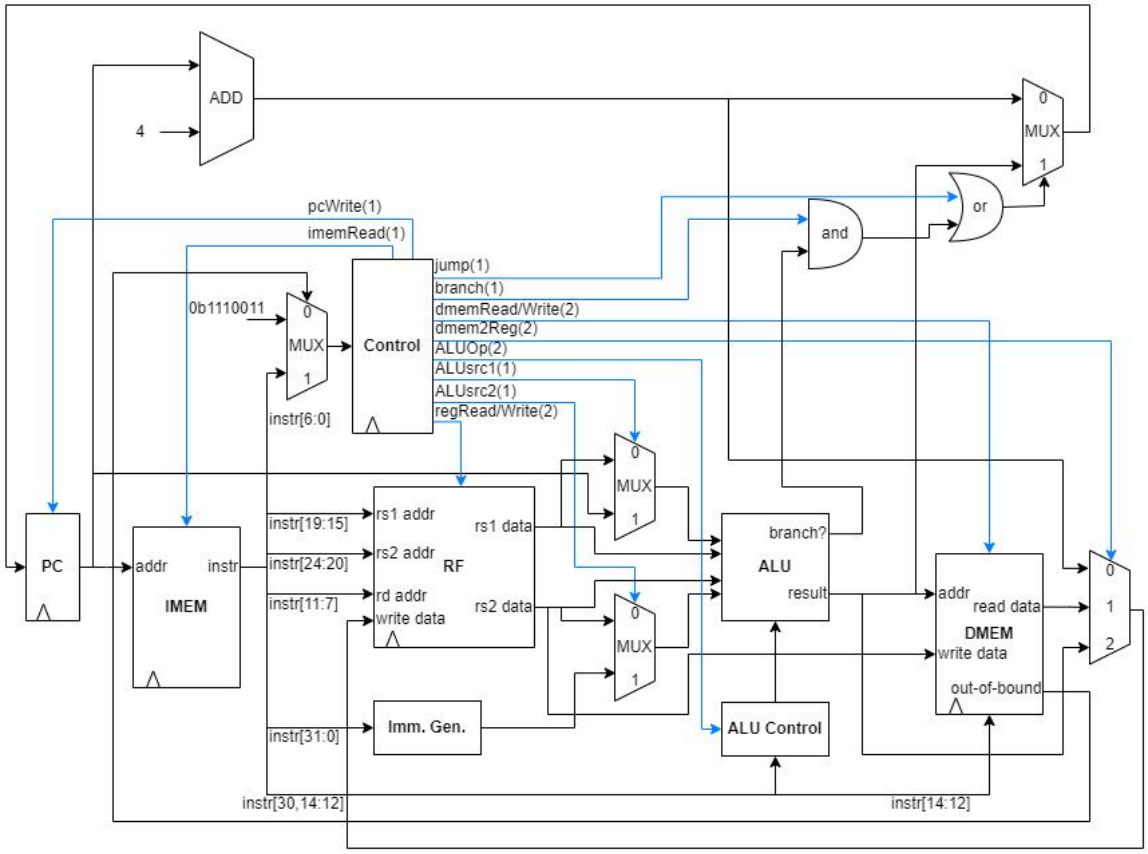
NYU-6463-RV321 Processor Design Project Report

Group 03
Members: Xiao Ding(xd2076), Xinran Tang(xt2191), Qing Xiang(qx657)
N-numbers: N18790475, N10257233, N11987251

Overview:

In this project, we have designed a functional processor based on the provided RISC-V ISA. It is capable of executing single instructions as well as complex functions written in RISC-V assembly code. The design is mainly written in VHDL, while testbenches are written in Verilog. In this report, we will explain details of our design and simulations with different functions. We will also show videos(separate files) of its implementation of complex functions on a FPGA board. Besides, we will analyze our design and reflect on how it can be improved.

Design Datapath:



The datapath includes eight main components:

Program Counter(PC): stores the current 32-bit instruction address.

Instruction Memory(IMEM): stores all 32-bit instructions that can be read. And

Control Unit(Control): contains a finite state machine that produces control signals based on different stages.

Register File(RF): contains 32 32-bit registers.

Immediate Generator(Imm. Gen.): generates 32-bit immediate values according to the immediate field of the instruction.

ALU Control Unit(ALU Control): selects operation type for ALU.

Arithmetic Logic Unit(ALU): performs selected operation on the two operands; and also performs branch comparison on the two register values when needed.

Data Memory(DMEM): stores all 32-bit data including special spaces for switches, LEDs, and N-numbers.

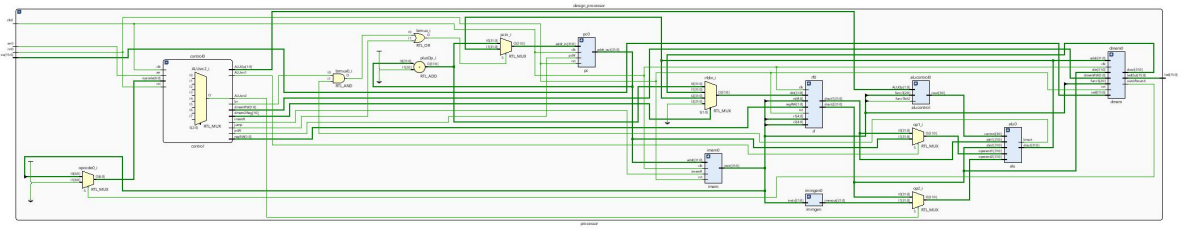
Besides, there are **5 multiplexers**, **1 adder**, **1 AND gate** and **1 OR gate**.

There are inputs and outputs for implementation on FPGA:

16-bit switches input: writes value to designated space for switches in data memory.

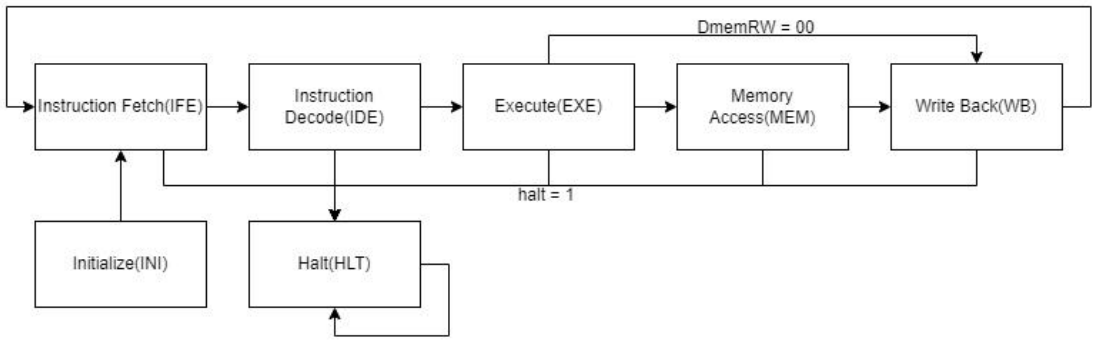
16-bit LEDs output: displays value stored in designated space for LEDs in data memory.
CLK100MHZ: board default clock used for clock signal.
BtnL: board left button used for reset signal.
BtnR: board right button used for enable signal.

Processor Schematic:



The generated schematic shows no unexpected components. The processor is multi-cycled and without pipeline, so each instruction will take several clock cycles depending on its type. The complete execution of an instruction is divided into stages that each take one clock cycle. Components are only active when they are needed for the specific stage, thus saving power. Memory access stage is often skipped when not needed, thus improving speed. Please see below for more detailed explanation of the stages.

Finite State Machine in Control Unit:



The diagram shows the basic cycle of the FSM(Moore). Each transition happens at the rising edge of a clock cycle.
The output control signals of each stage is determined by the current opcode, which is part of the instruction read in IFE stage. The purpose of IDE stage is to wait for the change of control signals to take effect. Then in EXE stage the processor reads from the register file and executes operations in the ALU. MEM stage is skipped if dmemRW = 00, meaning there is no memory read or write needed for the current instruction. WB stage updates register file and PC. Then the processor enters IFE stage again.
INI stage is used only for the very first time to get IMEM ready. After the processor transitions into IFE stage, it never goes back to INI stage. Even upon reset, the processor enters IFE stage instead of INI stage.
HLT stage is used to halt the program. Once the processor enters this stage, it will stay in this stage until it is reset. Signal halt determines whether the processor should enter HLT stage in the next clock cycle. To make sure the program halts every time when needed, there is a conditional to check if halt = 1 in every stage. There are 3 ways to make the program halt: a halt instruction is fetched and decoded, address to read from IMEM is out of bound(IMEM will return a halt instruction), and address to read from or write to DMEM is out of bound(DMEM will return outofbound = 1 that changes the input opcode to control for the next stage. This is done by a multiplexer shown in the datapath above).

