University of Moratuwa

electronic and telecommunication engineering

EN3030 : Circuits And System Design

# InFOS: RV32i RISC-V Processor

Group 12: INFOS

| | |
|---|---|
| Tharindu O.K.D. | 190622R |
| Sanjith. S | 190562G |
| Kajhanan K. | 190286M |
| Yasarantha D. D. K. B | 190719V |
| Wansooriya W.H.O. | 190664V |

February 13, 2023

The report is submitted as a partial fulfillment of the module EN3030.

## Contents

## List of Figures

## List of Tables

**Abstract**

This report presents the design and implementation of a single-cycle Reduced Instruction Set Computer (RISC-V) processor, based on the RV32I instruction set architecture. The processor is implemented using Verilog Hardware Description Language and simulated using Verilator together with the GTKWave toolbox. The processor includes the basic RISC-V instruction set, including arithmetic, logical, and control flow instructions, and is capable of executing these instructions in a single clock cycle. The latter part of the work focuses on a cache controller design for a direct mapped cache bonded with a fully associative victim cache.

The Executable code can be found here

# 1 Instruction Set Architecture

Based on the requirements of the design it is decided to give support for all 37 instructions of the RV32i instruction set. At earlier stages, the instruction encodings are analyzed thoroughly and the results together with instructions are as follows.

