CSN 221: Computer Architecture and Microprocessors

Project Report : **24-Bit CPU Design**

Dhruv Shandilya (20114034)
Pinnapu Reddy Harshavardhan Reddy (20114070)
Shaurya Rana (20114086)
Shaurya Semwal (20119048)
Uday Jangir (20114102)
Ujjwal (20114103)

Content

The Problem Statement

To design and implement a 24-bit processor which efficiently operates in 5 stages: Instruction Fetch, Decode, Execute, Memory Read and Memory Write and performs the memory management techniques using cache management methodologies.

Novelty of the Work Done

24-Bit Processor

Our processor operates on 24-bit instructions and this very fact makes our project uniques as not many processors that have been designed use 24-bit instructions.

Cache Management

We designed a working procedural implementation of a direct mapped cache for our processor. This takes the control for the load and store instructions and performs the operations in order to make the processor efficient in executing the instructions.
Our cache is designed in such a way that itreduces the time lag because of repeatedly fetching the data from the main memory. The implementation of cache enables us to use the temporal and spatial locality which are in the core of any speed-up procedures. To start with the procedure of the cache implementation, we understood the necessity of implementing the cache, which gave us an insight into efficient memory management techniques

Methodology

The Instruction Word

The size of the instruction word has been taken as 24 bits and the instructions are divided into a total of 4 types:

- Branch :

23-19	18-0
opcode	immediate

- Register :

23-19	18	17-15	14-12	11-0
opcode	imm	rd	rd1	rs2

- Immediate :

23-19	18	17-15	14-12	11-0
opcode	imm	rd	rd1	immeidiate

- Load/Store :

23-19	18	17-15	14-0
opcode	imm	rd	Address

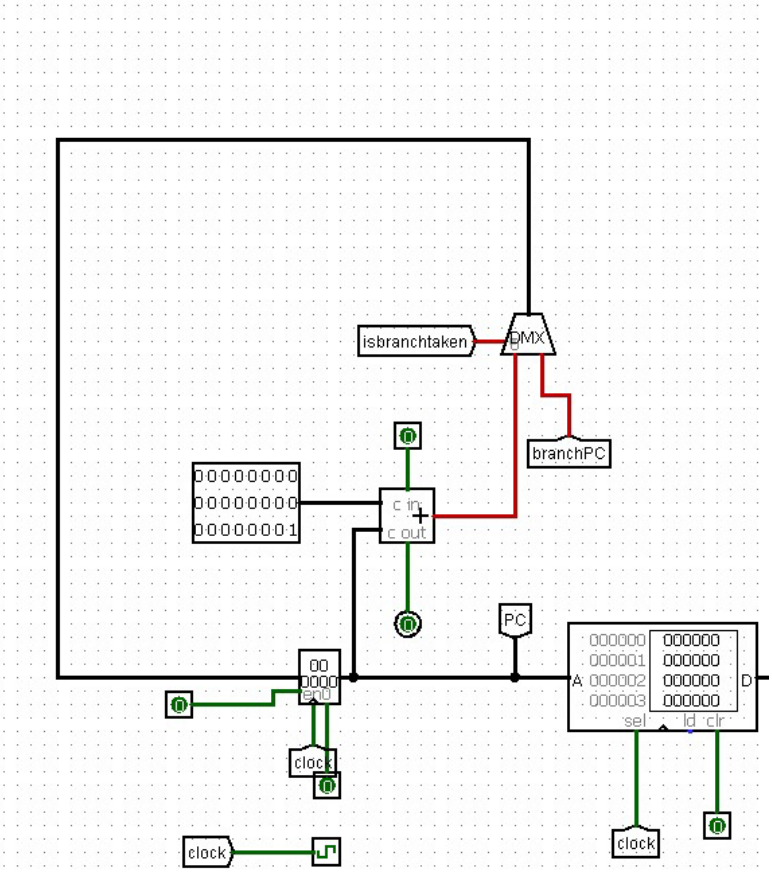
Opcode

In this implementation of a processor, 5 bit opcode is used. A total of 18 operations can be performed by the processor. These are :

Instruction	Code	Format
add	00000	Add rd,rs1,(rs2/imm)
sub	00001	Sub rd,rs1,(rs2/imm)
cmp	00101	Cmp rs1,(rs2/imm)
and	00110	And rd,rs1,(rs2/imm)
or	00111	Or rd,rs1,(rs2/imm)
xor	01000	Xor rd,rs1,(rs2/imm)
mov	01001	Mov rd,(rs2/imm)
lsl	01010	Lsl rd,rs1,(rs2/imm)
lsr	01011	Lsr rd,rs1,(rs2/imm)
asr	01100	Asr rd,rs1,(rs2/imm)
nop	01101	nop
ld	01110	Ld rd,[addr]
st	01111	St rd,[addr]
beq	10000	Beq offset
bgt	10001	Bgt offset
b	10010	B offset
call	10011	Call offset
Ret	10100	ret

Now, we will look at the 5 stages of the operation of the processor one by one.

Instruction Fetch



The first stage of the processor is the instruction fetch stage in which two steps are taken: Firstly, the instruction is fetched from the instruction memory (the RAM) and secondly, the address of the next instruction to be processed is found.

The program counter (pc) is the register which is used for knowing which instruction to load into the instruction memory

before its execution. It serves as an input address to the instruction memory where the instruction at the given address is fetched from the instruction memory. This is then given as an input to the next stage of the processor.

The implementation of the program counter is done in a way that if the current instruction is a branch instruction (conditional or unconditional) then the program counter gets fed by the corresponding branch address which is calculated by adding the offset and the program counter address. In the other cases, the program counter just gets incremented to the next address.

