VLSI Circuit Design II– EHB 425E

HOMEWORK VIII

Yiğit Bektaş GÜRSOY

040180063

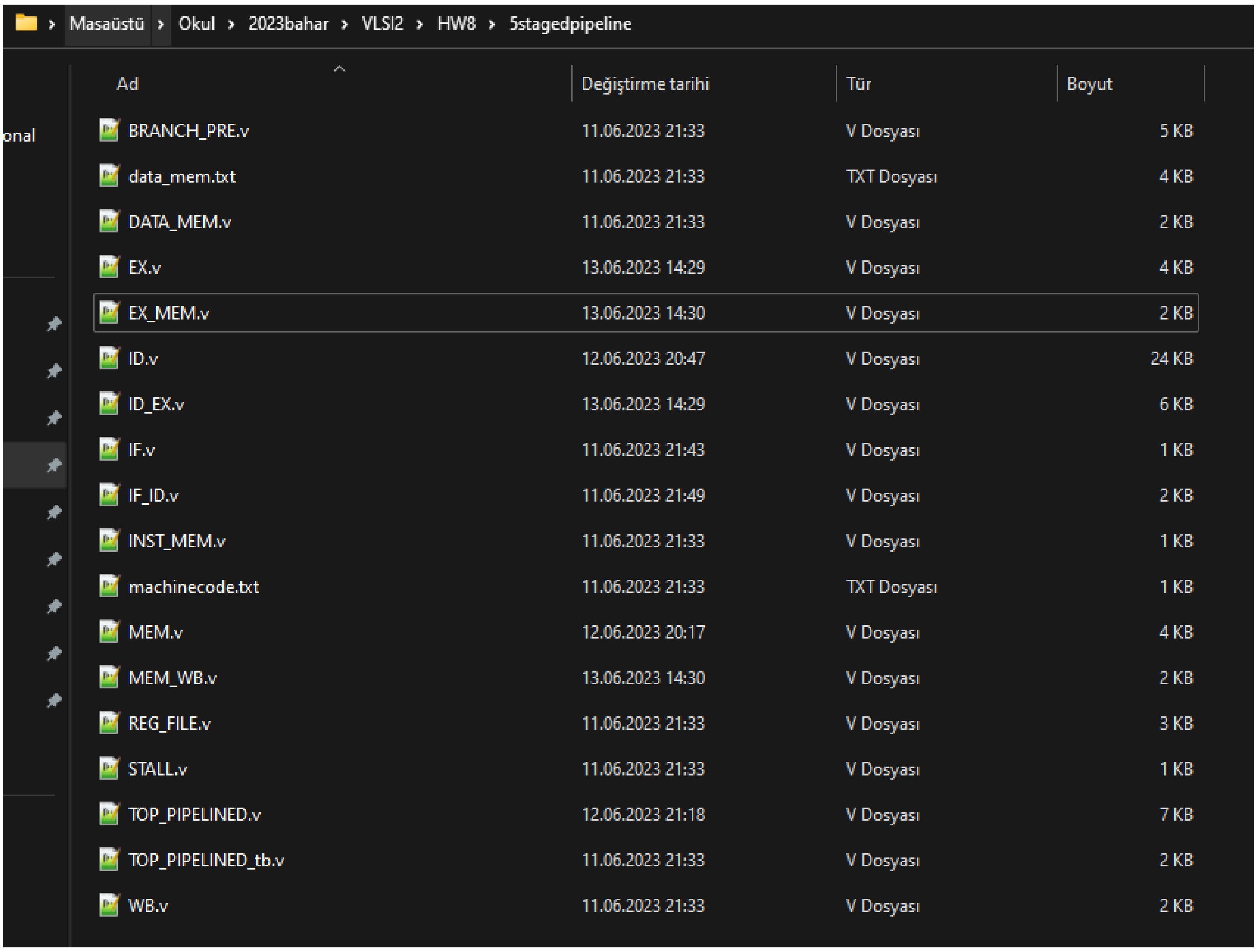
Rana TİLKİ

040180741

Class Lecturer: Sıddıka Berna Örs Yalçın

Class Assistant:
Yasin Fırat Kula

1. 5-STAGE PIPELINED RISC-V



Masaüstü › Okul › 2023bahar › VLSI2 › HW8 › 5stagedpipeline

Ad	Değiştirme tarihi	Tür	Boyut
BRANCH_PRE.v	11.06.2023 21:33	V Dosyası	5 KB
data_mem.txt	11.06.2023 21:33	TXT Dosyası	4 KB
DATA_MEM.v	11.06.2023 21:33	V Dosyası	2 KB
EX.v	13.06.2023 14:29	V Dosyası	4 KB
EX_MEM.v	13.06.2023 14:30	V Dosyası	2 KB
ID.v	12.06.2023 20:47	V Dosyası	24 KB
ID_EX.v	13.06.2023 14:29	V Dosyası	6 KB
IF.v	11.06.2023 21:43	V Dosyası	1 KB
IF_ID.v	11.06.2023 21:49	V Dosyası	2 KB
INST_MEM.v	11.06.2023 21:33	V Dosyası	1 KB
machinecode.txt	11.06.2023 21:33	TXT Dosyası	1 KB
MEM.v	12.06.2023 20:17	V Dosyası	4 KB
MEM_WB.v	13.06.2023 14:30	V Dosyası	2 KB
REG_FILE.v	11.06.2023 21:33	V Dosyası	3 KB
STALL.v	11.06.2023 21:33	V Dosyası	1 KB
TOP_PIPELINED.v	12.06.2023 21:18	V Dosyası	7 KB
TOP_PIPELINED_tb.v	11.06.2023 21:33	V Dosyası	2 KB
WB.v	11.06.2023 21:33	V Dosyası	2 KB

Figure 1: 5 Staged Pipeline’s source files

Above are the 5stage pipelined source files. Detailed description of the files is available below.

1- **IF.v:** This verilog module represents the "Instruction Fetch" (IF) stage for a RISCV processor. This stage pulls instructions from the program memory inside the processor and is responsible for forwarding them to the next stage. This module performs the instruction pull step of the RISCV processor and uses various control signals to determine the address of the next instruction.

2- **IF_ID.v:** This verilog module provides data communication between the "Instruction Fetch" (IF) and "Instruction Decode" (ID) stages for a RISCV processor. In these stages, the address and content of the instruction are transmitted to each other. This module performs data communication between the IF and ID stages. The program counter and instruction content from the IF stage are transmitted to the ID stage. In addition, the prediction signal from the ID stage is also transferred to the output. The module controls various signals depending on the given pause states and assigns appropriate values to its outputs.

3- **ID.v:** The Verilog module, with the filename "ID.v", performs the "Instruction Decode" (ID) stage of a 5-step RISC-V processor pipeline. The ID stage interprets the incoming instruction, generates the necessary control signals, and organizes the input data accordingly. This module represents the second stage of the 5-stage pipeline in RISC-V processors. In the ID stage, processing of the received instruction, editing of the input data and generation of various control signals are performed. Depending on the type of instruction, it determines which registers to read, which operation to take, whether to write or not.

4- **ID_EX.v:** The file "ID_EX.v" represents the ID/EX (Instruction Decode/Execute) pipeline stage on a 5-stage RISC-V processor. The purpose of this module is to transfer relevant information from the ID stage to the EX stage. The module receives inputs such as clock signal, reset signal, pause signals, ALU opcode, register values, write information, link address, and instruction. ALU opcode, connection address, instruction, register values, write information, etc. for the EX stage. Provides outputs such as In general, the ID_EX module enables the transfer of necessary information from the ID stage to the EX stage in the 5-stage RISC-V pipeline and enables the execution of instructions in successive pipeline stages.

5- **EX.v:** The "EX.v" file represents the EX (Execute) pipeline stage on a 5-stage RISC-V processor. This module is used to transfer relevant information from the ID/EX stage to the EX/MEM stage and to perform various operations. The module receives inputs such as reset signal, ALU opcode, operands, write data number, write control signal, link address, and instruction. The ALU opcode for the EX stage is assigned to the ALUop_i input, while the operands and other information are calculated based on the corresponding inputs. Generally, the EX module represents the EX stage in the 5-stage RISC-V pipeline. At this stage, various operations are performed on the operands depending on the ALU opcode and the result is assigned to the "WriteData_o" signal. Also, some other output signals are connected to the corresponding inputs or calculations. This ensures the execution of instructions and the preparation of relevant data.

6- **EX_MEM.v:** The file "EX_MEM.v" represents the EX/MEM (Execute/Memory) pipeline stage on a 5-stage RISC-V processor. This module is used to transfer relevant information from the EX stage to the MEM stage and to perform various operations. The module receives inputs such as reset signal, pause signal, write data number, write control signal, write data, ALU opcode, address and register. Information from the EX stage is used in the MEM stage based on these inputs. Generally, the EX_MEM module represents the EX/MEM stage in the 5-stage RISC-V pipeline. At this stage, the data from the EX stage is transmitted to the MEM stage and the relevant signals are transmitted. In this way, it is ensured that the information required for memory operations and write operations is transferred and prepared.

