COMPUTER ARCHITECTURE AND MICROPROCESSOR

CSN-221 PROJECT

IMPLEMENTING A FIVE STAGE PIPELINED PROCESSOR IN ARM ARCHITECTURE ON LOGISIM SIMULATOR

MEMBERS

ANSITA BEHERA (20114014)

CHINAYUSH WAMAN WASNIK (20114027)

DEEPAK AGARWAL (20114028)

NIKHIL (20114063)

PRANJAL SINGH (20114074)

PRATYUSH KUMAR (20114076)

PRIYANSH BHANDARI (20111032)

PUNEET (20114078)

Source Code for the Project:

5 Stage Pipelined processor with branch predictor and cache prefetching

INTRODUCTION

ARM assembly language assumes a machine model similar to that for SimpleRisc. For the register file, it assumes that there are 16 registers that are visible to the programmer at any point of time. All the registers in ARM are 32 bits or 4 bytes wide.

The registers are numbered from r0 to r15. Registers r0 to r12 are generic. The programmer and the compiler can use them in any way they like. However, the

registers r13(sp), r14(lr) and r15(pc) have special roles. sp is the stack pointer, lr is the return address register, and pc is the program counter.

The traditional ARM processor executes instruction in five stages all of which are processed in a single cycle. These five stages being,

- Instruction fetch- Fetches instruction from the op code.
- Operand fetch- Fetches the operand from registers.
- Execute- Performs the necessary logical or arithmetical operations.
- Memory Access- Accesses memory to fetch or store data.
- Register Write Back- Puts the result into the destination register.

Due to this architecture, at any given point of time four of the five stages remain idle. This creates an idle hog and slows the performance. Hence we try to pipeline the traditional processor, so that at any given point of time no stage remains idle.

The five stages are separated by clocked latches. At every clock edge a new instruction will enter the processor in the IF stage and subsequently an instruction will leave the RWB stage. But this Pipelining will create certain hazards in some situations. For example, if the OF stage is trying to fetch data from a register which was modified in the previous instruction. If at this point the previous instruction has not crossed the RWB stage, it will create a read after write hazard.

For example, consider the snippet,

[1]: add r1, r2, r3

[2]: sub r4, r1, r6

When [2] is in OF, [1] is in execute, I.e., r1 has yet not been updated. This creates a read after write hazard. In this project, we try to implement a mechanism to manage these hazards by stalling the program execution for certain number of cycles.

We also try to implement a branch predictor, which will predict the direction of a branch statement without evaluating the conditional statement. This will help in speeding up the execution as we will not have to introduce any more stalls.

PROBLEM STATEMENT

- To implement a five-stage pipelined processor in ARM architecture on Logisim simulator.
- Implementing the necessary hazard management mechanism.
- Implementing a branch predictor.

METHODOLOGY

- Through our lectures, we got to know how to design a Simple RISC processor.
- Based on the lectures and with the help of research papers from internet, we designed ARM Processor with the Logisim software.
- Further we did pipelining of the processor and implemented Branch Prediction and tried to implement Prefetching.
- We mutually collaborated our work using GitHub.

NOVELITY OF THE WORK DONE -

The proposed pipelined processor will speed up the execution by removing the idle hog in the traditional single cycle processor

For example, if a single cycle CPU has a clock cycle of 50ns, then an instruction is executed at every 50ns. Now if we implement the same architecture using pipelining, then assuming that every stage takes equal time the clock cycle would be (50/5) ns + latch delay, which is nearly one-fifth compared to the traditional processor. Hence the efficiency is increased by almost 400%.

Also the possible hazards will be taken care of through the hazard control mechanism.

We also implement a branch-predictor, which is now a days used in all state-of-the-art processor, to increase the functioning speed of the processor.

SOLUTION APPROACH

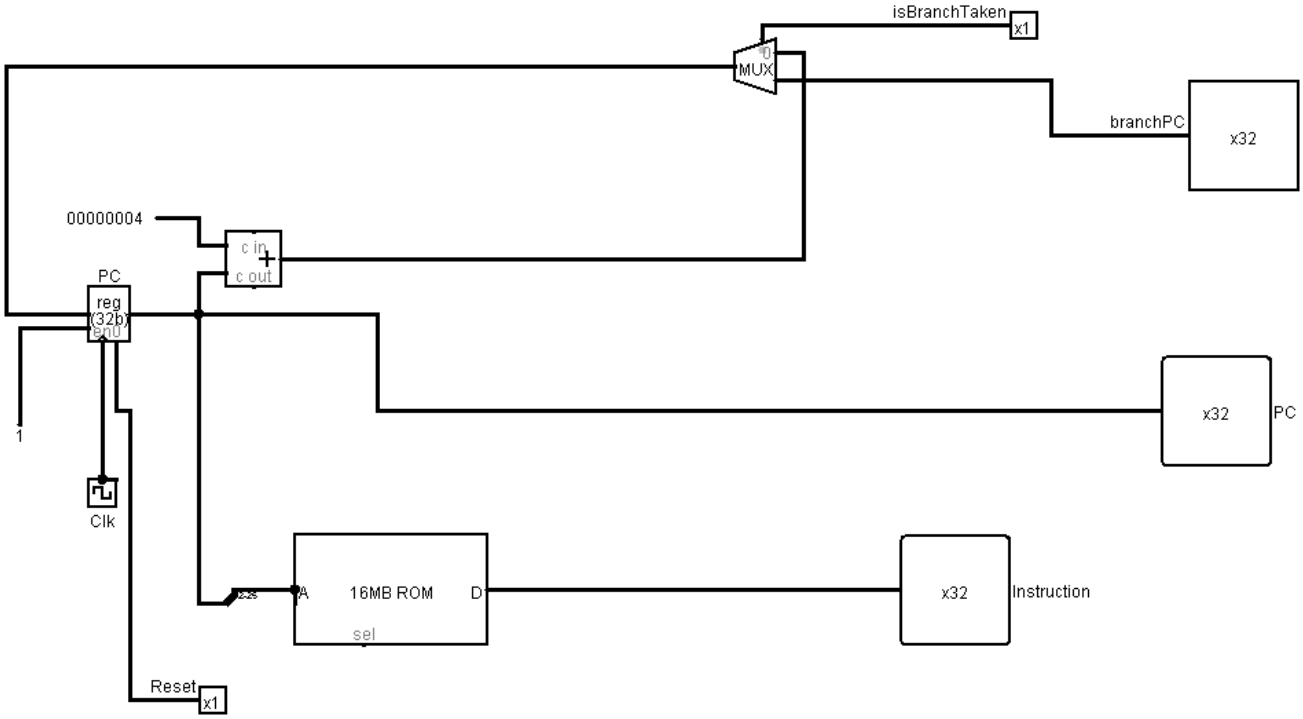
Now throwing some light on the design of the processor-