A demultiplexer (shown as DMX in the figure) is used to choose between **branchPC** and the current **pc** (address of current instruction) +1.
For this purpose, **isbranchtaken** is used as control signal for the demultiplexer. This is generated in the execute stage of the processor. In case **isbranchtaken** is equal to 0, current **pc** + 1 is taken and in case it is 0, the **branchPC** is taken for storing the address of the next instruction.

Instruction Decode

The instruction word that gets fetched is 24 bits in length. Depending on the type of operation, the instruction can be one of the four forms: register, branch, immediate and load/store.
The first 5 bits of the instruction [23-19] are for storing the opcode of the instruction and hence determine the type of operation to be performed. These are also used to generate the control signals for the processor.

The next bit, i.e, the 18th bit is used to determine whether the instruction contains an immediate or not.
The rest of the 18 bits are decoded in the operand fetch phase based on the type of instruction. For this purpose, two multiplexers are used. The first multiplexer is used to select between address of return register and **rs1**, with the selection decision taking place with the help of the control signal **isRet.**
Similarly, the second multiplexer selects between rd and rs1 with the selection taking place with the help of the control signal **isSt**.
Then, the addresses of both the registers which pass from multiplexers are taken as an input in the register bank where the corresponding address is used to select the register in the register bank . The output we get from the register bank is the values from the corresponding registers.

In this stage, the decoding of the immediate also takes place. In this, the 12 bits are used to have a 24-bit immediate in the following manner :
Input is 12 bits immediate value and output is 24 bits calculated immediate.
The instruction contains 12 bits (11-0) of immediate value in the format of 2+10 where 2 bits (11-10) are modifier bits and remaining 10 bits (9-0) represent the actual value.
The Modifier bits are defined as:

- 00 - Default (Signed)
- 01 - Unsigned
- 10 - High

When modifier bits are 00, the output has extended sign ,i.e. the 14 bits (23-11) are extended, same as most significant bit of the immediate.
When modifier bits are 01, output have most significant 14 bits (23-11) as 0 and remaining are same as immediate.
For modifier bits as 10, it shifts the 10 bits of immediate value to the most significant 10 bits (23-15) and remaining 14 bits are set to 0. The immediate calculator uses a multiplexer with 2 modifier bits as select bits and outputs the extended 24 bits immediate value.
In case of branchPC, the 19 bit address is scaled to 24 bits by first shifting it by 2 bits and later the extending sign.

The control signal is important for the general functioning of the processor as it helps to identify which instruction's opcode is executing and hence, which decisions need to be taken and which do not. For implementing the control signals of the 24-bit processor the usage of logic gates and opcode is taken where 19 control signals are produced corresponding to either an instruction or immediate value check. These control signals are then used in the processor later to determine which operation is to be performed.

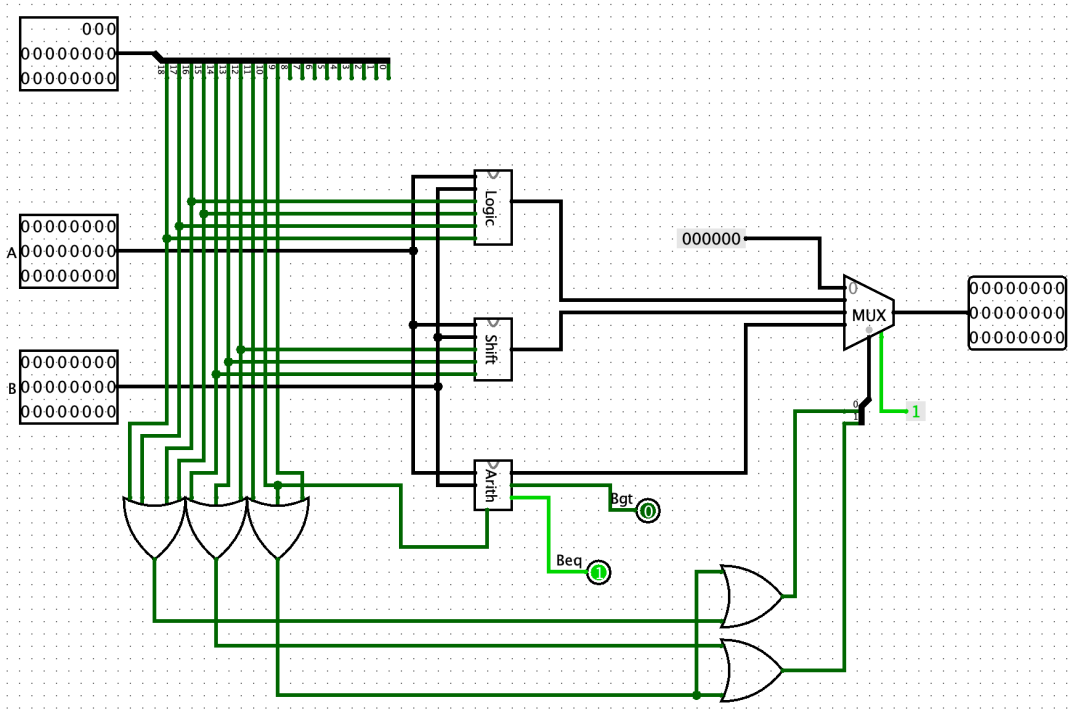
Execution

The output from previous stages is used in this stage. Here two things are done. First, the branch condition is checked. For this, the flags and control signals are used. Also, sending the branchPC occurs, which is selected by means of a multiplexer with the decision being taken between branchTarget and **op1**.This decision is taken using the control signal **isRet**.

