# PROJECT REPORT

*RISC-V Simulator + Cache Simulator*

**Group Members:**

1. **VUPPULA HEMANTH REDDY**
2. **SAMPADRAM KUMAR JOGADENU**

## INTRODUCTION

A cache-enabled RISC-V simulator simulates the behaviour of a RISC-V processor while including a cache to make memory access faster. It runs RISC-V instructions while modelling the behaviour of the cache, its hits and misses, and it supports replacement policies of LRU, FIFO, and RANDOM. This setup reflects real-world processor-memory interaction; hence, it is useful for testing performance, exploring cache policies, and understanding CPU-memory.

## APPROACH

The simulation is achieved by parsing through the file, saving the labels separately into arrays and then converting unnecessary syntax into spaces and then tokenising, followed by handling various instructions based on the first 'token', which represents the instruction operand and executing it.

## IMPLEMENTATION

### The run.c implementation:

The run.c file is responsible for checking the type of instruction and executing it. It contains a single function called run_instructions. When this function is called, it checks the first term of the instruction (named tokens[0]) with the available instructions in RISC-V. After finding the instruction, it generates the source and

destination register using a utility function in functions.c and then executes the instruction with the values of the registers from the array named register_values.

For I type instructions, the immediate value is calculated using the atoi / strtol function. For simple R and I instructions, it performs the corresponding arithmetic operation and stores the result in the destination register.

A struct type variable named MemEntry with attributes 'address' and 'values' is created to handle memory. The address works as an index value, and the value corresponds to the bits. For load instructions, the corresponding number of 2-digit hex representations (e.g., 8 values for ld, 4 values for lw, etc.) are taken with bit shifting to recreate the value from the mem_values array, and then this value is saved into the register.

For unsigned type instructions, a uint value is used and then type cast to extend it to a 64-bit value in the register. For store type instructions, the register value is shifted, creating a 2-digit hex representation, which is then saved into the mem_values array. This ensures that the value of the register is split into one byte each and then saved into the memory with the corresponding bytes being stored.

When a lui instruction is encountered, the immediate value is shifted to the right by 3 bits and then sign-extended to 64 bits.

To execute a branch instruction, it first checks whether the instruction contains a numerical value as the immediate or if there is a label. If it is an immediate value, it uses the immediate value and updates the pc value and line number so that the jump can occur (after testing the jump condition). If it is a label, then it searches for the label in the array named label_names and finds its line number. It then finds the immediate value and the corresponding pc value and updates it. Then it executes it in the same way as before. If the branch instruction is an unsigned branch, it checks the unsigned values of the register value for the branch condition.

Finally, there are 2 more instructions: the jal and the jalr instruction. The jal instruction execution is similar to the execution of a branch type, as it is just an unconditional branch. The only difference is that when running a jal instruction with a label in it, the label information gets pushed into the stack. Also, the value

of the current pc + 4 is stored in the register. This value is used to return control after the execution of the function in the label. In the jalr instruction, it returns to the address in the register in brackets and saves the pc + 4 into another register, although saving is not necessary as it may not be used later on. Also, it pops the stack when encountering a jalr instruction.

### Cache Simulation:

To model cache in our project, we have defined three structs:
1. cacheblock - this models  a block in a cache

2. cachesets - this models a “set” in cache. This is used for modeling the associativity of cache. Each set has n no. of blocks where n is associativity.

3. Cache - this is the actual cache implementation as a struct.

A variable named cache_in decides whether we want to enable cache or disable cache. Even if the cache is enabled, only load and store instructions are effected by  cache modeling.

When we encounter a load instruction, we first separate the address into byte offset, index (if it is not fully associative) and tag. Now we match the index and tag to search whether the address is present in the cache or not. In case of fully associative mapping we only need to match the tag (there is no index). If we found the address, it is a hit or else it is a miss.

To achieve this reading we have written a function named cache_read(). This function checks whether the current access is a hit and if it is a miss, which block is to be replaced. For replacing the block , if we encounter an empty block that block is filled, else based on the replacement policy we have three functions:

(a) fifo() - this implements the fifo policy, to achieve this we have used a queue (modeling it as an array) since a queue also follows the fifo policy.
(b) lru() - this implements the LRU policy. This is similar to the fifo policy, except that if we were to access the same block again, the “age” - lru_counter of the block will be reset. To implement this we have written a

function lru_counter_update()  which updates the lru_counter on each access whether it is a hit or miss appropriately.

(c) random_policy() - this implements the random policy. In this the block to be replaced if full is simply a random block generated (by the random function).

