

Lab Exercise-3

Vaibhav Shatalwar, DESE, IISc

ABSTRACT

The objective of this Lab exercise is to implement a Pipelined RISC-V Processor with Instruction Cache.

I PROCESSOR DESIGN

Design and Implement a 5-stage Pipelined RISC-V processor on Digilent BASYS3 FPGA Board. Target Device is Xilinx Artix-7 XC7A35T- ICPG236C (Family Artix-7, Part XC7A35T, Package CPG236, Speed Grade -1).

The Processor is a 32-bit RISC-V Processor. The processor should support basic arithmetic-logic instructions, branch instructions, `lui`, load-store instructions, and jump instructions. Atomic instructions, `auipc` and CSR instructions need not be supported. Implement hazard detection and forwarding (from EX, MEM, and WB stage outputs). Implement the Stall unit for load, and branch instructions.

1.1 Design Requirements

- 5-stage pipeline: Fetch, Decode, Execute, Memory, and Write-back
- Support for basic arithmetic-logic instructions, branch, `lui`, and load-store instructions
- Hazard detection and forwarding logic
- Stall unit for load and branch instructions
- Instruction cache implementation (direct-mapped with a block size of four words)
- Block RAM for main instruction memory
- No burst transfer for cache filling
- No support for `atomic`, `auipc`, and CSR instructions
- Aligned transfer for word, half-word, and byte data
- No data cache, data access via data memory

1.2 Processor Testing

Test the processor with a suitable algorithm, using a procedure call. The program can be written in C, and the corresponding assembly or binary code should be used for testing.

1.3 Design Flow

1. Design the Datapath block schematic.
2. HDL coding for the RISC-V processor.
3. Implement hazard detection and forwarding units.
4. Implement stall logic for load and branch instructions.
5. Create the instruction cache using block RAM on the FPGA.
6. Use IO Constraints and Timing Constraints to optimize performance.
7. Perform Timing Analysis and Timing simulation with a proper testbench.
8. Generate binary code from C program and load it into memory.
9. Implement the design on BASYS3 FPGA Board.

1.4 Submission Requirements

Submit the following:

- Block schematic of the Datapath design.
- State diagram of the controller (if any).
- Source Verilog codes.
- IO and Timing Constraint files.
- Timing Report.
- Resource utilization.
- Software code for testing.

II INSTRUCTION SET ARCHITECTURE AND INSTRUCTION FORMAT

2.1 R-Type Instructions (Arithmetic and Logic)

Operation: $Rd = Rs1 \text{ op } Rs2$

Assembly: *Instr Rd Rs1 Rs2*

Format:

Funct7	Rs2	Rs1	Funct3	Rd	Opcode
31-25	24-20	19-15	14-12	11-7	6-0

For all R-type instructions, Opcode = 0110011

Instruction	Operation	Funct3	Funct7
ADD	$Rd = Rs1 + Rs2$	000	0000000
SUB	$Rd = Rs1 - Rs2$	000	0100000
AND	$Rd = Rs1 \& Rs2$	111	0000000
OR	$Rd = Rs1 Rs2$	110	0000000
XOR	$Rd = Rs1 \oplus Rs2$	100	0000000
SLL	$Rd = Rs1 \ll Rs2$	001	0000000
SRL	$Rd = Rs1 \gg Rs2$	101	0000000
SRA	$Rd = Rs1 \ggg Rs2$	101	0100000

2.2 Load

Operation: $Rd = \text{mem}[Rs1 + \text{Imm}]$

Assembly: *Instr Rd Rs1 Imm*

Format:

Immediate	Rs1	Funct3	Rd	Opcode
31-20	19-15	14-12	11-7	6-0

For all load instructions, Opcode = 0000011

Instruction	Operation	Funct3
LW	Loads word	010
LH	Loads half word (sign extended)	001
LHU	Loads half word (zero extended)	101

LB	Loads byte (sign extended)	000
LBU	Loads byte (zero extended)	100

2.3 Store

Operation: $\text{mem}[\text{Rs1} + \text{Imm}] = \text{Rs2}$

Assembly: *Instr Rs2 Rs1 Imm*

Format:

Imm[12:6]	Rs2	Rs1	Funct3	Imm[5:1]	Opcode
31-25	24-20	19-15	14-12	11-7	6-0

For all store instructions, Opcode = 0100011

Instruction	Operation	Funct3
SW	Stores word	000
SH	Stores half word	001
SB	Stores byte	010

2.4 ADDI

Operation: $\text{Rd} = \text{Rs1} + \text{Imm}$

Assembly: *ADDI Rd Rs1 Imm*

Format:

Immediate	Rs1	Funct3	Rd	Opcode
31-20	19-15	14-12	11-7	6-0

For all immediate instructions, Opcode = 0010011

Instruction	Operation	Funct3
ADDI	Adds immediate to register	000

2.5 Branch Type Instructions

Operation: Goes to instruction at $\text{PC} + \text{Imm}_{\text{shifted}}$ if branching condition is satisfied.

Assembly: *Instr Rs1 Rs2 Imm*

Format:

Imm[12]	Imm[10:5]	Rs2	Rs1	Funct3	Imm[4:1]	Imm[11]	Opcode
31	30-25	24-20	19-15	14-12	11-8	7	6-0

Table 5: Format of Branch Type Instructions

For all branch instructions, Opcode = 1100011

Instruction	Operation	Funct3
BEQ	Goes to branch target if $\text{Rs1} = \text{Rs2}$	000
BNE	Goes to branch target if $\text{Rs1} \neq \text{Rs2}$	001

BLT	Goes to branch target if $Rs1 < Rs2$	100
BGE	Goes to branch target if $Rs1 \geq Rs2$	101

2.6 LUI

Operation: Loads an immediate value into the upper 20 bits of the destination register, setting the lower 12 bits to zero.

Assembly: *LUI Rd Imm*

Format:

Imm[31:12]	Rd	Opcode
31-12	11-7	6-0

For all LUI instructions, *Opcode* = 0110111.

2.7 JAL

Operation: Goes to instruction at $PC + Imm_{shifted}$ and stores the address of the next instruction (i.e., $PC + 4$) in the destination register.

Assembly: *JAL Rd Imm*

Format:

Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	Rd	Opcode
31	30-21	20	19-12	11-7	6-0

For all JAL instructions, *Opcode* = 1101111

2.8 JALR

Operation: Goes to instruction at $Rs1 + Imm$ and stores the address of the next instruction (i.e., $PC + 4$) in the destination register.