The implementation of the ALU is also carried in this stage of the processor.

Arithmetic Logic Unit (ALU)

The ALU is the digital logic circuit that is used to perform mathematical operations on the data.



The ALU contains 3 inputs, two of which are the operands input and the third is a set of 19 control signals which is used to indicate which operation is to be performed.

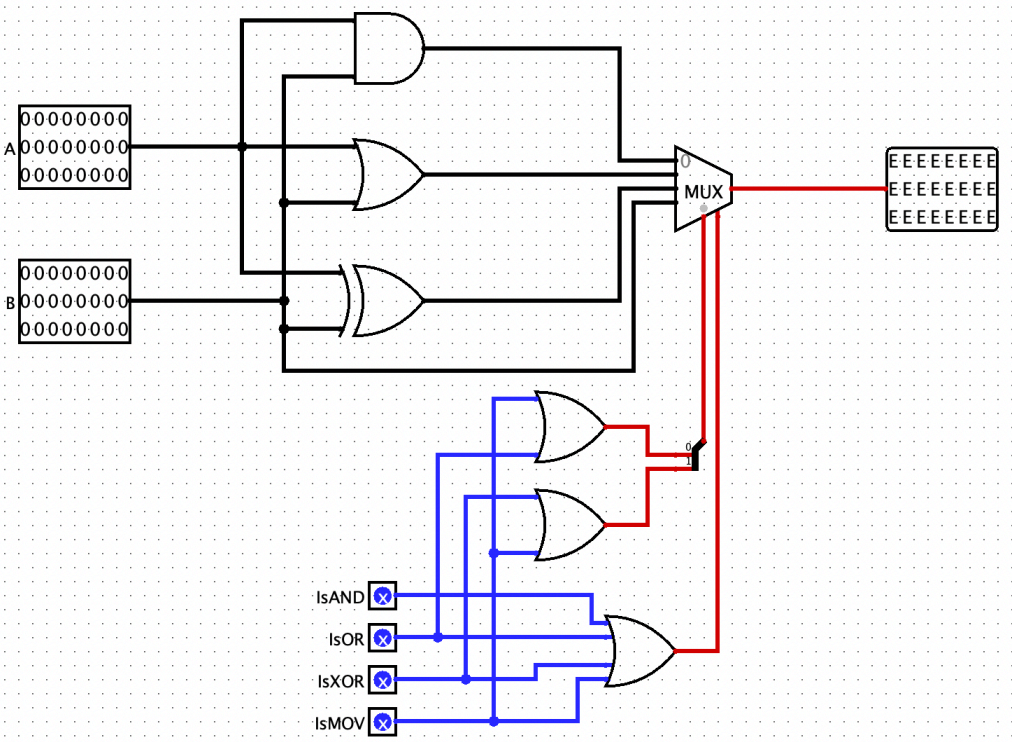
There are 3 outputs from the ALU. One is result of the mathematical operations performed on the operands, and the other two values (beq and bgt) are used to update the **Beq** and **Bgt** flags.

The ALU in divided into three parts

1. Logical Unit (**Logic**)
2. Shift Unit (**Shift**)
3. Arithmetic Unit (**Arith**)

The output of ALU is selected using a multiplexer. The inputs to the multiplexer are the outputs of the Logical Unit, Shift Unit and the Arithmetic Unit and the select input is the logical combination of the control signals. When the operation is not ALU (ex-branch) then any output can be given. The outputs of the three units of the ALU are calculated simultaneously.

Logical Unit

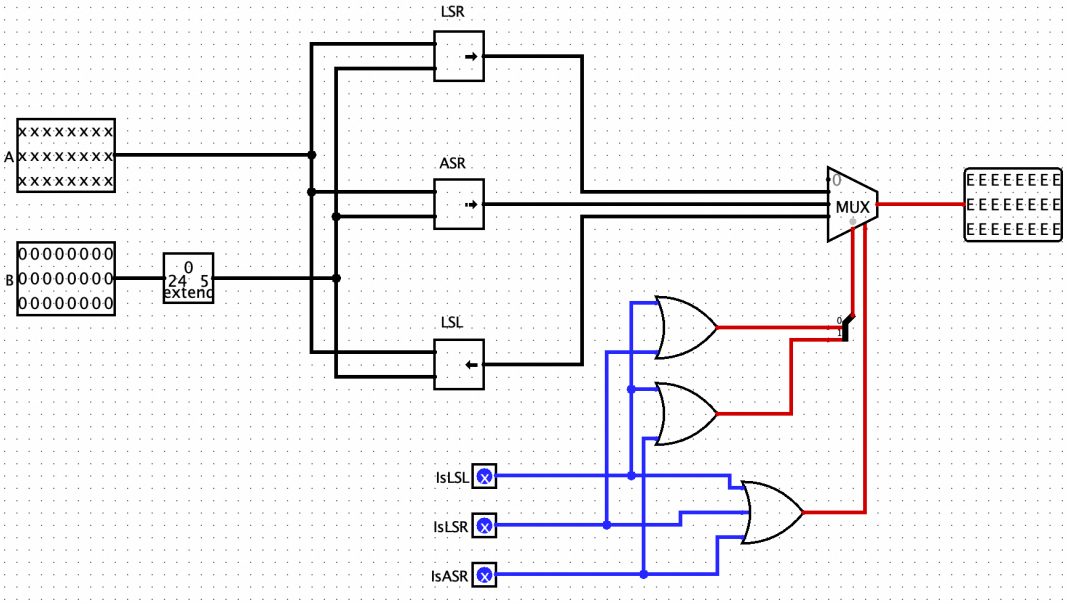


The logical unit performs 4 operations.

