Lab Assignment 2, Stage 3: Multi-cycle processor design

Kushal Kumar Gupta

2020CS10355

This stage involves modifying the design of stage 2 to make it multi-cycle design. The set of instructions to be executed and the variants to be supported remain same.

Files, entities and Architectures:

1. **ALU_stage3.vhd contains the entity ALU**

   `ALU`: ALU circuit which takes as input-

   I.)`opcode`: one of the following 16 DP opcode of type `optype`.

   `optype` is an enumerated type with DP opcodes:

   `andop, eor, sub, rsb, add, adc, sbc, rsc, tst, teq, cmp, cmn, orr, mov, bic, mvn`

   II.)`op1, op2`: the two operands, as 32-bit std_logic_vectors

   III.)`carry_in`: carry input as a std_logic

   And outputs:

   I.)`res`: the 32-bit std_logic_vector result of the operation specified by the opcode on op1 and op2

   II.) `carry_out`: carry output as a std_logic

```
entity ALU is
port(
    opcode: in optype;
    op1: in std_logic_vector(31 downto 0); --Inputs: opcode
specifying the DP instruction, operands 1 and 2, carry in
    op2: in std_logic_vector(31 downto 0);
    carry_in: in std_logic;
    res: out std_logic_vector(31 downto 0); --Outputs: result of
operation, carry out
    carry_out: out std_logic --no ; after the last port declaration
    );
end ALU;
```

For specific implementation of each operation see the architecture `alu_beh` of `ALU`.

Implementation considerations:

--No shifting or rotating of operands supported

--For 8 opcodes which do not affect the carry (`and, orr, eor, bic, mov, mvn, tst, teq`), carry_out has been assigned `carry_in`

--Uses 2's complement subtraction

--Uses 33 bit std_logic_vector to store temporary results and obtain carry output in operations that require addition/subtraction (`add, sub, rsb, adc, sbc, rsc, cmp, cmn`)

2. RegFile_stage1.vhd contains the entity RF

`RF`: Register File with register memory as an array of 16 std_logic_vectors of 32-bits

Inputs-

I.)`CLK`: clock as a single bit

II.)`read_addr1, read_addr2`: two read address ports, as 4-bit std_logic_vectors

III.)`write_addr`: one write address port as 4-bit std_logic_vector

IV.)`write_en`: write enable, write operation performed only when this is active

V.)`data_in`: 32-bit std_logic_vector one data port for 1-word write operation in Register File in address corresponding to write_addr

Outputs:

I.)`data_out1, data_out2`: 32-bit std_logic_vector two data ports for 1-word read operation from Register File from the addresses corresponding to read_addr1 and read_addr2 respectively

```
entity RF is
port(
    CLK: in bit;
    read_addr1: in std_logic_vector(3 downto 0); --two read address
ports and one write address port
    read_addr2: in std_logic_vector(3 downto 0);
    write_addr: in std_logic_vector(3 downto 0);
    write_en: in std_logic; --write enable
    data_in: in std_logic_vector(31 downto 0); --write port
    data_out1: out std_logic_vector(31 downto 0); --read ports
corresponding to read addr 1 and 2 respectively
    data_out2: out std_logic_vector(31 downto 0)
);
end RF;
```

For implementation, see the architecture `rf_beh` of `RF`.

Implementation considerations:

--Two data outputs on which contents of the array elements selected by read addresses are continuously available.

--If write enable is active, at rising clock edge the input data gets written in the array element selected by write address.

--Word- level addressing and only word level R/W supported.

3. Mem_stage3.vhd contains the entity Mem

```
entity Mem is
port(
    CLK: in bit;
    addr: in std_logic_vector(6 downto 0); --one address port only as
read and write never done together, word level addressing
    write_en: in std_logic_vector(3 downto 0); --byte level write enable
    data_in: in std_logic_vector(31 downto 0); --write port
    data_out: out std_logic_vector(31 downto 0) --read port
);
end Mem;
```

`Mem`: Combined program and data memory with memory implemented as an array of 128 std_logic_vectors of 32-bits

Inputs-

I.)`CLK`: clock as a single bit

II.)`addr`: 32-bit std_logic_vector address port, for both read/write

III.)`write_en`: std_logic_vector(3 downto 0), 4 bits for byte level write operation

IV.)`data_in`: 32-bit std_logic_vector one data port for 1-word write operation in memory in address corresponding to addr

Outputs:

I.)`data_out`: 32-bit std_logic_vector one data port for 1-word read operation from memory from the address corresponding to addr

For implementation, see the architecture `mem_beh` of `Mem`.

Implementation considerations:

--`data_out` has contents of the memory at address addr continuously available.