Processor design is a subfield of computer engineering and electronics engineering (fabrication) that deals with creating a processor, a key component of computer hardware.

The mode of operation of any processor is the execution of lists of instructions. Instructions typically include those to compute or manipulate data values using registers, change or retrieve values in read/write memory, perform relational tests between data values and to control program flow.

The processor which we propose here will execute instructions in the following stages:

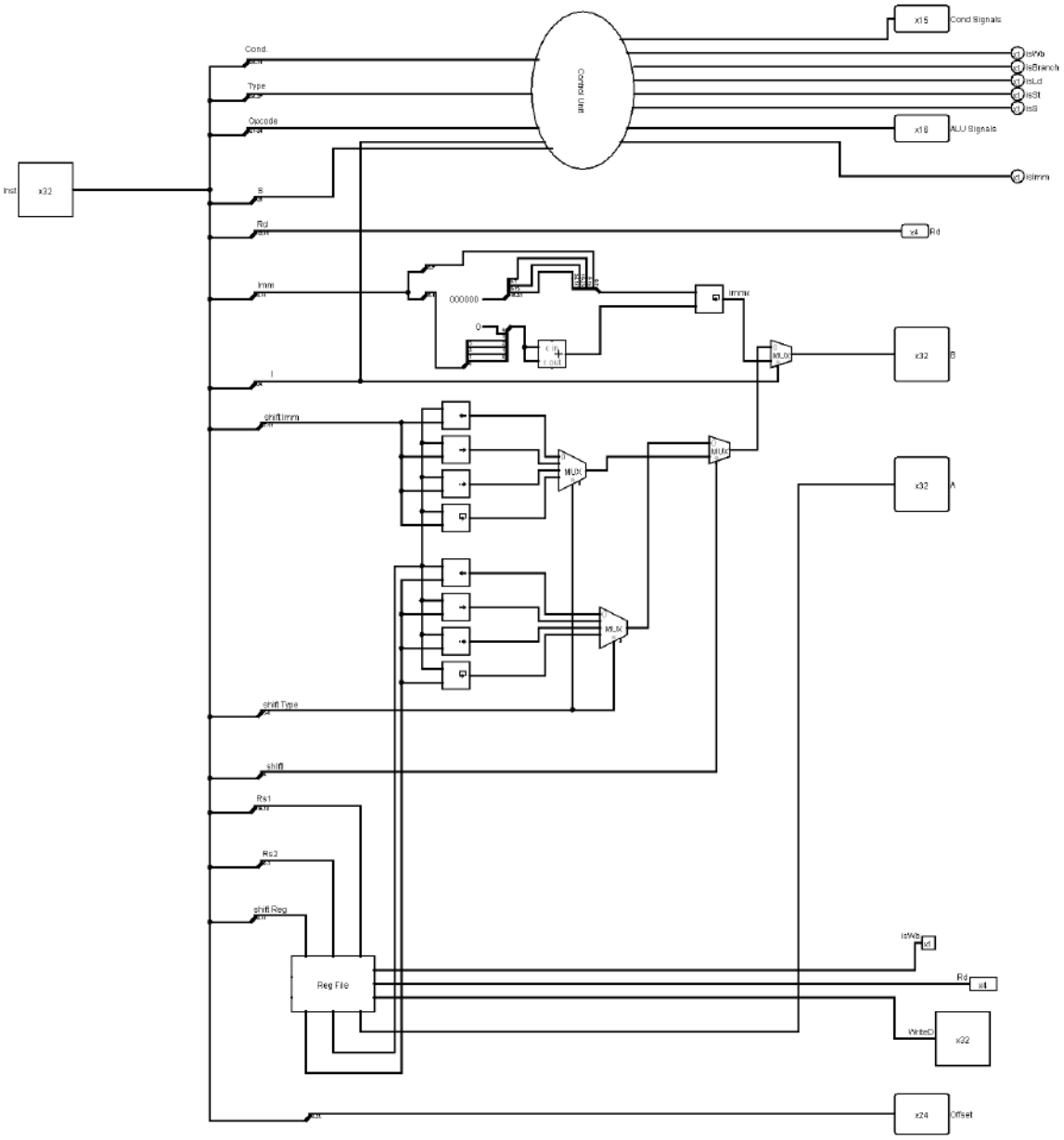
- **Instruction Fetch (IF)** –



This is the first and basic stage of 5 stage pipelined arm processor. This stage contains a fetch unit and an instruction memory. Fetch unit contains a multiplexer and a pc register which contains the program counter triggered by negative edge of clock. We use the pc to access the instruction memory. The multiplexer chooses between pc + 4 and branch target. It uses a control signal “isBranchTaken” which is generated by EX(Execute) stage. Now, what happens is that if isBranchTaken signal is 0, then pc will always take pc+4 address but if it is 1, then the given branch target from EX stage will be taken. The taken address by pc (32 bit) will be sent to instruction memory by converting it to 24-bit address through splitter and

