

**BERZIET UNIVERSITY**

**Faculty of Engineering & Technology – Electrical & Computer**

**Engineering Department**

**First Semester 2022/2023**

**COMPUTER ARCHITECTURE ENCS4370**

**Simple Pipelined RISC Processor**

**Project2 Report**

**Saleh Khatib 1200991**

**Taher Hasan 1191740**

**Tala Adel Abahra 1201002**

**Abstract:**

The objective of this project is to develop and evaluate a 32-bit **pipelined** RISC processor using Verilog. The processor incorporates a 32-bit program counter, where the least significant bit functions as a stop flag. It consists of 32 general-purpose registers and utilizes conditional execution for all instructions. Additionally, the processor includes a 2-bit field for instruction type and a 5-bit field for function.

The ALU (Arithmetic Logic Unit) within the processor produces a "zero" signal, which indicates whether the result of the previous ALU operation is zero. The processor supports four instruction types: R-type, I-type, J-type, and S-type. It incorporates a control stack for managing function calls and returns, and a stack pointer (SP) serves as a specialized register that points to the top of the control stack. The SP maintains the address of the empty element at the stack's top, and its initial value is set to zero.

The data path is implemented with a five-stage pipeline, and registers are utilized to facilitate data transfer between stages. The control logic is designed using a state machine approach. The development of the Pipelined-cycle Datapath and the associated control logic provides a foundation for further testing and enhancement of the RISC processor design.

Contents

[1. Designing the Datapath and Control Signals 5](#_Toc140091707)

[1.1. Fetch 6](#_Toc140091708)

[1.2. Decode stage: 7](#_Toc140091709)

[1.3. execute stage 10](#_Toc140091710)

[1.4. Memory Stage and Write Back Stage 12](#_Toc140091711)

[2. Building the State Machine Diagram with Control signals generation 14](#_Toc140091712)

[2.1 Control signals generation 14](#_Toc140091713)

[2.1.1 main control 14](#_Toc140091714)

[2.1.2 ALU table 15](#_Toc140091715)

[2.1.3 SP control and PC control tables 16](#_Toc140091716)

[2.1.4 Forwarding code 17](#_Toc140091717)

[2.1.5 Control hazard code 18](#_Toc140091718)

[2.2 state diagram 19](#_Toc140091719)

[3. Testing the Modules 21](#_Toc140091720)

[3.1 First Test 21](#_Toc140091721)

[3.2 Second Test 25](#_Toc140091722)

[4. Conclusion 32](#_Toc140091723)

[5. Appendix 33](#_Toc140091724)

# Table of Figures

[Figure 1: Datapath 6](#_Toc140091631)

[Figure 2: fetch 7](#_Toc140091632)

[Figure 3: RegFile 8](#_Toc140091633)

[Figure 4: Stack pointer 9](#_Toc140091634)

[Figure 5: Execute stage 11](#_Toc140091635)

[Figure 6: Memory Stage and Write Back Stage 13](#_Toc140091636)

[Figure 7: state machine diagram 20](#_Toc140091637)

[Figure 8: Test1\_cycle1 23](#_Toc140091638)

[Figure 9: Test1\_cycle2 23](#_Toc140091639)

[Figure 10: Test1\_cycle3 23](#_Toc140091640)

[Figure 11:Test1\_cycle4 24](#_Toc140091641)

[Figure 12: Test1\_cycle5 24](#_Toc140091642)

[Figure 13: Test1\_cycle6 24](#_Toc140091643)

[Figure 14: Test1\_cycle7 24](#_Toc140091644)

[Figure 15: Test1\_cycle8 25](#_Toc140091645)

[Figure 16: Test1\_cycle9 25](#_Toc140091646)

[Figure 17: Test1\_cycle10 25](#_Toc140091647)

[Figure 18: Test1\_cycle11 26](#_Toc140091648)

[Figure 19: Test2\_cycle1 27](#_Toc140091649)

[Figure 20: Test2\_cycle2 27](#_Toc140091650)

[Figure 21: Test2\_cycle3 27](#_Toc140091651)

[Figure 22: Test2\_cycle4 27](#_Toc140091652)

[Figure 23: Test2\_cycle5 28](#_Toc140091653)

[Figure 24: Test2\_cycle6 28](#_Toc140091654)

[Figure 25: Test2\_cycle7 28](#_Toc140091655)

[Figure 26: Test2\_cycle8 29](#_Toc140091656)

[Figure 27: Test2\_cycle9 29](#_Toc140091657)

[Figure 28: Test2\_cycle10 29](#_Toc140091658)

[Figure 29: Test2\_cycle11 29](#_Toc140091659)

[Figure 30: Test2\_cycle12 30](#_Toc140091660)

[Figure 31: Test2\_cycle13 30](#_Toc140091661)

[Figure 32: Test2\_cycle14 30](#_Toc140091662)

[Figure 33: Test2\_cycle15 31](#_Toc140091663)

[Figure 34: Test2\_cycle16 31](#_Toc140091664)

[Figure 35: Test2\_cycle17 31](#_Toc140091665)

[Figure 36:Test2\_cycle18 32](#_Toc140091666)

[Figure 37:Test2\_cycle19 32](#_Toc140091667)

# 1. Designing the Datapath and Control Signals

The data path was built as shown in the figure below:

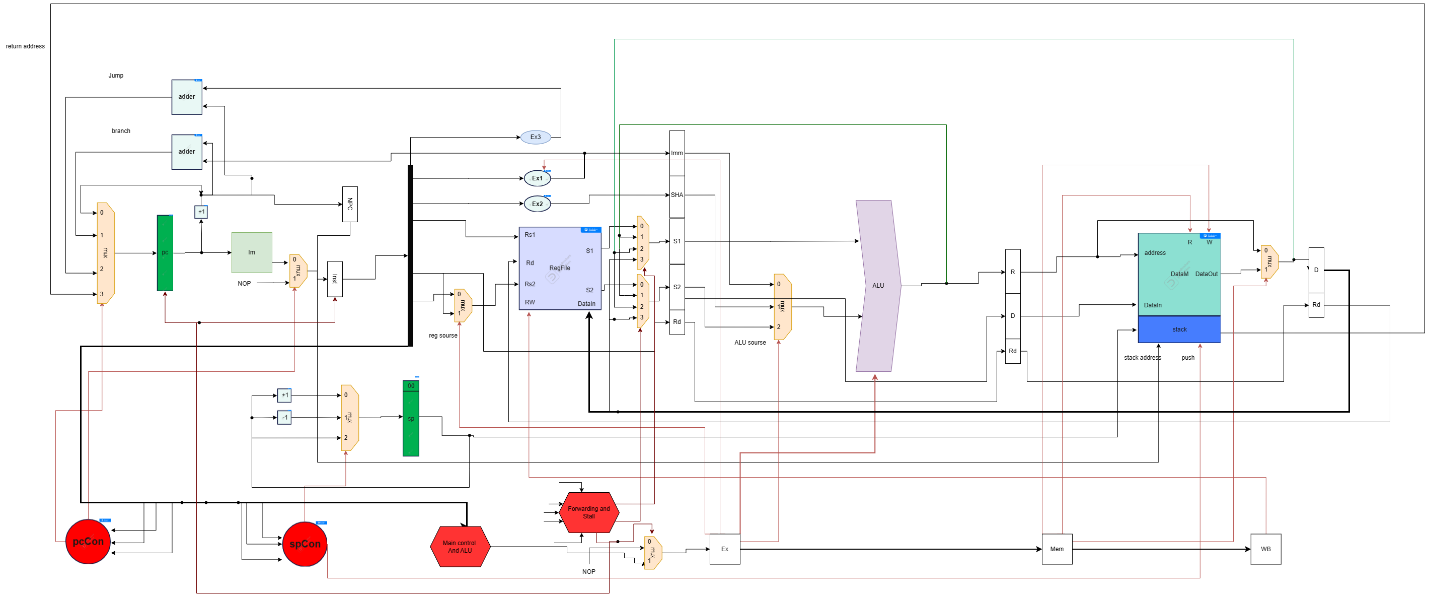


Figure 1: Datapath

**Design and Implementation**

Our design consists of four main blocks:

- Arithmetic and Logical Unit (ALU)

- Control Unit

- Register File

- Memory

Each one of these components will be used in the full Datapath along with the parts that are needed to enable pipelining and its hazard detection unit, forwarding unit, and the buffers between the stages. The design will include five stages: Fetching, Decoding, ALU, Memory and the Write Back stage.