Simulations:
We used 5 test set to test the performance of the whole CPU. First, the RISC-V instructions (translated into hexadecimal machine code) which will be read into the instruction memory will be read from a memory file “xxx(the name of the file).mem”. In line 55 of design source file “**imem.vhdl**”, there is a signal declaration “signal rom_words: instr_rom := instr_rom_readfile("**xxx.mem**");”, in which user can change the input memory file by changing the file name “**xxx.mem**”

here into the correct file name of 5 test cases (including **“main2.mem”, “main7.mem”, “special_addr.mem”, “bubble2_new2.mem”, “RC5_final3.mem”**) which is corresponding to the current top testbench (**“tb_main2_processor”, “tb_main7_processor”, “tb_special”, “tb_bubble2”, “tb_RC”**).

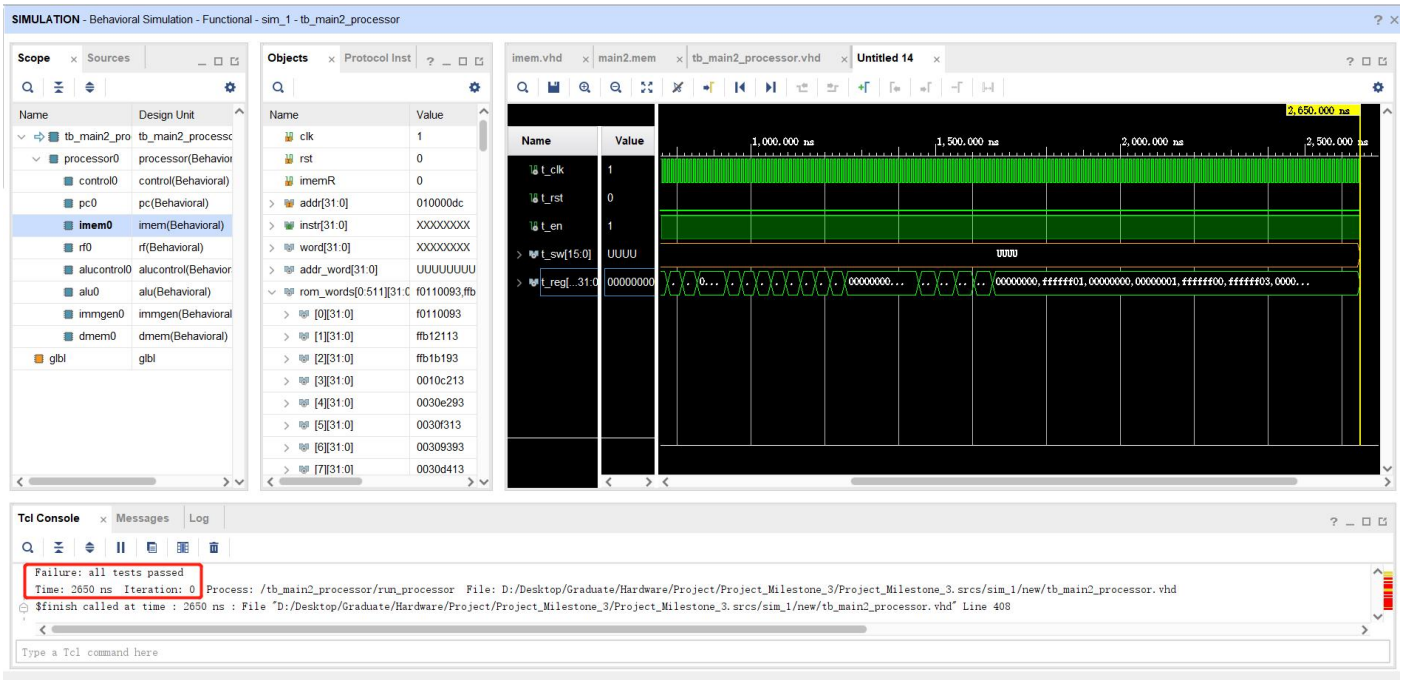
Tips: Every time before running the simulation, user should type in “reset_project” in Tcl Console and run, then right click “SIMULATION”, click “Reset Behavioral Simulation”. This will be useful to guarantee that the simulation runs successfully without any errors.

Low-level test case 1 (tb_main2_processor):

Testbench **tb_main2_processor** is used to test the functions of instructions without branch or jump. The instructions tested in this testbench are translated into hex machine code and stored in file **main2.mem**. Instructions tested here include: ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND, LUI, AUIPC, ADDI, LUI, SB, SH, SW, LB, LH, LW, LBU, LHU.

All of test instructions are loaded to the instruction memory by reading memory file “**main2.mem**”. After running the instructions, the data stored in registers x1 to x28 (by instructions) will be output as a signal “**reg_out[0:31][0:31]** (named in RegFile.vhd, in processor.vhd is “**regfile[0:31][0:31]**”)” and compared by a set of desired output (some are gotten by an online RISC-V simulator (reference below at end of the report), some are calculated by ourselves).

Simulation result:



In the “Scope” and “Object” window, it showed that the instructions (read from “main2.mem”) are stored in “rom_words” in the instruction memory.

RISC-V instructions in **main2.mem**: (before translated into hex machine code, same as “rom_words” above)