| Instruction | | Description | Operation | Type | funct7 (31–25) | (24–20) | (19–15) | funct3 (14–12) | (11–7) | opcode (6–0) |
|---|---|---|---|---|---|---|---|---|---|---|
| lui | rd,imm | Load Upper Immediate | rd ← imm_u, pc ← pc+4 | U | imm[31:12] | | | | rd | 0 1 1 0 1 1 1 |
| auipc | rd,imm | Add Upper Immediate to PC | rd ← pc + imm_u, pc ← pc+4 | U | imm[31:12] | | | | rd | 0 0 1 0 1 1 1 |
| jal | rd,pcrel_21 | Jump And Link | rd ← pc+4, pc ← pc+imm_j | J | imm[20\|10:1\|11\|19:12] | | | | rd | 1 1 0 1 1 1 1 |
| jalr | rd,imm(rs1) | Jump And Link Register | rd ← pc+4, pc ← (rs1+imm_i) ∧ ~1 | I | imm[11:0] | | rs1 | 0 0 0 | rd | 1 1 0 0 1 1 1 |
| beq | rs1,rs2,pcrel_13 | Branch Equal | pc ← pc + ((rs1==rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 0 0 0 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| bne | rs1,rs2,pcrel_13 | Branch Not Equal | pc ← pc + ((rs1!=rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 0 0 1 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| blt | rs1,rs2,pcrel_13 | Branch Less Than | pc ← pc + ((rs1<rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 1 0 0 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| bge | rs1,rs2,pcrel_13 | Branch Greater or Equal | pc ← pc + ((rs1>=rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 1 0 1 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| bltu | rs1,rs2,pcrel_13 | Branch Less Than Unsigned | pc ← pc + ((rs1<rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 1 1 0 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| bgeu | rs1,rs2,pcrel_13 | Branch Greater or Equal Unsigned | pc ← pc + ((rs1>=rs2) ? imm_b : 4) | B | imm[12\|10:5] | rs2 | rs1 | 1 1 1 | imm[4:1\|11] | 1 1 0 0 0 1 1 |
| lb | rd,imm(rs1) | Load Byte | rd ← sx(m8(rs1+imm_i)), pc ← pc+4 | I | imm[11:0] | | rs1 | 0 0 0 | rd | 0 0 0 0 0 1 1 |
| lh | rd,imm(rs1) | Load Halfword | rd ← sx(m16(rs1+imm_i)), pc ← pc+4 | I | imm[11:0] | | rs1 | 0 0 1 | rd | 0 0 0 0 0 1 1 |
| lw | rd,imm(rs1) | Load Word | rd ← sx(m32(rs1+imm_i)), pc ← pc+4 | I | imm[11:0] | | rs1 | 0 1 0 | rd | 0 0 0 0 0 1 1 |
| lbu | rd,imm(rs1) | Load Byte Unsigned | rd ← zx(m8(rs1+imm_i)), pc ← pc+4 | I | imm[11:0] | | rs1 | 1 0 0 | rd | 0 0 0 0 0 1 1 |
| lhu | rd,imm(rs1) | Load Halfword Unsigned | rd ← zx(m16(rs1+imm_i)), pc ← pc+4 | I | imm[11:0] | | rs1 | 1 0 1 | rd | 0 0 0 0 0 1 1 |
| sb | rs2,imm(rs1) | Store Byte | m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4 | S | imm[11:5] | rs2 | rs1 | 0 0 0 | imm[4:0] | 0 1 0 0 0 1 1 |
| sh | rs2,imm(rs1) | Store Halfword | m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4 | S | imm[11:5] | rs2 | rs1 | 0 0 1 | imm[4:0] | 0 1 0 0 0 1 1 |
| sw | rs2,imm(rs1) | Store Word | m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4 | S | imm[11:5] | rs2 | rs1 | 0 1 0 | imm[4:0] | 0 1 0 0 0 1 1 |
| addi | rd,rs1,imm | Add Immediate | rd ← rs1 + imm_i, pc ← pc+4 | I | imm[11:0] | | rs1 | 0 0 0 | rd | 0 0 1 0 0 1 1 |
| slti | rd,rs1,imm | Set Less Than Immediate | rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4 | I | imm[11:0] | | rs1 | 0 1 0 | rd | 0 0 1 0 0 1 1 |
| sltiu | rd,rs1,imm | Set Less Than Immediate Unsigned | rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4 | I | imm[11:0] | | rs1 | 0 1 1 | rd | 0 0 1 0 0 1 1 |
| xori | rd,rs1,imm | Exclusive Or Immediate | rd ← rs1 ⊕ imm_i, pc ← pc+4 | I | imm[11:0] | | rs1 | 1 0 0 | rd | 0 0 1 0 0 1 1 |
| ori | rd,rs1,imm | Or Immediate | rd ← rs1 ∨ imm_i, pc ← pc+4 | I | imm[11:0] | | rs1 | 1 1 0 | rd | 0 0 1 0 0 1 1 |
| andi | rd,rs1,imm | And Immediate | rd ← rs1 ∧ imm_i, pc ← pc+4 | I | imm[11:0] | | rs1 | 1 1 1 | rd | 0 0 1 0 0 1 1 |
| slli | rd,rs1,shamt | Shift Left Logical Immediate | rd ← rs1 << shamt_i, pc ← pc+4 | I | 0 0 0 0 0 0 0 | shamt | rs1 | 0 0 1 | rd | 0 0 1 0 0 1 1 |
| srli | rd,rs1,shamt | Shift Right Logical Immediate | rd ← rs1 >> shamt_i, pc ← pc+4 | I | 0 0 0 0 0 0 0 | shamt | rs1 | 1 0 1 | rd | 0 0 1 0 0 1 1 |
| srai | rd,rs1,shamt | Shift Right Arithmetic Immediate | rd ← rs1 >> shamt_i, pc ← pc+4 | I | 0 1 0 0 0 0 0 | shamt | rs1 | 1 0 1 | rd | 0 0 1 0 0 1 1 |
| add | rd,rs1,rs2 | Add | rd ← rs1 + rs2, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 0 0 0 | rd | 0 1 1 0 0 1 1 |
| sub | rd,rs1,rs2 | Subtract | rd ← rs1 - rs2, pc ← pc+4 | R | 0 1 0 0 0 0 0 | rs2 | rs1 | 0 0 0 | rd | 0 1 1 0 0 1 1 |
| sll | rd,rs1,rs2 | Shift Left Logical | rd ← rs1 << (rs2%XLEN), pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 0 0 1 | rd | 0 1 1 0 0 1 1 |
| slt | rd,rs1,rs2 | Set Less Than | rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 0 1 0 | rd | 0 1 1 0 0 1 1 |
| sltu | rd,rs1,rs2 | Set Less Than Unsigned | rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 0 1 1 | rd | 0 1 1 0 0 1 1 |
| xor | rd,rs1,rs2 | Exclusive Or | rd ← rs1 ⊕ rs2, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 1 0 0 | rd | 0 1 1 0 0 1 1 |
| srl | rd,rs1,rs2 | Shift Right Logical | rd ← rs1 >> (rs2%XLEN), pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 1 0 1 | rd | 0 1 1 0 0 1 1 |
| sra | rd,rs1,rs2 | Shift Right Arithmetic | rd ← rs1 >> (rs2%XLEN), pc ← pc+4 | R | 0 1 0 0 0 0 0 | rs2 | rs1 | 1 0 1 | rd | 0 1 1 0 0 1 1 |
| or | rd,rs1,rs2 | Or | rd ← rs1 ∨ rs2, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 1 1 0 | rd | 0 1 1 0 0 1 1 |
| and | rd,rs1,rs2 | And | rd ← rs1 ∧ rs2, pc ← pc+4 | R | 0 0 0 0 0 0 0 | rs2 | rs1 | 1 1 1 | rd | 0 1 1 0 0 1 1 |

