NYU-6463-RV32I Processor Design Project

Sikang Yang(sy3245)

Wei Wang(2385)

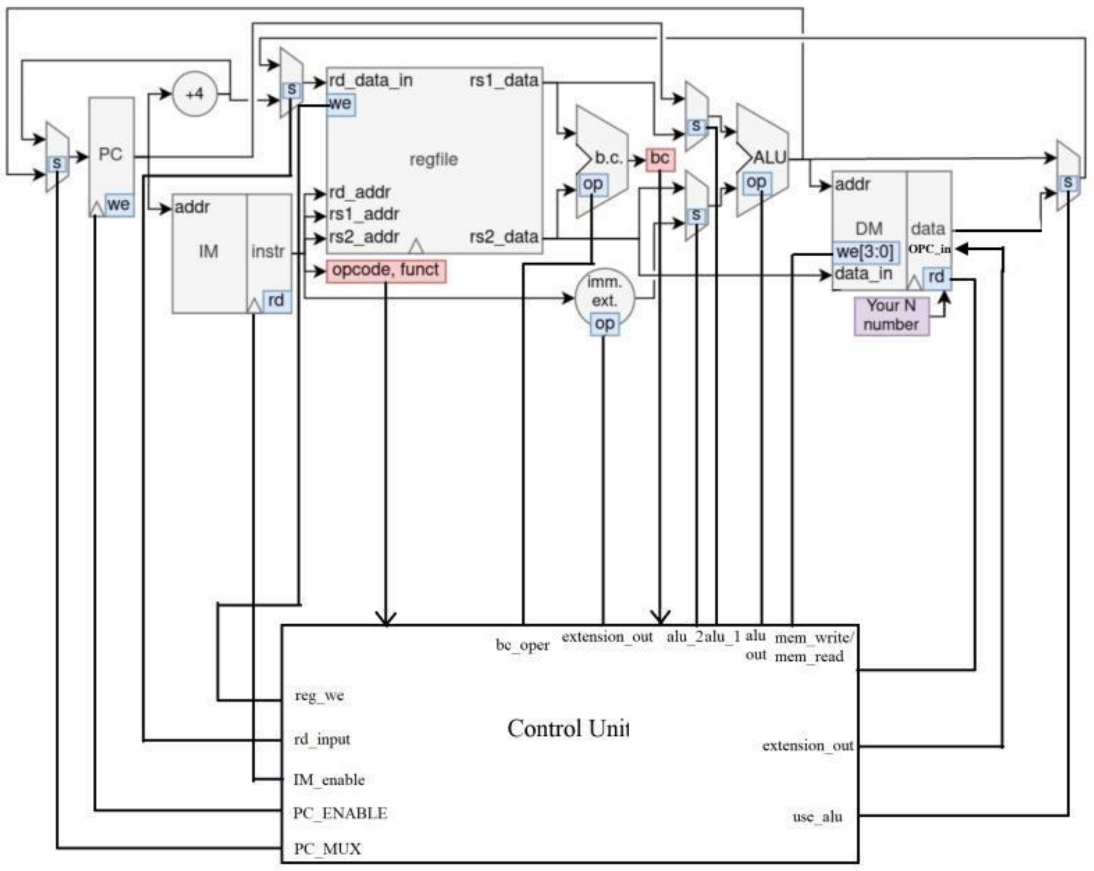
Zhuo Xu (zx1412)

NYU Tandon School of Engineering

12/12/2021

Contents

1. Complete design datapath

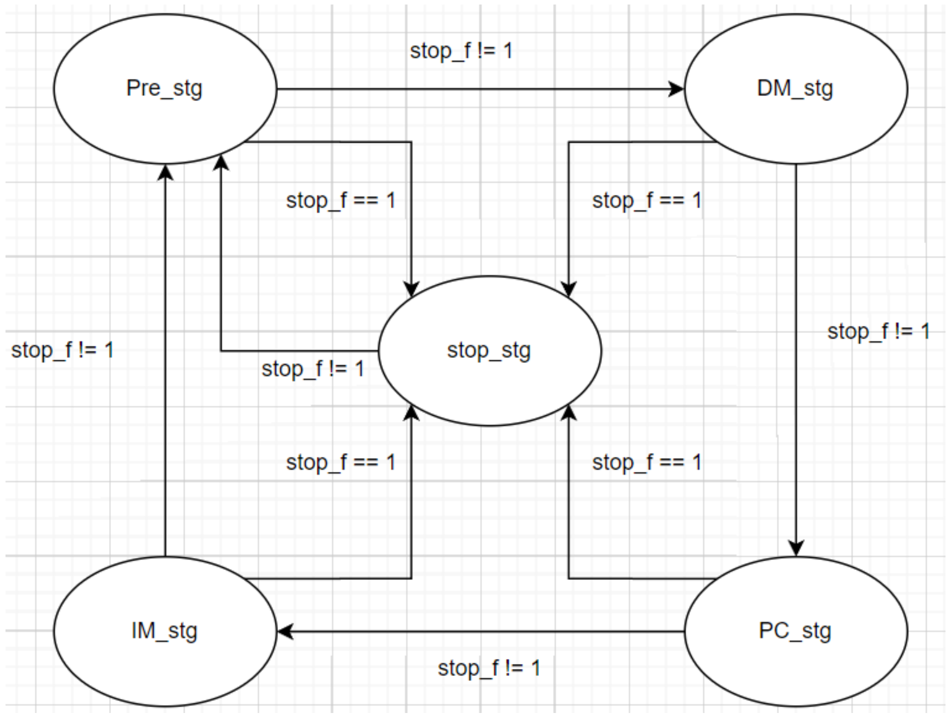


Different from the original design given in request, we modify the datapath, we delete the data extension function and combine the function into the data memory. We delete the Switches and LEDs, in our design, we do not implement them. We also add a control Unit part in the data flow for better illustration.

2. FSM diagram for multi-cycle control

In the control Unit, the FSM is divided into 5 stages: Pre_stg, DM_stg, PC_stg, IM_stg and stop_stg. Pre_stage is the stage that processes the command in Register File, PC_stage is the stage that processes the data in Program Counter Register, DM_stage is the stage that possesses the command in Data Memory and IM_stage is the stage that processes the command in Instruction Memory. Stop_stg is just the stage when the control Unit receives the signal of reset. In every stage, when the stop_f is not 1 and with one clock cycle, the FSM will move to the next stage. If there is a stop_f signal, no matter what stage current is, the FSM will go to stop_stg, in this stage, the processor stop working and waiting for the stop_f to be 0 again, when the signal become 0, the FSM will start from the Pre_stg again.

The whole FSM is designed with the data flow of the processor. When the processor is working, data will first flow to Register File which is the pre_stg, after that data that is out from Register File and ALU will go to the Data Memory which is the DM_stg. Then the data will flow to Program Counter which is the PC_stg and finally, the data will flow to Instruction Memory which is the IM_stg.



