台科大

PA2

别扣分了...

周 柏宇

2024/4/26

内容

Part1 5
R_FormatCPU 5
Description 6
Code Explanation 6
IO 7
Wires 7
Components 7
Process 9
Adder 9
Description 9
Code Explanation 9
IO 10
Wires 10
Components 10
Process 10
IM 11
Description 11
Code Explanation 11
IO 12
Wires 12
Components 12
Process 12
RF 13
Description 13
Code Explanation 13
IO 14
Wires 14
Components 14
Process 15
Control 15
Description 15
Code Explanation 15
IO 16
Wires 16
Components 16
Process 16
ALU_Control 17

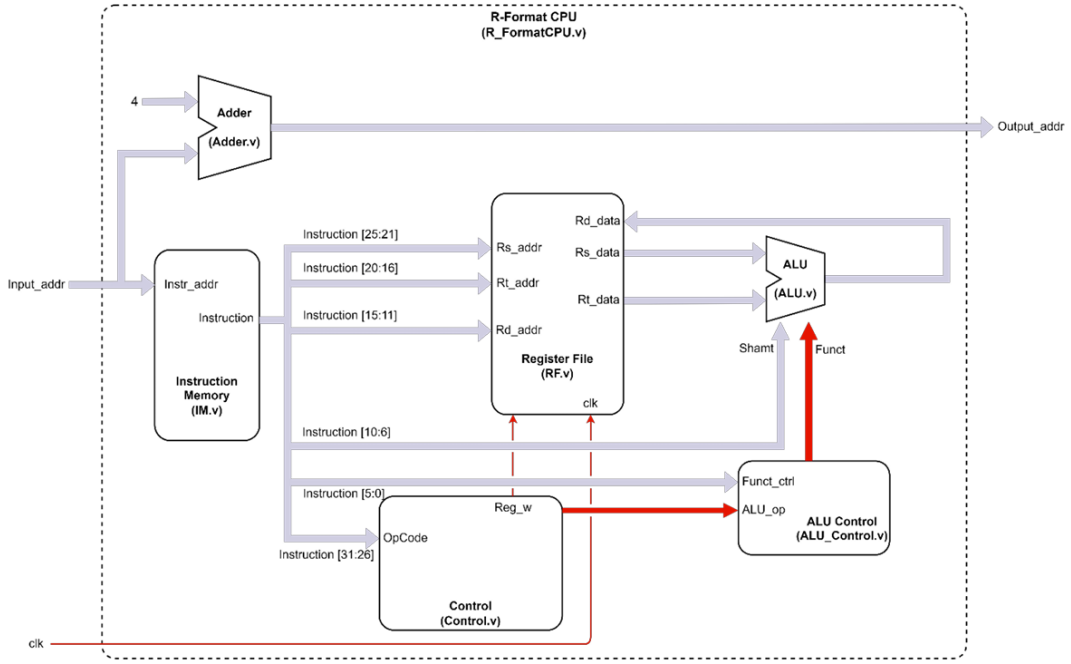
Description 17
Code Explanation 17
IO 18
Wires 18
Components 18
Process 18
ALU 19
Description 19
Code Explanation 19
IO 20
Wires 20
Components 20
Process 21
Part2 22
I_FormatCPU 22
Description 23
Code Explanation 24
IO 24
Wires 24
Components 25
Process 25
Adder 26
IM 26
RegMUX 26
Description 26
Code Explanation 27
IO 27
Wires 27
Components 27
Process 28
RF 28
Control 28
Description 29
Code Explanation 29
IO 30
Wires 30
Components 30
Process 31

GPRMUX....31
Description....31
Code Explanation....31
IO....32
Wires....32
Components....32
Process....32
ALU_Control....33
Description....33
Code Explanation....33
IO....34
Wires....34
Components....34
Process....34
ALU....35
Description....35
Code Explanation....35
IO....36
Wires....36
Components....36
Process....37
DM....37
Description....37
Code Explanation....38
IO....39
Wires....39
Components....39
Process....39
Part3....40
SimpleCPU....40
Description....42
Code Explanation....42
IO....43
Wires....43
Components....44
Process....45
Adder....45
GPRMUX....45

IM 45
RegMUX 46
RF 46
Control 46
Description 48
Code Explanation 48
IO 48
Wires 49
Components 49
Process 49
ALU_Control 49
ALU 50
Description 50
Code Explanation 50
IO 51
Wires 52
Components 52
Process 52
DM 52
Execution Results 53
Helper programs 53
Translator.py 53
Convertor.py 55
Part1 58
Part2 69
Part3 80
Custom Test Program 87
Part1 87
Part2 88
Part3 89
Conclusion and Insights 99

Part1

R_FormatCPU




```
29  module R_FormatCPU(
30      // Outputs
31      output  wire    [31:0]  Output_Addr,
32      // Inputs
33      input   wire    [31:0]  Input_Addr,
34      input   wire            clk
35  );
36      wire [31:0] Instruction;
37      wire [31:0] Rs_data;
38      wire [31:0] Rt_data;
39      wire [31:0] Rd_data;
40      wire RegWrite;
41      wire [1:0] ALU_op;
42      wire [5:0] funct;
43
44      Adder AdderInstance(
45          // Outputs
46          .OutputAddr(Output_Addr),
47          // Inputs
48          .InputAddr(Input_Addr),
49          .Offset(32'h4)
50      );
51
52      IM Instr_Memory(
53          // Outputs
54          .Instruction(Instruction),
55          // Inputs
56          .InputAddr(Input_Addr)
57      );
58
```

```
59      RF Register_File(
60          // Outputs
61          .RsData(Rs_data),
62          .RtData(Rt_data),
63          // Inputs
64          .RsAddr(Instruction[25:21]),
65          .RtAddr(Instruction[20:16]),
66          .RdAddr(Instruction[15:11]),
67          .RdData(Rd_data),
68          .RegWrite(RegWrite),
69          .clk(clk)
70      );
71
72      Control ControlInstance(
73          // Outputs
74          .RegWrite(RegWrite),
75          .ALU_op(ALU_op),
76          // Inputs
77          .opcode(Instruction[31:26])
78      );
79
80      ALU_Control ALU_ControlInstance(
81          // Outputs
82          .funct(funct),
83          // Inputs
84          .ALU_op(ALU_op),
85          .funct_ctrl(Instruction[5:0])
86      );
87
88      ALU ALUInstance(
89          // Outputs
90          .ALU_result(Rd_data),
91          // Inputs
92          .Rs_data(Rs_data),
93          .Rt_data(Rt_data),
94          .shamt(Instruction[10:6]),
95          .funct(funct)
96      );
```

Description

This module forms the core of a simple MIPS processor, capable of executing R-type instructions. It comprises components for instruction fetching, decoding, register operations, ALU operations, and memory address calculation, all controlled by synchronous clock signals.

Code Explanation

Line 29~35

This declares a Verilog module named R_FormatCPU.

It defines three ports:

1. Output_Addr: Output port for the calculated memory address.
2. Input_Addr: Input port for the memory address or instruction fetch.
3. clk: Input port for the clock signal.

Line 36~42

These lines declare internal wires to hold various signals and data within the module.

They represent components like registers, ALU control signals, and instruction

data.

Line 44~96

These are functional modules which describe how this module work.

IO

Inputs:

Wire Name	Description
Input_Addr (32-bit)	Address input for fetching instructions and accessing memory.
clk (1-bit)	Clock input for synchronous operation.

Outputs:

Wire name	Description
Output_Addr (32-bit)	Output address for accessing memory or performing other operations.

Wires

Wire Name	Description
Instruction (32-bit)	Holds the current instruction fetched from memory.
Rs_data, Rt_data, Rd_data (32-bit each)	Data read from registers, corresponding to the source and destination operands of instructions.
RegWrite (1-bit)	Control signal indicating whether to write data into registers.
ALU_op (2-bit)	Control signal determining the operation to be performed by the Arithmetic Logic Unit (ALU).
funct (6-bit)	Function code extracted from the instruction, used for ALU operations or control purposes

Components

Component Name	Description
AdderInstance	An instance of an adder component, used for calculating memory addresses.

	It adds the input address (Input_Addr) with an offset (32'h4) to generate the output address (Output_Addr).
Instr_Memory	Represents an Instruction Memory component. It outputs the instruction (Instruction) stored at the memory address provided by Input_Addr.
Register_File	This component simulates a register file, responsible for reading data from and writing data into registers. It reads data from two source registers (Rs_data, Rt_data) based on the register addresses extracted from the instruction (Instruction[25:21], Instruction[20:16]). It also allow to store data in the destination register (Rd_data) using RdData(Instruction[15:11]) and a control signal (RegWrite) to enable register write operations.
ControlInstance	Implements control logic based on the opcode of the instruction (Instruction[31:26]). It generates control signals like RegWrite and ALU_op to coordinate operations within the processor.
ALU_ControlInstance	Determines ALU operation based on the ALU control signals (ALU_op) and the function code (funct_ctrl) extracted from the instruction.
ALUInstance	Represents the Arithmetic Logic Unit (ALU), performing arithmetic and logical operations. It takes two input data (Rs_data, Rt_data), a shamt value (Instruction[10:6]), and a function code (funct) to produce the result (ALU_result).

Process

- The processor fetches an instruction from memory using the Input_Addr.
- The instruction is decoded, and control signals are generated based on the opcode and function code.
- Data is read from registers based on the instruction's source register addresses.
- The ALU operation is determined based on control signals and executed.
- Resultant data is written back to the destination register if RegWrite is asserted.
- The output address is calculated for the next memory access or operation.

Adder

```
1  module Adder(
2      // Outputs
3      output [31:0] OutputAddr,
4      // Inputs
5      input [31:0] InputAddr,
6      input [31:0] Offset
7  );
8
9      assign OutputAddr = InputAddr + Offset;
10
11 endmodule
```

Description

The Adder module performs a basic arithmetic addition operation by adding an input address (InputAddr) with an offset (Offset) to generate an output address (OutputAddr).

Code Explanation

Line 1~7

Module Declaration:

Declares a Verilog module named Adder.

Defines three ports:

1. OutputAddr: Output port for the calculated address.
2. InputAddr: Input port for the base address.
3. Offset: Input port for the offset to be added to the base address.

Line 9

Logic:

Uses an assign statement to perform the addition operation.

OutputAddr is assigned the result of adding InputAddr and Offset.

IO

Inputs:

Wire Name	Description
InputAddr (32-bit)	Base address for addition.
Offset (32-bit)	Offset value to be added to the base address.

Outputs:

Wire Name	Description
OutputAddr (32-bit)	Resultant address obtained by adding the input address and offset.

Wires

No internal wires are explicitly declared in this module. The addition operation is directly performed using the assign statement.

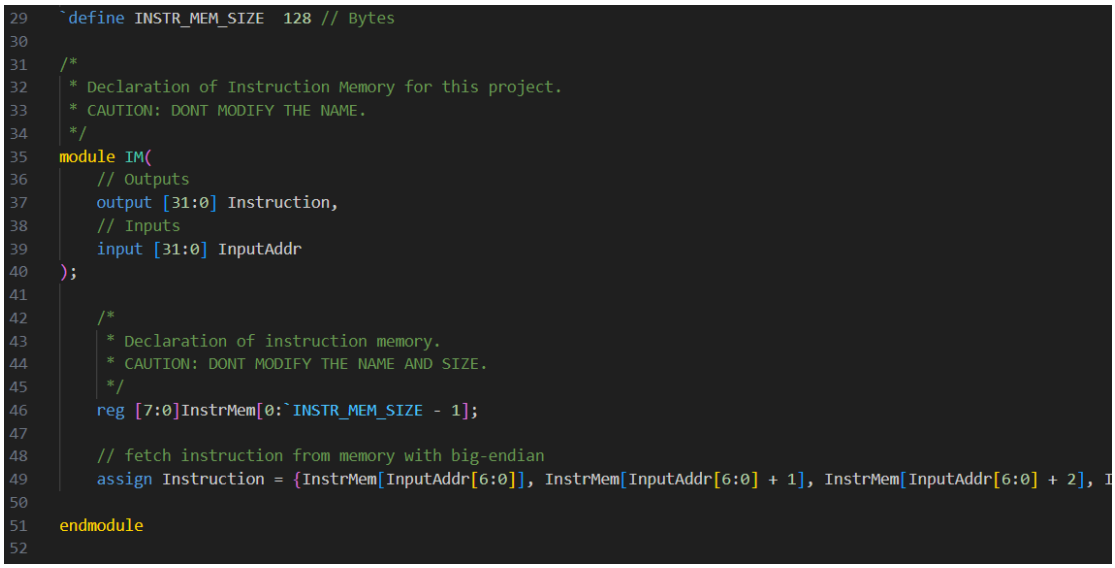
Components

The Adder module itself acts as a basic arithmetic component performing addition.

Process

- Input Acquisition: Receives the base address (InputAddr) and offset value (Offset) as inputs.
- Addition Operation: Adds the input address and offset to calculate the output address.
- Output Generation: Provides the calculated output address (OutputAddr) as the module's output.

IM



```
29  `define INSTR_MEM_SIZE  128 // Bytes
30
31  /*
32   * Declaration of Instruction Memory for this project.
33   * CAUTION: DONT MODIFY THE NAME.
34   */
35  module IM(
36      // Outputs
37      output [31:0] Instruction,
38      // Inputs
39      input [31:0] InputAddr
40  );
41
42      /*
43       * Declaration of instruction memory.
44       * CAUTION: DONT MODIFY THE NAME AND SIZE.
45       */
46      reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
47
48      // fetch instruction from memory with big-endian
49      assign Instruction = {InstrMem[InputAddr[6:0]], InstrMem[InputAddr[6:0] + 1], InstrMem[InputAddr[6:0] + 2], I
50
51  endmodule
52
```

Description

The IM module simulates an instruction memory unit that stores instructions and fetches them based on the input address using big-endian.

Code Explanation

Line 29

Define constant for later usage.

Line 35~40

Module Declaration:

Defines a Verilog module named IM.

Specifies two ports:

- Instruction: Output port for the fetched instruction.
- InputAddr: Input port for the address used to fetch the instruction.

Line 46

Memory Declaration:

Declares an array InstrMem to represent the instruction memory.

Each element of InstrMem is 8 bits wide (reg [7:0]).

The size of InstrMem is defined by the parameter INSTR_MEM_SIZE, which is set to 128 bytes.

Line 49

Instruction Fetch:

Utilizes big-endian byte addressing.

Concatenates four bytes fetched from InstrMem based on the input address InputAddr.

Forms a 32-bit instruction stored in Instruction.

IO

Inputs:

Wire Name	Description
InputAddr (32-bit)	Address input for fetching instructions from memory.

Outputs:

Wire Name	Description
Instruction (32-bit)	Fetched instruction output from memory.

Wires

No internal wires are explicitly declared in this module. The instruction is directly assigned using the assign statement.

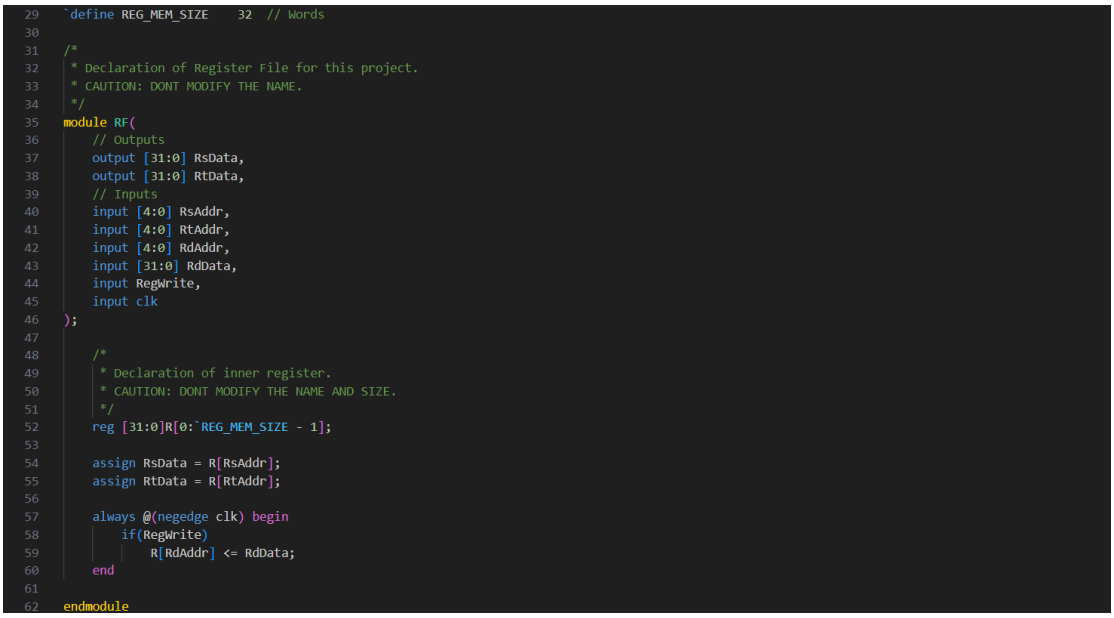
Components

The IM module itself acts as a memory component to store and fetch instructions.

Process

- Memory Initialization: The InstrMem array is initialized to store instructions.
- Instruction Fetch: Based on the input address, four bytes are fetched from memory and concatenated to form a 32-bit instruction.
- Output Generation: The fetched instruction is provided as the module's output.

RF



```
29  `define REG_MEM_SIZE    32  // Words
30
31  /*
32   * Declaration of Register File for this project.
33   * CAUTION: DONT MODIFY THE NAME.
34   */
35  module RF(
36      // Outputs
37      output [31:0] RsData,
38      output [31:0] RtData,
39      // Inputs
40      input [4:0] RsAddr,
41      input [4:0] RtAddr,
42      input [4:0] RdAddr,
43      input [31:0] RdData,
44      input RegWrite,
45      input clk
46  );
47
48      /*
49       * Declaration of inner register.
50       * CAUTION: DONT MODIFY THE NAME AND SIZE.
51       */
52      reg [31:0]R[0:`REG_MEM_SIZE - 1];
53
54      assign RsData = R[RsAddr];
55      assign RtData = R[RtAddr];
56
57      always @(negedge clk) begin
58          if(RegWrite)
59              R[RdAddr] <= RdData;
60      end
61
62  endmodule
```

Description

The RF module emulates a register file that provides read and write functionality for registers. It has multiple read ports (RsData and RtData) and a write port (RdData) along with control signals (RegWrite and clk) for writing data into registers.

Code Explanation

Line 29

Define constant for later usage.

Line 35~46

Module Declaration:

Defines a Verilog module named RF.

Specifies two output ports for data read from registers: RsData and RtData.

Specifies input ports for register addresses (RsAddr, RtAddr), write data (RdData), write address (RdAddr), write enable signal (RegWrite), and clock signal (clk).

Line 52

Register Declaration:

Declares an array R to represent the register file.

Each element of R is 32 bits wide (reg [31:0]).

The size of R is defined by the parameter REG_MEM_SIZE, which is set to 32 words.

Line 54~55

Register Read:

Assigns the data from the specified registers (RsAddr and RtAddr) to the output ports RsData and RtData, respectively.

Line 57~60

Register Write:

Utilizes an always block triggered on the negative edge(avoid racing) of the clock signal (clk).

If the RegWrite signal is asserted, the data RdData is written into the register specified by the address RdAddr.

IO

Inputs:

Wire Name	Description
RsAddr, RtAddr, RdAddr (5-bit each)	Addresses for reading from or writing to registers.
RdData (32-bit)	Data to be written into the register specified by RdAddr.
RegWrite (1-bit)	Write enable signal. It controls register writing.
clk (1-bit)	Clock signal for synchronous operation.

Outputs:

Wire Name	Description
RsData, RtData (32-bit each)	Data read from the registers specified by RsAddr and RtAddr, respectively.

Wires

No internal wires are explicitly declared in this module. Register data is directly assigned to output ports using the assign statement.

Components

The RF module itself acts as a register file component, providing read and write

functionality for registers.

Process

- Read Operation: Data from specified registers (RsAddr and RtAddr) is assigned to output ports RsData and RtData, respectively.
- Write Operation: On the negative edge of the clock signal, if the RegWrite signal is asserted, data RdData is written into the register specified by RdAddr.

Control

```
1  module Control(
2      // Outputs
3      output RegWrite,
4      output [1:0] ALU_op,
5      // Inputs
6      input [5:0] opcode
7  );
8
9      assign RegWrite = (opcode == 6'b000000) ? 1'b1 : 1'b0;
10     assign ALU_op = (opcode == 6'b000000) ? 2'b10 : 2'b00;
11
12 endmodule
```

Description

The Control module is responsible for generating control signals based on the opcode of the instruction. It determines whether the instruction requires register writes and specifies the operation to be performed by the Arithmetic Logic Unit (ALU)

Code Explanation

Line 1~7

Module Declaration:

Defines a Verilog module named Control.

Specifies two output ports for control signals: RegWrite (1-bit) and ALU_op (2-bit).

Specifies an input port opcode (6-bit) to receive the opcode of the instruction.

Line 9~10

The RegWrite signal is determined by checking if the opcode matches a specific value (6'b000000). If it does, RegWrite is set to 1'b1; otherwise, it is set to 1'b0.

The ALU_op signal is determined similarly. If the opcode matches 6'b000000,

indicating an R-type instruction, ALU_op is set to 2'b10 to indicate that the ALU should perform an operation. Otherwise, it is set to 2'b00.

IO

Inputs:

Wire Name	Description
opcode (6-bit)	Input port to receive the opcode of the instruction.

Outputs:

Wire Name	Description
RegWrite (1-bit)	Output port indicating whether register write is enabled.
ALU_op (2-bit)	Output port specifying the ALU operation code.

Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned using the assign statement.

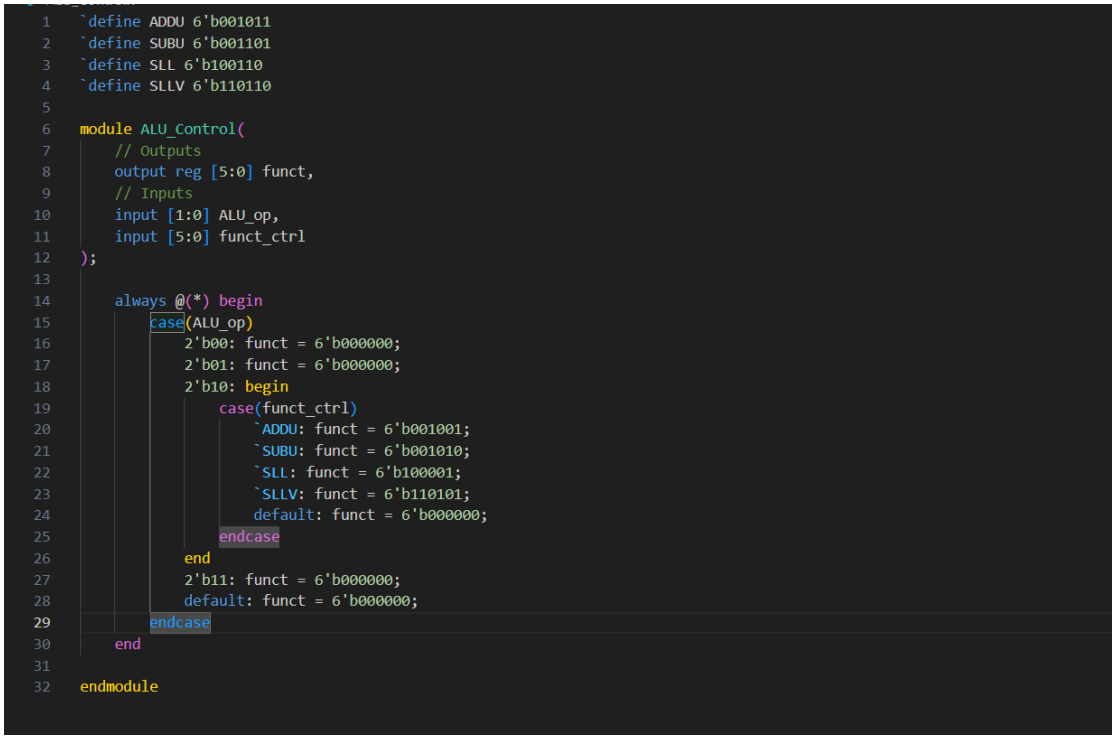
Components

The Control module itself acts as a control unit responsible for generating control signals.

Process

- Opcode Decoding: Receives the opcode of the instruction.
- Control Signal Generation: Based on the opcode, generates control signals such as RegWrite and ALU_op.
- Output: Provides the generated control signals as module outputs.

