IMPLEMENTATION OF CPU WITH CACHE MANAGEMENT

NOVEMBER 26

CSN-221 COMPUTER ARCHITECTURE AND ORGANISATION

Contents

TEAM MEMBERS

NAME	ENROLLMENT NUMBER
1. ACHINTYA NATH	20114003
2. AKSHAT AGARWAL	20113018
3. ALAPATI SRIVINAY	20114005
4. ARYA ANAND	20114017
5. KUMAR DEVESH	20114047
6. NALLA JAGANNATH REDDY	20114060
7. PRAFUL SINGHAL	20112079
8. UJJAWAL KUMAR DUBEY	20115160

Problem Statement

Design a CPU including ALUs and register files, the control circuitry, Instruction's flow etc, on Logisim simulator. Having memory management, cache management, etc would get additional points. Evaluate rigorously. BENCHMARK evaluations are preferred.

Our Implementation

In our implementation designed a CPU consisting of a P*rogram Counter*, R*egister File, Control Unit, Main Memory with Cache Management.*

We have designed a *direct cache mapping* to bring down the average access time and would take instructions of load and store from the CPU and perform the required operation after searching for the address in it.

The CPU has a custom opcode of 5-bits supporting 21 instructions including branch and memory access that will issue instructions to the Cache, the memory module, and to the Arithmetic and logic Unit so that suitable computations will be performed.

The following report describes the architecture of the implementation and the flow of data and instructions with the help of diagrams and screenshots taken from our design along with the relevant details.

The novelty of work done

In accordance with the problem statement, we build a CPU with cache management to reduce memory access times along with all the functionalities that a processor has including all arithmetic and logical operations that can be performed on the implemented processor including branch operations. We test our project implementation on all possible modes of failure to debug the implementation and provide exhaustive documentation of the working of our project.

We design the cache modules from scratch starting from the design of the digital logic for memory read, write operations and cache read, write operations, loading data from memory to cache and the eviction from the cache, the data path, and the way the signals change during dynamic execution and the working of the tag array in storing the tags and the control signals required for efficient working of the cache.

The working of the cache depends on a lot of inputs, such as, whether the input instruction is load or store, whether it results in hit or miss, whether the modified bit is set or whether the valid bit is set. Drawing different sub-circuits for each of the combinations of inputs would result in a lot of sub-circuits. Apart from this, there will be a lot of repeated sub-circuits. We tried to categorize the input combinations into different states and finally arrived at the following eight broad states in which our cache can be:

State 0: The instruction given by the CPU is neither load nor store. Nothing is done in this state.

State 1: Load, and cache hit (modified bit can be 0 or 1) – We simply load the data from the cache array into the destination register.

State 2: Load, Cache Miss, and Modified Bit are 0: In this case, we noticed that although there is a miss, we don't need to write back to memory as the modified bit is zero. All we need is to bring the block and store it in cache and then load it into the destination register. Our goal here was to reduce the sub-circuitry. We load the data into the cache array and update the tag in the tag array. Now, in the next clock cycle, the tag comparison takes place but now this is a cache hit, and the state changes to State 1.

State 3: Load, Cache Hit, and Modified Bit is 1: In this case, we need to write the data back in the main memory, load the new block and then produce the output result. For eviction, we write the data back in the main memory and after this, we update the tag's (which was stored in the tag array) modified bit to 0. We set the modified bit as zero because the main memory and data array have the same set of data. This now reduces it back to state 2. In this way, we tried converting current states to previous states. This reduced a lot of implementations and prevented the repetition of circuits.

State 4: Store and cache hit (Modified bit can be 1 or 0): We simply write the data in the data array and set the modified bit to 1.

State 5: Store, cache miss, and modified bit are 0: This is similar to state 2, we load the data from main memory to data array and update the tag. In the next clock cycle, the comparison results in a hit and it now reduces to state 4.

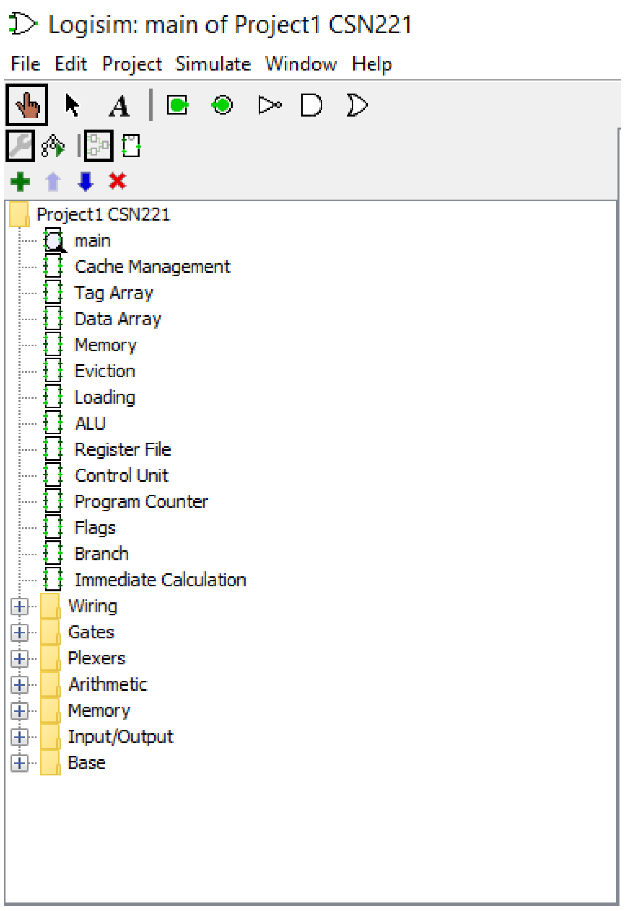
State 6: Store, Cache Miss, and Modified Bit are 1: In this case, we need to write the data back in the main memory, load the new block and then store the new data. For eviction, we write the data back in the main memory and after this, we update the tag's (which was stored in the tag array) modified bit to 0. We set the modified bit as zero because the block in the data array has the same value as a block in the main memory. This now reduces it back to state 5.

State 7: Valid bit is 0. We simply treat this as a missing case (even though if the comparison results in cache hit). This reduces this state to either State 2 or State 5, depending on whether the instruction is load or store.

In this, we tried to reduce cache from one state to another in order to *reduce the complexity of implementation*. It also helped us to r*educe repetitive circuits in our implementation*.

Modular Implementation

While preparing the initial design of the project and the modules to be made, we ensured that the modules being implemented were loosely connected and performed independent functions, and could easily be integrated into a single main circuit when the parts of the module were to be assembled. This modularized implementation helped us a lot in debugging our processor as we could easily pinpoint the module in which the desired behaviour was not observed and then rectify it.


Logisim: main of Project1 CSN221
File Edit Project Simulate Window Help

Project1 CSN221
- main
- Cache Management
- Tag Array
- Data Array
- Memory
- Eviction
- Loading
- ALU
- Register File
- Control Unit
- Program Counter
- Flags
- Branch
- Immediate Calculation
- Wiring
- Gates
- Plexers
- Arithmetic
- Memory
- Input/Output
- Base

The modules implemented in this project are:

- main module - This module assembles all of the individual modules in the project into a single module to get the CPU working.
- Cache management module - It contains all the modules related to the implementation of cache
- Tag array module - This module contains the tag array and is responsible for storing and managing the tags of the blocks stored in the cache and for generating a hit/miss signal.
- Data array - The data array in the cache stores the blocks of data loaded from the memory.
- Memory module - This module stores the data memory
- Eviction module - This module contains the logic to calculate the address in the data memory where the data needs to be stored and the corresponding address in the cache memory from where the data needs to be evicted.

- Loading module - This module implements a similar logic to calculate the address in the memory from where the data is to be fetched and the corresponding address in the cache memory where it needs to be stored.
- ALU - The ALU performs all the arithmetic and logical operations on operands for a given instruction.
- Register File - The register file contains all the general-purpose registers and return address, stack pointer registers, and the logic to read and write to these registers based on the writeback control signal and the registers to be read from or written to.
- Control unit - The control unit generates all of the control signals needed for the execution of instructions in the CPU.
- Program counter - This module contains the digital logic implementation for the program counter which contains the address of the next instruction to be fetched. In the case of our cache implementation the program counter changes to the next instruction when there is a cache hit and in case of a miss, cache replacement is performed and then there is a cache hit and the next instruction is executed.
- flags - contains the flag registers that store the result of compare instruction for conditional branch statement.
- branch - contains the logic to predict if a branch is taken or not
- immediate calculation - This module calculates the value of the immediate operand and decides the value of the second operand in instruction based on whether it is an immediate value or a value fetched from the register file.

Individual Contributions

ACHINTYA NATH (20114003)

1. Designed Cache Management (Tag Array, Loading, Eviction, Data Array, Modified Bit)
2. Determining cache specifications.
3. Implementing Cache Management (Tag Array, Loading, Eviction, Modified bit, Data Array).
4. Integration of various blocks within cache management.
5. Implementing immediate module and its integration to main circuit. Integration of Cache Management block with main memory
6. Made a presentation on Cache Management.

AKSHAT AGARWAL (20113018)

1. Integrated and Designed of Main Circuit
2. Designed Arithmetic and Logic Unit,
3. Design Branch Unit
4. Designed Flag Register
5. Contributed in the documentation part
6. Made a presentation on the ALU unit
7. Gave a presentation for ALU

ALAPATI SRIVINAY (20114005)

1. Designed the program counter circuit
2. Helped in the integration of main module
3. Designed the instruction memory
4. Made a presentation on the program counter and instruction memory

ARYA ANAND (20114017)

1. Designed the Control Unit
2. Made presentation for Control Unit
3. Made presentation for Immediate Calculation
4. Gave a presentation for Control Circuit
5. Gave a presentation for Immediate Calculation

KUMAR DEVESH (20114047)

1. Contributed to the Integration of Main Module.
2. Digital logical circuit reduction in cache.
3. Added the valid bit to cache and made corresponding changes to tag array logic.
4. Contributed in the cache management documentation.
5. Made a presentation for Instruction Flow.
6. Gave a presentation on Instruction Flow

NALLA JAGANNATH REDDY (20114060)

1. Made a presentation on Testing
2. Gave a presentation on Testing
3. Contributed to memory design
4. Testbenches for evaluation of CPU
5. Contributed to the testing part of documentation

PRAFUL SINGHAL (20112079)

1. Contributed to the integration of Main Module.
2. Designed Register File.
3. Contributed in the documentation part.
4. Made a presentation on Register File.
5. Gave a presentation on Register File.

UJJAWAL KUMAR DUBEY (20115160)

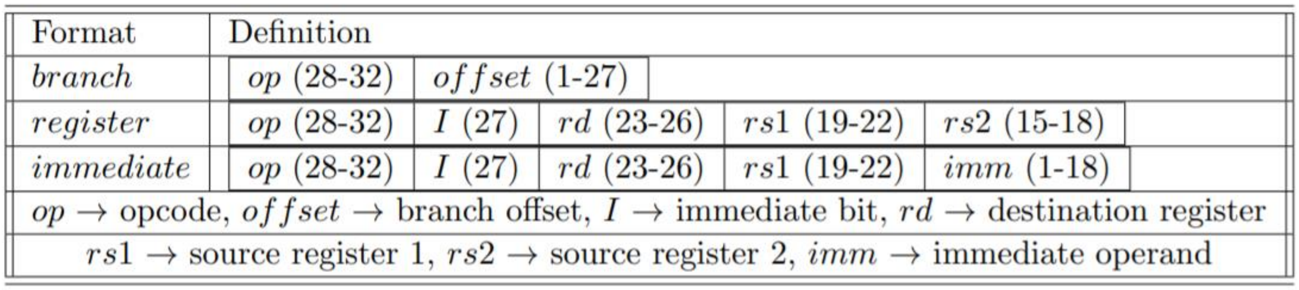
1. Designed Data Array
2. Designed Main Memory, Cache Memory
3. Made a presentation on Data Array and Memory
4. Gave a presentation on Data Array and Memory

Methodology

CPU and Instruction Set Architecture

The CPU architecture we have implemented is that of SimpleRisc which is a RISC-based ISA that has 21 instructions but captures all the functionalities that any standard Instruction Set offers. The SimpleRisc architecture has 16 registers out of which 14 registers are general-purpose registers that can be used to store values during the execution of our testbenches. The other two registers are a stack pointer register and a return address register. Instructions, the values stored in registers and memory are encoded in a 32-bit value. The memory consists of a data memory where the data values are stored and a dedicated instruction memory where the instructions are to be stored.

