Lab Assignment 2, Stage 8: BL, SWI instructions and predication

Kushal Kumar Gupta, 2020CS10355

Implemented instructions `bl, swi, ret, rte`. Full predication and conditional flag update for DP and Multiply instructions had already been implemented in the earlier stages.

Also implemented reset, user and supervisor mode, reading a byte-wide input port. ISRs for `reset` and `swi` input have also been included in the memory.

Files, entities and Architectures:

Changes from stage 7-

`I. Mem_stage8.vhd:`

- system area is 0 to 63rd word, user area is 64 to 127th word
- Input port is given address of 63rd word
- as same memory for both data and program, ensure that there is no overlap
- in testcases, assumed that program occupies 64 to 95th word (first half of user memory)

System area memory initialization:

On a reset signal, PC goes to `0x00000000` i.e the word 0 and mode is changed to supervisor. Hence, a branch has been placed at word 0 to go to Word 8, which has the ISR for Reset. In the ISR, first LR is loaded with address of the start of user program (byte address 64 * 4 = 256) and then RTE instruction is executed to transfer control to the start of user program and also change the mode back to supervisor.

`ISR for Reset:`

```
                              8 => X"E3A0EC01",
                              9 => X"E6000011",
     -- 8 : mov r14, #256 @ISR for reset
     -- 9 : RTE
```

SWI instruction assigns PC the value `0x00000008`i.e. Word 2. Hence, here a branch has been placed to direct to go to Word 16 which has the ISR of SWI. This ISR reads the input port and loads the 9 bit result in r0 of Register File. Then RTE resumes the user program execution.

`ISR for SWI:`

```
                                                  16 => X"E3A00000",
                                                  17 => X"E59000FC",
                                                  18 => X"E6000011",
    -- 16 : mov r0, #0 @ISR for swi, input data
    -- 17 : ldr r0, [r0, #252] @read input from port, port has word address 63,
hence byte address = 63 * 4 = 252
    -- 18 : RTE
```

`II.Processor_stage8.vhd`:

Included support for `BL, SWI, RET` and `RTE` instructions. Also included an input port. Also treated reset as an exception.

`BL` saves PC+4 in LR and adds offset to PC. No change in mode.

`SWI` saves PC+4 in LR and makes PC = `0x00000008`. Mode changes to supervisor.

`RET` copies LR into PC. No change in mode.

`RTE` copies LR into PC. Mode changes to user.

Implementation details:

- `LR` is `r14` in the Register File
- For BL and SWI instructions, `pc_out` is assigned to `LR` in state 3. Offset added to PC for BL and PC is assigned `0x00000008` in SWI.
- For `RET` and `RTE` instructions, `LR` is loaded in register `A` in state 2 and then register `A` is assigned to `pc_in`, and finally `pc_out` is assigned `pc_in`
- A 9-bit std_logic_vector Input Port has been included. This port can be supplied a byte wide input in bits 8 to 0 and the 9th bit acts a status bit which should be set to 1 when data is being supplied.
- A mode flag has been included with `'0'` as User mode and `'1'` as Supervisor mode. This mode flag is updated appropriately at Reset, SWI and RTE instructions.
- Included another state 0 in FSM. FSM transitions to state 0 whenever there is a reset signal. In this state PC is assigned the value `0x00000000`.

For more details regarding the implementation of `processor`, see the architecture `processor_beh`.

`III.mytypes_stage8.vhd`:

`instr_class_type` changed to include the rest of the instructions

```
type instr_class_type is (DP, DTtype00, DTtype01, MULT, BRN, SWI, RET, RTE,
none);
```

Introduced type `BRN_instr_type`

```
type BRN_instr_type is (branch, bl);
```

`IV.Decoder_stage8.vhd`:

```
entity Decoder is
Port (
    instruction : in word;
    instr_class : out instr_class_type;
    operation : out optype;
    DP_subclass : out DP_subclass_type;
     DT_instr : out DT_instr_type;
     MULT_instr: out MULT_instr_type;
     BRN_instr: out BRN_instr_type;
    DP_operand_src : out src_type;
    load_store : out load_store_type;
     pre_post : out pre_post_type;
     DT_write_back : out write_back_type;
    DT_offset_sign : out DT_offset_sign_type;
    DT_offset_src : out src_type;
    reg_shift_type : out shift_rotate_op_type;
    reg_shift_src : out src_type
);
end Decoder;
```