7- **MEM.v:** The "MEM.v" file represents the MEM (Memory) pipeline stage in a 5-stage RISC-V processor. This module takes care of memory access and data write operations in the processor. The module receives inputs such as reset signal, write control signal, write data address, ALU opcode, write data, memory address, register data, and memory data. Based on these inputs, it performs the operations that need to be done in the MEM phase and generates various signals. Generally, the MEM module represents the MEM stage in the 5-stage RISC-V pipeline. At this stage, memory accesses and data writes are performed and the corresponding signals are generated. In this way, memory operations are managed correctly and data communication is provided between the memory and the processor.

8- **MEM_WB.v:** The file "MEM_WB.v" represents the MEM-WB (Memory-Write Back) pipeline stage on a 5-stage RISC-V processor. This module transfers the results from the memory stage to the write back stage. The module receives inputs such as reset signal, memory write number, memory write control signal and memory write data. Based on these inputs, it performs the necessary actions in the MEM-WB stage and generates the corresponding signals. Generally, the MEM_WB module represents the MEM-WB stage in the 5-stage RISC-V pipeline. At this stage, the results from the memory operation are taken and transferred to the write back stage. In this way, it is ensured that the data obtained as a result of the memory operation is written back to the processor correctly.

9- WB.v: The "WB.v" file represents the Write Back stage on a 5-stage RISC-V processor. This module transfers the results from the memory stage to the write back stage. The module receives inputs such as reset signal, memory write number, memory write control signal and memory write data. Based on these inputs, it performs the necessary actions in the Write Back phase and generates the relevant signals. Generally, the WB module represents the Write Back stage in the 5-stage RISC-V pipeline. At this stage, the results from the memory operation are taken and transferred to the write back stage. In this way, it is ensured that the data obtained as a result of the memory operation is correctly written back to the general purpose registers of the processor.

10- **STALL.v:** The file "STALL.v" represents a module used to control latency in the pipeline on a 5-stage RISC-V processor. The module receives inputs such as the reset signal, the load delay signal, and the branch delay signal. Based on these inputs, it generates the corresponding signals to control the latency in the pipeline. The always block in the module checks the values of the corresponding signals. In general, the STALL module provides delay control in the 5-stage RISC-V pipeline. It is used to manage delays in loading and branching operations. In this way, it detects the delays caused by data dependencies in the pipeline and generates delay signals accordingly, ensuring the correct operation of the pipeline.

11- **BRANCH_PRE.v:** The file "BRANCH_PRE.v" represents a module used to manage pre-branch prediction on a 5-stage RISC-V processor. In general, the BRANCH_PRE module does pre-branch prediction in a 5-stage RISC-V pipeline. Estimates are made based on the value of the program counter and the specification of the instruction. The prediction results are used to determine the branching target address, to indicate that the prediction was accepted, and to determine the type of prediction. This provides a more efficient operation for branching instructions in the processor.

12- **TOP_PIPELINED.v:** The file "TOP_PIPELINED.v" represents the main module of the RISC-V processor with a 5-stage pipeline architecture. The module interacts with construction memory (inst_mem) and data memory (data_mem) along with clk and rst signals. Also including various submodules (IF.v, IF_ID.v, ID.v, ID_EX.v, EX.v, EX_MEM.v, MEM.v, MEM_WB.v, REG_FILE.v, BRANCH_PRE.v, STALL.v) and routes I/O signals between these submodules. Various signals defined in the module enable the transmission of data and control signals between pipeline stages. For example, with the inst_i signal, an instruction is taken from the instruction memory and these instructions go through various stages and results are produced.

The pipeline stages are:

- Instruction Fetch (IF): The stage where instructions are retrieved from memory. Receives instructions at addresses specified by the Program Counter (PC).

- Instruction Decode (ID): The stage where the instructions are decoded. Control processes are performed according to the type of instruction and necessary data is drawn.

- Execute (EX): The stage where the instruction is processed. Processor instructions are performed by the ALU (Arithmetic Logic Unit).

- Memory Access (MEM): The stage where memory access is made. The data memory is accessed and necessary actions are taken.

- Write Back (WB): The stage where the results are written. The last calculated values are written to the register file.

The module provides efficient processing by routing data and control signals between these stages. Submodules in each stage realize the features of that stage and generate the necessary signals to transmit the results to the next stage. In summary, the file "TOP_PIPELINED.v" represents the main module of a RISC-V processor with a 5-stage pipeline architecture. This module realizes a high-performance processor design by ensuring that instructions are processed and data is transferred correctly.

Below is a detailed drawing of the 5-Stage Pipeline design and the schematic resulting from the RTL. As expected, there are similarities between drawing and RTL. The difference between the two is the register gates. Register gates are required for multicycle but not for single cycle. The difference between these two designs is due to this.