# 2 Micro Architecture

## 2.1 Overview

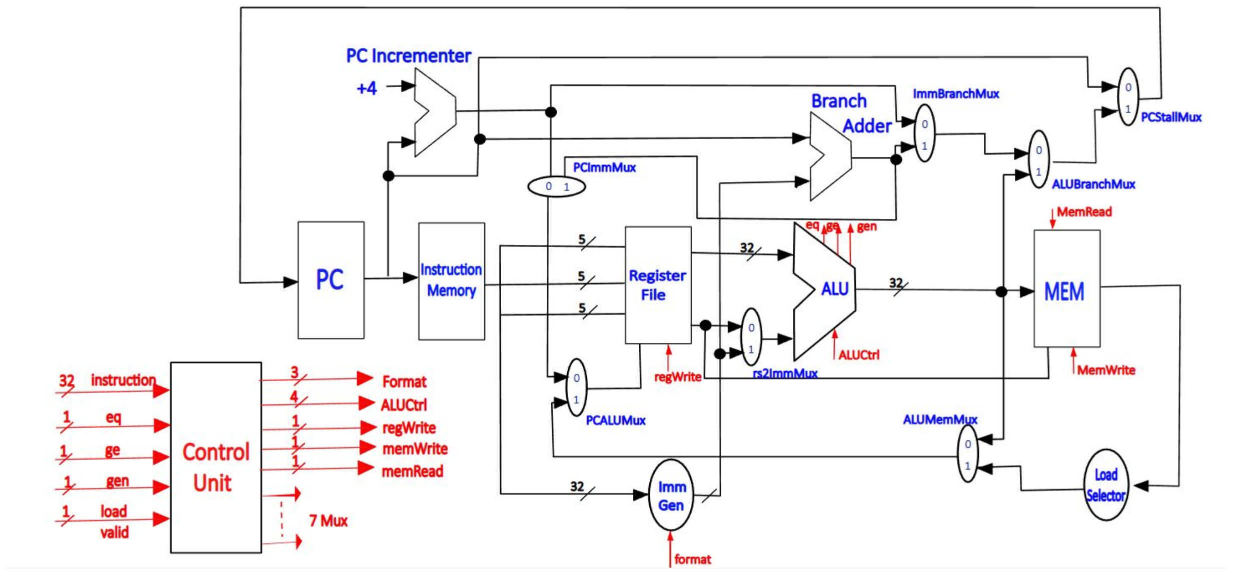


Figure 1: Micro Architecture Together with Data Paths

The processor design is completely based on a 32-bit architecture meaning that all the data paths are 32-bit in size. The main constituents include a Program Counter, an Instruction Memory, a Register File, a 32-bit ALU, and an Immediate Generator. Refer to the next section for more details about these blocks.

The branching addresses, on occurrence, when branching instructions are executed, are calculated by a separate adder (Branching Adder indicated in the above figure). A separate adder is used to add 4 to the program counter, to find the address of the instruction that is sequentially next to the current instruction. For lui, jal, jalr, and auipc instructions, a new Mux is inserted before RegisterFile (PCAluMux) so that the return address can be stored in the rd register.

## 2.2 Register Files

Our single-cycle processor architecture demands the register read access through combinational logic and write access at the clock risign edge. Totally a 32 different 32-bit registers are implemented. Among them x0 register is hard-wired to zero register.

## 2.3 Immediate Generator

The immediate generator extracts the corresponding immediate bits from the given instruction according to the type and sign-extends in to a 32-bit number.

## 2.4 ALU Design

The operations that should be implemented by the ALU were decided after inspecting all the supported instructions. We concluded that there is a total of 12 operations are required; hence, a 4-bit ALU signal is required to encode the necessary control signals. For the branching instructions, 3 flags are given by the ALU. The operations and flags are given in table 1.

| ALU Control | Description | Operation |
|---|---|---|
| 0000 | AND | A & B |
| 0001 | OR | A \| B |
| 0010 | XOR | A B |
| 0011 | ADD | A + B |
| 0100 | SUB | A - B |
| 0101 | Set Less Than | (A < B)?1:0 |
| 0110 | Logical Left Shift | << |
| 0111 | Set Less than Unsigned | |
| 1000 | Logical Right Shift | >> |
| 1001 | Arithmetic Right Shift | |
| 1010 | JALR Jump | |
| 1011 | select B | B |

Table 1: ALU Operation

## 2.5 Control Unit

The control unit is designed to echo the following flow of logic. First, it would read the opcode of the instruction and identify the corresponding instruction type. Then according to the instruction type, it would send the required signals to each module. Table 2 shows the outputs of the control unit that would realize the above flow of logic.

| **Type** | **Mux Control Signals** | | | | | | | **RW** | **MR** | **MW** |
|---|---|---|---|---|---|---|---|---|---|---|
| | **rs2Imm** | **aluMem** | **pcImm** | **pcAlu** | **immBran** | **aluBran** | **stall** | | | |
| **R** | 0 | 0 | X | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| **I** | 1 | 0 | X | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| **I_load** | 1 | 1 | X | 1 | 0 | 0 | ready | 1 | 1 | 0 |
| **I_jalr** | X | X | 0 | 0 | X | 1 | 1 | 1 | 0 | 0 |
| **S** | 1 | X | X | X | 0 | 0 | ∼wait | 0 | 0 | 1 |
| **B** | 0 | X | X | X | X | 0 | 1 | 0 | 0 | 0 |
| **U_lui** | 0 | 0 | X | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| **U_auipc** | X | X | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| **J_jal** | X | X | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

Table 2: Mux Control Signals

# 3 Memory Hierarchy

## 3.1 Overview

The operations inside the core processor can be done at a very high speed compared to memory-related operations. This creates a situation where the central processing unit has to wait for the data to be fetched from memory, which can slow down the overall performance of the system. To mitigate this bottleneck, systems often use caching by utilizing temporal and spatial locality.

In our design, we achieved such an approach by designing the memory path wider(four times) than the CPU data access path. A wider path together with a small faster memory element is giving access to more(three more in our case) neighbouring data elements of the memory within the same access time.

## 3.2 Design Specifications

In addition to the main memory, our memory design consist of two key components.