accordingly the instruction will be fetched from instruction memory and will be forwarded to EX stage along with address stored by pc. Also, the pc after sending to instruction memory will store the address of next instruction. This is how this stage is actually implemented.

- **Operand Fetch (OF)**-



The instruction fetch stage send the instruction and program counter to the operand fetch stage.

The instruction is of 32 bits (named with 0-31, 0 as Least significant bit and 31 as MSB) and has the following distribution:

- 31-28: Condition Field

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	Not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	Positive or zero
0110	VS	V set	overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N equals V	Greater or equal
1011	LT	N not equal to V	Less than
1100	GT	Z clear AND (N equals V)	Greater than
1101	LE	Z set OR (N not equal to V)	Less than or equal
1110	AL	(ignored)	always

- 27-26: Type Code
- 25 – Immediate Bit
- 24-21: OpCode

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 Exclusive OR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry

SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

- 20: Set condition codes
- 19-16: Rn – First Operand Register
- 15-12: Rd - destination register
- 11-0: Operand 2

It is divided in different ways according to different conditions:

→ If immediate is 1:

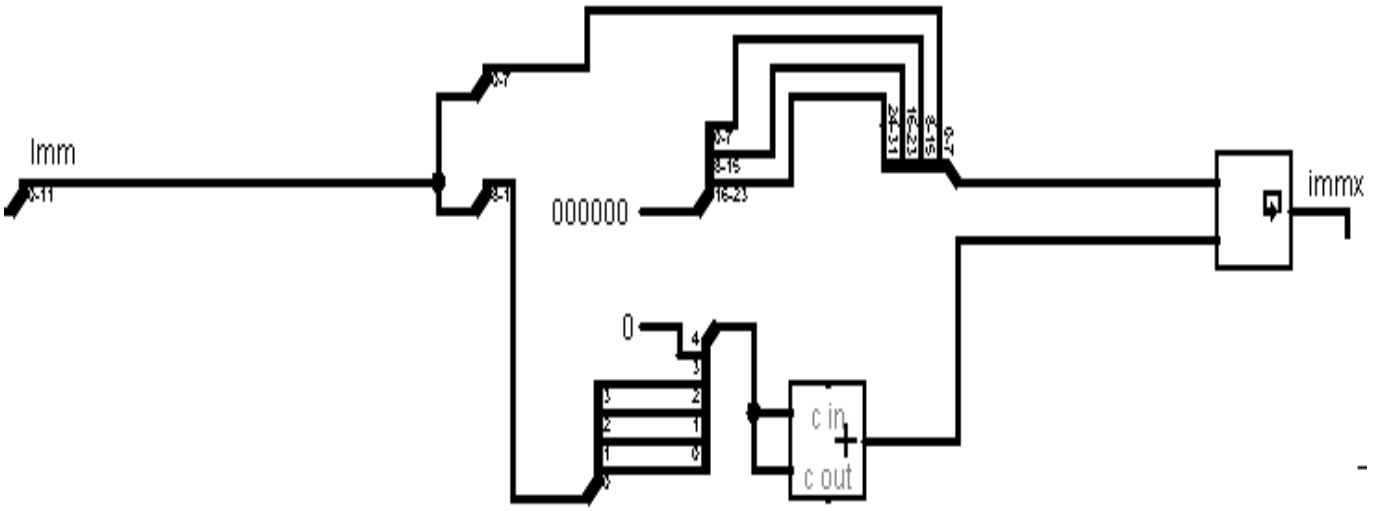
11-8: Rotate Bit (rot)

7-0: Immediate Bits (payload)

The immediate is calculated using:

Payload ror (rot*2)

(Circuit made using rot*2 = rot + rot)



→ If immediate is 0:

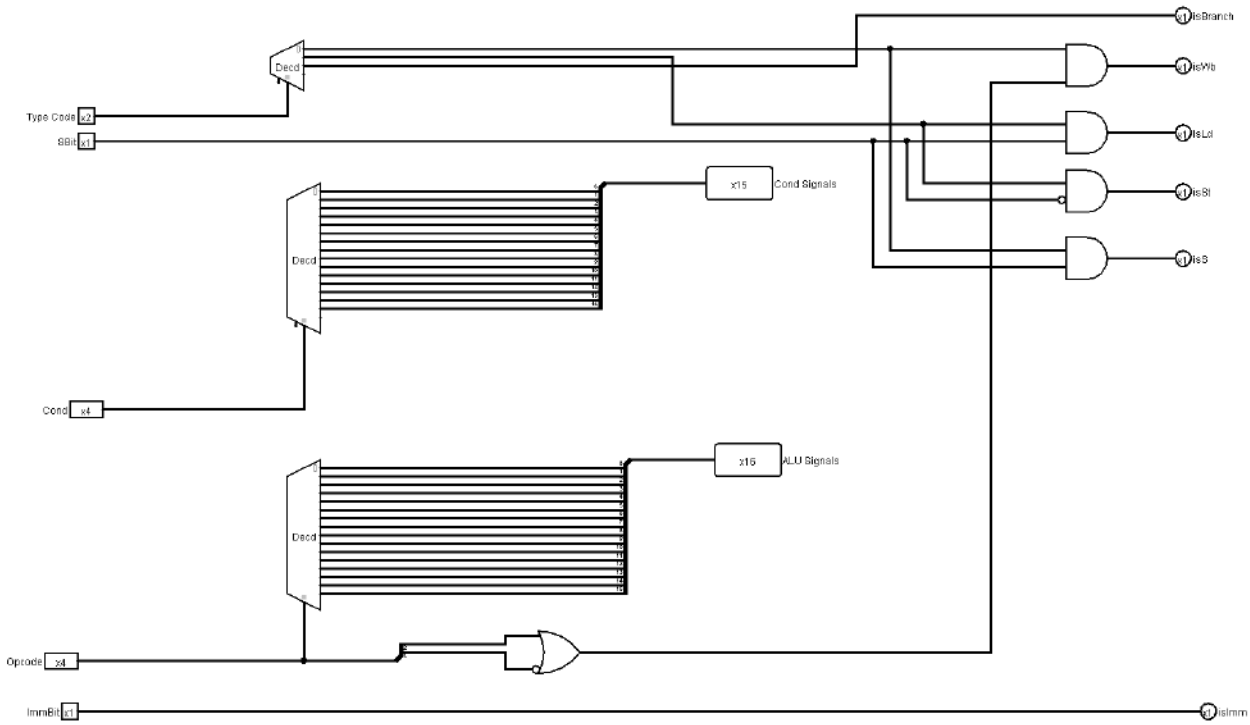
Then 4: ShiftL

- ❖ If ShiftL is 0:
  - 11-7 : Shift Imm
  - 6-5: Shift Type
  - 3-0: Rm - 2nd operand register
- ❖ If ShiftL is 1:
  - 11-8: Shift Register
  - 7: NA
  - 6-5: Shift Type
  - 3-0: Rm - 2nd operand register
- Shift Type:
  - Lsl 00
  - Lsr 01
  - Asr 10
  - Ror 11

For branch instructions, 23-0 are used for offset.

The condition field, type code and opcode are sent to the Control Unit.