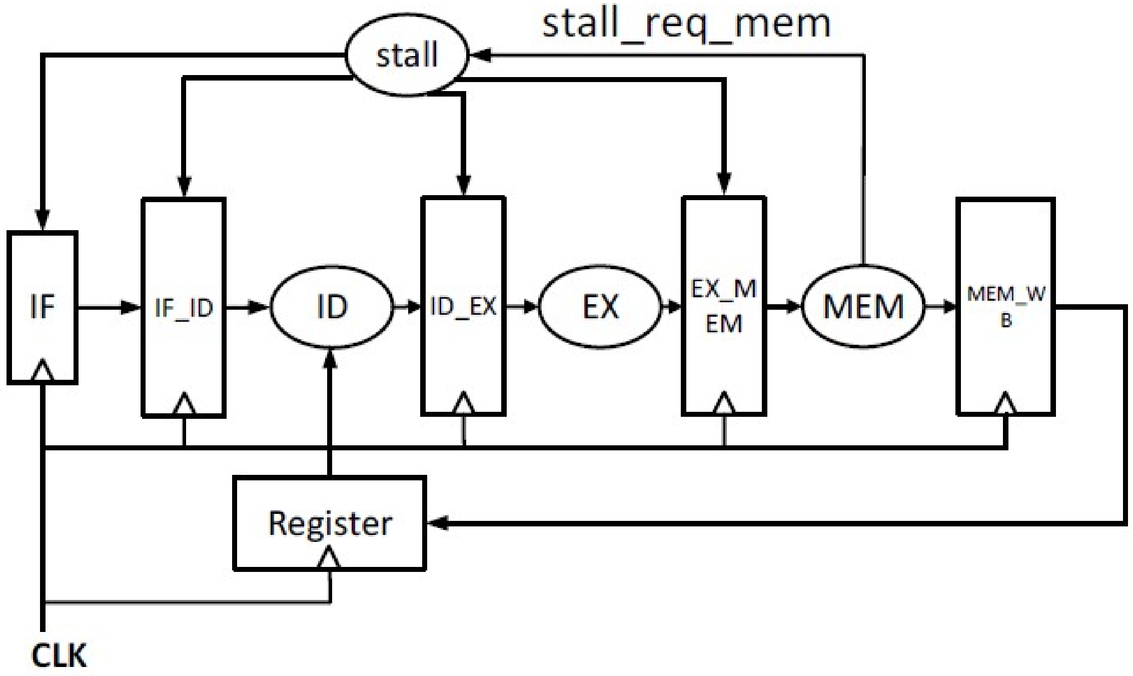


Figure 2: 5 Staged Pipeline RISCV Design

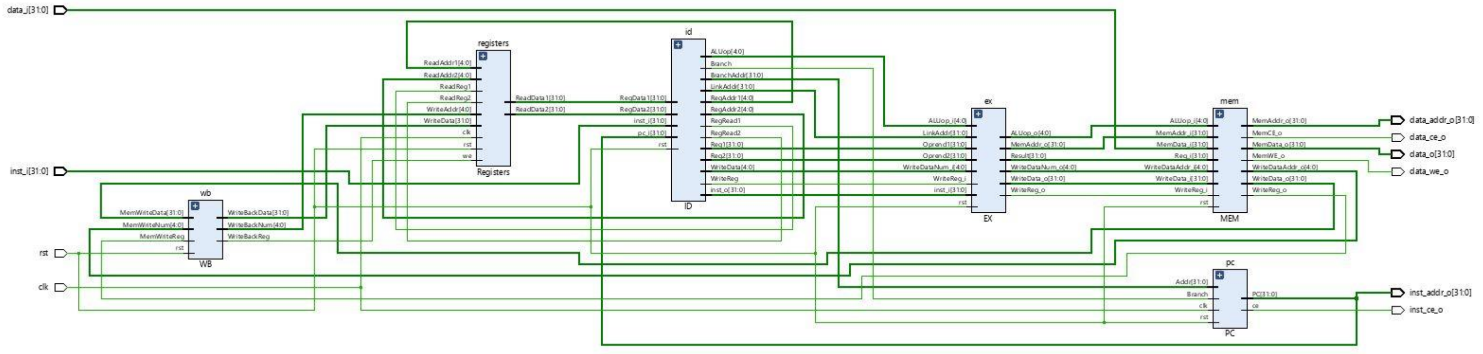
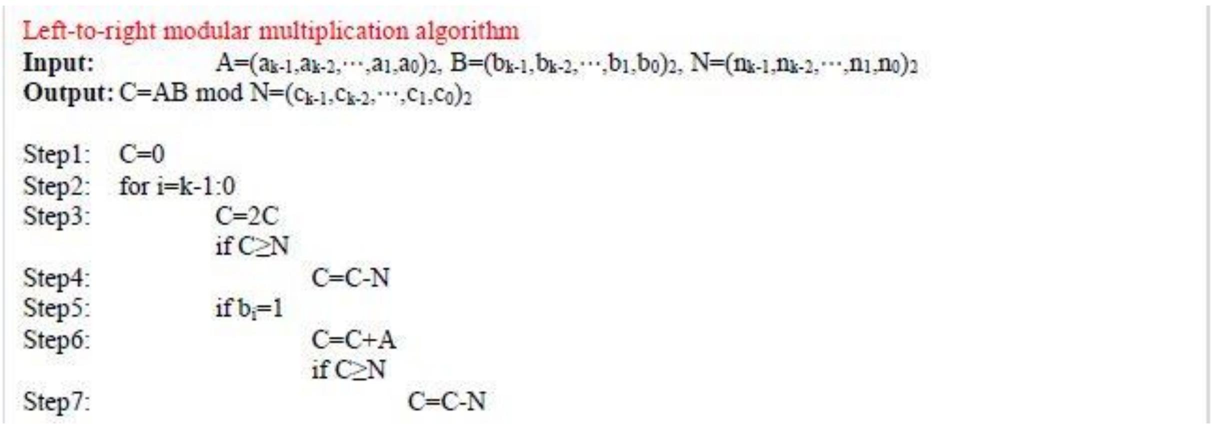


Figure 3: 5-Staged Pipeline RISCV Single Cycle RTL Schematic



Left-to-right modular multiplication algorithm

Input: $A=(a_{k-1},a_{k-2},\cdots,a_1,a_0)_2$, $B=(b_{k-1},b_{k-2},\cdots,b_1,b_0)_2$, $N=(n_{k-1},n_{k-2},\cdots,n_1,n_0)_2$

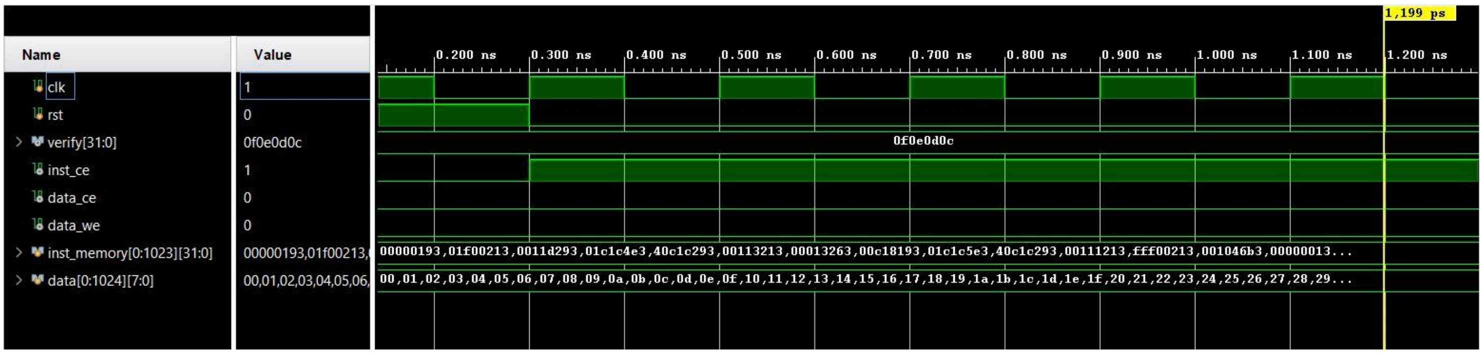
Output: $C=AB \bmod N=(c_{k-1},c_{k-2},\cdots,c_1,c_0)_2$

```
Step1:  C=0
Step2:  for i=k-1:0
Step3:          C=2C
                if C≥N
Step4:                  C=C-N
Step5:          if b_i=1
Step6:                  C=C+A
                        if C≥N
Step7:                          C=C-N
```

Figure 4: Left-to-right modular multiplication algorithm

```
`lui a3, 0` -> `00000000000000000000000110010011`
`lui t0, 31` -> `00000001111100000000001000010011`
`loop: sll a3, a3, 1` -> `00000000000100011101001010010011`
`blt a3, a2, skip_subtraction` -> `00000001110000011100010011100011`
`sub a3, a3, a2` -> `01000000110000011100001010010011`
`skip_subtraction: andi a4, a1, 1` -> `00000000000100010011001000010011`
`beq a4, x0, end_loop` -> `00000000000000010011001001100011`
`add a3, a3, a0` -> `00000000110000011000000110010011`
`blt a3, a2, skip_subtraction2` -> `00000001110000011100010111100011`
`sub a3, a3, a2` -> `01000000110000011100001010010011`
`skip_subtraction2: srl a1, a1, 1` -> `00000000000100010001001000010011`
`addi t0, t0, -1` -> `11111111111100000000001000010011`
`bge t0, x0, loop` -> `00000000000100000100011010110011`
`end_loop: nop` -> `00000000000000000000000000010011`
```

Figure 5: Machine Code of left-to-right modular multiplication algorithm



```
Name                       Value
clk                        1
rst                        0
verify[31:0]               0f0e0d0c
inst_ce                    1
data_ce                    0
data_we                    0
inst_memory[0:1023][31:0]  00000193,01f00213,
data[0:1024][7:0]          00,01,02,03,04,05,06,

1,199 ps
0.200 ns  0.300 ns  0.400 ns  0.500 ns  0.600 ns  0.700 ns  0.800 ns  0.900 ns  1.000 ns  1.100 ns  1.200 ns

0f0e0d0c
00000193,01f00213,0011d293,01c1c4e3,40c1c293,00113213,00013263,00c18193,01c1c5e3,40c1c293,00111213,fff00213,001046b3,00000013...
00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f,20,21,22,23,24,25,26,27,28,29...
```

Figure 6: 5-Staged Pipelined RISC-V Behavioral Simulation

2. STRUCTURAL HAZARDS

a- Structural hazards arise when multiple processor instructions attempt to access the same hardware resource at the same time. For example, in a system with a single ALU (Arithmetic Logic Unit), if one processor instruction uses the ALU while another instruction must use the same ALU, this can lead to a structural hazard.

b- Structural hazard is a situation where read and write operations are performed on the same ports at the same time, such a problem was encountered in the Register File module. That is, in the previous Register File design, read and write operations used the same ports. This can lead to a structural hazard when both a read and a write operation occur in one clock cycle. Because in this case, read and write operations have to share the same ports and this may cause one process to block (hazard) the other.

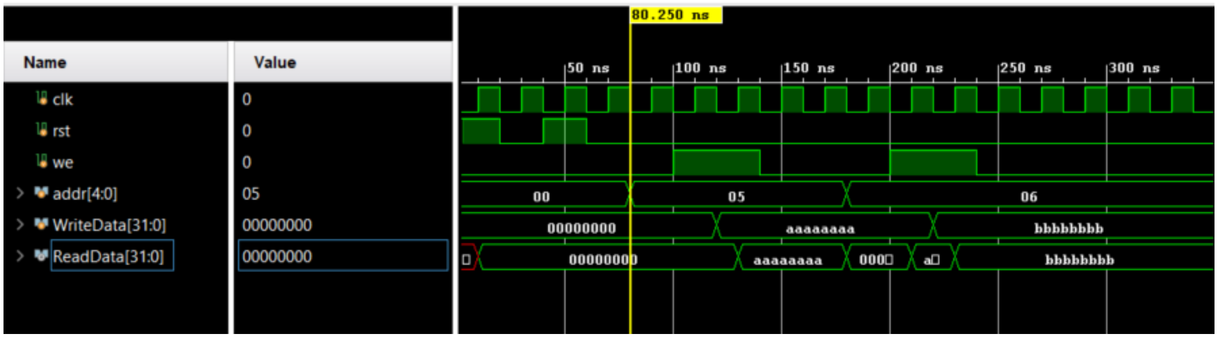
To avoid this structural hazard, the Register File module has been modified as follows:

- The 'we' (write enable) and 'WriteData' ports are used for writing. Writing only happens when the 'we' signal is high and the 'WriteAddr' address is not 0. This is controlled on 'posedge clk', meaning writing happens on the rising edge of each clock cycle.
- For read operations, two different read ports were used: 'ReadReg1' and 'ReadReg2'. Each has its own 'ReadAddr' (ReadAddr1 and ReadAddr2) and 'ReadData' (ReadData1 and ReadData2) ports. Read operations occur when any input changes (always (*) block).