1. and : Performs bitwise logical AND of the two operands.
2. or : Performs bitwise logical OR of the two operands
3. xor : Performs bitwise logical XOR of the two operands
4. mov : Gives the second operand as the output

The outputs of the operations of the logical unit are calculated simultaneously. Then, the output is selected by using a multiplexer, with the result of these operations as inputs to the multiplexer and the control signal for the operations being select input.

Shift Unit

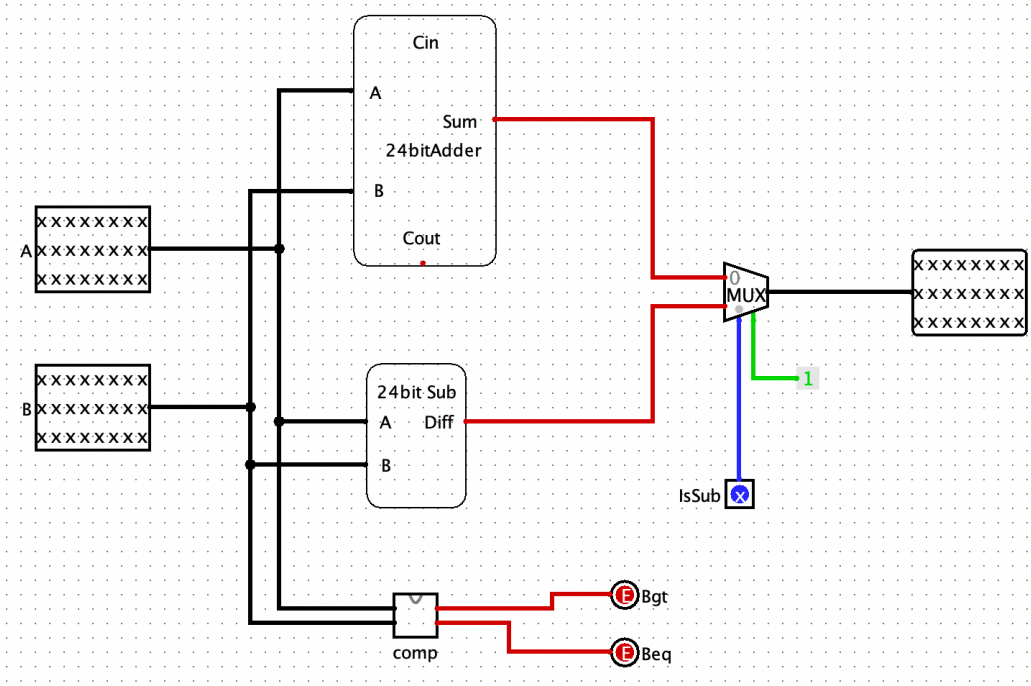


The Shift unit performs 3 operations.

1. LSL – First operand is shifted to the left by amount equal to the second operand
2. ASR – First operand is shifted to the right by amount equal to the second operand along with sign extension in the most significant bits. This operation is for signed numbers
3. LSR – First operand is shifted to the right by amount equal to the second operand and most significant bits are filled with 0's. This operation assumes the number to be unsigned

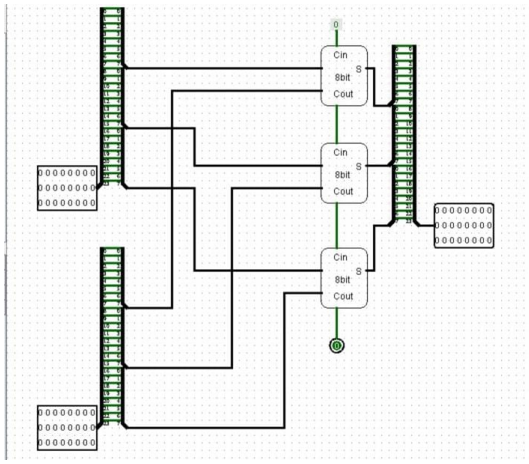
The outputs of the operations are calculated simultaneously. Then the output of the Shift Unit is selected by means of a multiplexer with results of these operations as inputs and the control signal for the operations as select input.

Arithmetic Unit



The Arithmetic Unit was made to perform three operations :

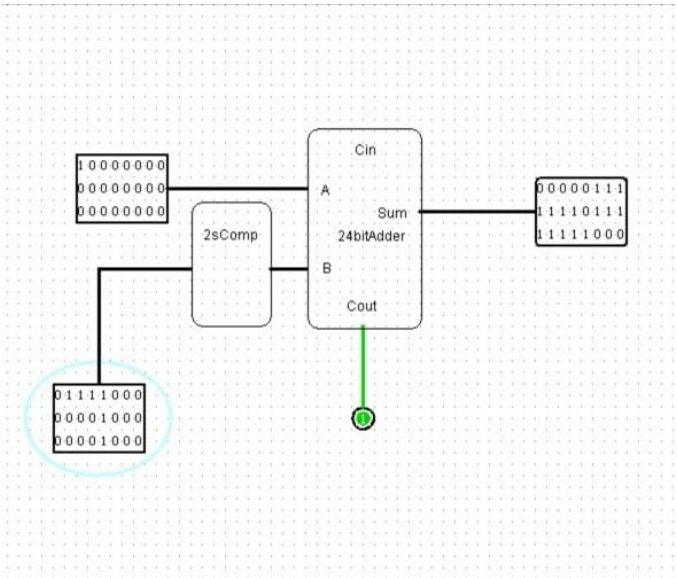
1. 24-Bit Adder :



First, a 1-bit adder was implemented with carry input, and carry out and sum as output. Then by daisy chaining in a serial

manner two 1-bit adders, a 2-bit full adder was made. In a similar process, a 4-bit adder was made using two 2-bit adders, the and 8-bit adder was made using two 4-bit adders. Three of these 8-bit adders were serially connected to obtain the required 24-bit adder. Using this 24-bit adder, the final output is obtained as a 24-bit sum and a final carry bit.