--Has 4 bit write enable to support byte level write operation in memory. At the rising edge of the clock, according to the bits set in write_en, the corresponding bytes are written in the word selected by the addr.

--Word- level addressing

--Word-level read operation from memory supported

--As same memory for both data and program, there should be no overlap

--in testcases, assumed that the program occupies first 64 words in memory.

4. mytypes_stage2.vhd

Defines

```
subtype word is std_logic_vector (31 downto 0);
subtype hword is std_logic_vector (15 downto 0);
subtype byte is std_logic_vector (7 downto 0);
subtype nibble is std_logic_vector (3 downto 0);
subtype bit_pair is std_logic_vector (1 downto 0);

type optype is (andop, eor, sub, rsb, add, adc, sbc, rsc, tst, teq, cmp,
cmn, orr, mov, bic, mvn);
type instr_class_type is (DP, DT, MUL, BRN, none);
type DP_subclass_type is (arith, logic, comp, test, none);
type DP_operand_src_type is (reg, imm);
type load_store_type is (load, store);
type DT_offset_sign_type is (plus, minus);
```

5. decoder_stage2.vhd contains the entity Decoder

`Decoder`: Instruction decoder is a combinational circuit which takes as input an instruction and outputs the following information about the instruction for selecting the appropriate control signals for the other modules-

```
entity Decoder is
Port (
    instruction : in word;
    instr_class : out instr_class_type;
    operation : out optype;
    DP_subclass : out DP_subclass_type;
    DP_operand_src : out DP_operand_src_type;
    load_store : out load_store_type;
    DT_offset_sign : out DT_offset_sign_type
);
end Decoder;
```

For implementation, see the architecture `Behavioral` of `Decoder`.
(This module has been posted on Moodle)

6. FlagUpdater_stage3.vhd contains the entity FlagUpdater

`FlagUpdater`: Updates the flags `Z, N, C, V` on the rising edge of clock if `Fset = '1'`. Flags are updated using the MSB's of ALU operands, ALU result and ALU carry and DP_subclass.

```
entity FlagUpdater is
Port (
    `CLK`: in bit;
    `DP_subclass` : in DP_subclass_type;  --multiply instructions not
included
    `Fset` : in std_logic; --if this is 1 when then at the rising edge
of the clock, flags are updated
    `carry_ALU, MSBop1, MSBop2`: in std_logic; --carry bit, MSB bits of
operands of ALU
    --shift carry not included
    `res_ALU`: in word; --result of ALU
    `Z, V, C, N`: out std_logic := '0'
);
end FlagUpdater;
```

Implementation considerations:

--Does not consider the shift carry.

--Instructions posted on Moodle regarding modifying flags have been followed.

For implementation, see the architecture `fu_beh` of FlagUpdater.

7. conditionChecker_stage3.vhd contains the entity ConditionChecker

`ConditionChecker`: Looks at the flags `Z, N, C, V` and the 4-bit condition field of the instruction, and returns whether the condition is true or not as `res`.

```
entity ConditionChecker is
Port (
    Z, V, C, N: in std_logic;
    cond_field: in std_logic_vector(3 downto 0); --31 to 28 bits of
insruction specifying the condition code
    res: out std_logic  --res is true if the cond_field satisfies the
appropriate flag requirements
);
end ConditionChecker;
```

Implementation considerations:

--Only `EQ, NE`, and always true condition (1110 and 1111) have been tested for this stage.

For implementation, see the architecture `cc_beh` of `ConditionChecker`.

8. programCounter_stage3.vhd contains the entity PC

`PC`: At the rising edge of the clock, if `write_en = '1'`, then `pc_out` is assigned `pc_in`.

```
entity PC is
Port (
    CLK: in bit;
    pc_in: in word; --input program counter value
    write_en: std_logic;
    pc_out: out word := X"00000000" --actual program counter register
);
end PC;
```

For implementation, see the architecture `pc_beh` of `PC`.

9. processor_stage3.vhd contains the entity processor

```
entity processor is
Port(
    CLK: in bit;
    reset: in bit
);
end processor;
```

`processor`: Instantiates and connects all the components described above and also contains an `FSM` as a `process`.

