Minimization of hardware resources is not required. . Assume availability of the following concurrent assignments in the architecture (these need not be included in the answer, also omit declaration of the signals appearing on the left hand side of these assignments). Instr is the signal carrying instruction.

```
F <= Instr (27 downto 26); OP <= Instr (24 downto 23); Cond <= Instr (29 downto 28);
Ubit <= Instr (23); Lbit <= Instr (20);
Imm <= Instr (7 downto 0); Offset <= Instr (11 downto 0); S_offset <= Instr (23 downto 0);
Rd <= to_integer (unsigned(Instr (15 downto 12)));
Rn <= to_integer (unsigned(Instr (19 downto 16)));
Rm <= to_integer (unsigned(Instr (3 downto 0)));
```

[8]

Solution:

```
entity processor is
   port (
      clock, reset : in std_logic
   );
end processor;
architecture behavioral of processor is
   signal PC, Instr : word;
   signal Zflag, predicate : std_logic;
   type RF_type is array (0 to 15) of word;
   signal RF : RF_type;
   type memory_type is array (0 to 63) of word;
   signal PM, DM : memory_type;
   signal PMadr, DMadr : integer range 0 to 63;
   signal DMadrv : signed (31 downto 0);
   signal Sext : std_logic_vector (5 downto 0);
begin
   PMadr <= to_integer (unsigned(PC (7 downto 2)));
   Instr <= PM (PMadr);
   with Cond select
      predicate <= '1'         when "10",
                   Zflag       when "00",
                   not Zflag when "01",
                   '0'          when others;
   DMadrv <= (signed(RF(Rn)) + signed(X"00000" & Offset)) when (Ubit = '1')
                 else (signed(RF(Rn)) – signed(X"00000" & Offset));
   DMadr <= to_integer(Dmadrv(7 downto 2));
   S_ext <= "111111" when (Instr(23) = '1') else "000000";
   process (reset, clock)
   begin
      if (reset = '1') then PC <= X"00000000";
      elsif (rising_edge(clock)) then
         if (F = "10") and (predicate = '1')) then
              PC <= std_logic_vector (signed(PC) + signed(S_ext & S_offset & "00") + 8);
           else PC <= std_logic_vector (signed(PC) + 4);
           end if;
      end if;
   end process;
   process (clock)
   begin
      if (rising_edge(clock)) then
         case F is
            when "00" =>
               case OP is
                  when "00" => RF(Rd) <= std_logic_vector(signed(RF(Rn)) – signed(RF(Rm)));
                  when "01" => RF(Rd) <= std_logic_vector(signed(RF(Rn)) + signed(RF(Rm)));
                  when "10" => if (RF(Rn) = RF(Rm)) then Zflag <= '1'; else Zflag <= '1'; end if;
                  when "11" => RF(Rd) <= X"000000" & Imm;
               end case;
            when "01" =>
               if (Lbit = '1') then RF(Rd) <= DM(DMadr); else DM(DMAdr) <= RF(Rd); end if;
         end case;
      end if;
   end process;
end behavioral;
```

Marking scheme:

Entity declaration (1), implementation of DP and DT instructions (6x0.5), implementation of branch instructions (1), checking predicate (1), PC initialization on reset (1), declaration of register file and memories (1).

3. Suppose all flags are removed from ARM processor. After performing add operation using instruction `add r0, r1, r2`, how would you find whether overflow occured or not and whether there was a carry or not? Write a sequence of instructions to put the value that flag V would have contained in register r3 and the value that flag C would have contained in register r4. [Hint: use logical and shift operations.]

[6]

Solution:

Let $a_{31} \ldots a_0$, $b_{31} \ldots b_0$ be the two operands and $s_{31} \ldots s_0$ be the sum. Then

$V = s_{31} . a_{31}' . b_{31}' + a_{31} . b_{31} . s_{31}'$

$c_{31} = s_{31}$ xor a_{31} xor b_{31}
$C = c_{32} = c_{31} . (a_{31} + b_{31}) + a_{31} . b_{31}$

```
add r0, r1, r2

bic r3, r0, r1              // r3 = r0 . r1'
bic r3, r3, r2              // r3 = r0 . r1' . r2'
and r5, r1, r2              // r5 = r1 . r2
bic r6, r5, r0              // r5 = r1 . r2 . r0'
orr r3, r3, r6              // r3 = (r0 . r1' . r2') + (r1 . r2 . r0')
mov r3, r3, LSR #31         // Get the leftmost bit in the rightmost position

xor r4, r0, r1              // r4 = r0 xor r1
xor r4, r4, r2              // r4 = r0 xor r1 xor r2          (denote by y)
orr r6, r1, r2              // r6 = r1 + r2
and r4, r4, r6              // r4 = y . (r1 + r2)
orr r4, r4, r5              // r4 = y . (r1 + r2) + (r1 . r2)
mov r4, r4, LSR #31         // Get the leftmost bit in the rightmost position
```

```
Marking scheme:

Computation of V (3 marks)
      1 mark for a fair attempt

Computation of C (3 marks)
      1 mark for a fair attempt
```