`instr_class` is determined as follows:

```
    instr_class <= RET when instruction (27 downto 25) = "011" and instruction
(4 downto 0) = "10000" else
                   RTE when instruction (27 downto 25) = "011" and instruction
(4 downto 0) = "10001" else
                   DTtype01 when instruction (27 downto 26) = "01" else
                   DP when instruction (27 downto 25) = "001" else
                   DP when instruction (27 downto 26) = "00" and (instruction
(4) = '0' or instruction(7) = '0') else
                   MULT when instruction (27 downto 26) = "00" and instruction
(6) = '0' and instruction(5) = '0' else
                   DTtype00 when instruction (27 downto 26) = "00" else
                   BRN when instruction (27 downto 26) = "10"   else
                   SWI when instruction (27 downto 26) = "11" else
                   none;
```

Additionally, type of branch instruction is determined using bit 24 of instruction.

```
    BRN_instr <= branch when instruction(24) = '0' else
                 bl;
```

`V.testbench.vhd`:

Testbench has been updated to provide data to the input port of processor.

```vhdl
    testInPort: process is
        begin

        wait for 200 ns;
        input_port <= "1" & X"23"; --supply input byte 23
        wait for 1000 ns;
        input_port <= "0" & X"00";
    end process testInPort;
```

How to use:

On edaplayground.com, upload testbench.vhd and run.do in the left column, and ALU_stage3.vhd, RegFile_stage1.vhd, Mem_stage8.vhd, mytypes_stage8.vhd , decoder_stage8.vhd, FlagUpdater_stage7.vhd, conditionChecker_stage4.vhd, programCounter_stage3.vhd, processor_stage8.vhd, shifter_stage5.vhd, PMconnect_stage6.vhd, Multiplier_stage7.vhd. Copy contents in design.vhd section as given in the design.vhd file. Select   Testbench + Design as VHDL. Then, for-

1.)Simulation

Type testbench in the Top entity. Select Aldec Riviera Pro 2020.04 simulator to simulate the design. Set the run time accordingly and select the EPWave option. Then Save and Run the simulation to get waves of the signals defined in these modules.

2.)Synthesis

Copy the VHDL file of the component you want to synthesise into design.vhd. (Only design.vhd entities are synthesised)

Then, select Mentor Precision 2021.1 to synthesise. Select netlist option to view the Verilog description. Then Save and Run to get the synthesis result. We get the report containing the resource table specifying number of IOs, LUTs, CLB slices, ports, nets, etc. used to implement this module in the given FPGA specified by run.do.

Results of EPWave (Simulation):

I. SWI for data input

The following memory initialization has been done with branches to ISRs of `reset` and `swi`. The ISR for `swi` takes input from the port and stores in `r0`.

`Memory initialization:`

```
    signal Mem_space : type_mem :=
                                ( 0 => X"EA000006",
                                  2 => X"EA00000C",
                                  8 => X"E3A0EC01",
                                  9 => X"E6000011",
                                  16 => X"E3A00000",
                                  17 => X"E59000FC",
                                  18 => X"E6000011",
                                  64 => X"EF000000",
                                  65 => X"E1A01420",
                                  66 => X"E3510001",
                                  67 => X"1AFFFFFB",
                                  68 => X"E3A01000",
                                  69 => X"E5C10180",
                                   others => X"00000000"
                                   );
    -- 0 : b 6 @branch with offset 6, therefore branch to word 0 + 2 + 6 = 8
    -- 2 : b 12 @branch to word 2 + 2 + 12 = 16
    -- 8 : mov r14, #256 @ISR for reset
    -- 9 : RTE
    -- 16 : mov r0, #0 @ISR for swi, input data
    -- 17 : ldr r0, [r0, #252] @read input from port, port has word address 63,
hence byte address = 63 * 4 = 252
    -- 18 : RTE
    --user program to test the input port
    -- 64 : L: SWI @data from port has been loaded in r0
    -- 65 : mov r1, r0, LSR #8
    -- 66 : cmp r1, #1 @check if status bit is 1
    -- 67 : bne L @if data not collected try again
    -- 68 : mov r1, #0
    -- 69 : strb r0, [r1, #384]  @if succesful, store the byte at 96 *4 = 384
```

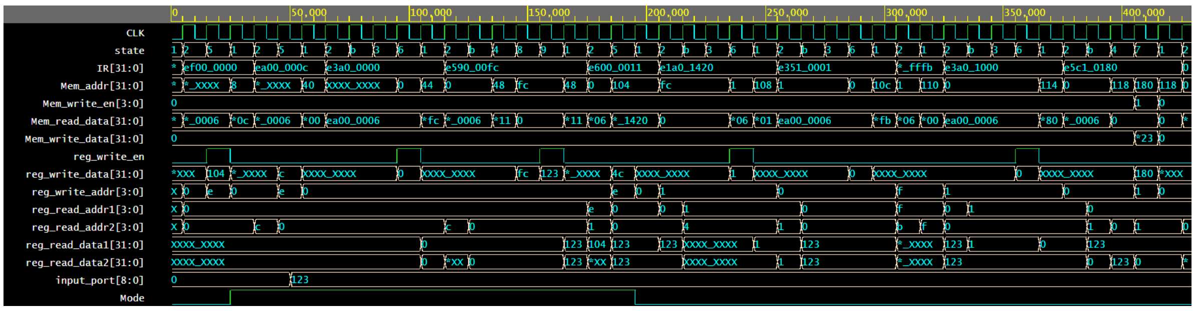
A 9-bit wide input port has been assigned status bit `1` and data `X"23"` in the testbench after 50 ns.