`FSM` has 9 states (as given in lec 10 slide 38), state transitions take place at the rising edge of clock, each state generates control signals (select signals for various multiplexers) such as

- write enable for `RF, Mem, PC`
- `Fset`, whether to update `Flags`
- value of `carry_in` for `ALU`
- opcode and operands of `ALU`
- write data for `RF`
- second read address of `RF`, address of `Mem`

The states are-

1. Load current instruction in `IR`, `PC` updated by 4
2. Load `A` and `B` with register contents of Register File

3. `DP` instruction, load `alu_res` in `Res`, update flags if required
4. next step in `DP`, store `Res` in Register file

4. `DT` instruction, load `Res` with `alu_res` computing address of Memory for load/store
5. if `str`, store contents of `B` in memory
6. if `ldr`, store memory contents in `DR`
7. next in `ldr`, store `DR` contents in `RF`

5. if conditon is true, update `PC` with the required offset, if condition false, add `0`

Implementation considerations:

--Various input and output signals of the different components are connected as described in slide 30 of Lec10. Some signals are directly connected through concurrent assignments and those requiring multiplexing have been connected through concurrent conditional assignments. The select signals for multiplexers are decided by the instruction (in `IR`), `decoder` outputs and the current state of `FSM`.

--Word-level addressing has been implemented internally in memories and register file, the processor interconnects have also been defined using word level addresses. Hence, unlike stage 2, now the instruction is assumed to follow word level addressing, for eg, `str r1, [r0, 1]` will store contents of `r1` in memory in word number `r0 + 1`.

--`PC` is updated by `4` in `state 1`, and for branch instructions -

- if the condition is not satisfied, in `state 5`, `-1` is provided as `op2` (and carry_in is 1 for ALU), so effectively `0` is added to `PC` in `state 5`.
- if condition is satified, 24 bit word offset is provided as `op2` (and carry_in is 1 for ALU), so effectively `4*offset + 4` is added to `PC` in `state 5`.

--If `reset = '1'` then the `FSM` returns to `state 1`.

For implementation, see the architecture `pro_beh` of `processor`.

10. Testbench.vhd

The program memory has been initialised with different machine code programs. In the testbench, `DUT` of `processor` is created, then it is simulated and the EPWave is observed.

As same memory is used for both data and program, there should be no overlap, hence in testcases, assumed that program occupies first 64 words in memory.

11. design.vhd

Dummy entity to run the simulation.

12. run.do

Specifies the FPGA to be used for synthesis and to report of the synthesis.

How to use:

On edaplayground.com, upload testbench.vhd and run.do in the left column, and ALU_stage3.vhd, RegFile_stage1.vhd, Mem_stage3.vhd, mytypes_stage2.vhd , decoder_stage2.vhd, FlagUpdater_stage3.vhd, conditionChecker_stage3.vhd, programCounter_stage3.vhd, processor_stage3.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:

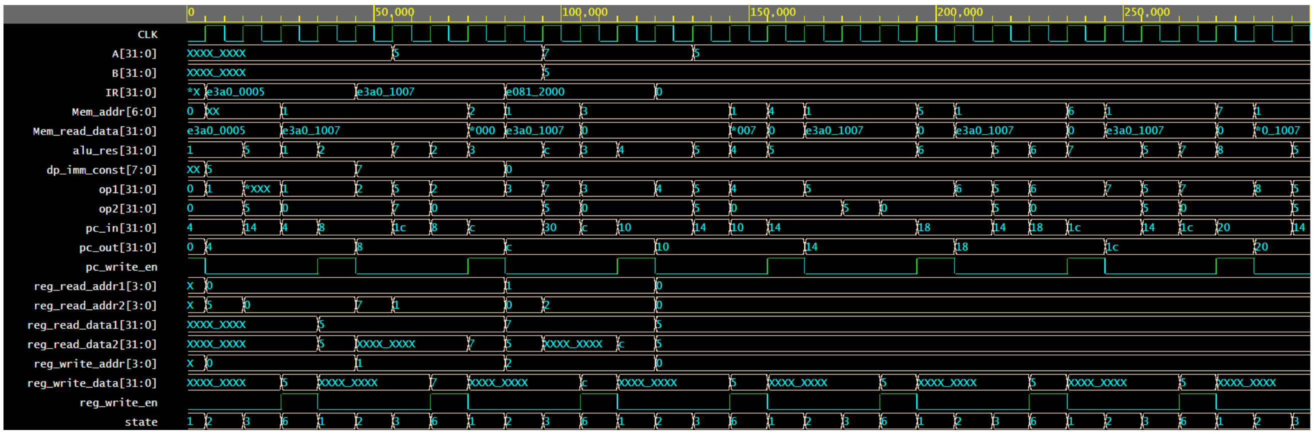
1.) EPWave (Simulation):

Can see the input and output signals of processor against the clock.