ADDI x1, x2, -255

SLTI x2, x2, -5

SLTIU x3, x2, -5

XORI x4, x1, 1

ORI x5, x1, 3

ANDI x6, x1, 3

SLLI x7, x1, 3

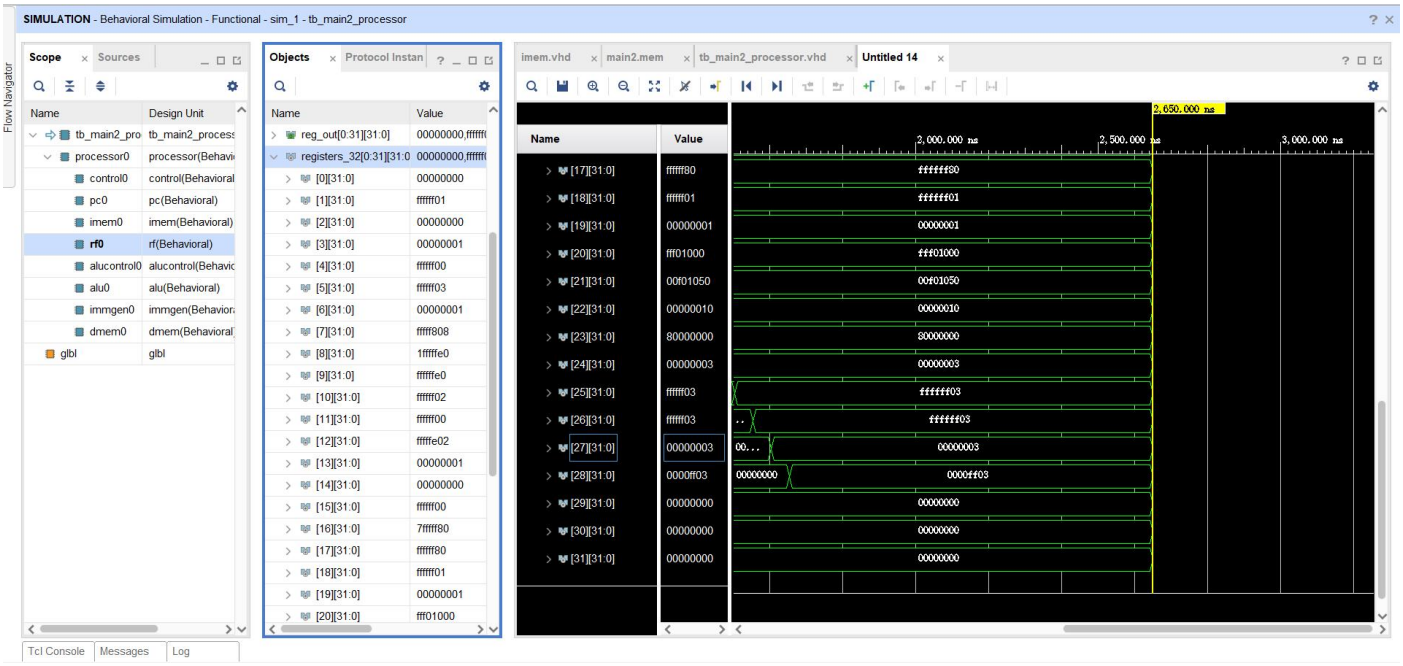
SRLI x8, x1, 3

SRAI x9, x1, 3

ADD x10, x1, x3

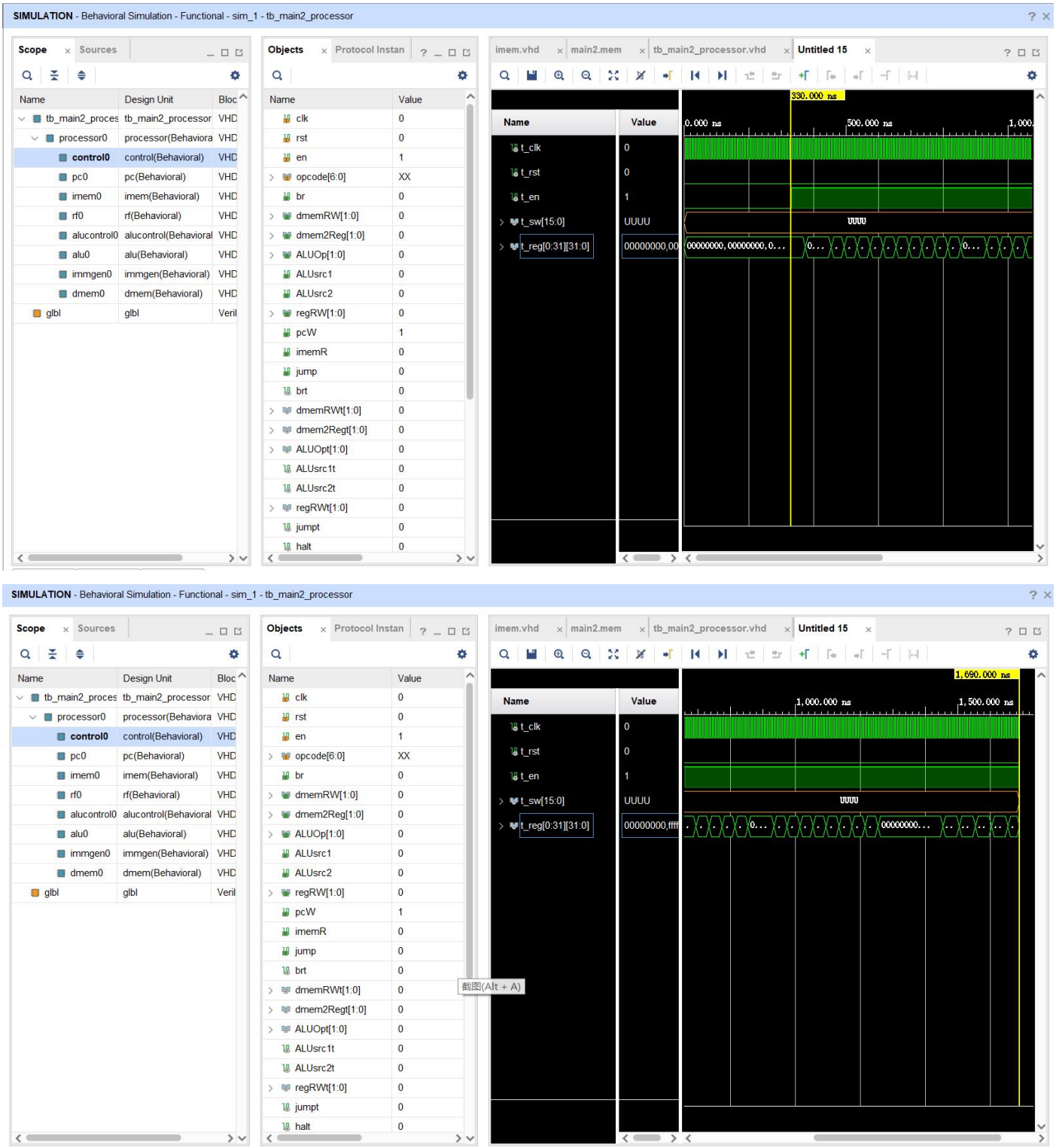
SUB x11, x1, x3
SLL x12, x1, x3
SLT x13, x1, x3
SLTU x14, x1, x3
XOR x15, x1, x3
SRL x16, x1, x3
SRA x17, x1, x3
OR x18, x1, x3
AND x19, x1, x3
LUI x20, -255
AUIPC x21, -255
ADDI x22, x0, 16
LUI x23, 524288
SB x5, 4(x23)
SH x5, 8(x23)
SW x5, 12(x23)
LB x24, 4(x23)
LH x25, 8(x23)
LW x26, 12(x23)
LBU x27, 12(x23)
LHU x28, 12(x23)

The result stored in the register file after running this test set:



31 signals are too much to show in a single window “Objects” or “Untitled 14”. So, I showed registers 0 to 20 at “Objects” window (left), and showed registers 17 to 31 at “Untitled 14” window (right).

Run time:



(On second figure, the opcode input in control is “XX”, which means that there is not next instruction. All instructions finished. The next

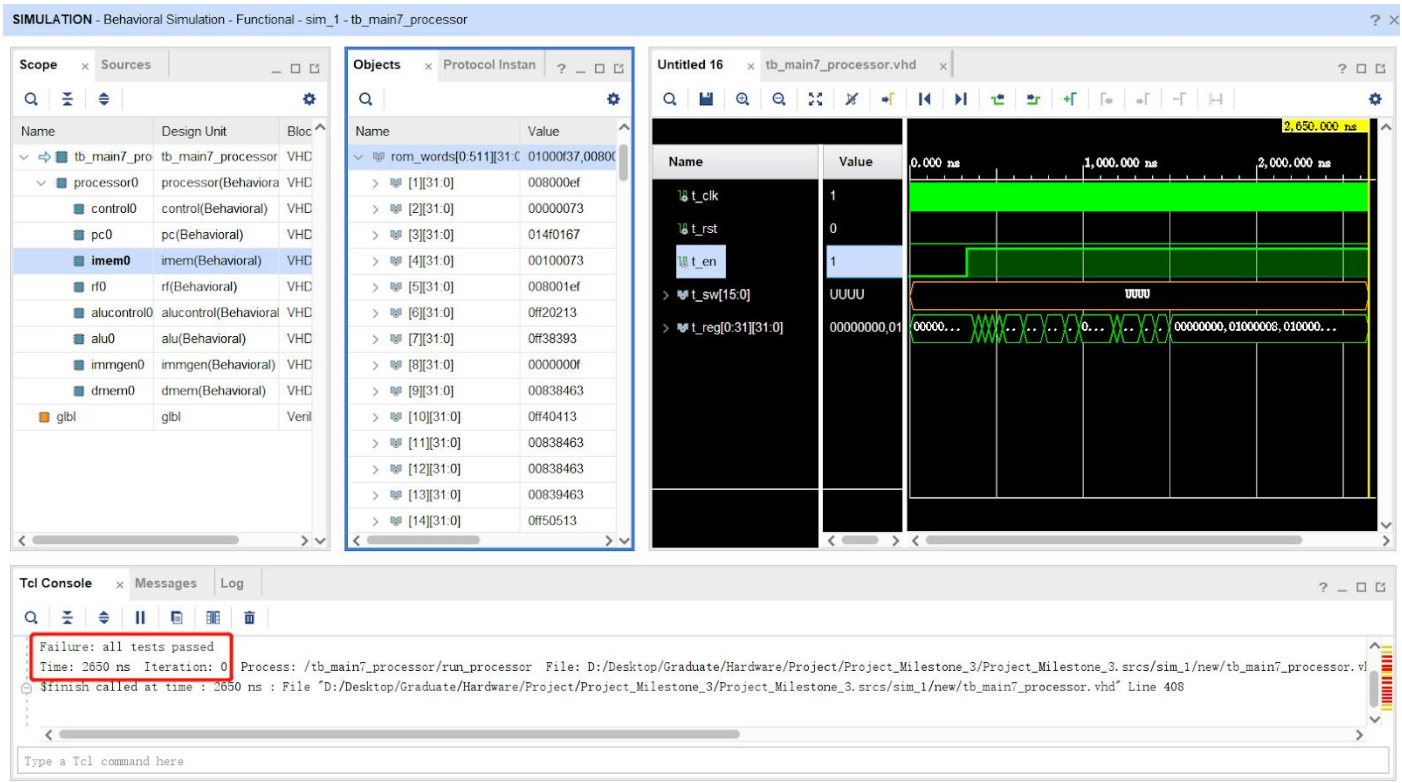
The total time for the running of all 31 instructions is 1360ns (330ns to 1690ns, other time are used to compare the test result stored in registers with the desired result), which is 136 clock cycles. It means that 4.39 clock cycles on average is needed for each instruction. The reason is that the load/store instructions take 5 clock cycles (for 5 state of the Control component), and other instructions take 4 clock cycles.