1. A completely directed mapped primary cache.
2. A small fully associative victim cache.

To imitate the physical constraints in our simulation, we scheduled the memory operations to be five clock cycles long whereas other cache-related operations can be performed with a single clock cycle. Although, the processor can address a total of four gigabytes of memory while supporting the byte addressing, based on the simulation constraints it is decided to keep the memory to 256kB in size. That is only the first eighteen bits of the CPU address are used in specifying the memory location.

We simulated a direct mapped cache of size two-kilo bytes together with a fully associative victim cache of size 128bytes. Based on the memory path specifications, memory and cache lines are considered to have 16 bytes. Based on our decisions, here are the sizes of some parameters

| PARAMETER | SIZE (bits) |
|---|---:|
| Cache Line | 128 |
| Cache Offset | 4 |
| Primary Cache Index | 7 |
| Primary Cache Tag | 7 |
| Victim Cache Index | 3 |
| Victim Cache Tag | 14 |

Table 3: Cache Parameters

Considering the fully associative nature of the victim cache, it is decided to implement a write-back policy together with the allocation as the writing policies. According to the policy, any memory writing operation will bring the data line into the memory and make it dirty. Once an eviction is requested from the victim, memory write-back will happen only if the such line is dirty.

## 3.3 Design Decisions

Considering the availability of data in three distinguished places, the memory delay for data accessing varies randomly. So it is the task of the cache controller to indicate to the CPU the state of the data. Since our data is always brought to the primary cache before accessing or modification, knowing when the data is ready in the cache is enough for the CPU to interact with memory. In our implementation, we directly compare the primary cache tag when the particular primary cache line is valid and use it as a ready signal for the CPU to proceed with the next operation. Hence, making sure the tag is get updated at the right time will indicate the readiness of the data to the CPU.

When the CPU request is a memory write operation, it is not necessary to keep the total processing unit in an IDLE state. The memory write operation can be performed parallelly. But it will be problematic when the CPU comes up with two subsequent write operations. As a way to tackle the issue, unlike in load, our implementation gives the CPU a wait signal which indicates the CPU to wait before proceeding to the next operation.

## 3.4 Eviction Line Selection

In the case of requesting space from an already filled victim cache, data spilling should be avoided by writing the dirty data line back into memory. Under this regard, the method of choosing the block to be evicted is done carefully to reduce the latency. To exploit the concept of temporal locality, we have made use of the not mostly used technique. We are using 2 bits to encode the most recently used (MRU) blocks. This means, at a time 3 blocks can be labelled as MRU. MRU approach is implemented by initializing MRU register values to 00 and when something is written to the victim cache, making the MRU bits to the block written as 11, keeping MRU values of other blocks with 00 unchanged and decrementing all other blocks by 1. After checking and shortlisting the 5 least recently used blocks, one is selected from the state of 2 bits [13:12] from the CPU address.

## 3.5 Realization of Cache Controller

Our approach to the cache controller design could be realized with the following finite state machine.

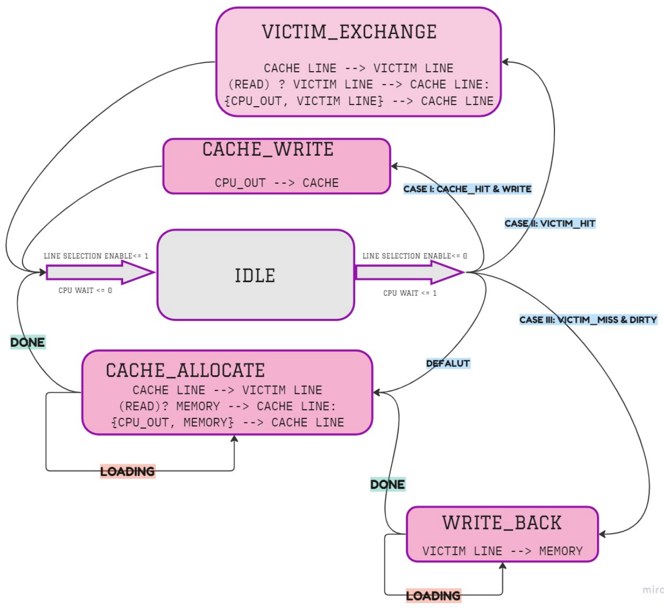

VICTIM_EXCHANGE
CACHE LINE --> VICTIM LINE
(READ) ? VICTIM LINE --> CACHE LINE:
{CPU_OUT, VICTIM LINE} --> CACHE LINE
CACHE_WRITE
CPU_OUT --> CACHE
CASE I: CACHE_HIT & WRITE
CASE II: VICTIM_HIT
LINE SELECTION ENABLE<= 1
CPU WAIT <= 0
IDLE
LINE SELECTION ENABLE<= 0
CPU WAIT <= 1
CASE III: VICTIM_MISS & DIRTY
DONE
DEFALUT
CACHE_ALLOCATE
CACHE LINE --> VICTIM LINE
(READ)? MEMORY --> CACHE LINE:
{CPU_OUT, MEMORY} --> CACHE LINE
LOADING
DONE
WRITE_BACK
VICTIM LINE --> MEMORY
LOADING
miro