Apart from this, we have also implemented a dedicated cache management system for caching the data that is being frequently accessed from the data memory for a reduced access time.



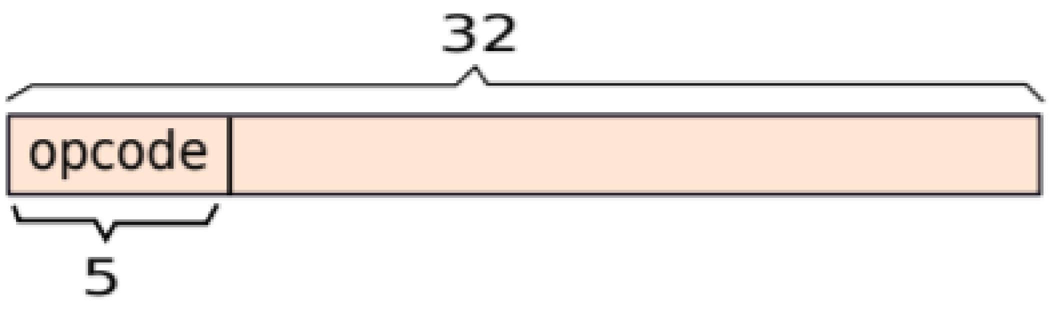
Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*rs*2 (15-18)
immediate	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*imm* (1-18)
$op \rightarrow$ opcode, $offset \rightarrow$ branch offset, $I \rightarrow$ immediate bit, $rd \rightarrow$ destination register					
$rs1 \rightarrow$ source register 1, $rs2 \rightarrow$ source register 2, $imm \rightarrow$ immediate operand					

Instructions for the SimpleRisc architecture are encoded in different formats for different types of instructions that are decoded based on the opcode and the value of the immediate bit I. All instructions have a 5-bit opcode for encoding the 21 instructions.

Branch instructions have a 5-bit opcode and a 27-bit offset value which gives the offset from the current program counter register.

Instructions with register operands are supported where there is a destination register *rd* where the ALU result or the value loaded from memory is stored depending on whether the instruction is a load instruction, or not. In case the instruction is a store instruction, the *rd* register stores the register from which the value is to be fetched to store in the memory and the ALU calculates the effective address in the memory where the data needs to be stored. The source operands in the instruction can be in two formats where either both operands can be register values or one of the operands can be an immediate value present in the instruction.

0 address instruction encoding

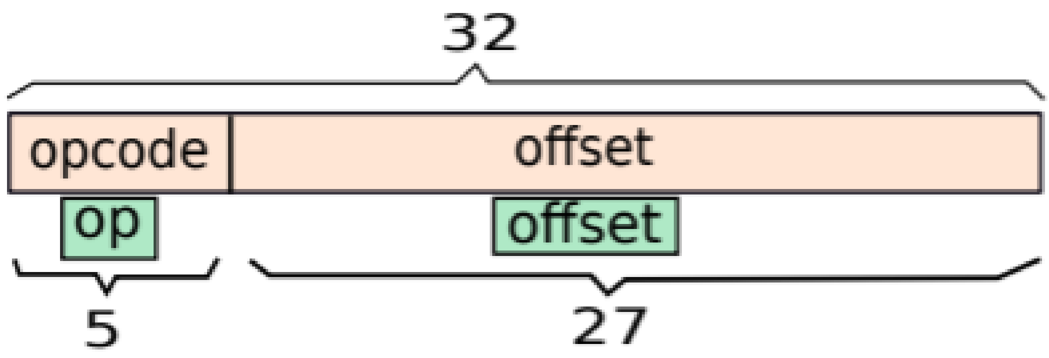


Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

The zero address instructions that have been implemented are:

- *ret* instruction which returns to the return address stored in the *ra* register which stores the value of *pc*+1 for the pc value of instruction when the *call* instruction is encountered.
- *nop* instruction where the instruction is used to introduce a pipeline stall or a delay in execution by not performing any operation.

1 address instruction encoding

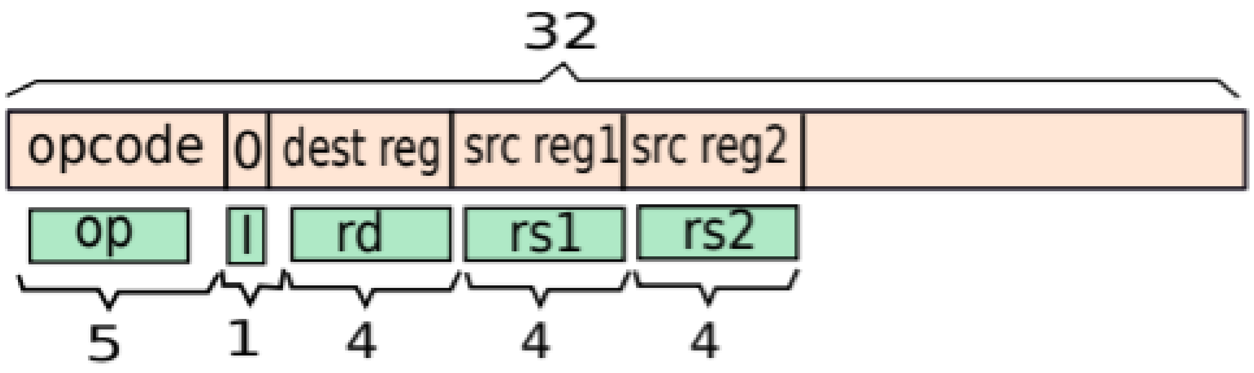


Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

The instructions with a single address that have been implemented are *call, b, beq, and bgt* which have a 5-bit opcode and another 27 bits for calculation of the branch offset based on a *pc* relative addressing.

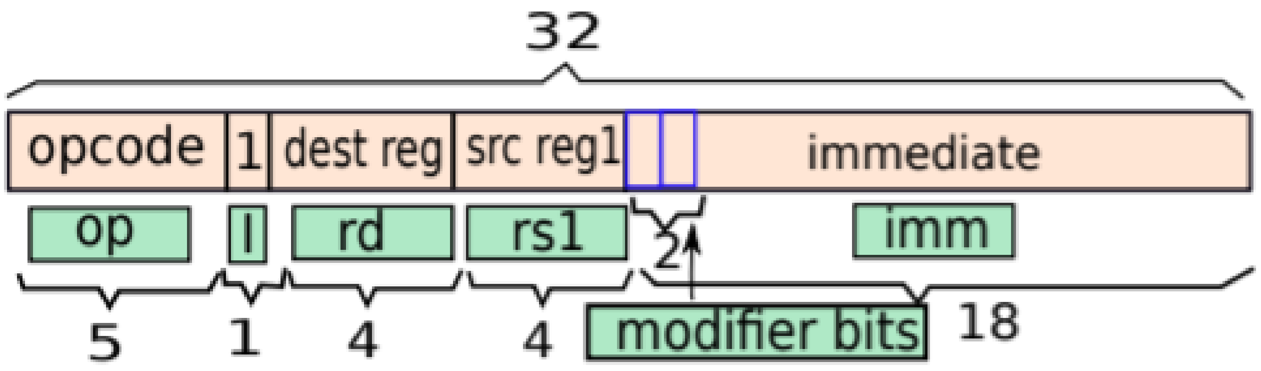
3 address instruction encoding

Instructions encoded in the three-address format are ALU operations. These instructions require two source operands for calculating the result and a destination register to store the results.



Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

A three-address instruction format where both the operands in the instruction are registers.

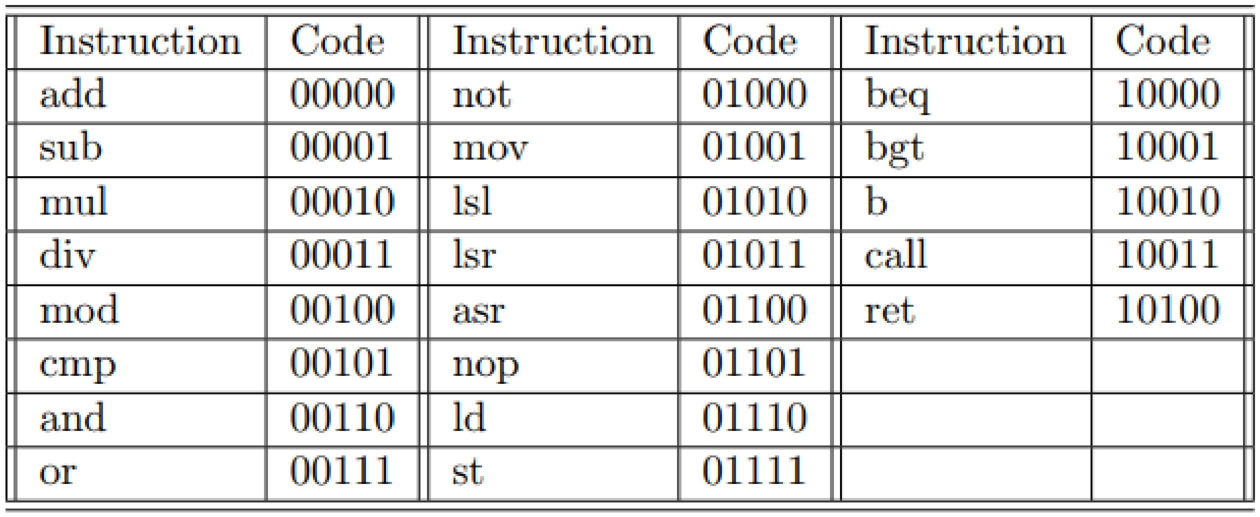


Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

A three-address instruction format where one operand in the instruction is a register and the other is an immediate value which is a number in 2's complement format for 00 modifier bits, an unsigned number for 01 modifier bits and an unsigned number to be loaded in the upper half of register for 10 modifier bits.

There are instructions with 2 operands as well but those instructions are encoded in the three-address format where one address contains a garbage value, for *compare, not,* and *mov* instructions.

Opcodes



Instruction	Code	Instruction	Code	Instruction	Code
add	00000	not	01000	beq	10000
sub	00001	mov	01001	bgt	10001
mul	00010	lsl	01010	b	10010
div	00011	lsr	01011	call	10011
mod	00100	asr	01100	ret	10100
cmp	00101	nop	01101		
and	00110	ld	01110		
or	00111	st	01111		

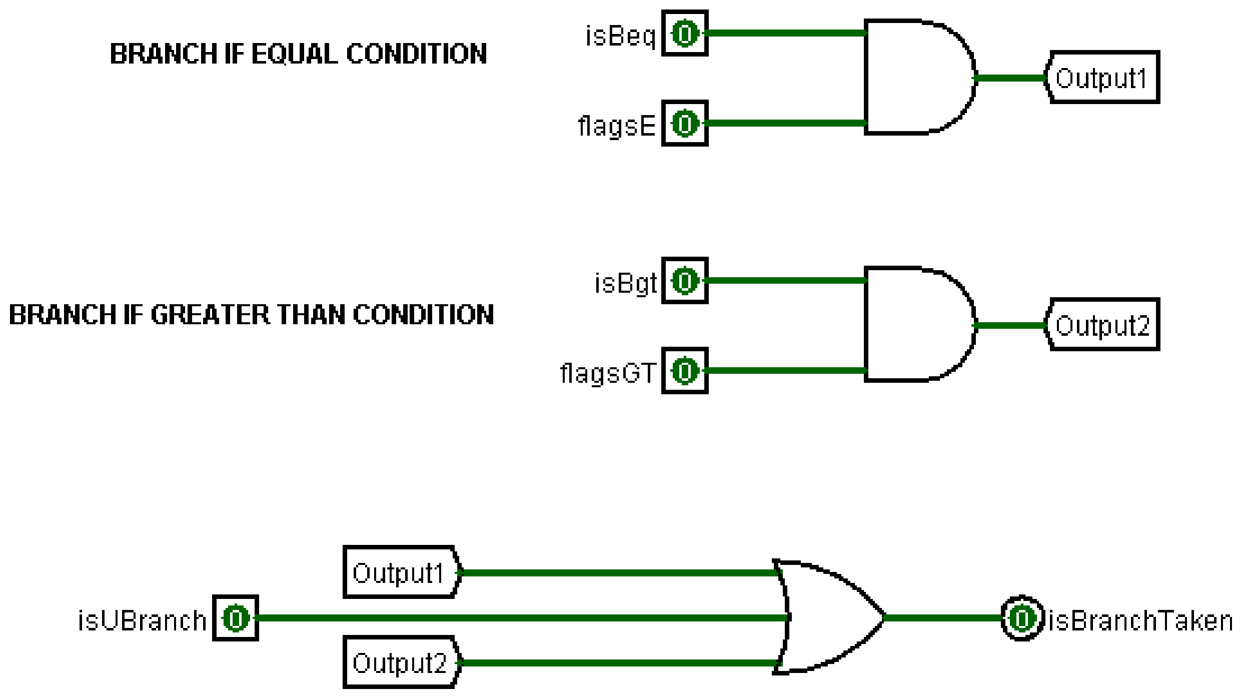
structure of a cache Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

The implemented instruction set consists of 21 instructions with 6 arithmetic instructions: *addition, subtraction, multiplication, division, compare, and mod operation*, 3 logical operations *and, not and or operations* and shift operations. The compare instruction is used to set the flag registers for conditional branch instructions *beq* and *bgt*.