Low-level test case 2 (tb_main7_processor):

Testbench **tb_main7_processor** is used to test the functions of instructions mainly about branch or jump. The instructions tested in this testbench are translated into hex machine code and stored in file **main7.mem**. Instructions tested here include: LUI, JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, ECALL, EBREAK, FENCE

All of test instructions are loaded to the instruction memory by reading memory file "**main7.mem**". After running the instructions, the data stored in registers x1 to x18 and x30 (by instructions) will be output as a signal "**reg_out[0:31][0:31]** (named in RegFile.vhd, in processor.vhd is "**regfile[0:31][0:31]**")" and compared by a set of desired result.

Simulation result:



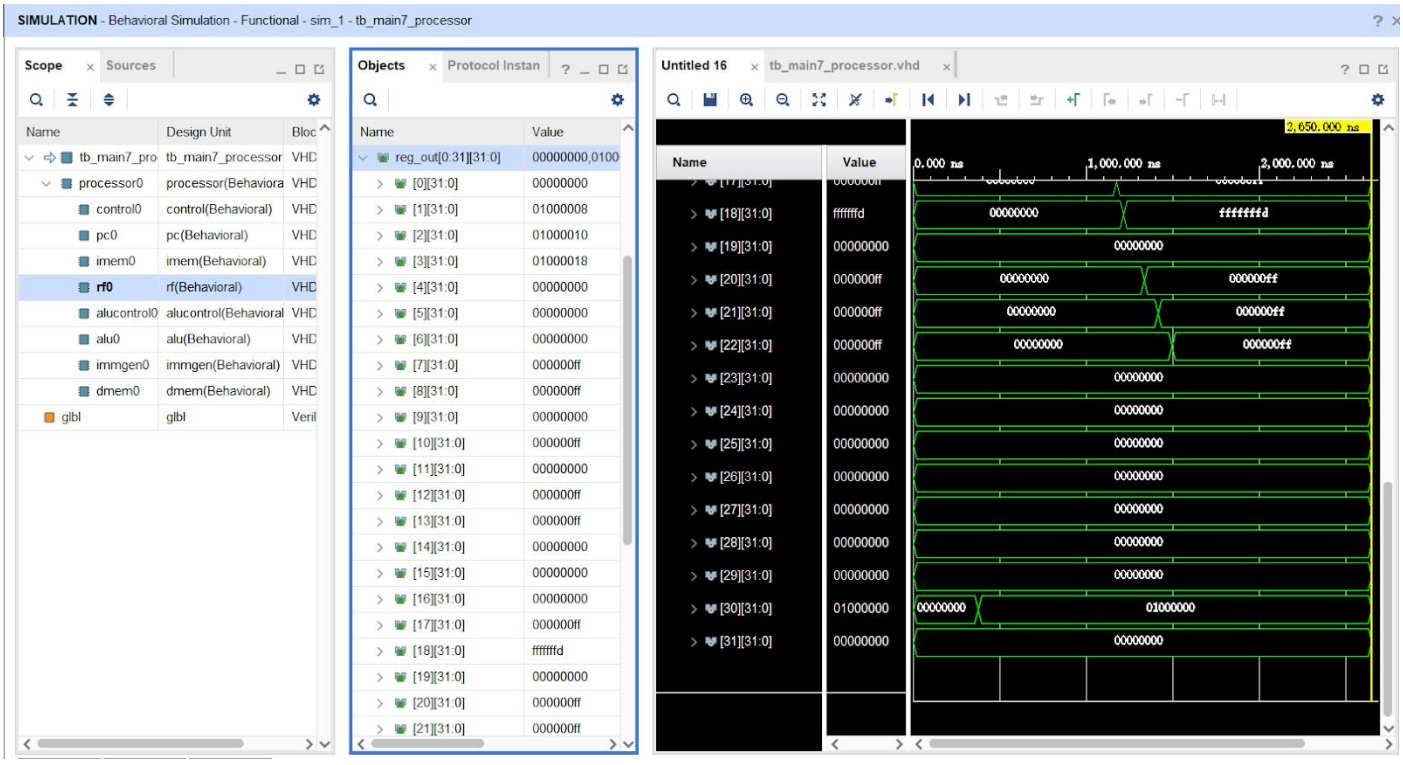
In the "Scope" and "Object" window, it showed that the instructions (read from "main7.mem") are stored in "rom_words" in the instruction memory.

RISC-V instructions in **main7.mem**: (before translated into hex machine code, same as "rom_words" above)

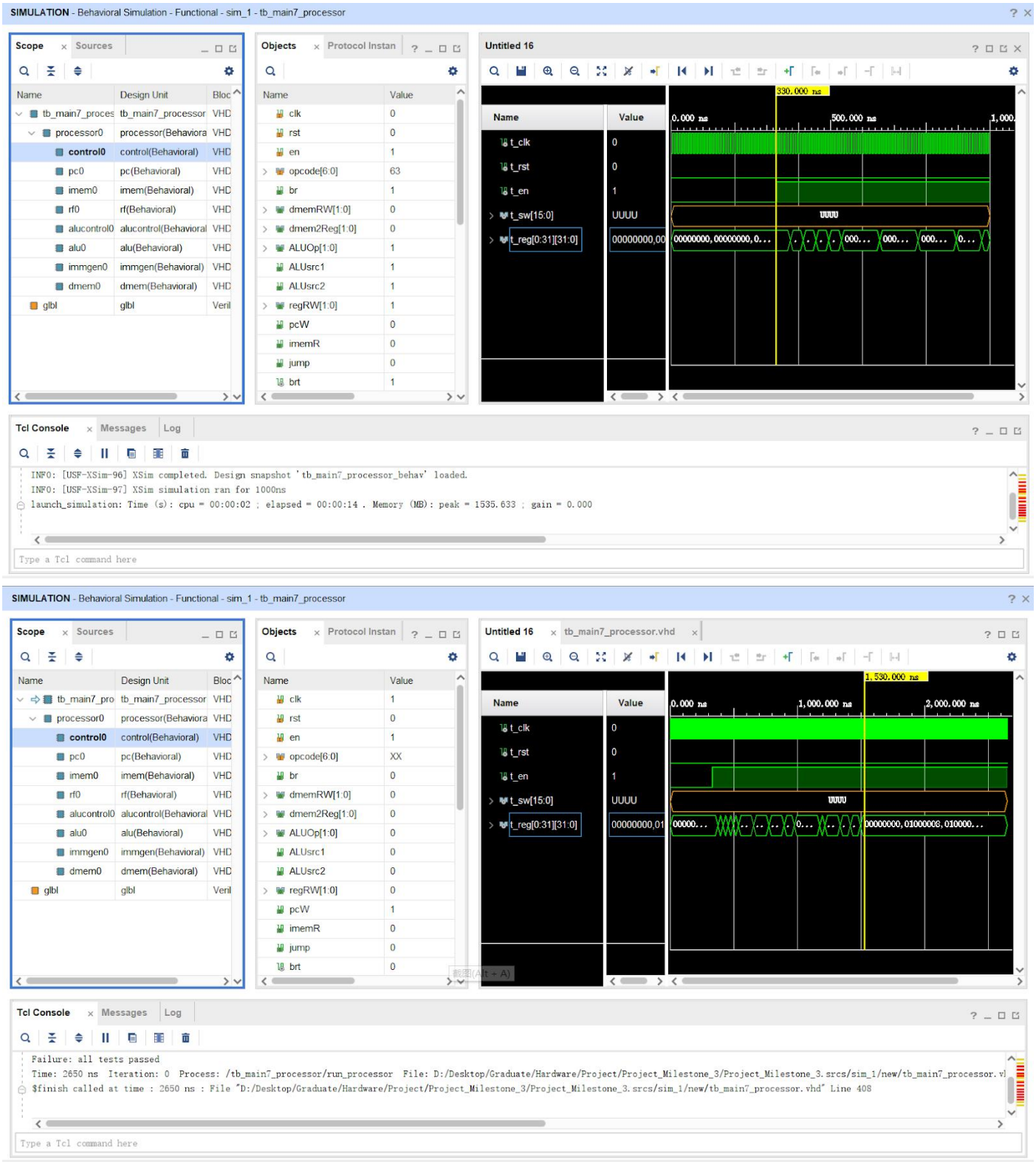
LUI x30, 4096
JAL x1, 8
ECALL
JALR x2, x30, 20
EBREAK
JAL x3, 8
ADDI x4, x4, 255
ADDI x7, x7, 255
FENCE
BEQ x7, x8, 8
ADDI x8, x8, 255
BEQ x7, x8, 8
ADDI x9, x9, 255
BNE x7, x8, 8
ADDI x10, x10, 255
BNE x7, x9, 8
ADDI x11, x11, 255