## 1.1. Fetch

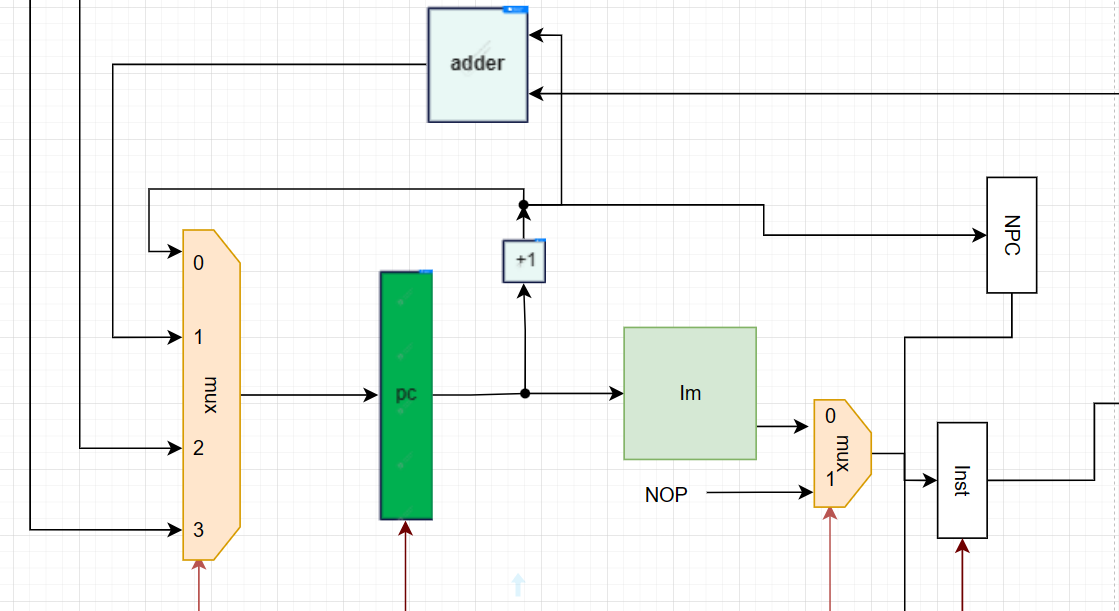


Figure 2: fetch

During the Fetch stage, the CPU relies on the PC Register to hold the memory address of the next instruction to fetch. The PC Register receives the address from the Memory code (IM) as input. A multiplexer (MUX) before the PC determines the source of the next address: 0 for the subsequent instruction, 1 for a branch instruction, 2 for a jump instruction, and 3 for a return address. Another MUX after the Instruction Memory (IM) allows the instruction to be accessed if the select value is zero, or performs a no-operation (NOP) if the value is 1. Additionally, two buffers are visible in the figure: the first buffer, Next PC (NPC), is connected to the memory stage's Stack to store the return address, while the second buffer, Instruction (Inst), stores the fetched instruction.

Overall, the Fetch stage involves the PC Register, MUXes for determining the next address source and accessing the instruction, and buffers for storing the next PC and fetched instruction. These components work together to retrieve the instruction from memory and prepare for subsequent stages of the instruction execution cycle.

## 1.2. Decode stage:

We have two parts in this stage, the first one for Register file and the second for stack pointer

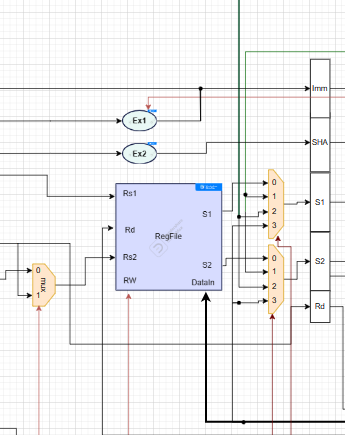


Figure 3: RegFile

The figure illustrates the Register File component, which consists of various registers. It includes Rs1 and Rs2 as the first and second source registers, respectively. Rd serves as the destination register, while RW acts as the read-write flag. DataIn represents the input data for writing into the register. The outputs S1 and S2 are also part of the Register File.

Additionally, there are three multiplexers (MUXes) depicted in the figure. The first MUX on the left selects Rs if the select value is zero, and Rd if the value is 1. The second and third MUXes operate as follows: 0 corresponds to the output of the Register File from the decode stage, 1 corresponds to the output of the Arithmetic Logic Unit (ALU) from the execution stage, 2 corresponds to the output of the Memory block from the memory stage, and 3 corresponds to the result of the write back stage.

Overall, the Register File and MUXes play crucial roles in managing register operations, data flow, and selection of appropriate sources and outputs at different stages of the instruction execution cycle.

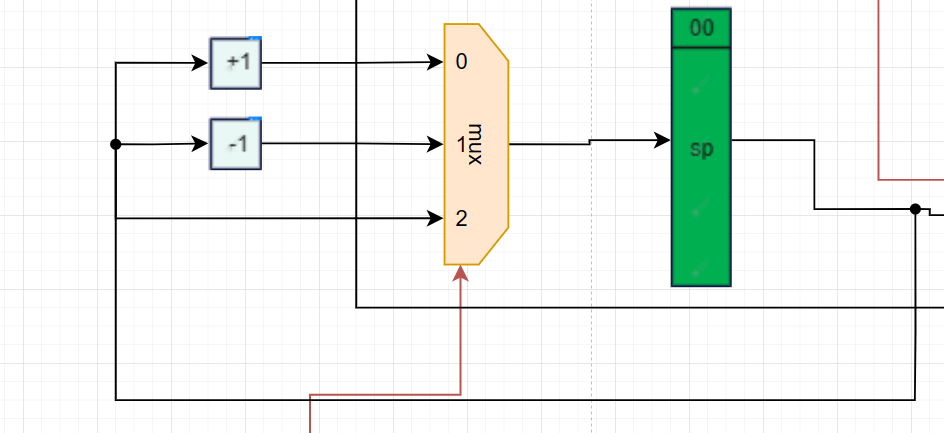


Figure 4: Stack pointer

The figure shows the presence of the Stack Pointer (SP) as the second part. The output of the SP is connected to the stack in the Memory component. A multiplexer (MUX) controls the behavior of the SP. When the select value of the MUX is 0, it triggers a "PUSH" operation, indicating that data is being added to the stack. Conversely, when the select value is 1, it triggers a "POP" operation, indicating that data is being removed from the stack. If the select value is 2, it implies that there is no change to the SP.

The SP plays a crucial role in managing the stack in memory, which is typically used for storing temporary data and managing function calls and returns. The MUX associated with the SP controls whether data is added to or removed from the stack, or if there is no change to the SP at that particular stage of the instruction execution cycle.

## 1.3. execute stage

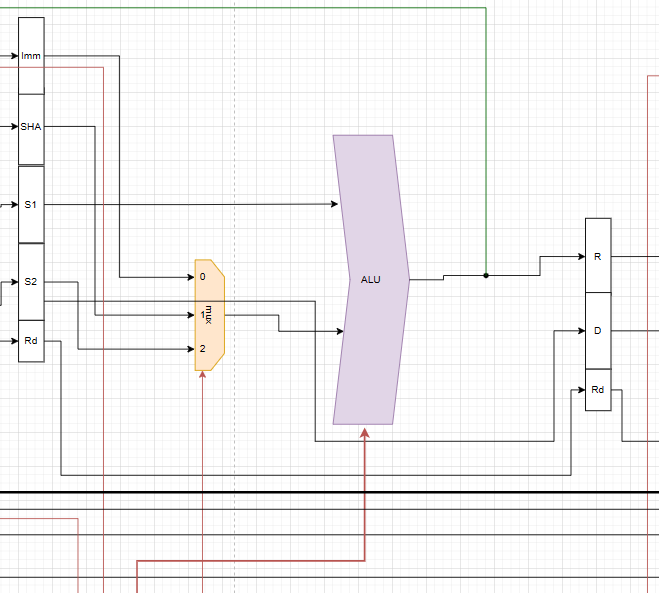


Figure 5: Execute stage

The figure highlights the primary component in this stage, which is the Arithmetic Logic Unit (ALU). The ALU has two inputs: the first input comes from the source register S1, while the second input comes from the multiplexer (MUX).

The MUX is responsible for selecting the appropriate input for the ALU. It operates as follows: when the select value is 0, the immediate value is chosen as the input. If the select value is 1, the shift immediate amount (SHA) is selected as the input. Finally, when the select value is 2, the second source (S2) is chosen as the input for the ALU.

The ALU performs arithmetic and logical operations on the inputs it receives, generating an output that is utilized in subsequent stages of the instruction execution cycle. The MUX allows for flexibility in selecting the appropriate input based on the specific operation being performed by the ALU.

## 1.4. Memory Stage and Write Back Stage

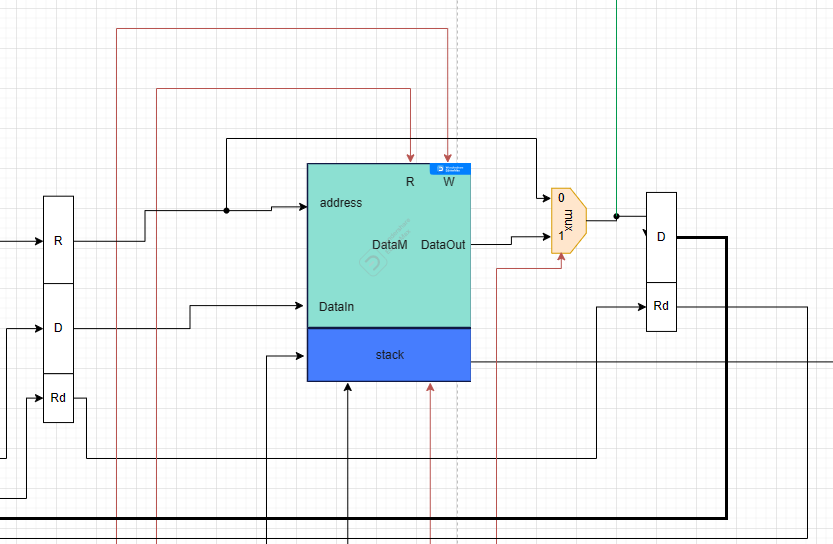


Figure 6: Memory Stage and Write Back Stage

The main component depicted is the Memory data block, which also encompasses the Stack. The Memory data block has four inputs: address and Datain, each consisting of 32 bits, and R and W which function as flags for read and write operations, respectively. It has one output. On the other hand, the Stack has three inputs: one connected to the NPC (Next PC) buffer to retrieve the return address, the second one from the Stack Pointer (SP), and the last one acts as a flag connected to the Stack Pointer condition (SpCon). The Stack also has one output.

The multiplexer (MUX) situated next to the memory block performs a selection based on the following: 0 indicates the output from the memory data, while 1 signifies the result obtained from the execution stage.

Furthermore, there are two additional buffers present. The first buffer, labeled as D, corresponds to the write-back stage and contains the data to be written onto the register. The second buffer, denoted as Rd, represents the address of the destination register where the write operation occurs.