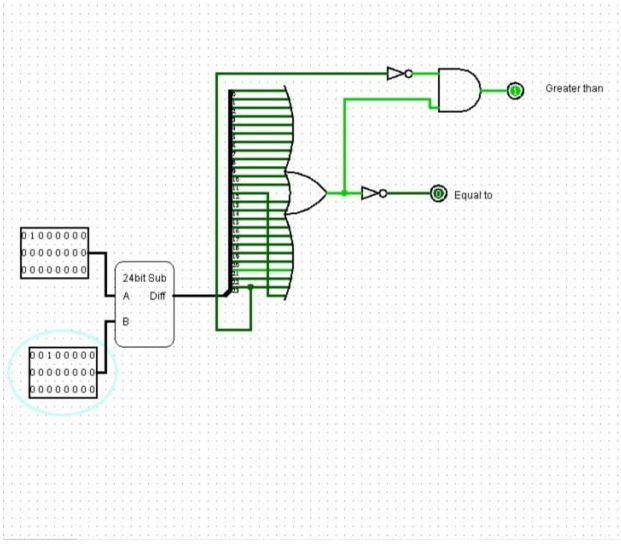
2. 24-Bit Subtractor :



The 24-bit subtractor was implemented using a 2s Complement circuit and a 24-bit adder. To make a 2s compliment circuit a circuit was designed that takes an 8 bit input and complements all the bits after splitting them. Then three of these are used to get a circuit that finds 1s compliment. Finally, to this output, 0x000001 (Hexadecimal) was added through our previously designed 24-bit adder to obtain the corresponding 2s compliment. Then, by adding the first number and the 2s complement of the second number

using the 24-bit adder, the subtractor was implemented.

3. 24-Bit Comparator :



To implement this circuit the 24-bit subtractor circuit was utilised. For this, the two 24-bit numbers to be compared are passed through our 24 bit subtractor. Two flags are taken as output in this circuit :

Equal to flag : This flag becomes active when the “OR” of all the bits of the subtractor result is 0. this essentially implies that all the bits are zero, meaning that the difference is zero, hence the two numbers are equal.

Greater than flag : This flag becomes active when the “OR” of all the bits is 1 and the most significant bit of the difference is 0. “OR” value equal to 1 implies that difference isn't zero and most significant bit being equal to 0 implies positive difference, hence meaning a greater than condition.

The outputs of the operations are calculated simultaneously. Then we select the output of the Arithmetic Unit through a multiplexer with the result of these operations as inputs and control signal for these operations acting as the select input.

Memory Read and Write

In this stage, both the reading of data from the RAM and then the writing of the output data into the registers is performed.

Memory Read

Cache Architecture

The instructions are stored in Instruction Memory and the data of the corresponding instructions in **ma** (main memory). This memory management scheme has a cache which stores the recently and frequently used data as per the application of temporal and spatial locality techniques. For implementation direct mapped cache was chosen, because of its simplicity.

Cache Design

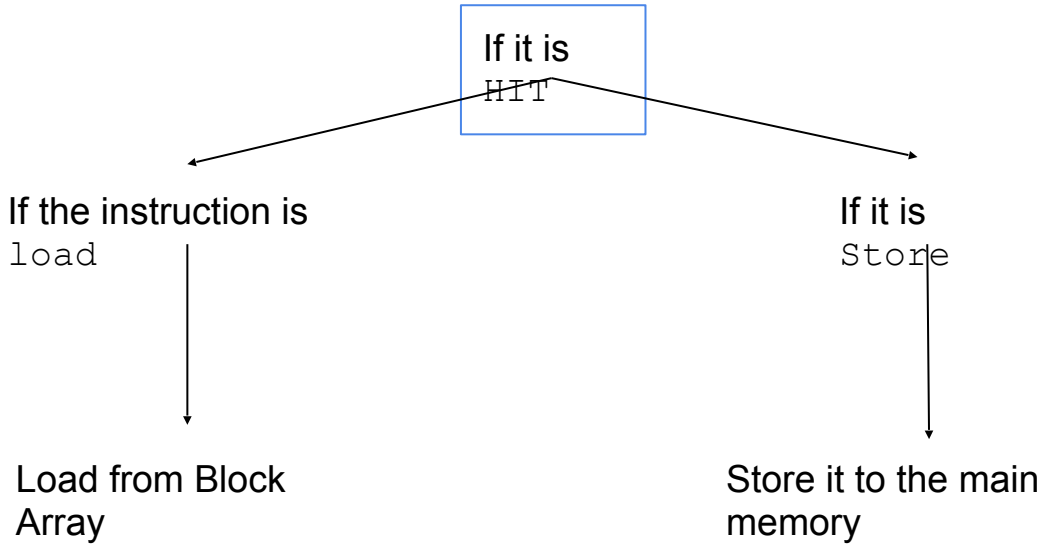
After the decision of which cache technique to be chosen, the cache capacity was decided. The ISA in this project supports the address of main memory as 15 bits, which means 2^15 units of data. For this, a block size of 32 was used for which the last 5 bits of data are considered as representing the block address.
After deciding the block size, it was time to decide the cache size. For this, a 16 entry direct mapping cache was chosen and to represent each of the 16 entries 4 data bits were used (bits [6-9]), the remaining 6 bits were used as Tag.