With these changes, even if both reading and writing to the RF occur at the same time, there is no structural hazard as these operations take place over different ports.

As a result, the blocks of Verilog code that have been changed or added are:

- Separate 'always @ (*)' blocks for each read operation.
- Separate 'ReadReg', 'ReadAddr' and 'ReadData' ports for each read port.
- 'always @ (posedge clk)' block where writing is controlled on 'posedge clk'.



Name	Value
clk	0
rst	0
we	0
addr[4:0]	05
WriteData[31:0]	00000000
ReadData[31:0]	00000000

Figure 7: Structural Hazard

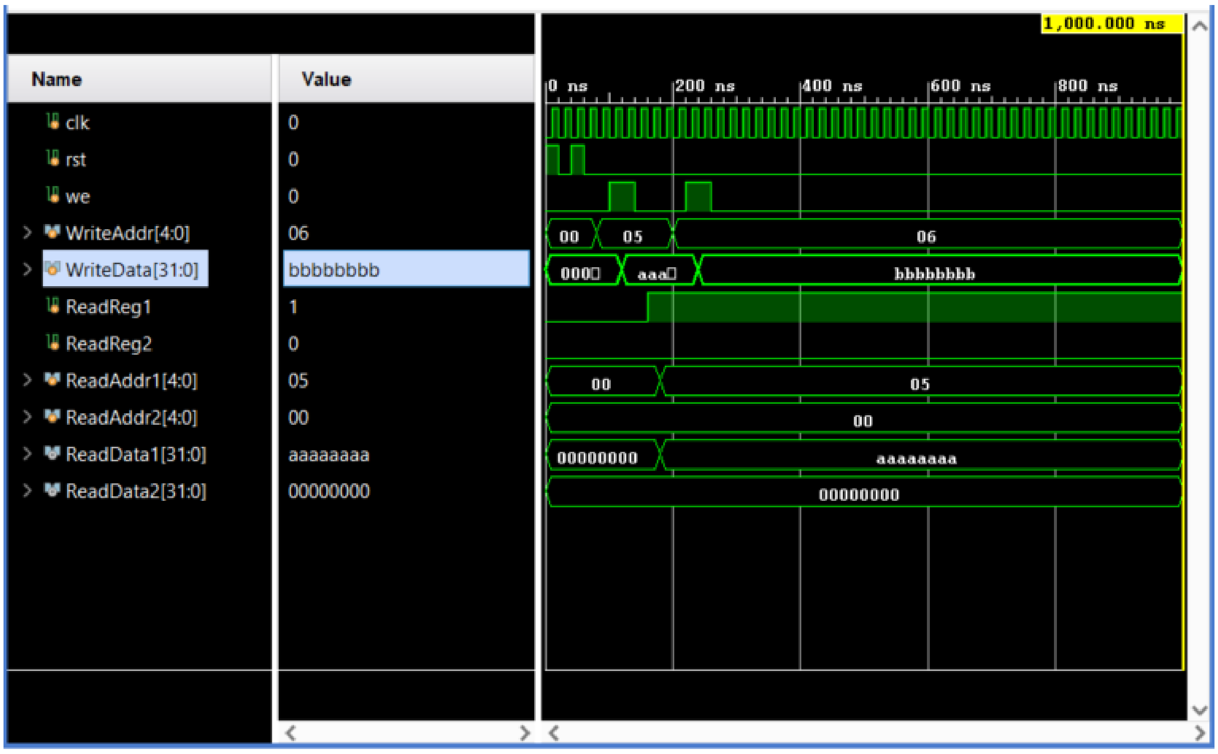


Figure 8: Handled Structural Hazard

3. DATA HAZARDS - READ-AFTER-WRİTE (RAW) HAZARDS

a- The Forwarding_Unit module implements forwarding logic that allows direct transfer of register data between various pipeline stages. The forwardA and forwardB outputs specify which value for registers A and B from which source. 2'b10 is from the EX/MEM pipeline and 2'b01 is from the MEM/WB pipeline. 2'b00 indicates that it will not do any forwarding.

b- Forwarding Unit Verilog Code

```
module Forwarding_Unit (
  input [1:0] EX_MEM_RegWrite,
  input [1:0] MEM_WB_RegWrite,
  input [1:0] ID_EX_RegWrite,
  input [1:0] EX_MEM_MemRead,
  input [1:0] MEM_WB_MemRead,
  input [1:0] ID_EX_MemWrite,
  input [1:0] EX_MEM_RD,
  input [1:0] MEM_WB_RD,
  input [4:0] EX_MEM_RS,
  input [4:0] EX_MEM_RT,
  input [4:0] MEM_WB_RS,
  input [4:0] MEM_WB_RT,
  input [4:0] ID_EX_RS,
  input [4:0] ID_EX_RT,
  output reg [1:0] forwardA,
  output reg [1:0] forwardB
);

  always @* begin
    // Forwarding for register A
    if (EX_MEM_RegWrite[0] && (EX_MEM_RD[0] != 0) && (EX_MEM_RD[0] == ID_EX_RS[0]))
      forwardA = 2'b10;  // Forward from EX/MEM pipeline register
    else if (MEM_WB_RegWrite[0] && (MEM_WB_RD[0] != 0) && (MEM_WB_RD[0] ==
ID_EX_RS[0]))
      forwardA = 2'b01;  // Forward from MEM/WB pipeline register
    else
      forwardA = 2'b00;  // No forwarding

    if (EX_MEM_RegWrite[1] && (EX_MEM_RD[1] != 0) && (EX_MEM_RD[1] == ID_EX_RS[0]))
      forwardA = 2'b10;  // Forward from EX/MEM pipeline register
    else if (MEM_WB_RegWrite[1] && (MEM_WB_RD[1] != 0) && (MEM_WB_RD[1] ==
ID_EX_RS[0]))
      forwardA = 2'b01;  // Forward from MEM/WB pipeline register
    else
      forwardA = 2'b00;  // No forwarding

    // Forwarding for register B
    if (EX_MEM_RegWrite[0] && (EX_MEM_RD[0] != 0) && (EX_MEM_RD[0] == ID_EX_RT[0]))
      forwardB = 2'b10;  // Forward from EX/MEM pipeline register
    else if (MEM_WB_RegWrite[0] && (MEM_WB_RD[0] != 0) && (MEM_WB_RD[0] ==
ID_EX_RT[0]))
      forwardB = 2'b01;  // Forward from MEM/WB pipeline register
    else
      forwardB = 2'b00;  // No forwarding

    if (EX_MEM_RegWrite[1] && (EX_MEM_RD[1] != 0) && (EX_MEM_RD[1] == ID_EX_RT[0]))
      forwardB = 2'b10;  // Forward from EX/MEM pipeline register
    else if (MEM_WB_RegWrite[1] && (MEM_WB_RD[1] != 0) && (MEM_WB_RD[1] ==
ID_EX_RT[0]))
      forwardB = 2'b01;  // Forward from MEM/WB pipeline register
    else
      forwardB = 2'b00;  // No forwarding
  end

endmodule
```

c- Behavioral Simulation

The Forwarding_Unit module implements forwarding logic that allows direct transfer of register data between various pipeline stages. The forwardA and forwardB outputs specify which value for registers A and B from which source. 2'b10 is from the EX/MEM pipeline and 2'b01 is from the MEM/WB pipeline. 2'b00 indicates that it will not do any forwarding.

As seen below, our module does not give correct results in some cases. Although we worked hard on it, we could not find where the error was in the module.