Memory initialisation with different programs and EPWave-

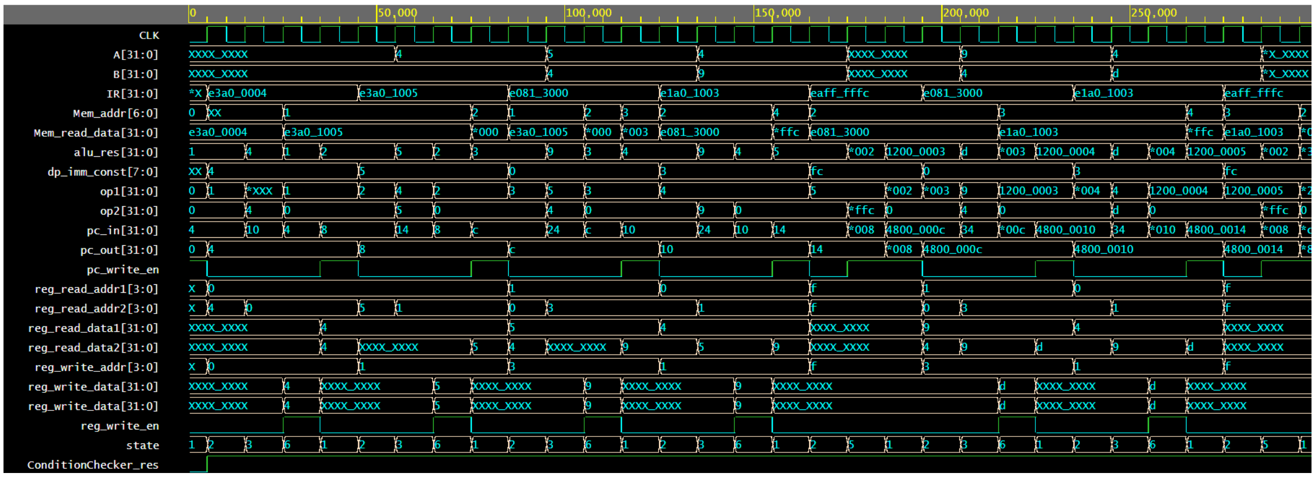
1.

```
signal Mem_space : type_mem := (0 => X"E3A00005",
                                1 => X"E3A01007",
                                2 => X"E0812000",
                                others => X"00000000"
                                );
--mov r0, #5
--mov r1, #7
--add r2, r1, r0
```



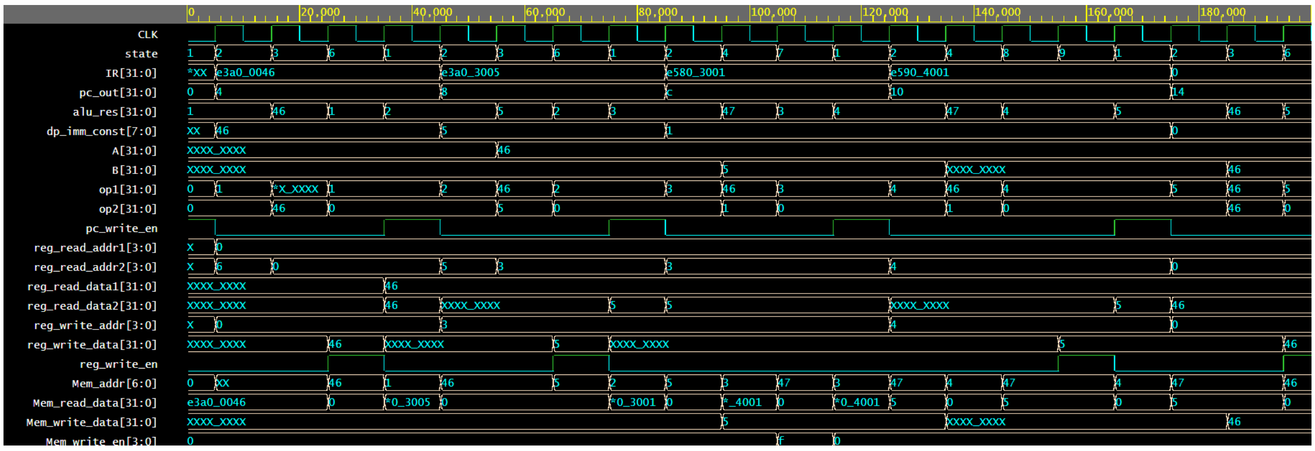
2.

```
signal Mem_space : type_mem := (0 => X"E3A00004",
                               1 => X"E3A01005",
                               2 => X"E0813000",
                               3 => X"E1A01003",
                               4 => X"EAFFFFFC",
                               others => X"00000000"
                               );

--mov r0, #4
--mov r1, #5
--L: add r3, r1, r0
--mov r1, r3
--b L
```



