Lab Assignment 2, Stage 2: A simple processor design

The module set includes ALU, Register File, Program Memory, Data Memory, Decoder, Flag Updater, Condition Checker and Program Counter.

Kushal Kumar Gupta

2020CS10355

This stage involves putting together the four modules designed at stage 1, namely ALU, Register File, Program Memory and Datat Memory, to form a simple processor which can execute the following subset of ARM instructions with limited variants/features. {add, sub, cmp, mov, ldr, str, beq, bne, b} Though the ALU has been designed for a larger set of operations, only some of these will get exercised at this stage.

Files, entities and Architectures:

1. **ALU_stage1.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 1-bit std_logic_vector

   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

```
      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_vector(0 downto 0);

      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
```

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

```
  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)
```

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. DataMemory_stage1.vhd contains the entity DM

DM: Data Memory with memory implemented as an array of 64 std_logic_vectors of 32-bits

Inputs-

I.)CLK: clock as a single bit

II.)addr: 32-bit std_logic_vector one 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

```
  CLK: in bit;

  addr: in std_logic_vector(5 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
```

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 dm_beh of DM.

Implementation considerations:

--Data_out has contents of the array elements selected by addr continuously available.

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

--Word- level addressing

--Word-level read operation from memory supported

4. ProgramMemory_stage2.vhd contains the entity PM

PM: Program Memory with memory implemented as an array of 64 std_logic_vectors of 32-bits

Inputs-

I.)addr: one read address port, as 32-bit std_logic_vectors

Outputs:

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

```
addr: in std_logic_vector(5 downto 0); --address port

data_out: out std_logic_vector(31 downto 0) --read port
```

For implementation, see the architecture pm_beh of PM.

Implementation considerations:

--The data output on which contents of the array elements selected by read addresses is continuously available.

--Word- level addressing

--Word-level read operation from memory supported

--Read Only Memory, write operation not supported

--Memory is initialised to a program with some intructions for testing

5. mytypes_stage2.vhd

Defines

1.subtypes `word, hword, byte, nibble, bit_pair`

2. types

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

6. 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-

```
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
```

For implementation, see the architecture Behavioral of Decoder.

7. FlagUpdater_stage2.vhd contains the entity FlagUpdater

FlagUpdater: Updates the flags Z, N, C, V on the rising edge of clock if the S bit of the instruction is set. Flags are updated using the MSB's of ALU operands, ALU result and ALU carry, instr_class and DP_subclass. Instructions posted

```
  CLK: in bit;

  instr_class : in instr_class_type; --Only DP instructions affect flags

  DP_subclass : in DP_subclass_type;  --multiply instructions not
included

  S : in std_logic; --S bit

  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
```

Implementation considerations:

--Does not consider the shift carry.

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

--Flags are only updated for DP instructions.

For implementation, see the architecture fu_beh of FlagUpdater.

8. conditionChecker_stage2.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.

```
  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 boolean  --res is true if the cond_field satisfies the
appropriate flag requirements
```

Implementation considerations:

--Only EQ, NE, and no condition (1110 and 1111) have been tested for this stage.

For implementation, see the architecture cc_beh of ConditionChecker.

9. programCounter_stage2.vhd contains the entity PC

PC: Maintains an internal program counter pc. At the rising edge of the clock, updates this pc and puts into pc_out. pc_out is incremented by 4 for non-branch instructions.

For branch instructions, checks the input condition, if the condition is true then pc_out is assigned pc + 8 + 4*offset, if condition is false, incremented by 4.

```
    CLK: in bit;

    cond: in boolean;  --whether the conditon provided by condition
checker (based on condition field and flag values) is true or false

    instr_class : in instr_class_type;  --DP, DT or BRN

    offset: in std_logic_vector(23 downto 0);  --24 bit offset
    pc_out: out word
```

Implementation considerations:

--Only B, BEQ, BNE branch instructions supported.

--24-bit Signed (word) offset is provided.

For implementation, see the architecture pc_beh of PC.

10. processor_stage2.vhd contains the entity processor

processor: Instantiates and connects all the components described above to make a single-cycle processor.

Implementation considerations:

--Various input and output signals of the different components are connected as described in slide 40 of Lec09. Some signals are directly connected through concurrent assignments and those requiring multiplexing have been connected through concurrent conditional assignments.

--Though word-level addressing has been implemented internally in memories and register file, the processor interconnects have been defined using byte level addresses of 32 bits. Hence Program Counter as well as machine codes should consider the processor as having byte-level addressing and the addresses provided should be word-aligned (i.e. multiples of 4).

For implementation, see the architecture pro_beh of processor.

11. 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.