3. Justification on the correctness of design

3.1 ALU:

The ALU is tested by testbench with multiple arithmetic operations as well as shift operations and SLT, SLTU operations. The testbench then compares the result with the expectation, report error if mismatched

Operations:

ADD: 10 + 4 = 14

SUB: 14 - 14 = 0

AND: 1010 and 0100 = 0000

OR: 1010 or 0100 = 1110

XOR: 1010 xor 0100 = 1110

SLL: 11110000 sll 4 = 111100000000

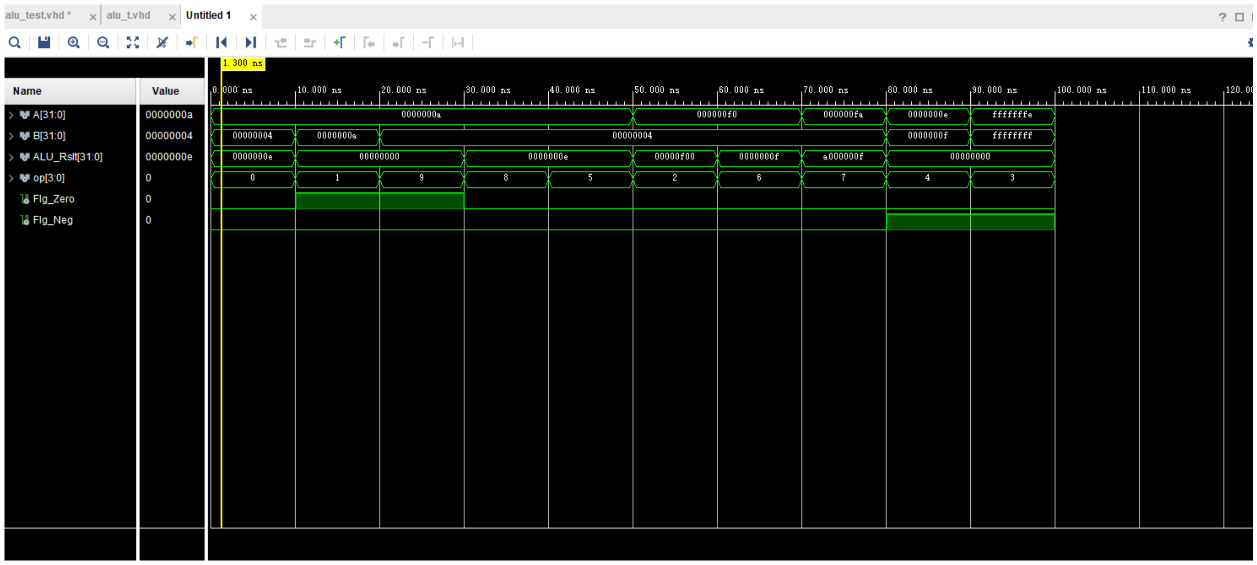
SRL: 11110000 srl 4 = 1111

SRA: X”000000FA” sra X"00000004" = X"A000000F"

SLTU: rs1 = 14, rs2 = 15, rd =1

SLT: rs1 = 14, rs2 = 15, rd =1

simulation screenshots and explanations of low-level test cases:



Immediate Generator:

The immediate generator is tested by a testbench that gives different instructions and input immediate values from instructions, then the testbench checks the output to see if it matches the expected output.

We created vast test cases, for the case of brevity, we will only give a few examples from those test cases.

I type: Load Word

Input immediate = X"00000D0D"

Output immediate = X"FFFFFD0D"

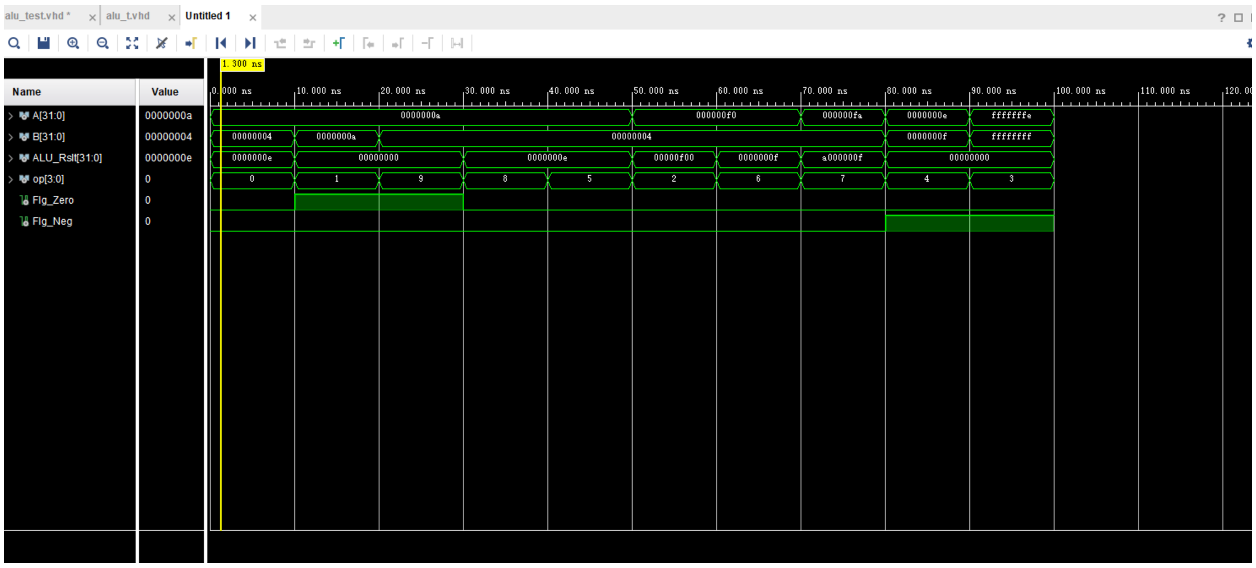
S type: SB

Input immediate = X”00000A99”

Output immediate = X"FFFFFA99"

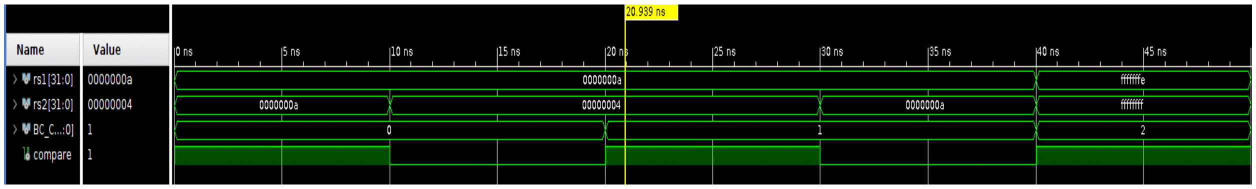
All the test cases passed in our simulation.

Simulation Result:



3.2 ALU Results:

The ALU_Results unit will call the corresponding ALU functionalities and set the flags for ALU operation according to the instructions saved in the instruction memory file. ALU_control port could be set to 0000; 0001; 0010; 0011; 0100; 0101; 0110; 0111; 1000 and 1001, which represent ADD, SUB SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND instructions for ALU. The control unit will set the appropriate control flag for ALU_Results.



3.3 Control Unit:

For the control Unit, The inputs are clk, rst, instruction and branch comparison result. The outputs are signals to memories, muxes, and other components of the processor.

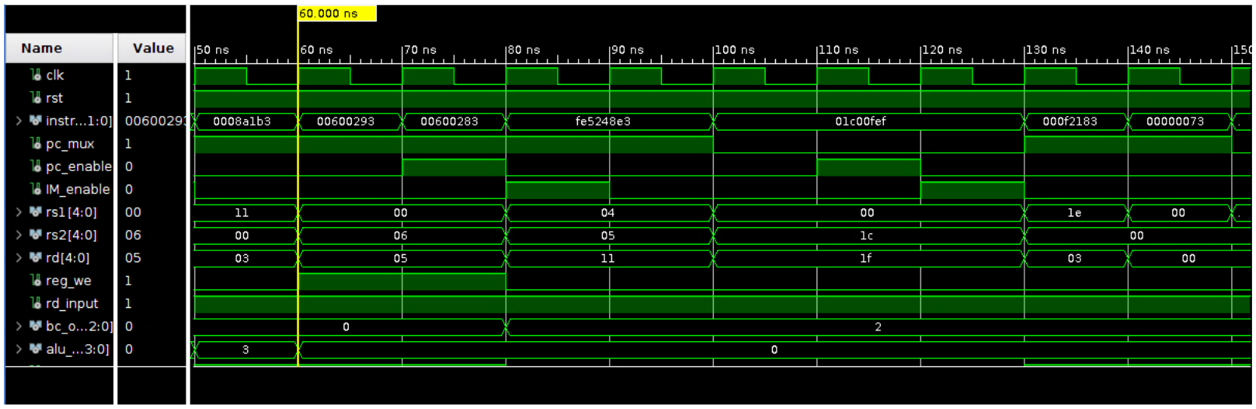
At the beginning, the system will read the opcode, funct3 and funct7 from instruction and perform the signal processing. Then, the system will first determine if there will be alu output or bc output. Then, it will record the rs1, rs2 and rd values. After that, the system will determine the date that will be input to the ALU by controlling two muxes before ALU. Then, with the opcode, the system will determine the immed value and if the register file uses the value from ALU or from Data Memory. With the input signal of BC, the system will determine the pc_mux output with the opcode.