Assembly: *JALR Rd Rs1 Imm*

Format:

Imm[11:0]	Rs1[4:0]	Funct3	Rd[4:0]	Opcode
31-20	19-15	14-12	11-7	6-0

For all JALR instructions, *Opcode* = 1100111

For all JALR instructions, *Funct3* = 000

2.9 NOP

Operation: No operation

Assembly: *NOP*

Machine code: 0x0000001B

2.10 Halt

Operation: Stops further execution

Assembly: *HALT*

Machine code: 0x0000001C

Opcodes of Instructions

The following table lists the opcode values for different instructions:

Instruction	Opcode
Load (LD)	0000011
Store (ST)	0100011
Add Immediate (ADDI)	0010011
Branch if Equal (BEQ)	1100011
Branch if Not Equal (BNE)	1100011
Branch if Less Than (BLT)	1100011
Branch if Greater Than (BGE)	1100011
R-Type (Arithmetic and Logic)	0110011
Load Upper Immediate (LUI)	0110111
Jump and Link (JAL)	1101111
Jump and Link Register (JALR)	1100111
No Operation (NOP)	0000001B
Halt (HALT)	0000001C

III CPU LEVEL 1 DIAGRAM:

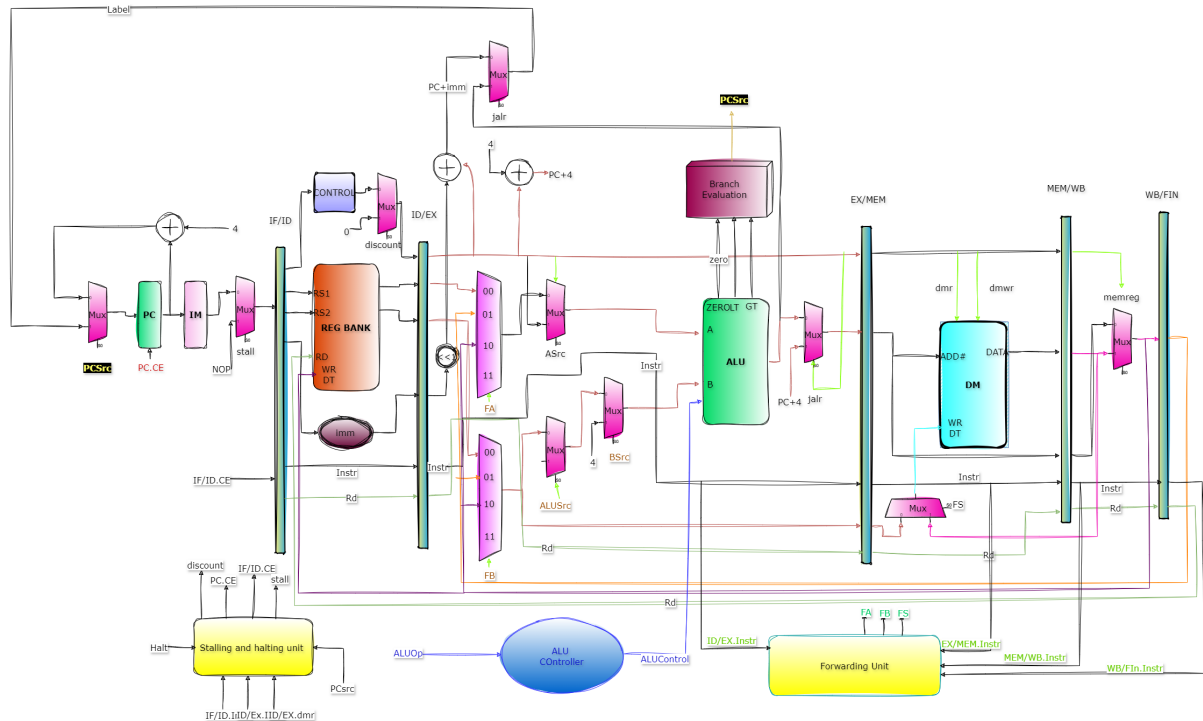


Figure 1: block diagram

3.1 Instruction Memory

Program Counter (PC) width = 32 bits

Address width = 32 bits

Bits 9 to 2 of the PC are used for pointing to the instructions. The lower 2 bits are not used so that the instructions are located at addresses which are multiples of 4. The address space is reduced for implementation purposes on the Basys 3 FPGA.

Instruction width = 32 bits

Memory space = 256 locations of 32 bits each.

This is realized using a distributed RAM implemented as a single port ROM in the FPGA.

3.2 Register File

A set of 32 registers x_0 to x_{31} , each 32 bits wide. This register file has 2 asynchronous read ports and 1 synchronous write port. Register x_0 always contains the value 0.

3.3 Immediate Generation Unit

A module that takes in the instruction, extracts the 12-bit immediate/offset field from it, and sign extends it to 32 bits. The Most Significant Bit (MSB) is sign extended to all the upper positions in the output.

3.4 Controller

Control Signal Purposes

The following table describes the purposes of various control signals:

Control Signal	Purpose
ALUOp[1:0]	Tells the ALU Decoder what control signal to generate/what field to look at to figure out which control signal to generate.
ASrc	Selects the first operand to be fed into the ALU, either the PC or register data.
ALUSrc	Selects the second operand to be passed on to the ALU, either register data or sign-extended immediate.
BSrc	Selects the second operand to be fed into the ALU, either the value passed by the ALUMux or 4.
memreg	Selects the data to be written to the register bank, i.e., either data from memory or computed result from ALU.
dmr	Enables the data memory to be read from.
dmw	Enables the data memory to be written to.
rwr	Enables write to happen to the register file.
jalm	Selects the data to be written to the register file in case of JALR instruction, i.e., selects PC+4. Also selects the appropriate branch target address in case of JALR instruction.
Branch	Asserted for branch type of instructions. Enables updating of PC with branch target address.
Jump	Asserted for jump type of instructions. Enables updating of PC with target address.
Halt	Asserted when HALT instruction is encountered. Disables further instructions filling into the pipeline.
lw, lh, lhu, lb, lbu	Generated when LOAD type instruction is encountered. Specify with what data the destination register has to be loaded, i.e., byte/half-word/word.
sw, sh, sb	Generated when STORE type instruction is encountered. Specify with what data the destination memory address has to be written with, i.e., byte/half-word/word.
PCSrc	Selects the result with which the PC needs to be updated, i.e., either the address of the next instruction (PC+4) or branch target address.

Table 7: Control Signal Purposes

IV ALU DECODER AND ALU

4.1 ALU

The ALU is a 16-bit unit that supports the following instructions:

- ADD
- SUB
- AND
- OR
- XOR
- NOT
- LSL (Logical Shift Left)
- LSR (Logical Shift Right)
- ASR (Arithmetic Shift Right)

In addition to the result, the ALU generates the following flags:

- Greater Than or Equal To (GE)
- Less Than (LT)
- Zero (Z)

4.2 ALU Decoder and Control

The control unit has two decoders:

1. **Main Decoder:** This decoder takes the opcode and issues the appropriate $ALUOp[1:0]$ signal to the ALU decoder.
2. **ALU Decoder:** Based on the function field and $ALUOp$ signal, the ALU decoder generates the $ALUControl[3:0]$ signal.

$ALUOp$ is generated as follows:

- For arithmetic operations, $ALUOp$ may specify the need for addition or subtraction.
- For logical operations, $ALUOp$ helps in selecting operations like AND, OR, XOR.
- For shift operations, $ALUOp$ will determine if a shift left or shift right is performed.