# 2. Building the State Machine Diagram with Control signals generation

## 2.1 Control signals generation

### 2.1.1 main control

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ins | RigW | Rsor | Alusor | MemR | MemW | DWD | EXT1 |
| AND | 1 | 0 | 2 | 0 | 0 | 0 | X |
| ADD | 1 | 0 | 2 | 0 | 0 | 0 | X |
| SUB | 1 | 0 | 2 | 0 | 0 | 0 | X |
| CMP | 0 | 0 | 2 | 0 | 0 | 0 | X |
| ANDI | 1 | X | 1 | 0 | 0 | 0 | 0 |
| ADDI | 1 | X | 1 | 0 | 0 | 0 | 1 |
| LW | 1 | X | 1 | 1 | 0 | 1 | 1 |
| SW | 0 | 1 | 1 | 0 | 1 | X | 1 |
| BEQ | 0 | 1 | 2 | 0 | 0 | X | 1 |
| J | 0 | X | X | 0 | 0 | X | X |
| JAL | 0 | X | X | 0 | 0 | X | X |
| SLL | 1 | X | 0 | 0 | 0 | 0 | X |
| SLR | 1 | X | 0 | 0 | 0 | 0 | X |
| SLLV | 1 | 0 | 2 | 0 | 0 | 0 | X |
| SLRV | 1 | 0 | 2 | 0 | 0 | 0 | X |

Truth table of Main Control

RigW = (CMP + SW + BEQ + J + JAL)~.

Rsor = (SW + BEQ).

ALusor(bit0) = (ANDI + ADDI + LW + SW).

ALusor(bit1) = (AND + ADD + SUB + CMP + BEQ + SLLV + SLRV).

MemR = LW.

MemW = SW.

DWD = LW.

EXT1 = (ADDI + LW + SW + BEQ).

### 2.1.2 ALU table

|  |  |  |
| --- | --- | --- |
| TYPE | FUNC | OP |
| R | AND | 010 |
| R | ADD | 000 |
| R | SUB | 001 |
| R | CMP | 001 |
| I | ANDI | 010 |
| I | ADDI | 000 |
| I | LW | 000 |
| I | SW | 000 |
| I | BEQ | X |
| J | J | X |
| J | JAL | X |
| S | SLL | 101 |
| S | SLR | 110 |
| S | SLLV | 101 |
| S | SLRV | 110 |

ALU table

### 2.1.3 SP control and PC control tables

|  |  |  |
| --- | --- | --- |
| Inst&Stop | SpSor | Push |
| Jal | 0 | 1 |
| S=1 | 1 | 0 |
| S=0 | 2 | 0 |

SP control table

|  |  |
| --- | --- |
| Inst&Stop | PcSor |
| J | 2 |
| Branch | 1 |
| S=1 | 3 |
| else | 0 |

PC control table

### 2.1.4 Forwarding code

If ( (RS1 == Rd2) and (EX.RegWr)) ForwardS1 = 1

Else if ( (RS1== Rd3) and (MEM.RegWr)) ForwardS1 = 2

Else if ((RS1 == Rd4) and (WB.RegWr)) ForwardS1 = 3

Else ForwardS1 = 0

If ( (RS2 == Rd2) and (EX.RegWr)) ForwardS2 = 1

Else if ( (RS2 == Rd3) and (MEM.RegWr)) Forward S2 = 2

Else if ( (RS2 == Rd4) and (WB.RegWr)) Forward S2 = 3

Else Forward S2 = 0

Forward

if ((EX.MemRd == 1) // Detect Load in EX stage and (ForwardS1==1 or ForwardS2==1)) Stall // RAW Hazard

### 2.1.5 Control hazard code

PC(Control hazard)

if ((BEQ && Zero))

{ Jmp=0; Br=1;Stop=0; Kill1=1; }

else if (J)

{ Jmp=1; Br=0;Stop=0; Kill1=1; }

Else if (Stop)

{ Jmp=0; Br=0;Stop=1; Kill1=1; }

Else:

{ Jmp=0; Br=0;Stop=0; Kill1=0; }

## 2.2 state diagram

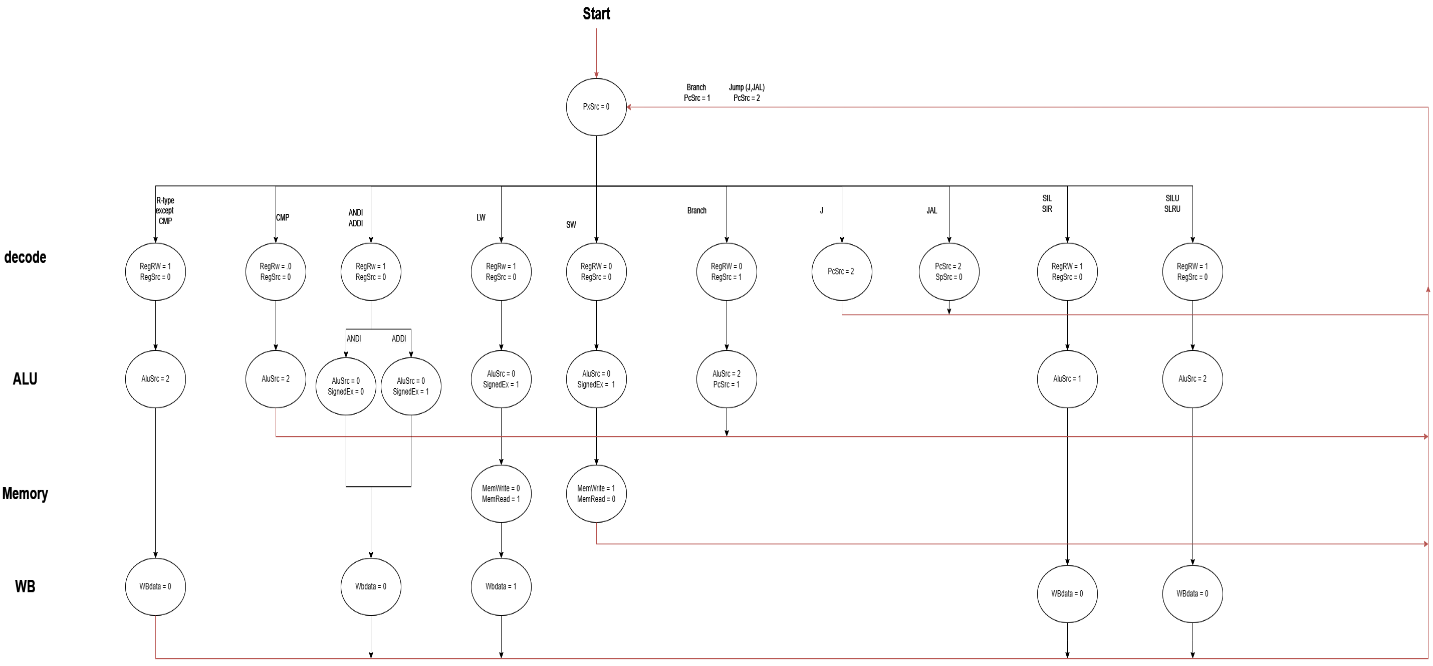


Figure 7: state machine diagram

A systematic approach was followed to construct the state machine diagram for the stages of the Pipelined processor. The process involved several steps.

Firstly, the operations to be executed in each stage and the corresponding conditions were thoroughly examined. This analysis helped identify the specific states required for the state machine diagram.

Next, the transitions between these states were defined. These transitions were triggered by control signals generated by the processor. For instance, the transition from the instruction fetch stage to the instruction decode stage occurred when the "state number" value indicated that the instruction had been fetched from memory and was ready for decoding.

Moreover, the actions to be performed during each state were defined. Examples of such actions include incrementing the program counter (PC) or writing to the register file.

Lastly, the accuracy of the state machine diagram was ensured by testing the processor's operation and comparing the results with the expected behavior. This verification step confirmed that the constructed diagram correctly represented the functioning of the Pipelined processor.

# 3. Testing the Modules

## 3.1 First Test

We have a code that we use to test our machine. The code helps us to ensure that the machine is working properly and that it is meeting our expectations.

reg [31:0] Mem [0:6] = '{

32'b00000000000000000000000000000000, // INITIAL INSTRUCTION TO GET READY

32'b00010000000010100000000000001100, LW R5, R0(LL)

32'b00000001010011000000000100001110, SLS R6, R5

32'b00001001110011100110000100001000, ADD R7, R7, R6

32'b00011001010100000110000100001110, SLRV R8, R5, R6

32'b00011000000100000000000000001100, SW R8, R0(LL)

32'b00010000000100100000000000001100}; LW R9, R0(LL)