BLT x7, x8, 8
ADDI x12, x12, 255
BLT x7, x9, 8
ADDI x13, x13, 255
BLT x9, x7, 8
ADDI x14, x14, 255
BGE x7, x8, 8
ADDI x15, x15, 255
BGE x7, x9, 8
ADDI x16, x16, 255
BGE x9, x7, 8
ADDI x17, x17, 255
ADDI x18, x18, -3
BLTU x7, x18, 8
ADDI x19, x19, 255
BLTU x7, x8, 8
ADDI x20, x20, 255
BLTU x18, x7, 8
ADDI x21, x21, 255
BGEU x7, x18, 8
ADDI x22, x22, 255
BGEU x7, x8, 8
ADDI x23, x23, 255
BGEU x18, x7, 8
ADDI x24, x24, 255

The result stored in the register file after running this test set:



31 signals are too much to show in a single window “Objects” or “Untitled 14”. So, I showed registers 0 to 20 at “Objects” window (left), and showed registers 17 to 31 at “Untitled 14” window (right).
Run time:



The total time for the running of all 30 instructions is 1200ns (330ns to 1530ns, other time are used to compare the test result stored in registers with the desired result), which is 120 clock cycles. It means that 4.00 clock cycles on average is needed for each instruction. The reason is that all of instructions take 4 clock cycles. There are no load/store instructions.

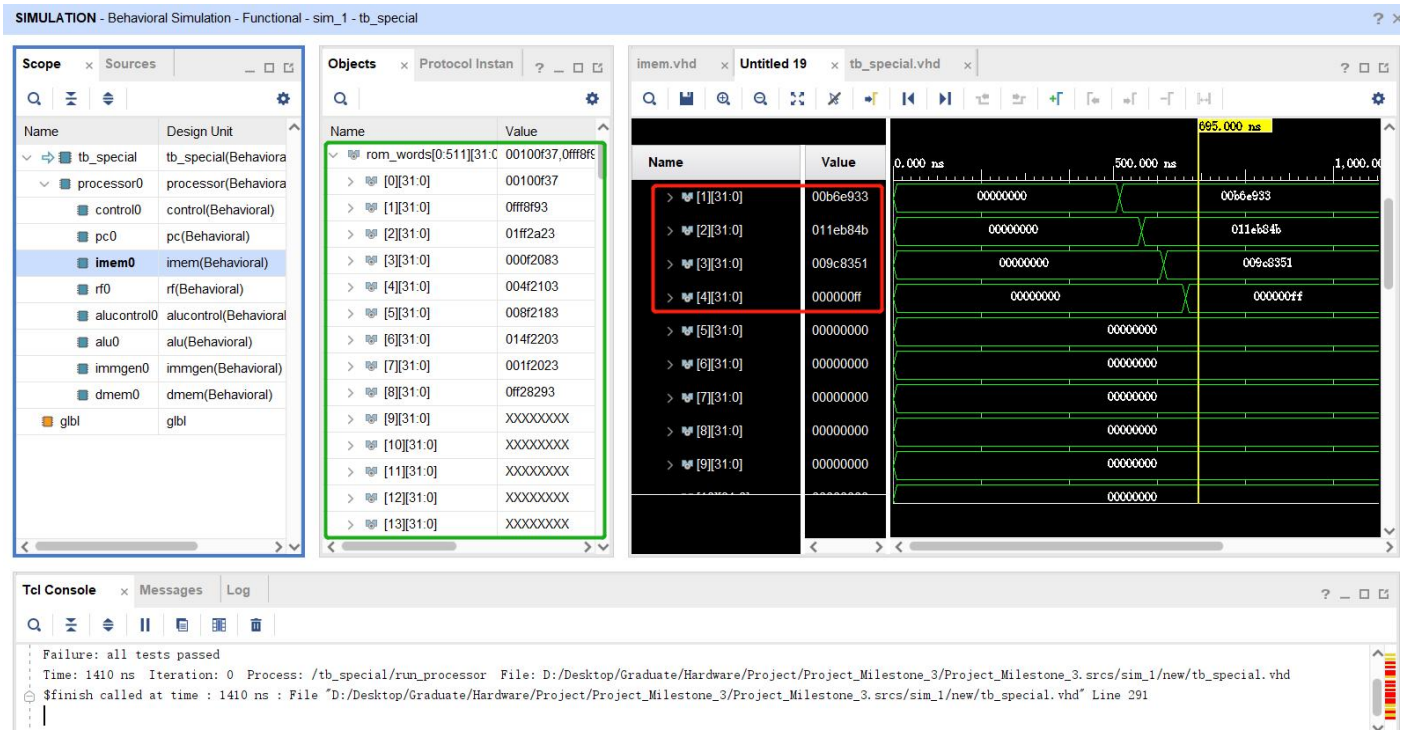
Low-level test case 3 (tb_special):

Testbench **tb_special** is used to test to load data from and store data into some special address of the data memory (from

0010000 to 0010014). The instructions tested in this testbench are translated into hex machine code and stored in file **special_addr.mem**.

After running the instructions, the data stored in registers x1 to x4 (by instructions) will be output as a signal "**reg_out[0:31][0:31]** (named in RegFile.vhd, in processor.vhd is "**regfile[0:31][0:31]**")" and compared by a set of desired output.

Simulation result:



In the "Scope" and "Object" window, it showed that the instructions (read from "special_addr.mem") are stored in "rom_words" in the instruction memory.

In the "Untitled 19 window", it showed the result stored in registers x1 to x4.

RISC-V instructions in **special_addr.mem**: (before translated into hex machine code, same as "rom_words" above)

lui x30, 256

addi x31, x31, 255

sw x31, 20(x30)

lw x1, 0(x30)

lw x2, 4(x30)

lw x3, 8(x30)

lw x4, 20(x30)

sw x1, 0(x30)

addi x5, x5, 255

High-level test case 1 (tb_RC5):

Testbench **tb_RC5** is used to test the encoding and decoding algorithm RC5 by our processor. The instructions tested in this testbench are translated into hex machine code and stored in file **RC5_final3.mem**.

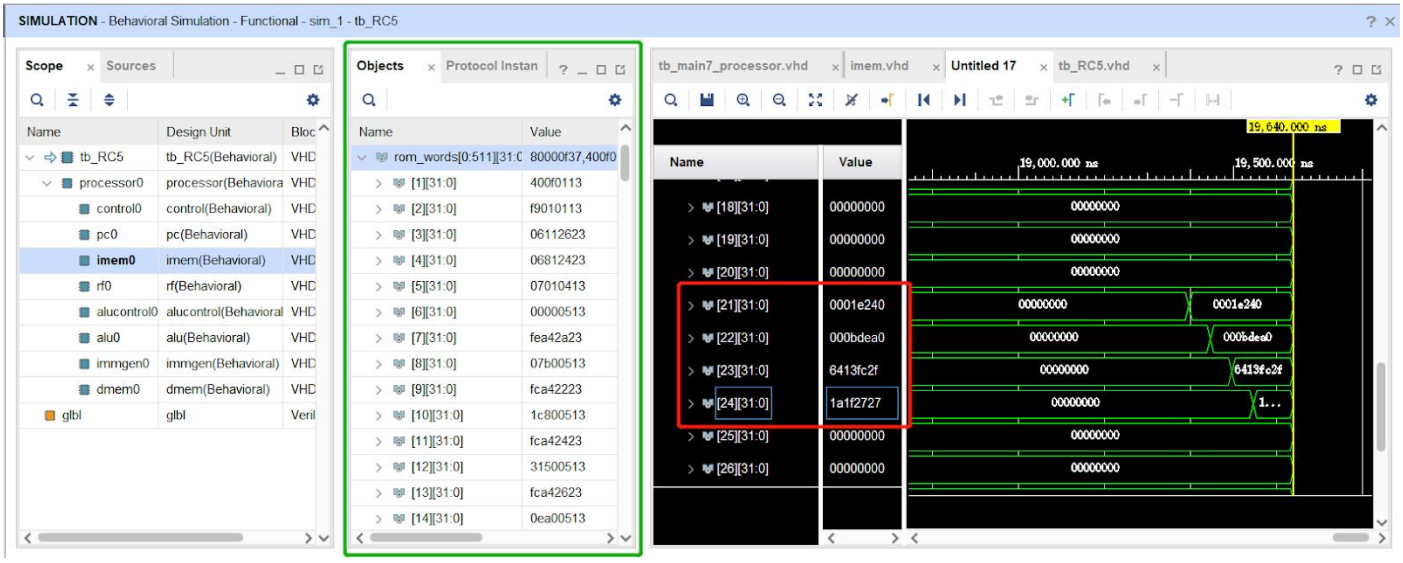
The input plaintexts of RC5 are two numbers: 123456, 777888 (1e240, bdea0 in hex)

The ciphertexts after encoding are: 1679031343, 438249255 (6413fc2f, 1a1f2727 in hex)

After running the instructions, the data stored in registers x21 to x24 (by instructions) will be output as a signal "**reg_out[0:31][0:31]** (named in RegFile.vhd, in processor.vhd is "**regfile[0:31][0:31]**")" and compared by a set of desired output.