$ALUOp$ is generated as

Instruction	$ALUOp$	Operation
R-type	10	Look at func3/func7
I-type	00	ADD
Loads	00	ADD
Stores	00	ADD
Branches	01	Look at func3
LUI	11	Look at func3
JAL	00	ADD
JALR	00	ADD
NOP	11	Other/Do nothing
HALT	11	Other/Do nothing

Table 8: $ALUOp$ and Corresponding Operations

$ALUControl$ is generated as

Instruction	Func 3/Func 7	$ALUControl$
ADD	000/0000000	0010
SUB	000/0100000	0110
AND	111	0000
OR	110	0001
XOR	100	0011
SLL	001	0100
SRL	101/0000000	0101
SRA	101/0100000	0111
NOP	000/0000000	1010
HALT	000/0000000	1010

Table 9: $ALUControl$ Values for R-Type Instructions

Instruction	Func 3	$ALUControl$
BEQ	000	0110
BNE	001	0110
BLT	100	1000
BGE	101	1001

Table 10: $ALUControl$ Values for Branch Instructions

For all other instructions, $ALUControl = 0010$ (ADD)

V DATA MEMORY AND LOAD AND STORE HARDWARE

The data memory is implemented using a distributed RAM with the following specifications:

- **Write Width:** 32 bits
- **Write Depth:** 1024 locations
- **Byte Writable:** The memory locations are byte-writable.
- **Read and Write Access:** Both reads and writes are synchronous operations.

The store hardware generates appropriate write enable signals for the data memory based on the control signals 'sw', 'sh', and 'sb'. The write enable signals are determined as follows:

- **'sw' (Store Word):**
 - Generates a write enable signal for the entire 32-bit word.
 - Writes the 32-bit value to the specified memory location.
- **'sh' (Store Half-word):**
 - Generates write enable signals for two contiguous 16-bit half-words.
 - Writes the lower 16 bits to the lower address and the upper 16 bits to the higher address if the memory is byte-addressable.
- **'sb' (Store Byte):**
 - Generates a write enable signal for a single byte in the 32-bit word.
 - Writes the byte to the specified byte-addressable location.

The control signals 'sw', 'sh', and 'sb' select the appropriate write enable signals, ensuring the correct data is written to the memory based on the type of store operation.

Instruction	A1	A0	sw	sh	sb	WE0	WE1	WE2	WE3
SW	0	0	1	0	0	1	1	1	1
SH	0	0	0	1	0	1	1	0	0
SH	1	0	0	1	0	0	0	1	1
SB	0	0	0	0	1	1	0	0	0
SB	0	1	0	0	1	0	1	0	0
SB	1	0	0	0	1	0	0	1	0
SB	1	1	0	0	1	0	0	0	1

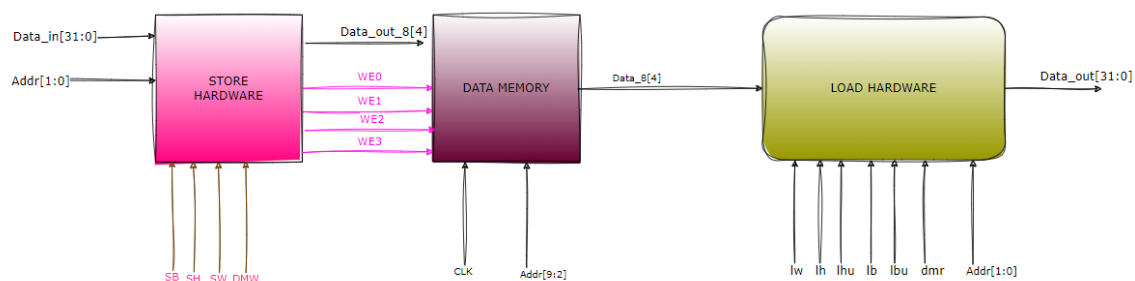


Figure 2: storage circuit logic

5.1 Load Hardware

The load hardware is situated in the Write-Back (WB) stage of the pipeline, as opposed to the MEM stage used for store hardware. This positioning is due to the synchronous read nature of the Distributed RAM used in the implementation. Consequently, all control signals for load operations are delayed until the WB stage.

In the WB stage, the load hardware is responsible for constructing the appropriate 32-bit word to be loaded into the CPU register. This construction is based on the control signals received, which determine the specific operation to be performed. The load hardware ensures that the data read from memory is correctly formatted and transferred to the register file, completing the data retrieval process.

The control signals used for load operations include those that determine the type of load instruction (e.g., byte, half-word, word) and manage the data path to ensure proper data retrieval and storage.

The following table summarizes the control signals and data formatting for various load instructions. The ‘Data’ field represents the data read from memory, with different bit widths and sign extension applied based on the instruction.

Instruction	A1	A0	LW	LH	LHU	LB	LBU	Data to Register
LW	0	0	1	0	0	0	0	Data[31:0]
LH	0	0	0	1	0	0	0	{16{Data[15], Data[15:0]}}
LH	0	1	0	1	0	0	0	{16{Data[31], Data[31:16]}}
LHU	0	0	0	0	1	0	0	{16{0}, Data[15:0]}
LHU	0	1	0	0	1	0	0	{16{0}, Data[31:16]}
LB	0	0	0	0	0	1	0	{24{Data[7], Data[7:0]}}
LB	0	1	0	0	0	1	0	{24{Data[15], Data[15:8]}}
LB	1	0	0	0	0	1	0	{24{Data[23], Data[23:16]}}
LB	1	1	0	0	0	1	0	{24{Data[31], Data[31:24]}}
LBU	0	0	0	0	0	0	1	{24{0}, Data[7:0]}
LBU	0	1	0	0	0	0	1	{24{0}, Data[15:8]}
LBU	1	0	0	0	0	0	1	{24{0}, Data[23:16]}
LBU	1	1	0	0	0	0	1	{24{0}, Data[31:24]}

VI FORWARDING UNIT

The forwarding unit generates appropriate control signals to forward data from subsequent stages to the input of the EX/MEM stages in case of a data hazard.

FA/FB Forwarding control signals are as follows:

FA/FB	Description
00	Input taken from IF.ID pipeline register (No forwarding)
01	Input taken from WB.FIN pipeline register
10	Input taken from MEM.WB pipeline register
11	Input taken from EX.MEM pipeline register

In the case where a load is followed by a store instruction, where $STORE.Rs2 = LOAD.Rd$, forwarding needs to be done from the MEM.WB pipeline register to the write data port of the data memory. The control signal FS facilitates this.

VII STALLING AND HALTING UNIT

The stalling and halting unit generates appropriate control signals to stall or halt execution.

In case of a LOAD instruction followed by a dependent instruction, only forwarding will not suffice. A 1-stall cycle needs to be introduced.

In case the branching condition is satisfied, 1 stall cycle and 1 NOP instruction need to be introduced into the pipeline.

The following table summarizes the conditions for stalling and halting, with the associated control signals.

Hazard	PCSrc	Halt	dmr ((IF/ID.RS1==ID/EX.Rd) && or (IF/ID.RS2==ID/EX.Rd))	Stall	Discont	PC_CE	IF_ID_CE
Data	0	0	1	0	1	0	0
Control	1	0	0	1	1	1	1
-	0	1	0	0	0	1	1

VIII INSTRUCTION CACHE

Blocks and Specifications:

1. Instruction Memory (Block RAM)

- Size: 256 words (1 kB)
- Address bits: $\log_2 256 = 8$
- Since addresses are multiples of 4, the 2 least significant bits of the PC are always 0.
- PC[9:2] are used to index into the main memory.

2. Instruction Cache Memory

- Size: 32 words
- Associativity: 2-way set associative
- Block/line size: 4 words
- Number of lines: $\frac{32}{4} = 8$
- Number of sets: $\frac{8}{2} = 4$

3. Instruction Tag RAM

- Locations: 8 (same as the number of lines)
- Each line contains used, valid, and tag bits:
 - USED (1 bit)
 - VALID (1 bit)
 - TAG (4 bits)

4. Cache Controller

- Issues control signals to manage cache hits and misses.