12. Design.vhd

Dummy entity to run the simulation.

13. run.do

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

How to use:

On edaplayground.com, upload testbench.vhd and run.do in the left column, and ALU_stage1.vhd, RegFile_stage1.vhd, DataMemory_stage1.vhd, ProgramMemory_stage2.vhd, mytypes_stage2.vhd , decoder_stage2.vhd, FlagUpdater_stage2.vhd, conditionChecker_stage2.vhd, programCounter_stage2.vhd, processor_stage2.vhd. Copy contents in design.vhd as given in the processor_stage2.vhd. 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:

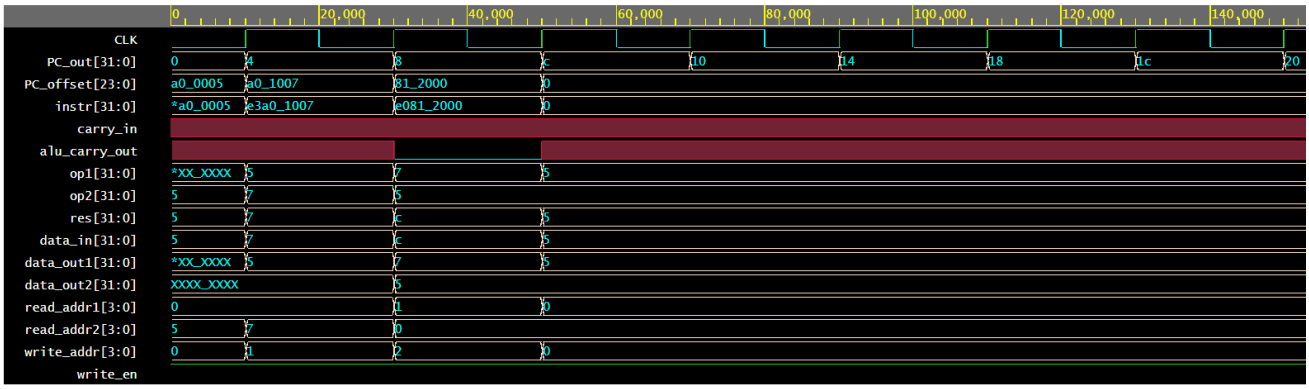
1.) EPWave (Simulation):

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

Program memory initialisation and EPWave-

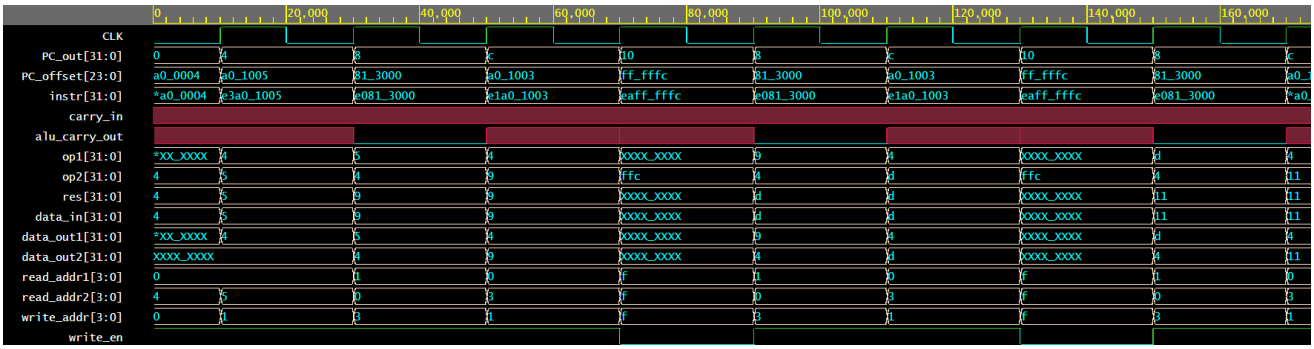
1.

```
signal PM_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
```