The *call* and *ret* instructions are for making a function call branching unconditionally and for returning from the currently called function.

The *ld and st* instructions are for loading a value from memory and for storing a value into memory respectively based on the implementation of data memory and the cache.

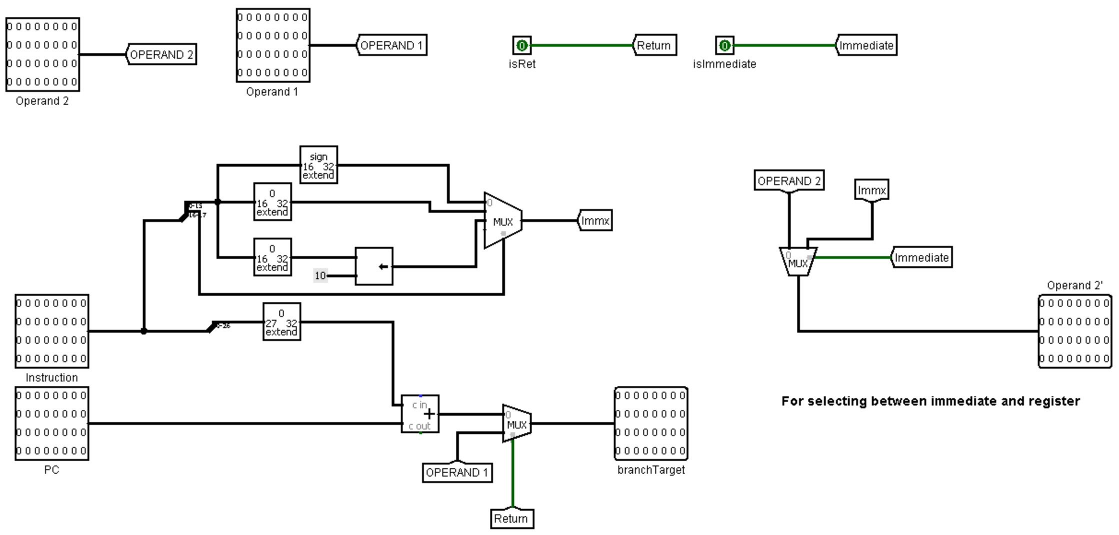
Branch Conditions



The control unit generates the signals *isUBranch, isBeq, isBgt* to check if a branch needs to be taken and sets the *isBranchTaken* value to 1.

If the instruction is a *ret, call, b* then there is an unconditional jump from the current instruction. Similarly, if the instruction is bgt or beq then there is a jump from the current instruction conditional on the flag values. If the flag corresponding to the either of the instructions is 1 then a branch is taken.

Immediate value calculation and Branch Target calculation



Immediate value calculation

The instruction set supported by the architecture that we have implemented consists of instructions where one operand can be an immediate value. Whether the instruction uses an immediate value or not is indicated by the *immediate* bit.

- If the bit is 0 then the operand value decoded from the operand fetch from registers is used as the operand value.
- If the bit is 1 then the *immx* value decoded from the instruction is used as the second operand.

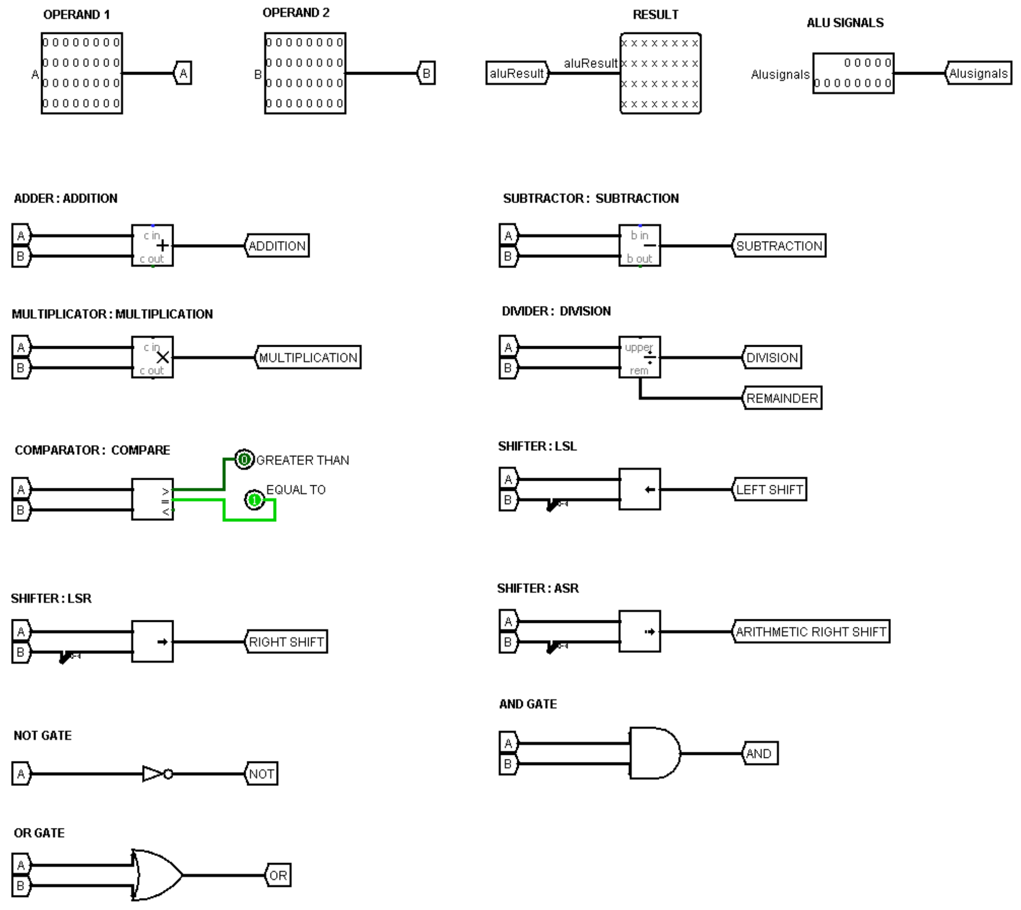
The immediate value is given by the bits 0-15 in the instruction and a 2-bit modifier. The immediate value is a number in 2's complement format for 00 modifier bits, an unsigned number for 01 modifier bits and an unsigned number to be loaded in the upper half of the register for 10 modifier bits.

Branch Target

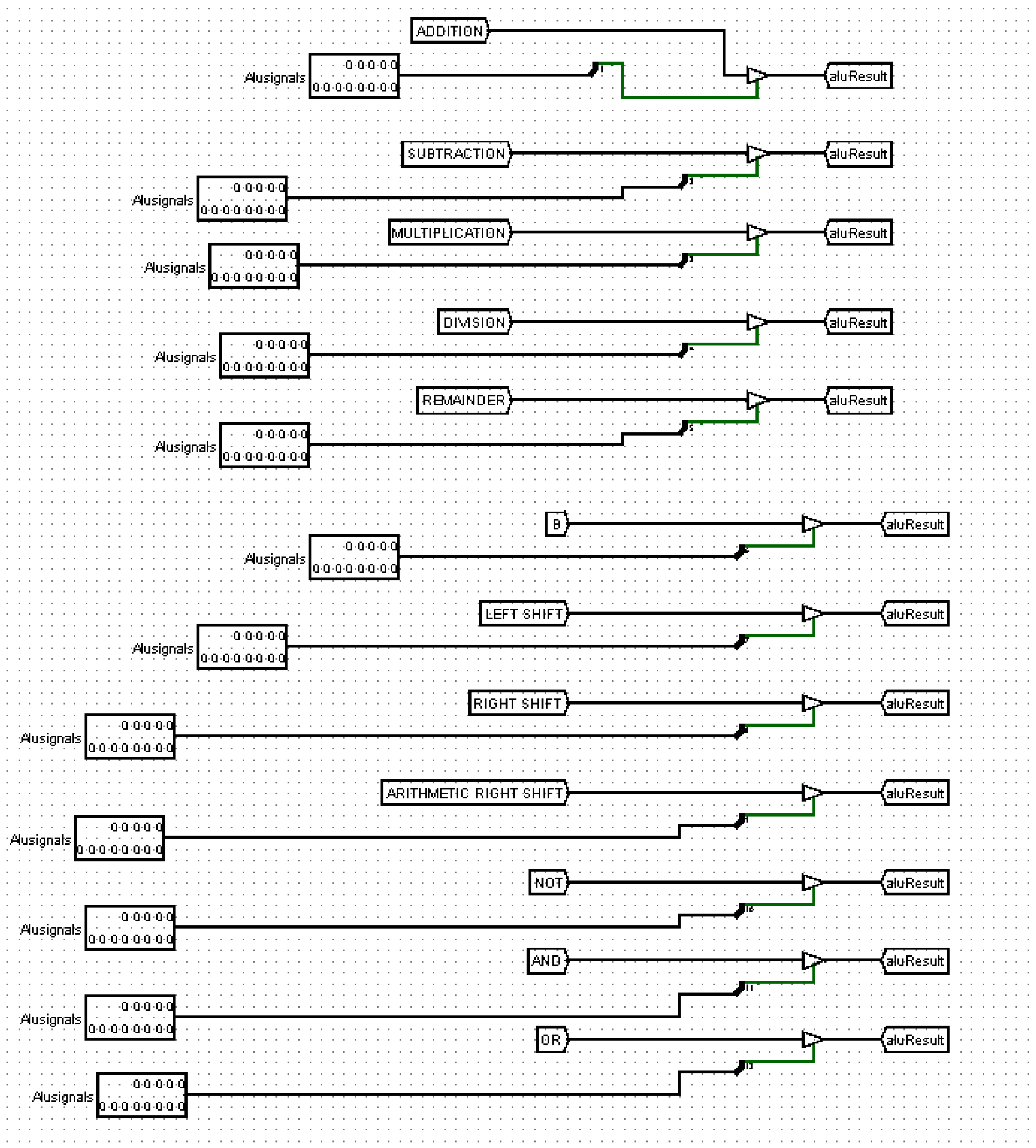
If the branch instruction is a *ret* instruction, then the branch target is the address contained in the *ra* register, fetched as *operand1*. If the instruction is not a return instruction, then the branch target is calculated using the 27-bit branch offset present in the instruction. The branch target is given by the *current pc+offset.*

ARITHMETIC AND LOGICAL UNIT

The ALU takes inputs two operands A and B from the decoded instruction based on their opcodes and performs arithmetic and logical operations on them. The opcode is used to decode the instruction that needs to be performed and the ALU result is chosen conditional on the operation that was supposed to be performed based on the *Alusignals*. The ALU designed requires one clock cycle for performing any of the ALU operations.