Tag(0-5)	Index(6-9)	Index(6-9)

Cache Functions

Cache acts as a midway data storage between the GPRs of the CPU and the main memory unit for the sake of fast data transfer. Whenever the execution of the requires writing or reading of the data from the memory, first it was checked if the data is present in the cache. If the data is found in the cache then it is said to be cache **hit** else, a cache **miss**.

Checking whether it is a hit or miss is done by comparing with each of the tags those are already stored in the Tag Array. If any of the already stored tags matches with the tag corresponding to the current address then we get a **hit**.



If it is not a hit, then the data should be fetched first from the main memory and stored in the block array and should be sent as the data output.

Necessity of Replacement Techniques?

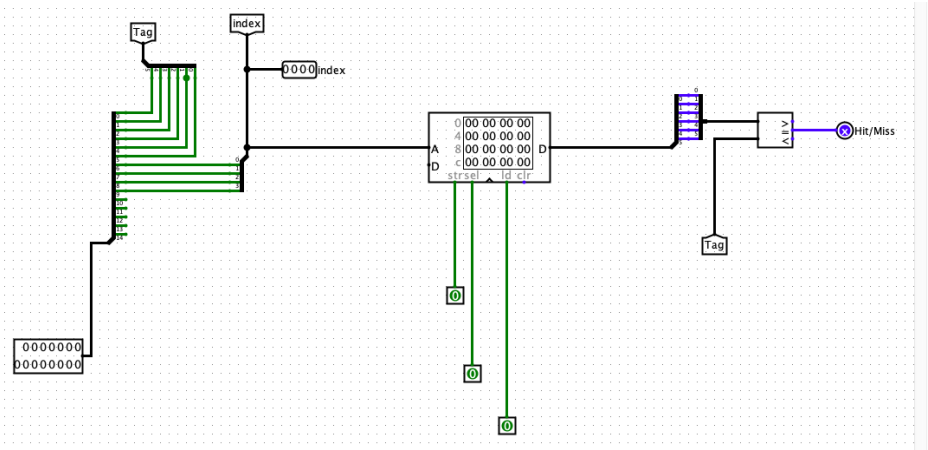
No, in direct map cache management there is no need for any replacement algorithms, as every memory address is uniquely mapped with one of the entires in the cache memory.
This way the new coming block automatically replaces the existing block in the cache memory existing at the specified index.

Implementation of Cache

The cache was implemented using 3 modules :

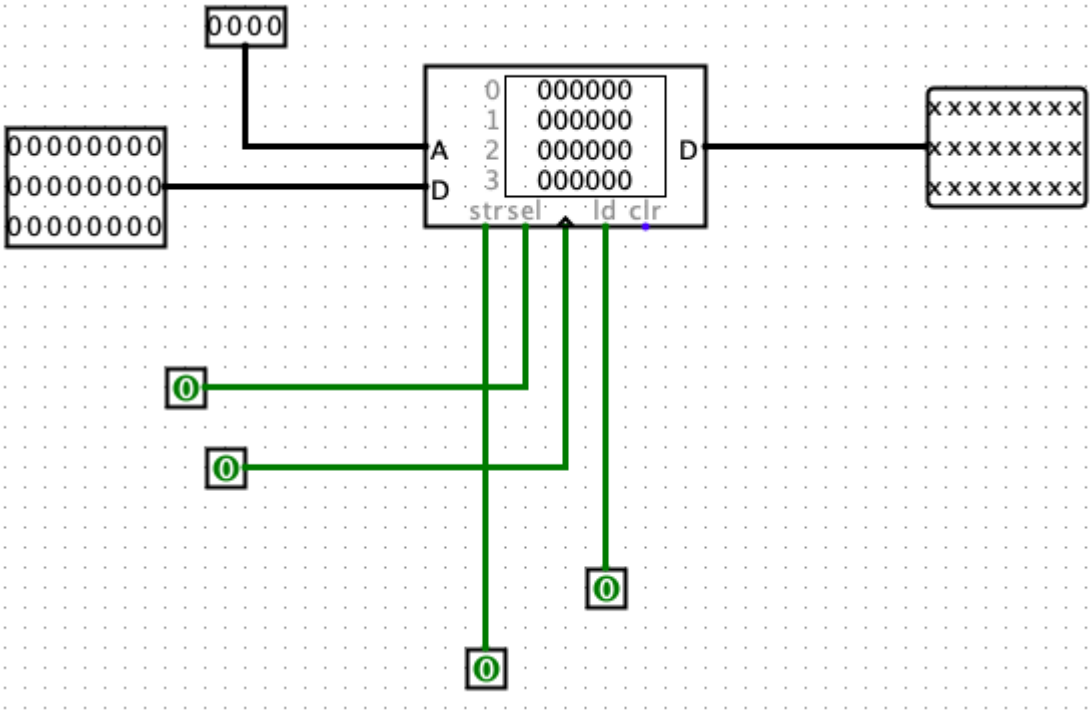
Tag Array

This module is responsible for storing the tags of all the blocks that are present in the cache memory.
This takes the address as the input and checks the tag with each of the tag that is stored in the tag array if there is any match by using a comparator and the result of which is set as hit/miss.
It send the index of the tag at which a match is occurred. If it is a hit then it outputs the index at which the match has occurred.



Block Array

This module is responsible for storing Data of the corresponding to the indices of the tags that are stored in the tag array and when an index has matched and is sent from the tag array, it outputs the data of the corresponding hit.