`Testbench:`

```
-- Code your testbench here
library IEEE;
use IEEE.std_logic_1164.all;
use work.MyTypes.all;
use IEEE.NUMERIC_STD.ALL;

entity testbench is
-- empty
end testbench;
```

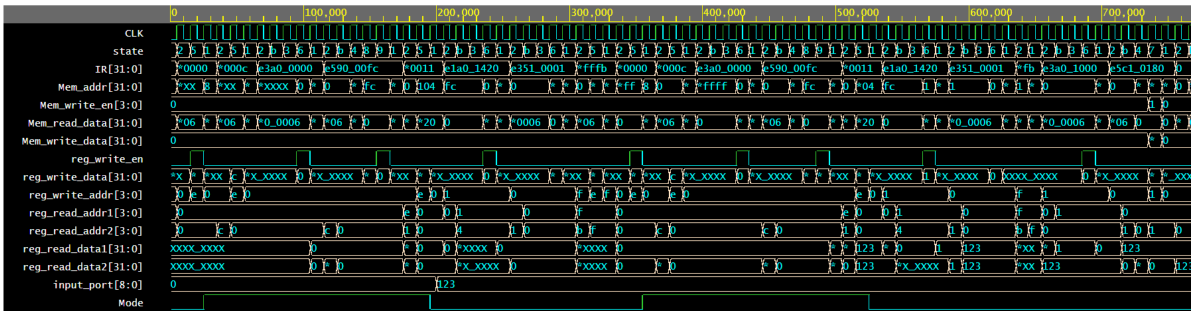
```
architecture tb of testbench is

component processor is
Port(
    CLK: in bit;
    input_port : in std_logic_vector(8 downto 0);
    reset: in bit
);
end component;

signal CLK : bit := '0';
signal input_port : std_logic_vector(8 downto 0) := "0" & X"00";
signal reset : bit := '0';

begin
    CLK <= not CLK after 5 ns;
    DUT_processor: processor port map(CLK, input_port, reset);
    testInPort: process is
        begin

        wait for 200 ns;
        input_port <= "1" & X"23"; --supply input byte 23
        wait for 1000 ns;
        input_port <= "0" & X"00";
    end process testInPort;
end tb;
```

As can be seen from the waveform, the load instruction in the `swi` ISR loads the input data in r0.



Also, we can see that if the input is assigned after 200 ns, the program loops back to `swi` again and then reads the data.

```
    testInPort: process is
        begin

        wait for 200 ns;
        input_port <= "1" & X"23"; --supply input byte 23
        wait for 1000 ns;
        input_port <= "0" & X"00";
    end process testInPort;
```



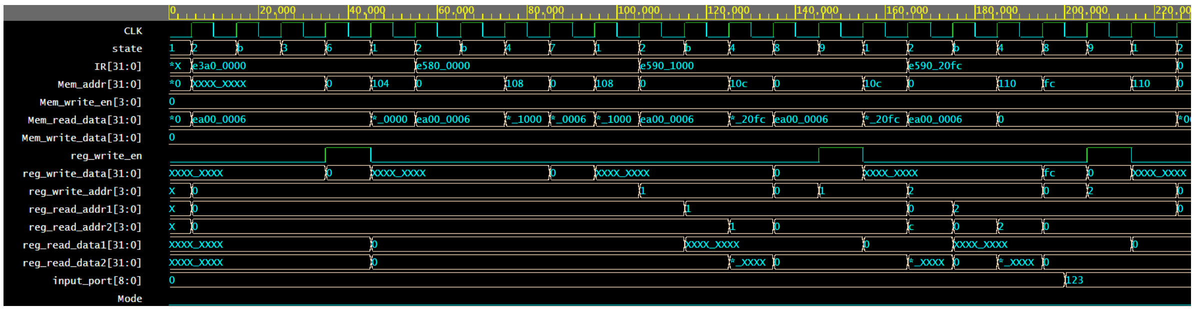
II. Memory protection

User can't access the system memory or the input port.