ALU_Control



```
1  `define ADDU 6'b001011
2  `define SUBU 6'b001101
3  `define SLL 6'b100110
4  `define SLLV 6'b110110
5
6  module ALU_Control(
7      // Outputs
8      output reg [5:0] funct,
9      // Inputs
10     input [1:0] ALU_op,
11     input [5:0] funct_ctrl
12 );
13
14     always @(*) begin
15         case(ALU_op)
16             2'b00: funct = 6'b000000;
17             2'b01: funct = 6'b000000;
18             2'b10: begin
19                 case(funct_ctrl)
20                     `ADDU: funct = 6'b001001;
21                     `SUBU: funct = 6'b001010;
22                     `SLL: funct = 6'b100001;
23                     `SLLV: funct = 6'b110101;
24                     default: funct = 6'b000000;
25                 endcase
26             end
27             2'b11: funct = 6'b000000;
28             default: funct = 6'b000000;
29         endcase
30     end
31
32 endmodule
```

Description

The ALU_Control module decodes the ALU operation code and selects the appropriate function code for the ALU based on both the ALU operation code and an additional control signal (funct_ctrl). The function code determines the specific operation to be performed by the ALU.

Code Explanation

Line 1~4

Function code table.

Line 6~12

Module Declaration:

Defines a Verilog module named ALU_Control.

Specifies an output port funct (6-bit) for the ALU function code.

Specifies input ports ALU_op (2-bit) and funct_ctrl (6-bit) for the ALU operation code and additional control signal, respectively.

Line 14~30

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(*)).

Decodes the ALU operation code ALU_op and selects the appropriate function code based on the provided control signal funct_ctrl.

Generates the function code funct according to the selected operation.

IO

Inputs:

Wire Name	Description
ALU_op (2-bit)	ALU operation code input.
funct_ctrl (6-bit)	Additional control signal used for selecting the ALU function code.

Outputs:

Wire Name	Description
funct (6-bit)	Output port representing the ALU function code.

Wires

No internal wires are explicitly declared in this module. The function code is directly assigned using an always block.

Components

The ALU_Control module itself acts as a control unit responsible for generating the ALU function code.

Process

- ALU Operation Code Decoding: Receives the ALU operation code (ALU_op).
- Function Code Selection: Based on the ALU operation code and the provided control signal (funct_ctrl), selects the appropriate function code for the ALU.
- Output Generation: Provides the generated function code (funct) as the module's output.

ALU

```
1  `define ADDU 6'b001001
2  `define SUBU 6'b001010
3  `define SLL 6'b100001
4  `define SLLV 6'b110101
5
6  module ALU(
7      // Outputs
8      output reg [31:0] ALU_result,
9      // Inputs
10     input [31:0] Rs_data,
11     input [31:0] Rt_data,
12     input [4:0] shamt,
13     input [5:0] funct
14 );
15     wire [31:0] ADDU_result, SUBU_result, SLL_result, SLLV_result;
16
17     assign ADDU_result = Rs_data + Rt_data;
18     assign SUBU_result = Rs_data - Rt_data;
19     assign SLL_result = Rs_data << shamt;
20     assign SLLV_result = Rs_data << Rt_data[4:0];
21
22     always @(*) begin
23         case(funct)
24             `ADDU: ALU_result = ADDU_result;
25             `SUBU: ALU_result = SUBU_result;
26             `SLL: ALU_result = SLL_result;
27             `SLLV: ALU_result = SLLV_result;
28             default: ALU_result = 32'b0;
29         endcase
30     end
31
32 endmodule
```

Description

The ALU module simulates an Arithmetic Logic Unit (ALU) which executes arithmetic and logical operations such as addition, subtraction, left logical shift, and variable left logical shift. It takes two input data operands (Rs_data, Rt_data), a shamt value (shamt), and a function code (funct) to produce the result (ALU_result).

Code Explanation

Line 1~4

Operation code table.

Line 6~14

Module Declaration:

Defines a Verilog module named ALU.

Specifies an output port ALU_result (32-bit) for the result of ALU operations.

Specifies input ports for the two data operands (Rs_data, Rt_data), shamt value (shamt), and function code (funct).

Line 15~20

ALU Operation:

Computes the results of various ALU operations (ADDU, SUBU, SLL, SLLV) and stores them in separate wires (ADDU_result, SUBU_result, SLL_result, SLLV_result).

Line 22~30

Result Selection:

Uses an always block triggered whenever the inputs change (@(*)).
Selects the result based on the function code (funct) using a case statement.
Assigns the selected result to the output port ALU_result.

IO

Inputs:

Wire Name	Description
Rs_data, Rt_data (32-bit each)	Input data operands for the ALU operations.
shamt (5-bit)	Shift amount for left logical shift operation.
funct (6-bit)	Function code specifying the operation to be performed by the ALU.

Outputs:

Wire Name	Description
ALU_result (32-bit)	Result of the ALU operation which is a register because it needs to be assigned as left value in always block.

Wires

ADDU_result, SUBU_result, SLL_result, SLLV_result (32-bit each): Wires to store the intermediate results of ALU operations.

Components

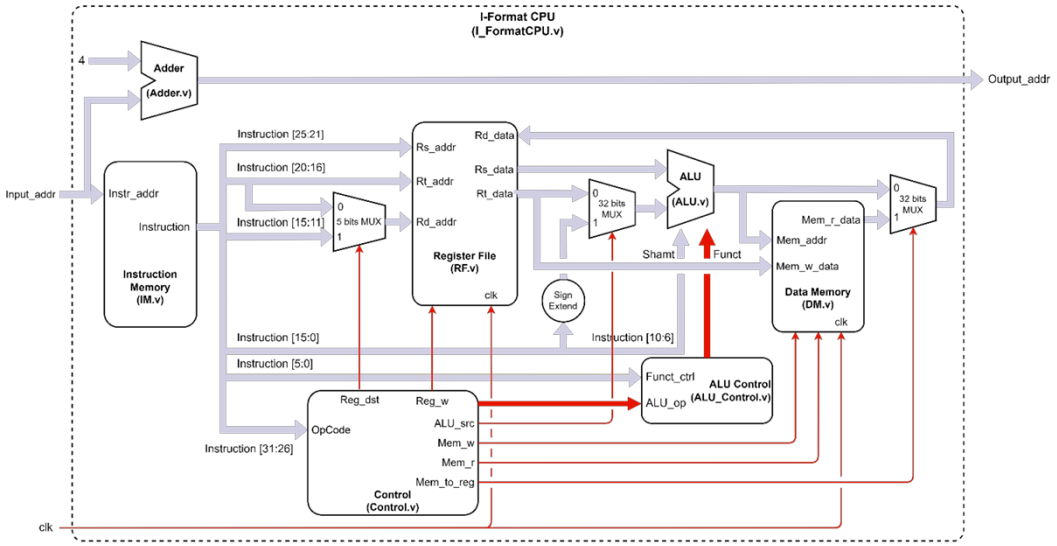
The ALU module itself acts as an Arithmetic Logic Unit (ALU) component capable of performing various arithmetic and logical operations.

Process

- ALU Operation: Computes the results of various ALU operations (ADDU, SUBU, SLL, SLLV).
- Result Selection: Selects the appropriate result based on the function code (funct) and assigns it to the output port ALU_result.

Part2

I_FormatCPU




```
29  module I_FormatCPU(
30      // Outputs
31      output  wire    [31:0]  Output_Addr,
32      // Inputs
33      input   wire    [31:0]  Input_Addr,
34      input   wire            clk
35  );
36
37      wire [31:0] Instruction;
38      wire [31:0] Rs_data, Rt_data;
39      wire [4:0] RdAddr;
40      wire [31:0] Rd_data;
41      wire Reg_dst, Reg_w, ALU_src, Mem_w, Mem_r, Mem_to_reg;
42      wire [31:0] imm;
43      wire [31:0] ALU_src2;
44      wire [5:0] funct;
45      wire [1:0] ALU_op;
46      wire [31:0] ALU_result;
47      wire [31:0] Mem_r_data;
48
49      // sign extend imm
50      assign imm = { {16{Instruction[15]}}, Instruction[15:0] };
51
52      Adder AdderInstance(
53          // Outputs
54          .OutputAddr(Output_Addr),
55          // Inputs
56          .InputAddr(Input_Addr),
57          .Offset(32'h4)
58      );
59
60      IM Instr_Memory(
61          // Outputs
62          .Instruction(Instruction),
63          // Inputs
64          .InputAddr(Input_Addr)
65      );
```

```
76      RF Register_File(
77          // Outputs
78          .RsData(Rs_data),
79          .RtData(Rt_data),
80          // Inputs
81          .RsAddr(Instruction[25:21]),
82          .RtAddr(Instruction[20:16]),
83          .RdAddr(RdAddr),
84          .RdData(Rd_data),
85          .RegWrite(Reg_w),
86          .clk(clk)
87      );
88
89      Control Control_Unit(
90          // Outputs
91          .RegDst(Reg_dst),
92          .RegWrite(Reg_w),
93          .ALU_op(ALU_op),
94          .ALU_src(ALU_src),
95          .Mem_w(Mem_w),
96          .Mem_r(Mem_r),
97          .Mem_to_Reg(Mem_to_reg),
98          // Inputs
99          .opcode(Instruction[31:26])
100     );
101
102     GPRMUX ALU_MUX(
103         // Outputs
104         .data(ALU_src2),
105         // Inputs
106         .src1(Rt_data),
107         .src2(imm),
108         .ctrl(ALU_src)
109     );
110
```

```
111     ALU_Control ALU_Control_Unit(
112         // Outputs
113         .funct(funct),
114         // Inputs
115         .ALU_op(ALU_op),
116         .funct_ctrl(Instruction[5:0])
117     );
118
119     ALU ALU_Instance(
120         // Outputs
121         .ALU_result(ALU_result),
122         // Inputs
123         .data1(Rs_data),
124         .data2(ALU_src2),
125         .shamt(Instruction[10:6]),
126         .funct(funct)
127     );
128
129     DM Data_Memory(
130         // Outputs
131         .Mem_r_data(Mem_r_data),
132         // Inputs
133         .Mem_addr(ALU_result),
134         .Mem_w_data(Rt_data),
135         .Mem_w(Mem_w),
136         .Mem_r(Mem_r),
137         .clk(clk)
138     );
139
140     GPRMUX Mem_MUX(
141         // Outputs
142         .data(Rd_data),
143         // Inputs
144         .src1(ALU_result),
145         .src2(Mem_r_data),
146         .ctrl(Mem_to_reg)
147     );
```

Description

The I_FormatCPU module is a CPU implementation tailored to execute instructions

encoded in the I-format, which typically involve operations with immediate values. It incorporates various components such as instruction memory, register file, ALU, and data memory to execute instructions and perform data operations.

Code Explanation

Line 29~35

Module Declaration:

Defines a Verilog module named I_FormatCPU.

Specifies output port Output_Addr (32-bit) for the computed output address.

Specifies input ports for the input address Input_Addr (32-bit) and clock signal clk.

Line 37~47

Wires used in this scope.

Line 50

Signed extension

Line 52~147

Functional modules

IO

Inputs:

Wire Name	Description
Input_Addr (32-bit)	Input address for fetching instructions.
clk (1-bit)	Clock signal for synchronous operation.

Outputs:

Wire Name	Description
Output_Addr (32-bit)	Computed output address.

Wires

Various wires are used within the module to connect different components and facilitate data flow and control signals. These wires include:

Wire Name	Description
Instruction (32-bit)	Instruction fetched from the instruction memory.
Rs_data, Rt_data (32-bit each)	Data read from the register file.

RdAddr (5-bit)	Register address selected for writing data.
Rd_data	Data to be written into the selected register.
Reg_dst, Reg_w, ALU_src, Mem_w, Mem_r, Mem_to_reg (1-bit each)	Control signals for various components.
Imm (32-bit)	Sign-extended immediate value extracted from the instruction.
ALU_src2 (32-bit)	Second input operand for the ALU.
funct (6-bit), ALU_op (2-bit)	Control signals for ALU operations.
ALU_result (32-bit)	Result produced by the ALU.
Mem_r_data	Data read from the data memory.

Components

The I_FormatCPU module integrates several components to execute instructions and perform data operations:

Component Name	Description
IM	Stores and provides instructions to the CPU.
Adder	Compute addresses for the next instruction and branch targets.
GPRMUX, RegMUX	Select between different data sources based on control signals.
RF	Stores and provides data from registers.
ALU	Performs arithmetic and logical operations.
DM	Stores and retrieves data values.
Control	Generates control signals based on instruction opcode.

Process

- Instruction Fetch: Fetches instructions from the instruction memory based on the input address.
- Control Signal Generation: Generates control signals based on the opcode of the fetched instruction.

- Register Read: Reads data from the register file based on the register addresses specified in the instruction.
- Immediate Extraction: Extracts and sign-extends the immediate value from the instruction.
- ALU Operation Selection: Selects appropriate ALU operations based on control signals and instruction encoding.
- ALU Operation Execution: Performs arithmetic and logical operations using the ALU.
- Data Memory Access: Reads or writes data to the data memory based on control signals.
- Register Write: Writes data into the register file based on control signals and selected register address.
- Output Address Computation: Computes and provides the output address as the module's output.

Adder

It's the same as Part1's.

IM

It's the same as Part1's.

RegMUX

```
1   module RegMUX(
2       // Outputs
3       output [4:0] RegDst,
4       // Inputs
5       input [4:0] Rt,
6       input [4:0] Rd,
7       input RegDst_ctrl
8   );
9
10      assign RegDst = (RegDst_ctrl) ? Rd : Rt;
11
12  endmodule
13
```

Description

The RegMUX module is a multiplexer that selects one of two input register addresses (Rt and Rd) as the output based on the control signal RegDst_ctrl. This module is typically used in digital processor designs to determine the destination register

address for data writes.

Code Explanation

Line 1~8

Module Declaration:

Defines a Verilog module named RegMUX.

Specifies an output port RegDst (5-bit) for the selected register address.

Specifies input ports Rt and Rd (5-bit each) for the two input register addresses, and RegDst_ctrl (1-bit) for the control signal.

Line 10

Multiplexing Operation:

Utilizes an assign statement to select one of the input register addresses (Rt or Rd) based on the value of the control signal RegDst_ctrl.

If RegDst_ctrl is asserted (1), the output RegDst will be equal to Rd; otherwise, it will be equal to Rt.

IO

Inputs:

Wire Name	Description
Rt, Rd (5-bit each)	Input register addresses.
RegDst_ctrl (1-bit)	Control signal for selecting the destination register address.

Outputs:

Wire Name	Description
RegDst (5-bit)	Selected register address.

Wires

No internal wires are explicitly declared in this module. The output RegDst is directly assigned using the assign statement.

Components

The RegMUX module itself acts as a multiplexer component responsible for selecting between two input register addresses based on a control signal.

Process

- Control Signal Interpretation: Receives the control signal RegDst_ctrl.
- Input Selection: Determines the selected output based on the control signal:
  - If RegDst_ctrl is asserted (1), selects the input register address Rd.
  - Otherwise, selects the input register address Rt.
- Output Assignment: Assigns the selected input register address to the output port RegDst.

RF

It's the same as Part1's.

Control

```
`define R_TYPE 6'b000000
`define SUBIU 6'b001101
`define SW 6'b010000
`define LW 6'b010001
`define SLTI 6'b101010

module Control(
    // Outputs
    output reg RegDst,
    output reg RegWrite,
    output reg [1:0] ALU_op,
    output reg ALU_src,
    output reg Mem_w,
    output reg Mem_r,
    output reg Mem_to_Reg,
    // Inputs
    input [5:0] opcode
);

    always @(*) begin
        case(opcode)
            `R_TYPE: begin
                RegDst <= 1'b1;
                RegWrite <= 1'b1;
                ALU_op <= 2'b10;
                ALU_src <= 1'b0;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
            end
            `SUBIU: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b1;
                ALU_op <= 2'b01;
                ALU_src <= 1'b1;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
            end
```

```
40              `SW: begin
41                  RegDst <= 1'b0;
42                  RegWrite <= 1'b0;
43                  ALU_op <= 2'b00;
44                  ALU_src <= 1'b1;
45                  Mem_w <= 1'b1;
46                  Mem_r <= 1'b0;
47                  Mem_to_Reg <= 1'b0;
48              end
49              `LW: begin
50                  RegDst <= 1'b0;
51                  RegWrite <= 1'b1;
52                  ALU_op <= 2'b00;
53                  ALU_src <= 1'b1;
54                  Mem_w <= 1'b0;
55                  Mem_r <= 1'b1;
56                  Mem_to_Reg <= 1'b1;
57              end
58              `SLTI: begin
59                  RegDst <= 1'b0;
60                  RegWrite <= 1'b1;
61                  ALU_op <= 2'b11;
62                  ALU_src <= 1'b1;
63                  Mem_w <= 1'b0;
64                  Mem_r <= 1'b0;
65                  Mem_to_Reg <= 1'b0;
66              end
67              default: begin
68                  RegDst <= 1'b0;
69                  RegWrite <= 1'b0;
70                  ALU_op <= 2'b00;
71                  ALU_src <= 1'b0;
72                  Mem_w <= 1'b0;
73                  Mem_r <= 1'b0;
74                  Mem_to_Reg <= 1'b0;
75              end
76          endcase
77      end
78  endmodule
```

Description

The Control module serves as the control unit of a CPU, responsible for generating control signals based on the opcode of an instruction. It determines how the CPU should process the instruction, including register operations, ALU operations, and memory access.

Code Explanation

Line 1~5

Opcode table.

Line 7~18

Module Declaration:

Defines a Verilog module named Control.

Specifies output ports for various control signals:

- RegDst: Register destination control signal.
- RegWrite: Register write enable control signal.
- ALU_op: ALU operation control signal.

- ALU_src: ALU source control signal.
- Mem_w: Memory write enable control signal.
- Mem_r: Memory read enable control signal.
- Mem_to_Reg: Memory to register control signal.

Specifies an input port opcode (6-bit) for the opcode of the instruction.

Line 20~77

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(*)).

Uses a case statement to select control signals based on the opcode of the instruction.

Generates specific control signals for different types of instructions, including R-type, immediate arithmetic, and memory access instructions. If not recognized, it'll use default condition.

IO

Inputs:

Wire Name	Description
opcode (6-bit)	Opcode of the instruction.

Outputs:

Wire Name	Description
RegDst (1-bit)	Register destination control signal.
RegWrite (1-bit)	Register write enable control signal.
ALU_op (2-bit)	ALU operation control signal.
ALU_src (1-bit)	ALU source control signal.
Mem_w (1-bit)	Memory write enable control signal.
Mem_r (1-bit)	Memory read enable control signal.
Mem_to_Reg (1-bit)	Memory to register control signal.

Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned based on the opcode using the always block.

Components

The Control module acts as a standalone component responsible for generating control signals based on the opcode of an instruction.

Process

- Opcode Interpretation: Receives the opcode of the instruction.
- Control Signal Selection: Determines the appropriate control signals based on the opcode using a case statement.
- Control Signal Assignment: Assigns the selected control signals to the output ports.

GPRMUX

```
1  module GPRMUX(
2      // Outputs
3      output [31:0] data,
4      // Inputs
5      input [31:0] src1,
6      input [31:0] src2,
7      input ctrl
8  );
9
10     assign data = ctrl ? src2 : src1;
11
12 endmodule
```

Description

The GPRMUX module functions as a multiplexer, allowing for the selection of one of two input data sources based on the value of a control signal. This type of multiplexer is commonly used in digital circuit design to facilitate data selection and routing.

Code Explanation

Line 1~8

Module Declaration:

Declares a Verilog module named GPRMUX.

Defines an output port data (32-bit) for the selected data.

Specifies two input ports src1 and src2 (both 32-bit) for the two input data sources.

Specifies an input port ctrl (1-bit) for the control signal.

Line 10

Multiplexing Operation:

Utilizes an assign statement to select one of the input data sources (src1 or src2)

based on the value of the control signal ctrl.
If ctrl is asserted (1), the output data will be equal to src2; otherwise, it will be equal to src1.

IO

Inputs:

Wire Name	Description
src1 (32-bit)	First input data source.
src2 (32-bit)	Second input data source.
ctrl (1-bit)	Control signal for selecting the data source.

Outputs:

Wire Name	Description
data (32-bit)	Selected output data.

Wires

No internal wires are explicitly declared in this module. The output data is directly assigned using the assign statement.

Components

The GPRMUX module itself acts as a multiplexer component responsible for selecting between two input data sources based on a control signal.

Process

- Control Signal Interpretation: Receives the control signal ctrl.
- Input Selection: Determines the selected output based on the control signal:
  - If ctrl is asserted (1), selects the second input data source src2.
  - Otherwise, selects the first input data source src1.
- Output Assignment: Assigns the selected input data source to the output port data.

ALU_Control

```
1  `define ADDU 6'b001011
2  `define SUBU 6'b001101
3  `define SLL 6'b100110
4  `define SLLV 6'b110110
5
6  module ALU_Control(
7      // Outputs
8      output reg [5:0] funct,
9      // Inputs
10     input [1:0] ALU_op,
11     input [5:0] funct_ctrl
12 );
13
14     always @(*) begin
15         case(ALU_op)
16             2'b00: funct = 6'b001001;
17             2'b01: funct = 6'b001010;
18             2'b10: begin
19                 case(funct_ctrl)
20                     `ADDU: funct = 6'b001001;
21                     `SUBU: funct = 6'b001010;
22                     `SLL: funct = 6'b100001;
23                     `SLLV: funct = 6'b110101;
24                     default: funct = 6'b000000;
25                 endcase
26             end
27             2'b11: funct = 6'b101010;
28             default: funct = 6'b000000;
29         endcase
30     end
31
32 endmodule
```

Description

The ALU_Control module functions as the control unit for the ALU (Arithmetic Logic Unit) in a CPU. It generates the appropriate function code for the ALU based on the ALU operation code and the function code control signal.

Code Explanation

Line 1~4

LUT.

Line 6~12

Module Declaration:

Defines a Verilog module named ALU_Control.

Specifies an output port funct (6-bit) for the generated ALU function code.

Specifies input ports for:

ALU_op (2-bit): ALU operation code.

funct_ctrl (6-bit): Function code control signal.

Line 14~30

ALU Function Code Generation:

Utilizes an always block triggered whenever the inputs change (@(*)).
Uses a case statement to select the appropriate ALU function code based on the ALU operation code (ALU_op) and the function code control signal (funct_ctrl).
Generates specific function codes for different ALU operations, including addition, subtraction, logical shift left, logical shift left variable, and set on less than (SLT). Opcode==00 is specified as ADD; opcode==01 is specified as SUB; opcode==11 is specified as SLTI.

IO

Inputs:

Wire Name	Description
ALU_op (2-bit)	ALU operation code.
funct_ctrl (6-bit)	Function code control signal.

Outputs:

Wire Name	Description
funct (6-bit)	Generated ALU function code.

Wires

No internal wires are explicitly declared in this module. The output funct is directly assigned within the always block.

Components

The ALU_Control module serves as a standalone component responsible for generating the ALU function code based on the provided inputs.

Process

- ALU Operation Interpretation: Receives the ALU operation code ALU_op.
- Function Code Selection: Determines the appropriate ALU function code based on the ALU operation code and the function code control signal.
- Function Code Assignment: Assigns the selected ALU function code to the output port funct.

ALU

```
1  `define ADDU 6'b001001
2  `define SUBU 6'b001010
3  `define SLL 6'b100001
4  `define SLLV 6'b110101
5  `define SLTI 6'b101010
6
7  module ALU(
8      // Outputs
9      output reg [31:0] ALU_result,
10     // Inputs
11     input [31:0] data1,
12     input [31:0] data2,
13     input [4:0] shamt,
14     input [5:0] funct
15 );
16
17     wire [31:0] ADDU_result, SUBU_result, SLL_result, SLLV_result, SLTI_result;
18
19     assign ADDU_result = data1 + data2;
20     assign SUBU_result = data1 - data2;
21     assign SLL_result = data1 << shamt;
22     assign SLLV_result = data1 << data2[4:0];
23     assign SLTI_result = data1 < data2;
24
25     always @(*) begin
26         case(funct)
27             `ADDU: ALU_result = ADDU_result;
28             `SUBU: ALU_result = SUBU_result;
29             `SLL: ALU_result = SLL_result;
30             `SLLV: ALU_result = SLLV_result;
31             `SLTI: ALU_result = SLTI_result;
32             default: ALU_result = 32'b0;
33         endcase
34     end
35
36 endmodule
```

Description

The Verilog module ALU represents the Arithmetic Logic Unit (ALU) of a processor. It performs arithmetic and logical operations on two input data sources (data1 and data2) based on the given function code (funct) and shift amount (shamt). The result of the operation is stored in the output ALU_result.

Code Explanation

Line 1~5

Operation code table.

Line 7~15

Module Declaration:

Defines a Verilog module named ALU.

Declares an output port ALU_result (32-bit) to store the result of the ALU operation.

Specifies input ports for:

data1 (32-bit): First input data source.

data2 (32-bit): Second input data source.

shamt (5-bit): Shift amount for logical shift operations.

funct (6-bit): Function code for selecting the ALU operation.

Line 17~23

ALU Operation Assignment:

Defines wires for the intermediate results of various ALU operations: ADDU_result, SUBU_result, SLL_result, SLLV_result, and SLTI_result.

Calculates the results of different ALU operations using assign statements based on the input data and shift amount.