5. Shift Register

- Size: 4 words (128 bits)
- Words are shifted into this register one at a time until the entire line is filled.

6. Counter Block

- Size: 3 bits (8 counts)
- Counts cycles and issues control signals to the cache controller to update the cache in a fixed number of cycles.

7. Hit Circuitry

- Determines if an instruction is a hit or miss:
 - If ($tag_0 = tag$ and $v_0 = 1$) then **Hit**
 - Else if ($tag_1 = tag$ and $v_1 = 1$) then **Hit**
 - Else **Miss**

8. Word MUX

- Selects the required word from the 4 words in the cache line based on PC[3:2].

The physical address is split as follows:

7 [TAG] 4 || 3 [SET] 2 || 1 [Block offset] 0

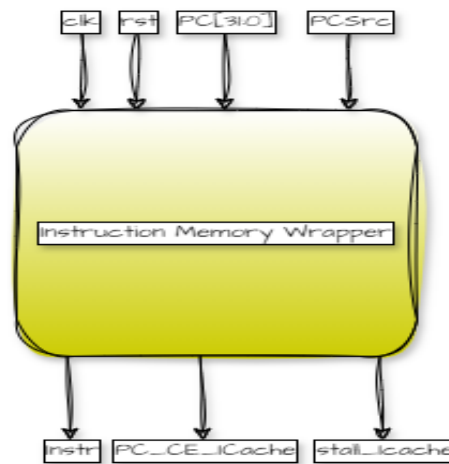


Figure 3: Level 0 of Instruction cache

8.1 Instruction Cache Controller FSM

- **TAG COMPARE:** The incoming address is decomposed into a set number and a tag. The set number indexes both banks of the instruction cache memory and both banks of the instruction tag RAM. A cache hit occurs if the tags match and the cache line is valid; otherwise, it results in a miss. In this state, it is determined if there is a cache miss or hit.
- **READ IMEM:** This state prepares the FSM to read the missed line from the instruction memory in a fixed number of clock cycles by resetting the counter.
- **WAIT IMEM:** The missed line is read from the instruction memory word by word over 4 clock cycles. In this state, the controller waits for the memory operation to complete and for the instruction to be available at the cache input.
- **UPDATE CACHE:** The control signals required to write to the cache are set in this state. These control signals are configured based on which cache line is to be updated.

8.2 Cache Controller Outputs

- **PC_CE_Icache:** When deasserted, it disables the PC from incrementing, allowing the processor to wait until the required instruction is brought into the cache.
- **Stall_Icache:** When asserted, this signal introduces NOP instructions into the Decode stage, ensuring that no operation is performed until the required instruction is fetched into the cache.
- **Imem_rd:** Connected to the enable pin of the Block RAM (BRAM). When asserted, data can be read from the BRAM.
- **next_u0, next_u1:** These are the updated values of the "used" bits for both lines in the selected set. They are written to the Instruction Tag RAM whenever a cache line is updated.
- **next_v0, next_v1:** These are the updated values of the "valid" bits for both lines in the selected set. They are written to the Instruction Tag RAM whenever a cache line is updated.
- **we_t0, we_t1:** These signals are asserted to enable writing to the Instruction Tag RAM. Both signals must be asserted on a cache miss.
- **we_I0, we_I1:** These signals enable writing to the Instruction Cache memory. The specific signal corresponding to the line that needs to be updated within the set should be asserted.
- **next_tag0, next_tag1:** These are the updated values of the tag bits for the line to be updated in the selected set. They are written to the Instruction Tag RAM whenever a cache line is updated.