The following program illustrates this:

```
    signal Mem_space : type_mem :=
                                    ( 0 => X"EA000006",
                                      2 => X"EA00000C",
                                      8 => X"E3A0EC01",
                                      9 => X"E6000011",
                                      16 => X"E3A00000",
                                      17 => X"E59000FC",
                                      18 => X"E6000011",
                                      64 => X"E3A00000",
                                      65 => X"E5800000",
                                      66 => X"E5901000",
                                      67 => X"E59020FC",
                                      others => X"00000000"
                                       );
    --64: mov r0, #0
    --65: str r0, [r0] @user trying to write in system memory
    --66: ldr r1, [r0] @user trying to read from system area
    --67: ldr r2, [r0, #252] @user trying to read input port
```

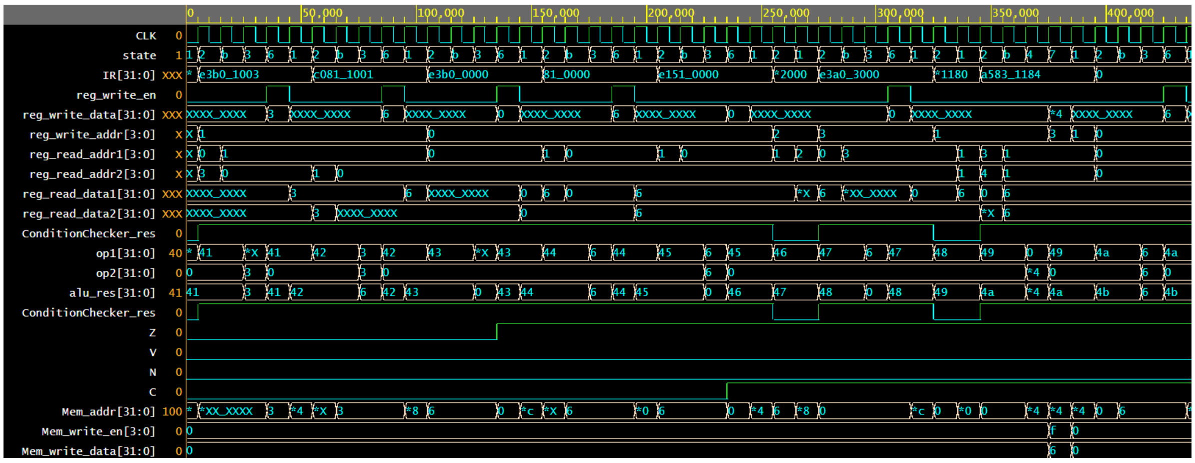
As can be seen from the waveform, memory write enable has been disabled when user tries to write in system area and 0 is given as output from memory if user tries to read from system area.



III. Full predication and S-bit capability

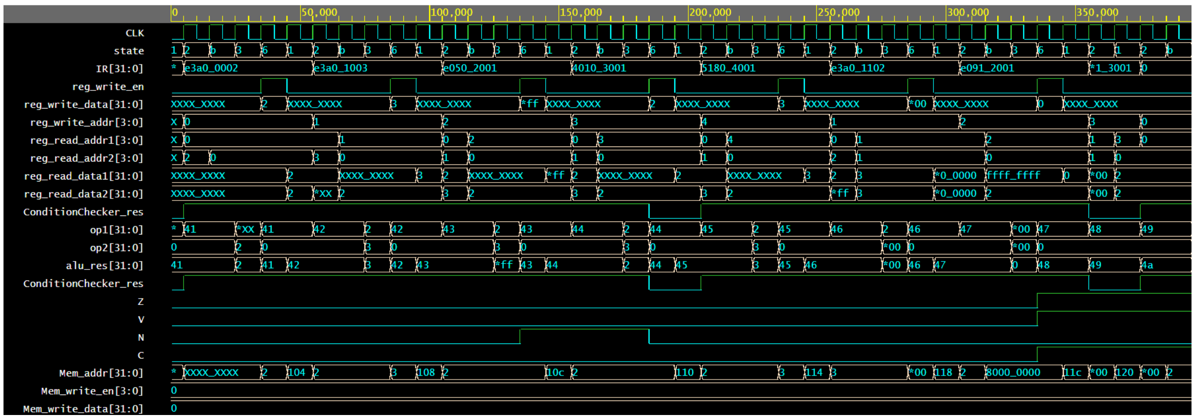
1.

```
signal Mem_space : type_mem :=
                             ( 0 => X"EA000006",
                             2 => X"EA00000C",
                             8 => X"E3A0EC01",
                             9 => X"E6000011",
                             16 => X"E3A00000",
                             17 => X"E59000FC",
                             18 => X"E6000011",
                             64 => X"E3B01003",
                             65 => X"C0811001",
                             66 => X"E3B00000",
                             67 => X"00810000",
                             68 => X"E1510000",
                             69 => X"10412000",
                             70 => X"E3A03000",
                             71 => X"B5831180",
                             72 => X"A5831184",
                             others => X"00000000");

    --64 movs r1, #3
    --65 addgt r1, r1, r1 @condition true
    --66 movs r0, #0
    --67 addeq r0, r1, r0 @condition true
    --68 cmp r1, r0
    --69 subne r2, r1, r0 @condition false
    --70 mov r3, #0
    --71 strlt r1, [r3, #384] @condition false
    --72 strge r1, [r3, #388] @condition true
```