Figure 2: Cache Controller as a Finite State Machine

Normally the memory will be in IDLE state. When the CPU provides a valid read or write instruction,

1. The controller will check whether the data is available in the primary cache. Suppose the data is available and the instruction is of load type the controller will do nothing while allowing the CPU to access the data.
2. Suppose the data is available and the instruction is of write type, the controller will turn into CACHE_WRITE state and allow the CPU to write into the cache while tracking the modification via a dirty flag unique to the line.
3. When there is a cache miss, the controller will search for the data inside the victim. Availability of the data into victim excites the cache into VICTIM_SWAP state while allowing the data swap between the primary cache and victim cache. The controller is then forwarded to IDLE state.
4. Victim misses together with a cache miss, trigger the controller into either one of WRITE_BACK state or CACHE_ALLOCATE state based on whether the selected victim line is dirty. Where the victim line selection will happen according to an algorithm that always preserves the most recently used three lines.
5. On the WRITE_BACK state, the dirty victim line will be written back into the memory and the controller is forwarded to CACHE_ALLOCATE state. In this state, the data will be fetched from both memory and the CPU to put into the appropriate cache line according to the instruction.

## 3.6 Architecture

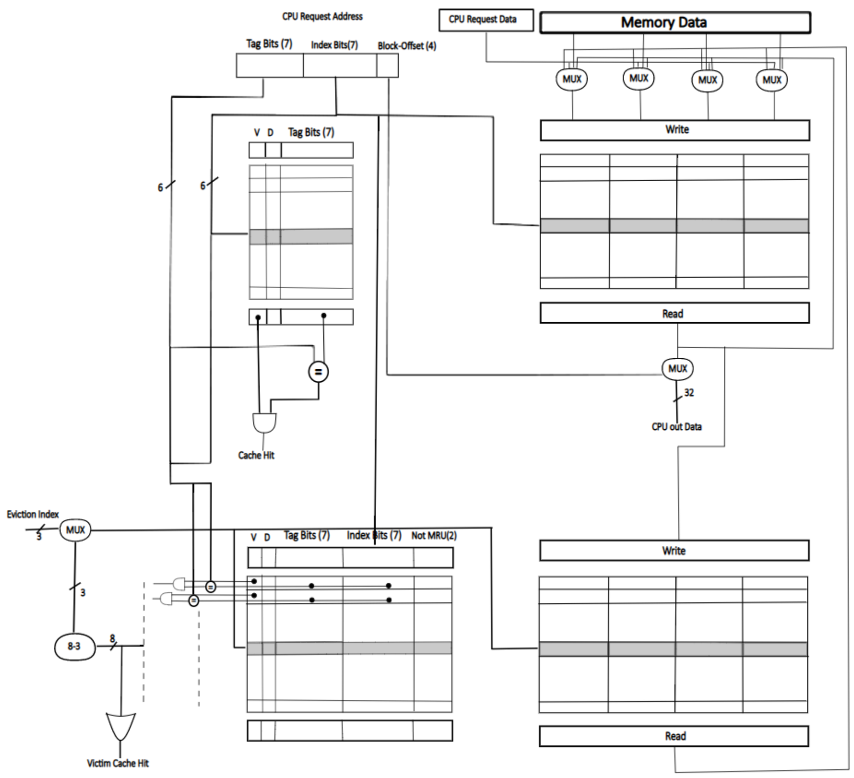


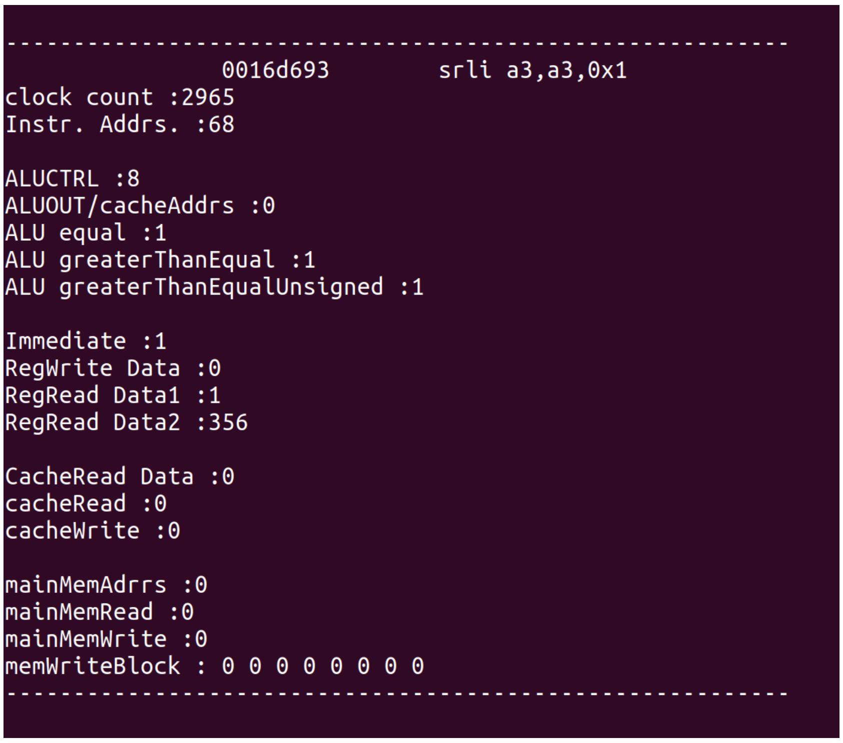
Figure 3: Memory Architecture

Look at the cache_memory.v file for implementation details.