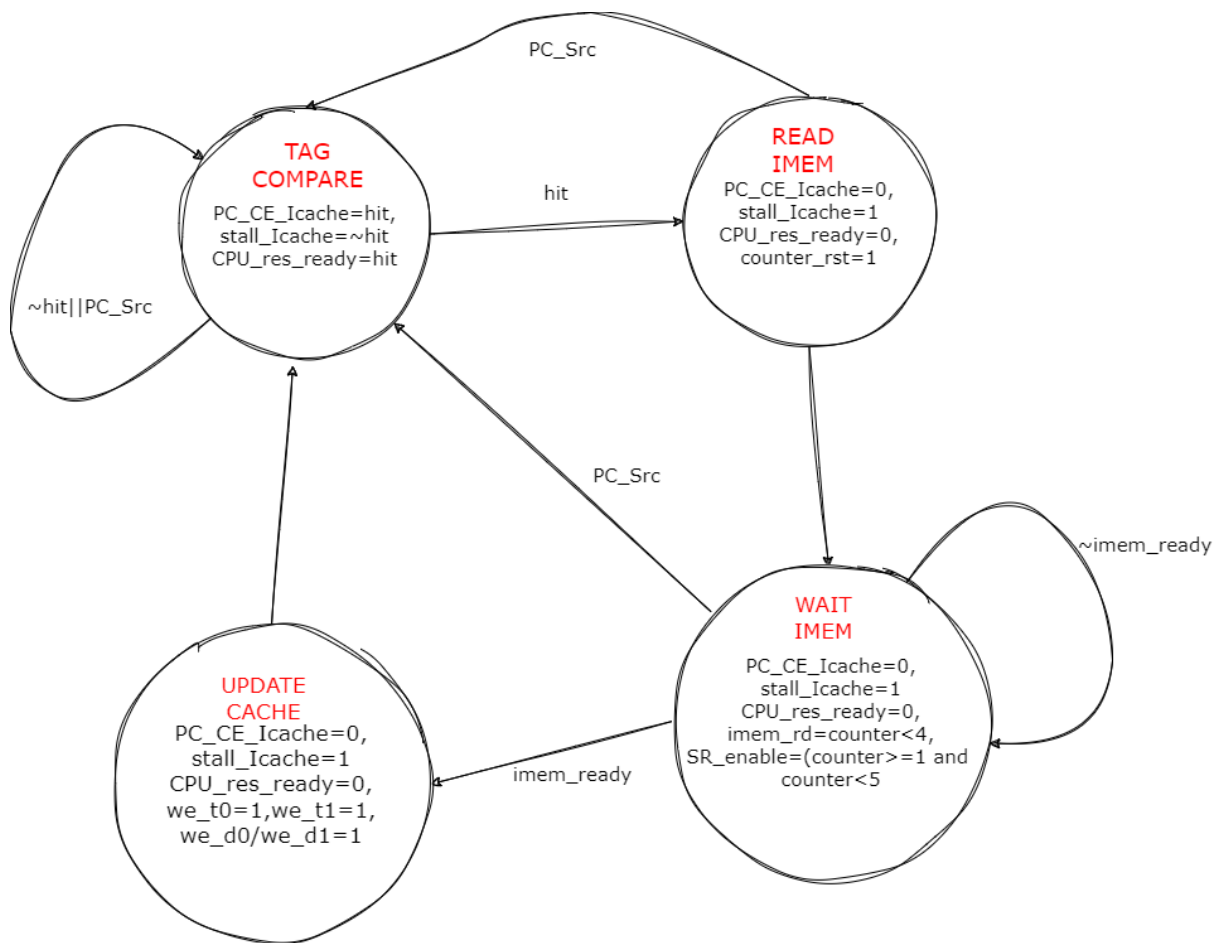


Figure 4: FSM of cache

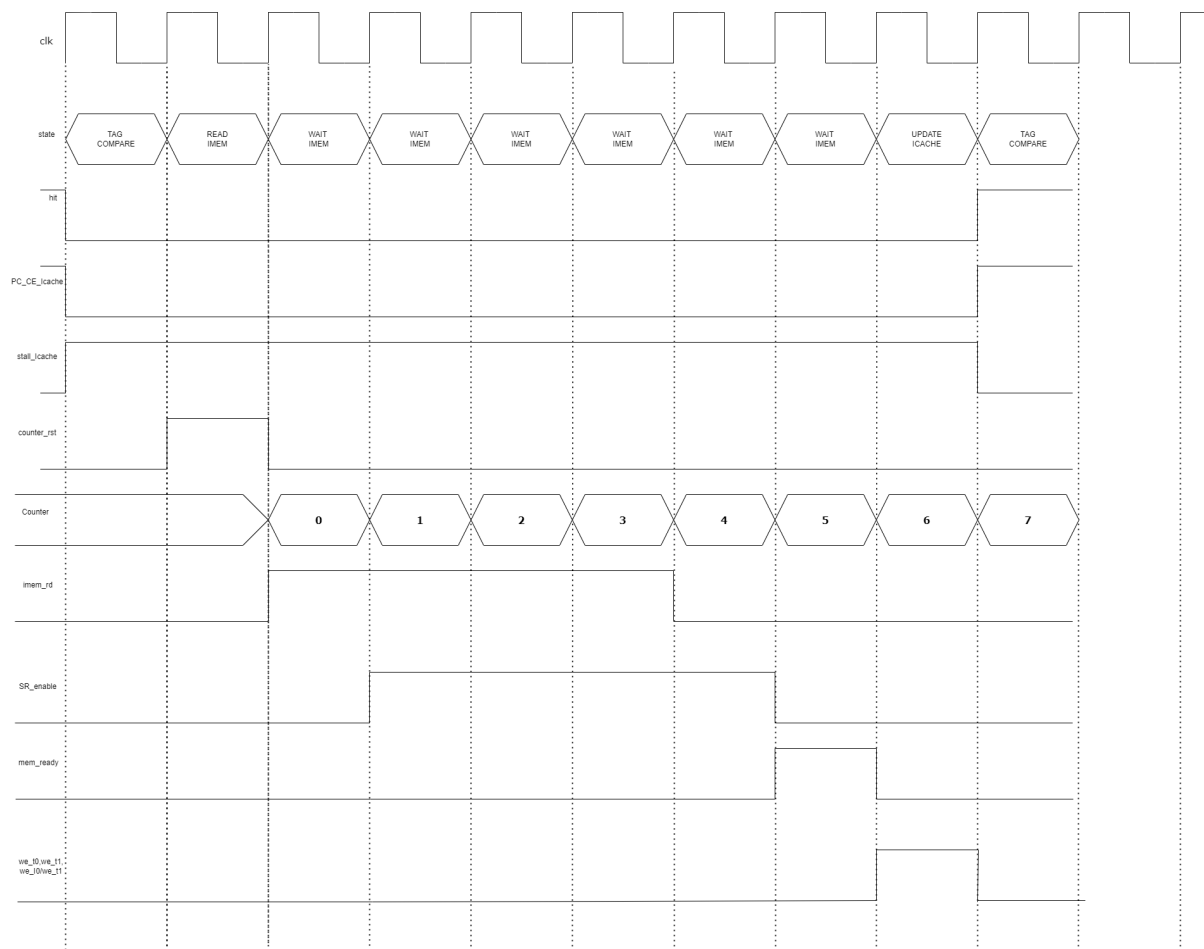


Figure 5: timing cache

8.3 Rules for Cache Line Update

1. If neither line contains valid data, update line 0.
2. If only one line contains valid data, update the line with invalid data.
3. If both lines contain valid data, update the line that was least recently accessed. The "used" bit indicates recent access: $u = 0$ means the line was least recently used, while $u = 1$ means it was recently used.

8.4 Cache Operation Scenarios

Scenario 1: Instruction Hit

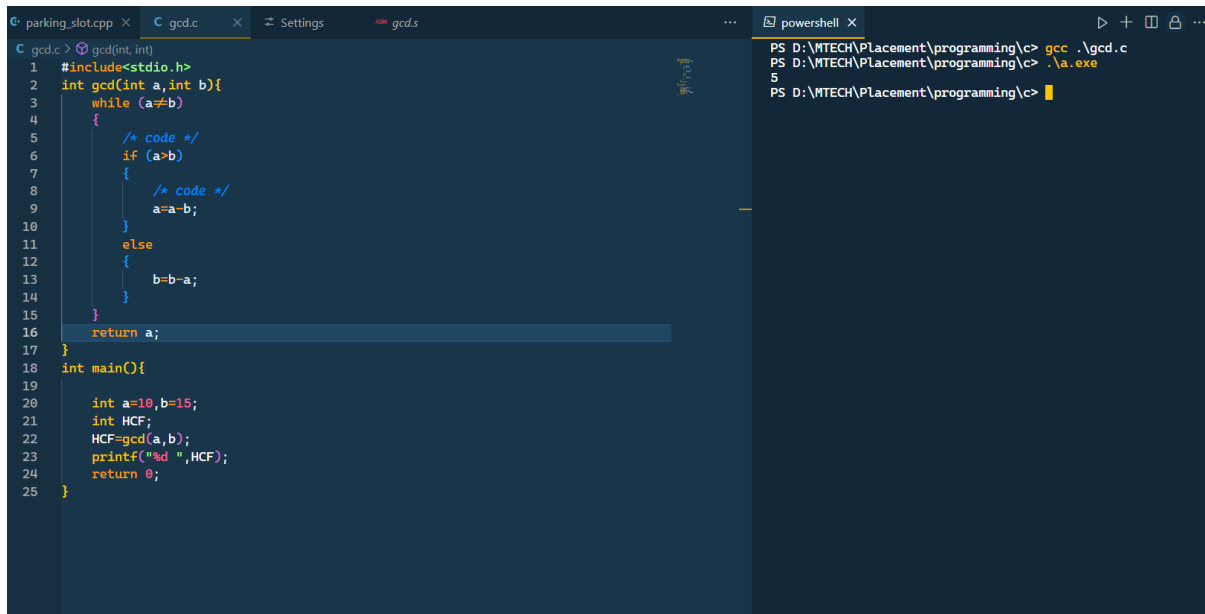
1. The set number and tag are extracted from the program counter (PC).
2. The tags of both lines are compared with the extracted tag, and the valid bits are checked to determine if the lines have valid entries.
3. If one of the lines contains the required instruction, the check results in a hit.
4. The instruction is then passed through the stall multiplexer to the input of the IF/ID pipeline register.

Scenario 2: Instruction Miss

1. The set number and tag are extracted from the PC.
2. The tags of both lines are compared with the extracted tag, and the valid bits are checked to determine if the lines have valid entries.
3. If no line contains the required instruction, the check results in a miss. The PC_CE_Icache signal is deasserted, and Stall_Icache is asserted.
4. A NOP instruction is passed to the Decode and further stages. Meanwhile, the cache prepares to fetch the words from memory.
5. The imem_rd signal is asserted, enabling the instruction word to be read. Over 4 consecutive cycles, 4 words (including the required word and three additional words to form a complete cache line) are shifted into a 128-bit shift register.
6. After all 4 words are shifted in, the data becomes available at the cache input in the next cycle.
7. The appropriate signals to write to the cache are asserted, allowing the data to be written at the next clock edge.
8. The check from step 2 is repeated, resulting in a hit this time.
9. The instruction is passed through the stall multiplexer to the input of the IF/ID pipeline register.