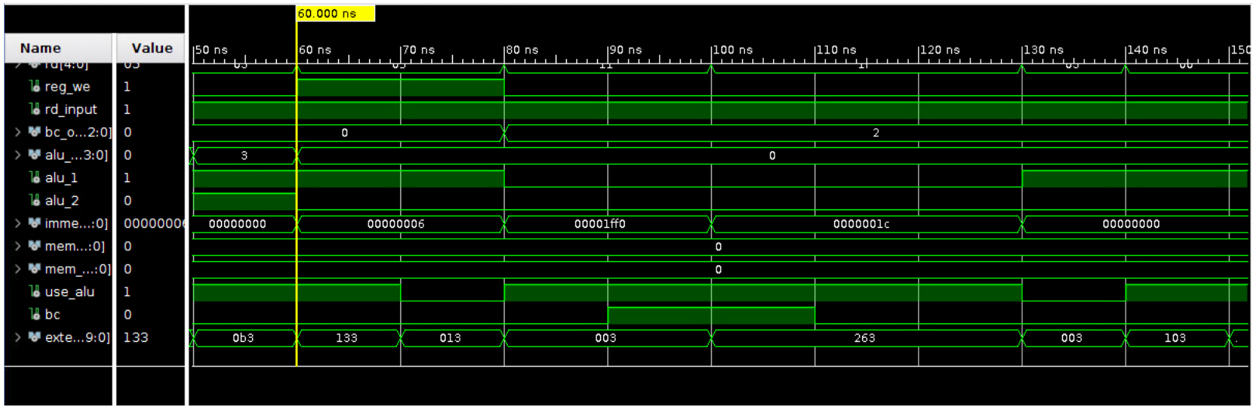
After that, the system will process the reset signal, if there is an rst signal, the system will raise a stop flag.

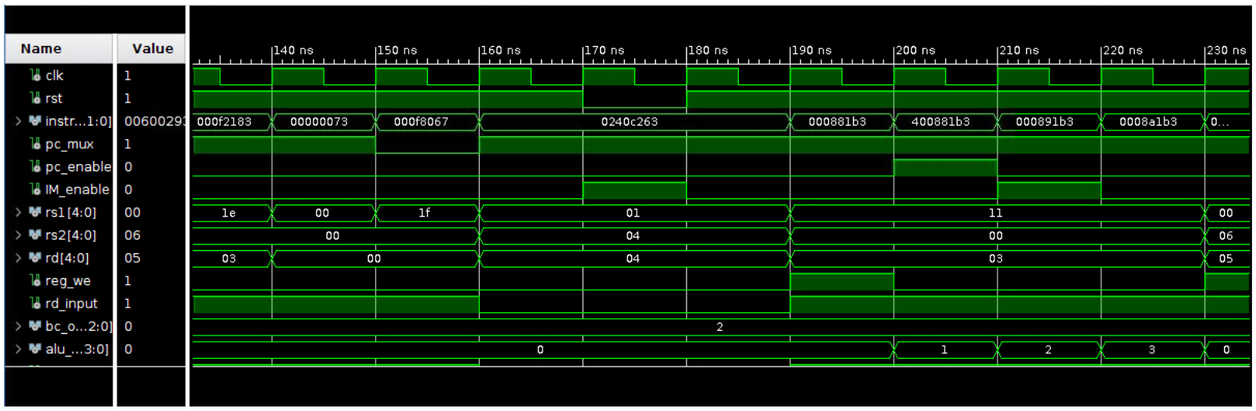
Then, it will be the FSM stages. There are four stages of the FSM: Pre_stage, PC_stage, IM_stage, DM_stage. Pre_stage is the stage that processes the command in Register File, PC_stage is the stage that processes the data in Program Counter Register, DM_stage is the stage that possesses the command in Data Memory and IM_stage is the stage that processes the command in Instruction Memory.

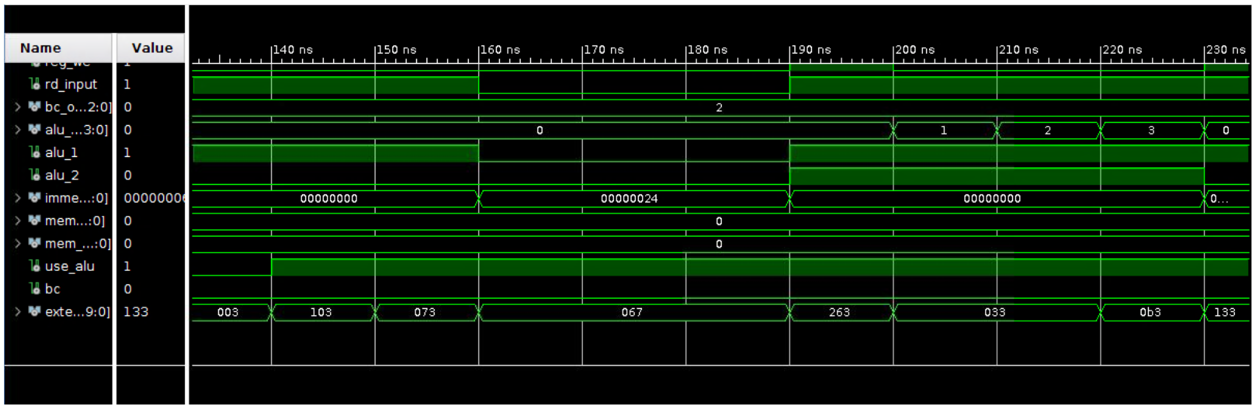
In the test bench of the control Unit, I set one clock cycle to be 10 ns to make sure the time response for processing each instruction is enough. Then I reset the Control Unit variables and process some testing commands that include multiple combinations of input and all the output variables are covered. The responses of the Control Unit are the same as I expected, so I can make sure the control unit is working correctly.

The testing results are attached below:



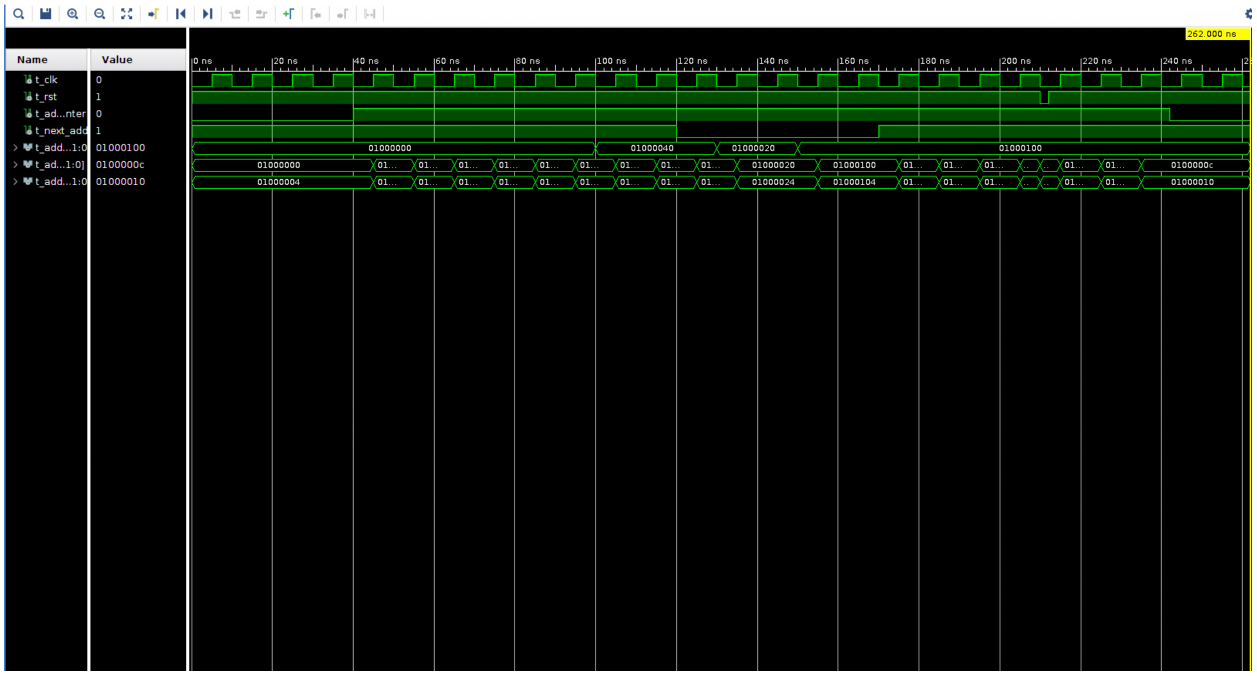


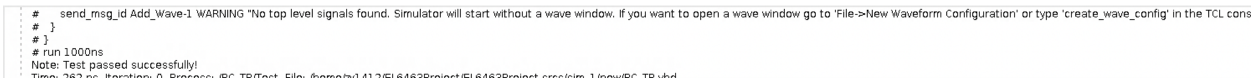




3.4 Program Counter Register

For the program Counter Register. This is a 32-bit register that contains the address of the next instruction to be executed by the processor. Upon reset, this will equal the start address of instruction memory (0x01000000). Port advcounter is a control signal to advance PC. Port next_addr is the control signal to select PC address input from 1 to 4. The starting address is X"01000000".

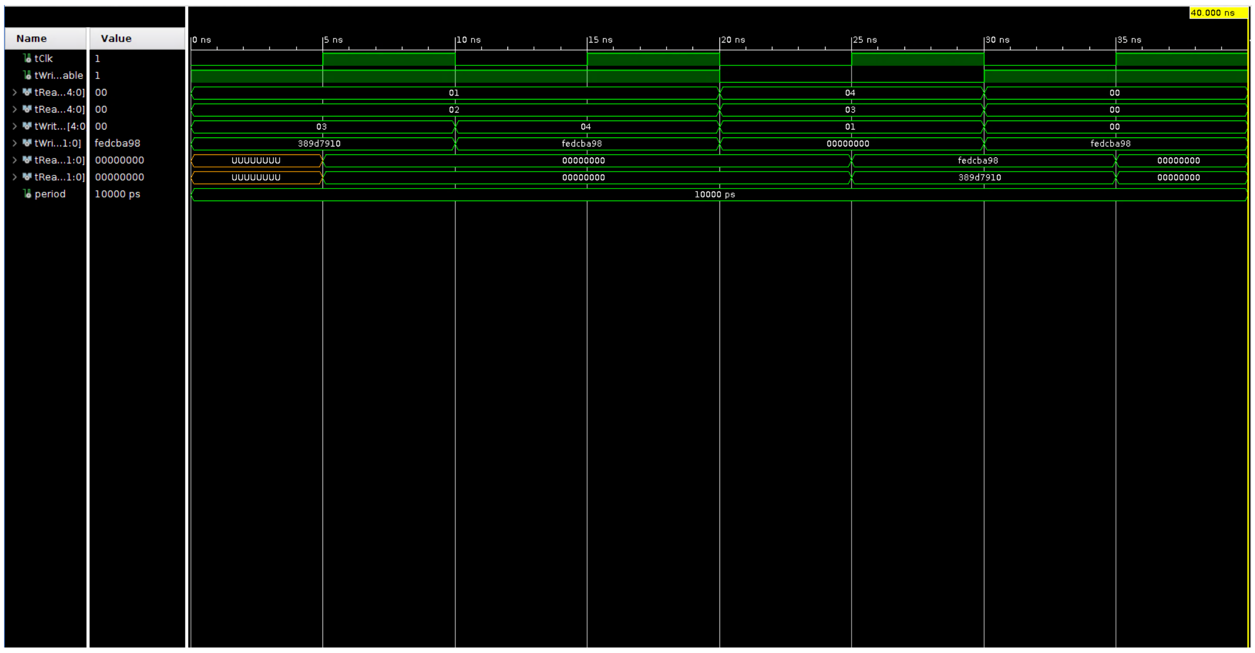




```
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform Configuration' or type 'create_wave_config' in the TCL cons
#  }
# }
# run 1000ns
Note: Test passed successfully!
Time: 262 ns  Iteration: 0  Process: /PC_TB/Test  File: /home/zy1412/EL6463Project/EL6463Project.srcs/sim_1/new/PC_TB.vhd
```

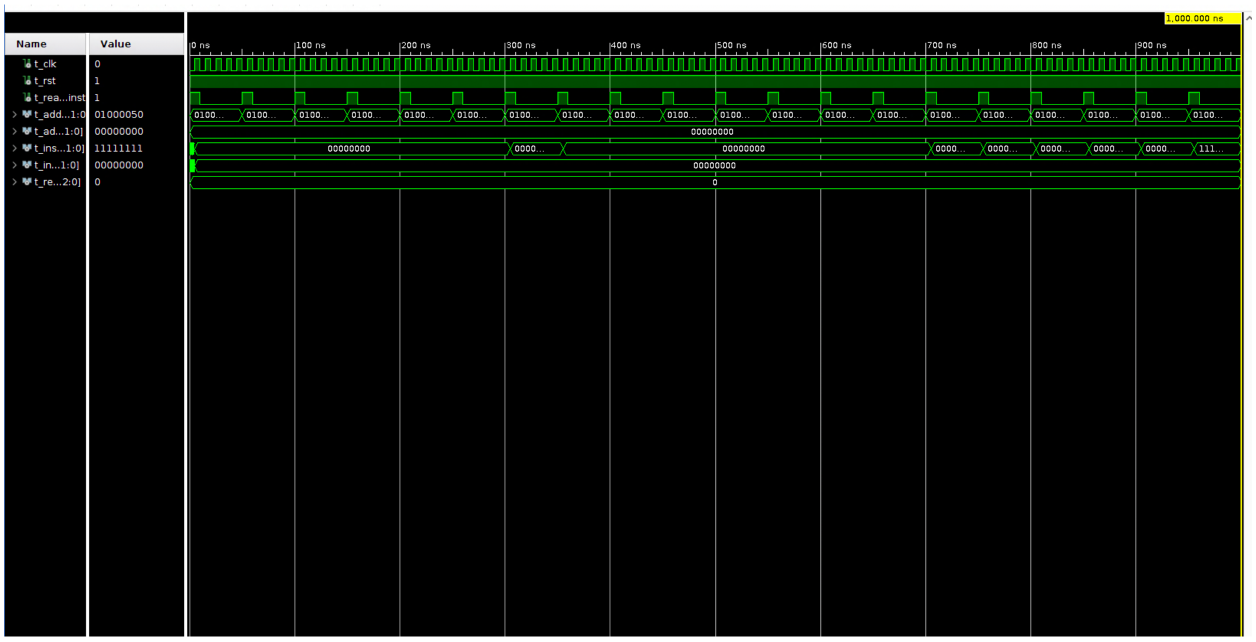
3.5 Register File

This block contains 32 32-bit registers. The register file supports two independent register reads and one register write in one clock cycle. 5 bits are used to address each register. We defined a Register_Type which is an array of 32 32-bit registers, then use process block to read and write from them.