Control Unit :-

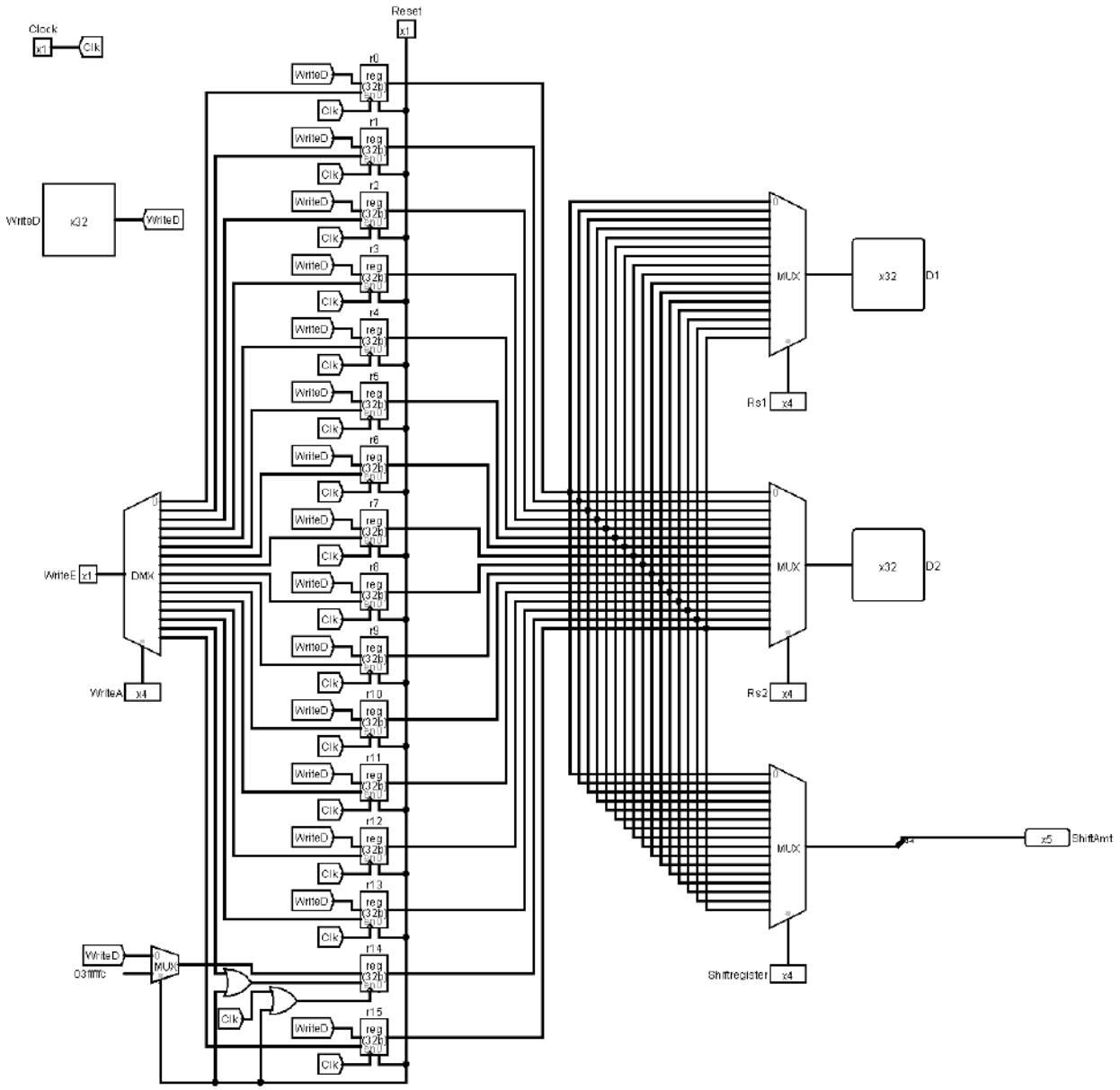


It sends the condition signals, ALU signals to further stages to execute the processor. Now, comes to working of CU.

There is a decoder which will take 2 bit type code as input. Along with the S bit , the output will be determined through various AND gates. Like if type code is 00 and S bit is 0/1, then these bits will go to AND gates as inputs and the output signal “isWb” will be executed. In a similar way, the output signals will be implemented corresponding to these bits. Also for “isWb” output signal , 2 bits of opcode passing through OR gate is also taken as input in AND gate only for this signal. Now, for both cond and op code, there are separate decoders for both having their 4 bits code as input and corresponding to that, they decode those bits into no. Bits which represent different conditions and operations which further through reverse splitter gets converted to a single 16 bit code. For e.g., opcode is 0000, then decoder will give 0 as output which further converted to 16 bit binary code. In this way, both the decoders work. The I, i.e. , ImmBit decides whether the operand will be immediate or not and give output 1 when it will be an immediate.

Rn, Shift Register, Rm are connected to the Register File which output the appropriate register data using multiplexers in which selection line is Rs1, Rs2, Shift Register.

Register File :-



- **Execute(EX)**

Contains an Arithmetic-Logical Unit (ALU)

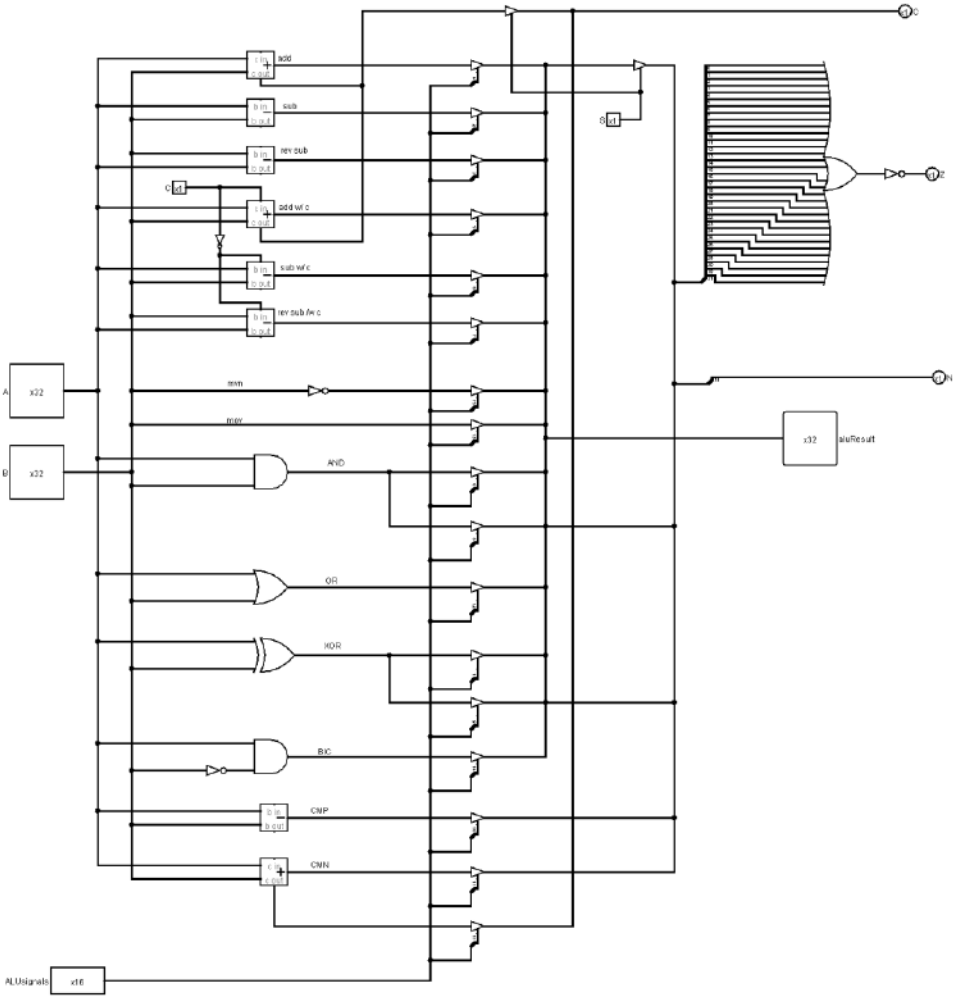
This unit can perform all arithmetic operations -