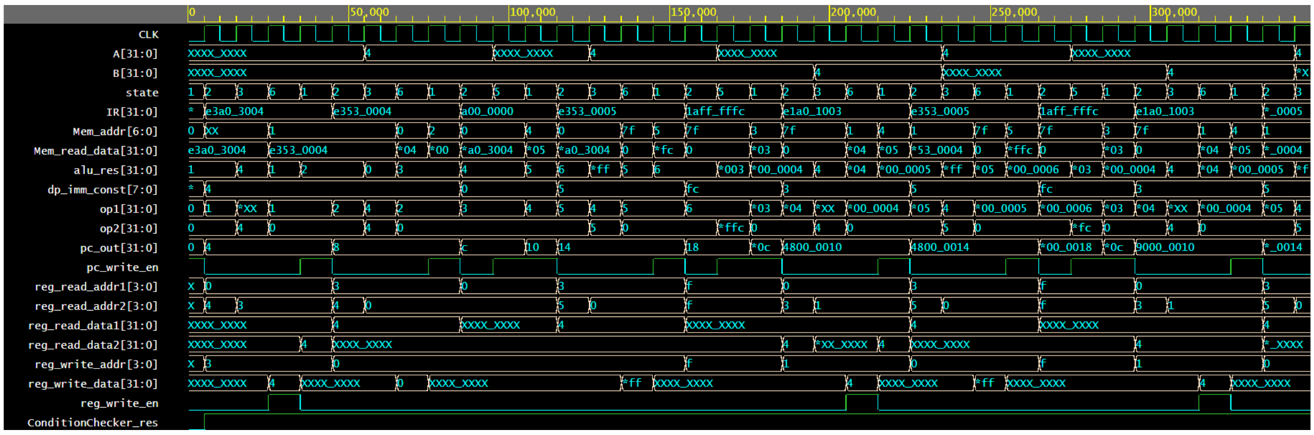
3.

```
        signal Mem_space : type_mem := (0 => X"E3A00046",
                                        1 => X"E3A03005",
                                        2 => X"E5803001",
                                        3 => X"E5904001",
                                        others => X"00000000"
                                        );
        --mov r0, #70
        --mov r3, #5
        --str r3, [r0, #1] --store in word number 71 in memory
        --ldr r4, [r0, #1]
```



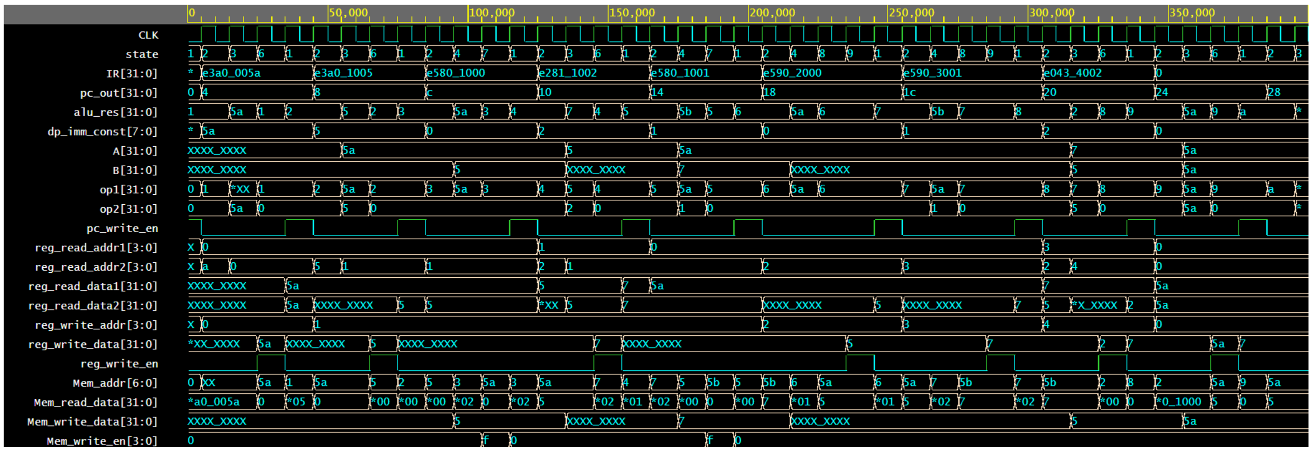
4.

```
signal Mem_space : type_mem := (0 => X"E3A03004",
                                1 => X"E3530004",
                                2 => X"0A000000",
                                3 => X"E1A01003",
                                4 => X"E3530005",
                                5 => X"1AFFFFFC",
                                others => X"00000000"
                                );
--mov r3, #4
--cmp r3, #4
--beq L
--K: mov r1, r3
--L: cmp r3, #5
--bne K
```