**Fetch decode ALU Memory WB**

**C1 LW**

**C2 SLS LW**

**C3 ADD SLS LW**

**C4 ADD SLS \_\_\_ LW**

**C5 SLRV ADD SLS \_\_\_ LW**

**C6 SW SLRV ADD SLS \_\_\_**

**C7 LW SW SLRV ADD SLS**

**C8 LW SW SLRV ADD**

**C9 LW SW SLRV**

**C10 LW SW**

**C11 LW**

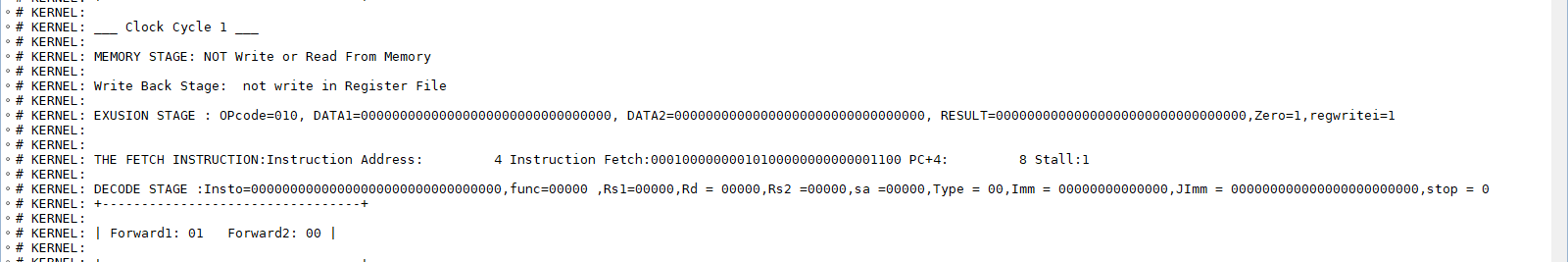


Figure 8: Test1\_cycle1

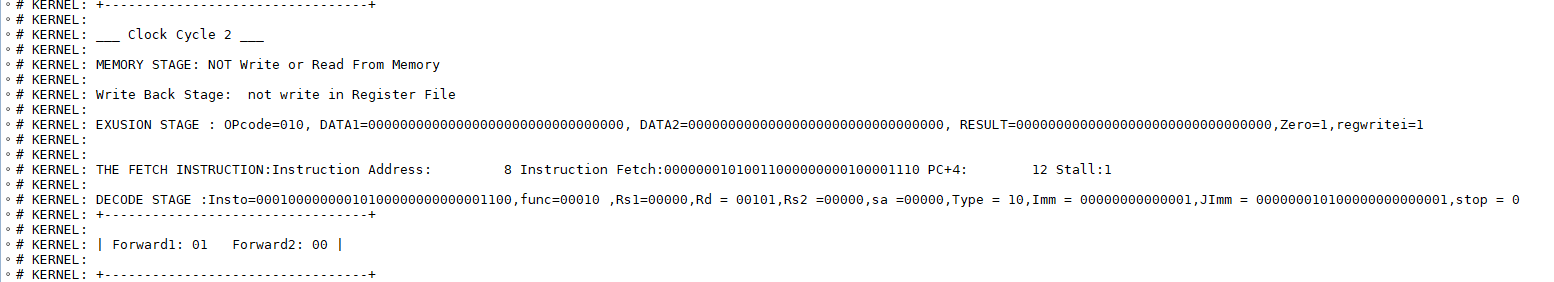
At the first cycle, we can see the LW instruction(32'b00010000000010100000000000001100) is getting into the Fetch stage, which it's the first instruction in Instruction Memory.

Figure 9: Test1\_cycle2

The LW instruction has moved to the next stage (decode stage), while we got a new instruction SLS instruction (32'b00000001010011000000000100001110) in the fetch stage.

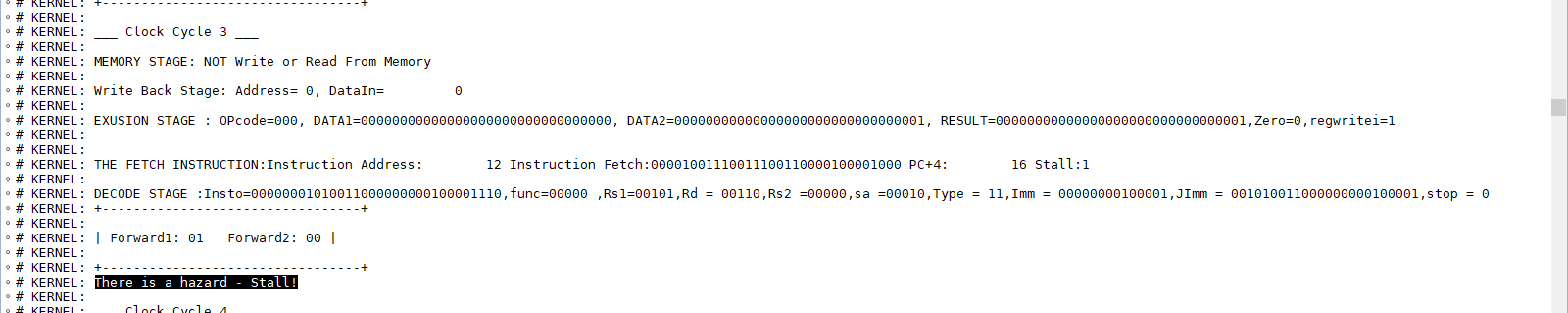


Figure 10: Test1\_cycle3

The LW instruction has moved to the Execution stage, while the SLS moved to the decode stage, and we got a new instruction ADD (32'b00001001110011100110000100001000) in the fetch stage. As we can see, the machine got a hazard between LW and SLS with the share register (R5) 'Read after Write.'

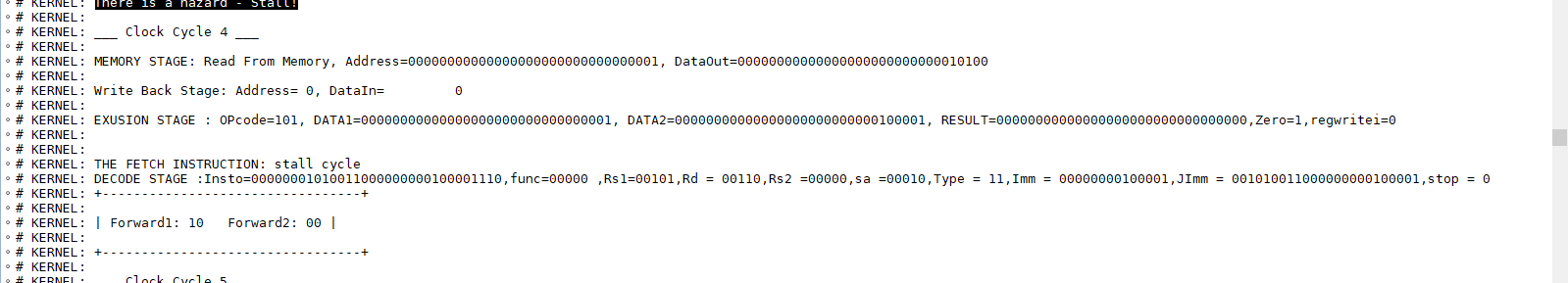


Figure 11:Test1\_cycle4

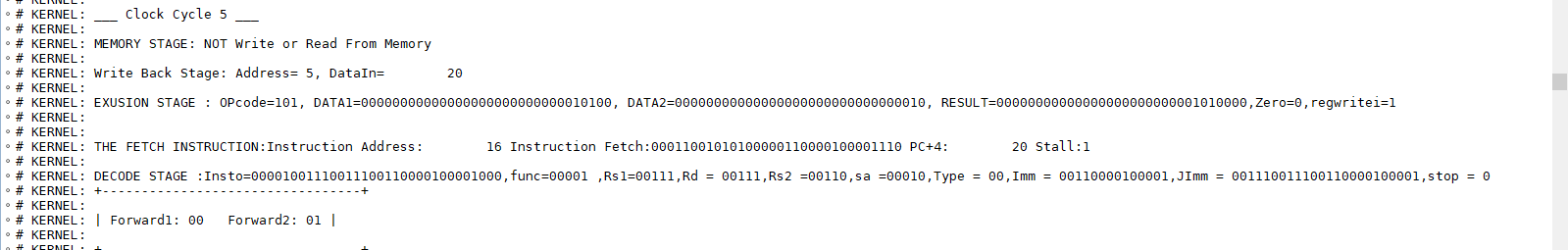
Because of the hazard, two instructions, ADD and SLS, should stay at their previous stages during the last cycle; as a reason, we got a stall cycle in the Execution stage while the LW instruction will move on to the Memory stage.

Figure 12: Test1\_cycle5

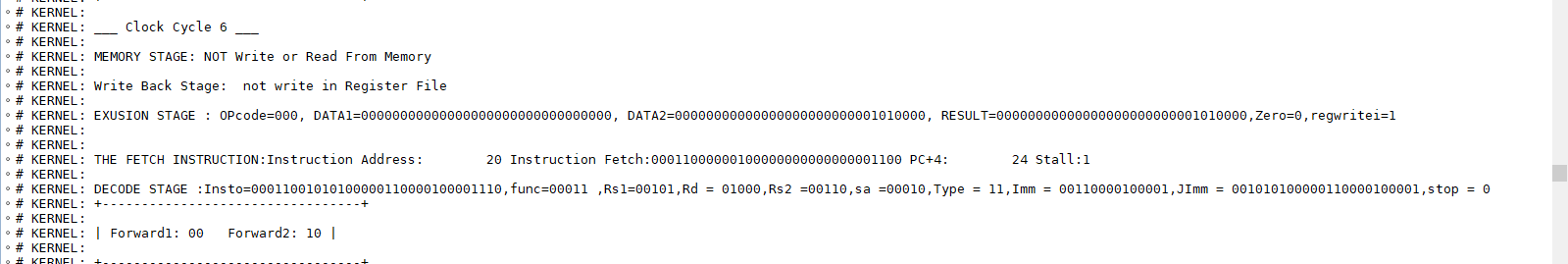
The previous instructions will continue moving to the next stage, and so on the stall cycle will do, we will get a new instruction that will show in fetch stage SLRV (32'b00011001010100000110000100001110) 

Figure 13: Test1\_cycle6

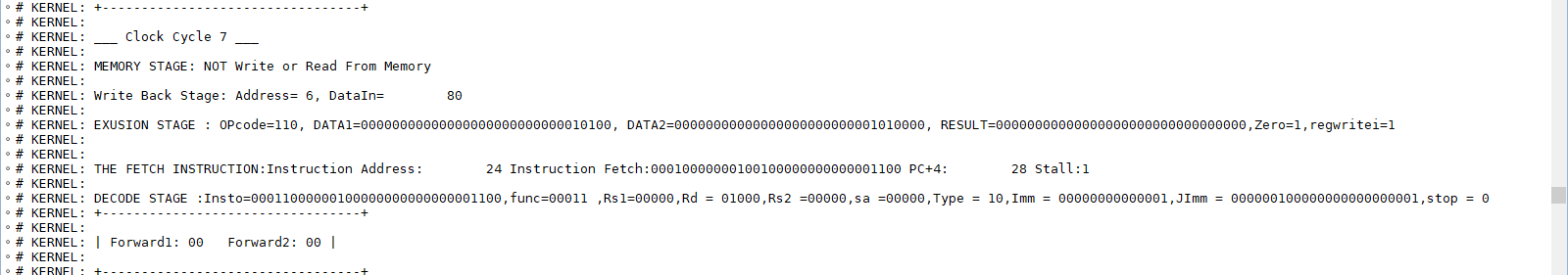
The instruction will keep moving to the next stage, and the LW instruction has ended, new instruction will show in stage Fetch SW (32'b00011000000100000000000000001100).