## 4 Verification of the Design

### 4.1 Pre-stage Verifications

At earlier stages, simple modules are designed and verified using Modelsim integrated environment available in Quartus Prime. Later the core design is completely designed and simulated using chisel scripts. After complete verification of the core processor data paths, the working of each instruction is verified using the verilator script. Here in the fig4 the implementation of SRLI instruction is illustrated.



```
----------------------------------------------------------
                0016d693        srli a3,a3,0x1
clock count :2965
Instr. Addrs. :68

ALUCTRL :8
ALUOUT/cacheAddrs :0
ALU equal :1
ALU greaterThanEqual :1
ALU greaterThanEqualUnsigned :1

Immediate :1
RegWrite Data :0
RegRead Data1 :1
RegRead Data2 :356

CacheRead Data :0
cacheRead :0
cacheWrite :0

mainMemAdrrs :0
mainMemRead :0
mainMemWrite :0
memWriteBlock : 0 0 0 0 0 0 0 0
----------------------------------------------------------
```

Figure 4: Processor State After an SRLI Instruction

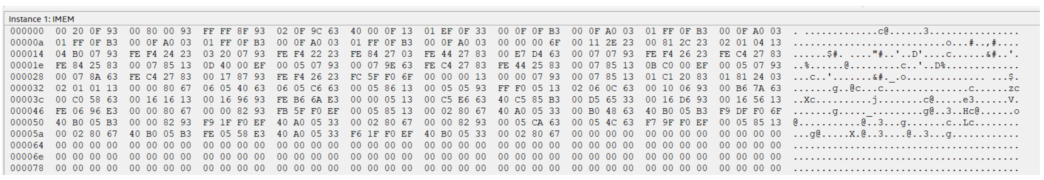
### 4.2 Core Verification

The proper functioning of the design is verified by running a simple c code snippet that calculates the least common multiple of two given integers.

```c
int main() {
    int n1 = 75, n2 = 50, max;
    max = (n1 > n2) ? n1 : n2;
    while (1) {
        if ((max % n1 == 0) && (max % n2 == 0)) {
            break;
        }
        ++max;
    }
    return 0;
}
```

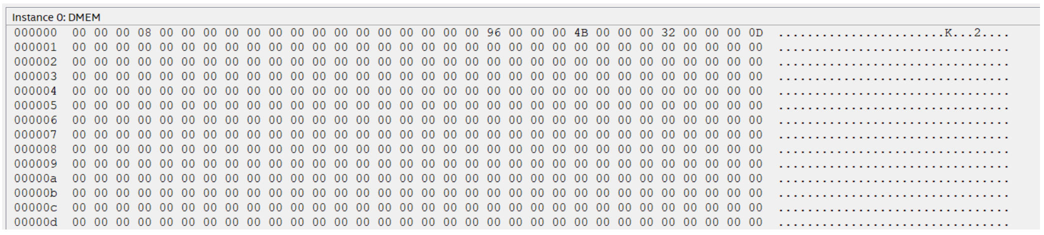
Initially, the code snippet was converted into assembly-level instructions using the open-source riscv-rv32i-gcc tool-chain to compile a given C code. Using a custom-made python assembler for our design, we derived binaries from the converted assembly-level instructions. These instructions are loaded into the instruction memory using the memory content editor available with Quartus Prime Software and the data memory is monitored to check whether the expected outcome is attained.

Register monitoring are performed using a Verilator script and the results are as follows, We used 50 and 75 as our numbers and the least common multiple 150 is observable at the appropriate location in the cache.