Line 25~34

ALU Result Assignment:

Utilizes an always block triggered whenever the inputs change (@(*)).

Uses a case statement to select the appropriate ALU result based on the given function code (funct).

Assigns the selected ALU result to the output port ALU_result.

IO

Inputs:

Wire Name	Description
data1 (32-bit)	First input data source.
data2 (32-bit)	Second input data source.
shamt (5-bit)	Shift amount for logical shift operations.
funct (6-bit)	Function code for selecting the ALU operation.

Outputs:

Wire Name	Description
ALU_result (32-bit)	Result of the ALU operation.

Wires

ADDU_result, SUBU_result, SLL_result, SLLV_result, SLTI_result: Intermediate results of various ALU operations.

Components

The ALU module contains combinatorial logic to perform arithmetic and logical

operations based on the given function code.

Process

- Operation Calculation: Calculates the results of different ALU operations based on the input data sources and shift amount.
- Result Selection: Selects the appropriate ALU result based on the given function code.
- Result Assignment: Assigns the selected ALU result to the output port ALU_result.

DM

```
`define DATA_MEM_SIZE   128 // Bytes

/*
 * Declaration of Data Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module DM(
    // Outputs
    output [31:0] Mem_r_data,
    // Inputs
    input [31:0] Mem_addr,
    input [31:0] Mem_w_data,
    input Mem_w,
    input Mem_r,
    input clk
);

    /*
     * Declaration of data memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];

    // write data to memory
    always @(negedge clk) begin
        if(Mem_w) begin
            DataMem[Mem_addr[6:0]] <= Mem_w_data[31:24];
            DataMem[Mem_addr[6:0] + 1] <= Mem_w_data[23:16];
            DataMem[Mem_addr[6:0] + 2] <= Mem_w_data[15:8];
            DataMem[Mem_addr[6:0] + 3] <= Mem_w_data[7:0];
        end
    end

    // read data from memory
    assign Mem_r_data = {DataMem[Mem_addr[6:0]], DataMem[Mem_addr[6:0] + 1], DataMem[Mem_addr[6:0] + 2], DataMem[M

endmodule
```

Description

The Verilog module DM represents the Data Memory component of a processor. It is responsible for storing and retrieving data values based on memory addresses. The module consists of a memory array (DataMem) and logic for writing data to memory on the falling edge of the clock signal (clk) when the write enable signal (Mem_w) is

asserted. Additionally, it provides logic(big-endian) for reading data from memory based on the memory address provided (Mem_addr). The output Mem_r_data contains the retrieved data.

Code Explanation

Line 29

Define the memory size.

Line 35~44

Module Declaration:

Defines a Verilog module named DM.

Specifies an output port Mem_r_data (32-bit) to provide the retrieved data from memory.

Specifies input ports for:

Mem_addr (32-bit): Memory address for read/write operations.

Mem_w_data (32-bit): Data to be written into memory.

Mem_w: Write enable signal.

Mem_r: Read enable signal.

clk: Clock signal.

Line 50

Data Memory Declaration:

Declares a memory array named DataMem to store data values.

The memory array has a size determined by the parameter DATA_MEM_SIZE.

Line 53~60

Write Operation:

Utilizes an always block triggered on the falling edge(avoid racing) of the clock (negedge clk).

Checks if the write enable signal (Mem_w) is asserted.

If asserted, writes the 32-bit data Mem_w_data into the memory array DataMem at the specified address Mem_addr.

Line 63

Read Operation:

Utilizes an assign statement to concatenate four bytes from memory and assign them to Mem_r_data.

Reads data from memory at the specified address Mem_addr and provides it as output.

IO

Inputs:

Wire Name	Description
Mem_addr (32-bit)	Memory address for read/write operations.
Mem_w_data (32-bit)	Data to be written into memory.
Mem_w (1-bit)	Write enable signal.
Mem_r (1-bit)	Read enable signal.
clk (1-bit)	Clock signal.

Outputs:

Wire Name	Description
Mem_r_data (32-bit)	Retrieved data from memory.

Wires

No internal wires are explicitly declared in this module.

Components

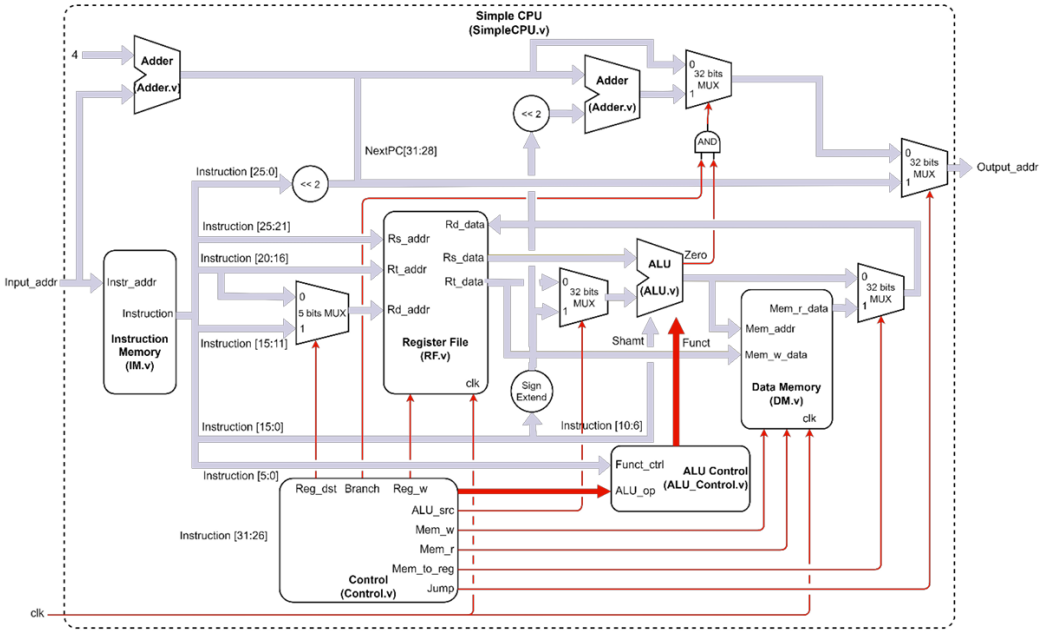
The DM module consists of a memory array (DataMem) and associated logic for read and write operations.

Process

- Write Operation: On the falling edge of the clock, if the write enable signal (Mem_w) is asserted, write the provided data into memory at the specified address.
- Read Operation: When the read enable signal (Mem_r) is asserted, retrieve data from memory at the specified address and provide it as output (Mem_r_data).

Part3

SimpleCPU




```
29  module SimpleCPU(
30      // Outputs
31      output  wire    [31:0]  Output_Addr,
32      // Inputs
33      input   wire    [31:0]  Input_Addr,
34      input   wire            clk
35  );
36
37      wire [31:0] Instruction;
38      wire [31:0] Rs_data, Rt_data;
39      wire [4:0] RdAddr;
40      wire [31:0] Rd_data;
41      wire Reg_dst, Reg_w, ALU_src, Mem_w, Mem_r, Mem_to_reg;
42      wire [31:0] imm, imm2;
43      wire [31:0] ALU_src2;
44      wire [5:0] funct;
45      wire [1:0] ALU_op;
46      wire [31:0] ALU_result;
47      wire [31:0] Mem_r_data;
48      wire Branch, Jump;
49      wire [31:0] NextPC, BranchPC, NextPC2, JumpPC;
50      wire Zero;
51
52      // sign extend imm
53      assign imm = { {16{Instruction[15]}}, Instruction[15:0] };
54      assign imm2 = { imm[29:0], 2'b00 };
55      assign JumpPC = { NextPC[31:28], Instruction[25:0], 2'b00 };
56
57      Adder NextAddr(
58          // Outputs
59          .OutputAddr (NextPC),
60          // Inputs
61          .InputAddr(Input_Addr),
62          .Offset(32'h4)
63      );
64
```

```
    Adder BranchAddr(
        // Outputs
        .OutputAddr(BranchPC),
        // Inputs
        .InputAddr (NextPC),
        .Offset(imm2)
    );

    GPRMUX BranchMUX(
        // Outputs
        .data(NextPC2),
        // Inputs
        .src1 (NextPC),
        .src2(BranchPC),
        .ctrl(Branch & Zero)
    );

    GPRMUX JumpMUX(
        // Outputs
        .data(Output_Addr),
        // Inputs
        .src1(NextPC2),
        .src2(JumpPC),
        .ctrl(Jump)
    );

    IM Instr_Memory(
        // Outputs
        .Instruction(Instruction),
        // Inputs
        .InputAddr(Input_Addr)
    );

```

```
    RegMUX Reg_MUX1(
        // Outputs
        .RegDst(RdAddr),
        // Inputs
        .Rt(Instruction[20:16]),
        .Rd(Instruction[15:11]),
        .RegDst_ctrl(Reg_dst)
    );

    RF Register_File(
        // Outputs
        .RsData(Rs_data),
        .RtData(Rt_data),
        // Inputs
        .RsAddr(Instruction[25:21]),
        .RtAddr(Instruction[20:16]),
        .RdAddr(RdAddr),
        .RdData(Rd_data),
        .RegWrite(Reg_w),
        .clk(clk)
    );

    Control Control_Unit(
        // Outputs
        .RegDst(Reg_dst),
        .RegWrite(Reg_w),
        .ALU_op(ALU_op),
        .ALU_src(ALU_src),
        .Mem_w(Mem_w),
        .Mem_r(Mem_r),
        .Mem_to_Reg(Mem_to_reg),
        .Branch(Branch),
        .Jump(Jump),
        // Inputs
        .opcode(Instruction[31:26])
    );
```

```
135     GPRMUX ALU_MUX(
136         // Outputs
137         .data(ALU_src2),
138         // Inputs
139         .src1(Rt_data),
140         .src2(imm),
141         .ctrl(ALU_src)
142     );
143 
144     ALU_Control ALU_Control_Unit(
145         // Outputs
146         .funct(funct),
147         // Inputs
148         .ALU_op(ALU_op),
149         .funct_ctrl(Instruction[5:0])
150     );
151 
152     ALU ALU1(
153         // Outputs
154         .ALU_result(ALU_result),
155         .Zero(Zero),
156         // Inputs
157         .data1(Rs_data),
158         .data2(ALU_src2),
159         .shamt(Instruction[10:6]),
160         .funct(funct)
161     );
```

```
162 
163     DM Data_Memory(
164         // Outputs
165         .Mem_r_data(Mem_r_data),
166         // Inputs
167         .Mem_addr(ALU_result),
168         .Mem_w_data(Rt_data),
169         .Mem_w(Mem_w),
170         .Mem_r(Mem_r),
171         .clk(clk)
172     );
173 
174     GPRMUX Mem_MUX1(
175         // Outputs
176         .data(Rd_data),
177         // Inputs
178         .src1(ALU_result),
179         .src2(Mem_r_data),
180         .ctrl(Mem_to_reg)
181     );
182 
183 
184 endmodule
185 
```

Description

The Verilog module SimpleCPU represents a basic single-cycle CPU design. It consists of various components such as instruction memory, register file, ALU (Arithmetic Logic Unit), data memory, control unit, and multiplexers. The CPU executes instructions fetched from memory, performs arithmetic and logical operations, accesses registers and memory, and controls the flow of instructions.

Code Explanation

Line 29~35

Module Declaration:

Defines a Verilog module named SimpleCPU.

Specifies an output port Output_Addr (32-bit) to provide the address of the next

instruction to be executed.

Specifies input ports for:

Input_Addr (32-bit): Address input for fetching instructions from memory.

clk: Clock signal.

Line 37~50

Internal Signals:

These declarations specify various wires for managing internal signals and data paths within the CPU.

Line 53~55

imm for I-type operation. imm2 for branching. JumpPC is J's address.

Line 57~181

Component Instantiation:

The module instantiates various components required for CPU operation, including instruction memory (Instr_Memory), adders (NextAddr and BranchAddr), multiplexers (BranchMUX and JumpMUX), register file (Register_File), ALU (ALU1), data memory (Data_Memory), and control unit (Control_Unit).

Each component is instantiated with appropriate inputs and outputs, and connections are made between these components to establish the CPU datapath and control logic.

IO

Inputs:

Wire Name	Description
Input_Addr (32-bit)	Address input for fetching instructions from memory.
clk (1-bit)	Clock signal.

Outputs:

Wire Name	Description
Output_Addr (32-bit)	Address of the next instruction to be executed.

Wires

Wire Name	Description
Instruction (32-bit)	Instruction fetched from IM using Input_Addr.

Rs_data, Rt_data (32-bit each)	Data read from the register file corresponding to the source registers specified in the instruction. They are used in ALU operations or data transfers.
RdAddr (5-bit)	Address of the destination register where the result of an operation will be stored.
Rd_data (32-bit)	Data read from the register file corresponding to the destination register specified in the instruction.
Reg_dst, Reg_w, ALU_src, Mem_w, Mem_r, Mem_to_reg (1-bit each)	Flags to control the whole cpu.
imm, imm2 (32-bit each)	Immediate value extracted from the instruction. imm is sign-extended to 32 bits, while imm2 is sign-extended and shifted left by two bits, used for branch instructions.
ALU_src2 (32-bit)	The second operand for ALU operations.
funct (6-bit), ALU_op (2-bit)	Control signals for ALU operations.
ALU_result (32-bit)	Result of ALU operation.
Mem_r_data	Data read from the data memory.
Branch, Jump (1-bit each)	Whether a branch or jump instruction will be being executed.
NextPC, BranchPC, NextPC2, JumpPC (32-bit each)	These wires hold the addresses of the next instruction, branch target, next instruction after a branch, and jump target, respectively.
Zero (1-bit)	Indicates whether the result of an ALU operation is zero, used for branch instructions to determine whether to take the branch or not.

Components

Component Name	**Description**
IM	Stores and provides instructions to the CPU.

Adder	Compute addresses for the next instruction and branch targets.
GPRMUX, RegMUX	Select between different data sources based on control signals.
RF	Stores and provides data from registers.
ALU	Performs arithmetic and logical operations.
DM	Stores and retrieves data values.
Control	Generates control signals based on instruction opcode.

Process

- Instruction Fetch: Fetches instructions from memory based on the provided address.
- Decode and Control: Decodes the fetched instruction and generates control signals.
- Register and Memory Access: Accesses registers and memory based on the decoded instruction.
- ALU Operation: Performs arithmetic or logical operations based on the instruction and operand values.
- Writeback: Writes results back to registers or memory.
- Branch and Jump Handling: Computes next instruction address based on branch or jump instructions.

Adder

It's the same as Part1's.

GPRMUX

It's the same as Part2's.

IM

It's the same as Part1's.

RegMUX

It’s the same as Part2’s.

RF

It’s the same as Part1’s.

Control

```
`define R_TYPE 6'b000000
`define SUBIU 6'b001101
`define SW 6'b010000
`define LW 6'b010001
`define SLTI 6'b101010
`define BEQ 6'b010011
`define J 6'b011100

module Control(
    // Outputs
    output reg RegDst,
    output reg RegWrite,
    output reg [1:0] ALU_op,
    output reg ALU_src,
    output reg Mem_w,
    output reg Mem_r,
    output reg Mem_to_Reg,
    output reg Branch,
    output reg Jump,
    // Inputs
    input [5:0] opcode
);

    always @(*) begin
        case(opcode)
            `R_TYPE: begin
                RegDst <= 1'b1;
                RegWrite <= 1'b1;
                ALU_op <= 2'b10;
                ALU_src <= 1'b0;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
            end
```

```
            `SUBIU: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b1;
                ALU_op <= 2'b01;
                ALU_src <= 1'b1;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
            end
            `SW: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b0;
                ALU_op <= 2'b00;
                ALU_src <= 1'b1;
                Mem_w <= 1'b1;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
            end
            `LW: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b1;
                ALU_op <= 2'b00;
                ALU_src <= 1'b1;
                Mem_w <= 1'b0;
                Mem_r <= 1'b1;
                Mem_to_Reg <= 1'b1;
                Branch <= 1'b0;
                Jump <= 1'b0;
            end
```

```
            `SLTI: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b1;
                ALU_op <= 2'b11;
                ALU_src <= 1'b1;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b0;
            end
            `BEQ: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b0;
                ALU_op <= 2'b01;
                ALU_src <= 1'b0;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b1;
                Jump <= 1'b0;
            end
            `J: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b0;
                ALU_op <= 2'b00;
                ALU_src <= 1'b0;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
                Branch <= 1'b0;
                Jump <= 1'b1;
            end
```

```
            end
            default: begin
                RegDst <= 1'b0;
                RegWrite <= 1'b0;
                ALU_op <= 2'b00;
                ALU_src <= 1'b0;
                Mem_w <= 1'b0;
                Mem_r <= 1'b0;
                Mem_to_Reg <= 1'b0;
            end
        endcase
    end
endmodule
```

Description

The Control module serves as the control unit of a CPU, responsible for generating control signals based on the opcode of an instruction. It determines how the CPU should process the instruction, including register operations, ALU operations, and memory access.

Code Explanation

Line 1~7

Opcode table.

Line 9~22

Module Declaration:

Defines a Verilog module named Control.

Specifies output ports for various control signals:

- RegDst: Register destination control signal.
- RegWrite: Register write enable control signal.
- ALU_op: ALU operation control signal.
- ALU_src: ALU source control signal.
- Mem_w: Memory write enable control signal.
- Mem_r: Memory read enable control signal.
- Mem_to_Reg: Memory to register control signal.
- Branch, Jump: Control signal for branching.

Specifies an input port opcode (6-bit) for the opcode of the instruction.

Line 24~113

Control Signal Generation:

Utilizes an always block triggered whenever the inputs change (@(*)).

Uses a case statement to select control signals based on the opcode of the instruction.

Generates specific control signals for different types of instructions, including R-type, immediate arithmetic, and memory access instructions. If not recognized, it'll use default condition.

IO

Inputs:

Wire Name	**Description**

opcode (6-bit)	Opcode of the instruction.

Outputs:

Wire Name	Description
RegDst (1-bit)	Register destination control signal.
RegWrite (1-bit)	Register write enable control signal.
ALU_op (2-bit)	ALU operation control signal.
ALU_src (1-bit)	ALU source control signal.
Mem_w (1-bit)	Memory write enable control signal.
Mem_r (1-bit)	Memory read enable control signal.
Mem_to_Reg (1-bit)	Memory to register control signal.
Branch, Jump (1-bit each)	Flags for branching

Wires

No internal wires are explicitly declared in this module. Control signals are directly assigned based on the opcode using the always block.

Components

The Control module acts as a standalone component responsible for generating control signals based on the opcode of an instruction.

Process

- Opcode Interpretation: Receives the opcode of the instruction.
- Control Signal Selection: Determines the appropriate control signals based on the opcode using a case statement.
- Control Signal Assignment: Assigns the selected control signals to the output ports.

ALU_Control

It's the same as Part2's.

ALU

```
1  `define ADDU 6'b001001
2  `define SUBU 6'b001010
3  `define SLL 6'b100001
4  `define SLLV 6'b110101
5  `define SLTI 6'b101010
6
7  module ALU(
8      // Outputs
9      output reg [31:0] ALU_result,
10     output Zero,
11     // Inputs
12     input [31:0] data1,
13     input [31:0] data2,
14     input [4:0] shamt,
15     input [5:0] funct
16 );
17
18     wire [31:0] ADDU_result, SUBU_result, SLL_result, SLLV_result, SLTI_result;
19
20     assign ADDU_result = data1 + data2;
21     assign SUBU_result = data1 - data2;
22     assign SLL_result = data1 << shamt;
23     assign SLLV_result = data1 << data2[4:0];
24     assign SLTI_result = data1 < data2;
25
26     assign Zero = (ALU_result == 32'b0);
27
28     always @(*) begin
29         case(funct)
30             `ADDU: ALU_result = ADDU_result;
31             `SUBU: ALU_result = SUBU_result;
32             `SLL: ALU_result = SLL_result;
33             `SLLV: ALU_result = SLLV_result;
34             `SLTI: ALU_result = SLTI_result;
35             default: ALU_result = 32'b0;
36         endcase
37     end
38
```

Description

The Verilog module ALU represents an Arithmetic Logic Unit (ALU) designed to perform arithmetic and logical operations on two input operands. It supports operations such as addition, subtraction, left logical shift, and set less than (SLT) comparison.

Code Explanation

Line 1~5

LUT.

Line 7~16

Module Declaration:

Defines a Verilog module named ALU.

Specifies output ports:

ALU_result (32-bit): Result of the ALU operation.

Zero: Output indicating whether the ALU result is zero.

Specifies input ports for:

data1, data2 (32-bit): Input operands to the ALU.

shamt (5-bit): Shift amount for logical shift operations.

funct (6-bit): Function code determining the operation to be performed by the ALU.

Line 18~24

Operation Execution:

Intermediate results are computed based on the input operands and specified operations:

ADDU_result: Addition of data1 and data2.

SUBU_result: Subtraction of data2 from data1.

SLL_result: Left logical shift of data1 by shamt.

SLLV_result: Left logical shift of data1 by the lower 5 bits of data2.

SLTI_result: Comparison of data1 and data2 for the SLT operation.

Line 26

Zero Flag:

The Zero output is assigned based on whether the ALU_result is equal to zero.

Line 28~37

ALU Operation:

The ALU_result is determined based on the function code (funct):

If funct matches predefined operation codes, the corresponding intermediate result is assigned to ALU_result.

If no match is found, ALU_result is set to zero.

IO

Inputs:

Wire Name	**Description**
data1, data2 (32-bit)	Input operands to the ALU.
shamt (5-bit)	Shift amount for logical shift operations.
funct (6-bit)	Function code determining the operation to be performed by the ALU.

Outputs:

Wire Name	**Description**
ALU_result (32-bit)	Result of the ALU operation.
Zero (1-bit)	Output indicating whether the ALU result is zero.

Wires

ADDU_result, SUBU_result, SLL_result, SLLV_result, SLTI_result: Wires for storing intermediate computation results of various ALU operations.

Components

This module does not instantiate any external components. It internally computes the ALU operations based on input operands and function codes.

Process

- Input Processing: Receive input operands (data1, data2), shift amount (shamt), and function code (funct).
- Operation Execution: Perform the specified ALU operation based on the function code and input operands.
- Result Assignment: Assign the computed result (ALU_result) to the output port.
- Zero Flag Evaluation: Determine if the result is zero and assign the appropriate value to the Zero output.

DM

It's the same as Part2's.

Execution Results

All the registers' data are using the data from testbench. There is no change in those files.

Helper programs

Translator.py

```python
def R_type(machine_code, R_table, width=10):
    opcode = machine_code >> 26
    rs = (machine_code >> 21) & 0x1F
    rt = (machine_code >> 16) & 0x1F
    rd = (machine_code >> 11) & 0x1F
    shamt = (machine_code >> 6) & 0x1F
    funct = machine_code & 0x3F

    text = f"{R_table.get(funct, 'Unknown')}"
    print(f"{text:<{width}}", end="")
    text = f"rs: {rs}"
    print(f"{text:<{width}}", end="")
    text = f"rt: {rt}"
    print(f"{text:<{width}}", end="")
    text = f"rd: {rd}"
    print(f"{text:<{width}}", end="")
    text = f"shamt: {shamt}"
    print(f"{text:<{width}}", end="")

def I_type(machine_code, I_table, width=10):
    opcode = machine_code >> 26
    rs = (machine_code >> 21) & 0x1F
    rt = (machine_code >> 16) & 0x1F
    imm = machine_code & 0xFFFF

    text = f"{I_table.get(opcode, 'Unknown')}"
    print(f"{text:<{width}}", end="")
    text = f"op: {opcode}"
    print(f"{text:<{width}}", end="")
    text = f"rs: {rs}"
    print(f"{text:<{width}}", end="")
    text = f"rt: {rt}"
    print(f"{text:<{width}}", end="")
    text = f"imm: {imm}"
    print(f"{text:<{width}}", end="")
```

```python
def translate(machine_code, tables):
    opcode = machine_code >> 26
    if opcode == 0:
        R_type(machine_code, tables[0])
    elif opcode == 0b011100:
        print("Jump to address {%d || PC[31:28]}" % (machine_code & 0x3FFFFFF), end="")
    else:
        I_type(machine_code, tables[1])


Rtable = {0b001011: "ADDU", 0b001101: "SUBU", 0b100110: "SLL", 0b110110: "SLLV"}
Itable = {
    0b001101: "SUBI",
    0b010000: "SW",
    0b010001: "LW",
    0b101010: "SLTI",
    0b010011: "BEQ",
}
tables = [Rtable, Itable]

with open("testbench/IM.dat", "r") as f:
    _ = f.readline()

    while True:
        machine_code = ""
        for _ in range(4):
            line = f.readline()
            if not line:
                break
            machine_code += line[0:2]

        if not machine_code:
            break

        machine_code = int(machine_code, 16)
        translate(machine_code, tables)
        print()

```

Description

This Python script is a simple translator for MIPS machine code instructions. It translates the instructions into human-readable assembly mnemonics along with their corresponding fields, such as opcode, registers, and immediate values.