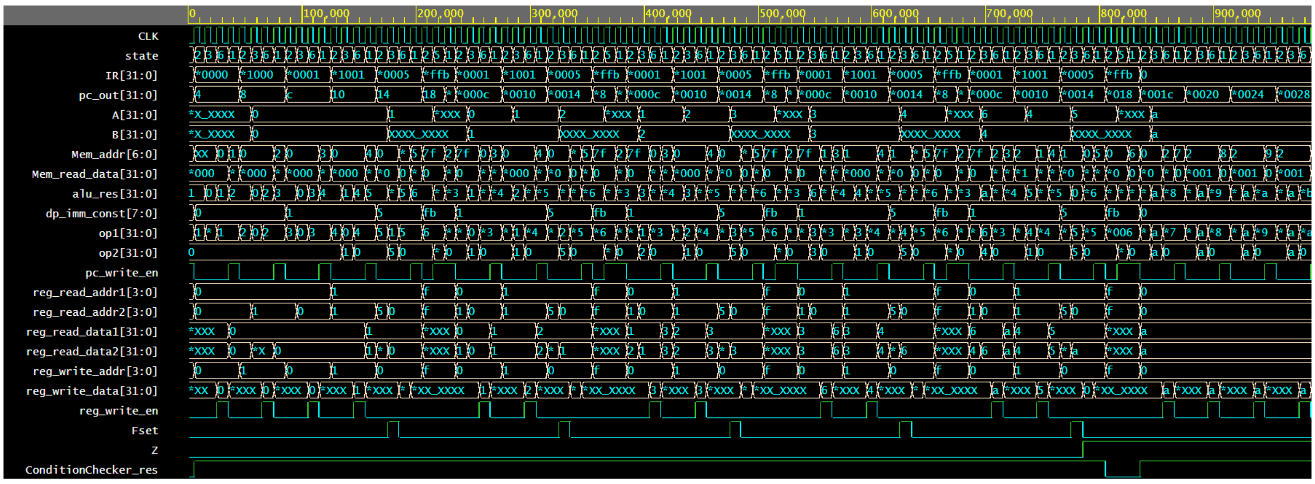
5.

```
        signal Mem_space : type_mem :=
                                    (0 => X"E3A0005a",
                                     1 => X"E3A01005",
                                     2 => X"E5801000",
                                     3 => X"E2811002",
                                     4 => X"E5801001",
                                     5 => X"E5902000",
                                     6 => X"E5903001",
                                     7 => X"E0434002",
                                     others => X"00000000"
                                     );
        --mov r0, #90
        --mov r1, #5
        --str r1, [r0]
        --add r1, r1, #2
        --str r1, [r0, #1]
        --ldr r2, [r0]
        --ldr r3, [r0, #1]
        --sub r4, r3, r2
```



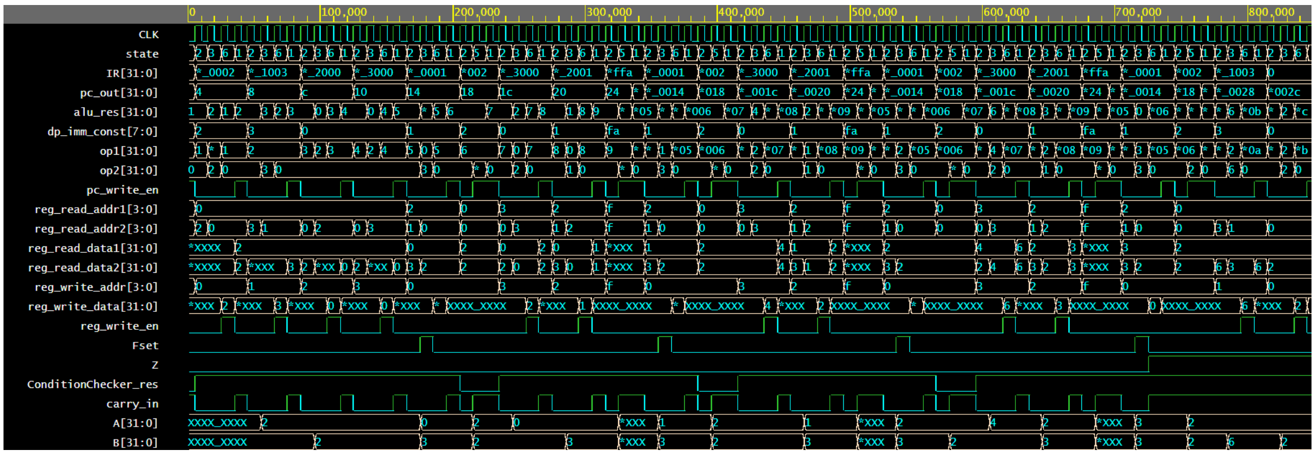
6.

```
signal Mem_space : type_mem :=
                                        (0 => X"E3A00000",
                                         1 => X"E3A01000",
                                         2 => X"E0800001",
                                         3 => X"E2811001",
                                         4 => X"E3510005",
                                         5 => X"1AFFFFFB",
                                         others => X"00000000"
                                         );
--mov r0, #0
--mov r1, #0
--Loop: add r0, r0, r1
--add r1, r1, #1
--cmp r1, #5
--bne Loop
```