In case of store instructions, we have 2 different policies:

1. Write through: In the write through case if we have a hit, we update both memory and cache(attribute of block in struct) with the new value from the store instruction.  In case of a miss, we directly just update the value to the memory and don't do anything in the cache.
2. Write back: In this case, if we have a hit, we only update the data in cache and mark it as dirty, if we get a miss, since we are following the write-allocate policy, we bring the correct block from memory and update the data only in cache. In both cases, if the data value has been changed then we mark the block as dirty - set the dirty attribute of block as 1.

### The stack.c file:

The stack.c file is used to create the stack and its properties/attributes required for the show-stack and to define the push, pop and top functions.

### The main.c file:

The main.c file contains the main function, which parses through a file and keeps the terminal in an infinite loop, waiting for commands. It also simulates the commands. When in the infinite loop, it waits for one of the following instructions:

1. load <file_name>:

   This command loads the input file and initializes all the memory values and register values to 0. It reads all the lines from the file and saves a copy of each line into an array named array_of_lines. This array is used for easy

parsing and string operations. The stack is also created and initialized here. Additionally, the break 'switches' are initialized to zero, and the main label is pushed into the stack for incrementing while running.

2. run:

   For each instruction in the array_of_lines array, it parses the instruction, tokenizes it, and removes the labels to have only the instruction. The .data section is handled here. Depending on the type (e.g., .dword or .word), the values in that line are parsed and saved in little endian format in the memory. An if loop is implemented to check for breakpoints and stop execution at those points. After looping through all the lines, the stack is popped to empty it.

3. step:

   This command has the same execution as the run command, except that a variable "stepper" is implemented to run an instruction every time the step command is input and incremented. The data section is loaded into memory only when the step command is used just after the load command, since otherwise the values are already loaded into memory. At the end, the stack is popped to empty it.

4. regs:

   This command prints the values in the registers in 64-bit hex format.

5. mem <address> <count>:

   This command prints the values in the memory from the given address to address + count - 1.

6. exit:

   This command exits the simulator and prints a message saying it has exited.

7. break <line>:

   This command is used to handle breakpoints. An array is created to hold

the 'switch' of that line number. The switch value corresponds to 0 if there is no breakpoint at that line number, and 1 if a breakpoint is set at that line number.

8. del break <line>:

   This command changes the value of the switch from 1 to 0 at the specified line number to denote that the breakpoint has been deleted.

9. show-stack:

   This command prints the label name and last executed line number.

10. cache_sim enable <filename>:

    This command enables the cache (cache_in = 1), and reads the file to get the details for modeling the cache.

11. cache_sim disable:

    Disables the cache by setting cache_in = 0 and clears the cache

12. cache_sim status:

    Prints the information about the cache

13. cache_sim invalidate:

    Evicts the dirty blocks into the memory and set the valid bit for each block as 0.

14. cache_sim dump <file_name>:

    Prints the information whether the block is dirty or clean for all blocks.

15. cache_sim stats:

    Prints the number of hits, misses and the hit rate for the simulation.

### The functions.c file:

The functions.c file contains all the utility functions required by any of the other files. This file includes the following functions:

1. `char **string_split(char *string)`: This function splits the string into substrings using spaces as the delimiter.
2. `int non_int_char_finder(char *str)`: This function returns 1 if it finds that there are no digits in the string.
3. `int ischarinstring(char *string, char x)`: This function searches for the given character x in the string and returns 1 if found.
4. `int register_finder(char *str)`: This function returns the integer number corresponding to the register number of the argument.
5. `char* deepCopyString(char* str)`: This function is used to deep copy a string so that we can use the copy without modifying the original.
6. `char* trim_space(char* string)`: This function is used to trim any spaces at the start of the line.

## LIMITATIONS/ASSUMPTIONS:

1. Assumed that there is no empty label
2. Assumed that there is no blank line
3. Instruction is assumed to be syntactically correct
4. Assumed that the values in .data are in the same line as corresponding .dword / .word etc
5. Assumed that there is no label in the .data section.
6. The register values have been printed in 64-bit hex format with extended sign bit instead of precise values like 0x1, 0xff.
7. In show-stack when no instruction has been executed main displays the .text line number.
8. Break line numbers and show-stack line numbers are counted from .data line. This corresponds to the line number in the file.
9. Break line must not be given in the .data to .text lines.
10. Only implemented FIFO,LRU,RANDOM replacement polices.this doesn't support any other replacement policies.
11. Only implemented (Write-back,Write-allocate) and (Write-through,No Write-allocate) policies for write policies.hence,this doesn't support other write policies.
12. Assumed that there are cache accesses doesn't span across multiple blocks.