IX TESTING:

PC	Label	Instruction	Machine Code
0		addi x1, x0, 10	00a00093
4		addi x2, x0, 15	00f00113
8	while_loop	beq x1, x2, 24	00208C63
12	if_loop	blt x1, x2, 12	0020C663
16		sub x1, x1, x2	402080B3
20		jal x0, -12	ff5ff06f
24	else_loop	sub x2, x2, x1	40110133
28		jal x0, -20	fedff06f
32		halt	0000001C

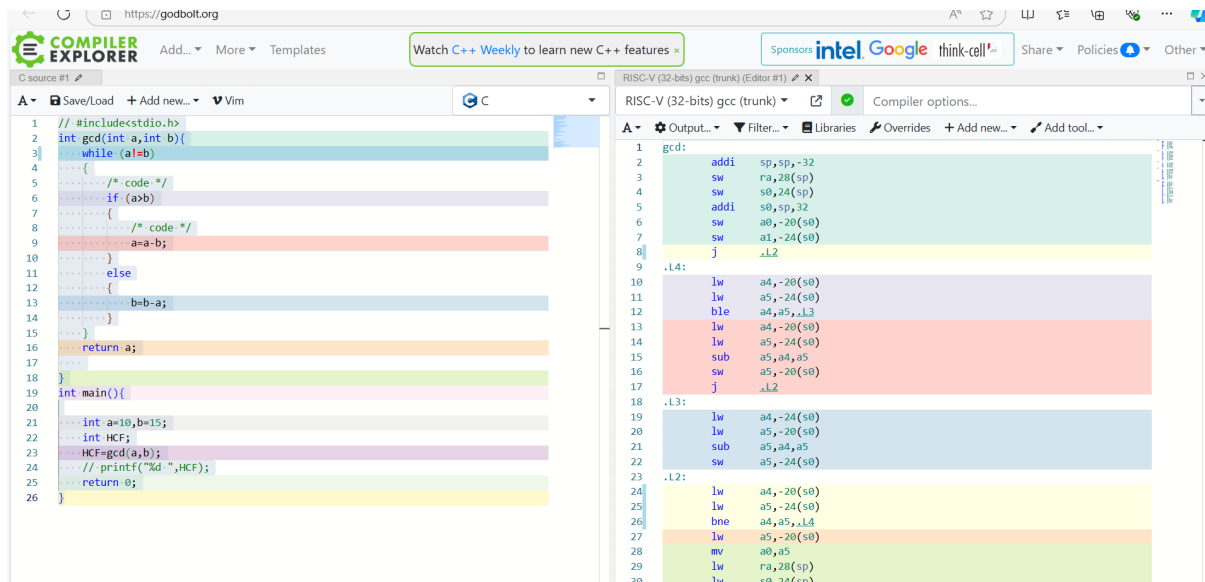


```

1 #include<stdio.h>
2 int gcd(int a,int b){
3     while (a!=b)
4     {
5         /* code */
6         if (a>b)
7         {
8             /* code */
9             a=a-b;
10        }
11        else
12        {
13            b=b-a;
14        }
15    }
16    return a;
17 }
18 int main(){
19     int a=10,b=15;
20     int HCF;
21     HCF=gcd(a,b);
22     printf("%d ",HCF);
23     return 0;
24 }

```

Figure 6: C prog of GCD



```

1 // #include<stdio.h>
2 int gcd(int a,int b){
3     while (a!=b)
4     {
5         /* code */
6         if (a>b)
7         {
8             /* code */
9             a=a-b;
10        }
11        else
12        {
13            b=b-a;
14        }
15    }
16    return a;
17 }
18 int main(){
19     int a=10,b=15;
20     int HCF;
21     HCF=gcd(a,b);
22     // printf("%d ",HCF);
23     return 0;
24 }

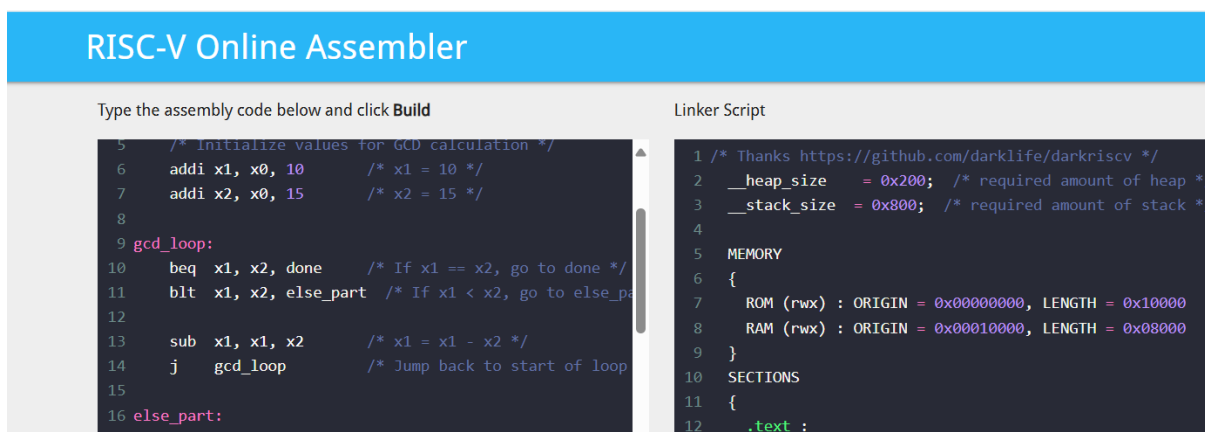
```

```

1 gcd:
2     addi    sp,sp,-32
3     sw      ra,28(sp)
4     sw      s0,24(sp)
5     addi    s0,sp,32
6     sw      a0,-20(s0)
7     sw      a1,-24(s0)
8     j       .L2
9 .L4:
10    lw       a4,-20(s0)
11    lw       a5,-24(s0)
12    ble      a4,a5,.L3
13    lw       a4,-20(s0)
14    lw       a5,-24(s0)
15    sub      a5,a4,a5
16    sw       a5,-20(s0)
17    j       .L2
18 .L3:
19    lw       a4,-24(s0)
20    lw       a5,-20(s0)
21    sub      a5,a4,a5
22    sw       a5,-24(s0)
23 .L2:
24    lw       a4,-20(s0)
25    lw       a5,-24(s0)
26    bne      a4,a5,.L4
27    lw       a5,-20(s0)
28    mv       a0,a5
29    lw       ra,28(sp)
30    lw       s0,24(sp)

```

Figure 7: C code and assembly code conversion



```

5 /* Initialize values for GCD calculation */
6 addi x1, x0, 10 /* x1 = 10 */
7 addi x2, x0, 15 /* x2 = 15 */
8
9 gcd_loop:
10 beq x1, x2, done /* If x1 == x2, go to done */
11 blt x1, x2, else_part /* If x1 < x2, go to else_part */
12
13 sub x1, x1, x2 /* x1 = x1 - x2 */
14 j gcd_loop /* Jump back to start of loop */
15
16 else_part:
17 sub x2, x2, x1 /* x2 = x2 - x1 */