→ ADDITION(ADD)
→ SUBTRACTION(SUB)
→ REVERSE SUBTRACTION(RSB)

→ ADD WITH CARRY(ADC)
→ SUBTRACT WITH CARRY (SBC)
→ REVERSE SUBTRACT WITH CARRY(RSC)
→ MOVE(MOV)
→ MOVE WHEN NEGATIVE(MVN)
→ COMPARE(CMP)
→ COMPARE WHEN NEGATIVE(CMN)
→ TEST EQUALITY (TEQ)
→ TEST(TST)

and logical operations-

→ AND(AND)
→ OR(ORR)
→ XOR(EOR)
→ BIT CLEAR(BIC)



Now we will explain the working of the ALU:

From control unit we will get the ALU signals and corresponding buffer will get active which will send the output to the ALU result .

For example, if we need to add two inputs A and B of 32 bits respectively

Then the corresponding ALU signal will be sent by control unit and we will get the corresponding result.

If we need to set the flags via adding two numbers then here the 'S' input will be set to '1'. Thus, the buffer will get active and will set the zero, negative and the carry flag.

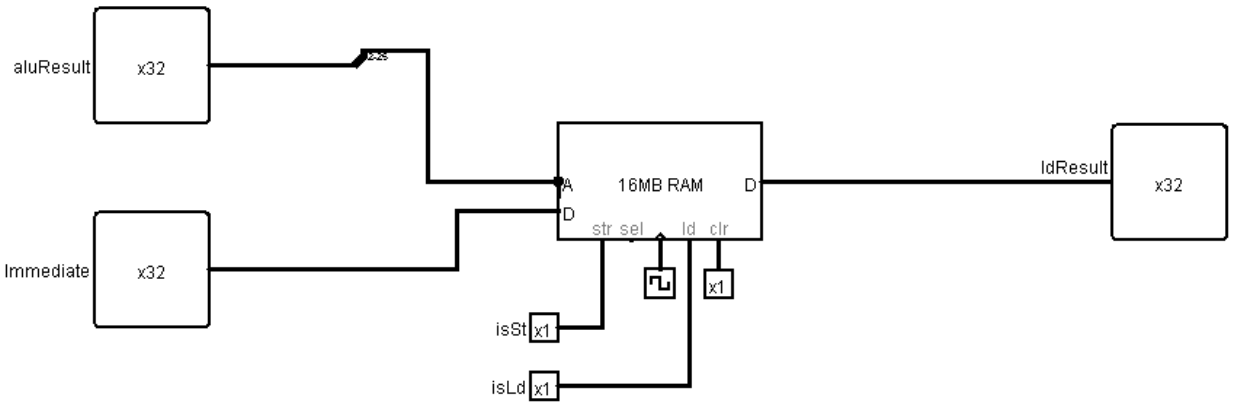
Contains the branch unit for computing the branch condition -

- BRANCH IF EQUAL(beq)
- BRANCH IF NOT EQUAL(BNE)
- BRANCH CARRY SET (BCS)
- BRANCH CARRY CLEAR(BCC)
- BRANCH NEGATIVE/MINUS (BMI)
- BRANCH POSITIVE OR ZERO/PLUS (BPL)
- BRANCH UNSIGNED HIGHER (BHI)
- BRANCH UNSIGNED LOWER OR EQUAL (BLS)
- BRANCH SIGNED GREATER THAN OR EQUAL (BGE)
- BRANCH SIGNED LESS THAN (BLT)
- BRANCH SIGNED GREATER THAN (BGT)
- BRANCH SIGNED LESS THAN OR EQUAL(BLE)
- BRANCH ALWAYS (BAL)

Contains the flags register (updated by the cmp instruction)

WORKING -

- **Memory Access(MA)**



Memory Access is the fourth stage of the pipeline. Its purpose of existence is to implement load(ld) and store(st) instruction.

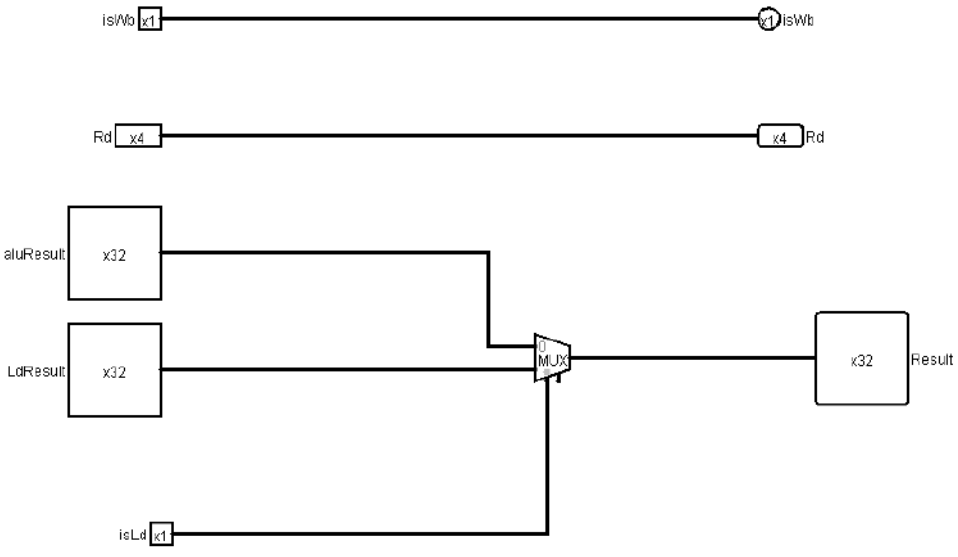
→ For load instruction

Here the result obtained from the ALU is the memory address from where the data must be fetched. The MA unit fetches this data and passes to the RWB unit.