Conversion Results

Part1:

ADDU	rs: 10	rt: 11	rd: 20	shamt: 0
SUBU	rs: 12	rt: 13	rd: 21	shamt: 0
SLL	rs: 14	rt: 15	rd: 24	shamt: 3
SLLV	rs: 15	rt: 16	rd: 26	shamt: 6

Part2:

SUBI	op: 13	rs: 10	rt: 25	imm: 9
SW	op: 16	rs: 12	rt: 25	imm: 3
LW	op: 17	rs: 12	rt: 27	imm: 3
SLTI	op: 42	rs: 10	rt: 26	imm: 20

Part3:

BEQ	op: 19	rs: 0	rt: 19	imm: 30

SUBU rs: 19 rt: 2 rd: 19 shamt: 0

Jump to address {0 || PC[31:28]}

Convertor.py

```python
funct_table = {"ADDU": 0b001011, "SUBU": 0b001101, "SLL": 0b100110, "SLLV": 0b110110}
opcode_table = {
    "ADDI": 0b001100,
    "SUBI": 0b001101,
    "SW": 0b010000,
    "LW": 0b010001,
    "SLTI": 0b101010,
    "BEQ": 0b010011,
    "J": 0b011100,
}
register_table = {
    "$zero": 0,
    "$at": 1,
    "$v0": 2,
    "$v1": 3,
    "$a0": 4,
    "$a1": 5,
    "$a2": 6,
    "$a3": 7,
    "$t0": 8,
    "$t1": 9,
    "$t2": 10,
    "$t3": 11,
    "$t4": 12,
    "$t5": 13,
    "$t6": 14,
    "$t7": 15,
    "$s0": 16,
    "$s1": 17,
    "$s2": 18,
    "$s3": 19,
    "$s4": 20,
    "$s5": 21,
    "$s6": 22,
    "$s7": 23,
    "$t8": 24,
    "$t9": 25,
    "$k0": 26,
    "$k1": 27,
    "$gp": 28,
    "$sp": 29,
    "$fp": 30,
    "$ra": 31,
}


def twos_complement(number, bit_width):
    # Convert number to positive binary representation
    positive_binary = bin(abs(number))[2:]

    # Pad with zeros to reach the desired bit width
    positive_binary = positive_binary.zfill(bit_width)

    if number < 0:
        # Take the bitwise NOT of the positive binary representation
        inverted_binary = "".join("1" if bit == "0" else "0" for bit in positive_binary)

        # Add 1 to the inverted binary representation
        inverted_plus_one = bin(int(inverted_binary, 2) + 1)[2:]

        # Pad with zeros to reach the desired bit width
        inverted_plus_one = inverted_plus_one.zfill(bit_width)

        return inverted_plus_one
    else:
        return positive_binary.zfill(bit_width)
```

```python
def convert_to_binary(instructions):
    machine_code = ""
    label_addresses = {}
    current_address = 0

    for instruction in instructions:
        instruction = instruction.strip()
        if ":" in instruction:
            # If the instruction is a label, record its address
            label = instruction.split(":")[0]
            label_addresses[label] = current_address
            instruction = instruction.split(":")[1].strip()
        current_address += 1

    for instruction in instructions:
        instruction = instruction.strip()
        if ":" in instruction:
            instruction = instruction.split(":")[1].strip()
        instruction = instruction.split()
        instruction[0] = instruction[0].upper()
        args = "".join(instruction[1:]).split(",")
        # R-type
        if instruction[0] in funct_table:
            funct = funct_table[instruction[0]]
            rs = register_table[args[1]]
            rt = register_table[args[2]]
            rd = register_table[args[0]]
            machine_code += f"1000000{rs:05b}{rt:05b}{rd:05b}00000{funct:06b}\n"
        # J-type
        elif instruction[0] == "J":
            opcode = opcode_table[instruction[0]]
            imm = label_addresses.get(args[0], -1)
            if imm == -1:
                imm = twos_complement(int(args[0]), 26)
            else:
                imm = twos_complement(imm, 26)
            machine_code += f"1{opcode:06b}{imm}\n"
```

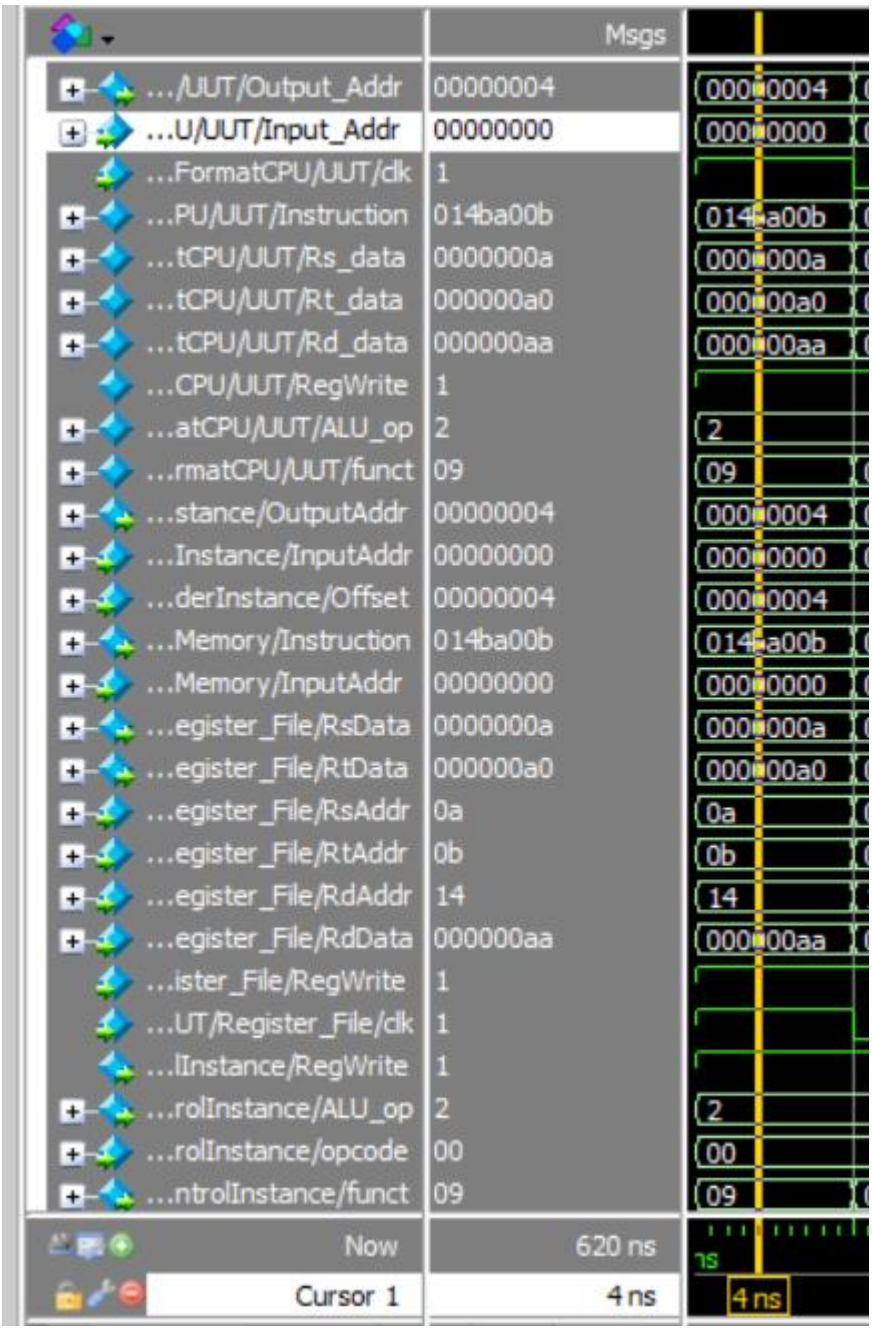
```python
            machine_code += f"1{opcode:06b}{imm}\n"
        elif instruction[0] == "SW" or instruction[0] == "LW":
            opcode = opcode_table[instruction[0]]
            rs = register_table[args[1].split("(")[1].replace(")", "")]
            rt = register_table[args[0]]
            imm = twos_complement(int(args[1].split("(")[0]), 16)
            machine_code += f"1{opcode:06b}{rs:05b}{rt:05b}{imm}\n"
        # I-type
        elif instruction[0] in opcode_table:
            opcode = opcode_table[instruction[0]]
            rs = register_table[args[1]]
            rt = register_table[args[0]]
            imm = label_addresses.get(args[2], -1)
            if imm == -1:
                imm = twos_complement(int(args[2]), 16)
            else:
                if instruction[0] == "BEQ":
                    imm = twos_complement(imm - current_address - 1, 16)
                else:
                    imm = twos_complement(imm, 16)

            machine_code += f"1{opcode:06b}{rs:05b}{rt:05b}{imm}\n"
        else:
            continue
        current_address += 1

    return machine_code, label_addresses


with open("testbench/program.txt", "r") as f:
    instructions = f.readlines()

    machine_code, labels = convert_to_binary(instructions)
    machine_code = machine_code.split("\n")[:-1]
    for i in range(len(machine_code)):
        print(f"{i}: {hex(int(machine_code[i], 2))[3:]}")
    print("Labels:", labels)
```

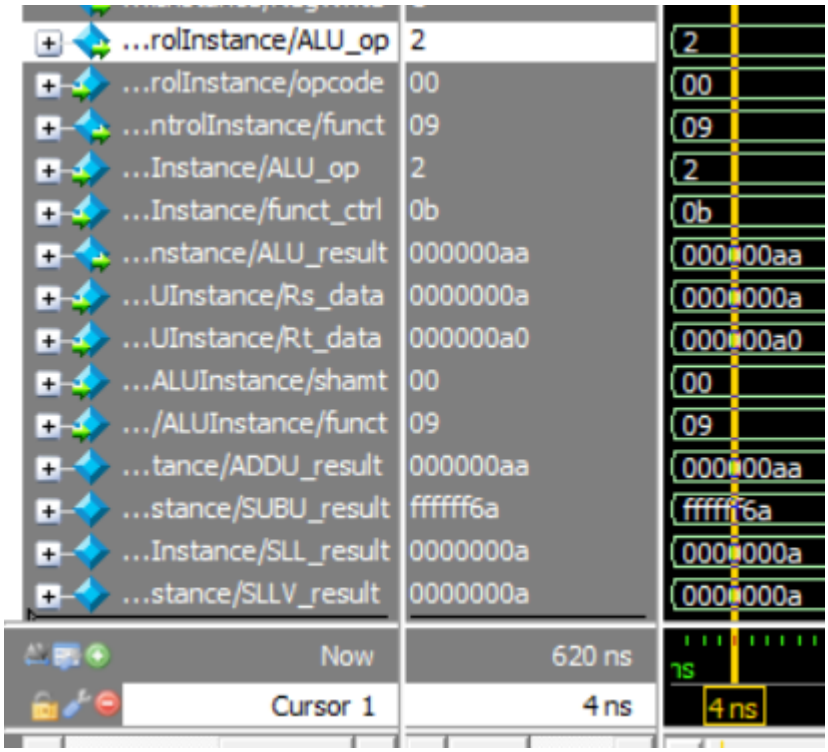
```
143
144 with open("testbench/IM.dat", "w") as f:
145     # Output:
146     byte = 0
147     for code in machine_code:
148         h = hex(int(code, 2))[3:]
149         # from highest byte write to lowest byte
150         f.write(f"{h[0:2]}\n")
151         f.write(f"{h[2:4]}\n")
152         f.write(f"{h[4:6]}\n")
153         f.write(f"{h[6:8]}\n")
154         byte += 4
155
156     for _ in range(128 - byte):
157         f.write("FF\n")
158
```

Description

This program can convert MIPS assembly into machine code required in this PA. This can compile labels.

Part1


Msgs
.../UUT/Output_Addr 00000004
...U/UUT/Input_Addr 00000000
...FormatCPU/UUT/clk 1
...PU/UUT/Instruction 014ba00b
...tCPU/UUT/Rs_data 0000000a
...tCPU/UUT/Rt_data 000000a0
...tCPU/UUT/Rd_data 000000aa
...CPU/UUT/RegWrite 1
...atCPU/UUT/ALU_op 2
...rmatCPU/UUT/funct 09
...stance/OutputAddr 00000004
...Instance/InputAddr 00000000
...derInstance/Offset 00000004
...Memory/Instruction 014ba00b
...Memory/InputAddr 00000000
...egister_File/RsData 0000000a
...egister_File/RtData 000000a0
...egister_File/RsAddr 0a
...egister_File/RtAddr 0b
...egister_File/RdAddr 14
...egister_File/RdData 000000aa
...ister_File/RegWrite 1
...UT/Register_File/clk 1
...lInstance/RegWrite 1
...rolInstance/ALU_op 2
...rolInstance/opcode 00
...ntrolInstance/funct 09
Now 620 ns
Cursor 1 4 ns
4 ns

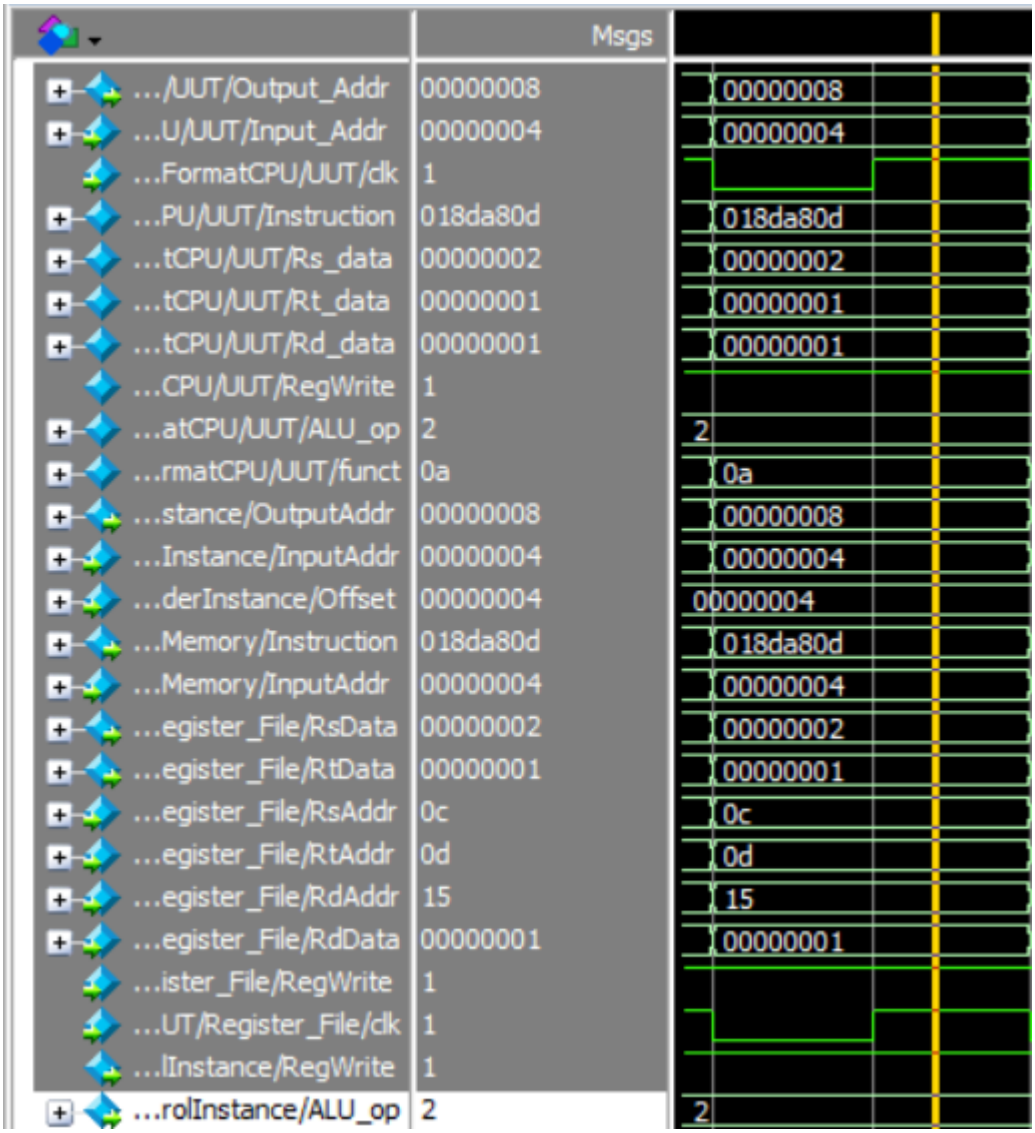


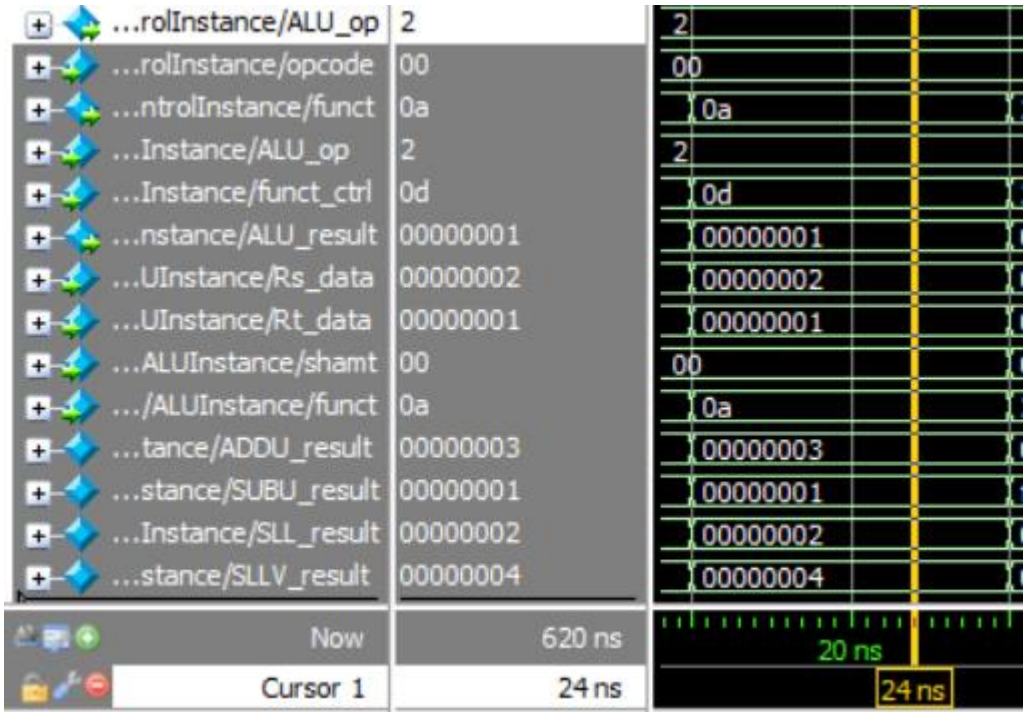
Descriptions:

Cycle 1

- Input_Addr is 0, Output_Addr should be 0+4=4 which is correct.

- Instruction is read from IM in big-endian. 0x014ba00b is valid operation. For specific, it is ADDU $20, $10, $11.
- Rs_data, Rt_data are 0x0000_000A and 0x0000_00A0, respectively verified by looking up in the input file.
- Rd_data is the ALU result. In this case, it's rd=rs+rt=0x0000_00AA which is correct.
- Because it's R-type, RegWrite is high; ALU_op is 2'b10 which is correct.
- UUT/funct is 0x09. This is generated from ALU_Control to tell ALU to execute ADDU.
- The following 3 vars is IO of Adder to compute next PC addr. Offset is a constant, 32'h4.
- And then is the operation of fetching instruction from IM with address 0 and store to Instruction.
- Following is the result of fetching RsData, RtData from RF with address specified by RsAddr(10), RtAddr(11). This process also save RdData to RdAddr(20). The saving process will be done at negedge clk to avoid racing.
- ALU_op=2, opcode=0 for R-type. funct_ctrl is 0x0b specified for ADDU by looking up the PA2 pdf. This should set funct to 0x09 to tell ALU to do ADDU operation.
- The following vars are the IO of ALU. We can see that the ALU is actually generating 4 outputs which are ADDU, SUBU, SLL, SLLV, and I select the ADDU by funct(0x09).


Msgs
.../UUT/Output_Addr
...U/UUT/Input_Addr
...FormatCPU/UUT/clk
...PU/UUT/Instruction
...tCPU/UUT/Rs_data
...tCPU/UUT/Rt_data
...tCPU/UUT/Rd_data
...CPU/UUT/RegWrite
...atCPU/UUT/ALU_op
...rmatCPU/UUT/funct
...stance/OutputAddr
...Instance/InputAddr
...derInstance/Offset
...Memory/Instruction
...Memory/InputAddr
...egister_File/RsData
...egister_File/RtData
...egister_File/RsAddr
...egister_File/RtAddr
...egister_File/RdAddr
...egister_File/RdData
...ister_File/RegWrite
...UT/Register_File/clk
...lInstance/RegWrite
...rolInstance/ALU_op
00000008
00000004
1
018da80d
00000002
00000001
00000001
1
2
0a
00000008
00000004
00000004
018da80d
00000004
00000002
00000001
0c
0d
15
00000001
1
1
1
2



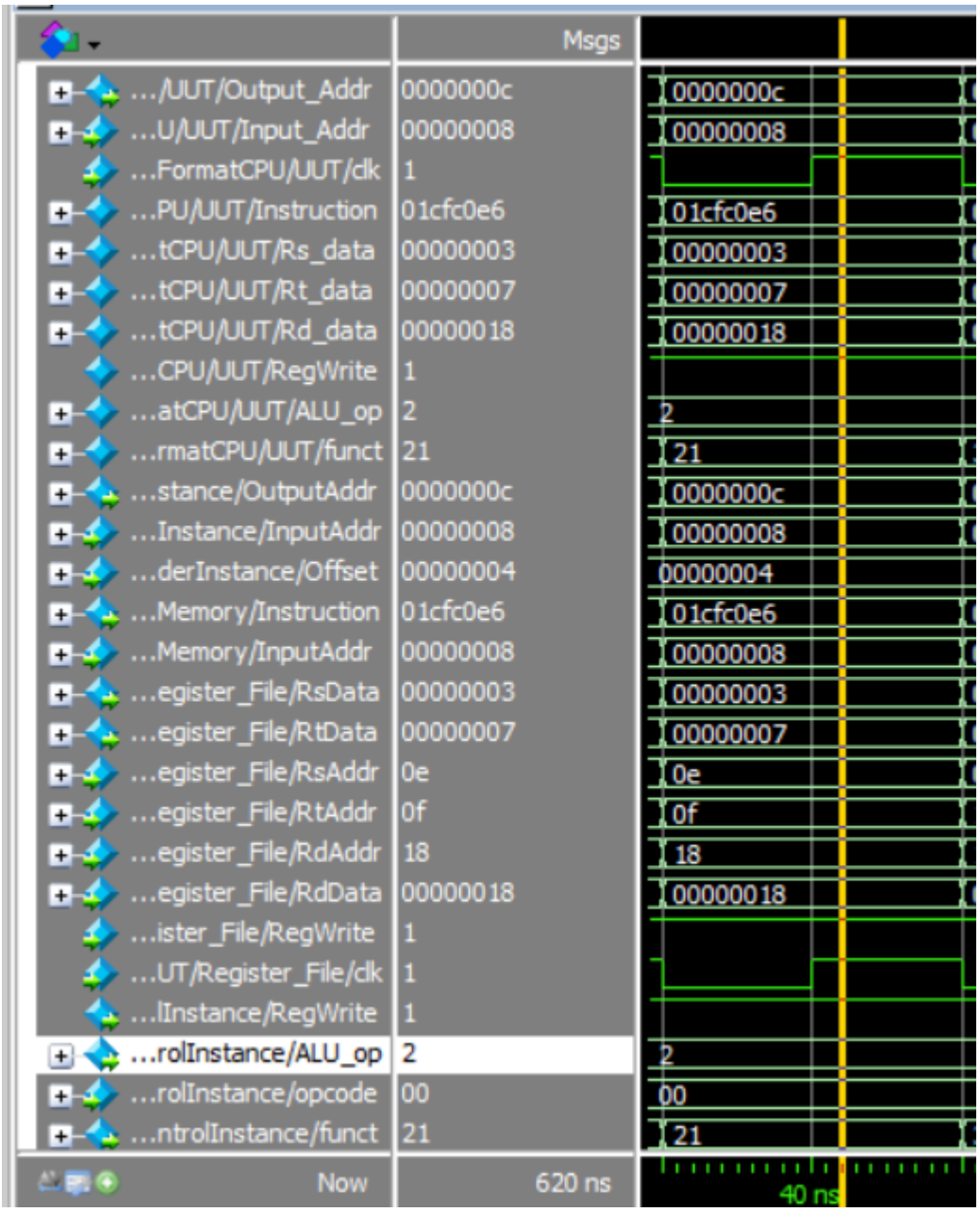
Cycle 2

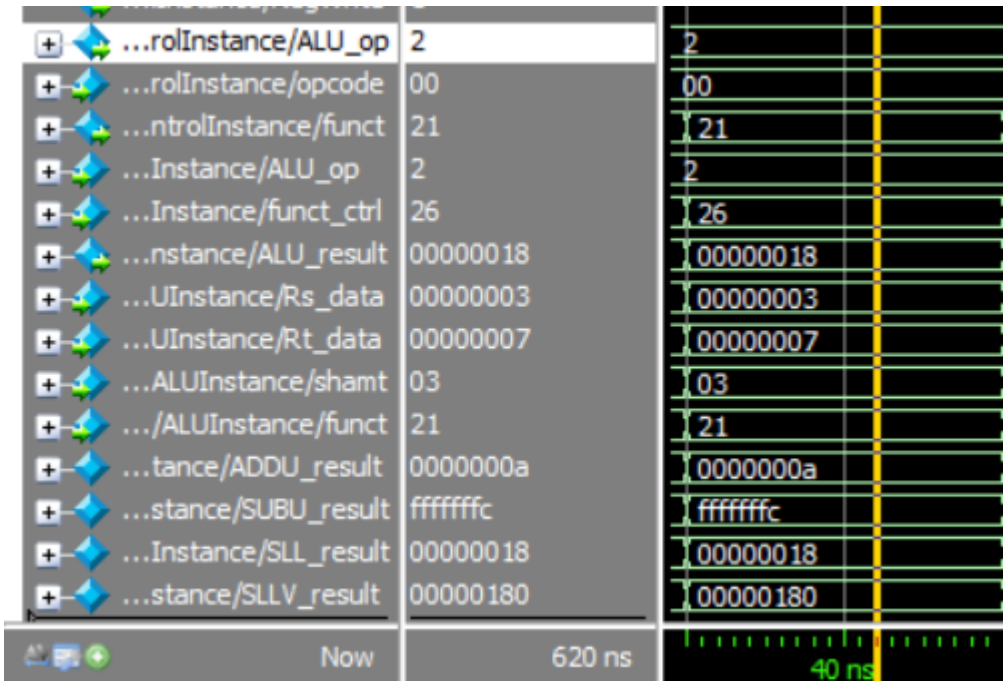
- The next address is given after the negedge clk because all by component is combinational circuit we can see that the output is already generated as the instruction address is updated.