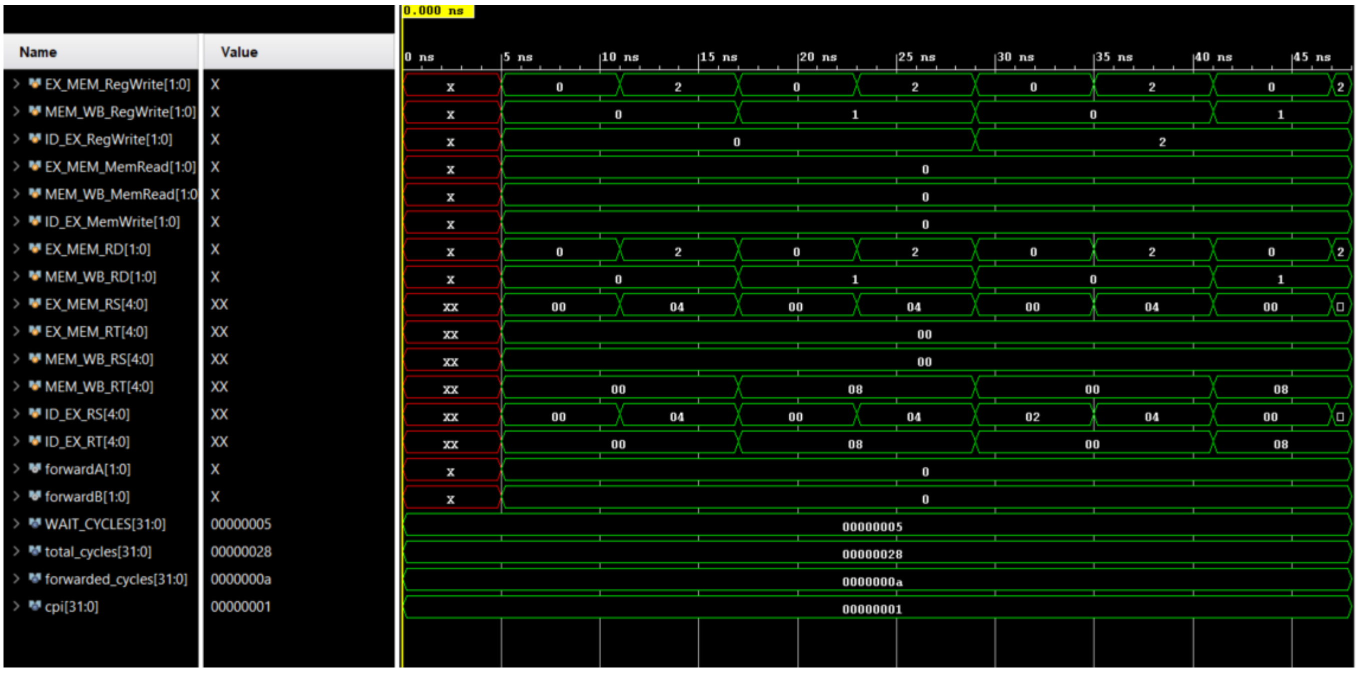
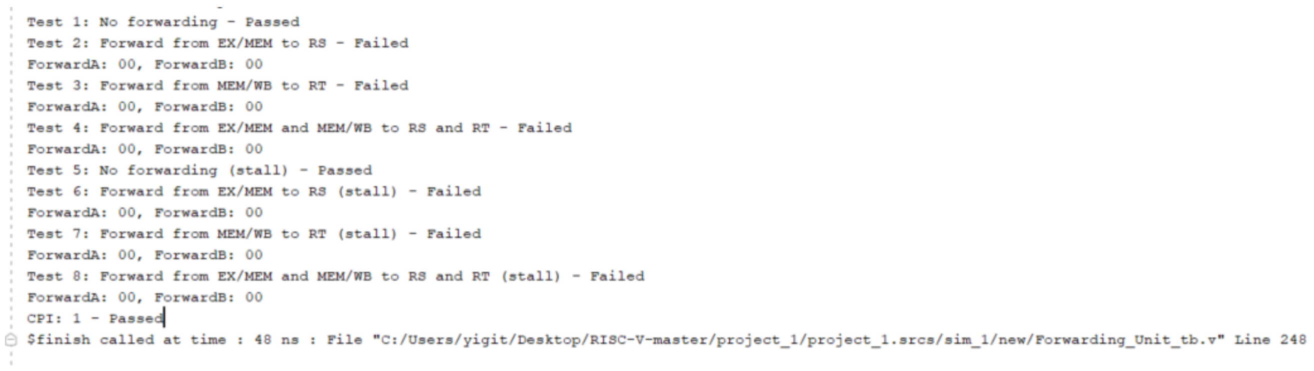


Figure 8: Forwarding Unit Behavioral Simulation



```
Test 1: No forwarding - Passed
Test 2: Forward from EX/MEM to RS - Failed
ForwardA: 00, ForwardB: 00
Test 3: Forward from MEM/WB to RT - Failed
ForwardA: 00, ForwardB: 00
Test 4: Forward from EX/MEM and MEM/WB to RS and RT - Failed
ForwardA: 00, ForwardB: 00
Test 5: No forwarding (stall) - Passed
Test 6: Forward from EX/MEM to RS (stall) - Failed
ForwardA: 00, ForwardB: 00
Test 7: Forward from MEM/WB to RT (stall) - Failed
ForwardA: 00, ForwardB: 00
Test 8: Forward from EX/MEM and MEM/WB to RS and RT (stall) - Failed
ForwardA: 00, ForwardB: 00
CPI: 1 - Passed
$finish called at time : 48 ns : File "C:/Users/yigit/Desktop/RISC-V-master/project_1/project_1.srcs/sim_1/new/Forwarding_Unit_tb.v" Line 248
```

Figure 8: Forwarding Unit TCL Console Output

4. DATA HAZARDS - LOAD-USE HAZARDS

a- Verilog Code

```
module STALL(

    input   wire        rst,
    input   wire        StallLoad,
    input   wire        StallBranch,
    output  reg [5:0]   stall

);

/*
 * This always part controls the signal stall.
 */
always @ (*) begin
    if (rst)
        stall <= 6'b0;
    else if (StallBranch)
        stall <= 6'b000010;
    else if (StallLoad)
        stall <= 6'b000111;
    else
        stall <= 6'b0;
end

endmodule
```

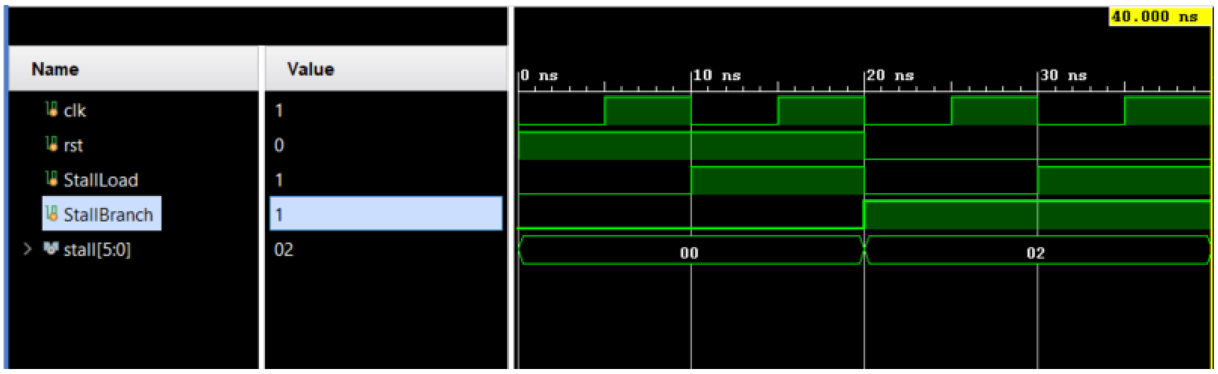
b- Behavioral Simulation

The STALL module is responsible for controlling the stall signal based on the inputs rst, StallLoad, and StallBranch. The purpose of the STALL module is to handle data hazards, specifically load-use hazards, in a processor pipeline. A load-use hazard occurs when a load instruction depends on the result of a previous instruction (e.g., an arithmetic or logic operation) that is still being processed in the pipeline. In such cases, the load instruction needs to be stalled or delayed until the data dependency is resolved. The STALL module monitors the rst, StallLoad, and StallBranch signals to determine if a stall is required. Here's a breakdown of the behavior:

- When the rst signal is active (rst = 1), indicating a reset condition, the stall signal is set to 0 to ensure no stalls are applied.

- If the StallBranch signal is active (StallBranch = 1), indicating a branch hazard, the stall signal is set to 000010 to insert stalls in the pipeline to handle the hazard.
- If the StallLoad signal is active (StallLoad = 1), indicating a load-use hazard, the stall signal is set to `000111` to stall the pipeline until the data dependency is resolved.
- In all other cases, when neither StallBranch nor StallLoad is active, the stall signal is set to 0 to allow normal pipeline operation without any stalls.

Overall, the STALL module plays a crucial role in managing data hazards in a processor pipeline by controlling the `stall` signal appropriately. By inserting stalls when necessary, it ensures correct execution and data flow in the pipeline, preventing incorrect or inconsistent results due to data hazards.