Figure 14: Test1\_cycle7

The stall cycle will end here, and we got the last instruction LW (32'b00010000000100100000000000001100). The instructions will keep moving in this way:

Fetch decode ALU MEM WB

LW 🡪 SW 🡪 SLRV 🡪 ADD 🡪 SLS

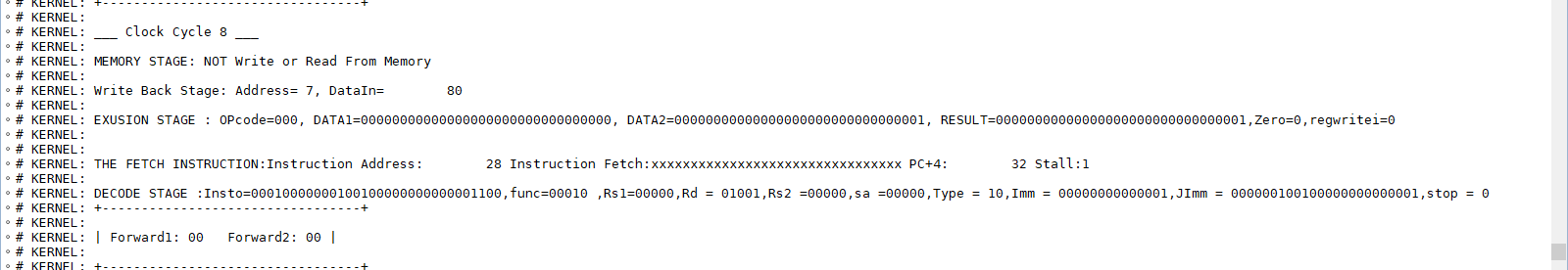


Figure 15: Test1\_cycle8

Fetch decode ALU MEM WB

LW 🡪 SW 🡪 SLRV 🡪 ADD

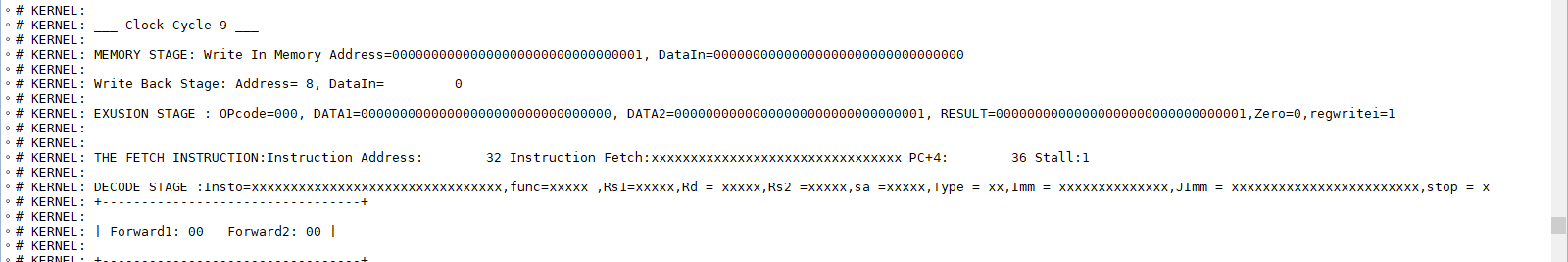


Figure 16: Test1\_cycle9

Fetch decode ALU MEM WB

LW 🡪 SW 🡪 SLRV

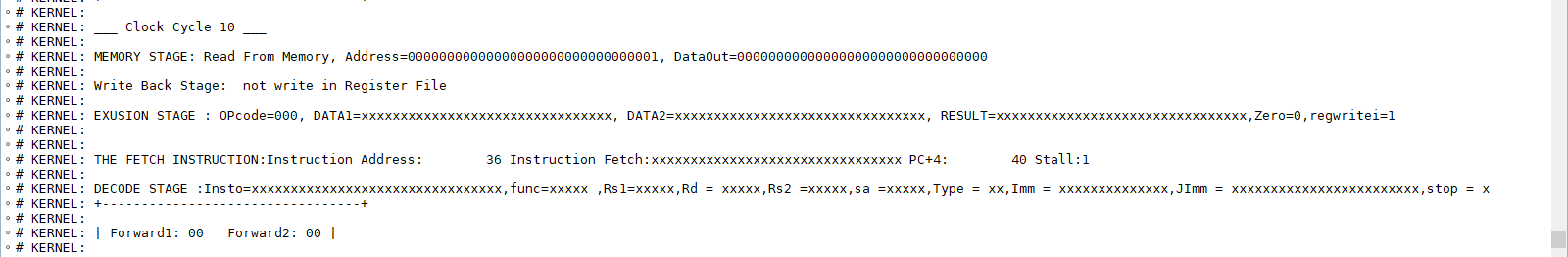


Figure 17: Test1\_cycle10

Fetch decode ALU MEM WB

LW 🡪 SW

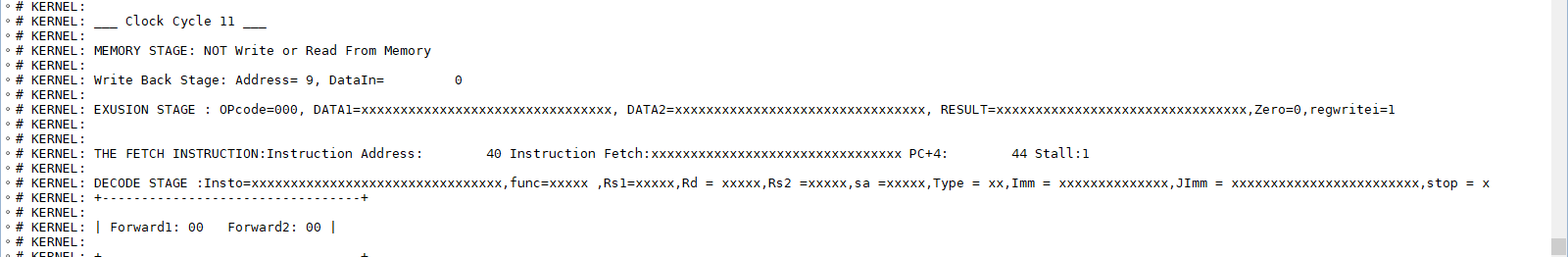


Figure 18: Test1\_cycle11

Fetch decode ALU MEM WB

LW

And here we reached the last cycle. All cycles needed to do this code is 11 cycles in the pipe

Number of cycles = (number of instructions – 1) + number of stages + stall cycle

Number of cycles = 5 + 5 + 1

Number of cycles = 11

## 3.2 Second Test

We have a code that we use to test our machine. The code helps us to ensure that the machine is working properly and that it is meeting our expectations.

reg [31:0] Mem [0:6] = '{

32'b00000000000000000000000000000000, // INITIAL INSTRUCTION TO GET READY

32'b00010000000001000000000000001100, LW R2,0(LL)

32'b00001000000000000000000001100010, JAL AA

32'b00010000000001100000000000001100, LW R3,0(LL)

32'b00001000010001000011000010100000, ADD R1, R2, R3

32'b00000111111111111111111111100010, J for ever

32'b00001000010000100000000010100100, ADDI R1, R1,20

32'b00011000010000100010000000000000, CMP R1, R2

32'b00100000010001011111111110100100, BEQ R1, R2 0(AA)

32'b00010000010000100010000000000001}; SUB R1, R1, R2 🡪 STOP = 1

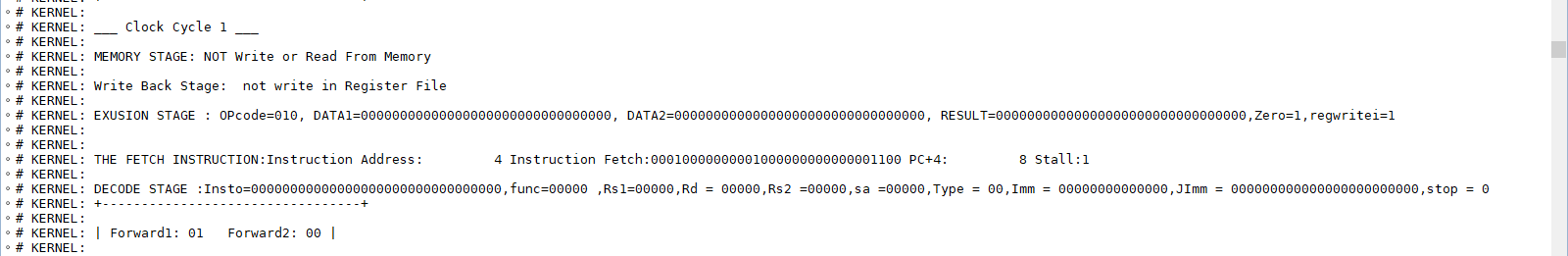


Figure 19: Test2\_cycle1

At the first cycle, we can see the LW instruction (32'b 00010000000001000000000000001100) is getting into the Fetch stage, which it's the first instruction in Instruction Memory.