- This instruction is 0x018da80d which is SUBU $21, $12, $13. That is R[21]=R[12]-

R[13]=2-1=1

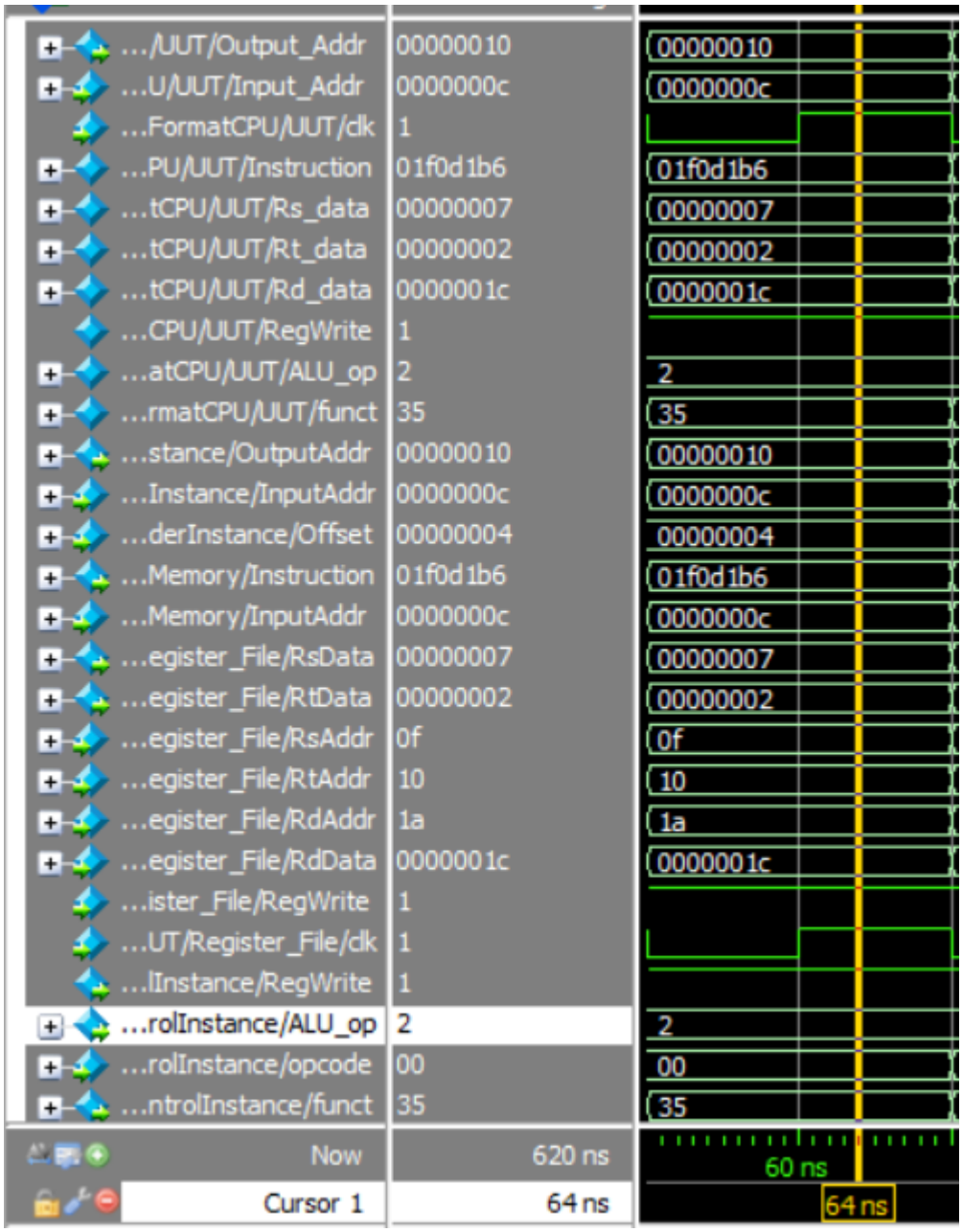
- UUT/funct is 0x0a which is SUBU's funct
- The other parts' logic remain the same as Cycle 1.

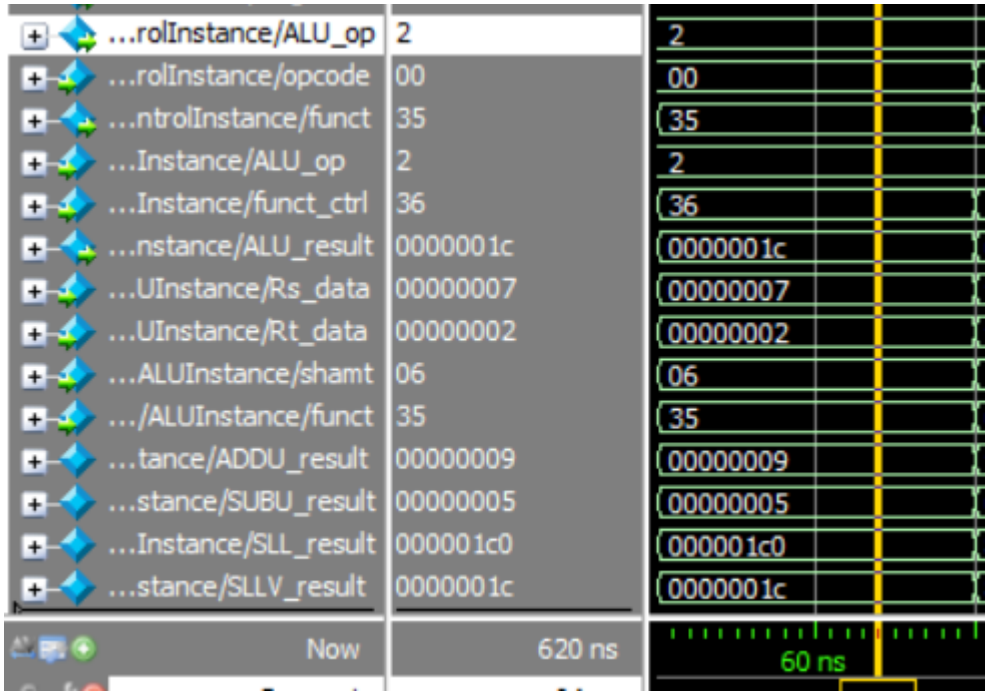




Cycle 3

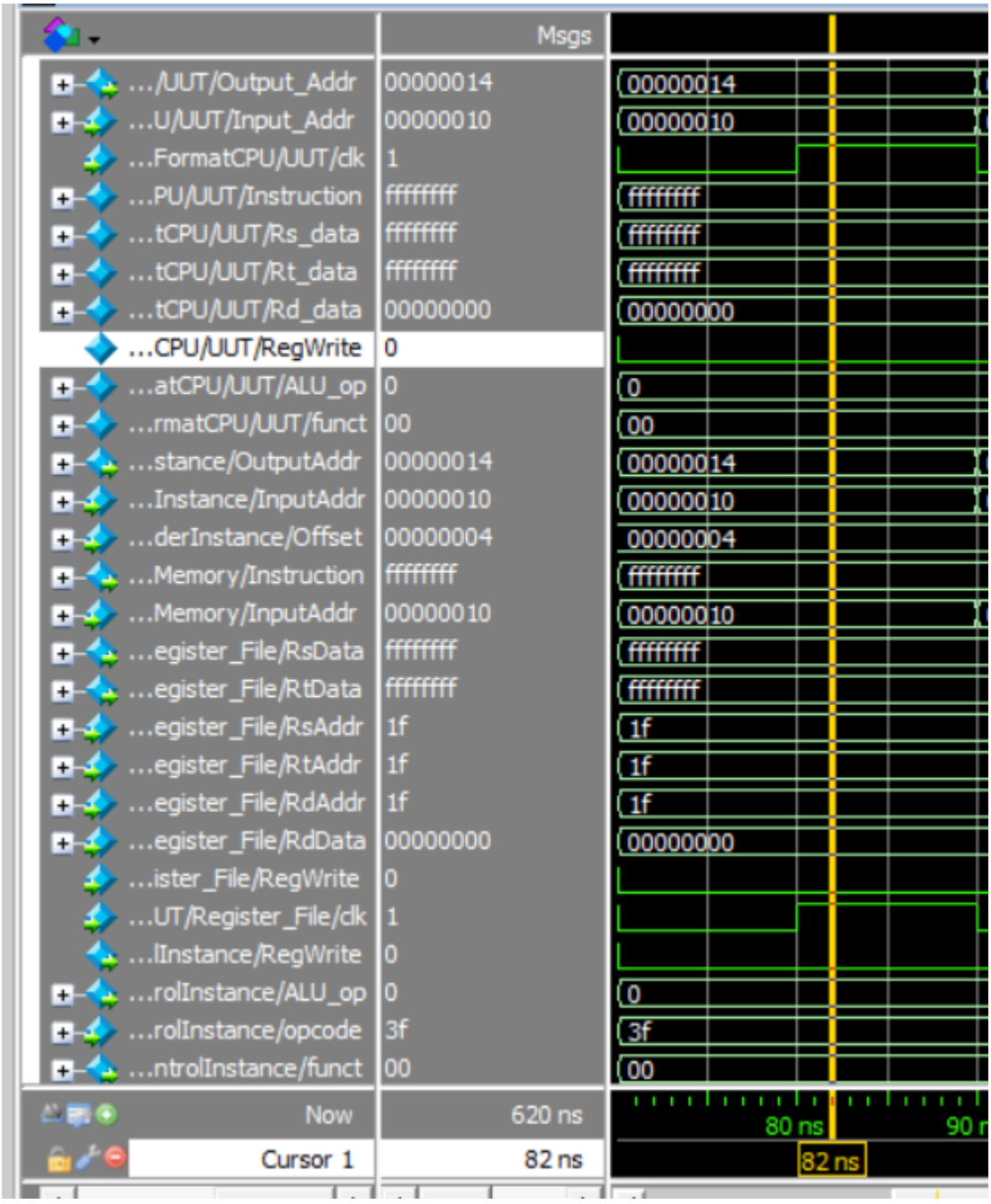
- The instruction is 0x01cfc0e6 which is SLL $24, $14, 3. R[24]=R[14]<<3 = 3 << 3 = 24 = 0x18. Rt's address is also given which is 15 and the data(0x7) is fetched, but it didn't used
- The remaining logic remains the same as Cycle 1.


.../UUT/Output_Addr 00000010
...U/UUT/Input_Addr 0000000c
...FormatCPU/UUT/clk 1
...PU/UUT/Instruction 01f0d1b6
...tCPU/UUT/Rs_data 00000007
...tCPU/UUT/Rt_data 00000002
...tCPU/UUT/Rd_data 0000001c
...CPU/UUT/RegWrite 1
...atCPU/UUT/ALU_op 2
...rmatCPU/UUT/funct 35
...stance/OutputAddr 00000010
...Instance/InputAddr 0000000c
...derInstance/Offset 00000004
...Memory/Instruction 01f0d1b6
...Memory/InputAddr 0000000c
...egister_File/RsData 00000007
...egister_File/RtData 00000002
...egister_File/RsAddr 0f
...egister_File/RtAddr 10
...egister_File/RdAddr 1a
...egister_File/RdData 0000001c
...ister_File/RegWrite 1
...UT/Register_File/clk 1
...lInstance/RegWrite 1
...rolInstance/ALU_op 2
...rolInstance/opcode 00
...ntrolInstance/funct 35
Now 620 ns
Cursor 1 64 ns
60 ns
64 ns



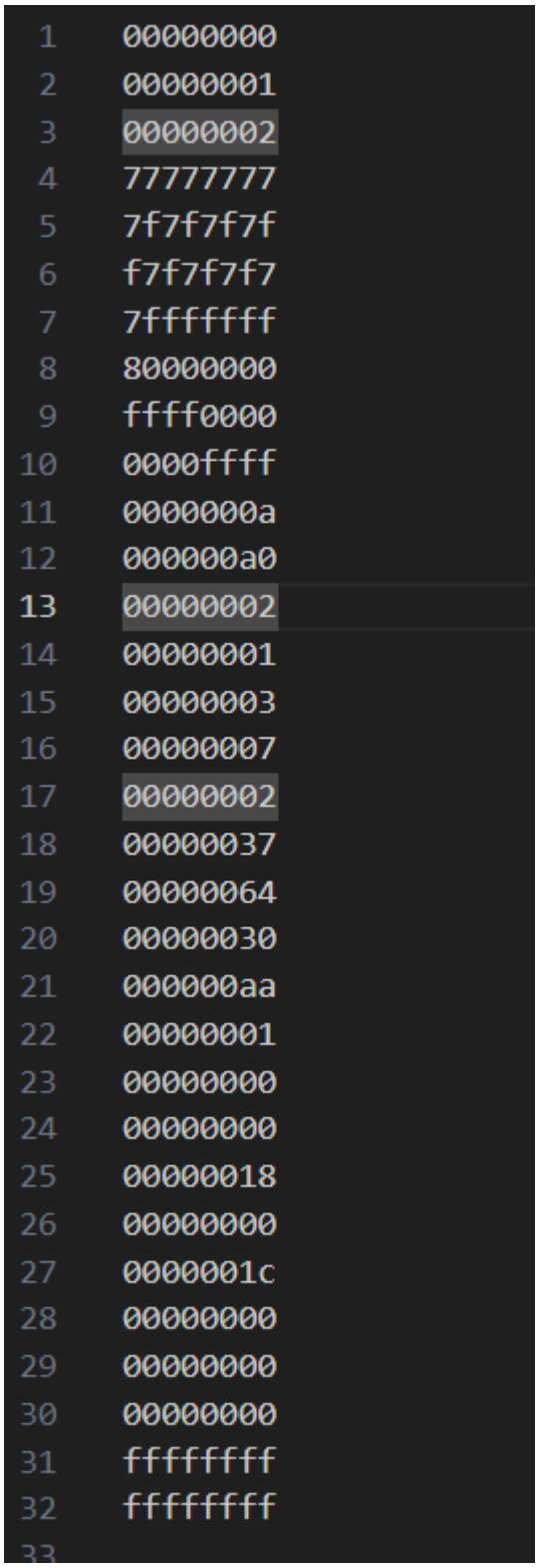
Cycle 4

- The next instruction is 0x01f0d1b6 which is SLLV $26, $15, $16. R[26]=R[15]<<R[16][4:0]=7<<2=28=0x1c.
- The remaining logic is the same as Cycle 1.



The program will execute all the way to the end. To not incorrectly affect the output, I add default condition to Control. If the opcode is not recognized. It'll use default operation which is do nothing.

After the operations above. This is the result:



$20 (line 21) = $10+$11 = 0xaa

$21 (line 22) = $12-$13 = 0x01

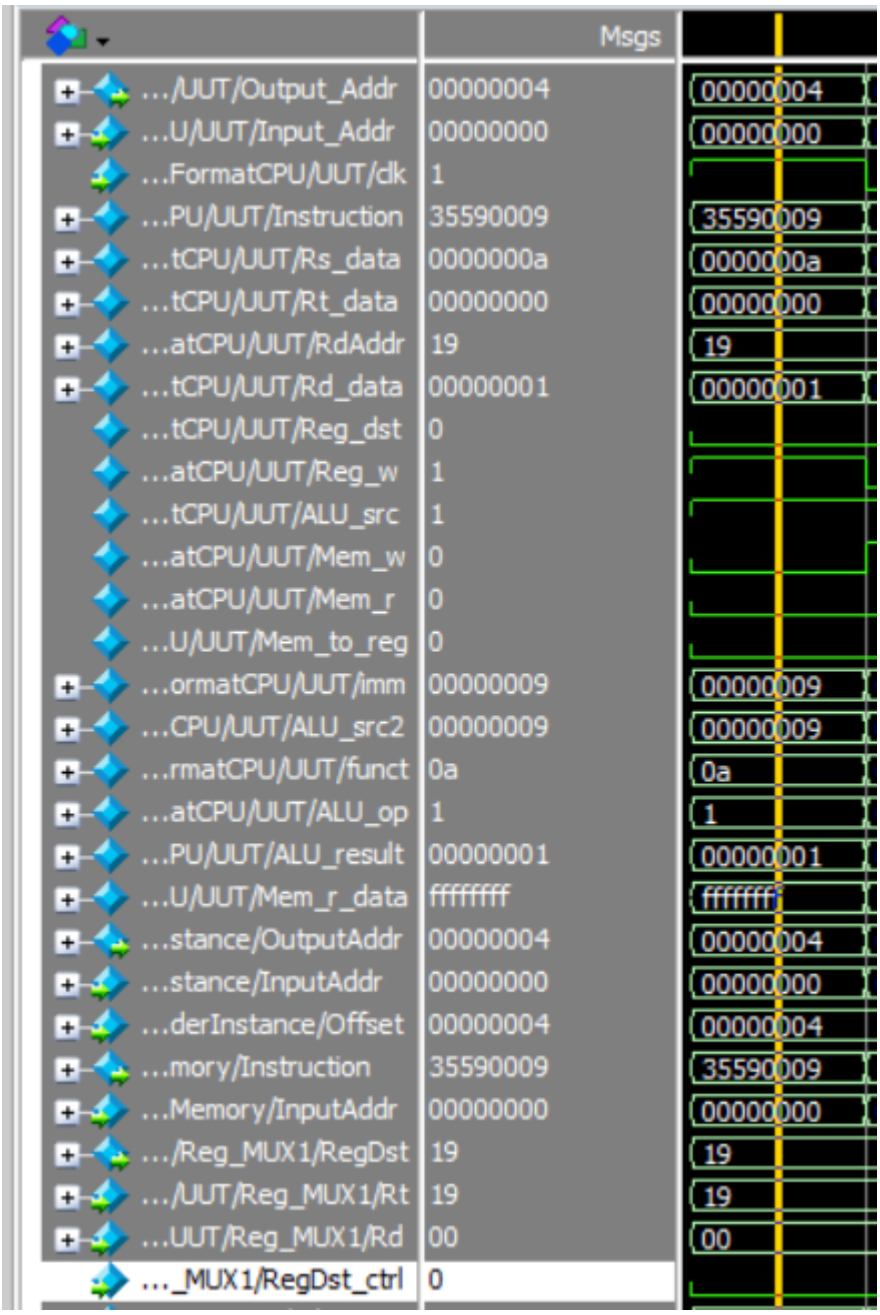
$24 (line 25) = $14<<3 = 0x18

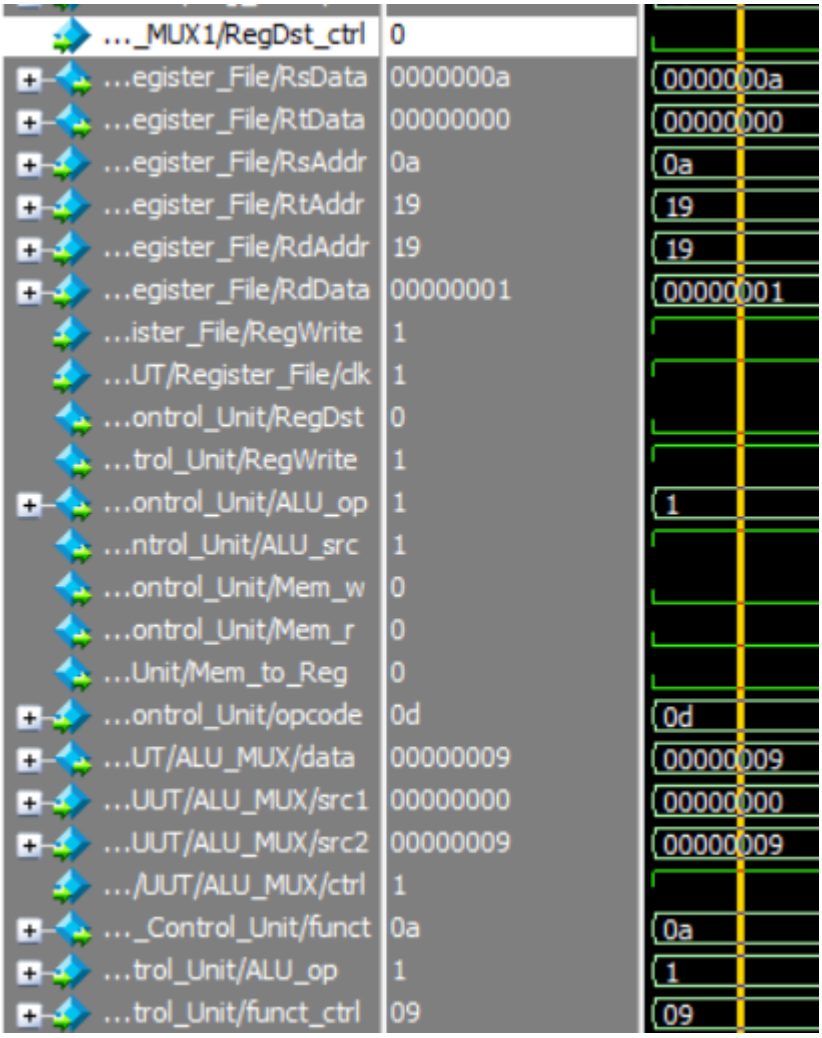
$26 (line 27) = $15<<$16[4:0] = 0x1c

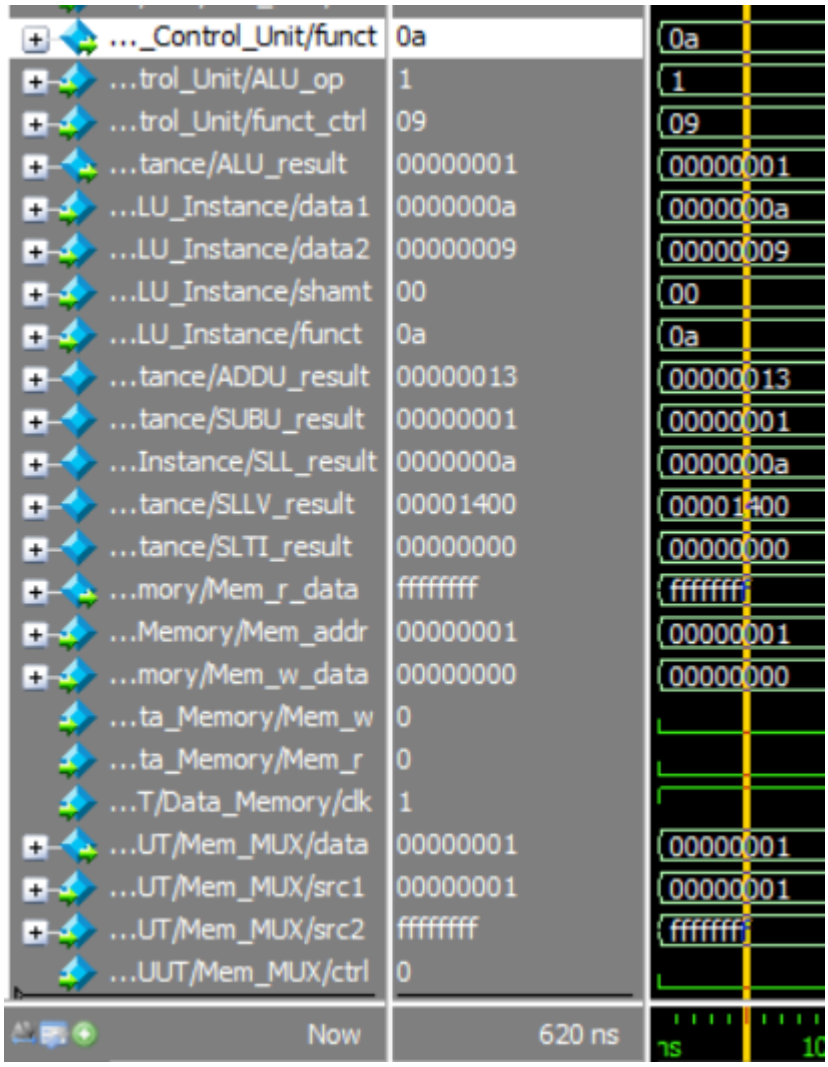
And other register remain the same protected by the default condition of the control

units.