The ALU performs the arithmetic and logical operations on the operands A, B to generate the aluResult of the operation performed. The operands A and B are both read from registers, or one can be an immediate value based on the ALU instruction. The ALU performs arithmetic and logical operations based on the ALU signals (13 bits) generated in the control unit. In the case of a load or store instruction, the ALU calculates the effective address in the memory where the load or store operation is to be performed.

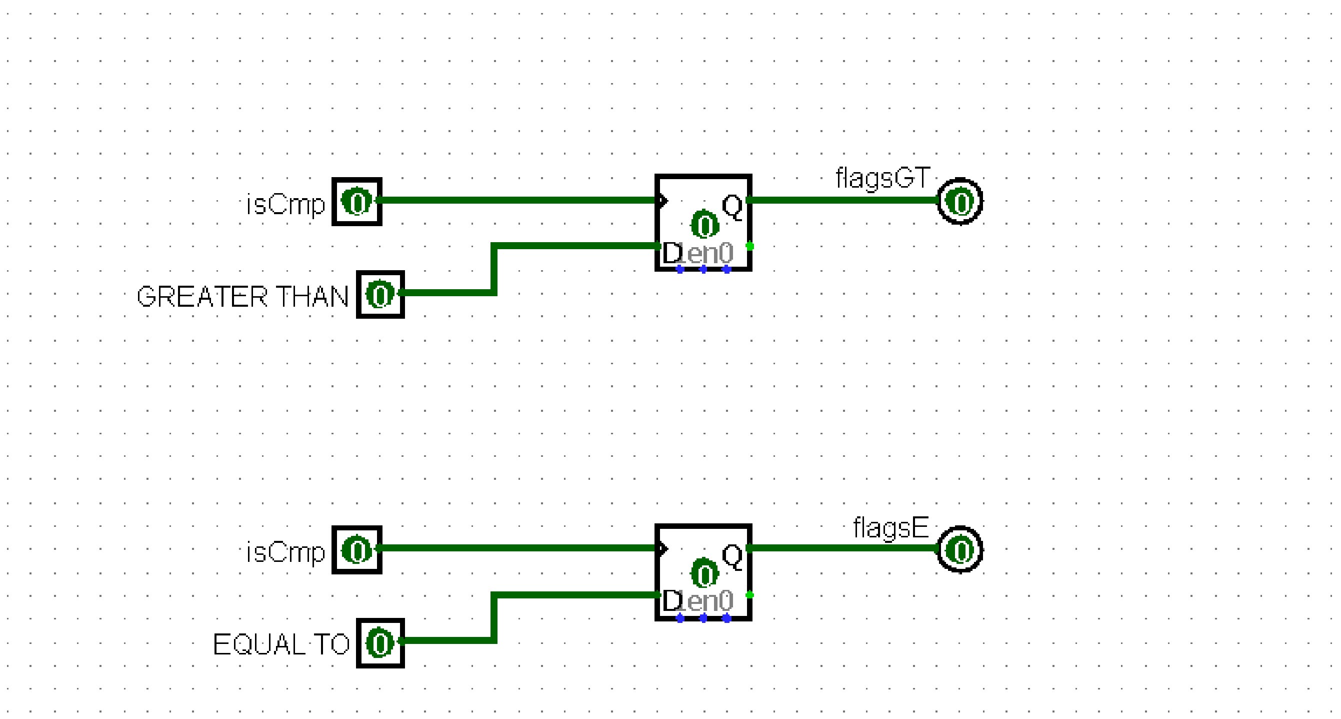


The ALU executes the instructions *add, sub, mul, div, mod, lsl, lsr, asr, not, and, or, mov and cmp* to set the flags for *beq and bgt* instructions.

The comparator module in the ALU has two inputs A and B i.e., the values that need to be compared. It has two output signals which are used to set the values of flag registers for a conditional branch statement.

- **Greater than:** It will be set to 1 if the value of A is greater than the value of B.
- **Equal to:** Its value will be set to 1 if the values of A and B are equal.

Flags



The task of this sub unit is to set the value of the flag registers if a compare instruction is being executed.

There are two flag registers:

1. *flagsGT*: Its value will be 1 if both *isCmp* and GREATER THAN are equal to 1 simultaneously.
2. *flagsE*: Its value will be 1 if both *isCmp* and EQUAL TO are equal to 1 simultaneously.

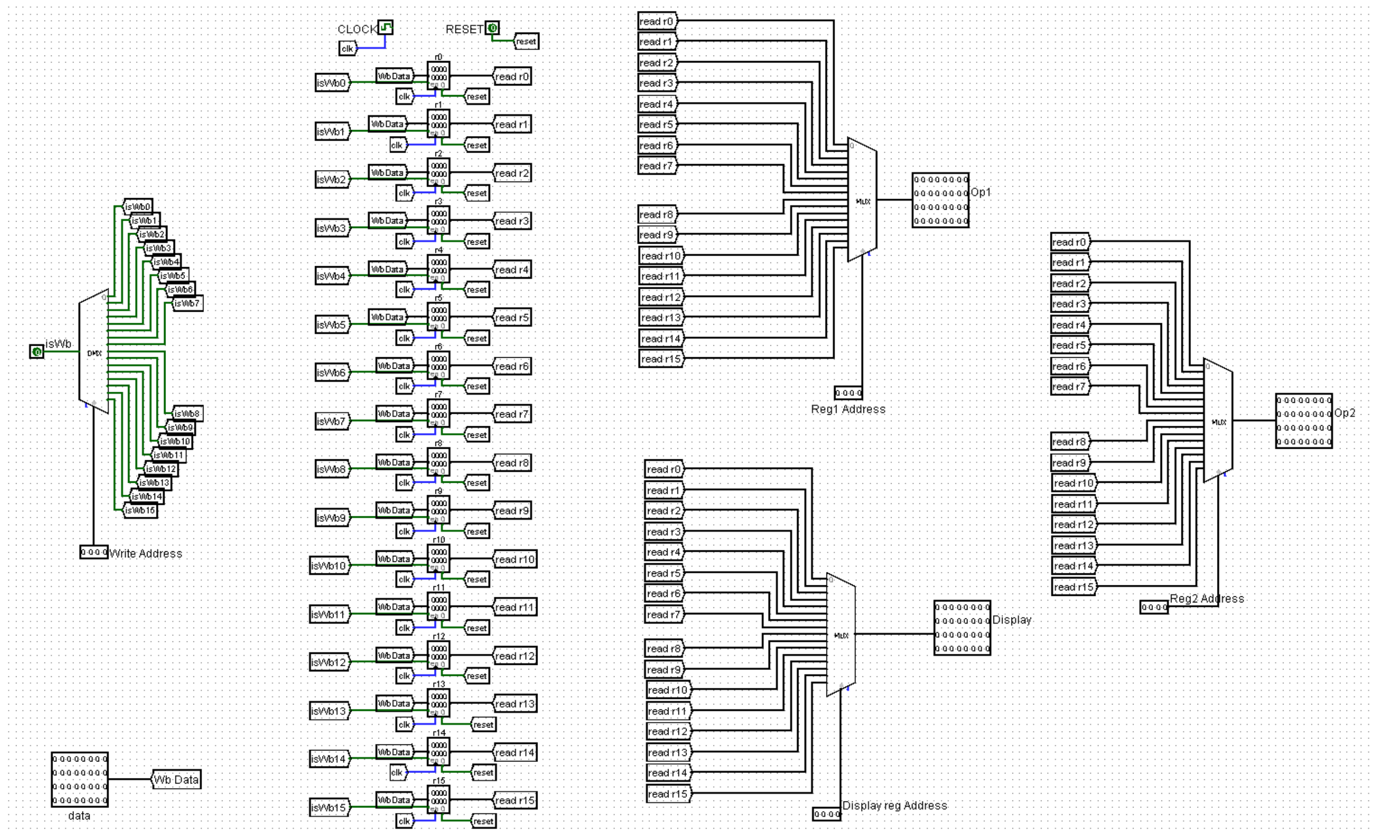
The value of flag registers can change only when *isCmp* is equal to 1 as it is connected as a clock input. The value of the flag registers will remain intact until the next compare statement.

Register Files

Register files are used to store data in the CPU for fast access during program execution as the registers use flip flops as their storage mechanism rather than slower memory components.

The register file consists of 16 registers out of which 14 are general-purpose registers that can be used during program execution to store values and the other two are stack pointer register and return address register.

The return address register stores the value of the instruction *pc+1* where *pc* points to the call instruction for a function call. For a function call stack, the *sp register* is used to store the stack top for the function call stack.



Reading data from the register file

At each clock cycle, the register file unit will look for the address of the register to be decoded. After getting the address of the register which has to be read the address is passed to the selection line of the multiplexers used. The input line of the multiplexers is connected to the output of all the register, based on the address provided in the selection line, the respective register's data will get decoded and will be stored in *Op1* and *Op2*.

Writing data to the register file

When a function call is made, the return address needs to be written to the register file, similarly, when a load operation is performed or ALU operation is performed data needs to be written back to the register file.

The signal *isWb* is used to check that whether the instruction needs to write a new data in the register. The address of the address is provided to the selection line of the *demultiplexer*.

Suppose, some new data needs to be written in the register r5 then in that case the selection line value will be **0101.** Hence corresponding to 0101 *isWb5* becomes 1 and the remaining outputs of the *demultiplexer* will remain 0.

The output of the *demultiplexer* is connected to the enable line of the register. The value of enable should be 1 if any new data needs to written in the flip flop at clock cycle.

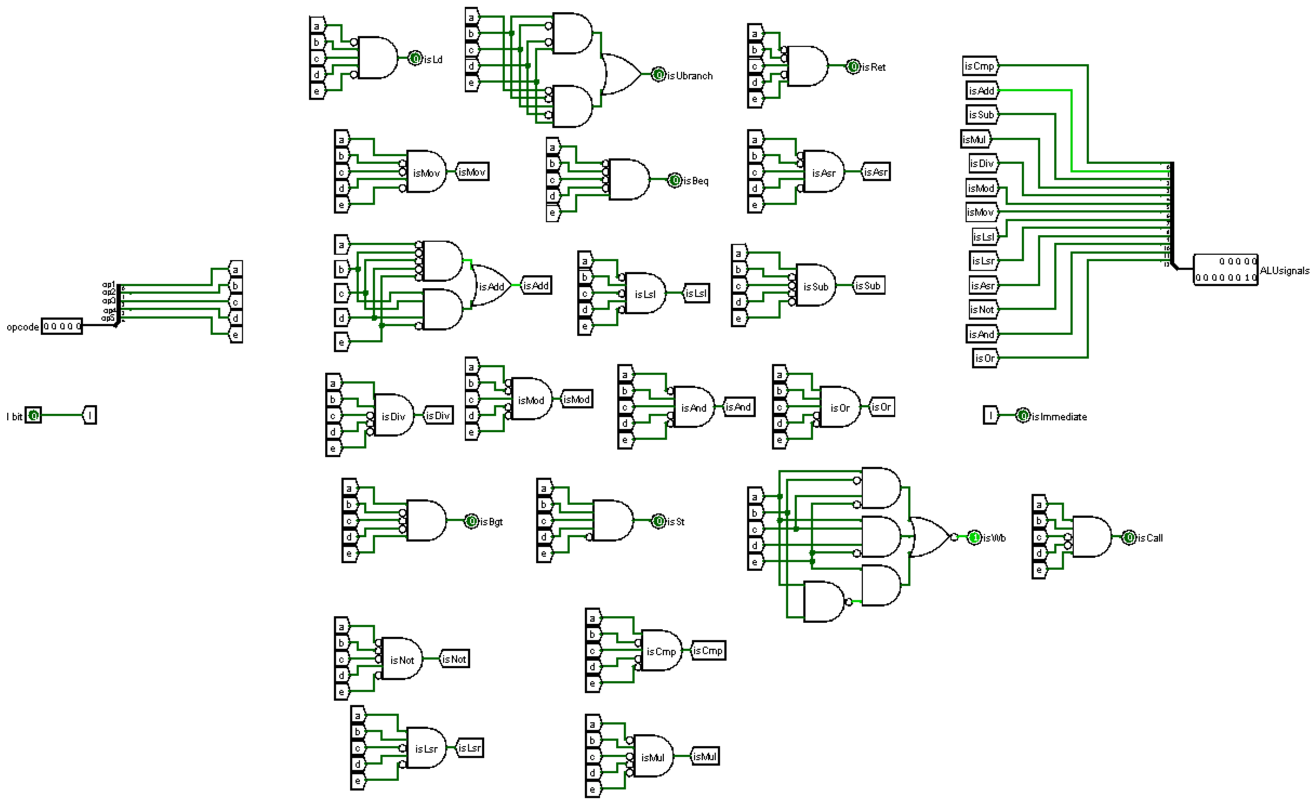
In the above example the enable of register r5 will become 1 hence the new data coming into the register file will get written in r5 during the next clock cycle.