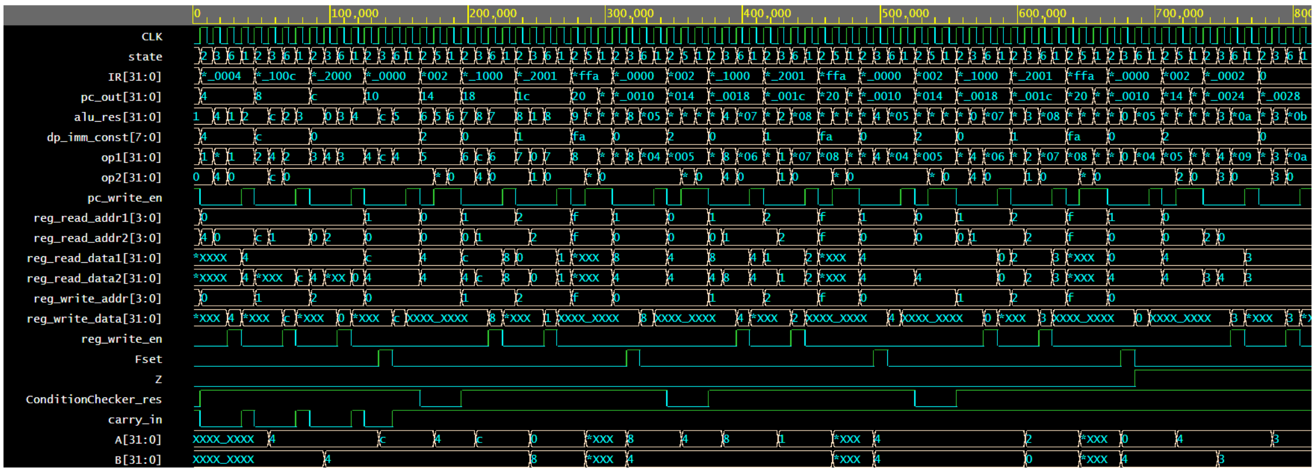
7.

```
        signal Mem_space : type_mem :=
                                        (0 => X"E3A00002",
                                         1 => X"E3A01003",
                                         2 => X"E3A02000",
                                         3 => X"E3A03000",
                                         4 => X"E1520001",
                                         5 => X"0A000002",
                                         6 => X"E0833000",
                                         7 => X"E2822001",
                                         8 => X"EAFFFFFA",
                                         9 => X"E1A01003",
                                         others => X"00000000");
        --Code for multiplying r0*r1
        --0  mov r0, #2
        --1  mov r1, #3
        --2  mov r2, #0
        --3  mov r3, #0
        --4  B: cmp r2, r1
        --5  beq C
        --6  add r3, r3, r0
        --7  add r2, r2, #1
        --8  b B
        --9 mov r1, r3 --r1 stores the result
```



8.