Part2


Msgs
.../UUT/Output_Addr 00000004
...U/UUT/Input_Addr 00000000
...FormatCPU/UUT/clk 1
...PU/UUT/Instruction 35590009
...tCPU/UUT/Rs_data 0000000a
...tCPU/UUT/Rt_data 00000000
...atCPU/UUT/RdAddr 19
...tCPU/UUT/Rd_data 00000001
...tCPU/UUT/Reg_dst 0
...atCPU/UUT/Reg_w 1
...tCPU/UUT/ALU_src 1
...atCPU/UUT/Mem_w 0
...atCPU/UUT/Mem_r 0
...U/UUT/Mem_to_reg 0
...ormatCPU/UUT/imm 00000009
...CPU/UUT/ALU_src2 00000009
...rmatCPU/UUT/funct 0a
...atCPU/UUT/ALU_op 1
...PU/UUT/ALU_result 00000001
...U/UUT/Mem_r_data ffffffff
...stance/OutputAddr 00000004
...stance/InputAddr 00000000
...derInstance/Offset 00000004
...mory/Instruction 35590009
...Memory/InputAddr 00000000
.../Reg_MUX1/RegDst 19
.../UUT/Reg_MUX1/Rt 19
...UUT/Reg_MUX1/Rd 00
..._MUX1/RegDst_ctrl 0


..._MUX1/RegDst_ctrl 0
...egister_File/RsData 0000000a
...egister_File/RtData 00000000
...egister_File/RsAddr 0a
...egister_File/RtAddr 19
...egister_File/RdAddr 19
...egister_File/RdData 00000001
...ister_File/RegWrite 1
...UT/Register_File/clk 1
...ontrol_Unit/RegDst 0
...trol_Unit/RegWrite 1
...ontrol_Unit/ALU_op 1
...ntrol_Unit/ALU_src 1
...ontrol_Unit/Mem_w 0
...ontrol_Unit/Mem_r 0
...Unit/Mem_to_Reg 0
...ontrol_Unit/opcode 0d
...UT/ALU_MUX/data 00000009
...UUT/ALU_MUX/src1 00000000
...UUT/ALU_MUX/src2 00000009
.../UUT/ALU_MUX/ctrl 1
..._Control_Unit/funct 0a
...trol_Unit/ALU_op 1
...trol_Unit/funct_ctrl 09



Cycle 1

```
39  def translate(machine_code, tables):
40      opcode = machine_code >> 26
41      if opcode == 0:
42          R_type(machine_code, tables[0])
43      elif opcode == 0b011100:
44          print("Jump to address {%d || PC[3
45      else:
46          I_type(machine_code, tables[1])
```

1. opcode != 0 && opcode != 0b011100 -> I type

```python
def I_type(machine_code, I_table, width=10):
    opcode = machine_code >> 26
    rs = (machine_code >> 21) & 0x1F
    rt = (machine_code >> 16) & 0x1F
    imm = machine_code & 0xFFFF

    text = f"{I_table.get(opcode, 'Unknown')}"
    print(f"{text:<{width}}", end="")
    text = f"op: {opcode}"
    print(f"{text:<{width}}", end="")
    text = f"rs: {rs}"
    print(f"{text:<{width}}", end="")
    text = f"rt: {rt}"
    print(f"{text:<{width}}", end="")
    text = f"imm: {imm}"
    print(f"{text:<{width}}", end="")
```

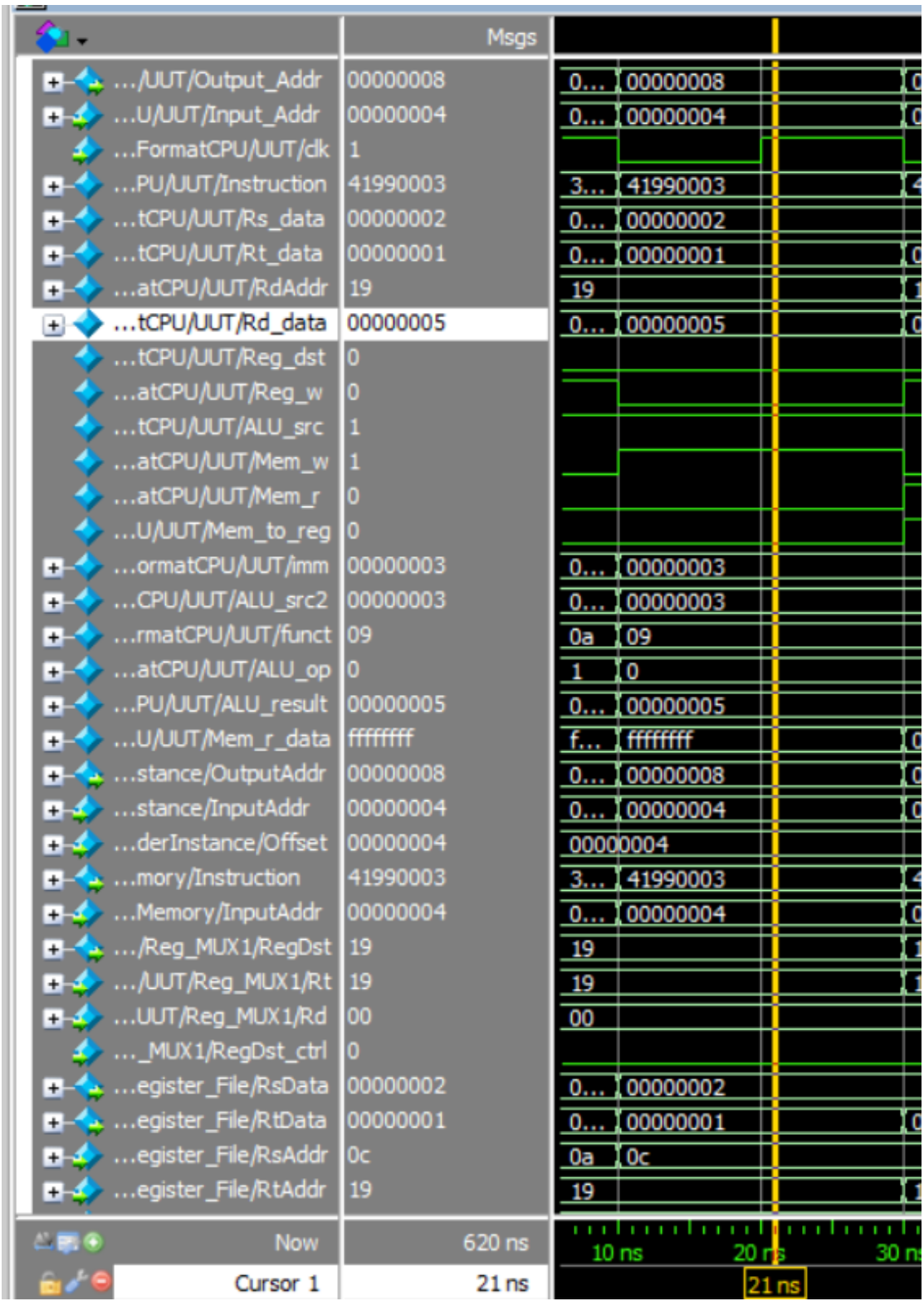
```python
Rtable = {0b001011: "ADDU", 0b001101: "SUBU", 0b100110: "SLL", 0b110110: "SLLV"}
Itable = {
    0b001101: "SUBI",
    0b010000: "SW",
    0b010001: "LW",
    0b101010: "SLTI",
    0b010011: "BEQ",
}
tables = [Rtable, Itable]
```

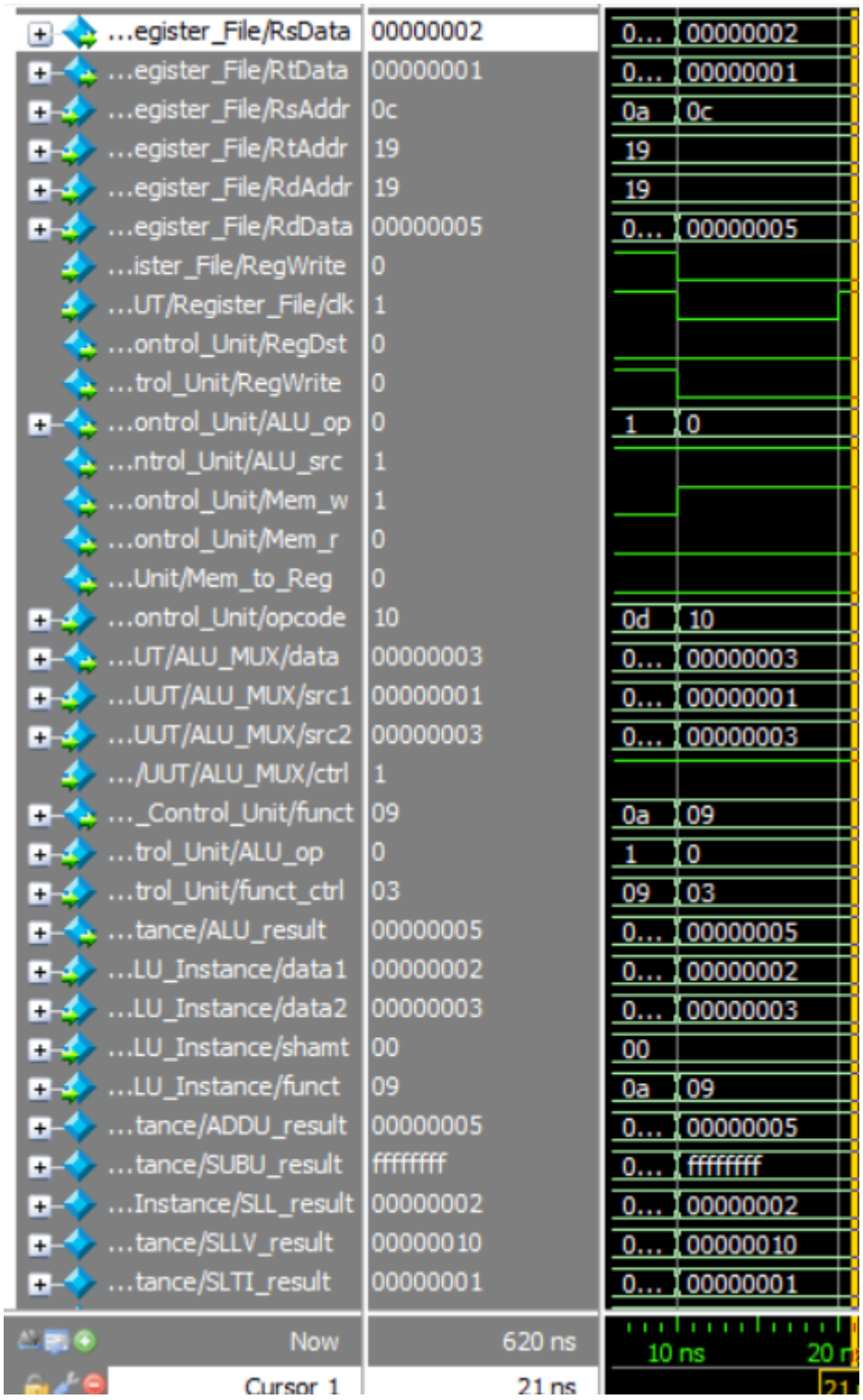
2. table[1] == Itable, find 0b001101 get SUBI; rs = 10; rd = 25; imm = 9.

Above is the flow of using translator.

- The instruction fetched is 0x35590009 which is SUBI $25, $10, 9. R[25]=R[10]-0x9=0xa-0x9=0x1.

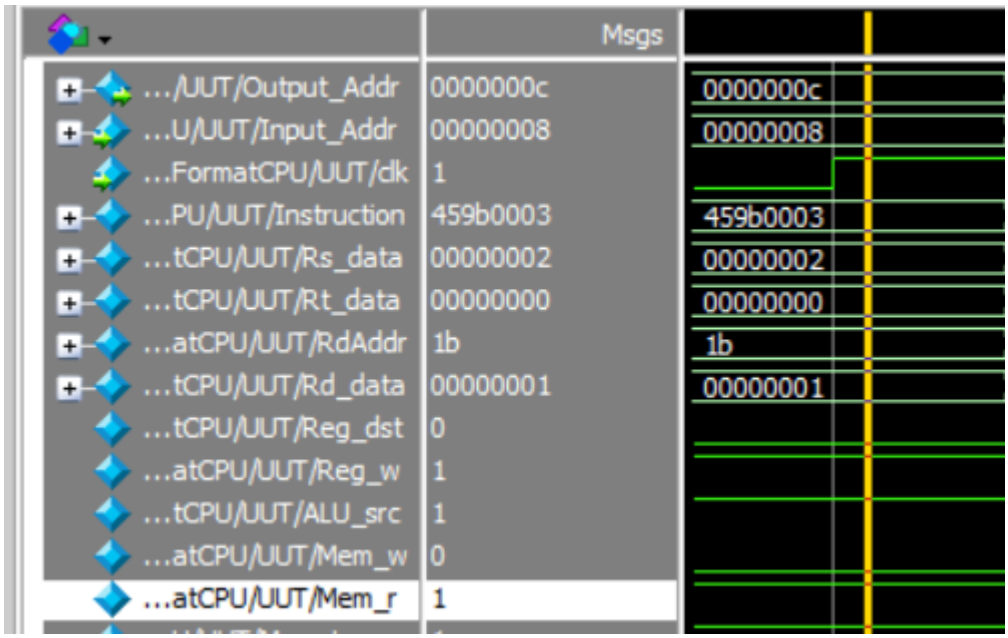
- Control now has more flag to control. In this case, Reg_w, ALU_src is set to ALU use imm as src2 and store the result back to R[RdAddr]
- UUT/funct = 0a to tell ALU to do subtraction.
- UUT/Mem_r_data is read from DM. Mem_r will decide it should pass the MUX or not.
- There are many MUXs. The mechanisms are the same as the text book.
- The remaining logic is the same as previous.


Msgs
.../UUT/Output_Addr 00000008
...U/UUT/Input_Addr 00000004
...FormatCPU/UUT/clk 1
...PU/UUT/Instruction 41990003
...tCPU/UUT/Rs_data 00000002
...tCPU/UUT/Rt_data 00000001
...atCPU/UUT/RdAddr 19
...tCPU/UUT/Rd_data 00000005
...tCPU/UUT/Reg_dst 0
...atCPU/UUT/Reg_w 0
...tCPU/UUT/ALU_src 1
...atCPU/UUT/Mem_w 1
...atCPU/UUT/Mem_r 0
...U/UUT/Mem_to_reg 0
...ormatCPU/UUT/imm 00000003
...CPU/UUT/ALU_src2 00000003
...rmatCPU/UUT/funct 09
...atCPU/UUT/ALU_op 0
...PU/UUT/ALU_result 00000005
...U/UUT/Mem_r_data ffffffff
...stance/OutputAddr 00000008
...stance/InputAddr 00000004
...derInstance/Offset 00000004
...mory/Instruction 41990003
...Memory/InputAddr 00000004
.../Reg_MUX1/RegDst 19
.../UUT/Reg_MUX1/Rt 19
...UUT/Reg_MUX1/Rd 00
..._MUX1/RegDst_ctrl 0
...egister_File/RsData 00000002
...egister_File/RtData 00000001
...egister_File/RsAddr 0c
...egister_File/RtAddr 19
Now 620 ns
Cursor 1 21 ns
10 ns
20 ns
30 ns
21 ns


...egister_File/RsData 00000002
...egister_File/RtData 00000001
...egister_File/RsAddr 0c
...egister_File/RtAddr 19
...egister_File/RdAddr 19
...egister_File/RdData 00000005
...ister_File/RegWrite 0
...UT/Register_File/clk 1
...ontrol_Unit/RegDst 0
...trol_Unit/RegWrite 0
...ontrol_Unit/ALU_op 0
...ntrol_Unit/ALU_src 1
...ontrol_Unit/Mem_w 1
...ontrol_Unit/Mem_r 0
...Unit/Mem_to_Reg 0
...ontrol_Unit/opcode 10
...UT/ALU_MUX/data 00000003
...UUT/ALU_MUX/src1 00000001
...UUT/ALU_MUX/src2 00000003
.../UUT/ALU_MUX/ctrl 1
..._Control_Unit/funct 09
...trol_Unit/ALU_op 0
...trol_Unit/funct_ctrl 03
...tance/ALU_result 00000005
...LU_Instance/data1 00000002
...LU_Instance/data2 00000003
...LU_Instance/shamt 00
...LU_Instance/funct 09
...tance/ADDU_result 00000005
...tance/SUBU_result ffffffff
...Instance/SLL_result 00000002
...tance/SLLV_result 00000010
...tance/SLTI_result 00000001
Now 620 ns
Cursor 1 21 ns
10 ns

Cycle 2

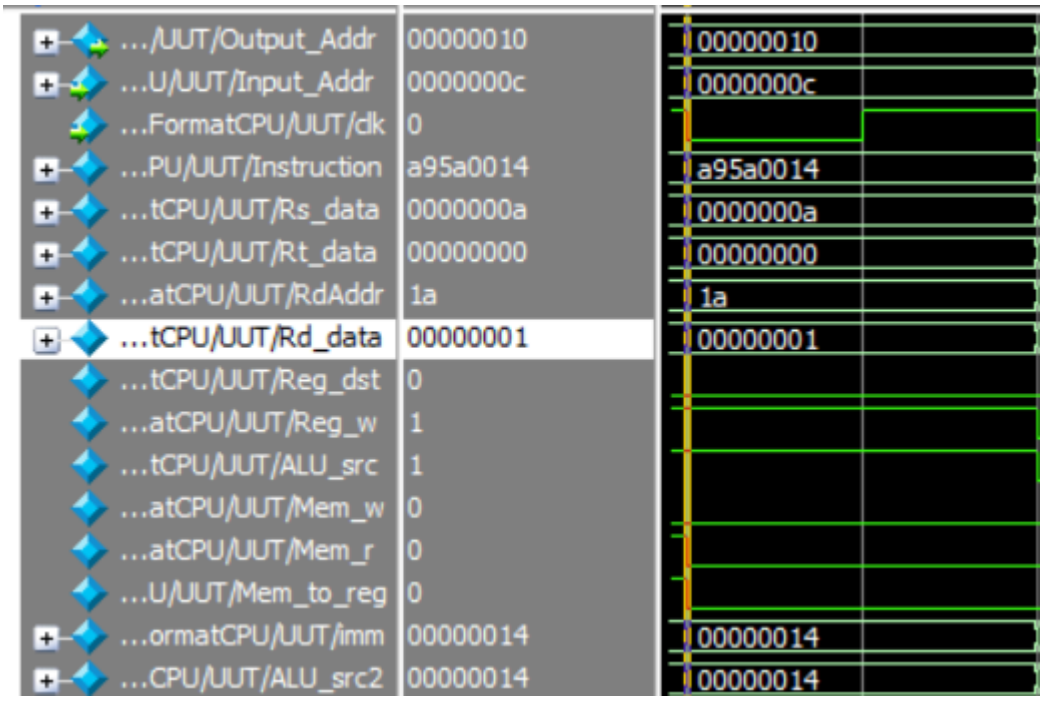
- SW op: 16 rs: 12 rt: 25 imm: 3 which is saving the previous Rd into memory R[12]+3=5. So the DM's {5,6,7,8} will be 0x00000001. How I know it's sw with these parms is using translator.
- The remaining logic is the same as previous.



Cycle 3

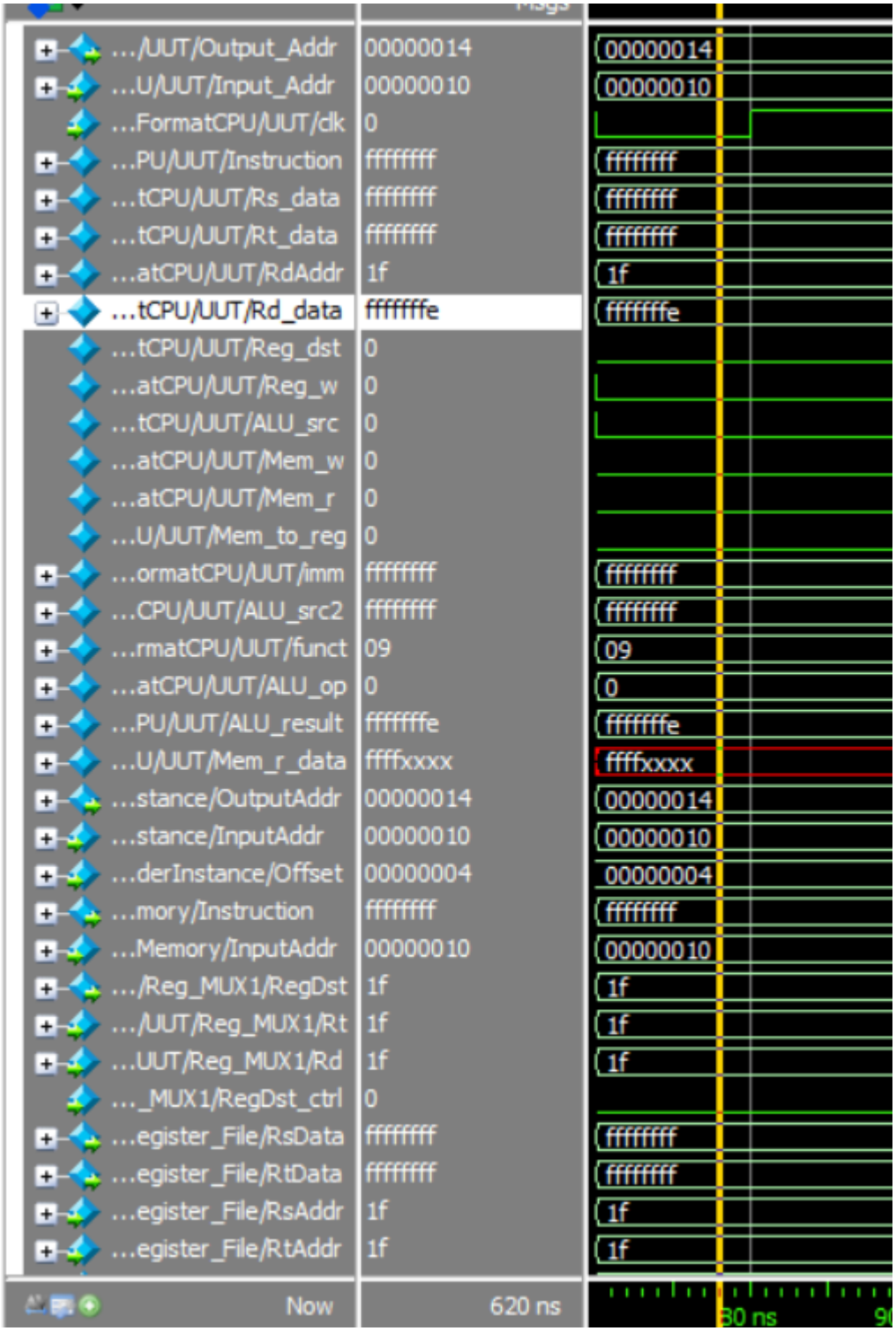
- LW op: 17 rs: 12 rt: 27 imm: 3 which is loading the previously saved data into R[27]. Mem addr is R[12]+3. From translator.

- The remaining part isn’t important.