3.6 Instruction Memory

The instruction memory is initialized to contain the program to be executed. Instruction memory width is 4 bytes (32-bits). Our Instruction memory can contain up to 512 32 bits instructions. The instructions are read from a memory file. Before running the code please change the path to the memory file to the path spectific to your computer. The memory file will contain the instructions needed to run. We used readline() from std.textio and hread() from ieee.std_logic_textio to do read from memory file.



3.7 Data Memory:

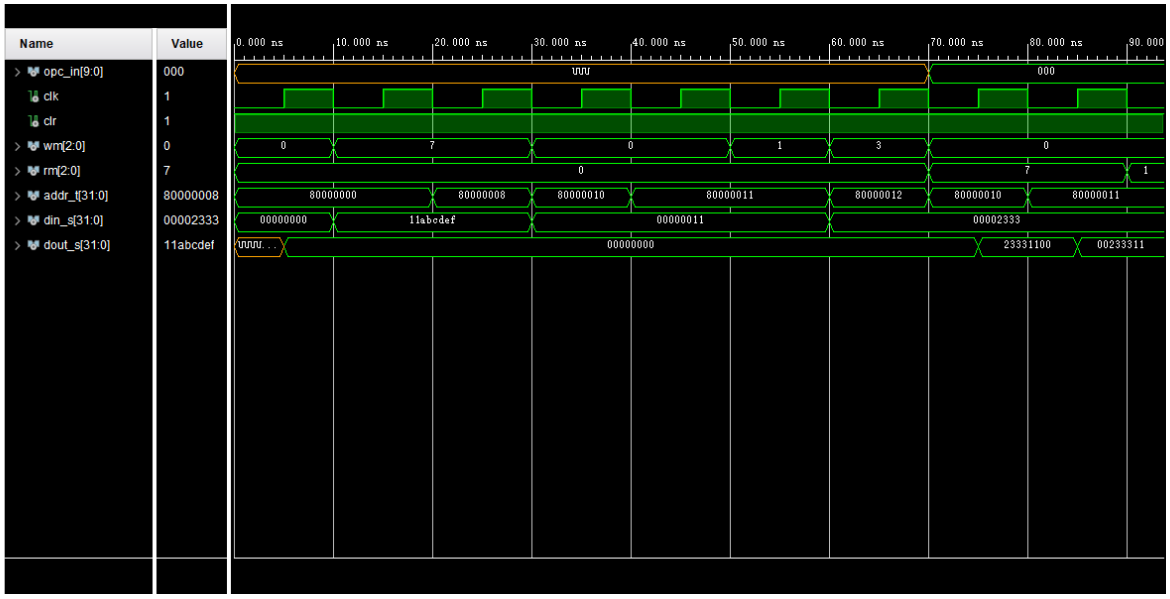
The correctness of Data Memory is verified by testbench that performs storing, reading and resetting operations. The testbench first verifies the correctness by storing data into different places by using various addresses, then reads the data according to the storing addresses and compares the data read with the expected value.

Address: *X"80000000" contains X"11abcdef"*
X"80000008" contains X"11abcdef"
X"80000011" contains X"00000011"
X"80000012" contains X"00002333"

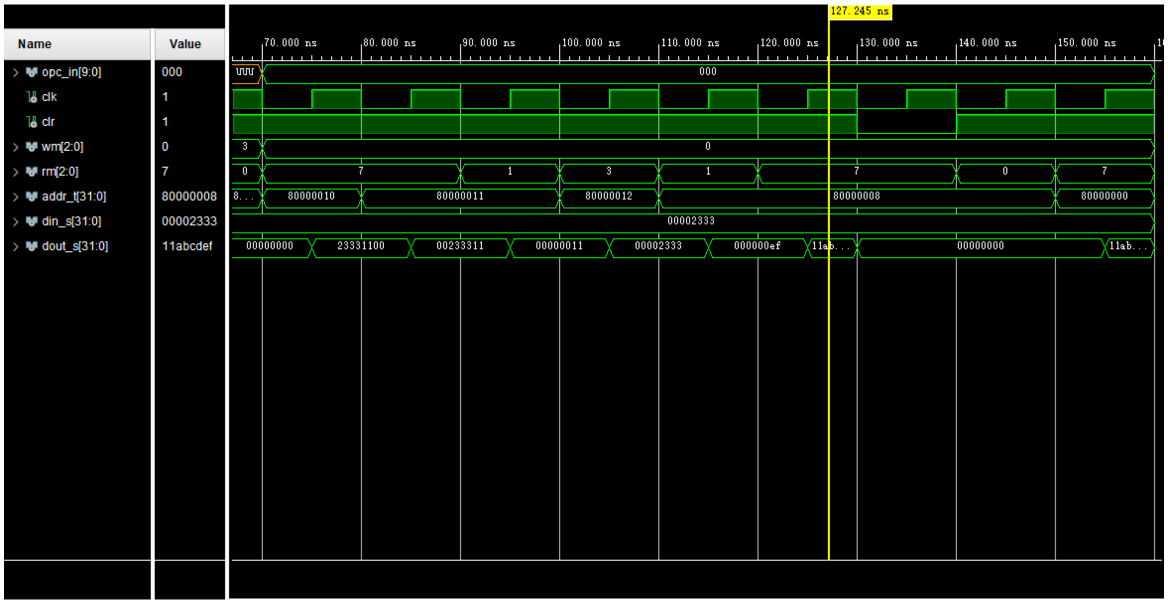
simulation screenshots and explanations of low-level test cases:

All the read and write modules function appropriately, the Data Memory supports all the byte-addressing operations.

The writing state:

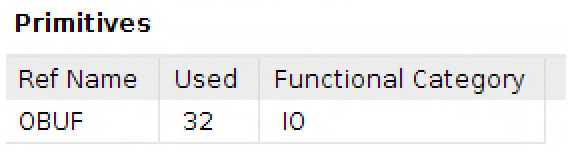


The reading state:



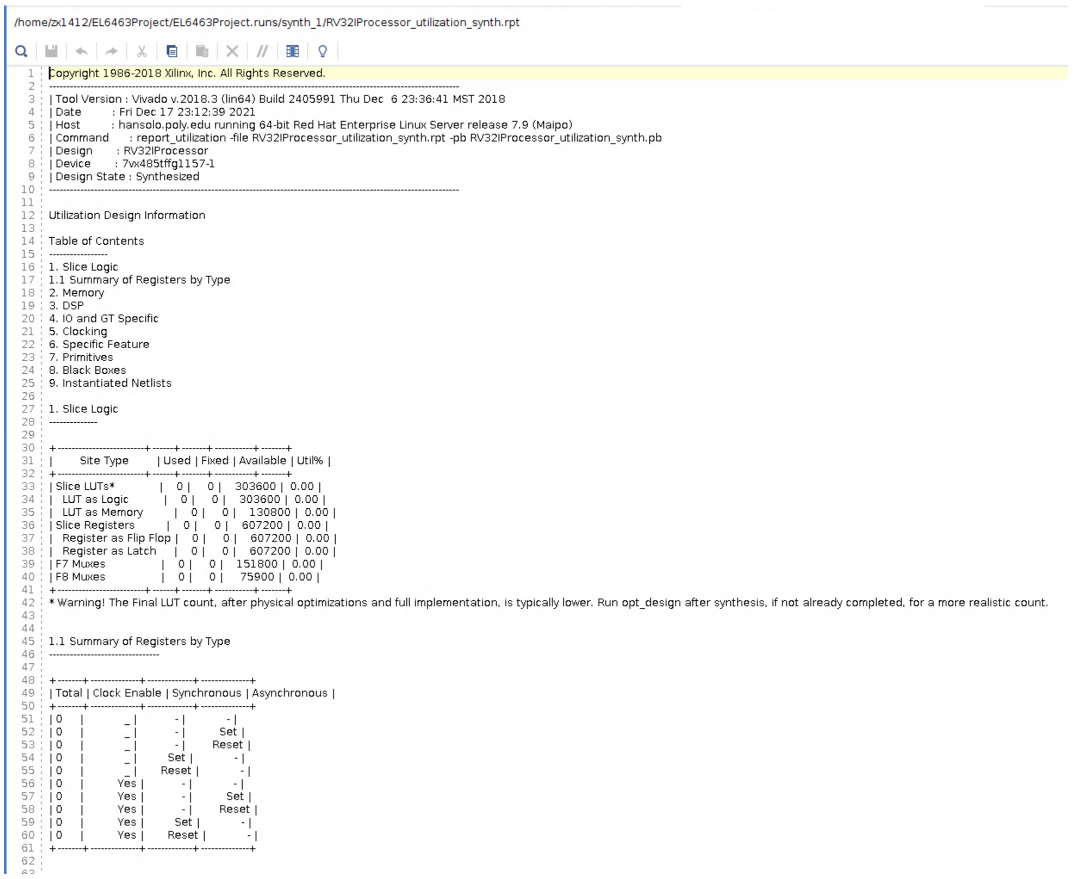
5. Performance and area analysis of design

After complete the program, we did synthesis and get the report from Vivido:



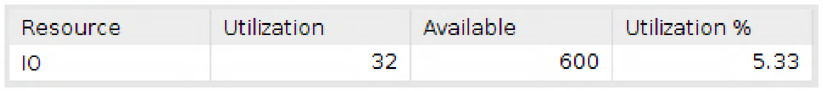
Primitives

Ref Name	Used	Functional Category
OBUF	32	IO

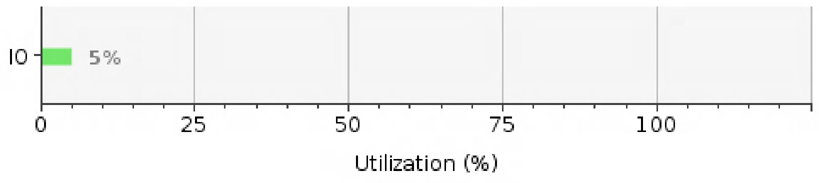


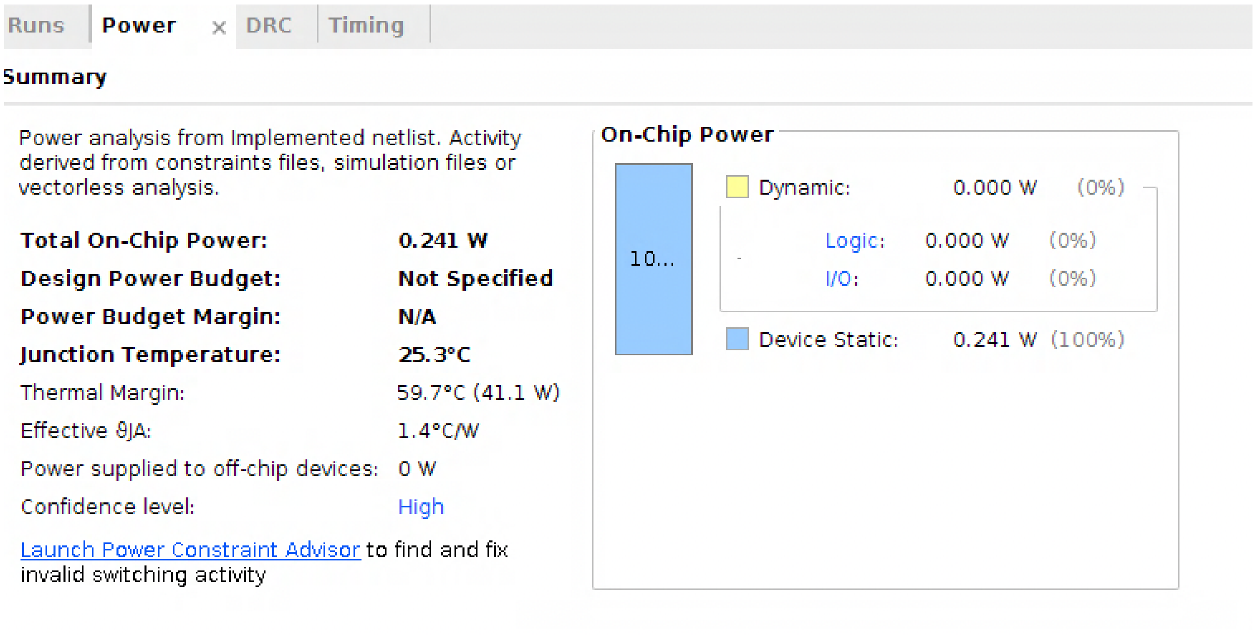
/home/zx1412/EL6463Project/EL6463Project.runs/synth_1/RV32IProcessor_utilization_synth.rpt