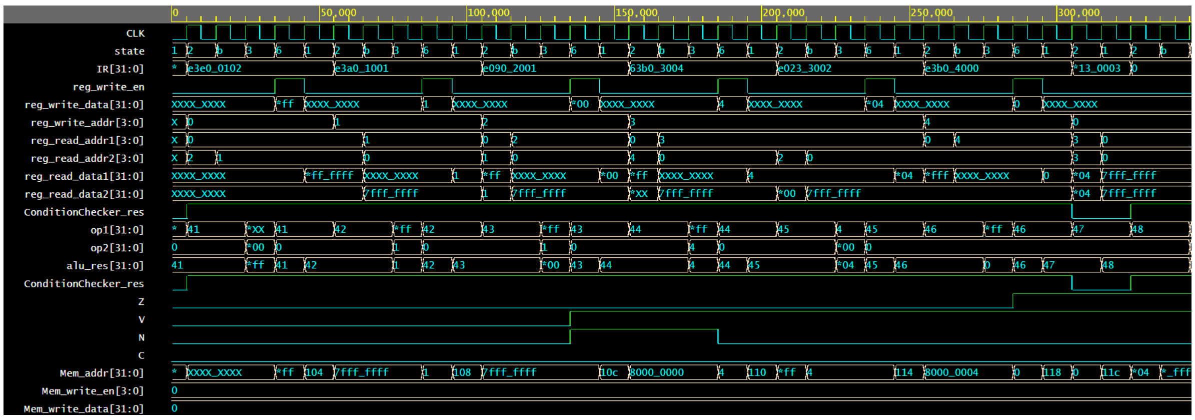
2.

```
signal Mem_space : type_mem :=
                              ( 0 => X"EA000006",
                              2 => X"EA00000C",
                              8 => X"E3A0EC01",
                              9 => X"E6000011",
                              16 => X"E3A00000",
                              17 => X"E59000FC",
                              18 => X"E6000011",
                              64 => X"E3A00002",
                              65 => X"E3A01003",
                              66 => X"E0502001",
                              67 => X"40103001",
                              68 => X"51804001",
                              69 => X"E3A01102",
                              70 => X"E0912001",
                              71 => X"30013001",
                              others => X"00000000");

    --64 mov r0, #2
    --65 mov r1, #3
    --66 subs r2, r0, r1
    --67 andmis r3, r0, r1 @condition true as negative (MI)
    --68 orrpl r4, r0, r1 @condition true as positive(PL)
    --69 mov r1, #0x80000000
    --70 adds r2, r1, r1
    --71 andcc r3, r1, r1 @condition false as carry is set
```



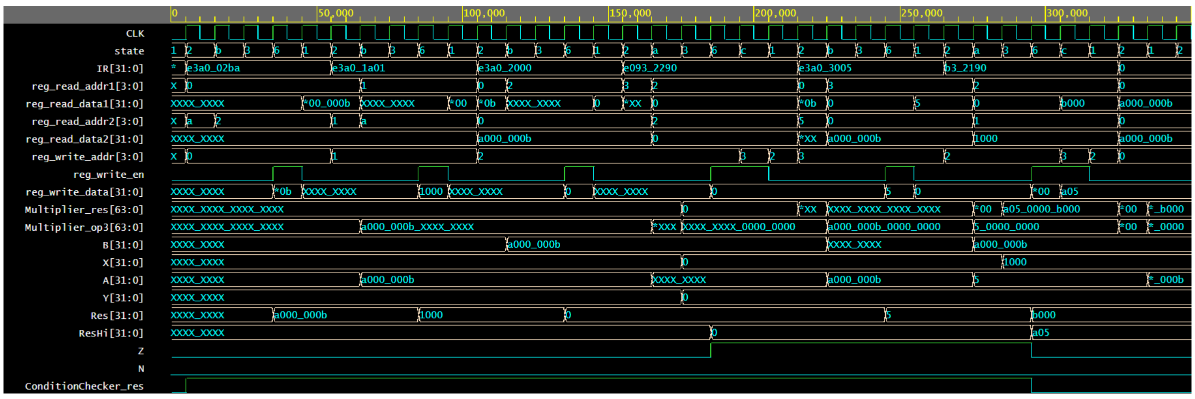
3.

```
signal Mem_space : type_mem :=
                            ( 0 => X"EA000006",
                            2  => X"EA00000C",
                            8  => X"E3A0EC01",
                            9  => X"E6000011",
                            16  => X"E3A00000",
                            17  => X"E59000FC",
                            18  => X"E6000011",
                            64  => X"E3E00102",
                            65  => X"E3A01001",
                            66  => X"E0902001",
                            67  => X"63B03004",
                            68  => X"E0233002",
                            69  => X"E3B04000",
                            70  => X"11130003",
                            others  => X"00000000");

    --64 mov r0, #0x7fffffff
    --65 mov r1, #1
    --66 adds r2, r0, r1
    --67 movvss r3, #4 @overflow is set, hence condition true
    --68 eoral r3, r3, r2 @al is always true
    --69 mvns r4, #0xffffffff
    --70 tstne r3, r3 @condition false
```