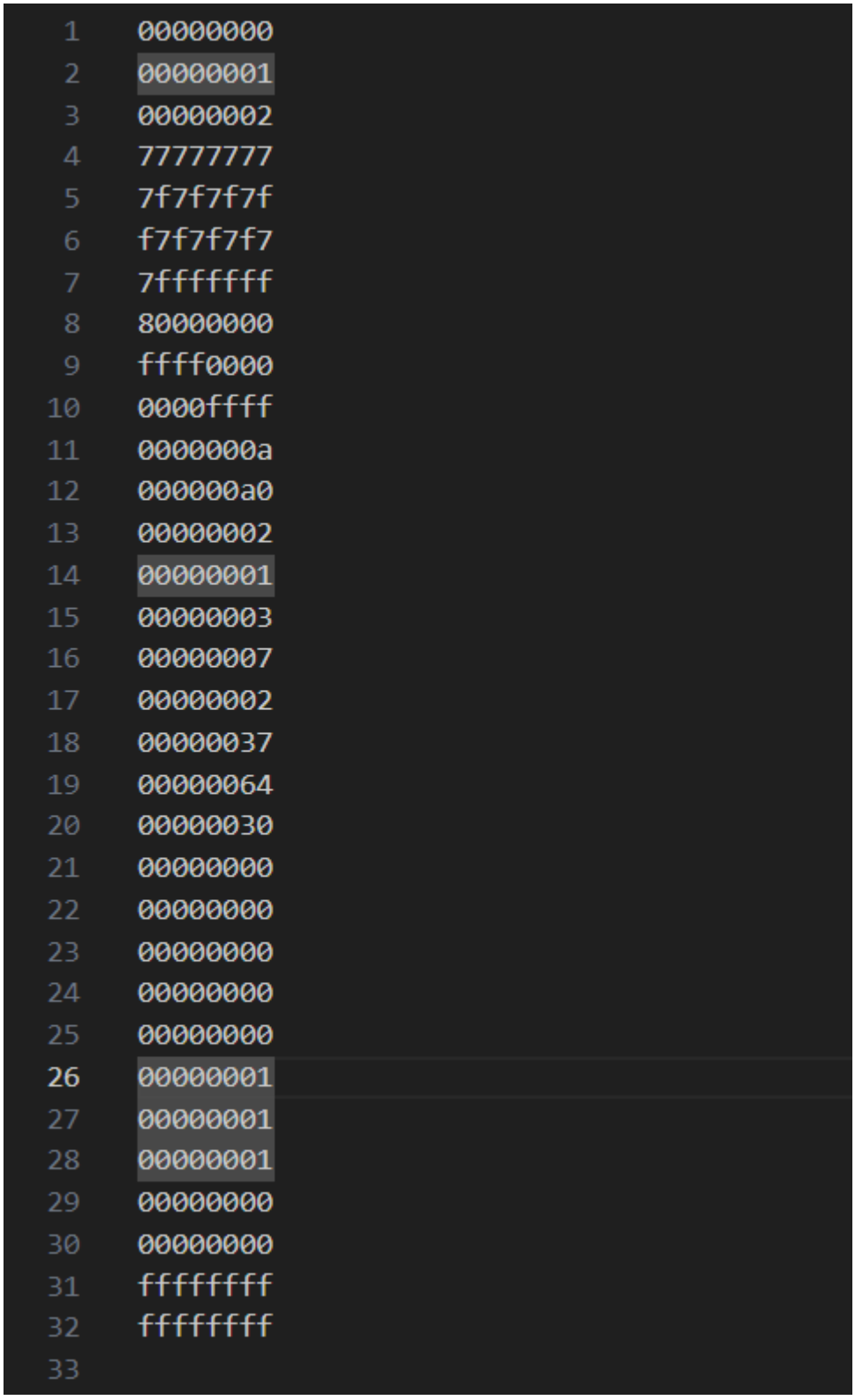
Cycle 4

- SLTI op: 42 rs: 10 rt: 26 imm: 20 which is if R[10]=10<20, R[26]=1 else R[26]=0.
- The remaining part isn’t important



This is the default condition. All the write flags are disabled to prevent modifying the data.

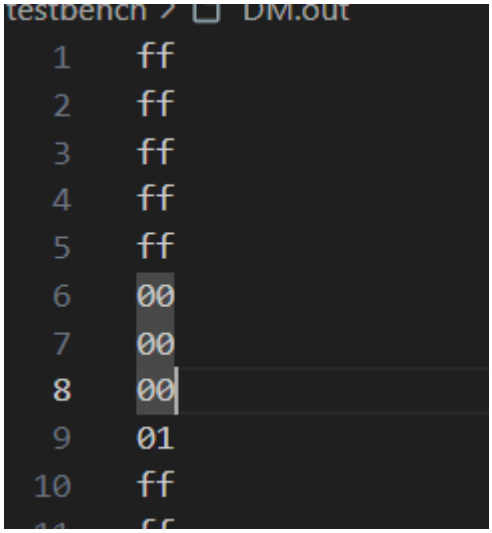
The corresponding output is below:



$25=1 by subiu

$27=1 by lw

$26=1 by slti



{5,6,7,8}(0 index) is set to 0x00000001 by sw

Part3

The test program is this using translator:

BEQ op: 19 rs: 0 rt: 19 imm: 30

SUBU rs: 19 rt: 2 rd: 19 shamt: 0

Jump to address {0 || PC[31:28]}

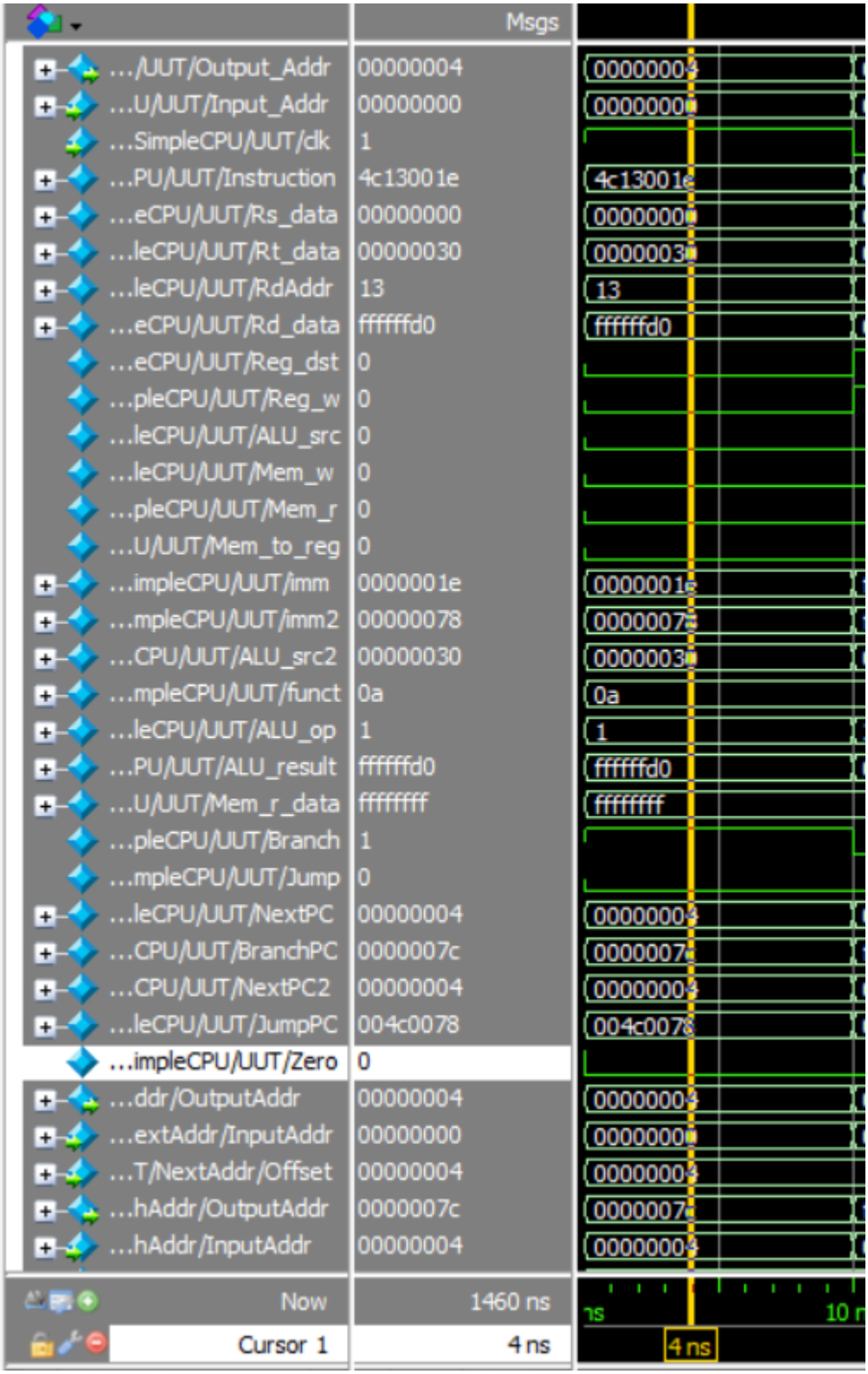
Which means

While (R[19] != R[0])

{

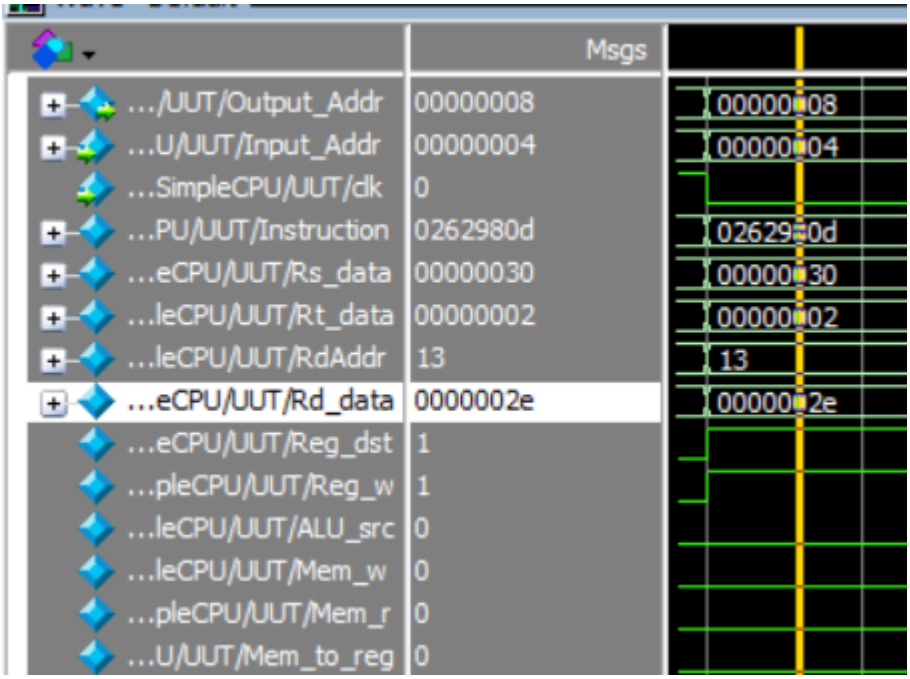
R[19] -= R[2]

}

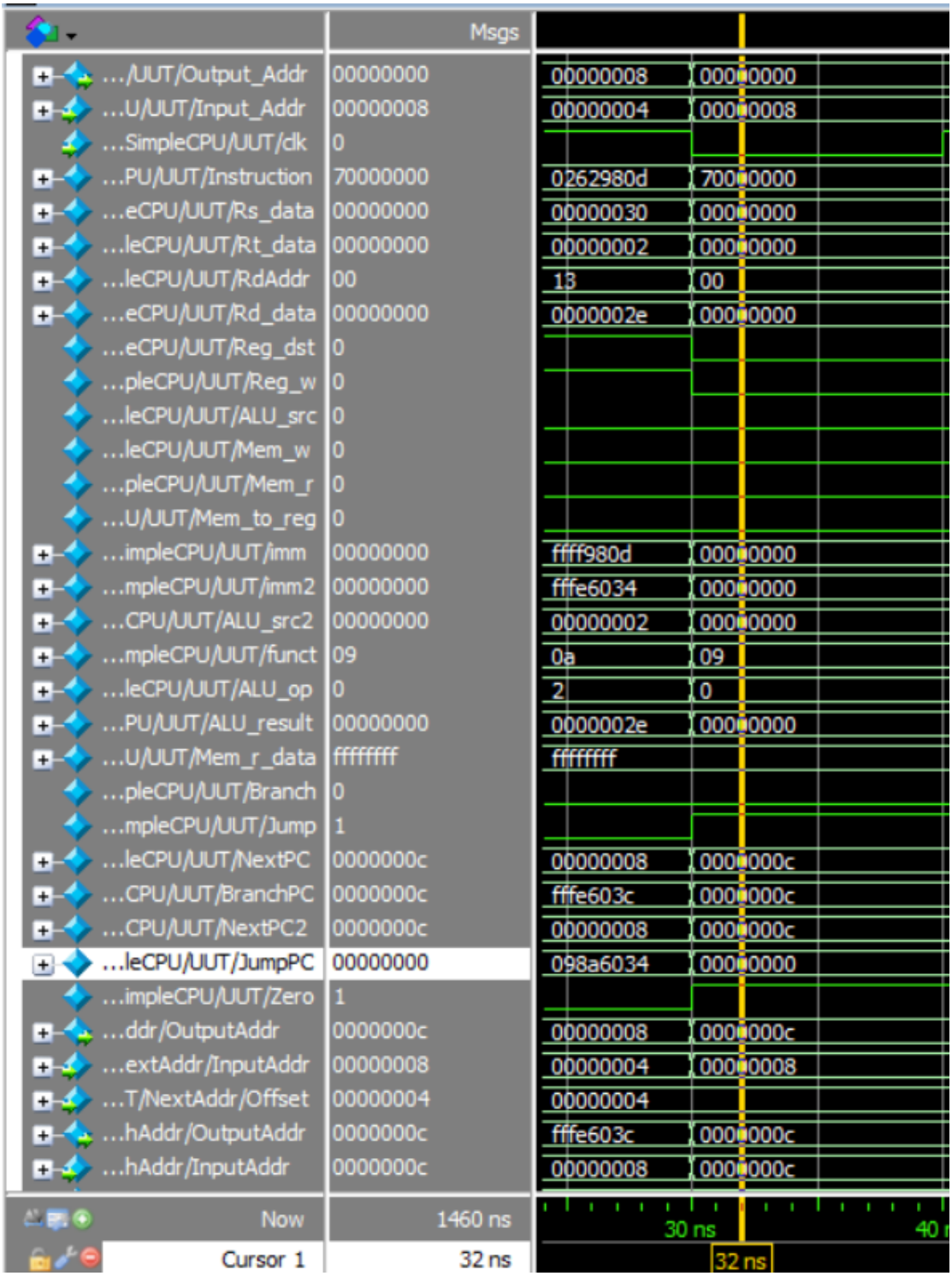


BEQ

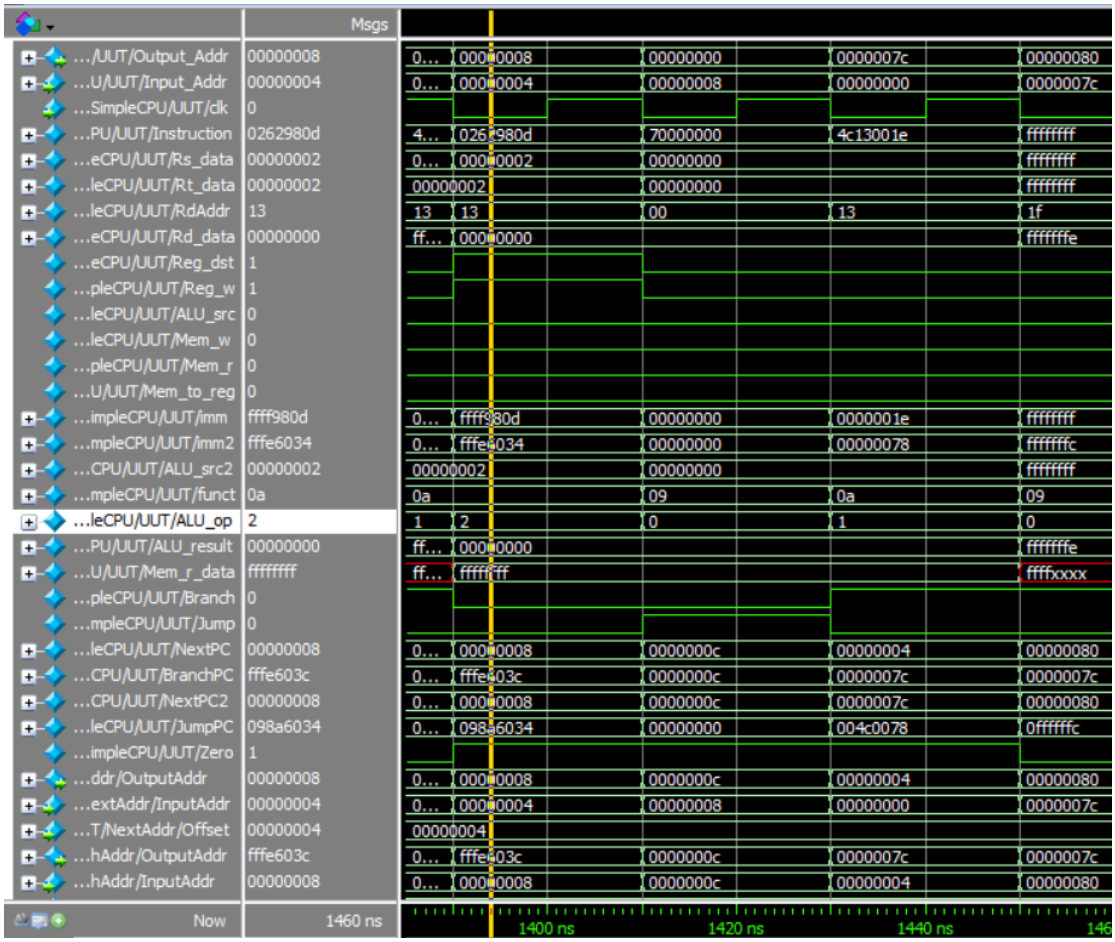
- Zero = 0, not branching



SUBU: 0x30-0x2 = 0x2e


Msgs
.../UUT/Output_Addr 00000000 00000008 0000000
...U/UUT/Input_Addr 00000008 00000004 0000008
...SimpleCPU/UUT/clk 0
...PU/UUT/Instruction 70000000 0262980d 7000000
...eCPU/UUT/Rs_data 00000000 00000030 0000000
...leCPU/UUT/Rt_data 00000000 00000002 0000000
...leCPU/UUT/RdAddr 00 13 00
...eCPU/UUT/Rd_data 00000000 0000002e 0000000
...eCPU/UUT/Reg_dst 0
...pleCPU/UUT/Reg_w 0
...leCPU/UUT/ALU_src 0
...leCPU/UUT/Mem_w 0
...pleCPU/UUT/Mem_r 0
...U/UUT/Mem_to_reg 0
...impleCPU/UUT/imm 00000000 ffff980d 0000000
...mpleCPU/UUT/imm2 00000000 fffe6034 0000000
...CPU/UUT/ALU_src2 00000000 00000002 0000000
...mpleCPU/UUT/funct 09 0a 09
...leCPU/UUT/ALU_op 0 2 0
...PU/UUT/ALU_result 00000000 0000002e 0000000
...U/UUT/Mem_r_data ffffffff ffffffff
...pleCPU/UUT/Branch 0
...mpleCPU/UUT/Jump 1
...leCPU/UUT/NextPC 0000000c 00000008 000000c
...CPU/UUT/BranchPC 0000000c fffe603c 000000c
...CPU/UUT/NextPC2 0000000c 00000008 000000c
...leCPU/UUT/JumpPC 00000000 098a6034 0000000
...impleCPU/UUT/Zero 1
...ddr/OutputAddr 0000000c 00000008 000000c
...extAddr/InputAddr 00000008 00000004 0000008
...T/NextAddr/Offset 00000004 00000004
...hAddr/OutputAddr 0000000c fffe603c 000000c
...hAddr/InputAddr 0000000c 00000008 000000c
Now 1460 ns
Cursor 1 32 ns
30 ns
32 ns

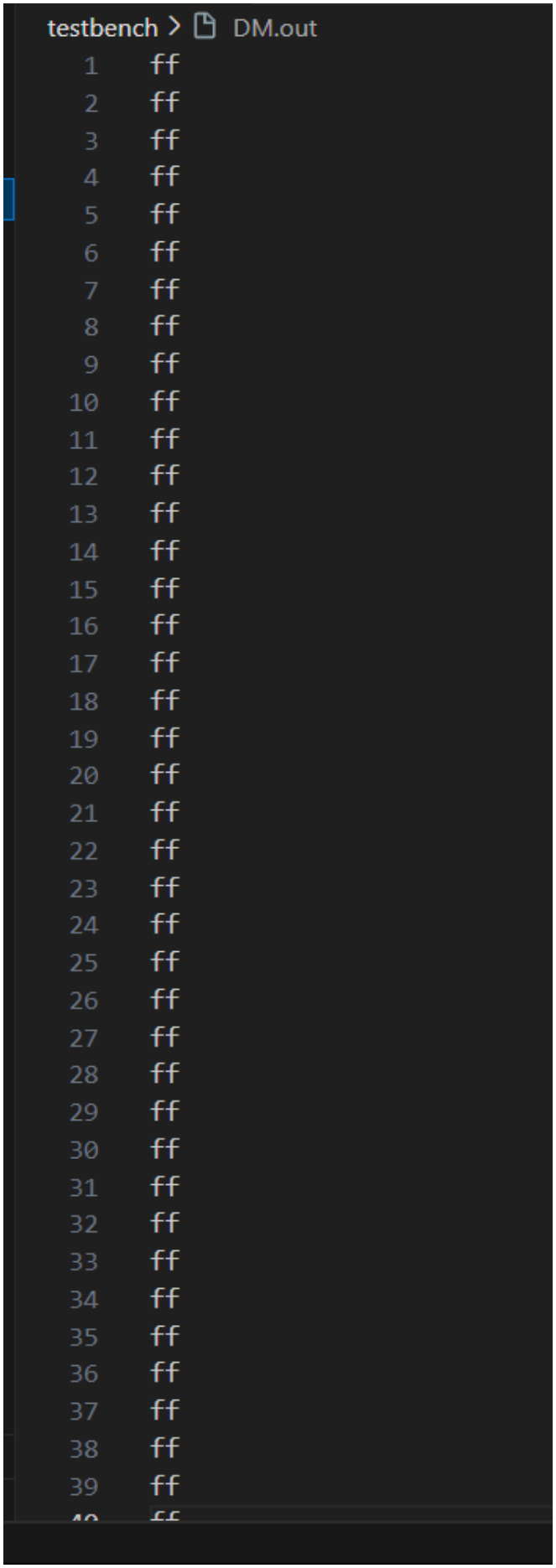
Jump to PC = 0



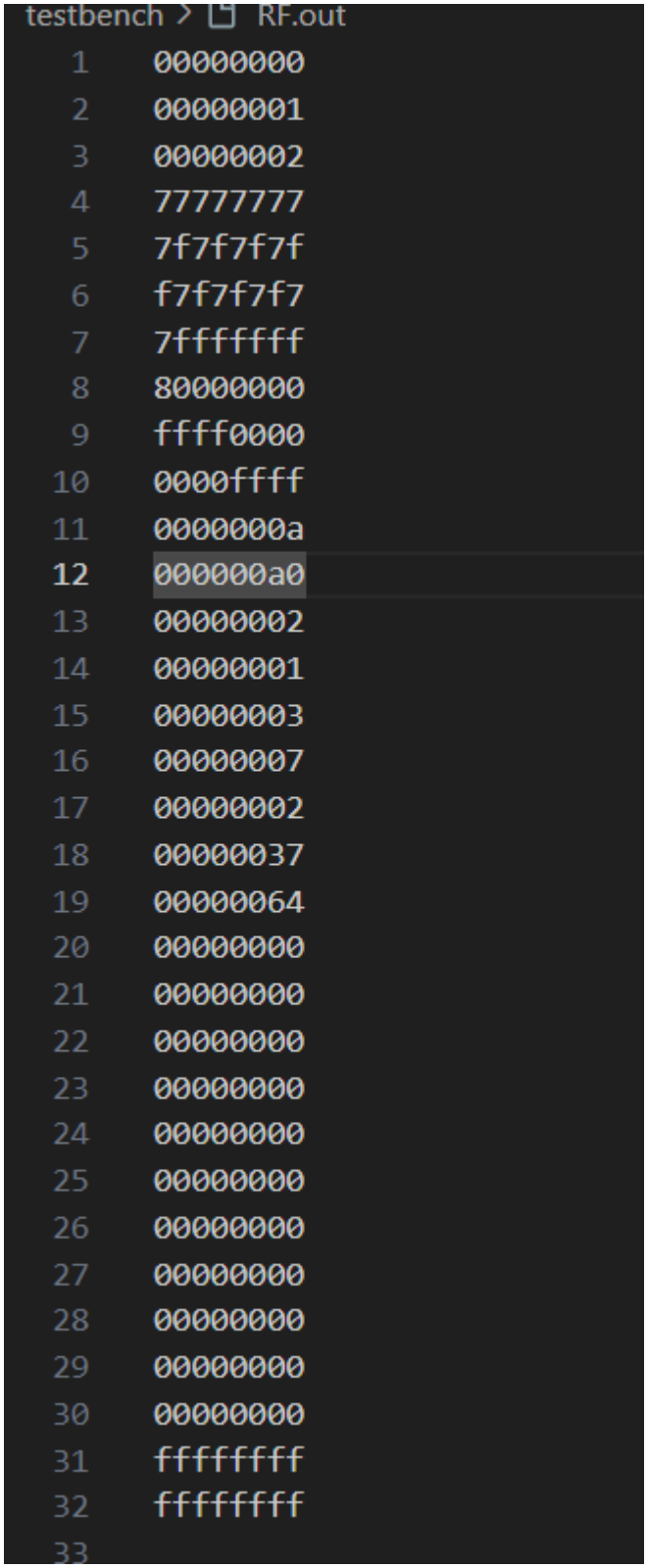
This the last iteration.