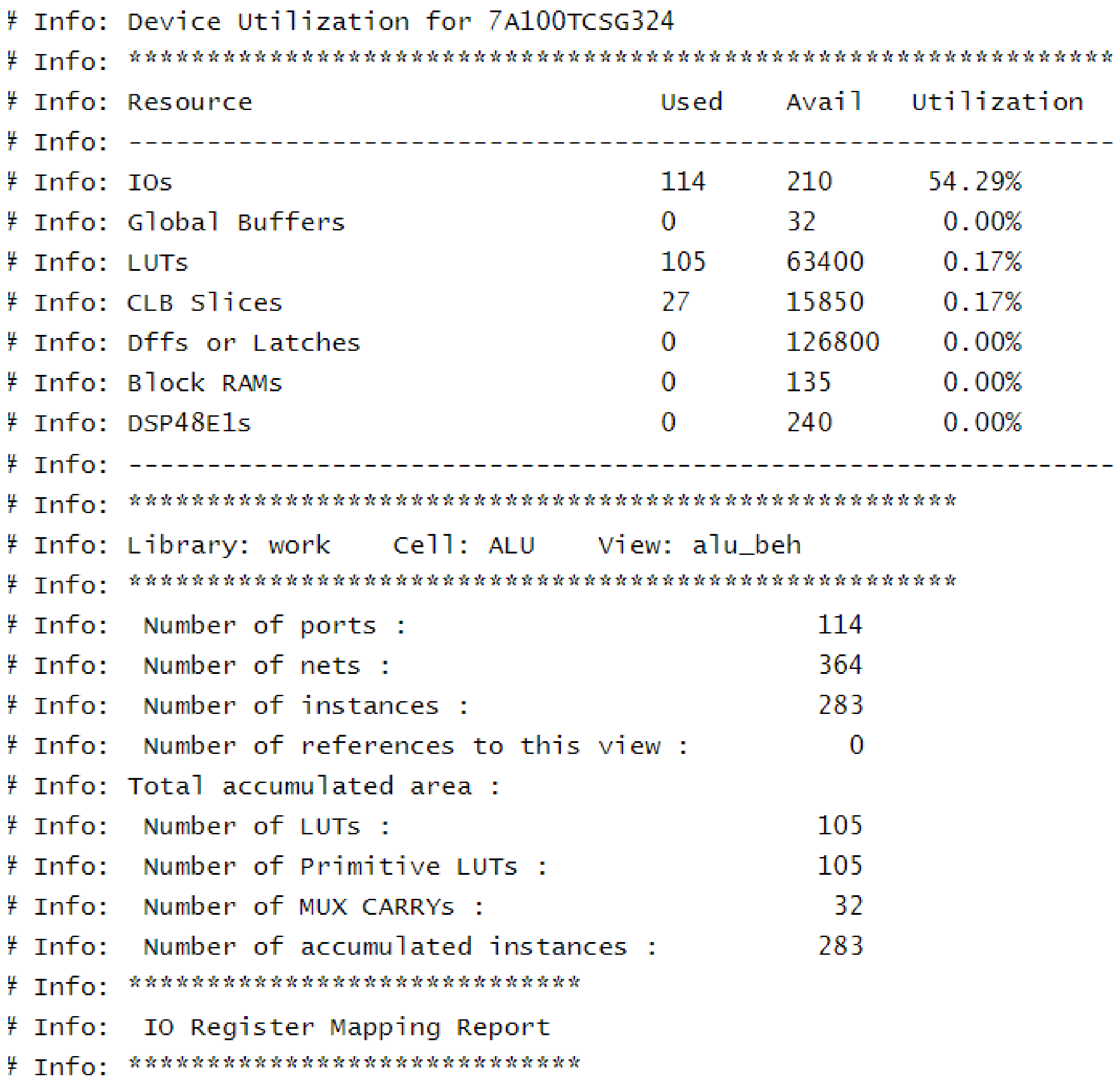
```
    signal Mem_space : type_mem :=
                                    (0 => X"E3A00004",
                                     1 => X"E3A0100C",
                                     2 => X"E3A02000",
                                     3 => X"E3510000",
                                     4 => X"0A000002",
                                     5 => X"E0411000",
                                     6 => X"E2822001",
                                     7 => X"EAFFFFFA",
                                     8 => X"E1A00002",
                                     others => X"00000000");



    --code for dividing r1/r0
    --0 mov r0, #4
    --1 mov r1, #12
    --2 mov r2, #0
    --3 a: cmp r1, #0
    --4 beq b
    --5 sub r1, r1, r0
    --6 add r2, r2, #1
    --7 b a
    --8 b: mov r0, r2 --r0 stores the quotient
```



2.) Resource Table (Synthesis)

Can see the resources used by the different module implementations on given FPGA.

For example, for ALU -



```
# Info: Device Utilization for 7A100TCSG324
# Info: ****************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ----------------------------------------------------------------
# Info: IOs                               114     210      54.29%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              105     63400     0.17%
# Info: CLB Slices                        27      15850     0.17%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: DSP48E1s                          0       240       0.00%
# Info: ----------------------------------------------------------------
# Info: ******************************************************
# Info: Library: work    Cell: ALU    View: alu_beh
# Info: ******************************************************
# Info:  Number of ports :                          114
# Info:  Number of nets :                           364
# Info:  Number of instances :                      283
# Info:  Number of references to this view :          0
# Info: Total accumulated area :
# Info:  Number of LUTs :                           105
# Info:  Number of Primitive LUTs :                 105
# Info:  Number of MUX CARRYs :                      32
# Info:  Number of accumulated instances :          283
# Info: ******************************
# Info:  IO Register Mapping Report
# Info: ******************************
```