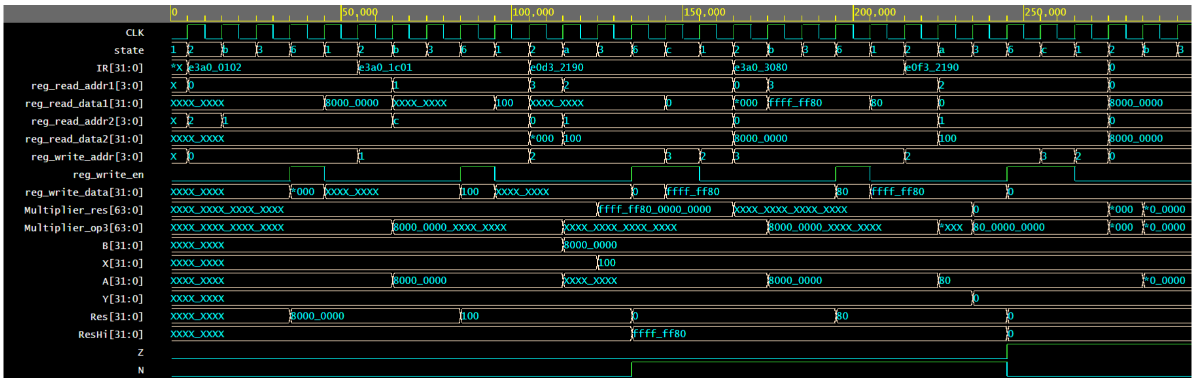
4.

```
    signal Mem_space : type_mem :=
                                    ( 0  => X"EA000006",
                                      2  => X"EA00000C",
                                      8  => X"E3A0EC01",
                                      9  => X"E6000011",
                                      16  => X"E3A00000",
                                      17  => X"E59000FC",
                                      18  => X"E6000011",
                                      64  => X"E3A002BA",
                                      65  => X"E3A01A01",
                                      66  => X"E3A02000",
                                      67  => X"E0932290",
                                       68  => X"E3A03005",
                                       69  => X"00B32190",

                                      others => X"00000000"
                                       );
--  mov r0, #0xA000000B
--  mov r1, #0x1000
--  mov r2, #0
--  umulls r2, r3, r0, r2 @result should be 0
--  mov r3, #5
--  umlaleqs r2, r3, r0, r1 @result should be r3 = 00000A05 r2 = 0000B000
```



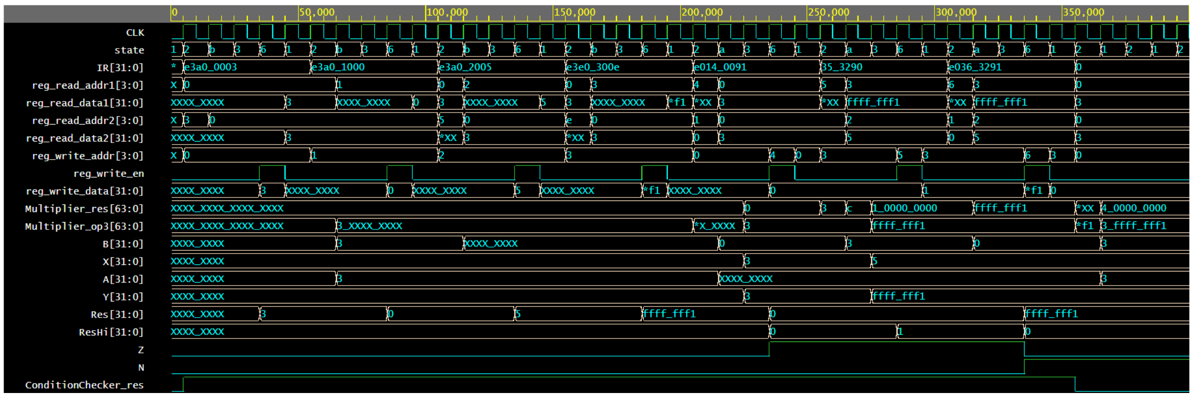
5.

```
    signal Mem_space : type_mem :=
                                  ( 0  => X"EA000006",
                                    2  => X"EA00000C",
                                    8  => X"E3A0EC01",
                                    9  => X"E6000011",
                                    16  => X"E3A00000",
                                    17  => X"E59000FC",
                                    18  => X"E6000011",
                                    64  => X"E3A00102",
                                    65  => X"E3A01C01",
                                    66  => X"E0D32190",
                                    67  => X"E3A03080",
                                     68  => X"E0F32190",

                                    others => X"00000000"
                                     );

--  mov r0, #-0x80000000
--  mov r1, #0x100
--  smulls r2, r3, r0, r1 @result is -2^31 * 256 = FFFF FF80 00000000
--  mov r3, #-0xFFFFFF80
--  smlals r2, r3, r0, r1 @ result is 0
```