Ciphertext results are stored in registers x23, x24; plaintext results gotten from decoding are stored in x21, x22 by instructions.

Simulation result:



In the “Scope” and “Object” window, it showed that the instructions (read from “RC5_final3.mem”) are stored in “rom_words” in the instruction memory.
In the “Untitled 19 window”, it showed the result stored in registers x21 to x24. 123456, 777888 (1e240, bdea0 in hex) are in x21 and x22, 1679031343, 438249255 (6413fc2f, 1a1f2727 in hex) are in x23 and x24.

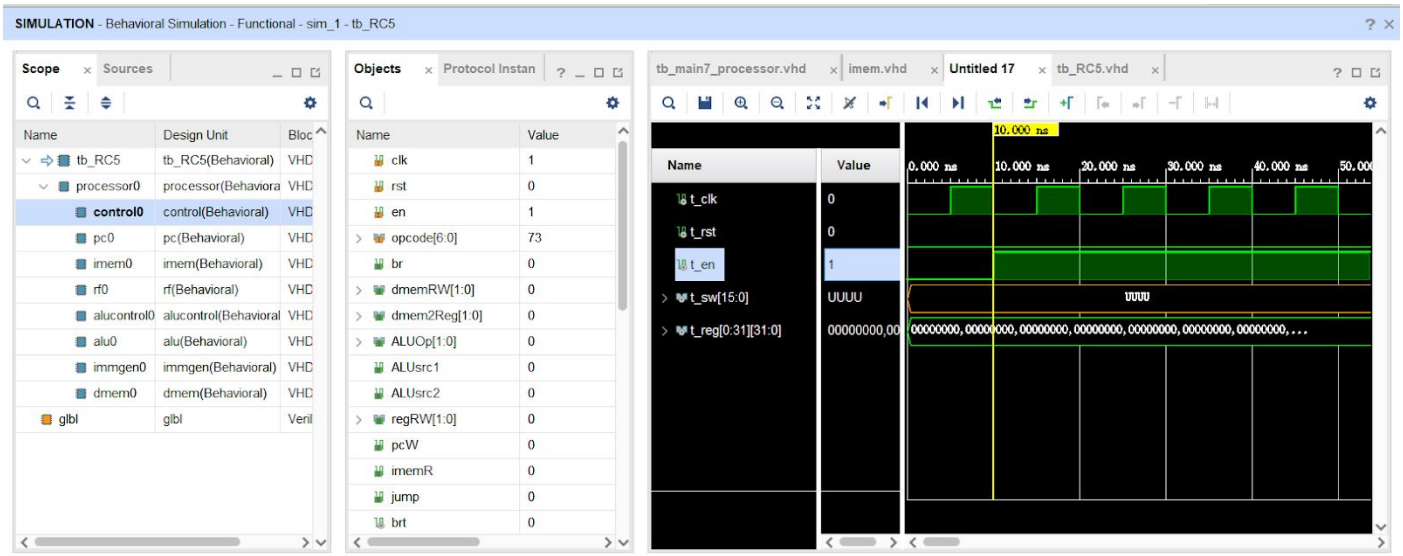
RISC-V instructions in **RC5_final3.mem**: (before translated into hex machine code, same as “rom_words” above)
addi x2 x2 -112
sw x1 108(x2)
sw x8 104(x2)
addi x8 x2 112
addi x10 x0 0
sw x10 -12(x8)
addi x10 x0 123
sw x10 -60(x8)
addi x10 x0 456
sw x10 -56(x8)
addi x10 x0 789
sw x10 -52(x8)
addi x10 x0 234
sw x10 -48(x8)
lui x10 2
addi x11 x10 1684
sw x11 -44(x8)
addi x11 x10 -415
sw x11 -40(x8)
lui x11 1
addi x12 x11 -763
sw x12 -36(x8)
addi x12 x10 1795
sw x12 -32(x8)
addi x12 x0 1367

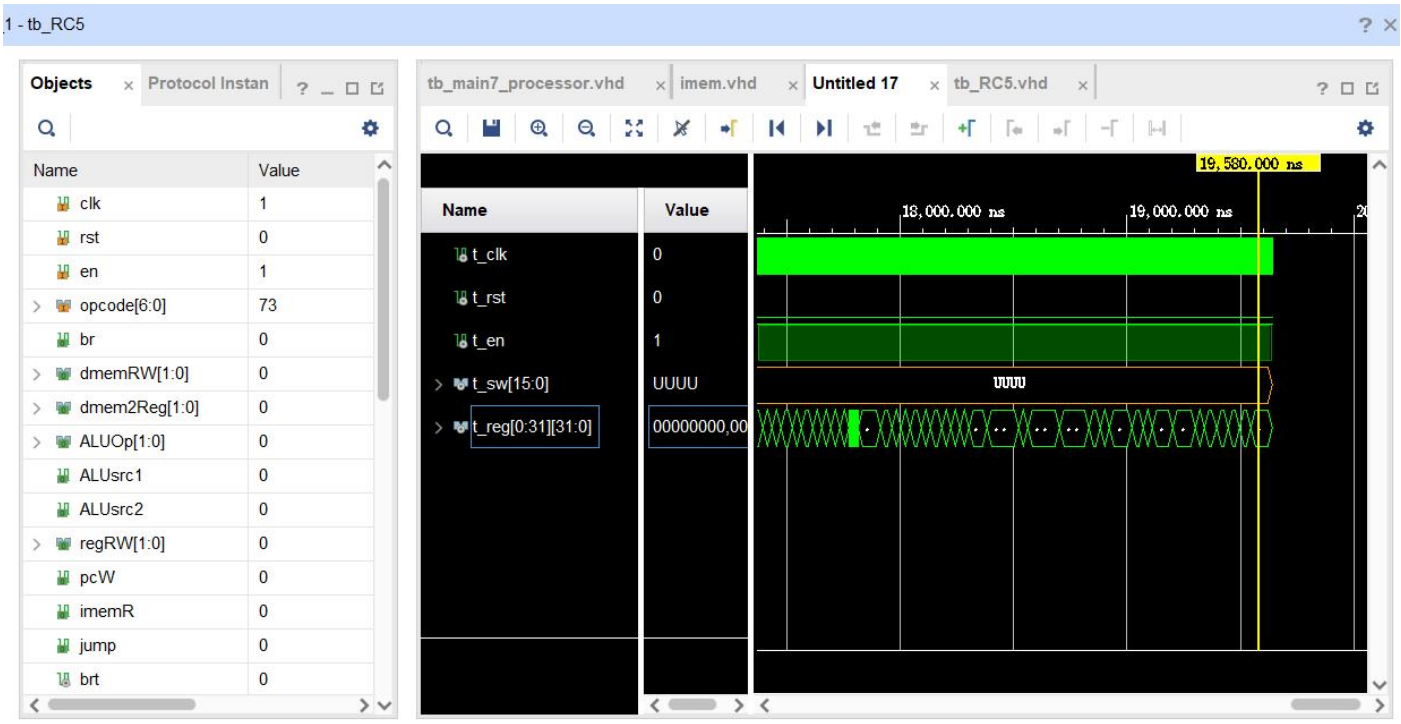
```
sw x12 -28(x8)
addi x11 x11 -311
sw x11 -24(x8)
addi x10 x10 1551
sw x10 -20(x8)
addi x10 x0 432
sw x10 -16(x8)
lui x10 30
addi x10 x10 576
sw x10 -68(x8)
lui x10 190
addi x10 x10 -352
sw x10 -64(x8)
lw x10 -68(x8)
lw x11 -60(x8)
add x10 x10 x11
sw x10 -80(x8)
lw x10 -64(x8)
lw x11 -56(x8)
add x10 x10 x11
sw x10 -84(x8)
addi x10 x0 1
sw x10 -88(x8)
jal x0 4
lw x11 -88(x8)
addi x10 x0 5
blt x10 x11 136
jal x0 4
lw x10 -80(x8)
lw x12 -84(x8)
xor x11 x10 x12
sll x10 x11 x12
sub x12 x0 x12
srl x11 x11 x12
or x10 x10 x11
lw x11 -88(x8)
slli x11 x11 3
addi x12 x8 -60
add x11 x11 x12
lw x11 0(x11)
add x10 x10 x11
sw x10 -80(x8)
lw x10 -84(x8)
lw x13 -80(x8)
xor x11 x10 x13
sll x10 x11 x13
sub x13 x0 x13
srl x11 x11 x13
or x10 x10 x11
```

```
lw x11 -88(x8)
slli x11 x11 3
add x11 x11 x12
lw x11 4(x11)
add x10 x10 x11
sw x10 -84(x8)
jal x0 4
lw x10 -88(x8)
addi x10 x10 1
sw x10 -88(x8)
jal x0 -140
lw x10 -80(x8)
sw x10 -76(x8)
lw x10 -84(x8)
sw x10 -72(x8)
lw x10 -72(x8)
sw x10 -92(x8)
lw x10 -76(x8)
sw x10 -96(x8)
addi x10 x0 5
sw x10 -100(x8)
jal x0 4
lw x11 -100(x8)
addi x10 x0 0
bge x10 x11 136
jal x0 4
lw x10 -92(x8)
lw x11 -100(x8)
slli x12 x11 3
addi x11 x8 -60
add x12 x12 x11
lw x12 4(x12)
sub x13 x10 x12
lw x12 -96(x8)
srl x10 x13 x12
sub x14 x0 x12
sll x13 x13 x14
or x10 x10 x13
xor x10 x10 x12
sw x10 -92(x8)
lw x10 -96(x8)
lw x12 -100(x8)
slli x12 x12 3
add x11 x11 x12
lw x11 0(x11)
sub x12 x10 x11
lw x11 -92(x8)
srl x10 x12 x11
sub x13 x0 x11
```