The signal *isWb* is used to check if a register write needs to be performed. If *isWb* is 1 then the value in *Wb Data* is written into the register with address *Write Address*.

Display register value

The value stored is registers can also be displayed to monitor the values stored in a register during program execution in the same fashion in which it is read from the registers.

Control Circuitry



The control circuitry takes as input the 5-bit opcode of the current instruction and the immediate bit and generates control signals for the execution in the CPU.

Serial No.	Signal	Condition
1	*isSt*	Instruction: *st*
2	*isLd*	Instruction: *ld*
3	*isBeq*	Instruction: *beq*
4	*isBgt*	Instruction: *bgt*
5	*isRet*	Instruction: *ret*
6	*isImmediate*	*I* bit set to 1
7	*isWb*	Instructions: *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *not*, *mov*, *ld*, *lsl*, *lsr*, *asr*, *call*
8	*isUBranch*	Instructions: *b*, *call*, *ret*
9	*isCall*	Instructions: *call*
	aluSignals	
10	*isAdd*	Instructions: *add*, *ld*, *st*
11	*isSub*	Instruction: *sub*
12	*isCmp*	Instruction: *cmp*
13	*isMul*	Instruction: *mul*
14	*isDiv*	Instruction: *div*
15	*isMod*	Instruction: *mod*
16	*isLsl*	Instruction: *lsl*
17	*isLsr*	Instruction: *lsr*
18	*isAsr*	Instruction: *asr*
19	*isOr*	Instruction: *or*
20	*isAnd*	Instruction: *and*
21	*isNot*	Instruction: *not*
22	*isMov*	Instruction: *mov*

The control circuit generates 13 control signals for the ALU, *isAdd, isSub, isCmp, isMul, isDiv, isLsl, isAsr, isOr, isAnd, isNot and isMov* for performing the corresponding ALU operations addition, subtraction, comparing two numbers and setting the value of flag registers accordingly, multiplication, division, logical left and right shift, arithmetic right shift, logical and, logical not, logical or and mov operation for copying the value in one register to another register or copying the value of an immediate operand to a register.

Control signals required for memory access are *isLd, isSt* for performing load from memory and store to memory operations. The load from memory operation loads a value from the cache memory to the register and the store to memory stores a register value to cache data memory.

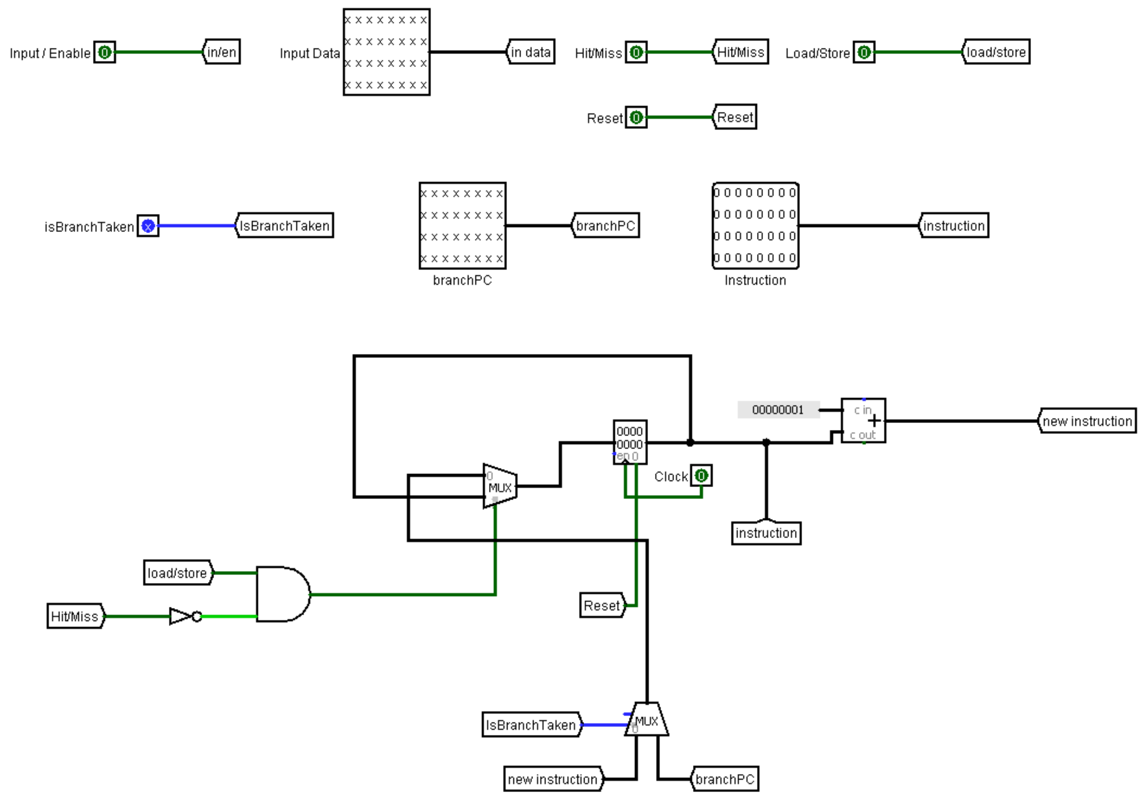
IsImmediate control signal is used to identify if an instruction contains an immediate value and the IsWb control signal is used to check if a register writeback needs to be done for an ALU operation or a call instruction where the return address needs to be stored in *ra* register.

Other signals generated include the branch signals, unconditional branch *isUBranch* in case the instruction is branch, call, return - *b, call or ret* and a conditional branch for branch if equal and branch if greater than signals *isBeq, isBgt.*

For a return instruction, *isRet* signal is generated as the return address value stored in ra register needs to be read unlike in other branch instructions where the offset value is present in the instruction itself.

Program Counter

Before an instruction is executed the instruction needs to be fetched from the instruction memory. The instruction to be executed is not necessarily the instruction after the current instruction. There can be branch instructions, function calls, or returns from a function which can lead to an instruction other than the instruction after the current instruction to be executed. This requires a program counter which stores the address of the next instruction in the instruction memory that needs to be executed.



If the current instruction is a load or store instruction then the tag array is checked if it contains the tag for the memory address for the instruction. If it is a hit then the load store operation is performed in the same cycle requiring 1 cycle for execution.

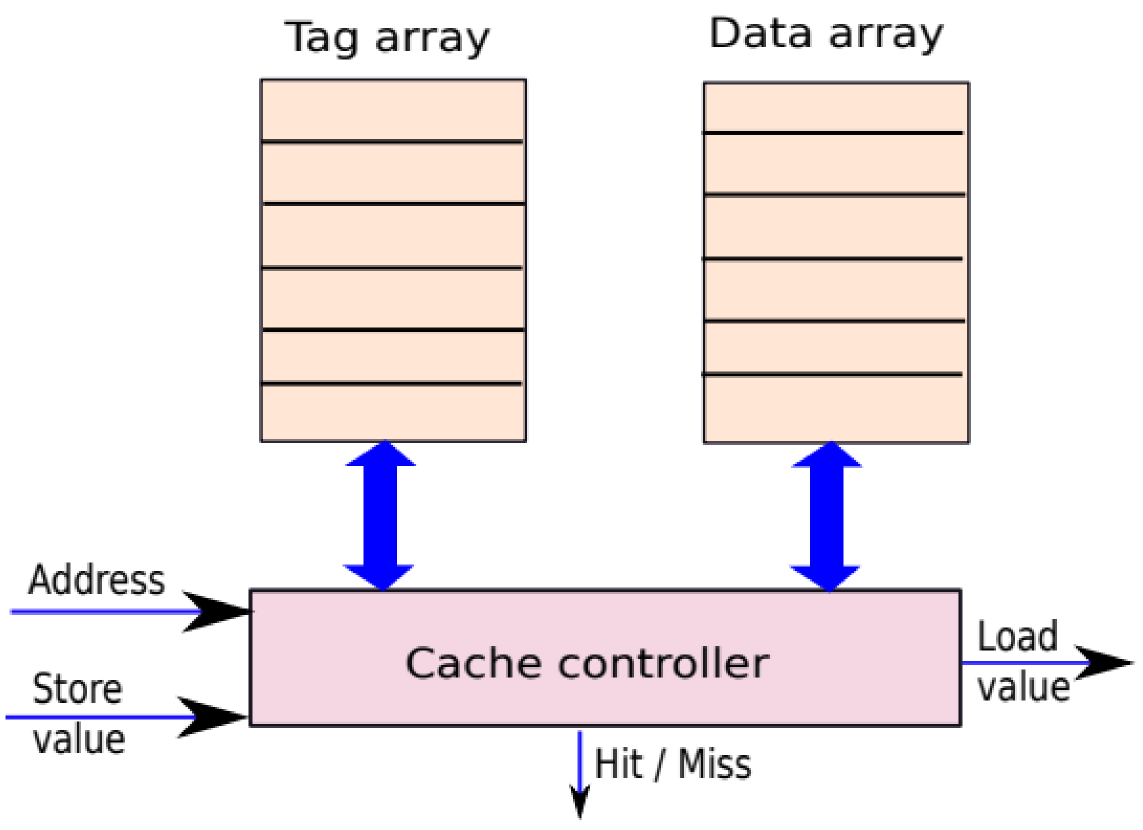
In case of a miss the block is checked if it is valid or not. If the block in data array is invalid, or if the block is valid but not modified then the data is fetched from the memory into the cache which takes additional 8 cycles. The next cycle is a hit and the next instruction is then fetched taking a total of 10 cycles for execution.

When the data in the block array is modified and there is a cache miss then the data in the block needs to be evicted which takes 8 cycles and then the required block needs to be fetched from the memory which takes another 8 cycles after which there is a cache hit which requires a total of 18 clock cycles.

If the current instruction is not a branch instruction or a load-store instruction, then the next program counter value is *pc+1* to fetch the next instruction from the memory.

If the current instruction is a branch instruction then the *branchPC* is calculated as (current instruction pc + branch offset) which is the next instruction to be fetched or the return address stored in the *ra* register for the *return* instruction.