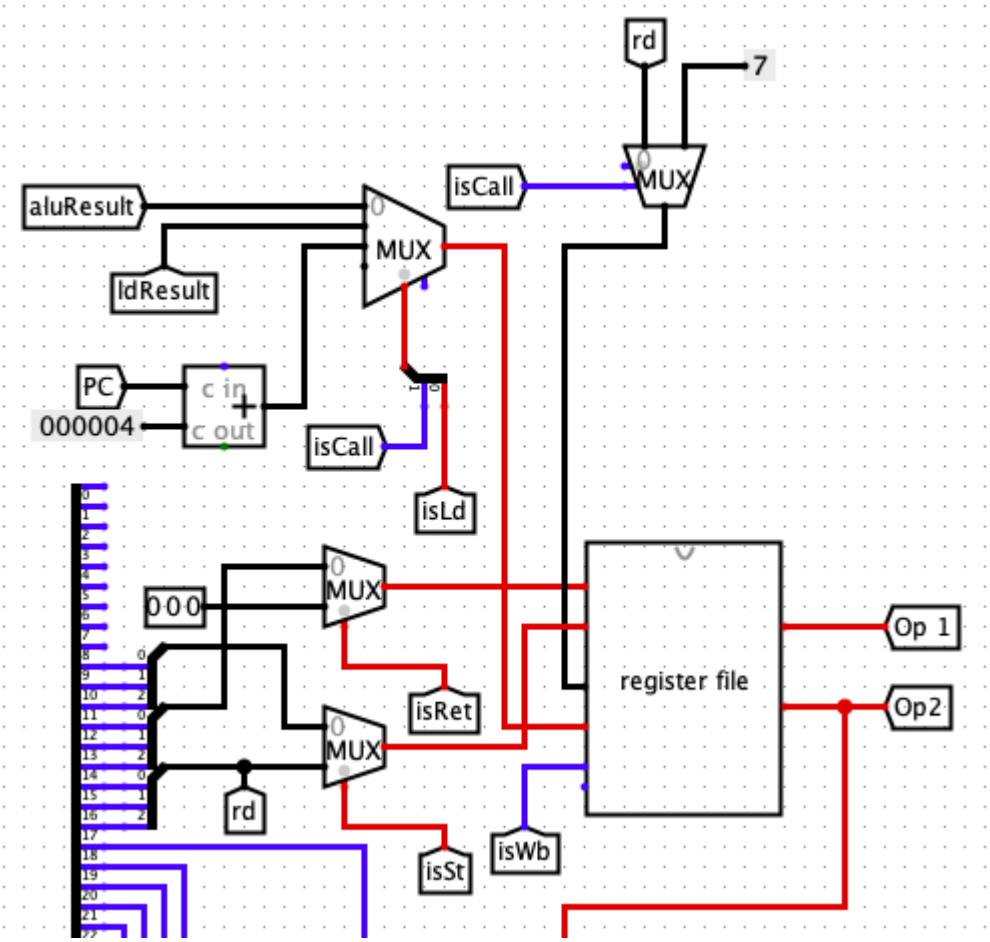


Cache

This module is responsible for connecting the two different parts (block array and the tag array) with the main memory and takes the address and data as input if it is write back and if it is read the it outputs the data corresponding to the particular address.

This methodology is used to implement the cache of the processor.

Register Writeback (RW)



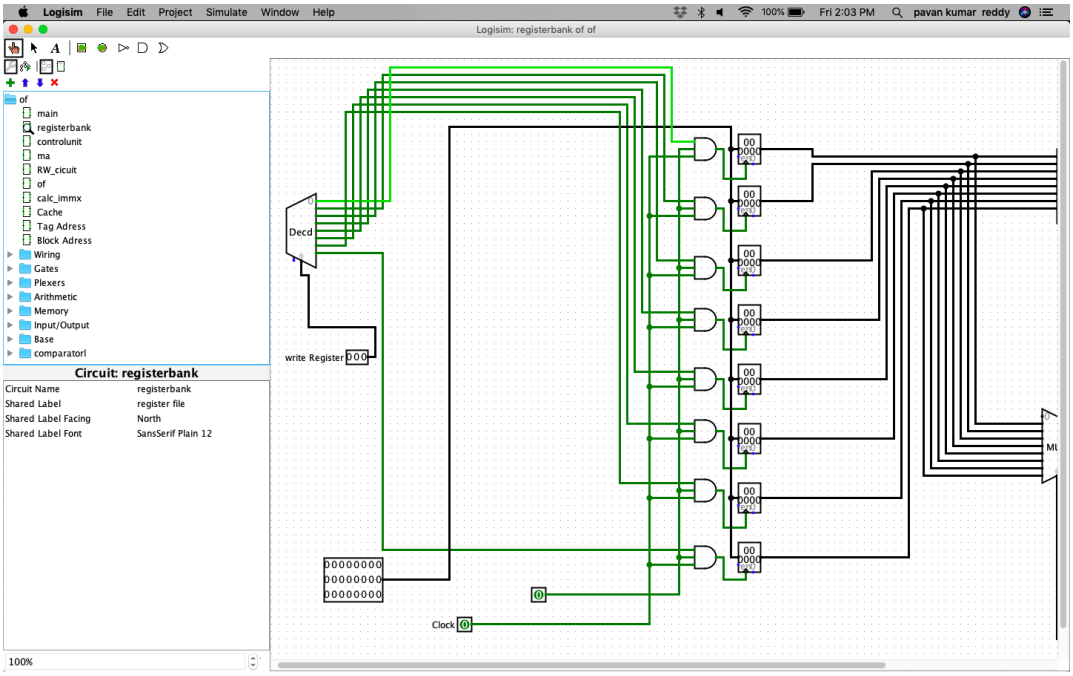
The implementation of RW unit uses another module called **register file** that stores all the general purpose registers (GPRs) of the circuit.