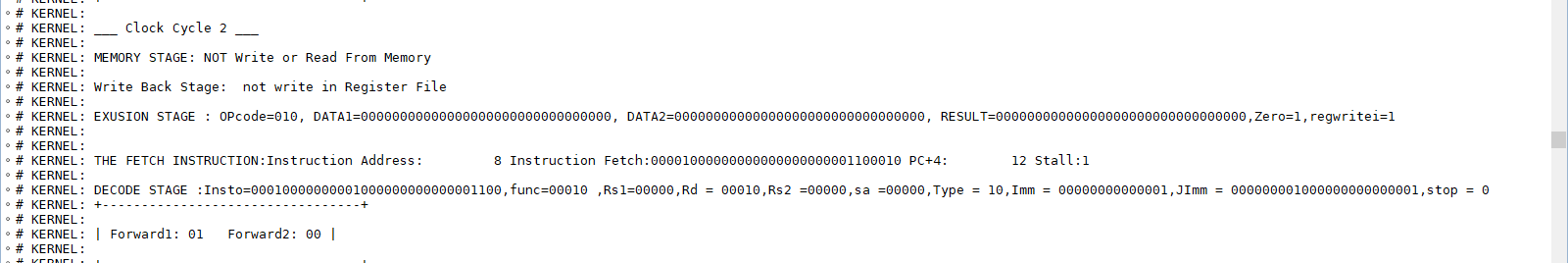


Figure 20: Test2\_cycle2

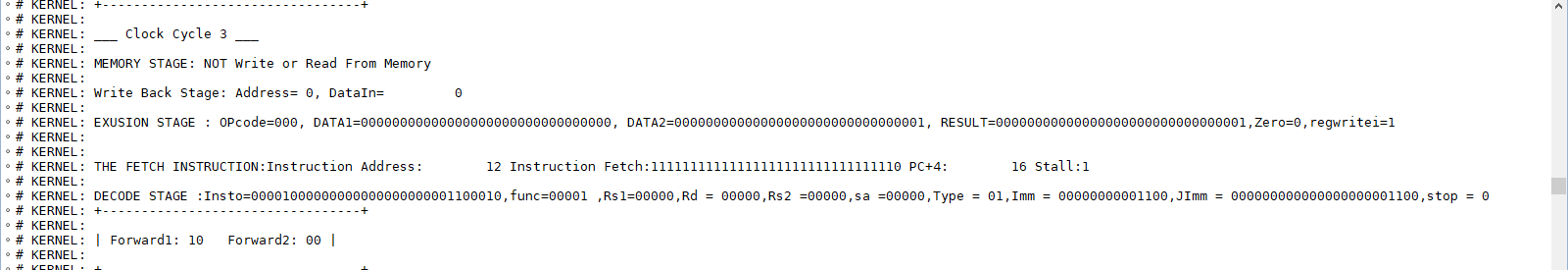
At the next cycle, the LW instruction has moved on to the next stage, the "decode stage," and a new one has come to the Fetch stage, which is JAL (32'b 00001000000000000000000001100010).

Figure 21: Test2\_cycle3

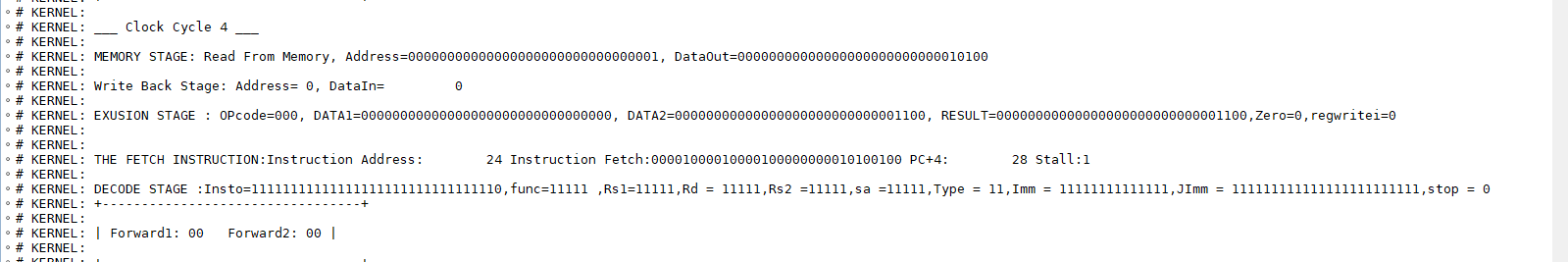
Next and, because of the JAL, which was in the fetch stage before, the machine couldn’t know the address in the JAL instruction; this caused a stall cycle in fetch, the two instructions moved to the next stage.

Figure 22: Test2\_cycle4

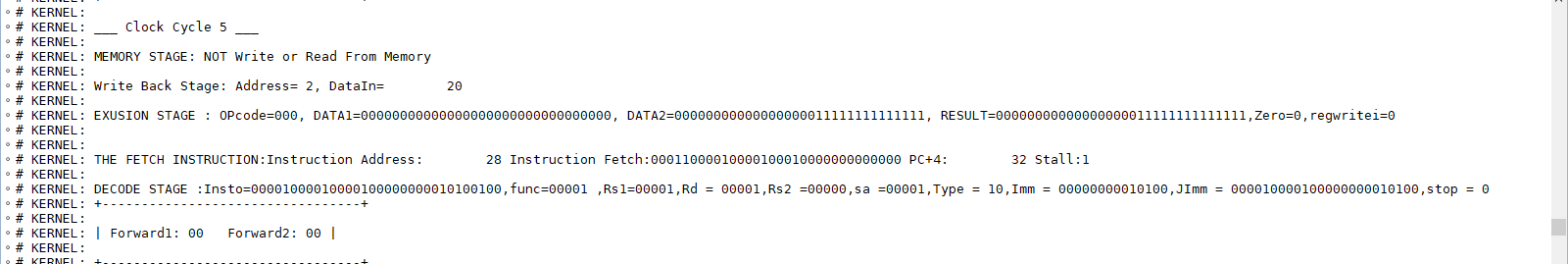
A new instruction has come in the fetch stage, ADDI (32’b 00001000010000100000000010100100). The two instructions with the stall cycle keep moving to the next stage.

Figure 23: Test2\_cycle5

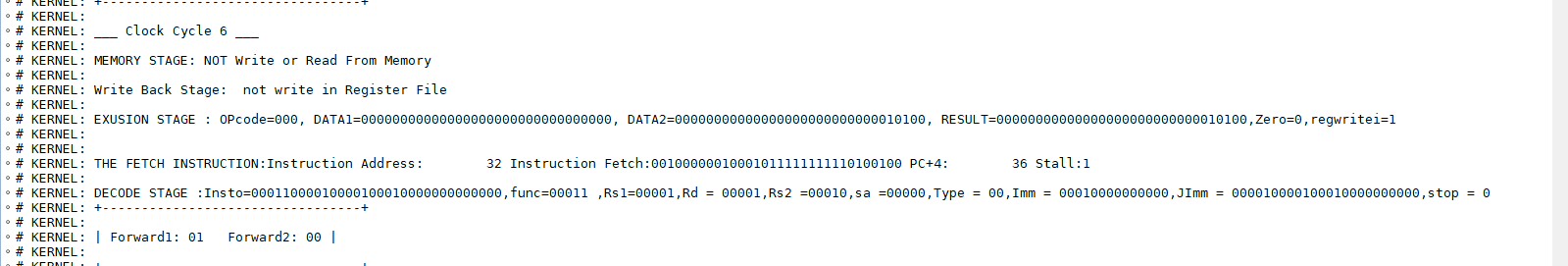
A new instruction has come too in the fetch CMP (32’b 00011000010000100010000000000000). As a pipe is linear, the other instructions keep moving to the next stage.

Figure 24: Test2\_cycle6

In this cycle the LW instruction has done its job, new instruction has reached fetch stage BEQ(32’b 00100000010001011111111110100100).

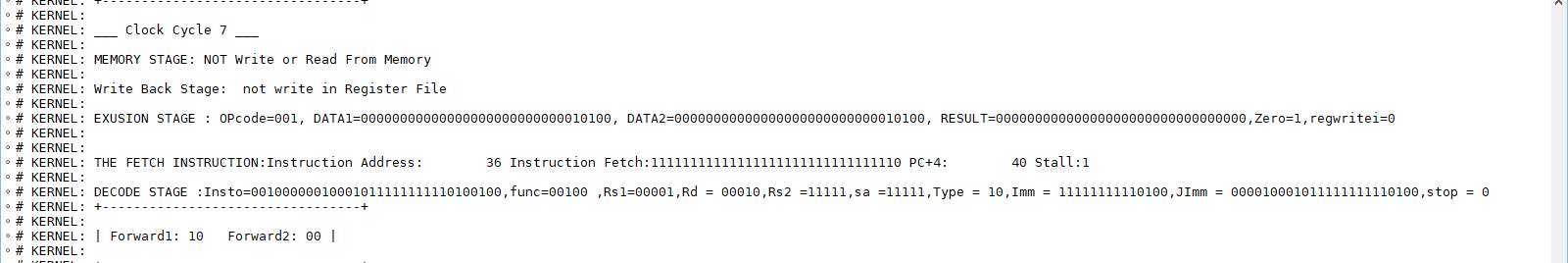


Figure 25: Test2\_cycle7

In cycle seven and after the BEQ has reached the decode, the machine is working on (**not taking**) while

 R1 == R2 == 20, a stall cycle will appear in the fetch stage.