6.

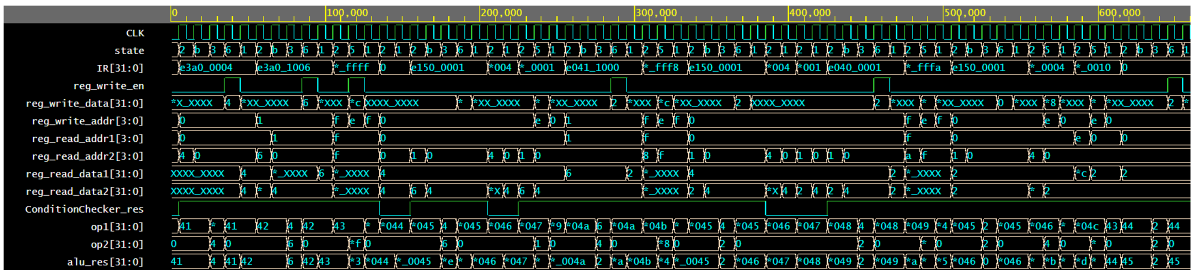
```
    signal Mem_space : type_mem :=
                                            ( 0  => X"EA000006",
                                              2  => X"EA00000C",
                                              8  => X"E3A0EC01",
                                              9  => X"E6000011",
                                              16  => X"E3A00000",
                                              17  => X"E59000FC",
                                              18  => X"E6000011",
                                              64  => X"E3A00003",
                                               65  => X"E3A01000",
                                               66  => X"E3A02005",
                                               67  => X"E3E0300E",
                                               68  => X"E0140091",
                                               69  => X"00353290",
                                               70  => X"E0363291",

                                              others  => X"00000000"
                                               );

--  mov r0, #3
--  mov r1, #0
--  mov r2, #5
--  mov r3, #-15
--  muls r4, r1, r0
--  mlaeqs r5, r0, r2, r3
--  mlas r6, r1, r2, r3
```



IV: BL and RET:

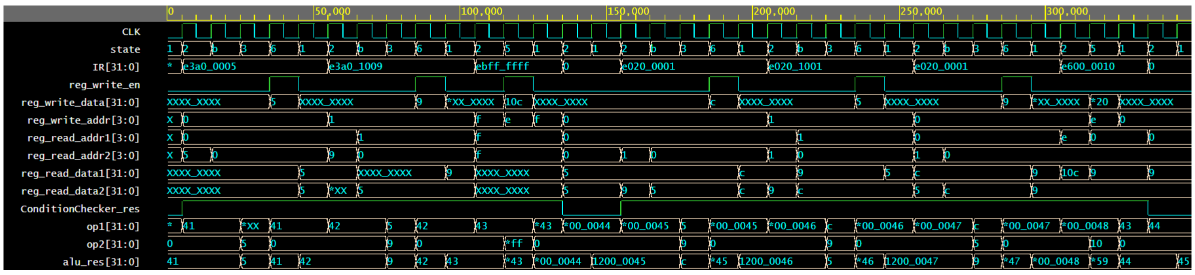
1.

```
    signal Mem_space : type_mem :=
                                ( 0 => X"EA000006",
                                  2 => X"EA00000C",
                                  8 => X"E3A0EC01",
                                  9 => X"E6000011",
                                  16  => X"E3A00000",
                                  17  => X"E59000FC",
                                  18  => X"E6000011",
                                  64  => X"E3A00004",
                                   65  => X"E3A01006",
                                   66  => X"EBFFFFFF",

                                   68  => X"E1500001",
                                   69  => X"0A000004",
                                   70  => X"BA000001",
                                   71  => X"E0400001",

                                   72  => X"EAFFFFFA",

                                   73  => X"E0411000",

                                   74  => X"EAFFFFF8",

                                   75  => X"E6000010",

                                  others => X"00000000"
                                   );

    --code for finding GCD of r0 and r1 and store result in r2
    --64 mov r0, #4
    --65 mov r1, #6
    --66 bl gcd

    --68 gcd: cmp r0, r1
    --69 beq B
    --70 blt C
    --71 sub r0, r0, r1
    --72 b gcd
    --73 C: sub r1, r1, r0
    --74 b gcd
    --75 B: ret
```