Name	Value
clk	1
rst	0
StallLoad	1
StallBranch	1
stall[5:0]	02

Figure 10: STALL Module’s Behaivoral Simulation

5. CONTROL HAZARDS – JUMPS

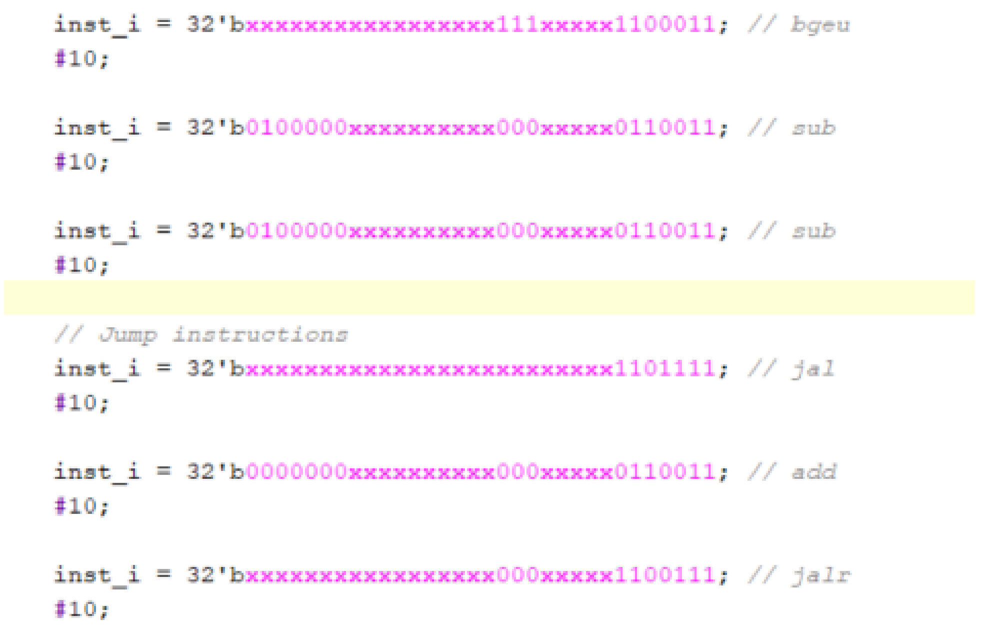
a- Verilog Code

```
module RISC_V_Processor (
  // Inputs and outputs of the processor module
  input         clk,
  input         rst,
  input  [31:0] inst_i,
  output [31:0] result
);

  // Pipeline stages
  reg [31:0] inst1, inst2, inst3;
  reg [4:0]  stage;

  // Control signals
  wire        isJump;
  reg         flush;

  // Instruction execution logic
  always @(posedge clk) begin
    if (rst) begin
      // Reset all pipeline stages and control signals
      inst1 <= 32'b0;
      inst2 <= 32'b0;
      inst3 <= 32'b0;
      stage <= 0;
      flush <= 0;
    end
    else begin
      // Fetch stage
      if (stage == 0) begin
        if (flush) begin
          // Flush the fetch stage
          inst1 <= 32'b0;
        end
        else begin
          // Fetch the instruction
          inst1 <= inst_i;
        end
        stage <= 1;
      end
      // Decode stage
      else if (stage == 1) begin
        if (flush) begin
          // Flush the decode stage
          inst2 <= 32'b0;
        end
        else begin
          // Decode the instruction
          inst2 <= inst1;
        end
        stage <= 2;
      end
      // Execute stage
      else if (stage == 2) begin
        if (flush) begin
          // Flush the execute stage
          inst3 <= 32'b0;
        end
        else begin
          // Execute the instruction
          inst3 <= inst2;
        end
        stage <= 0;
      end
    end
  end

  // Jump detection logic
  assign isJump = (inst3[6:0] == 7'b1101111) || (inst3[6:0] == 7'b1100111);

  // Flushing logic
  always @(posedge clk) begin
    if (rst) begin
      flush <= 0;
    end
    else begin
      if (isJump) begin
        flush <= 1;
      end
      else begin
        flush <= 0;
      end
    end
  end

  assign result = inst3;

endmodule
```

b- Behavioral Simulation

- Pipeline Stages: The module includes three registers (`inst1`, `inst2`, `inst3`) to represent the pipeline stages, holding the instructions fetched, decoded, and executed in each stage. The `stage` register keeps track of the current stage of the pipeline.
- Control Signals: The `isJump` wire is assigned the value of 1 if the current instruction is a jump instruction (either JAL or JALR), and 0 otherwise. The `flush` register is used to control the flushing operation in the pipeline.
- Instruction Execution Logic: Inside the `always @(posedge clk)` block, the code implements the logic for executing the instructions in the pipeline stages. It handles the stages based on the current `stage` value, fetches the instruction from `inst_i`, and propagates the instructions through the pipeline stages.
- Jump Detection Logic: The `isJump` wire is assigned the value of 1 when the current instruction (`inst3`) is a jump instruction (JAL or JALR). This is achieved by comparing the opcode field (bits 6 to 0) of `inst3` with the jump instruction opcodes.
- Flushing Logic: The `flush` register is updated based on the rising edge of the clock. If the `rst` signal is active, indicating a reset condition, `flush` is set to 0. If the current instruction (`inst3`) is a jump instruction (`isJump` is 1), `flush` is set to 1 to trigger the flushing operation in the pipeline. Otherwise, `flush` is set to 0 to allow normal pipeline operation.
- Result Assignment: The `result` output is assigned the value of the instruction in the execute stage (`inst3`). This represents the final result produced by the processor.

By examining the waveform outputs, you can verify that the instructions are executed in the pipeline stages, the flushing operation occurs when required, and the correct result is produced by the processor.



```
inst_i = 32'bxxxxxxxxxxxxxxxxx111xxxxx1100011; // bgeu
#10;

inst_i = 32'b0100000xxxxxxxxxx000xxxxx0110011; // sub
#10;

inst_i = 32'b0100000xxxxxxxxxx000xxxxx0110011; // sub
#10;


// Jump instructions
inst_i = 32'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111; // jal
#10;

inst_i = 32'b0000000xxxxxxxxxx000xxxxx0110011; // add
#10;

inst_i = 32'bxxxxxxxxxxxxxxxxx000xxxxx1100111; // jalr
#10;
```

Figure 11: Some instructions on testbench

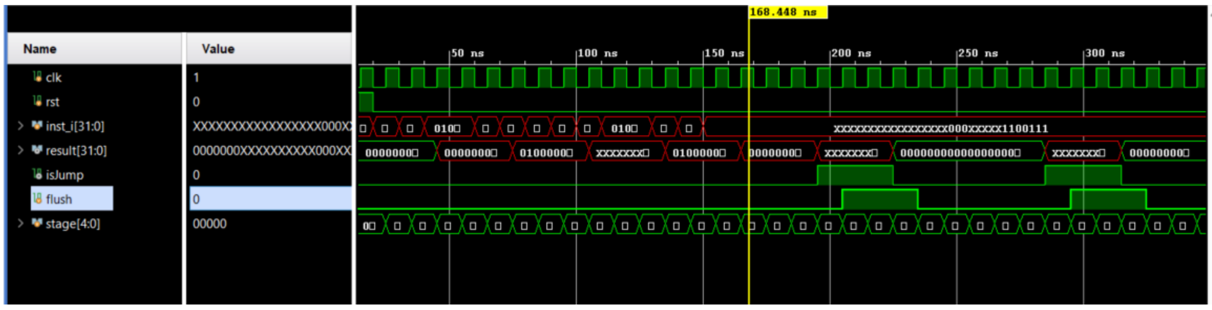


Figure 12: ControlHazards_Jumps Module’s Behavioral Simulation

6. CONTROL HAZARDS – BRANCHES

a- Verilog Code

```
module ControlHazards_Branches (
  // Inputs and outputs of the processor module
  input          clk,
  input          rst,
  input   [31:0] inst_i,
  output  [31:0] result
);

  // Pipeline stages
  reg [31:0] inst1, inst2, inst3;
  reg [4:0]  stage;

  // Control signals
  wire        isBranch;
  reg         flush;

  // Instruction execution logic
  always @(posedge clk) begin
    if (rst) begin
      // Reset all pipeline stages and control signals
      inst1 <= 32'b0;
      inst2 <= 32'b0;
      inst3 <= 32'b0;
      stage <= 0;
      flush <= 0;
    end
    else begin
      // Fetch stage
      if (stage == 0) begin
        if (flush) begin
          // Flush the fetch stage
          inst1 <= 32'b0;
        end
        else begin
          // Fetch the instruction
          inst1 <= inst_i;
        end
        stage <= 1;
      end
      // Decode stage
      else if (stage == 1) begin
        if (flush) begin
          // Flush the decode stage
          inst2 <= 32'b0;
        end
        else begin
          // Decode the instruction
          inst2 <= inst1;
        end
      // Execute stage
      else if (stage == 2) begin
        if (flush) begin
          // Flush the execute stage
          inst3 <= 32'b0;
        end
        else begin
          // Execute the instruction
          inst3 <= inst2;
        end
        stage <= 0;
      end
    end
  end

  // Branch detection logic
  assign isBranch = (inst_i[6:0] == 7'b1100011)? 1'b1: 1'b0;

  // Flushing logic
  always @(posedge clk) begin
    if (rst) begin
      flush <= 0;
    end
    else begin
      if (isBranch) begin
        flush <= 1;
      end
      else begin
        flush <= 0;
      end
    end
  end

  // Result assignment (you need to modify this according to your design)
  assign result = inst3;

endmodule
```

In the code, the isBranch signal is used to determine whether the given instruction is a branch instruction. Bits 6 to 0 of the instruction are compared against a specific pattern that specifies branch instructions in the RISC-V architecture. If the instruction is a branch instruction, the isBranch signal is set to 1, otherwise it is set to 0. The flush signal is used to flush the pipeline when a branch instruction is detected. If a branch instruction is detected, the flush signal is set to 1 and the pipeline data is reset until the next instruction pull step.