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (lin64) Build 2405991 Thu Dec  6 23:36:41 MST 2018
| Date         : Fri Dec 17 23:12:39 2021
| Host         : hansolo.poly.edu running 64-bit Red Hat Enterprise Linux Server release 7.9 (Maipo)
| Command      : report_utilization -file RV32IProcessor_utilization_synth.rpt -pb RV32IProcessor_utilization_synth.pb
| Design       : RV32IProcessor
| Device       : 7vx485tffg1157-1
| Design State : Synthesized
----------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |    0 |     0 |    303600 |  0.00 |
|   LUT as Logic          |    0 |     0 |    303600 |  0.00 |
|   LUT as Memory         |    0 |     0 |    130800 |  0.00 |
| Slice Registers         |    0 |     0 |    607200 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    607200 |  0.00 |
|   Register as Latch     |    0 |     0 |    607200 |  0.00 |
| F7 Muxes                |    0 |     0 |    151800 |  0.00 |
| F8 Muxes                |    0 |     0 |     75900 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
| 0     |          Yes |           - |        Reset |
| 0     |          Yes |         Set |            - |
| 0     |          Yes |       Reset |            - |
+-------+--------------+-------------+--------------+
```



Resource	Utilization	Available	Utilization %
IO	32	600	5.33





Runs | **Power** × | DRC | Timing

Summary

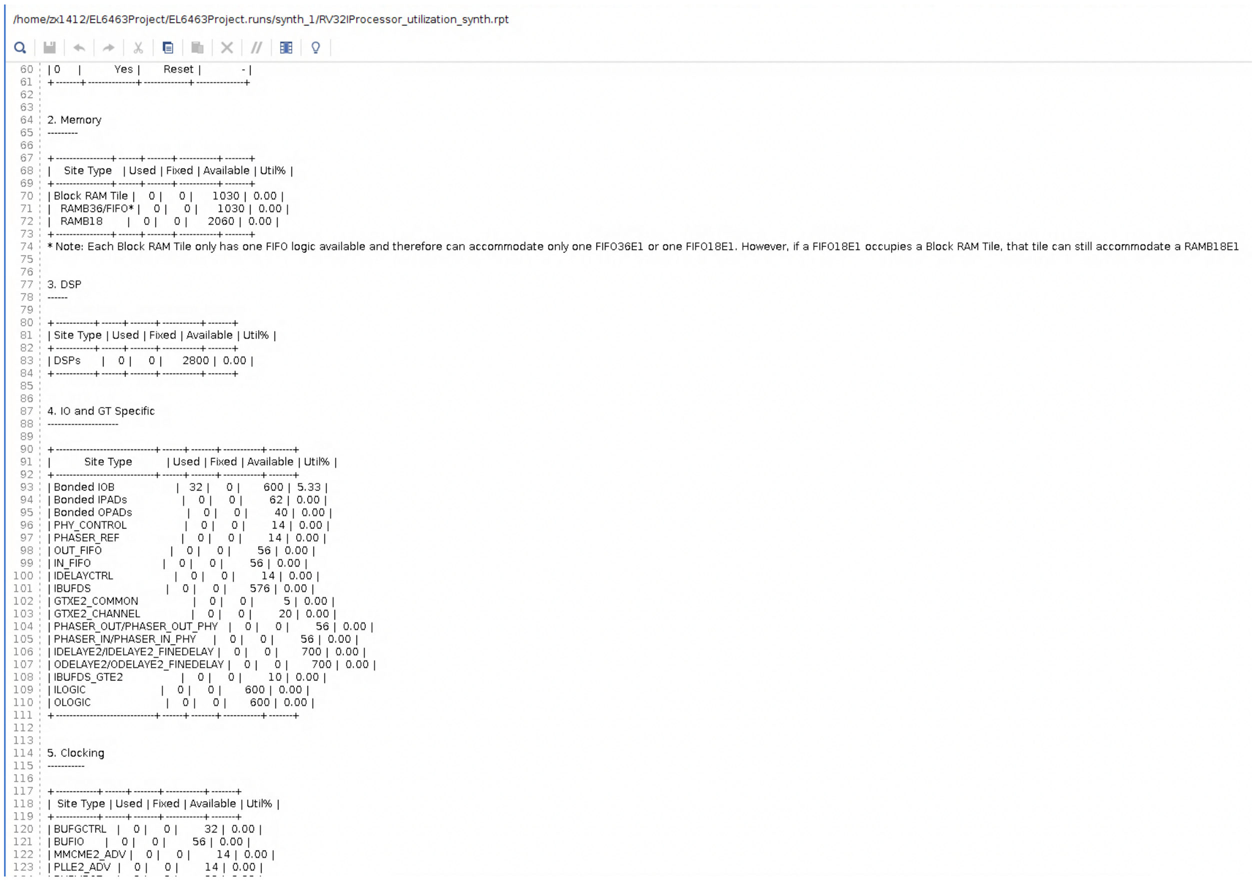
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.241 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.3°C**
Thermal Margin:	59.7°C (41.1 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

Launch Power Constraint Advisor to find and fix invalid switching activity

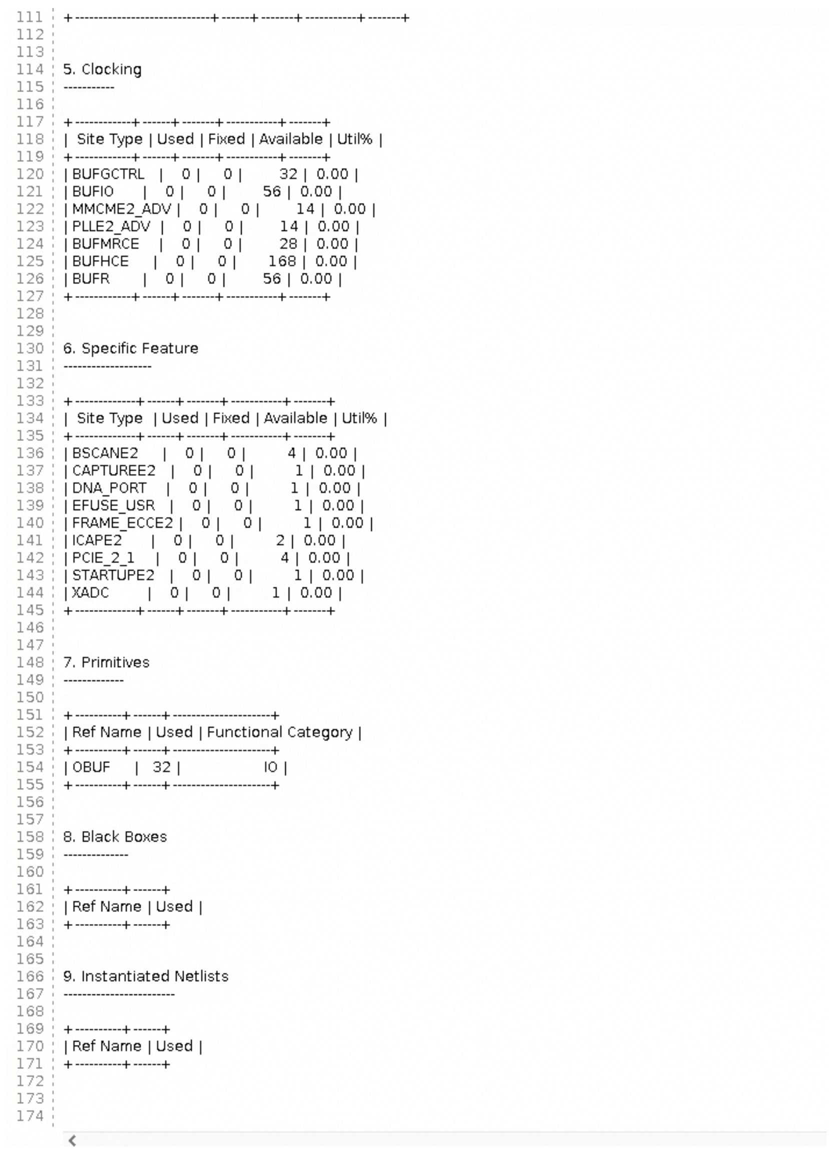
On-Chip Power

Dynamic:	0.000 W	(0%)
Logic:	0.000 W	(0%)
I/O:	0.000 W	(0%)
Device Static:	0.241 W	(100%)



/home/zx1412/EL6463Project/EL6463Project.runs/synth_1/RV32IProcessor_utilization_synth.rpt

```
| 0    |         Yes |      Reset |          - |
+-------+-------------+------------+-------------+


2. Memory
---------

+----------------+------+-------+-----------+-------+
|   Site Type    | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| Block RAM Tile |    0 |     0 |      1030 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |      1030 |  0.00 |
|   RAMB18       |    0 |     0 |      2060 |  0.00 |
+----------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1


3. DSP
------

+-----------+------+-------+-----------+-------+
| Site Type | Used | Fixed | Available | Util% |
+-----------+------+-------+-----------+-------+
| DSPs      |    0 |     0 |      2800 |  0.00 |
+-----------+------+-------+-----------+-------+


4. IO and GT Specific
--------------------

+-----------------------------+------+-------+-----------+-------+
|          Site Type          | Used | Fixed | Available | Util% |
+-----------------------------+------+-------+-----------+-------+
| Bonded IOB                  |   32 |     0 |       600 |  5.33 |
| Bonded IPADs                |    0 |     0 |        62 |  0.00 |
| Bonded OPADs                |    0 |     0 |        40 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |        14 |  0.00 |
| PHASER_REF                  |    0 |     0 |        14 |  0.00 |
| OUT_FIFO                    |    0 |     0 |        56 |  0.00 |
| IN_FIFO                     |    0 |     0 |        56 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |        14 |  0.00 |
| IBUFDS                      |    0 |     0 |       576 |  0.00 |
| GTXE2_COMMON                |    0 |     0 |         5 |  0.00 |
| GTXE2_CHANNEL               |    0 |     0 |        20 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        56 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        56 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       700 |  0.00 |
| ODELAYE2/ODELAYE2_FINEDELAY |    0 |     0 |       700 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |        10 |  0.00 |
| ILOGIC                      |    0 |     0 |       600 |  0.00 |
| OLOGIC                      |    0 |     0 |       600 |  0.00 |
+-----------------------------+------+-------+-----------+-------+