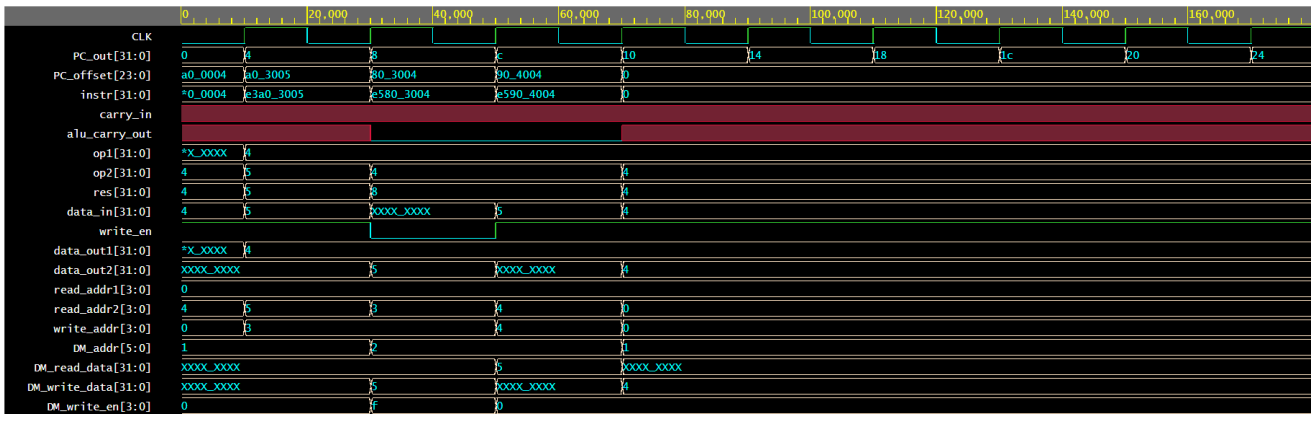
2.

```
signal PM_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
```



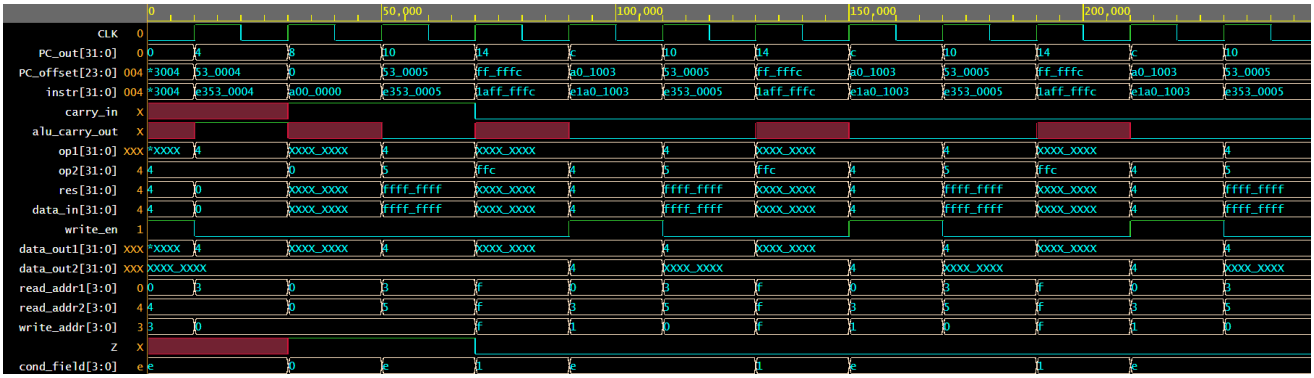
3.

```
signal PM_space : type_mem := (0 => X"E3A00004",
                               1 => X"E3A03005",
                               2 => X"E5803004",
                               3 => X"E5904004",
                               others => X"00000000"
                               );
--mov r0, #4
--mov r3, #5
--str r3, [r0, #4]
--ldr r4, [r0, #4]
```