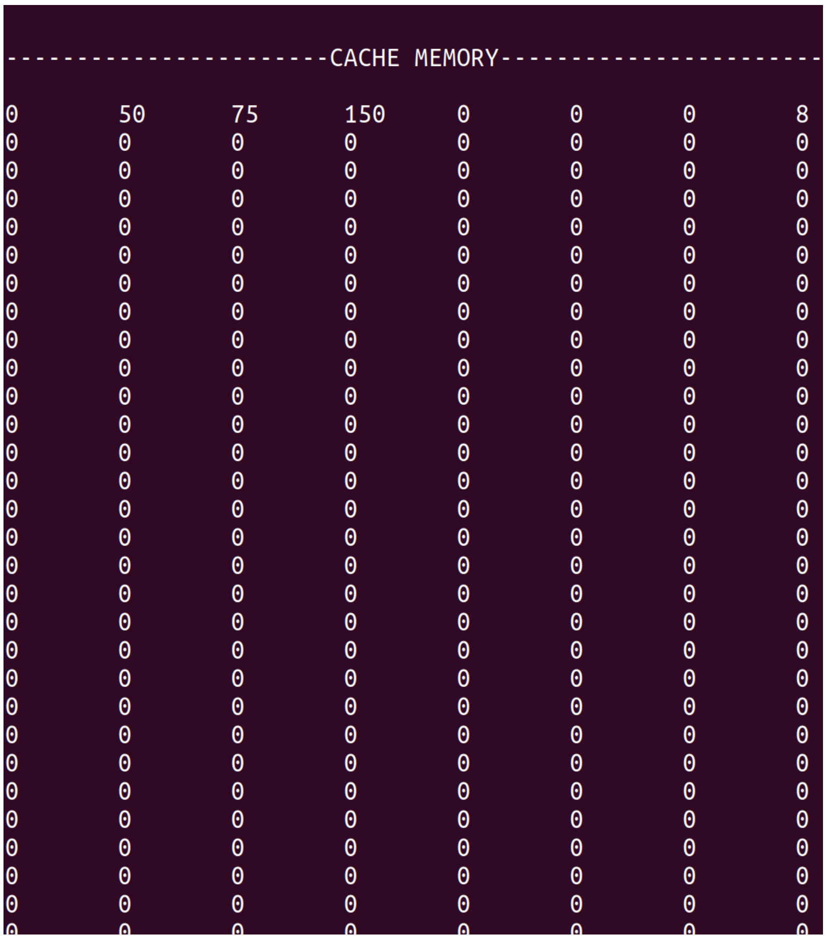
```
Instance 1: IMEM
000000  00 20 0F 93  00 80 00 93  FF FF 8F 93  02 0F 9C 63  40 00 0F 13  01 EF 0F 33  00 0F 0F B3  00 0F A0 03  01 FF 0F B3  00 0F A0 03  . .............c@......3...............
00000a  01 FF 0F B3  00 0F A0 03  01 FF 0F B3  00 0F A0 03  01 FF 0F B3  00 0F A0 03  00 00 00 6F  00 11 2E 23  00 81 2C 23  02 01 04 13  ..........................o...#..,#....
000014  04 B0 07 93  FE F4 24 23  03 20 07 93  FE F4 22 23  FE 84 27 03  FE 44 27 83  00 E7 D4 63  00 07 07 93  FE F4 26 23  FE C4 27 83  ......$#. ...."#..'..D'....c......&#..'.
00001e  FE 84 25 83  00 07 85 13  0D 40 00 EF  00 05 07 93  00 07 9E 63  FE C4 27 83  FE 44 25 83  00 07 85 13  0B C0 00 EF  00 05 07 93  ..%......@.........c..'..D%............
000028  00 07 8A 63  FE C4 27 83  00 17 87 93  FE F4 26 23  FC 5F F0 6F  00 00 00 13  00 00 07 93  00 07 85 13  01 C1 20 83  01 81 24 03  ...c..'.......&#._.o............. ...$.
000032  02 01 01 13  00 00 80 67  06 05 40 63  06 05 C6 63  00 05 86 13  00 05 05 93  FF F0 05 13  02 06 0C 63  00 10 06 93  00 B6 7A 63  .......g..@c...c..............c......zc
00003c  00 C0 58 63  00 16 16 13  00 16 96 93  FE B6 6A E3  00 00 05 13  00 C5 E6 63  40 C5 85 B3  00 D5 65 33  00 16 D6 93  00 16 56 13  ..Xc.........j........c@.....e3......V.
000046  FE 06 96 E3  00 00 80 67  00 00 82 93  FB 5F F0 EF  00 05 85 13  00 02 80 67  40 A0 05 33  00 B0 48 63  40 B0 05 B3  F9 DF F0 6F  .......g....._.........g@..3..Hc@......o
000050  40 B0 05 B3  00 00 82 93  F9 1F F0 EF  40 A0 05 33  00 02 80 67  00 00 82 93  00 05 CA 63  00 05 4C 63  F7 9F F0 EF  00 05 85 13  @...........@..3...g.......c..Lc.......
00005a  00 02 80 67  40 B0 05 B3  FE 05 58 E3  40 A0 05 33  F6 1F F0 EF  40 B0 05 33  00 02 80 67  00 00 00 00  00 00 00 00  00 00 00 00  ...g@.....X.@..3....@..3...g...........
000064  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ........................................
00006e  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ........................................
000078  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ........................................
```

(a) Loaded Instruction Memory



```
Instance 0: DMEM
000000  00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 96 00 00 00 4B 00 00 00 32 00 00 00 0D  .......................K...2....
000001  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000002  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000003  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000004  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000005  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000006  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000007  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000008  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
000009  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
00000a  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
00000b  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
00000c  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
00000d  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................................
```

(b) Data Memory



```
-----------------------CACHE MEMORY-----------------------

0       50      75      150     0       0       0       8
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
```