```

```

1 /* Thanks https://github.com/darklife/darkriscv */
2 __heap_size = 0x200; /* required amount of heap */
3 __stack_size = 0x800; /* required amount of stack */
4
5 MEMORY
6 {
7     ROM (rwx) : ORIGIN = 0x00000000, LENGTH = 0x10000
8     RAM (rwx) : ORIGIN = 0x00010000, LENGTH = 0x08000
9 }
10 SECTIONS
11 {
12     .text :

```

Figure 8: assembly code to machine code conversion

X RESULTS

10.1 RTL Schematic:

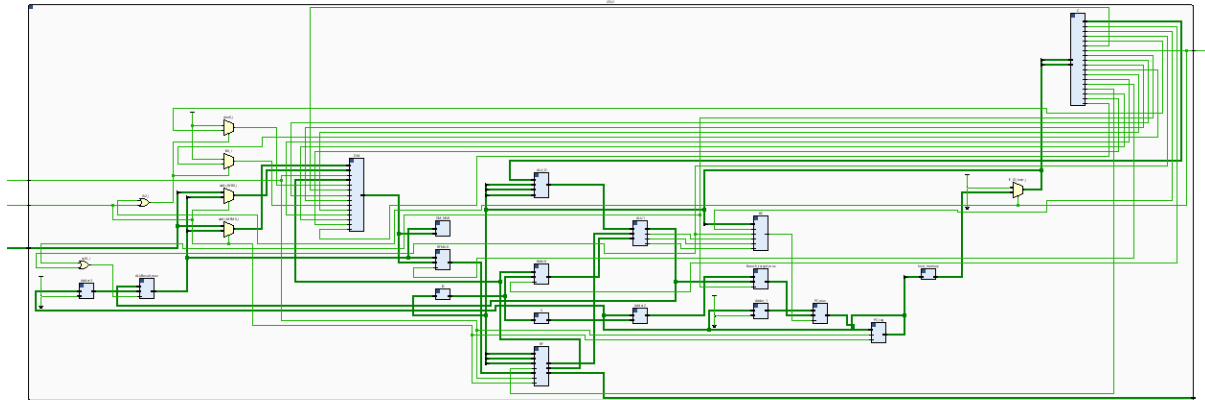


Figure 9: RTL schematic of CPU

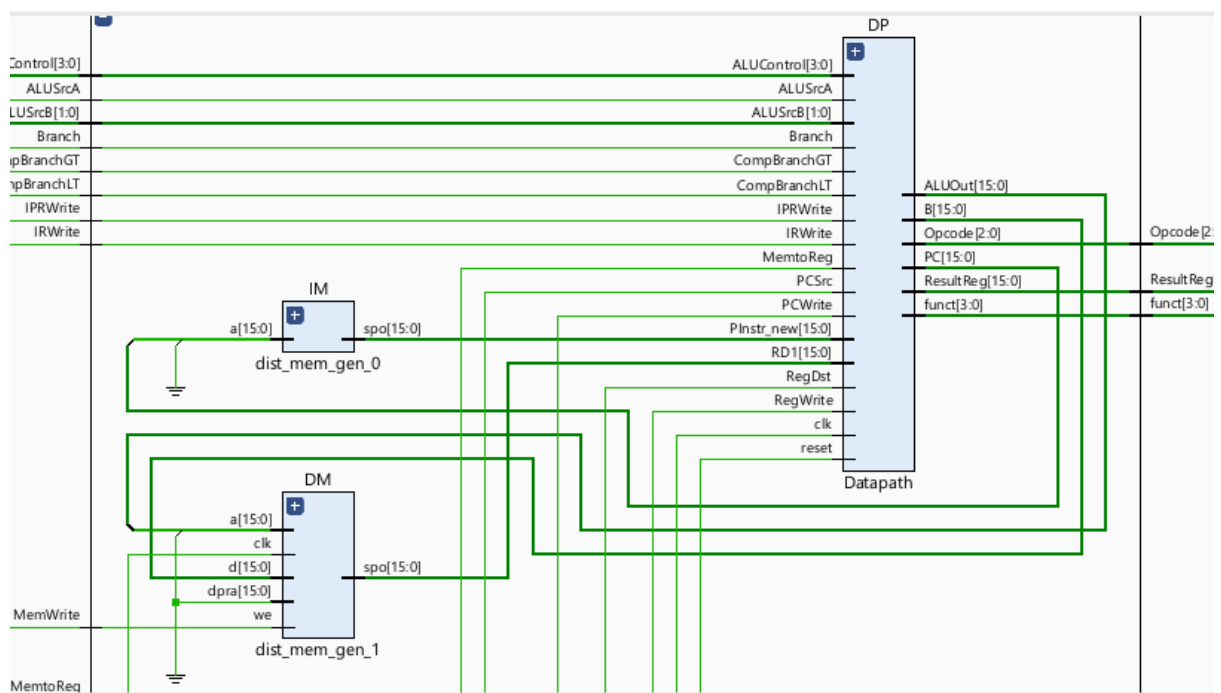


Figure 10: RTL schematic of Datapath with memories

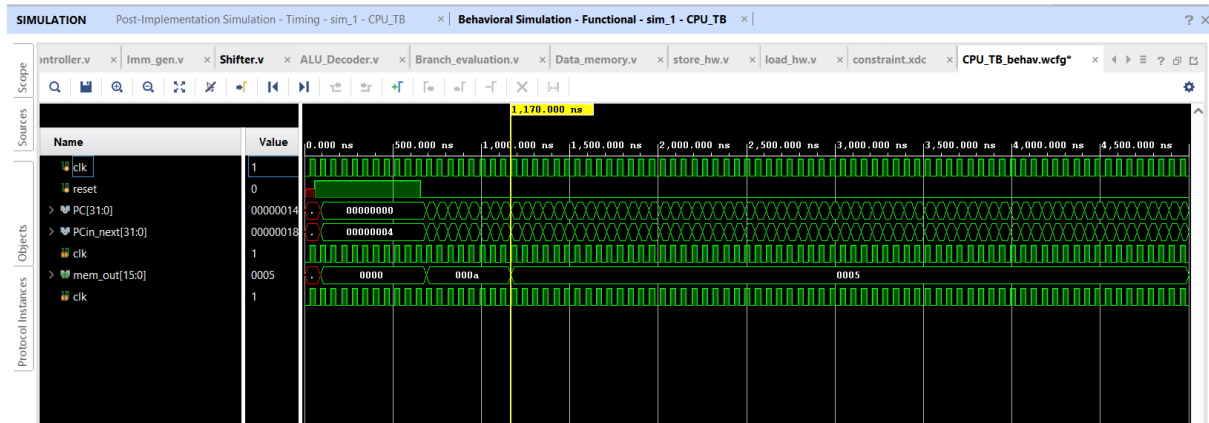


Figure 11: : Behavioral simulation of GCD

10.2 Behavioral simulation:

10.3 Post implementation timing simulation:

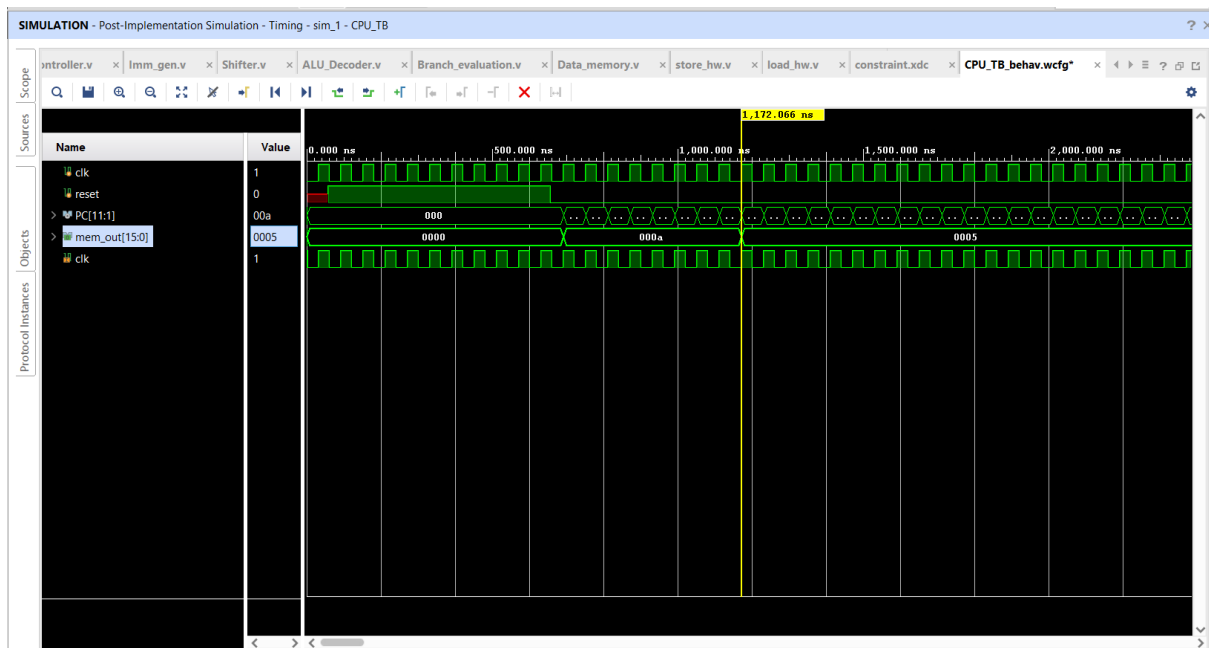


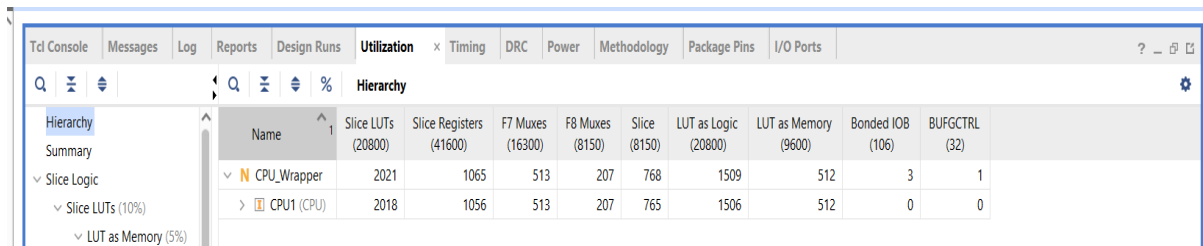
Figure 12: : Post implementation timing simulation of GCD

10.4 Utilization:

10.5 Timing summary:

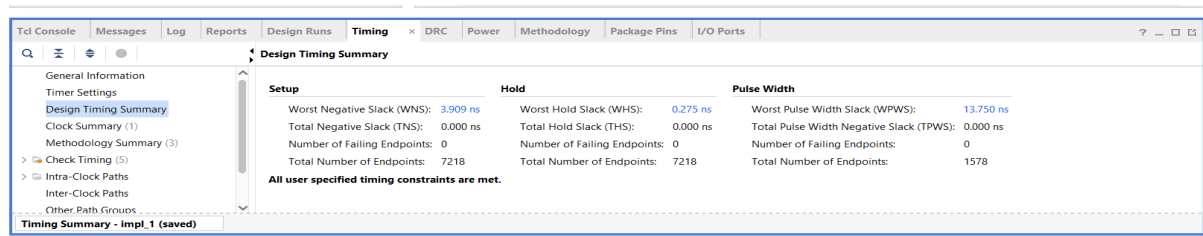
The minimum time period is calculated as:

$$\text{Minimum time period} = 30 - 3.9 = 26.1 \text{ ns}$$