sll x12 x12 x13
or x10 x10 x12
xor x10 x10 x11
sw x10 -96(x8)
jal x0 4
lw x10 -100(x8)
addi x10 x10 -1
sw x10 -100(x8)
jal x0 -140
lw x10 -92(x8)
lw x11 -56(x8)
sub x10 x10 x11
sw x10 -64(x8)
lw x10 -96(x8)
lw x11 -60(x8)
sub x10 x10 x11
sw x10 -68(x8)
addi x10 x0 0
lw x1 108(x2)
lw x8 104(x2)
addi x2 x2 112
lw x21, 956(x30)
lw x22, 960(x30)
lw x23, 948(x30)
lw x24, 952(x30)
jalr x0 x1 0

Run time:





The total time for the running of all 427 instructions (after counting all of loops) is 19570ns (10ns to 19580ns, other time are used to compare the test result stored in registers with the desired result), which is 1957 clock cycles.
It means that 4.58 clock cycles on average is needed for each instruction. The reason is that the load/store instructions take 5 clock cycles (for 5 state of the Control component), and other instructions take 4 clock cycles.

High-level test case 2 (tb_bubble2):

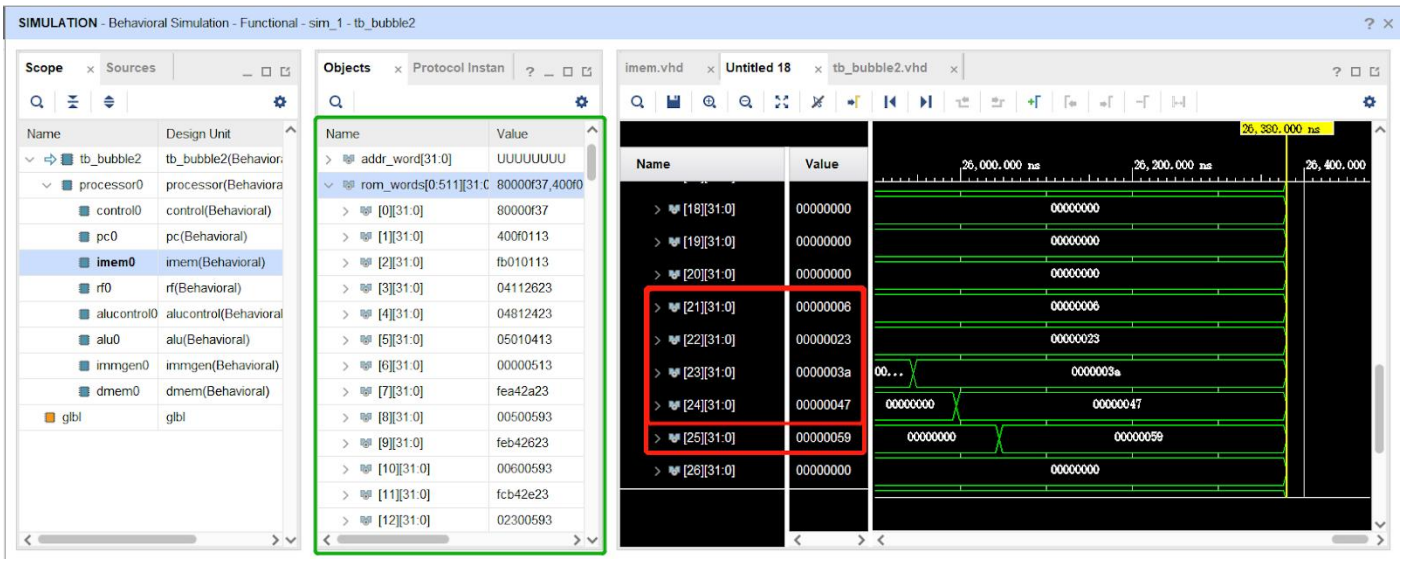
Testbench **tb_ bubble2** is used to test the bubble sort algorithm by our processor. The instructions tested in this testbench are translated into hex machine code and stored in file **bubble2_new2.mem**.

The input of bubble sort is 5 numbers not in order: 58,89,71,35,6

The output of bubble sort is 5 numbers in order, from small to large (stored in x21 to x25): 6, 35, 58, 71, 89 (6, 23, 3a, 47, 59 in hex)

After running the instructions, the data stored in registers x21 to x25 (by instructions) will be output as a signal "**reg_out[0:31][0:31]** (named in RegFile.vhd, in processor.vhd is "**regfile[0:31][0:31]**")" and compared by a set of desired output.

Simulation result:



In the "Scope" and "Object" window, it showed that the instructions (read from "RC5_final3.mem") are stored in

“rom_words” in the instruction memory.
In the “Untitled 19 window”, it showed the result stored in registers x21 to x25. 6, 35, 58, 71, 89 (6, 23, 3a, 47, 59 in hex) are in x21 to x25, respectively.

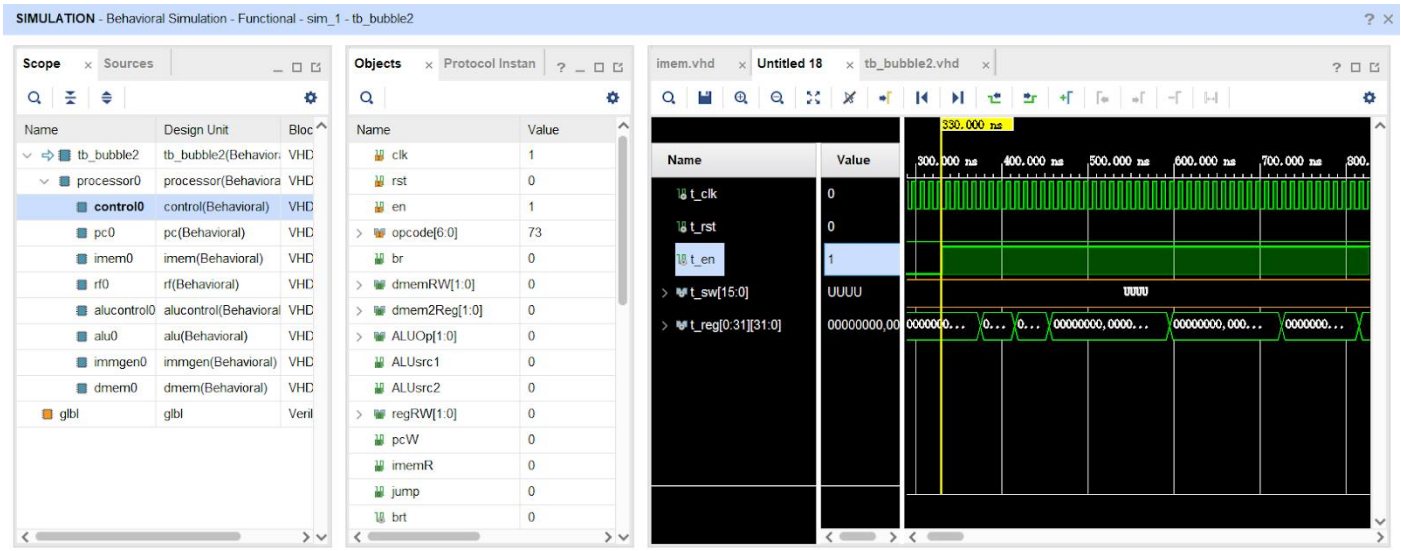
RISC-V instructions in **bubble2_new2.mem**: (before translated into hex machine code, same as “rom_words” above)