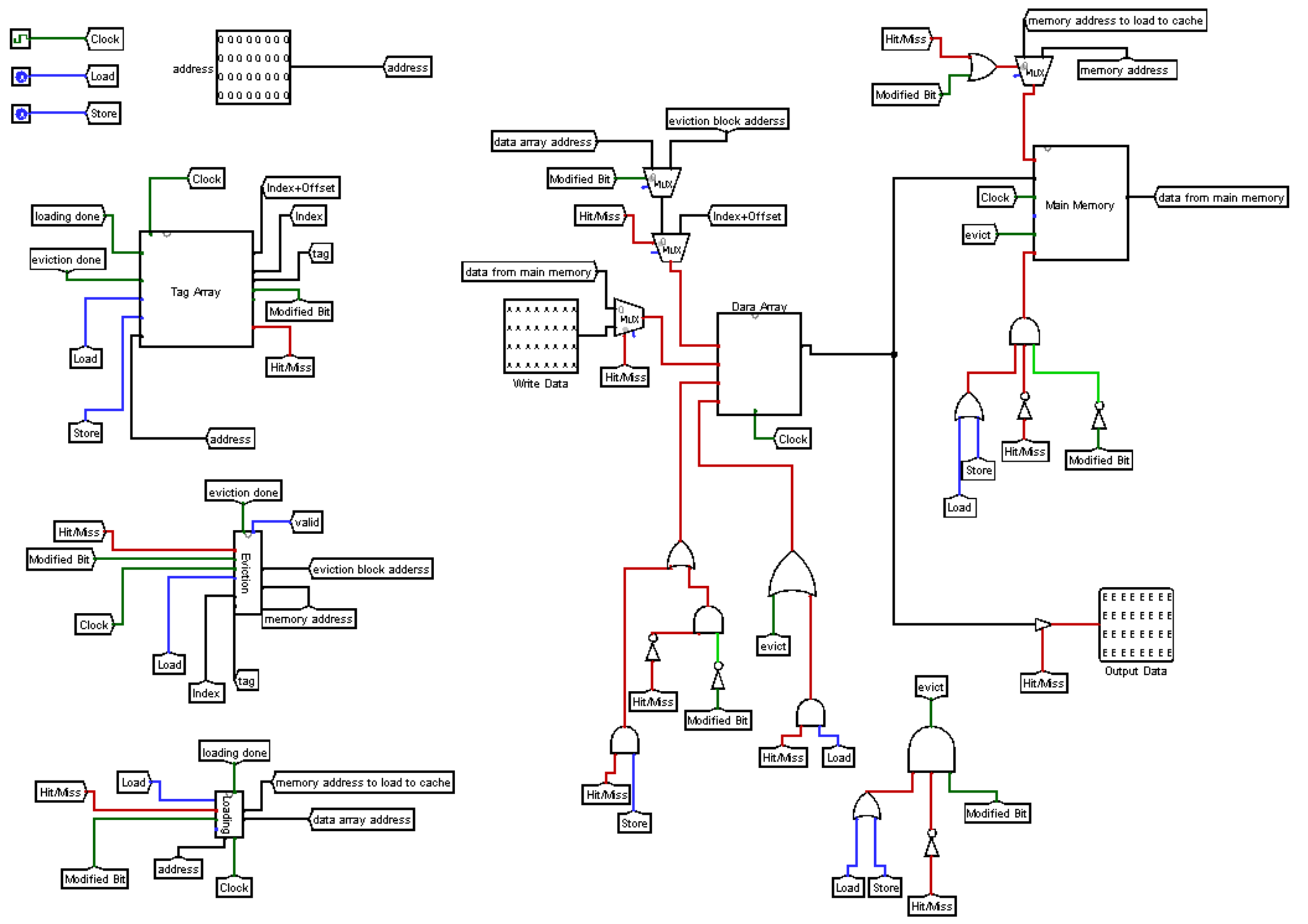
Cache Management



structure of a cache Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

At the heart of the project is its cache management system that provides an efficient alternative to frequent memory access. A cache is designed as a memory with faster access times than the main memory which stores the data as cache memory is SRAM based which has faster access time compared to a *dram* based memory offering a trade-off between speed and cost.

Cache exploits the patterns in which data is accessed from the memory like spatial and temporal locality in access by storing a block of contiguous memory locations from the main memory into the cache and by storing the frequently accessed memory locations respectively.



cache management module

The cache management system in our implementation has a modular implementation of the various parts in a cache management system for performing operations like loading the data from memory into the cache using the **loading module.** The **eviction module** performs eviction from the cache in case the data array contains a block from the memory which is valid and has been modified as the cache implemented is a writeback cache and the changes made to the cache block need to be reflected in the main memory as well. Apart from this there is the **tag array module**, and the **data array module** which performs operations on the tag array such as storing the tag of the block that is being stored or is replacing a block in the cache, and the data array which performs the loading and eviction for the part of the cache based on the signals generated in the cache management module.

Cache Memory implementation details

The cache memory we have implemented is a direct-mapped cache where each block in the memory can be mapped to one block in the cache memory. The cache uses a writeback policy which implies that the modified blocks are replaced after the changes made to the block are reflected in the main memory and the block is evicted.

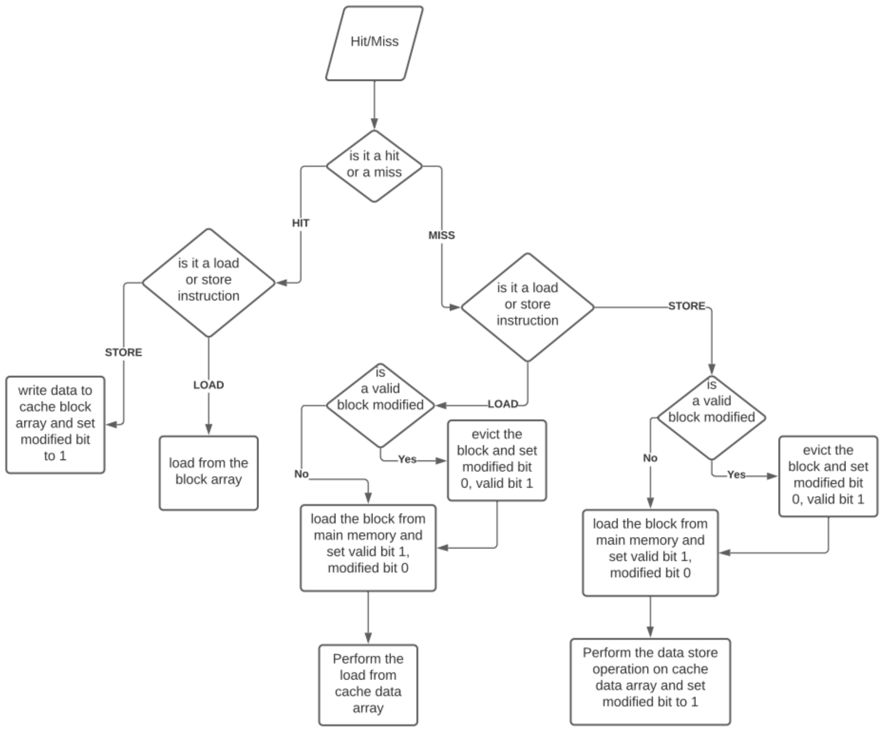
Each block in the cache stores 8 words which are 4 bytes each totalling to a block size of 32 bytes. This gives an offset of 3-bits to represent each word in a block.

The index of the cache is 10-bits which implies that the cache can store a total of $\mathbf{2^{10}}$ blocks of storage which totals 32kB of cache memory for the data array.

Similarly, the tag array contains tags which are 13-bits each, 11-bits for the tag, 1-bit to check if the data stored in the tag has been modified or not, and 1-bit to check if the block is valid (present in the cache) or not.

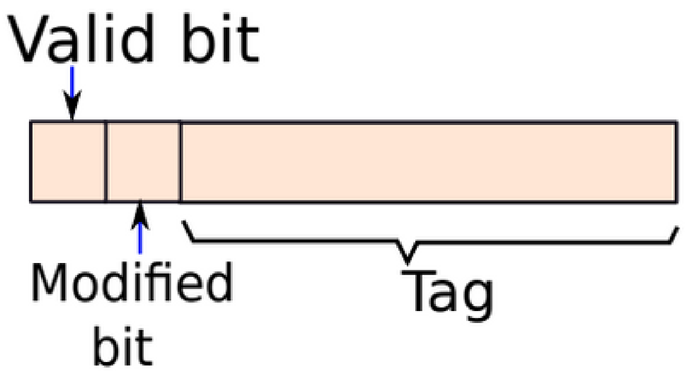
The main memory consists of a 24-bit address which gives a total of 2^{24} words, 2^{24} * 4 bytes or 64MB of main memory.

Flowchart for the working of cache



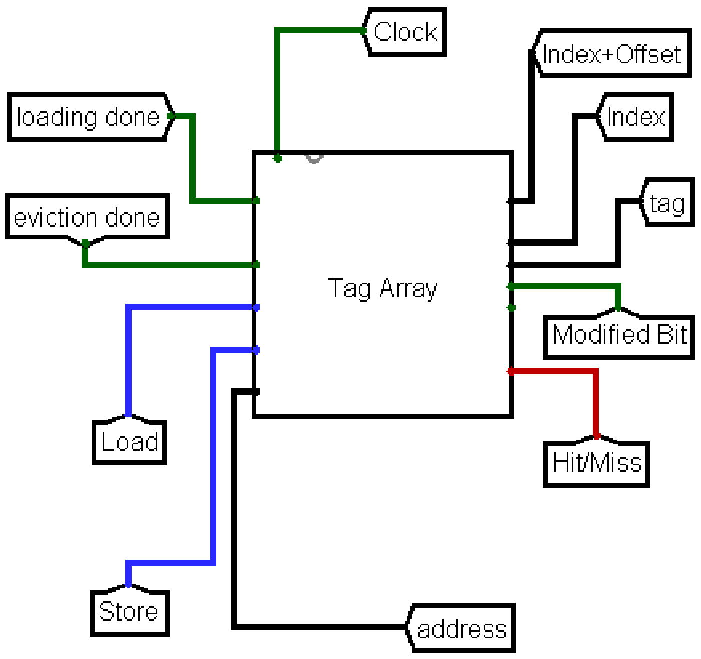
Hit/Miss
is it a hit or a miss
HIT
MISS
is it a load or store instruction
STORE
LOAD
write data to cache block array and set modified bit to 1
load from the block array
is it a load or store instruction
STORE
LOAD
is a valid block modified
No
Yes
evict the block and set modified bit 0, valid bit 1
load the block from main memory and set valid bit 1, modified bit 0
Perform the load from cache data array
is a valid block modified
No
Yes
evict the block and set modified bit 0, valid bit 1
load the block from main memory and set valid bit 1, modified bit 0
Perform the data store operation on cache data array and set modified bit to 1

Tag Array



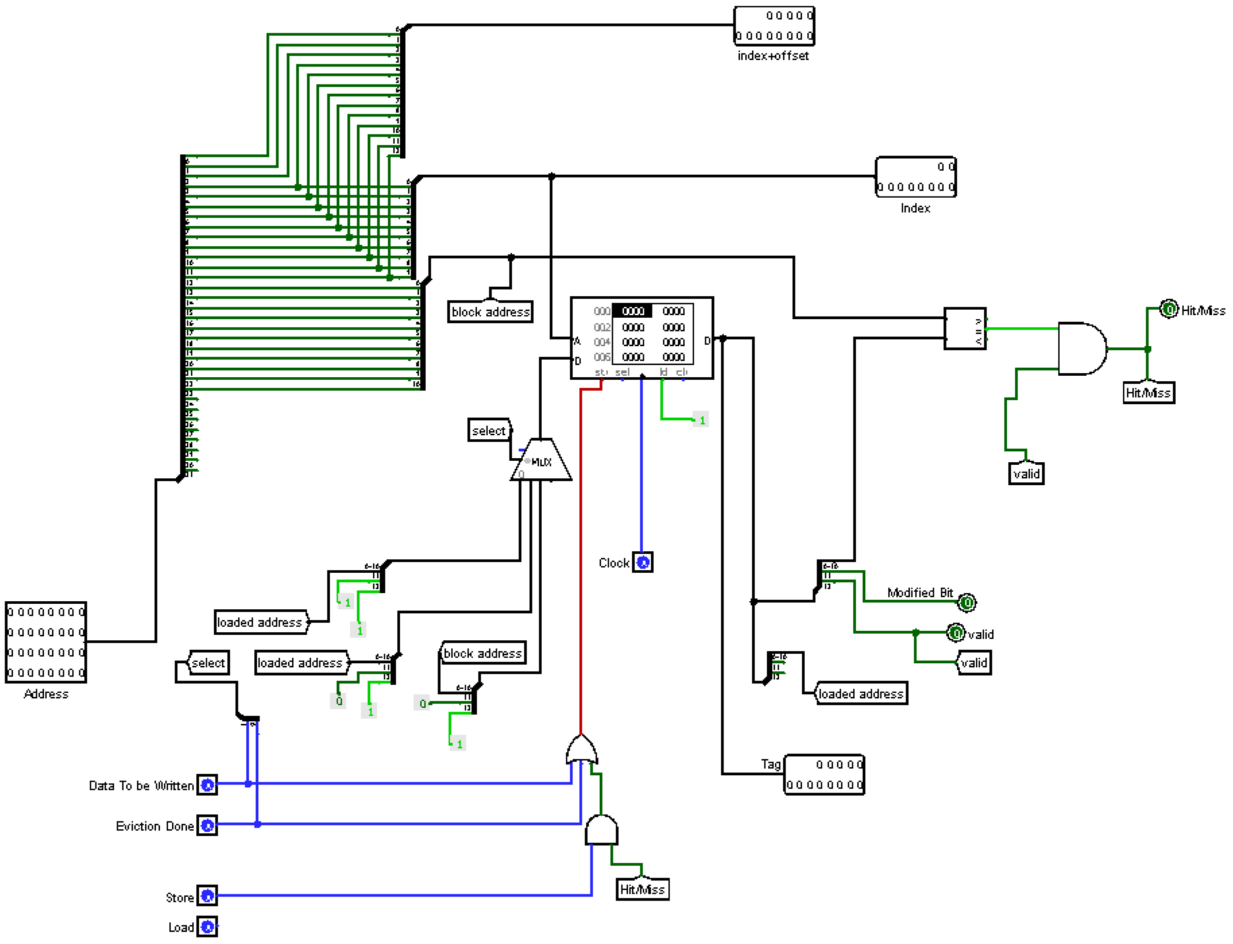
tag stored in the tag array
Source:https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html

Tags corresponding to each block stored in the data array of the cache are stored in the tag array. Each tag consists of a tag from the memory address, a modified bit and a valid bit.