During the simulation, it should be seen that the instructions are given in order and processed correctly by the processor. Especially in the case of branch instructions, it should be checked that the pipeline is cleaned and correct results are obtained. Validation indicates that the result signal contains the expected results.

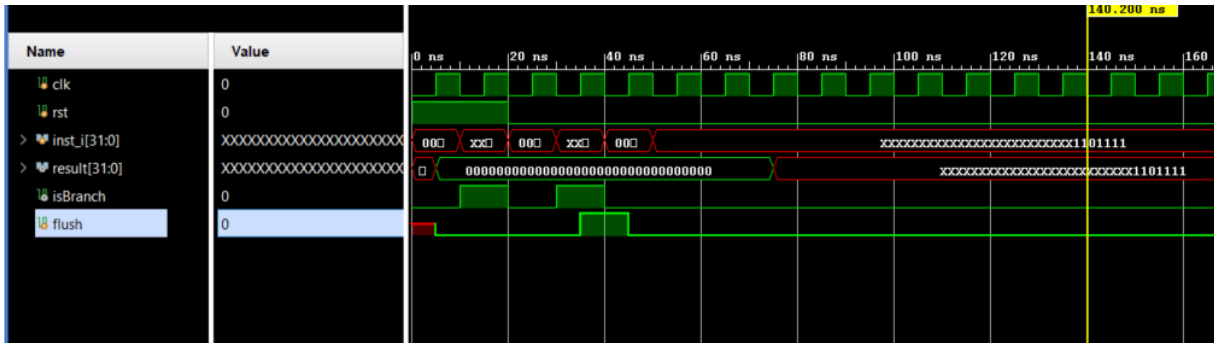
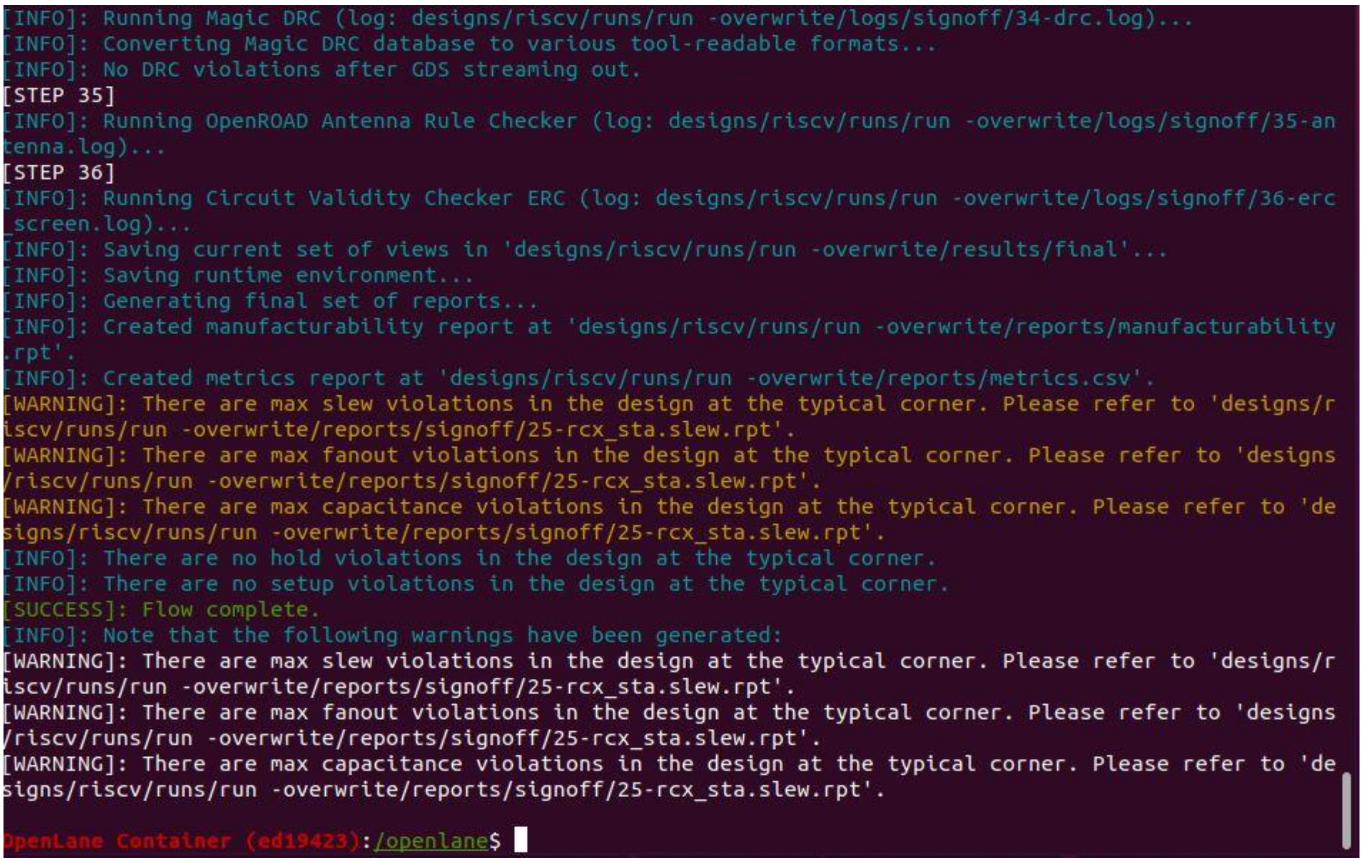


Figure 13: ControlHazards_Jumps Module's Behavioral Simulation

7. OPENLANE FLOW

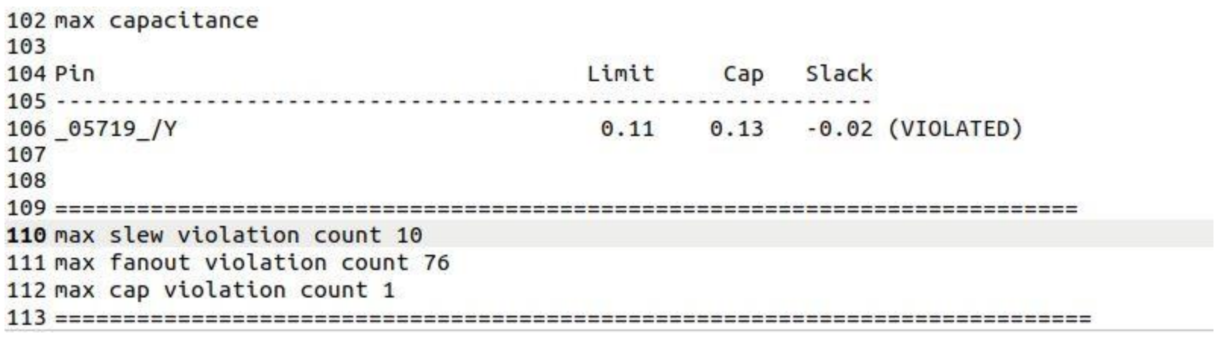
1- As can be seen below, our module has been successfully synthesized.



```
[INFO]: Running Magic DRC (log: designs/riscv/runs/run -overwrite/logs/signoff/34-drc.log)...
[INFO]: Converting Magic DRC database to various tool-readable formats...
[INFO]: No DRC violations after GDS streaming out.
[STEP 35]
[INFO]: Running OpenROAD Antenna Rule Checker (log: designs/riscv/runs/run -overwrite/logs/signoff/35-an
tenna.log)...
[STEP 36]
[INFO]: Running Circuit Validity Checker ERC (log: designs/riscv/runs/run -overwrite/logs/signoff/36-erc
screen.log)...
[INFO]: Saving current set of views in 'designs/riscv/runs/run -overwrite/results/final'...
[INFO]: Saving runtime environment...
[INFO]: Generating final set of reports...
[INFO]: Created manufacturability report at 'designs/riscv/runs/run -overwrite/reports/manufacturability
.rpt'.
[INFO]: Created metrics report at 'designs/riscv/runs/run -overwrite/reports/metrics.csv'.
[WARNING]: There are max slew violations in the design at the typical corner. Please refer to 'designs/r
iscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.
[WARNING]: There are max fanout violations in the design at the typical corner. Please refer to 'designs
/riscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.
[WARNING]: There are max capacitance violations in the design at the typical corner. Please refer to 'de
signs/riscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.
[INFO]: There are no hold violations in the design at the typical corner.
[INFO]: There are no setup violations in the design at the typical corner.
[SUCCESS]: Flow complete.
[INFO]: Note that the following warnings have been generated:
[WARNING]: There are max slew violations in the design at the typical corner. Please refer to 'designs/r
iscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.
[WARNING]: There are max fanout violations in the design at the typical corner. Please refer to 'designs
/riscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.
[WARNING]: There are max capacitance violations in the design at the typical corner. Please refer to 'de
signs/riscv/runs/run -overwrite/reports/signoff/25-rcx_sta.slew.rpt'.

OpenLane Container (ed19423):/openlane$
```