## ERROR HANDLING:.

Since it is considered that the instructions are syntactically correct, not much error handling has been done for these cases.

1. If the "del break" command is used without any breakpoints, the message "no breakpoint exists" is printed.
2. If the input file doesn’t open or does not exist in the current directory, an error message is thrown in the terminal and is exited from the simulator.

## CHALLENGE FACED:

1. To handle memory we had to create a new struct type and introduce a lof of if-else conditions.
2. We faced difficulty while printing the execution line because of a \r character in the input line, because of this the lines were being jumbled. For this we added it as a delimiter to the strtok function.
3. Carrying over the line number from run to step to continue executing the code was challenging and we tried many different variables.
4. We faced difficulty while implementing the cache blocks initially and had to rewrite it after finding a major flaw in it.
5. We had to modify or cache_read function such that it supports all load instructions.To implement this we gave a void* result and a int resultsize as arguments to cache_read and typecast result inside the function to corresponding data type.
6. In cache_write,we had trouble while loading the store_value into cache with corresponding address.
7. While implementing random policy,we found that our random policy function always returns a same block number to replace in a single execution of code.we later found that srand(time(NULL)) should only be called once per execution to get a new random block number for each call of random policy function.

## TEST-CASES:

1. **Sample test case 1 (without cache):**

```
.data
.dword 1,6,12
.text
    lui x3,0x10
    addi x9,x3,0x200
    ld x10,0(x3)
    addi x11,x11,1
    addi x10,x10,1
l1:beq x10,x11,exit
    slli x12,x11,1
    addi x12,x12,-1
    slli x12,x12,3
    add x12,x3,x12
    ld x4,0(x12)
    ld x5,8(x12)
    beq x4,x0,zerogcd
    beq x5,x0,zerogcd
    blt x4,x5,div
    ld x4,8(x12)
    ld x5,0(x12)
div:blt x5,x4,l2
    sub x5,x5,x4
```

```
    beq x5,x0,l3
    beq x0,x0,div
l2:addi x6,x4,0
    addi x4,x5,0
    addi x5,x6,0
    blt x4,x5,div
zerogcd:addi x4,x0,0
l3:addi x13,x11,-1
    slli x13,x13,3
    add x9,x9,x13
    sd x4,0(x9)
    sub x9,x9,x13
    addi x11,x11,1
    beq x0,x0,l1
exit:add x0,x0,x0
```

2. **Sample test case 2 (without cache):**

```
.data
.dword 1,2,3
.text
lui x3, 0x10
ld x4, 0(x3)
ld x5, 8(x3)
jal x1,loop
add x6,x5,x4
```

```
beq x0,x0,exit

loop: addi x4,x4,2

addi x5,x5,2

jalr x0, 0(x1)

exit:add x0,x0,x0
```

3. **Sample test case 3 (for cache)**

```
.data

.dword 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8

.text

lui x3,0x10

ld x4,0(x3)

ld x5,16(x3)

ld x6,32(x3)

ld x7,48(x3)

ld x8,64(x3)

ld x10,16(x3)

ld x9,80(x3)

ld x12,0(x3)

ld x12,32(x3)
```

4. **Sample test case 4(for cache)**

```
.data

.dword 8, 8

.text
```

```
lui x3, 0x10
ld x4, 0(x3)
ld x5, 8(x3)
addi x3, x3, 16
addi x10, x0, 0
add x12, x3, x0
If1: beq x4, x10, end1
addi x11, x0, 0
slli x15, x10, 3
add x12, x3, x15
If2:    beq x5, x11, end2
sd x20, 0(x12)
slli x15, x5, 3
add x12, x12, x15
addi x11, x11, 1
beq x0, x0, If2
end2:   addi x10, x10, 1
beq x0, x0, If1
end1: add x0, x0, x0
```

The above cache related sample test cases were tested with various configurations of caches.some of those configurations were:

1. 64,32,1,LRU,WB
2. 64,32,0,FIFO,WT
3. 1024,16,2,RANDOM,WB
4. 256,16,0,LRU,WB etc

## CONCLUSION

This project successfully implements RISC-V assembly code with cache modeling. It takes input from the terminal and processes the given commands. The supported commands include load, run, step, exit, mem, show-stack, break, del break, cache_sim enable <filename>,  cache_sim disable, cache_sim status, cache_sim invalidate, cache_sim dump <file_name>, cache_sim stats. This project has improved our understanding of how RISC-V assembly code is executed in RIPES, how cache works in computers ,how to implement caches etc.It has also helped us enhance our coding skills in the C language.