5. Clocking
-----------

+------------+------+-------+-----------+-------+
|  Site Type | Used | Fixed | Available | Util% |
+------------+------+-------+-----------+-------+
| BUFGCTRL   |    0 |     0 |        32 |  0.00 |
| BUFIO      |    0 |     0 |        56 |  0.00 |
| MMCME2_ADV |    0 |     0 |        14 |  0.00 |
| PLLE2_ADV  |    0 |     0 |        14 |  0.00 |
```



```
+-------------------------+------+-------+-----------+-------+


5. Clocking
-----------

+------------+------+-------+-----------+-------+
|  Site Type | Used | Fixed | Available | Util% |
+------------+------+-------+-----------+-------+
| BUFGCTRL   |    0 |     0 |        32 |  0.00 |
| BUFIO      |    0 |     0 |        56 |  0.00 |
| MMCME2_ADV |    0 |     0 |        14 |  0.00 |
| PLLE2_ADV  |    0 |     0 |        14 |  0.00 |
| BUFMRCE    |    0 |     0 |        28 |  0.00 |
| BUFHCE     |    0 |     0 |       168 |  0.00 |
| BUFR       |    0 |     0 |        56 |  0.00 |
+------------+------+-------+-----------+-------+


6. Specific Feature
-------------------

+-------------+------+-------+-----------+-------+
|  Site Type  | Used | Fixed | Available | Util% |
+-------------+------+-------+-----------+-------+
| BSCANE2     |    0 |     0 |         4 |  0.00 |
| CAPTUREE2   |    0 |     0 |         1 |  0.00 |
| DNA_PORT    |    0 |     0 |         1 |  0.00 |
| EFUSE_USR   |    0 |     0 |         1 |  0.00 |
| FRAME_ECCE2 |    0 |     0 |         1 |  0.00 |
| ICAPE2      |    0 |     0 |         2 |  0.00 |
| PCIE_2_1    |    0 |     0 |         4 |  0.00 |
| STARTUPE2   |    0 |     0 |         1 |  0.00 |
| XADC        |    0 |     0 |         1 |  0.00 |
+-------------+------+-------+-----------+-------+


7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| OBUF     |   32 |                  IO |
+----------+------+---------------------+


8. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+


9. Instantiated Netlists
------------------------

+----------+------+
| Ref Name | Used |
+----------+------+



```

In the report, we can see that the utilization of the register and mux are good, there are still enough registers and mux that can be used. However in our design, there is a main problem which is the ALU. The efficiency of ALU is low, because in ALU, we have too many small units in ALU and that causes the low efficiency. These designs will cause long access time and big power consumption. Also small units design will need more space when making design into hardware.

6. Description of high-level test-cases in assembly

In the high-level test, we use 2 programs in assembly code: Find the Nth Fibonacci Number and Find the recursive sum of number N.

For the program of Finding the recursive sum of number N, the assembly code is:

```
START:
        add t3,x0,x0
        add t31,x0,x0
        add t30,x0,x0
        add t29,x0,x0
        add t4,x0,x0
        addi t1,x0,10
        jal t31, MAIN
HALT:
        add t3,x0,x0
        add t31,x0,x0
        add t30,x0,x0
        add t29,x0,x0
        add t4,x0,x0
        ECALL
MAIN:
        addi t3, t31, 0
        jal t31, PUSH
        addi t4, x0, 2
        blt t1, t4, ADD_ONE
        add t3, x0, x1
        jal t31, PUSH
        addi t1, t1, -1
        jal t31, MAIN
        jal t31, POP
        add t29, t29, t3
        addi $3, x0, 0
        beq t3, x0, ADD
ADD_ONE:
        addi t29, x0, 1
ADD:
        jal t31, POP
        add t31, x0, t3
        jalr x0, t31, 0

PUSH:
        addi t30,t30,4
        sw   t30,t3, 0
        jalr x0,t31,0
POP:
        lw t3, t30,0
        addi t30,t30,-4
        jalr x0,t31,0

SWAP:
        add t3,x0,t1
        add t1,x0,t2
        add t2,x0,t3
        add t3,x0,x0
        jalr x0,t31,0
```

In this program, After Initialization, we define t31 as the return address, t30 as the pointer of the stack, t29 as the return value, t1 as the input of N. When the program starts, it will jump to the main function and according to the number of N, it will push a new process in the stack, until it reaches the number of recursive calls. Then it will do the addition for every recursive process, after addition, the result will be saved and the recursive process will be popped. When all the recursive processes are popped, the process is finished and the result will be in the t29. The instruction code after translated from the assembly code are:

```
00000000000000000000000110110011
00000000000000000000111110110011
00000000000000000000111100110011
00000000000000000000111010110011
00000000000000000000001000110011
00000000101000000000000010010011
00000001110000000000111111101111
00000000000000000000000110110011
00000000000000000000111110110011
00000000000000000000111100110011
00000000000000000000111010110011
00000000000000000000001000110011
00000000000000000000000001110011
00000000000011111000000110010011
00000011110000000000111111101111
00000000001000000000001000010011
00000010010000000110000100110001 1
00000000000100000000000110110011
00000010110000000000111111101111
```

```
11111111111100001000000010010011
11111110010111111111111111101111
00000010110000000000111111101111
00000001110100011000111010110011
00000000000000000000000110010011
00000000001100000000000101100011
00000000000000000000111010010011
00000001100000000000111111101111
00000000001100000000111110110011
00000000000011111000000001100111
00000000010011101111111100010011
00000000001111110010000000100011
00000000000011111000000001100111
00000000000011110010000110000011
11111111110011110000111100010011
00000000000011111000000001100111
00000000000100000000000110110011
00000000000000010000000010110011
00000000000000011000000100110011
00000000000000000000000110110011
00000000000011111000000001100111
```

For the Program of Find the Nth Fibonacci Number, the assembly code are:

```
addi t5,x0, 6
addi t1,x0,1
addi t2,x0,0
addi t3,x0,0
addi t4,x0,1
FOR:
   add t3, t1, t2
   add t2, x0, t1
   add t1, x0, t3
   addi t4, t4, 1
   blt t4, t5, FOR
HALT:
   ecall
```

In this program, N is stored in t5, and we have t1 and t2 as calculation variables, the answer is stored in t3, the counter of the For loop i is stored in t4. First, we initialize all the variables we need, then we get into a For loop. In this loop, we are finding the Fibonacci Number of the current i and store the results to calculate variable t1 and t2. Everytime it gets into a new iteration, we will add the t1 and t2 together to get a new Fibonacci Number into t3. When the iteration number is larger or equal to the max iteration number i, the program will stop and the result is stored in t3. The instruction code translated from the assembly code are:

```
00000000011000000000001010010011
00000000000100000000000010010011
00000000000000000000000100010011
00000000000000000000000110010011
00000000000100000000001000010011
00000000000010001000000110110011
00000000000100000000000100110011
00000000001100000000000010110011
00000000000100100000001000010011
11111110010100100100100011100011
00000000000000000000000001110011
```

7. Future Improvement and Optimization

Our works, though very close to our expectations, still need further improvements. For example, there are still a few glitches that we do not have time to fix, and our test cases may not be comprehensive enough to test all the possible situations. Thus, further refinement of our project is still needed.

Additionally, due to the inconvenience of meeting in person under the context of the pandemic, we have to design our codes in a highly modular way to assemble all the pieces of codes that come from different group members. This design style caused a decrease in performance. Many modules can be integrated to reduce the access time of our project. For instance, the branching module and the Control Unit can be merged into one module.

8. Videos

https://drive.google.com/drive/folders/1oDY3a31Zh_6S19pMCgP6IekH2gpGUZ_c?usp=sharing