Figure 14: Openlane Sythensize Results

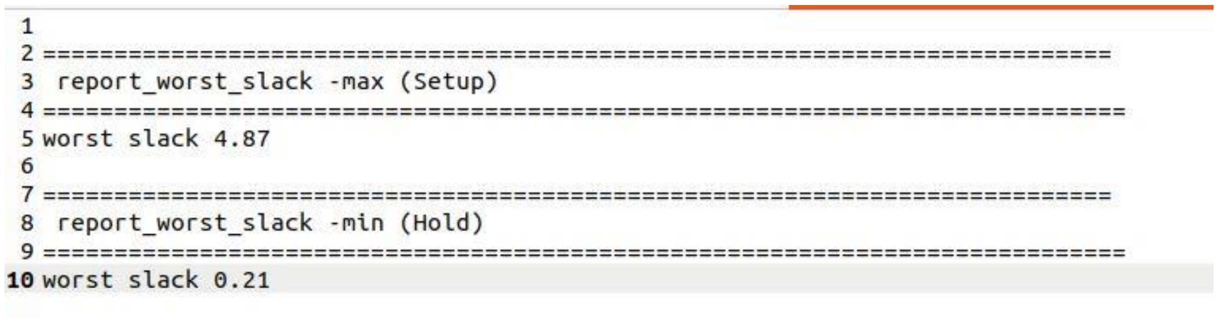
2-



```
max capacitance

Pin                                  Limit     Cap    Slack
------------------------------------------------------------
_05719_/Y                             0.11     0.13   -0.02 (VIOLATED)


===========================================================================
max slew violation count 10
max fanout violation count 76
max cap violation count 1
===========================================================================
```

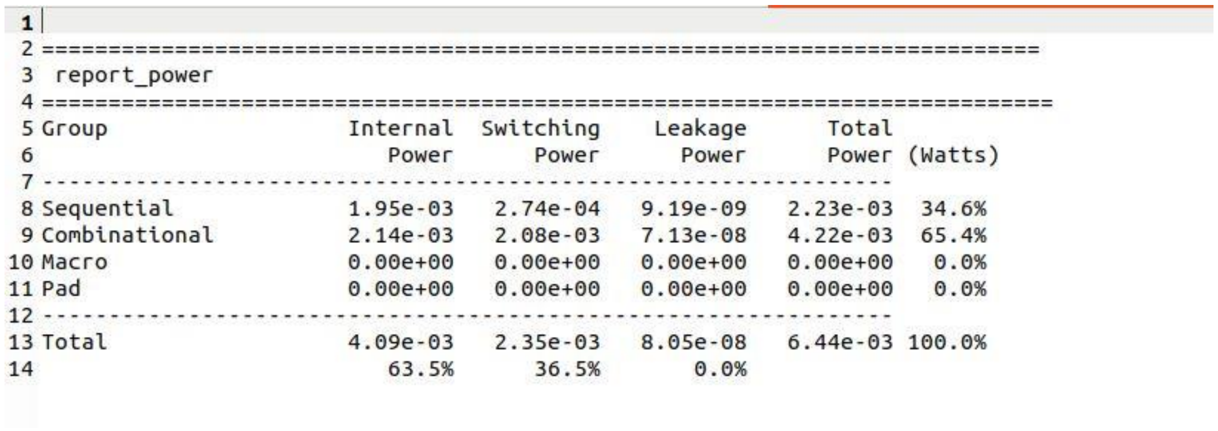
Figure 15: Max fanout, capacitance and slew reports

Maximum clock frequency is 1/4.87*10^-9=205.338MHz



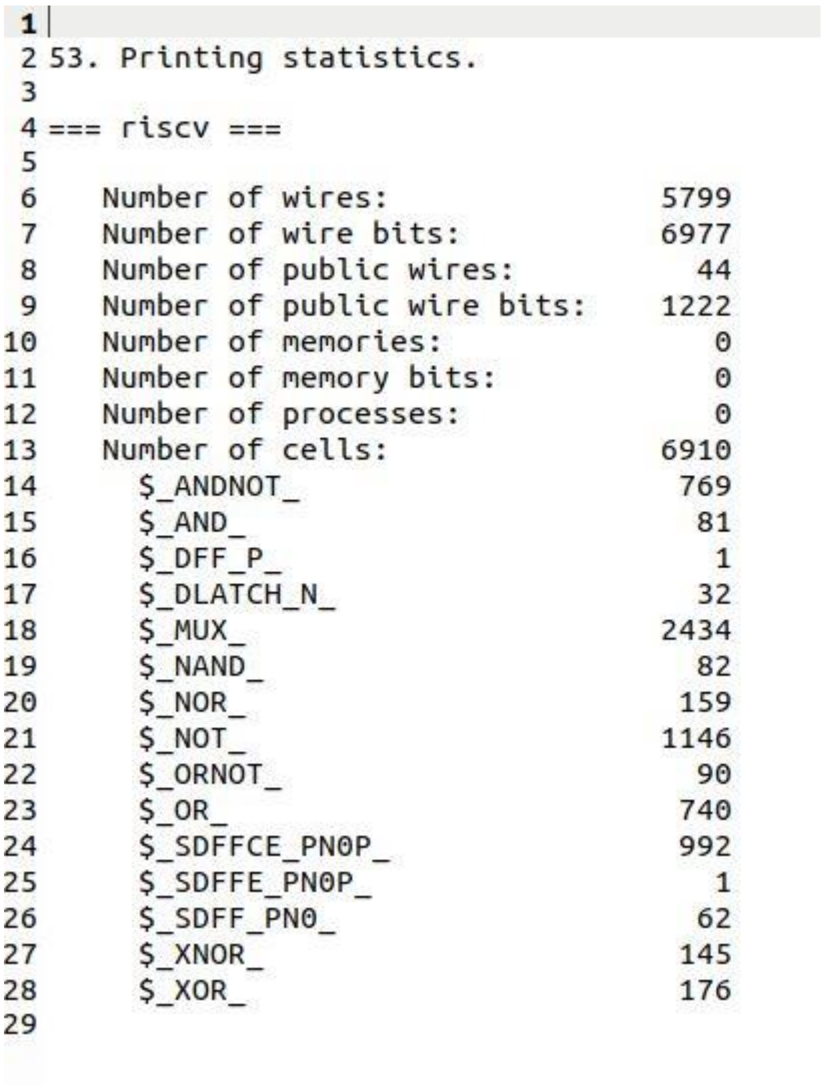
```

===========================================================================
 report_worst_slack -max (Setup)
===========================================================================
worst slack 4.87

===========================================================================
 report_worst_slack -min (Hold)
===========================================================================
worst slack 0.21
```

Figure 16: Worst slack report



```

======================================================================
 report_power
======================================================================
Group                  Internal  Switching    Leakage      Total
                          Power      Power      Power      Power (Watts)
----------------------------------------------------------------
Sequential             1.95e-03   2.74e-04   9.19e-09   2.23e-03  34.6%
Combinational          2.14e-03   2.08e-03   7.13e-08   4.22e-03  65.4%
Macro                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
Pad                    0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
----------------------------------------------------------------
Total                  4.09e-03   2.35e-03   8.05e-08   6.44e-03 100.0%
                          63.5%      36.5%       0.0%
```

Figure 17: Power consumption report



```

53. Printing statistics.

=== riscv ===

   Number of wires:               5799
   Number of wire bits:           6977
   Number of public wires:          44
   Number of public wire bits:    1222
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               6910
     $_ANDNOT_                     769
     $_AND_                         81
     $_DFF_P_                        1
     $_DLATCH_N_                    32
     $_MUX_                       2434
     $_NAND_                        82
     $_NOR_                        159
     $_NOT_                       1146
     $_ORNOT_                       90
     $_OR_                         740
     $_SDFFCE_PN0P_                992
     $_SDFFE_PN0P_                   1
     $_SDFF_PN0_                    62
     $_XNOR_                       145
     $_XOR_                        176

```

Figure 18: Total cell count report