2.

```
signal Mem_space : type_mem :=
                                        ( 0 => X"EA000006",
                                        2 => X"EA00000C",
                                        8 => X"E3A0EC01",
                                        9 => X"E6000011",
                                        16 => X"E3A00000",
                                        17 => X"E59000FC",
                                        18 => X"E6000011",
                                        64 => X"E3A00005",
                                        65 => X"E3A01009",
                                        66 => X"EBFFFFFF",

                                        68 => X"E0200001",
                                        69 => X"E0201001",
                                        70 => X"E0200001",
                                        71 => X"E6000010",
                                        others => X"00000000");

--code for swapping registers r0 and r1
-- mov r0, #5
-- mov r1, #9
-- bl abs
-- abs: eor r0, r0, r1
-- eor r1, r0, r1
-- eor r0, r0, r1
-- ret
```



V. Reset

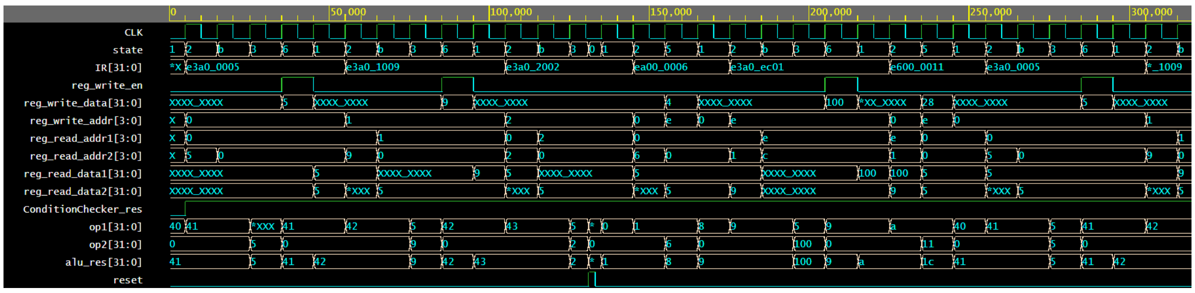
When reset signal is passed by the testbench, Reset ISR is invoked and PC goes back to the start of the program.

`Testbench:`

```
-- Code your testbench here
library IEEE;
use IEEE.std_logic_1164.all;
use work.MyTypes.all;
use IEEE.NUMERIC_STD.ALL;

entity testbench is
-- empty
end testbench;

architecture tb of testbench is

component processor is
Port(
    CLK: in bit;
    input_port : in std_logic_vector(8 downto 0);
    reset: in bit
);
end component;

signal CLK : bit := '0';
signal input_port : std_logic_vector(8 downto 0) := "0" & X"00";
signal reset : bit := '0';

begin
    CLK <= not CLK after 5 ns;
    DUT_processor: processor port map(CLK, input_port, reset);
    testInPort: process is
        begin
            wait for 126 ns;
            reset <= '1';
            wait for 2 ns;
            reset <= '0';
            wait for 1000 ns;
    end process testInPort;
end tb;
```

`Sample user program:`

```
signal Mem_space : type_mem :=
                            ( 0 => X"EA000006",
                            2 => X"EA00000C",
                            8 => X"E3A0EC01",
                            9 => X"E6000011",
                            16 => X"E3A00000",
                            17 => X"E59000FC",
                            18 => X"E6000011",
                            64 => X"E3A00005",
                            65 => X"E3A01009",
                             66 => X"E3A02002",
                             67 => X"E3A03006",
                             others => X"00000000");

    -- 64 mov r0, #5
    -- 65 mov r1, #9
    -- 66 mov r2, #2
    -- 67 mov r3, #6
```


CLK
state
IR[31:0]
reg_write_en
reg_write_data[31:0]
reg_write_addr[3:0]
reg_read_addr1[3:0]
reg_read_addr2[3:0]
reg_read_data1[31:0]
reg_read_data2[31:0]
ConditionChecker_res
op1[31:0]
op2[31:0]
alu_res[31:0]
reset
e3a0_0005
e3a0_1009
e3a0_2002
ea00_0006
e3a0_ec01
e600_0011
e3a0_0005