The *tag array* takes the inputs:

- if the instruction is a load or a store instruction.
- if the loading is done which implies that the tag of the block (data) needs to be written to the tag array.
- if the eviction is done which means that the modified bit can be set to 0 and the loading from memory can be started.



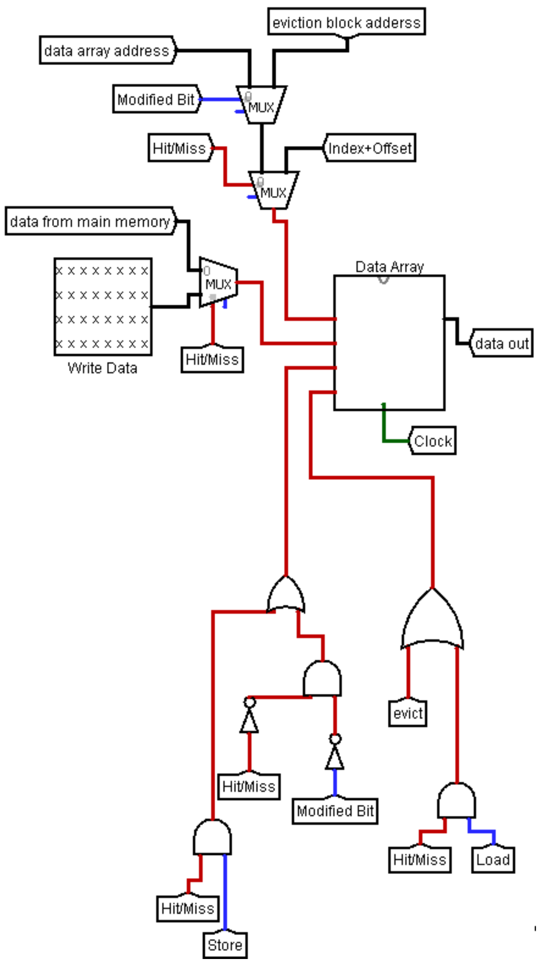
The tag array memory is checked at the required *index* given by 10 bits of the memory address (bits 3-12) if the tag contained at the given address in the tag array is a match or not. If the tag matches and is valid, then there is a cache hit. No eviction or load from main memory is required and the load-store operations can be performed in a single cycle.

If the tag in the array matches but the cache block is invalid then no eviction needs to be performed but the data needs to be loaded from the memory into the cache. When the loading is done, the *block address* is written to the tag array which is valid and is not modified. This leads to a hit in the next cycle and the load-store operation is performed.

Loading Done	Eviction Done	Valid bit	Modified bit	Memory data to be written
0	0	X	X	Loaded address
0	1	1	0	Loaded address
1	0	1	0	Block address
1	1	X	X	X

table for the values written to the tag array for the different stages of a load store instruction.

Data Array



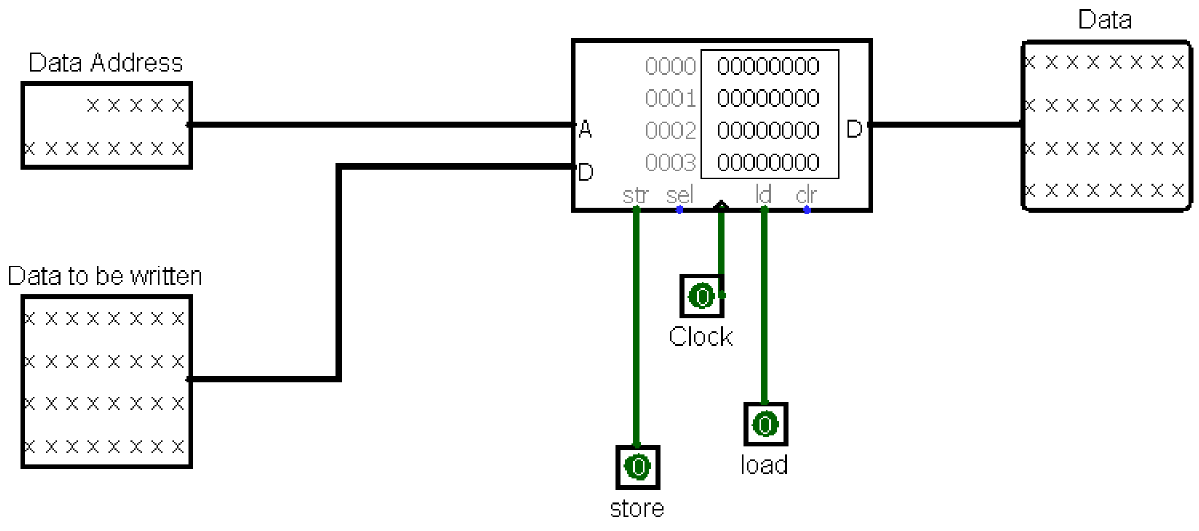
The data array is used to store the data cached from the main memory.

Cache hit

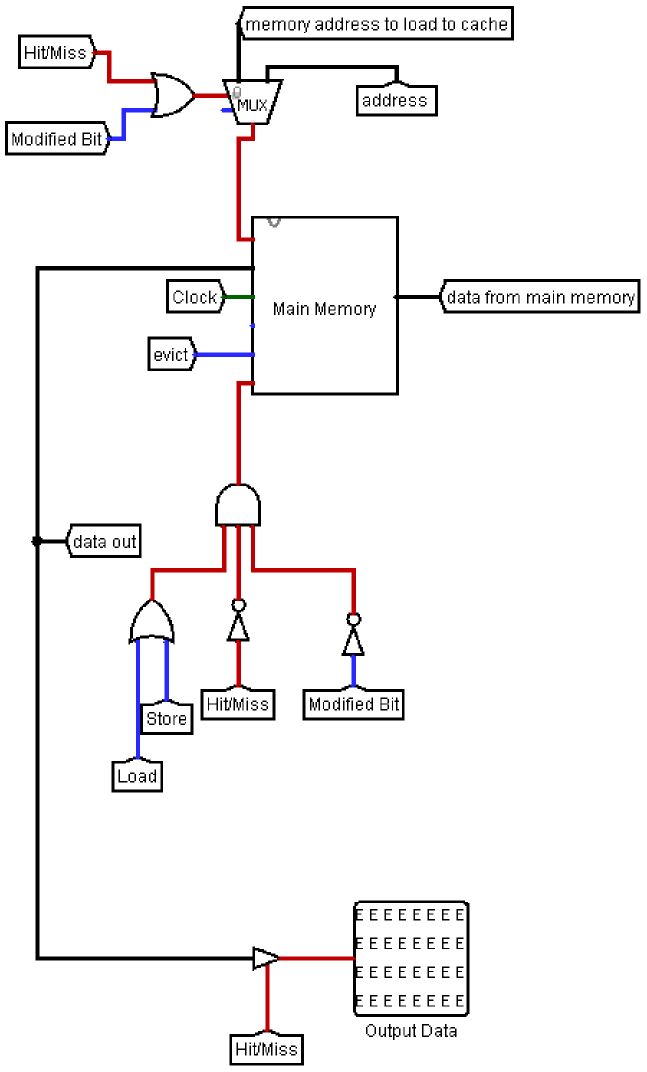
- If there is a cache hit, and the instruction is store, the data to be written can be added to the data array. The write port is activated and the *Write data* is written to the cache.
- If there is a cache hit and the instruction to be executed is a load then data needs to be read from the cache. The read port is activated and the value at *index+offset* is read from the cache.

Cache miss

- If there is a cache miss and the block in the cache is not modified, it can simply be replaced by loading *data from main memory* into the data array.
- If the block is modified then it needs to be evicted and then the block which needs to be accessed is fetched into the memory. For performing eviction, if the *evict* bit is 1 the data stored at *eviction block address* is read from the data array and stored into the main memory. The data loading process is then performed by loading the *data from main memory* into the cache.



Memory



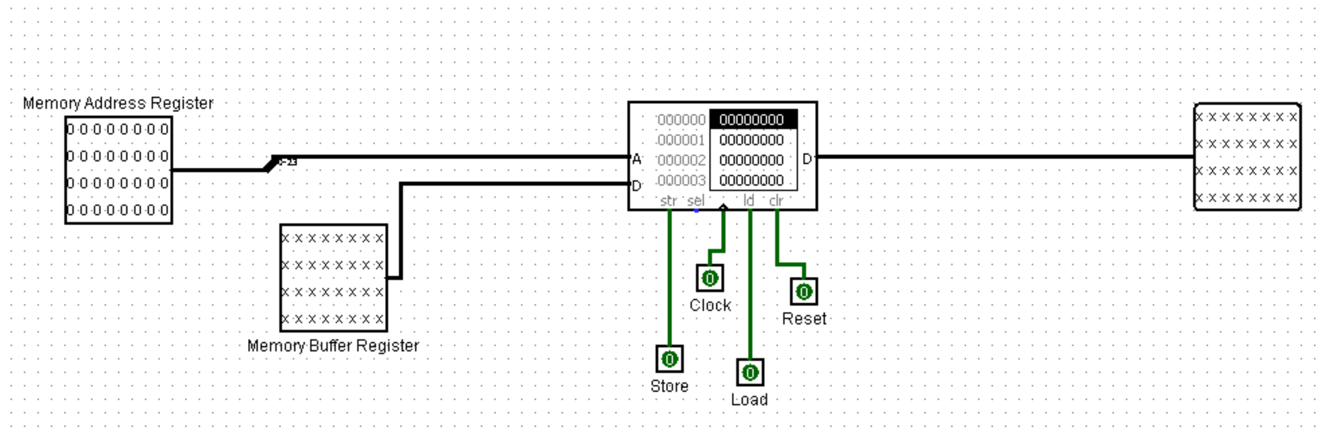
Cache hit

In the case where there is a cache hit the main memory does not require access but when there is a cache miss there is a penalty to be paid for it which is the miss penalty for the higher memory access times. This cache penalty totals to 8 cycles for our implementation where the data needs to be fetched from the memory to the data array of the cache or 16 cycles when the eviction needs to be performed as well.

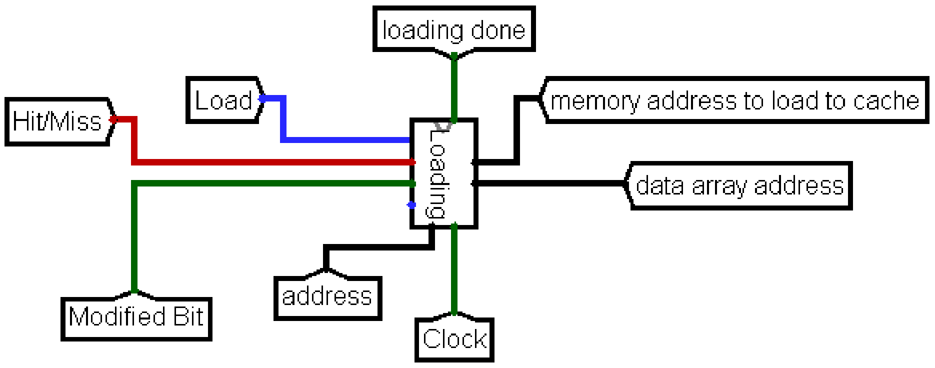
Cache miss

For a load store instruction when there is a cache miss if the modified bit is 0, then the *data from main memory* can be loaded to the cache for which the read port is activated. The data from the *memory address* calculated in the loading module is then stored in the data array of the cache at *data array address* calculated in the loading module.