→ For Store instruction

Here the ALU result contains the address where the data must be stored. The MA unit fetches the data directly from the 32 bit instruction and stores it in the designated memory location.

- **Register Write Back(RWB)**



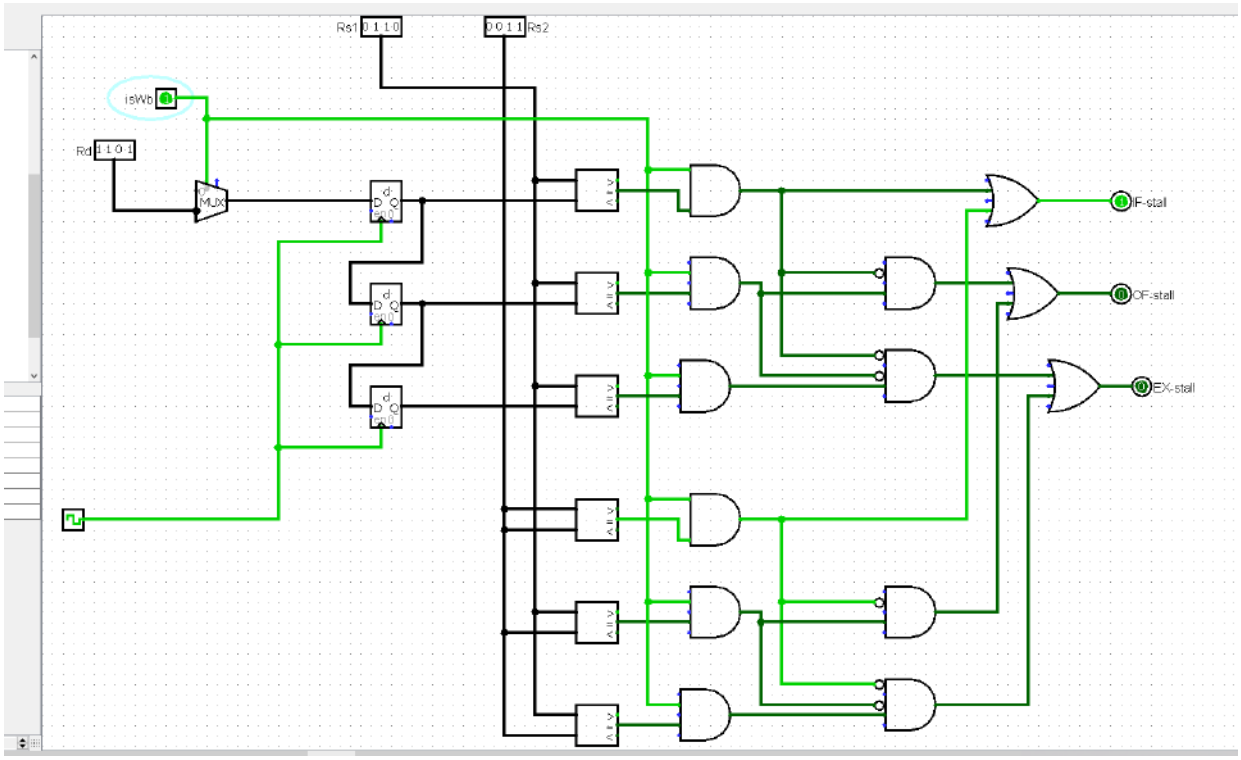
This stage receives the result of the arithmetic or logical operation performed by the ALU. It extracts the destination register from the 32 bit instruction and writes the aluResult in it. It also checks for the possible hazards and adds stall to prevent them.

DATA STALLING

In any Pipeline processor, it is important to keep in mind that there is very high probability that we need to read a file which has not been written to yet. Such data hazards need to be resolved otherwise our required result will not be correct.

For an in-order pipeline such as ours, we only need to be careful of the RAW (Read after Write) or Real Data Hazards.

In out processor, we do this by stalling the pipeline till the required register has been written to.