(c) Primary Cache Memory

Figure 5: Monitoring the Memory Contents

## 4.3 Memory Hierarchy Verification

Cache functions are verified completely using a verilator script and the result are as follows.

```
HELLO... CACHE CONTROLLER HERE...!
CLOCK NUMBER 00000101
@echo: Variables Are Properly Initiated.

FIRST WRITE REQUEST --> ADDR:16, DATA:12
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00000110
@echo: CPU can proceed to the next instruction from here.

LOADING THE SAME DATA (ADDR: 16)
LOADED SUCCESSFULLY...!
CLOCK NUMBER 00001100
16:     00000000000000000000000000001100
CPU DATA IN LINE:       00000000000000000000000000001100
@echo: Observe that the data is only available after five more clock cycles from
Write Request.
@echo: This is because controller will consider the line valid only after
neighbouring elements are fetched from Main Memory

CACHE LINE:     1
TAG:    0000000
DATA:   00: 00000000000000000000000000001100    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1


WRITE REQUEST ON THE SAME LINE...! --> ADDR: 2064, DATA: 23
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00001101

-----------ANALYZING THE MEMORY CONTENT----------------
CLOCK NUMBER 00010011
@echo: Overiding the cache line, forced the content in the first primary cache line
to a line in the victim cache.

CACHE LINE:     1
TAG:    0000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1

VICTIM LINE: 0
TAG:    00000000000001
DATA:   00: 00000000000000000000000000001100    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   11
```

```
READING THE EARLIER DATA AGAIN(ADDR: 16)
LOADED SUCCESSFULLY...!
CLOCK NUMBER 00010110
16:     00000000000000000000000000001100
CPU DATA IN LINE:       00000000000000000000000000001100

-----------ANALYZING THE MEMORY CONTENT----------------
CLOCK NUMBER 00010111
@echo: Victim Hit: Data will be swapped between primary cache and victim cache.

CACHE LINE:     1
TAG:    0000000
DATA:   00: 00000000000000000000000000001100    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1

VICTIM LINE: 0
TAG:    00000010000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   11

FILLING THE VICTIM...!
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00011000
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00011110
LOADED SUCCESSFULLY...!
CLOCK NUMBER 00101010
28688:  00000000000000000000000000000000
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00101011
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00110001
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00110111
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 00111101

@echo: Victim is filled while leaving a line clean(Load Instruction).
MEMORY LINE:    3586
DATA:   00: 00000000000000000000000000000000    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
-----------------------VIEWING THE VICTIM CONTENT------------------------------

VICTIM LINE: 0
```

```
TAG:    00000010000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 1
TAG:    00000000000001
DATA:   00: 00000000000000000000000000001100    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 2
TAG:    00011000000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 3
TAG:    00000110000001
DATA:   00: 00000000000000000000000000100010    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 4
TAG:    00011100000001
DATA:   00: 00000000000000000000000000000000    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  0
NMRU:   00


VICTIM LINE: 5
TAG:    00000100000001
DATA:   00: 00000000000000000000000000101011    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
```

```
NMRU:   01


VICTIM LINE: 6
TAG:    0001010000001
DATA:   00: 00000000000000000000000000110001    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   10


VICTIM LINE: 7
TAG:    0001000000001
DATA:   00: 00000000000000000000000111011110    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   11

------------------------------- VICTIM END -------------------------------------

WRITING ONE MORE DATA ON THE SAME LINE.
DATA WRITE REQUESTED SUCCESSFULLY...!
CLOCK NUMBER 01001101
@echo: Controller will choose the dirty less line and Override it(Best option since
no need for a WRITE_BACK stage)
-----------------------VIEWING THE VICTIM CONTENT------------------------------

VICTIM LINE: 0
TAG:    0000010000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 1
TAG:    0000000000001
DATA:   00: 00000000000000000000000000001100    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 2
TAG:    0001100000001
DATA:   00: 00000000000000000000000000010111    01: 00000000000000000000000000000000
```

```
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 3
TAG:    00000110000001
DATA:   00: 00000000000000000000000000100010    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 4
TAG:    01111110000001
DATA:   00: 00000000000000000000100100010011    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   11


VICTIM LINE: 5
TAG:    00000100000001
DATA:   00: 00000000000000000000000000101011    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   00


VICTIM LINE: 6
TAG:    00010100000001
DATA:   00: 00000000000000000000000000110001    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   01


VICTIM LINE: 7
TAG:    00010000000001
DATA:   00: 00000000000000000000000111011110    01: 00000000000000000000000000000000
        10: 00000000000000000000000000000000    11: 00000000000000000000000000000000
VALIDITY:       1
DIRTY:  1
NMRU:   10
------------------------------- VICTIM END -------------------------------------
```

## A Contributions

| PERSON | CONTRIBUTION |
|---|---|
| Tharindu O.K.D - 190622R | Design and Implementation of Control Unit in Chesel. Data Path Design for CPU. Verilator Simulation of the CPU. FPGA board Implementation |
| Sanjith. S - 190562G | Design and Implementation of Memory Hierarchy in verilog. Verification of Cache Controller with Verilator. |
| Kajhanan. K - 190286M | Design of Memory Unit. Verilog Implementation of Eviction Unit. |
| Yasarathna D.D.K.B. - 190719V | Verilog Implementation of ALU and Compiler Design |
| Wansooriya W.H.O. - 190664V | Compiler Design |

## References

1. Computer Organization and Design: the Hardware/Software Interface: Third Edition.
2. Instruction Set
3. A verilog based implemetation
4. What is RISC architecture?
5. RISC-V RV32I Instruction Encoding
6. Two-level Cache controller implementation