((Go back to the loop))

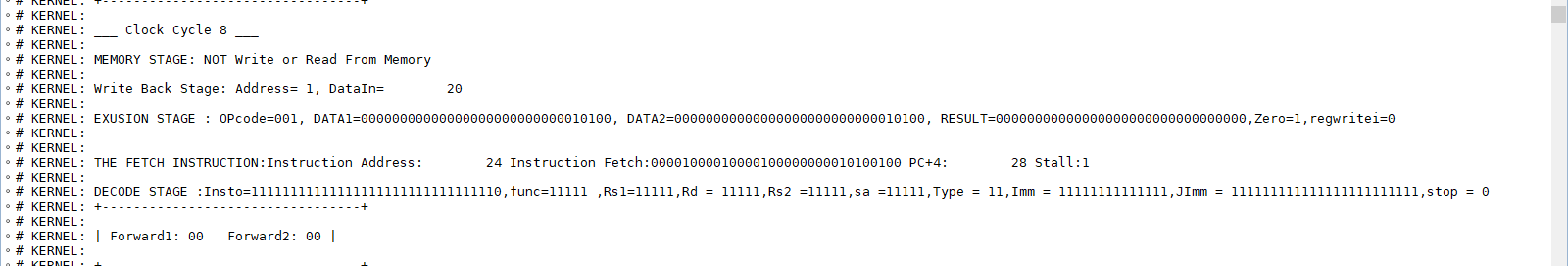


Figure 26: Test2\_cycle8

New instruction has come, ADDI which it’s the first line in the loop.

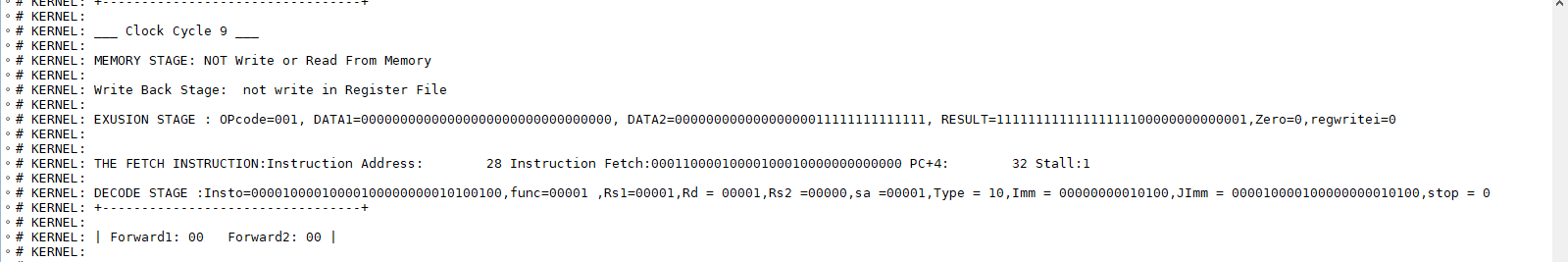


Figure 27: Test2\_cycle9

In the second line in the loop CMP in the fetch stage, the other instructions keep moving linearly.

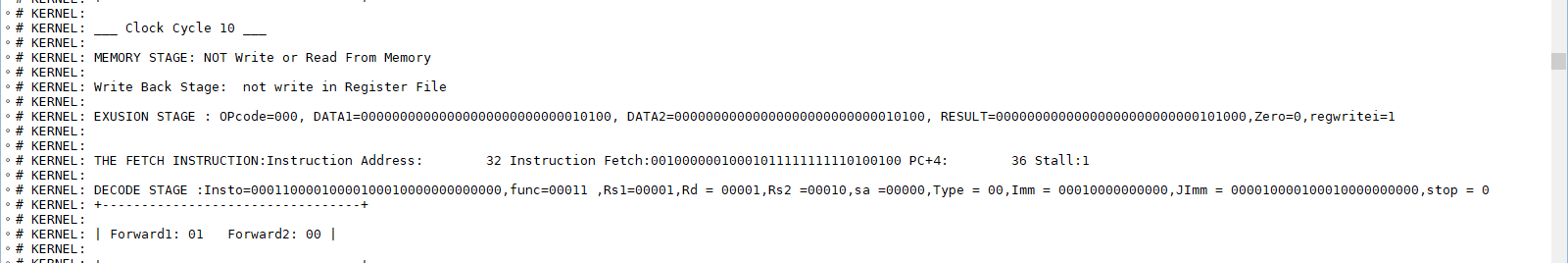


Figure 28: Test2\_cycle10

Here we have reached BEQ again, NOT TAKING, R1 != R2, then it will break the loop.

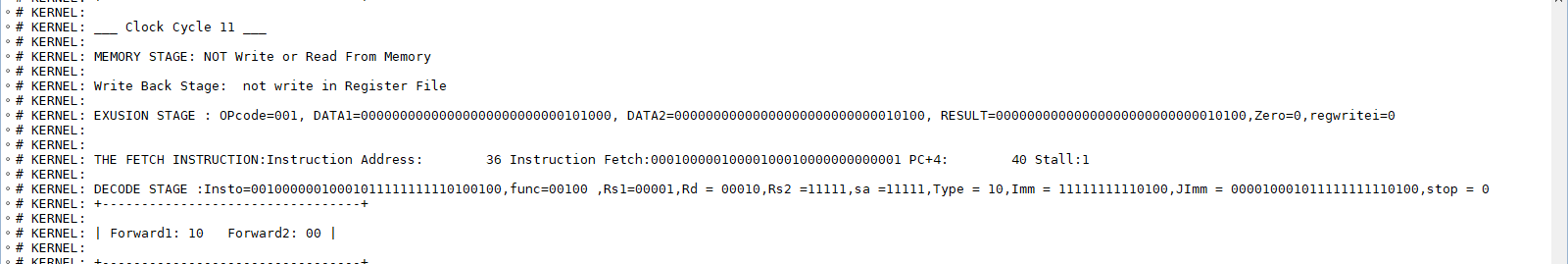


Figure 29: Test2\_cycle11

At cycle 11, new instruction has come to the fetch stage, SUB

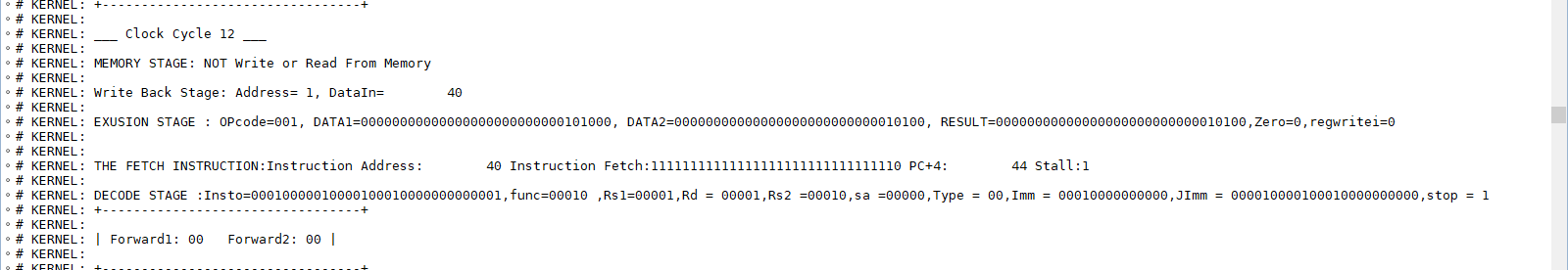
(32’b 00010000010000100010000000000001).

Figure 30: Test2\_cycle12

At cycle 12, as SUB is the last instruction in the loop, it has the stop flag = 1, then it will jump to the return address, while the default behavior is pc + 4, then it will give a stall cycle in fetch stage.

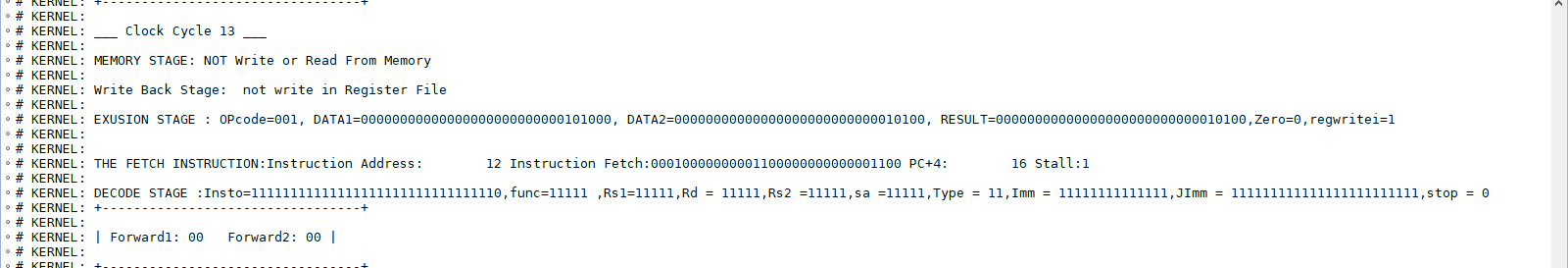


Figure 31: Test2\_cycle13

A new instruction has come to the fetch stage, LW (32’b 00010000000001100000000000001100). Other instructions keep moving linearly.