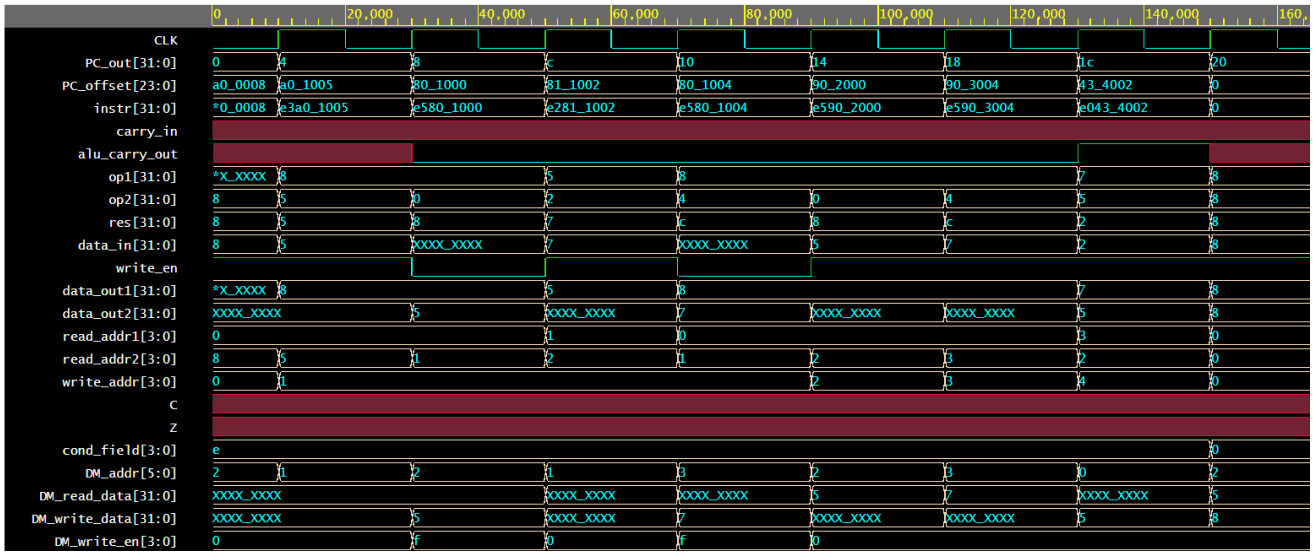
4.

```
signal PM_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.

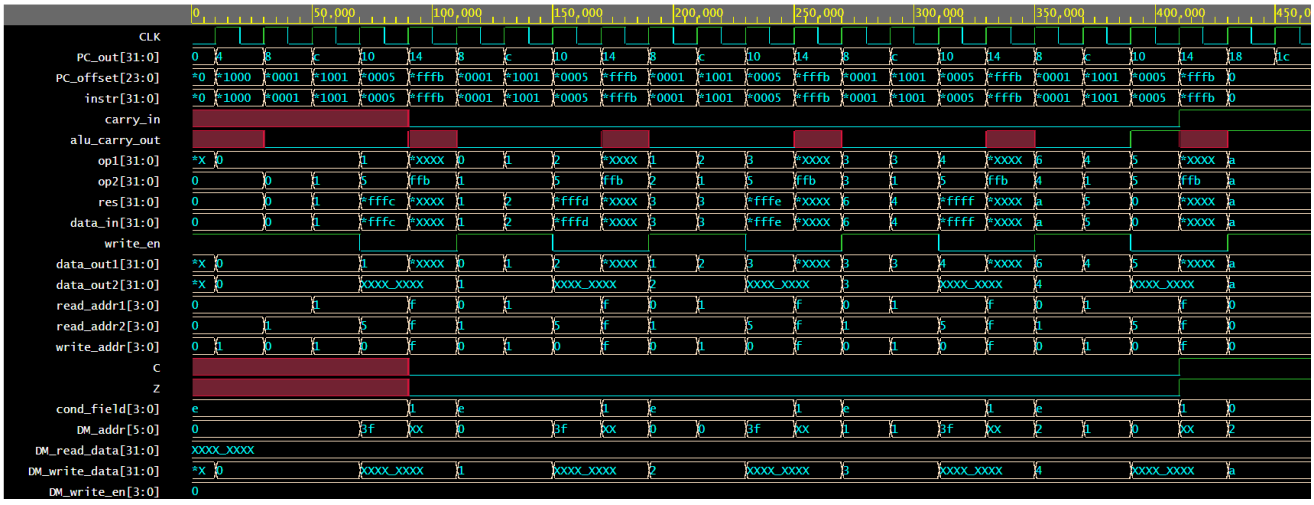
```
signal PM_space : type_mem :=
                               (0 => X"E3A0000A",
                                1 => X"E3A01005",
                                2 => X"E5801000",
                                3 => X"E2811002",
                                4 => X"E5801004",
                                5 => X"E5902000",
                                6 => X"E5903004",
                                7 => X"E0434002",
                                others => X"00000000"
                                );
--mov r0, #10
--mov r1, #5
--str r1, [r0]
--add r1, r1, #2
--str r1, [r0, #4]
--ldr r2, [r0]
--ldr r3, [r0, #4]
--sub r4, r3, r2
```



6.

```
signal PM_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
```



2.) Resource Table (Synthesis)

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

The processor uses instances of the different components and itself does not consume resources as can be seen below.

```
# Info: ************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ------------------------------------------------------------
# Info: IOs                               1       210       0.48%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              0       63400     0.00%
# Info: CLB Slices                        0       15850     0.00%
# 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: processor    View: pro_beh
# Info: ************************************************************
# Info:  Number of ports :                            1
# Info:  Number of nets :                             0
# Info:  Number of instances :                        0
# Info:  Number of references to this view :          0
# Info: Total accumulated area :
# Info:  Number of gates :                            0
# Info:  Number of accumulated instances :            0
# Info: ****************************
# Info:  IO Register Mapping Report
# Info: ****************************
# Info: Design: work.processor.pro_beh
```

You can synthesise other components and see the resource table.