Else if *evict* is 1 and the data needs to be evicted from the cache to the main memory for which the write port is activated and the data loaded from the cache corresponding to the *Data Array Address* calculated in the eviction module is stored in the *memory address* calculated in the eviction module. Load from memory is then performed for loading the block that needs to be accessed currently.

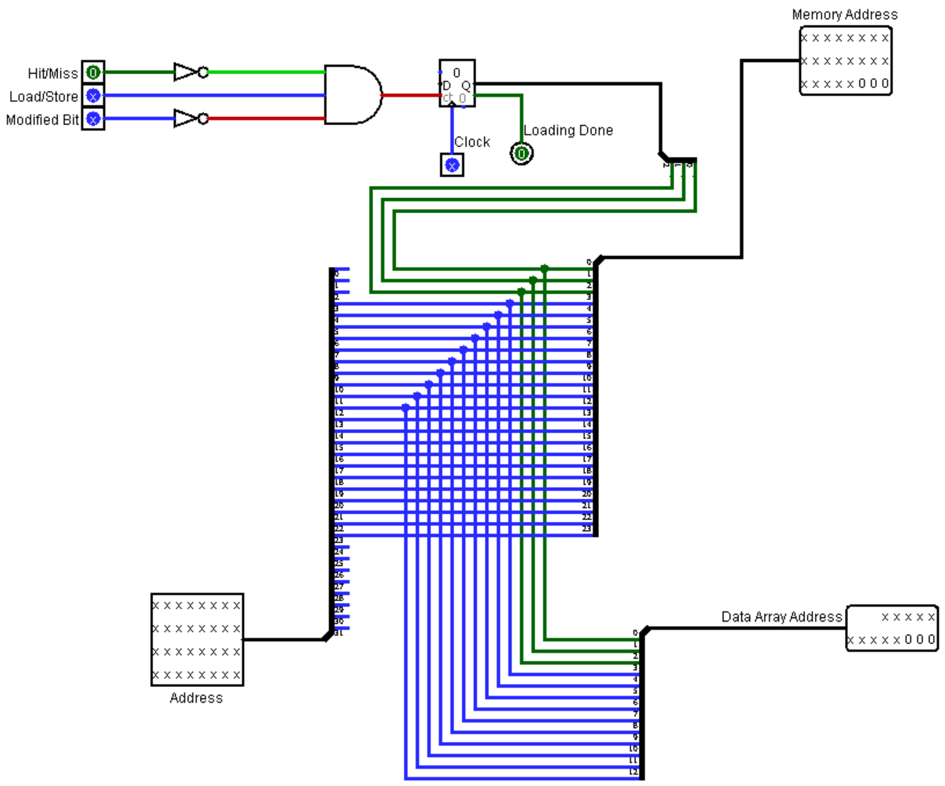


Loading from Memory

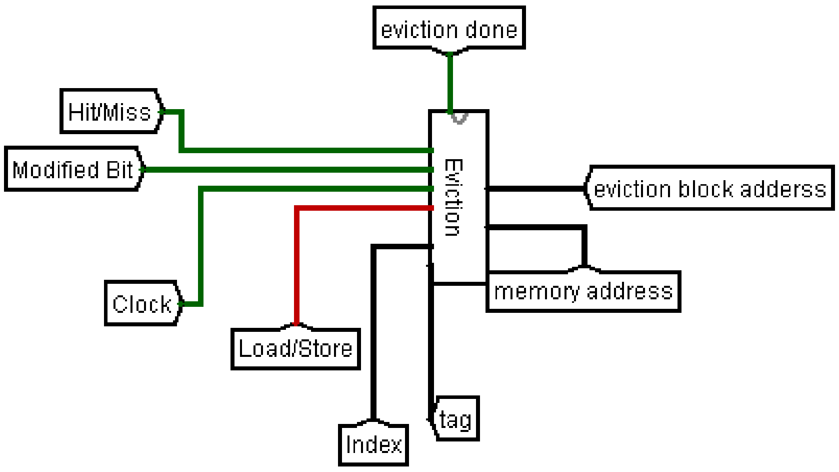


In case of a cache miss, the block of data needs to be loaded from the memory, one word per cycle for a total of 8 cycles. When the eviction is done for a modified block or the modified bit is identified as 0, the loading operation is performed for 8 cycles one cycle for each word loaded from memory.

The loading block uses a 3-bit counter to compute the memory address that needs to be loaded from the memory to the cache data array and the address in the data array where it is to be stored. After all the 8 words have been loaded, the module sets the loading done bit to 1 after which the data when accessed from the cache would lead to a hit.

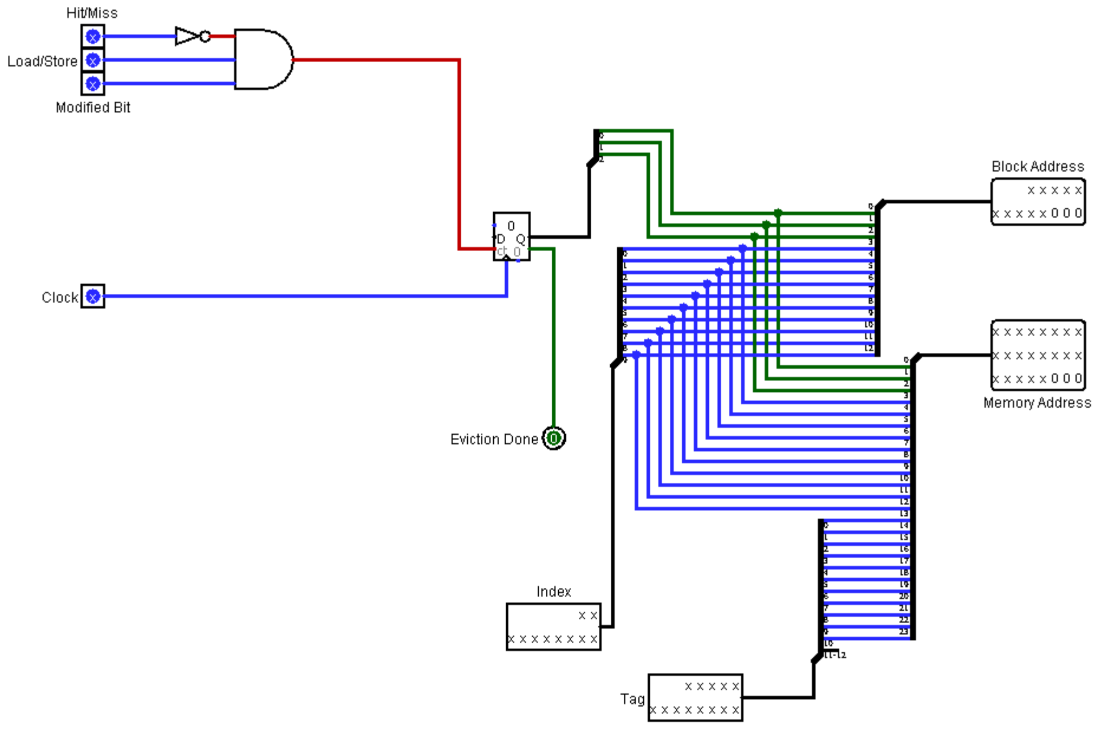


Eviction from Cache



In the case of a cache miss is the block is valid and has been modified then it needs to be evicted before the data is written to it as the cache, we have implemented is a writeback cache.

Eviction is done in a similar way as the loading was done, where the block address which is to be evicted and the memory address to which the evicted data is to be written is calculated using a 3-bit counter for all the 8 words in a block.



Evaluation parameters and Approach

We tested our CPU using few test cases which include some programs like calculating whether a number is prime, factorial of a number using loops and recursions. Also, each instruction of SimpleRISC were tested and verified. The main test cases of them are as shown:

Test 1

```
1. mov r0, 43
2. mov r1, 81
3. mov r2, 1
4. st r3, 3[r2]
5. ld r4, 2[r2] (This resulted in a HIT)
@ALU testing
6. or r5, r0, r1
7. and r10, r5, r4
8. lsl r5, r5, 2
9. mov r6, -7
10. asr r6, r6, 1
11. add r7, r6, r1
12. sub r7, r7, 2
13. add r7, r7, r5
14. mod r7, r7, r4
15. mul r7, r2, r2
@cache hit
16. st r7, 2[r2] (This resulted in a HIT)

@cache miss
17. ld r8, 8192[r2] (This resulted in a MISS)
```

In (4) the store operation was with a `MISS` in the cache. Here at the given set index there was a block (at least one out of four) with a valid bit as 0, so data is transfered from memory to data array. This operation took 8 clock cycles, as a block of data contains 8 data units and due to availability of only one read port, 8 cycles were required.

In (5) the load operation was with a `HIT` in the cache and the value at 2[r2] got loaded successfully to r4. This operation took 1 clock cycle.

In (16) the store operation was with a `HIT` in the cache and the value at r7 got stored in 2[r2] successfully. This operation took 1 clock cycle

In (17) the load operation was with a `MISS` in the cache and modified bit was `1` thus, it took eviction (8) +hit/miss(1)+loading(8)+hit/miss(1)=18 clock cycles.

Test 2

```
1. mov r0, 37
2. mov r1, 0
3. ld r2,r1[3]
4. ld r3,r1[2]
5. add r5, r1, r3
6. add r5, r3, 1
7. mul r5, r5, r4
8. div r5, r5, 3
9. st r5, r1[6]
10. mov r1, 1
11. ld r6, r1[8192]
```

In (3) the load operation was with a `MISS` in the cache. Here at the given set index, there was a block (at least one out of four) with a valid bit as 0, so data is transferred from memory to data array. This operation took 8 clock cycles, as a block of data contains 8 data units and due to availability of only one read port, 8 cycles were required.

In (4) the load operation was with a `HIT` in the cache and the value at 2[r1] got loaded successfully to r4. This operation took 1 clock cycle.

In (9) the store operation was with a `HIT` in the cache and the value at r5 got stored at 2[r1] successfully. This operation took 1 clock cycle.

In (11) the load operation was with a `MISS` in the cache and modified bit was `1` thus, it took eviction (8) +hit/miss (1) +loading (8) +hit/miss (1) =18 clock cycles.

Results

We were able to successfully run the stimulation and has tested our CPU Design implementation rigorously by using all the possible variations that might have caused the problem. All the glitches were successfully rectified through evaluation. Special care has been taken that no garbage value should be written in the cache and the registers.

Load and store operations are successfully executed using the cache design.

Cache Design has significantly reduced the memory access time by loading the suitable blocks of data from the main memory.

Conclusion

We have learned a lot from this project. Firstly, we looked for the videos and tutorials of Logisim to get familiar with the working of Logisim. After having decent information on Logisim we started with the designing part of our project.

At the end, when all the subcomponents were ready, we have integrated all the components which made the circuit functional and all the components got synchronized. We got an in-depth knowledge of the Cache architecture, we learned how to bring a block of data from the main memory to the cache, how to search for the data from cache and if found then how to evict it, secondly if it's a miss then we need to search for it in the main memory and then load that data from the main memory into the cache. If any changes are made in the data, then first change that data in the cache and after doing that modify the main memory.

This was truly a highly informative experience for all the team members. We had a wonderful experience in coordinating the work done by each other and finally getting the final task done.

References

1. Computer Architecture and Organization: Smruti Ranjan Sarangi https://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html
2. Computer Organization and Architecture: William Stallings
3. Logisim Docs: http://www.cburch.com/logisim/docs/2.7/en/html/guide/tutorial/index.html
4. Logisim Tutorial videos explaining the Docs: https://www.youtube.com/playlist?list=PL9Tu_yD7oJURQqPEAQ78FggiDeiK7MqVb