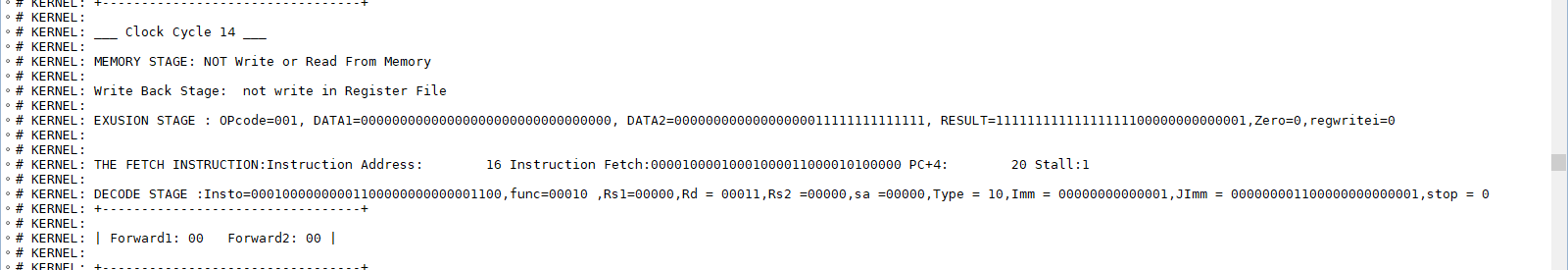


Figure 32: Test2\_cycle14

ADD instruction has come to fetch stage (32’b 00001000010001000011000010100000).

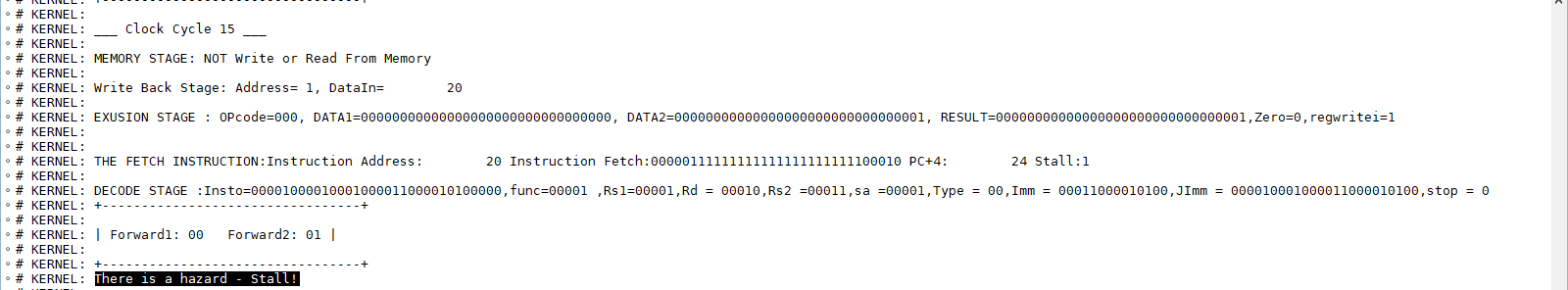


Figure 33: Test2\_cycle15

In this cycle, the program got a hazard because the two instructions, LW and ADD, share the same register R3; the LW wants to write on it, while the ADD instruction wants to read it**. Read after write. This** will cause a stall cycle in the ALU stage at the next cycle.

New instruction has reached in the fetch stage, which is jump to the same location

Pc next = PC + 4 – 4 = PC

The instructions will keep moving in this way:

Fetch decode ALU MEM WB

J 🡪 ADD 🡪 LW 🡪 \_\_\_ 🡪 SUB

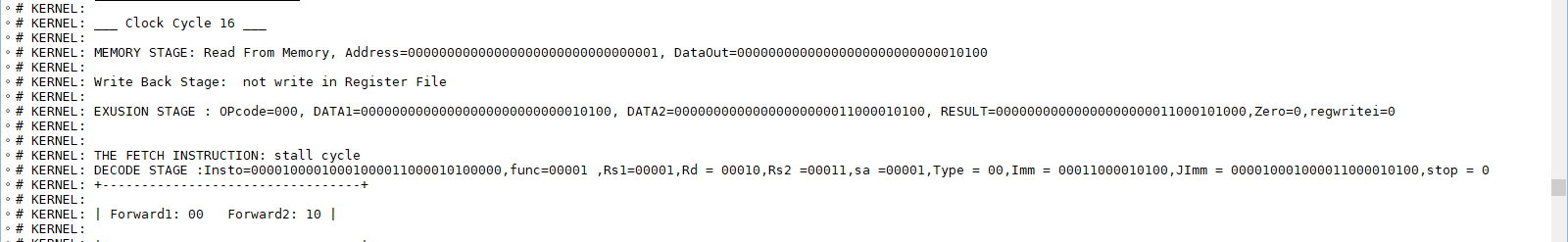


Figure 34: Test2\_cycle16

Fetch decode ALU MEM WB

\_\_\_ 🡪 J 🡪 ADD 🡪 LW 🡪 \_\_\_

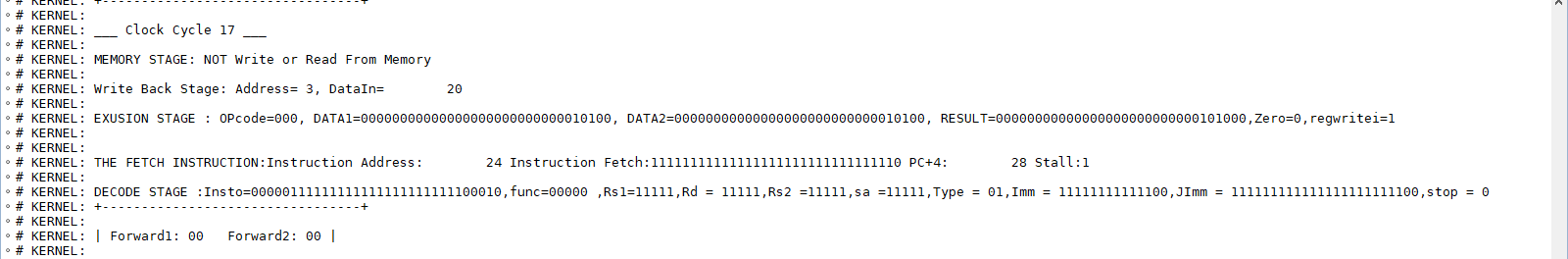


Figure 35: Test2\_cycle17

Fetch decode ALU MEM WB

J 🡪 \_\_\_ 🡪 J 🡪 ADD 🡪 LW

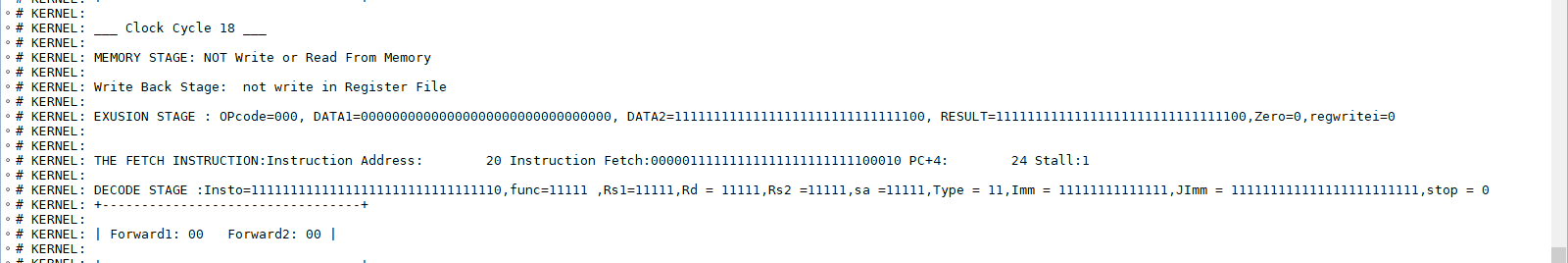


Figure 36:Test2\_cycle18

Fetch decode ALU MEM WB

\_\_\_ 🡪 J 🡪 \_\_\_ 🡪 J 🡪 ADD

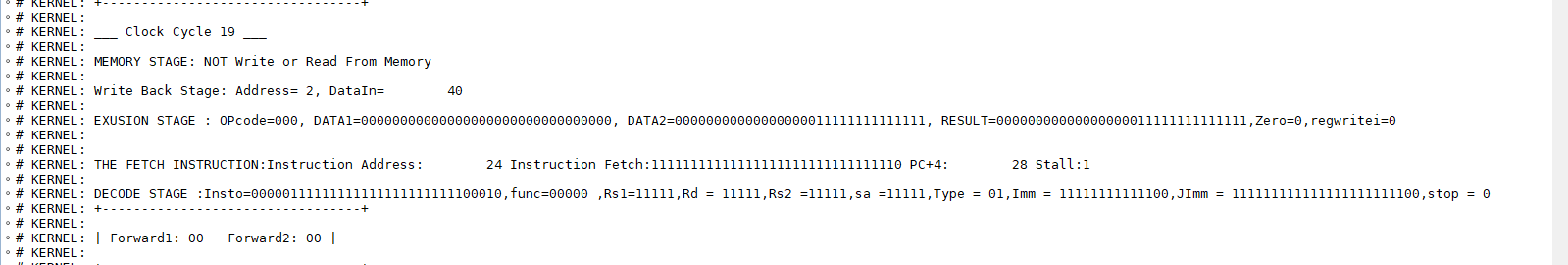


Figure 37:Test2\_cycle19

Fetch decode ALU MEM WB

J 🡪 \_\_\_ 🡪 J 🡪 \_\_\_ 🡪 J

# 4. Conclusion

In conclusion, an efficient MIPS CPU was constructed using the pipelined approach. Throughout the project, a more in-depth understanding of how the MIPS CPU functions with the pipelined approach was obtained while designing the various components. The pipelined approach was chosen due to its efficiency benefits compared to the single-cycle approach, as it allows for concurrent execution of multiple instructions by dividing them into separate pipeline stages. Moreover, valuable insights were gained on connecting the modules using code and performing thorough testing. Looking ahead, future work could focus on optimizing the final module using the pipelined approach to reduce the cycles per instruction (CPI) in the CPU.

# 5. Appendix

All the work is attached below:

<https://drive.google.com/drive/folders/19xbbsy7xNCyeo8WGs_iSkMmjjBZOQNjK?usp=sharing>