The register writeback writes the data into the register file provided the address of the GPR has been specified using the multiplexer which we can see in the north part of the above figure. This multiplexer either returns return address **ra** (i.e, register number 7) or the destination register **rd** based on the control signal **isCall**.

Once the address has been specified, the data that needs to be stored in the address is decided using another multiplexer that takes the result from the ALU (**aluResult**), **ldresult** and the PC+offset as the input data and works based on the control signals **isCall** and **isLd**.
This second multiplexer determines the data that needs to be written into the address of the register file. The control signal **isWb** is used to specify whether the instruction is a write instruction or read instruction.
The outputs of the register files are the two operands of the instruction.

Example test :
Let the instruction be add r1, r2, r3. For this instruction, after the calculation of the addition by the Arithmetic Unit of the ALU, the result data is send into the **aluResult** tunnel, this happens when the control units **isLd** and **isCall** are both equal to 0. This means that the instruction is just a normal program instruction without the presence of any load or branch calls.
So now the multiplexer feeds the **aluResult** into the register file. The address of the register file to where it is stored is **rd**(destination register). If **isCall** is 0 (as is the case of this example), it is stored in **rd** (which is r1 in this case). Hence, in this way the working of the RW circuit is as expected.

Register File (Register Bank)



This is the register file of the processor. In this, for the implementation of the 24-bit processor, 8 registers are used in the register bank. The register with address 111 is a special purpose register named **ra**, used for storing the return address in case of return instruction and the rest 7 registers are General Purpose Registers (GPRs).

The implementation is done using a decoder that specifies the write address of the register bank, as shown in the figure. The data that needs to be fed into the file is specified using **writeData** input that determines the data that need to be written.

If the instruction is a read instruction then since **isWb** is 0, the output of the register file is the output of the two read ports, which is implemented using two multiplexers based on the read-port addresses specified.

Testing and Results

We simulated and tested our cache processor implementation using some programs with some set of instructions being fed into the instruction memory. Initially, we found some errors which were resolved latter on during the course of the project.

Test 1

ld r0,[1000000000000001] 0111000000100000000001

This instruction loaded the value stored in the address 1000000000001 into the register r0, as expected.

add r0,r1,r2

This adds the content of register r1 and r2 and stores them in the r0 destination register.

add r0,r1, mm

This operations does logical AND of the contents of the register r1 and immediate and then stores them into destination register r0.

Test 2

We gave the instructions:

1. load reg1 address1 (This resulted in HIT)
2. load reg2 address2 (This was a MISS but with empty space in cache)
3. add reg3 reg1 reg2

4. store reg3 address3

In 1, the load operation was with a HIT in the cache and the value at address1 got loaded successfully to reg1. This operation took 1 clock cycle.

The following instructions were given:

1. load reg1 address1 (This resulted in a MISS with no empty space in cache, but no write-back occurred)
2. load reg2 address2 (This was a MISS with no empty space and writeback also occurred)
3. and reg3 reg1 reg2
4. store reg3 address3

The above operations of both the tests were followed by ALU operations that gave correct results corresponding to the required operation and finally store operation was performed (There are 4 variations of the store operation and we have made testbenches and successfully tested the same).

Conclusions

We took this project because of our interest in knowing more about Processor Design and Cache Architecture and to test whether we will be able to apply the knowledge that we learned from the course classes.

Despite a few hurdles at the start due to availability of less material on the web, we were able to successfully run simulations and test our implementation of this architecture after thinking out of the box and making all the designs from scratch.

We were able to learn a lot from this project. Firstly, we learnt to use Logisim very well and came to know all its features and what functionalities they possess. By the end, we were fully comfortable with using Logisim for the design of digital circuits and were fully confident about dealing with any errors that come up while implementing any circuit in Logisim.

We got hands-on experience in the design of processors and also got an in-depth understanding of the Cache Architecture.

This was a learning and highly informative experience for us and we were able to know things in depth and understand the challenges faced in reality.

Also, working as part of group was an amazing experience. We learned how to effectively work in a team and divide the required task amongst our team members and finally to integrate the work done by each individual.

Individual Contributions

Dhruv Shandilya (20114034)

1. Implemented adder, subtractor and comparator
2. Implementation of setting flag registers
3. Took part in contributing to the presentation file

Pinnapu Reddy Harshavardhan Reddy (20114070)

1. Implementation of **ma** in processor design
2. Implementation of cache management system and it's modules
3. Tag array implementation
4. Explanation of cache in the report
5. Contributed to the presentation file

Shaurya Rana (20114086)

1. Design of ISA and control signals
2. Implementation of Control units and generation of control signals
3. Implemented register bank and register writeback
4. Testing and results analysis in report making
5. Contributed to the presentation file

Shaurya Semwal (20119048)

1. Implementation of Instruction Fetch stage
2. Implementation of the Program Counter
3. Explanation of the processor stages in the project report

4. Contributed to the presentation file

Uday Jangir (20114102)

1. Implementation of OF and decoding the immediate
2. Logic and design of branch decision
3. Contributed to the presentation file

Ujjwal (20114103)

1. Implementation of Logic and Shift Units
2. Combined all three units of the ALU
3. Contributed to the presentation file

References

1. Lecture notes of CSN-221 Course
2. Textbook on Computer Architecture
3. Tutorial on Logisim