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Bonded IOB (106)	BUFGCTRL (32)
CPU1 (CPU)	2018	1056	513	207	765	1506	512	0	0

Figure 13: :Utilization summary



Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.909 ns	Worst Hold Slack (WHS): 0.275 ns	Worst Pulse Width Slack (WPWS): 13.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 7218	Total Number of Endpoints: 7218	Total Number of Endpoints: 1578

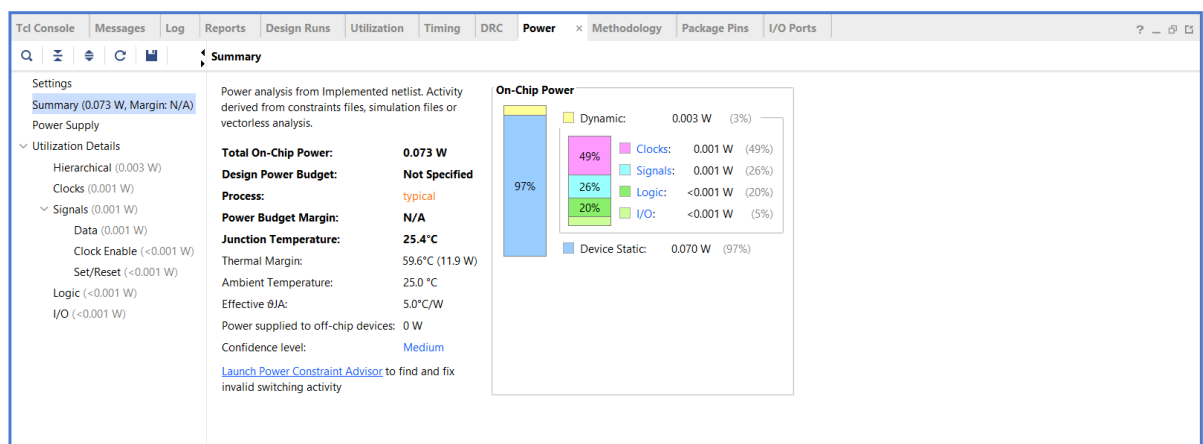
All user specified timing constraints are met.

Figure 14: :Timing Summary

The maximum frequency is given by:

$$\text{Maximum frequency} = \frac{1}{26.1 \text{ ns}} = 38.31 \text{ MHz}$$

10.6 Power Report:



Power Analysis Metric	Value
Total On-Chip Power	0.073 W
Design Power Budget	Not Specified
Process	typical
Power Budget Margin	N/A
Junction Temperature	25.4°C
Thermal Margin	59.6°C (11.9 W)
Ambient Temperature	25.0 °C
Effective θJA	5.0°C/W
Power supplied to off-chip devices	0 W
Confidence level	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

Figure 15: :Power Report