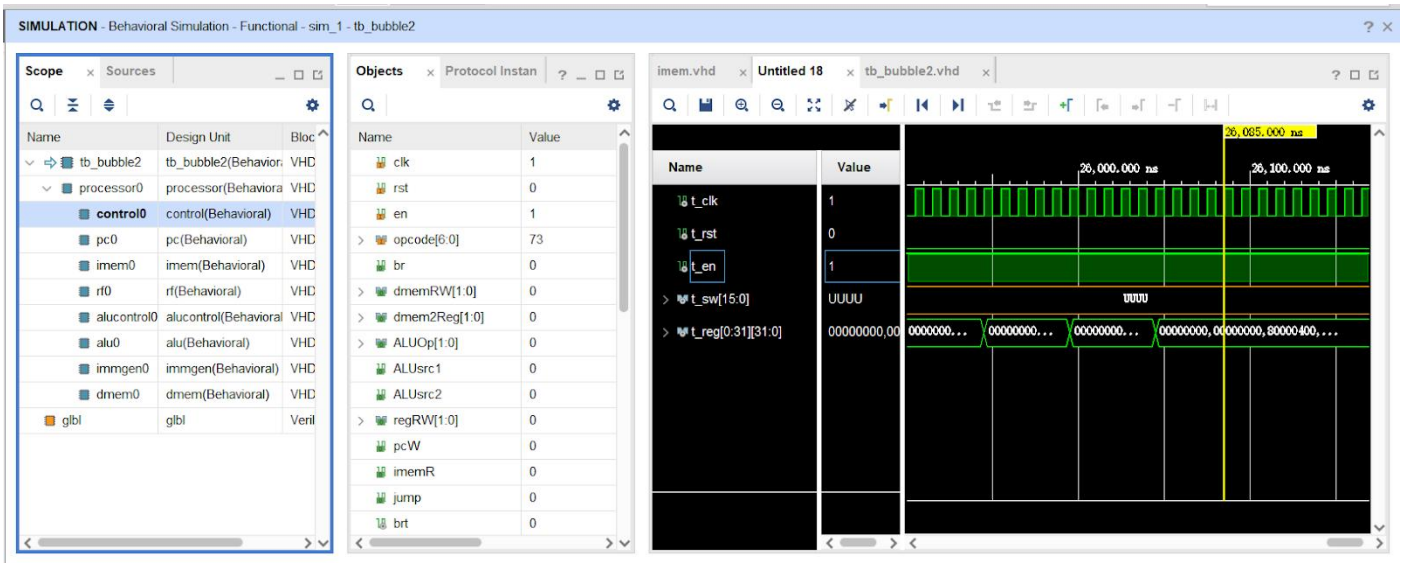
```
addi x2 x2 -80
sw x1 76(x2)
sw x8 72(x2)
addi x8 x2 80
addi x10 x0 0
sw x10 -12(x8)
addi x11 x0 5
sw x11 -20(x8)
addi x11 x0 6
sw x11 -36(x8)
addi x11 x0 35
sw x11 -40(x8)
addi x11 x0 71
sw x11 -44(x8)
addi x11 x0 89
sw x11 -48(x8)
addi x11 x0 58
sw x11 -52(x8)
sw x10 -56(x8)
sw x10 -60(x8)
sw x10 -64(x8)
sw x10 -68(x8)
sw x10 -72(x8)
sw x10 -32(x8)
jal x0 4
lw x10 -32(x8)
lw x11 -20(x8)
addi x11 x11 -1
bge x10 x11 224
jal x0 4
addi x10 x0 0
sb x10 -13(x8)
sw x10 -24(x8)
jal x0 4
lw x10 -24(x8)
lw x12 -20(x8)
lw x11 -32(x8)
xori x11 x11 -1
add x11 x11 x12
bge x10 x11 136
jal x0 4
lw x10 -24(x8)
slli x11 x10 2
addi x10 x8 -52
```

```
add x10 x10 x11
lw x11 0(x10)
lw x10 4(x10)
bge x10 x11 84
jal x0 4
lw x10 -24(x8)
slli x10 x10 2
addi x12 x8 -52
add x10 x10 x12
lw x10 0(x10)
sw x10 -28(x8)
lw x10 -24(x8)
slli x10 x10 2
add x11 x12 x10
lw x10 4(x11)
sw x10 0(x11)
lw x10 -28(x8)
lw x11 -24(x8)
slli x11 x11 2
add x11 x11 x12
sw x10 4(x11)
addi x10 x0 1
sb x10 -13(x8)
jal x0 4
jal x0 4
lw x10 -24(x8)
addi x10 x10 1
sw x10 -24(x8)
jal x0 -152
lbu x10 -13(x8)
andi x10 x10 1
addi x11 x0 0
bne x10 x11 12
jal x0 4
jal x0 24
jal x0 4
lw x10 -32(x8)
addi x10 x10 1
sw x10 -32(x8)
jal x0 -232
addi x10 x0 0
sw x10 -32(x8)
jal x0 4
lw x10 -32(x8)
lw x11 -20(x8)
bge x10 x11 60
jal x0 4
lw x10 -32(x8)
slli x12 x10 2
```

addi x10 x8 -52
add x10 x10 x12
lw x10 0(x10)
addi x11 x8 -72
add x11 x11 x12
sw x10 0(x11)
jal x0 4
lw x10 -32(x8)
addi x10 x10 1
sw x10 -32(x8)
jal x0 -64
lw x10 -12(x8)
lw x1 76(x2)
lw x8 72(x2)
addi x2 x2 80
lw x21, 972(x30)
lw x22, 976(x30)
lw x23, 980(x30)
lw x24, 984(x30)
lw x25, 988(x30)
jalr x0 x1 0

Run time:





The total time for the running of all 571 instructions (after counting all of loops) is 25750ns (330ns to 26080ns, other time are used to compare the test result stored in registers with the desired result), which is 2575 clock cycles.
It means that 4.51 clock cycles on average is needed for each instruction. The reason is that the load/store instructions take 5 clock cycles (for 5 state of the Control component), and other instructions take 4 clock cycles.

Possible Optimization and Improvement:

1. Use Pipelined Design

By executing different operation steps of multiple instructions simultaneously, the overall speed of instruction flow is accelerated, and program execution time is reduced. his reduces the number of clock cycles required for the processor to execute instructions and increases the issue-rate of instructions. Take the Adders or Multipliers as an example. Work faster by adding more loops, which can be relatively reduced if replaced by pipelining.

2. Add MAF（multiply-add-fused）

This unit implements multiplication and addition instructions for floating-point data, and can support multiplication, addition, subtraction, multiplication, and subtraction instructions for floating-point, as well as some fixed-point instructions, and vector and scalar instructions for fixed-point and floating-point. Without the MAF, the operation of adding the product of multiplication and the value of accumulator A to the accumulator may require two instructions, but the MAF can realize it in one instruction. Many operations (such as convolution, dot product, matrix, digital filter, and even polynomial evaluation) can be decomposed into several MAF thus increasing the efficiency of these operations.

3. Future Work

- Instruction Memory

  The instruction memory will be implemented to have the ability select input program. The user can enter different inputs to select different running program.
- Control Unit

  The output of the control unit will be more comprehensive. In more detail, the current control unit cannot control all other components individually. In the future work, all other components will be controlled by only the control unit.

Tools and References:

- C to RICS V
  https://godbolt.org/
- RISC V to machine code
  https://venus.kvakil.me/
- online simulation RISC
  https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/
- C codes are modified from ChatGPT