1. 2-2=0
2. Jump to 0
3. 0==0, branch to 30*4+4=124 which is the last position of IM

The cycles passed: ((0x30/2) times * 3 clk/time + 1) * 20 ns/clk = 1460 ns. The data between 1450~1460 is the next cycles' data, so the commands here are executed correctly.


testbench > DM.out
1 ff
2 ff
3 ff
4 ff
5 ff
6 ff
7 ff
8 ff
9 ff
10 ff
11 ff
12 ff
13 ff
14 ff
15 ff
16 ff
17 ff
18 ff
19 ff
20 ff
21 ff
22 ff
23 ff
24 ff
25 ff
26 ff
27 ff
28 ff
29 ff
30 ff
31 ff
32 ff
33 ff
34 ff
35 ff
36 ff
37 ff
38 ff
39 ff

All 0xff in DM.out



RF.out is just like using “cat RF.dat > RF.out”.

Custom Test Program

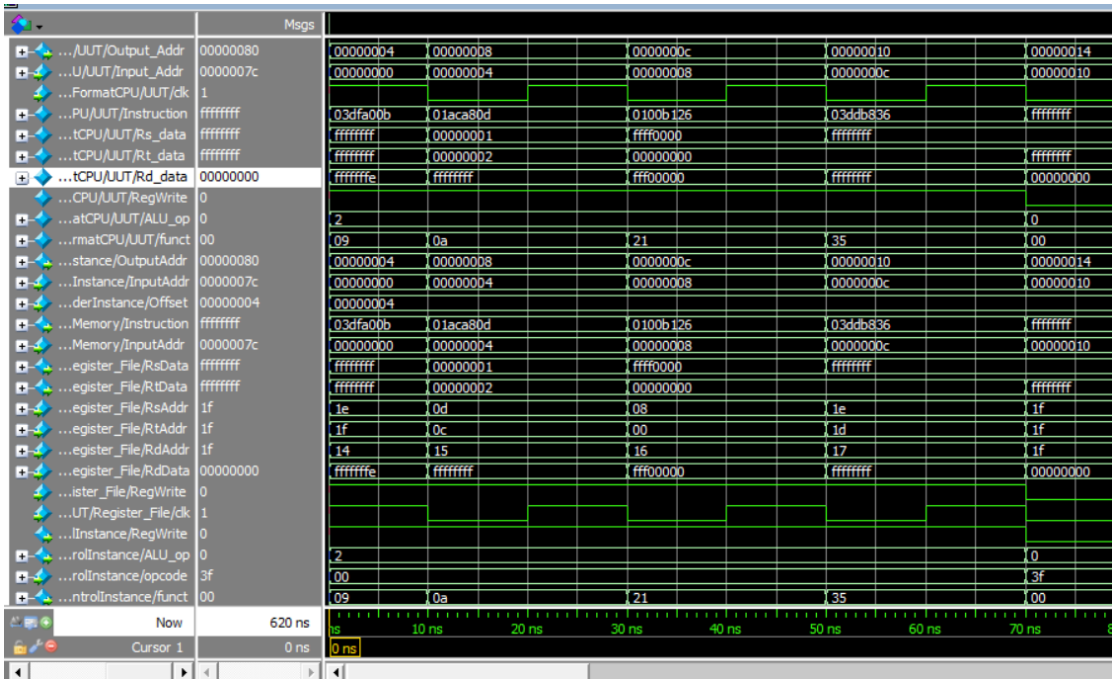
Part1

testbench > program.txt

```
1   ADDU $s4, $fp, $ra
2   SUBU $s5, $t5, $t4
3   SLL $s6, $t0, 4
4   SLLV $s7, $fp, $sp
```

```
(base) D:\Programming\Verilog\PA2\Part1>C:/Users/abc22/anac
0: 03dfa00b
1: 01aca80d
2: 0100b126
3: 03ddb836
Labels: {}

(base) D:\Programming\Verilog\PA2\Part1>C:/Users/abc22/anac
ADDU      rs: 30    rt: 31    rd: 20    shamt: 0
SUBU      rs: 13    rt: 12    rd: 21    shamt: 0
SLL       rs: 8     rt: 0     rd: 22    shamt: 4
SLLV      rs: 30    rt: 29    rd: 23    shamt: 0
Unknown   op: 63    rs: 31    rt: 31    imm: 65535
```

```
20  00000030
21  fffffffe
22  ffffffff
23  fff00000
24  ffffffff
25  00000000
```



R[20]=R[30]+R[31]=0xFFFF_FFFF+0xFFFF_FFFF=0xFFFF_FFFE. This is test for overflow.
R[21]=R[13]-R[12]=0x0000_0001-0x0000_0002=0xFFFF_FFFF. Test for negative result.
R[22]=R[8]<<4=0xFFFF_0000<<4=0XFFF0_0000. Test for overflow and functionality.
R[23]=R[30]<<R[29][4:0]=0xFFFF_FFFF<<0b00000=0xFFFF_FFFF. Test for shift amount = 0.

Part2

testbench > program.txt

```
1    SUBI $s4, $zero, -20
2    SW $s4, 0($s4)
3    LW $s5, 0($s4)
4    SLTI $s6, $s5, 0
```

```
0: 3414ffec
1: 42940000
2: 46950000
3: aab60000
Labels: {}

(base) D:\Programming\Verilog\PA2\Part2>C:/Users/abc22/a
SUBI      op: 13    rs: 0     rt: 20    imm: 65516
SW        op: 16    rs: 20    rt: 20    imm: 0
LW        op: 17    rs: 20    rt: 21    imm: 0
SLTI      op: 42    rs: 21    rt: 22    imm: 0
```

```
17  00000002
18  00000037
19  00000064
20  00000030
21  00000014
22  00000014
23  00000000
24  00000000
```

```
20  ff
21  00
22  00
23  00
24  14
25  ff
```

R[20]=R[0]-(-20)=0x0000_0014. Test for minus negative number.
Mem[R[20]+0]=R[20]=0x0000_0014. Save 20 to position {20, 21, 22, 23}
R[21]=Mem[R[20]+0]=0x0000_0014. Fetch 20 from position {20, 21, 22, 23} and save to R[21]. Test for memory save/load functionality.

Part3

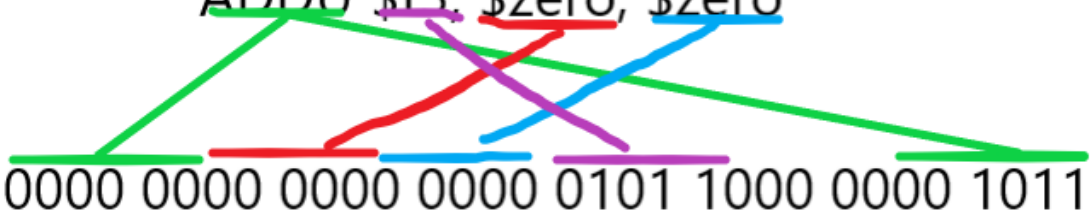
Fibonacci sequence converted by convertor

```
1    ADDU $t0, $zero, $zero
2    SUBI $t1, $zero, -1
3    ADDU $t3, $zero, $zero
4    SUBI $t4, $zero, -10
5    loop:ADDU $t2, $t0, $t1
6    SW $t0, 0($t3)
7    ADDU $t0, $t1, $zero
8    ADDU $t1, $t2, $zero
9    SUBI $t3, $t3, -4
10   SUBI $t4, $t4, 1
11   BEQ $t4, $zero, end
12   J loop
13   end:ADDU $v0, $zero, $zero
```

-> 0x0000400b

-> 0x3409ffff

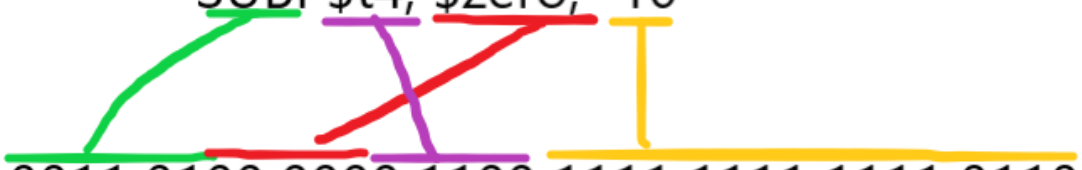
ADDU $t3, $zero, $zero



0000 0000 0000 0000 0101 1000 0000 1011

-> 0x000580b

SUBI $t4, $zero, -10



0011 0100 0000 1100 1111 1111 1111 0110

-> 0x340cfff6

loop:ADDU $t2, $t0, $t1

0000 0001 0000 1001 0101 0000 0000 1011

-> 0x0109500b

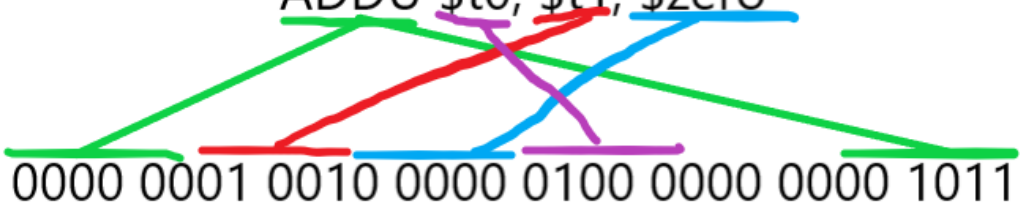
loop = 4

SW $t0, 0($t3)

0100 0001 0110 1000 0000 0000 0000 0000

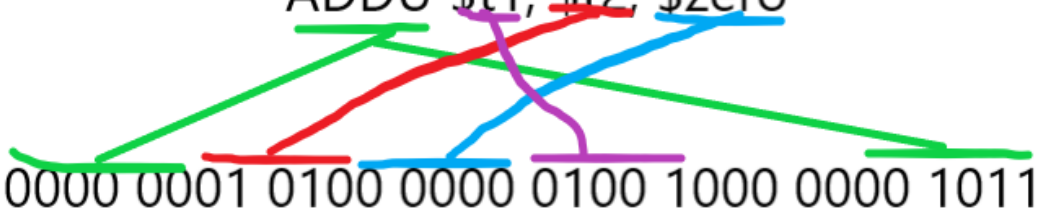
-> 0x41680000

ADDU $t0, $t1, $zero



-> 0x0120400b

ADDU $t1, $t2, $zero



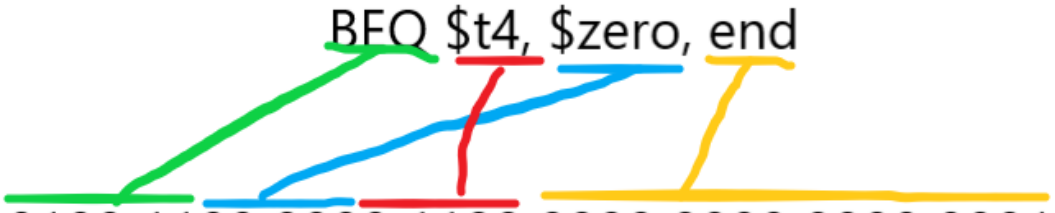
-> 0x0140480b

SUBI $t3, $t3, -4

-> 0x356bfffc

SUBI $t4, $t4, 1

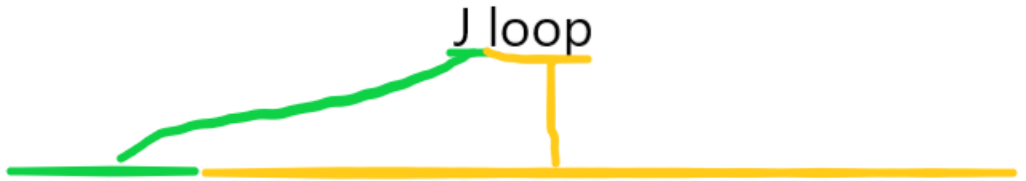
-> 0x358c0001



-> 0x4c0c0001

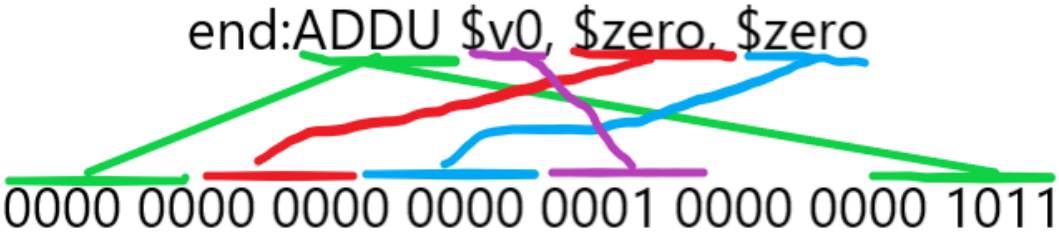
end = 1 because end-current-1=12-10-1=1

I converted wrong here, but it's not important. There is no difference between $t4==$zero and $zero==$t4. The circuit is still using rs_data and rt_data to judge.



-> 0x70000004

loop = 4 from previous result



-> 0x0000100b

end = 12. this is the reason why end in line 10(BEQ) will be 12.

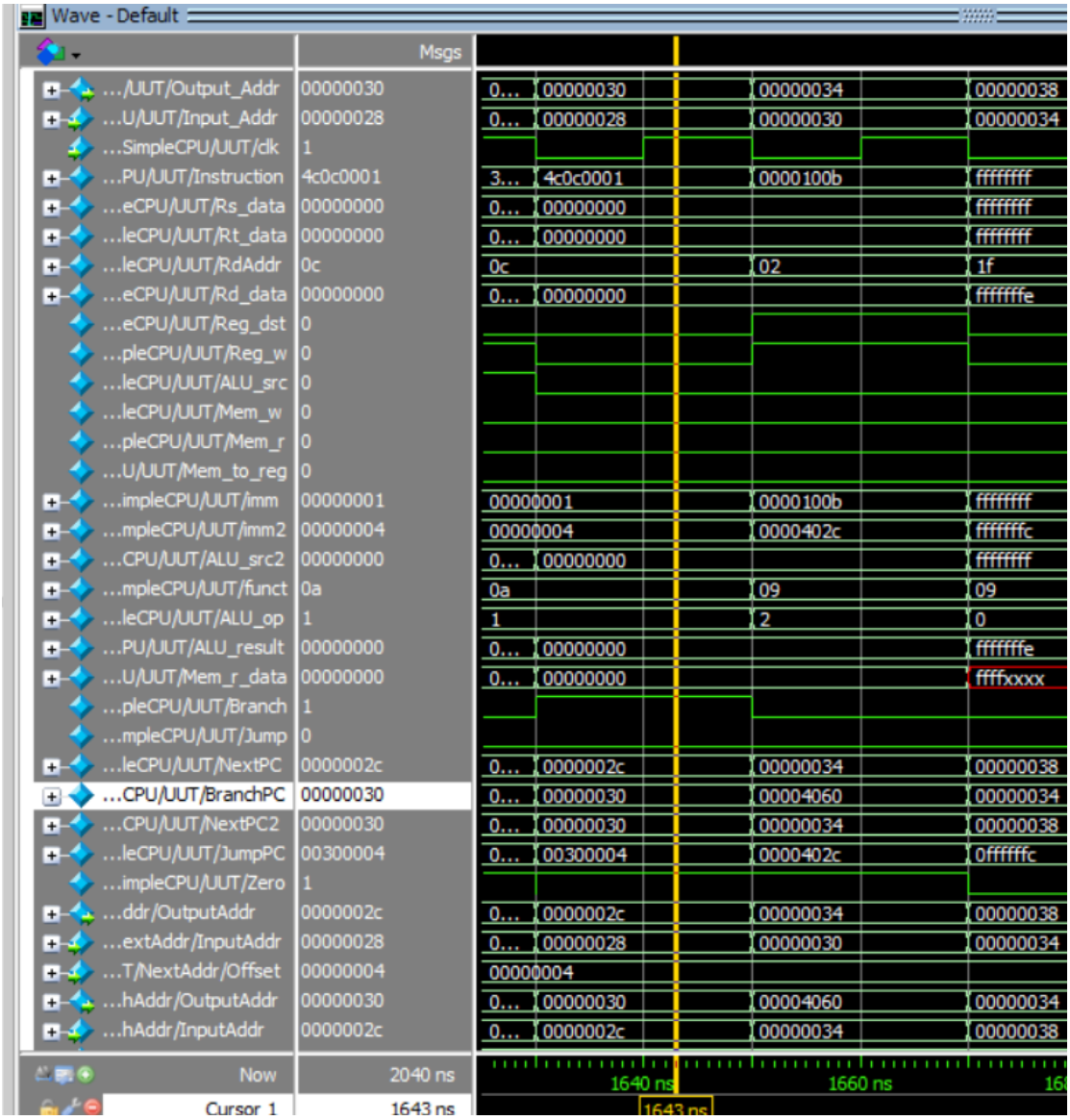
```
0: 0000400b
1: 3409ffff
2: 0000580b
3: 340cfff6
4: 0109500b
5: 41680000
6: 0120400b
7: 0140480b
8: 356bfffc
9: 358c0001
10: 4c0c0001
11: 70000004
12: 0000100b
Labels: {'loop': 4, 'end': 12}
```

```
ADDU      rs: 0     rt: 0     rd: 8     shamt: 0
SUBI      op: 13    rs: 0     rt: 9     imm: 65535
ADDU      rs: 0     rt: 0     rd: 11    shamt: 0
SUBI      op: 13    rs: 0     rt: 12    imm: 65526
ADDU      rs: 8     rt: 9     rd: 10    shamt: 0
SW        op: 16    rs: 11    rt: 8     imm: 0
ADDU      rs: 9     rt: 0     rd: 8     shamt: 0
ADDU      rs: 10    rt: 0     rd: 9     shamt: 0
SUBI      op: 13    rs: 11    rt: 11    imm: 65532
SUBI      op: 13    rs: 12    rt: 12    imm: 1
BEQ       op: 19    rs: 0     rt: 12    imm: 1
Jump to address {4 || PC[31:28]}
ADDU      rs: 0     rt: 0     rd: 2     shamt: 0
63        op: 63    rs: 31    rt: 31    imm: 65535
```

```
00
00
40
0b
34
09
ff
ff
00
00
58
0b
34
0c
ff
f6
01
09
50
0b
41
68
00
00
01
20
40
0b
01
40
48
0b
35
6b
ff
fc
```

```
35
8c
00
01
4c
0c
00
01
70
00
00
04
00
00
10
0b
FF
FF
FF
FF
```

Big endian



The functionality is tested previously. Here I skipped the detail. The correctness of my design can be proven by the result of this program.

Lines in 0~3 are initialization. It set $t0 to 0; $t1 to 1; $t3 to 0; $t4 to 10.

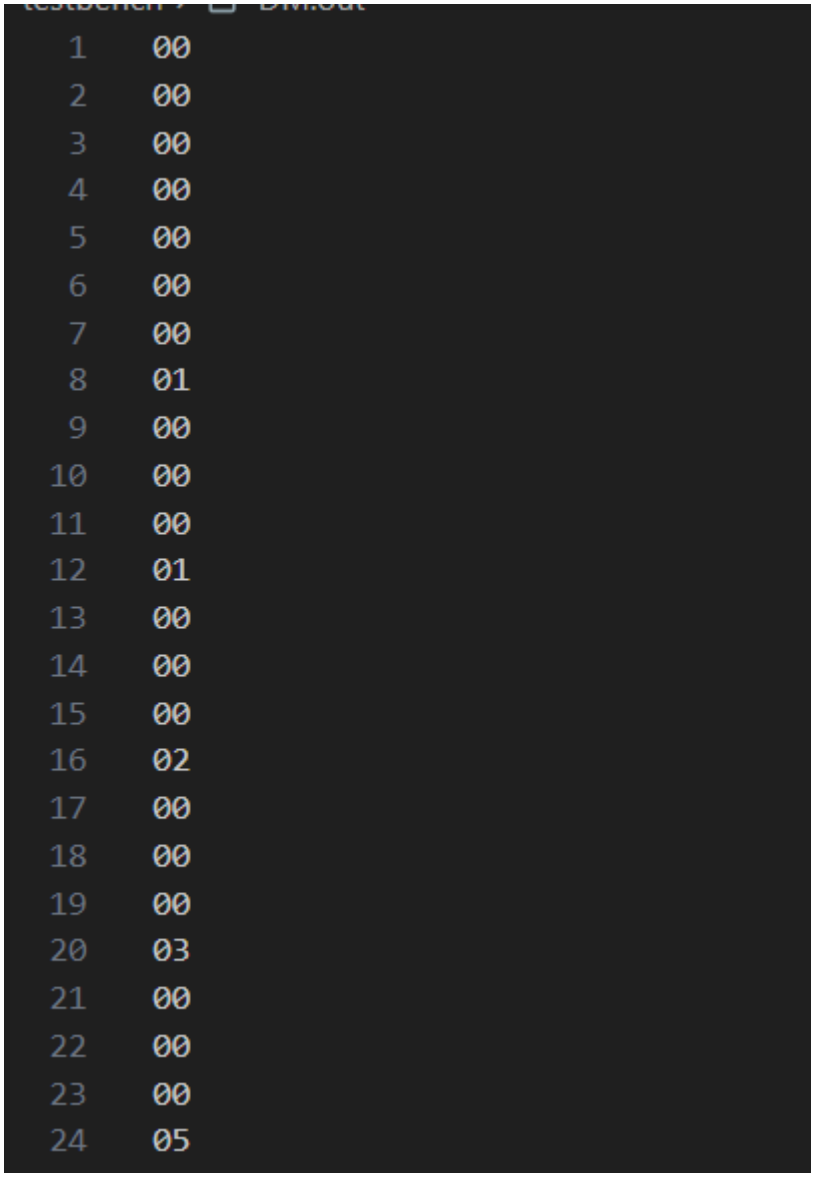
For 1 iteration, the program will do {

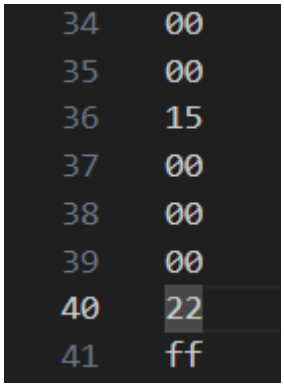
```
ADDU $t2, $t0, $t1        // compute the next Fibonacci num
SW $t0, 0($t3)            // save the first register
ADDU $t0, $t1, $zero      // set the first's value to the second's
ADDU $t1, $t2, $zero      // set the second' value to the next num
SUBI $t3, $t3, -4         // add 4 to the base address
SUBI $t4, $t4, 1          // counter--
```

}

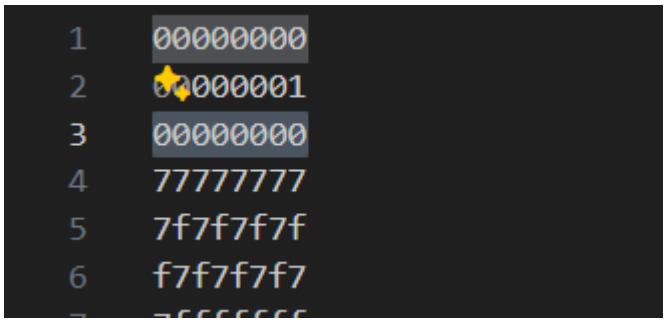
After finish 10 iteration, PC branches to <end> which is +8 bytes from current, to IM = {48, 49, 50, 51} to perform ADDU $v0, $zero, $zero.

Time: (4 setup +8 instr/iter * 10 iter – 1 the jump is not perform in the last iter) * 20ns = 1660ns. The last one cycle is ADDU $v0, $zero, $zero.





[00000000, 00000001, 00000001, 00000002, 00000003, …, 00000022] which is exactly 10 Fibonacci sequence numbers.



And the R[2] is set to 0 as well. This program works.

Conclusion and Insights

By implementing the whole MIPS single cycle CPU, I can say that I'm already fully understand how this structure works.

By observing the testbench, I find that machine code has 0 readability, so I write a python script to convert it to MIPS instruction with important info printed to cmd. And the report also requires custom program, so I make a simple compiler to compile the program which can use label for BEQ, J to branch.

Because of the last PA I've got -12 point in the report, I write as detailed as possible this time. And for each module, I follow the same format whether it has that functionality or not.