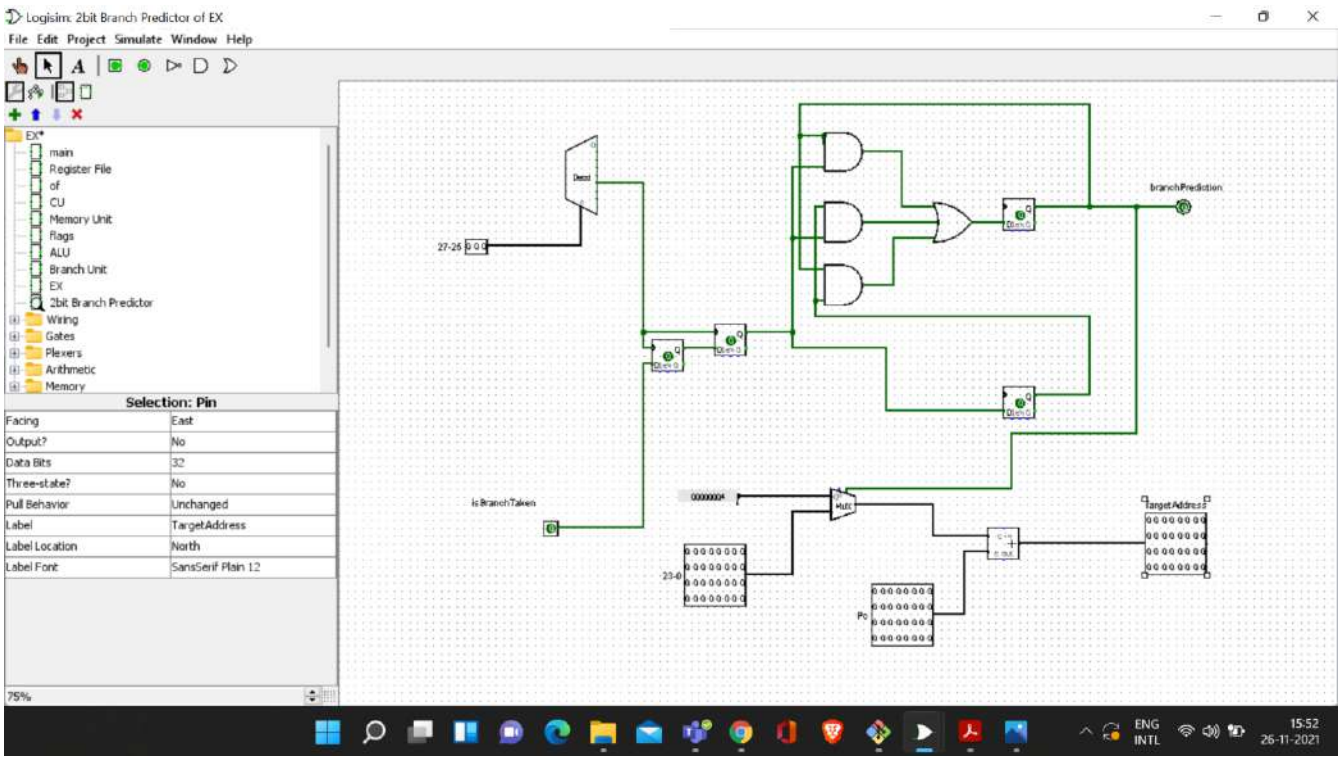
In our data stall unit, the register which are to be written to are stored as long as they have not been written completely. In case we encounter a register to read such that it is currently in the process of being written, then the pipeline is stalled, as in it does not move forward till the required register has been written with the correct value.

BRANCH PREDICTION

The ability to predict the direction of branches is an important issue in modern computer architecture and advanced compilers. Today, all state-of-the-art

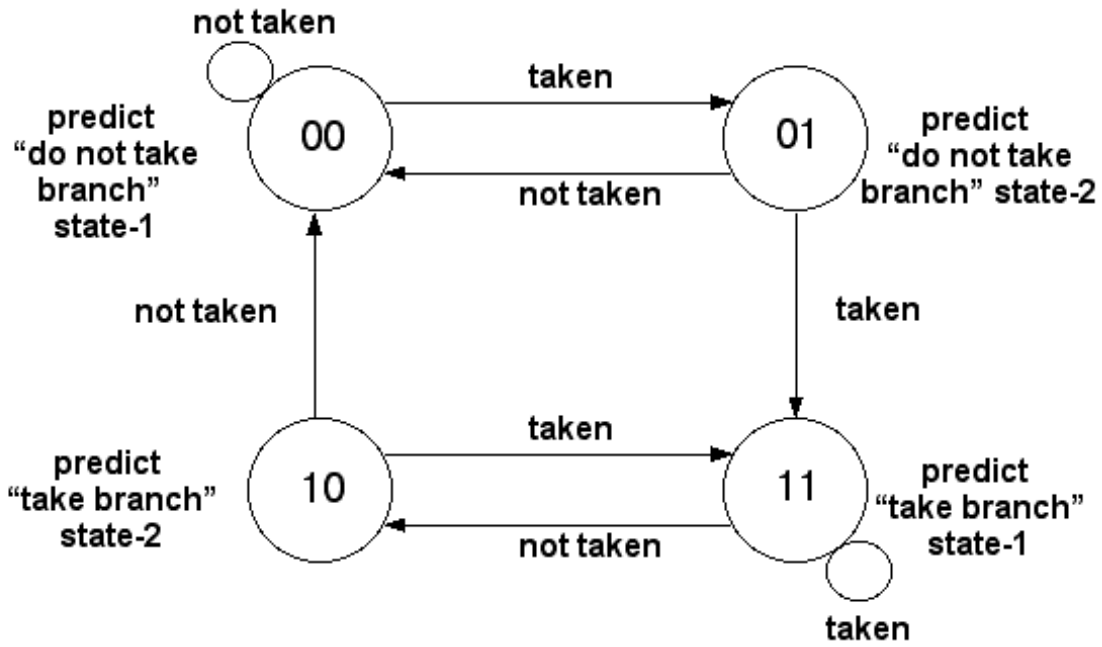
microprocessors have branch prediction of static (software) and dynamic (hardware).

In this project we have implemented a two-bit dynamic branch predictor.



The two D flip-flops serve a master slave role and store the information of weather the branch was taken or not the last time a branch statement was called.

The branch predictor here works as a finite state machine corresponding to the following state table.

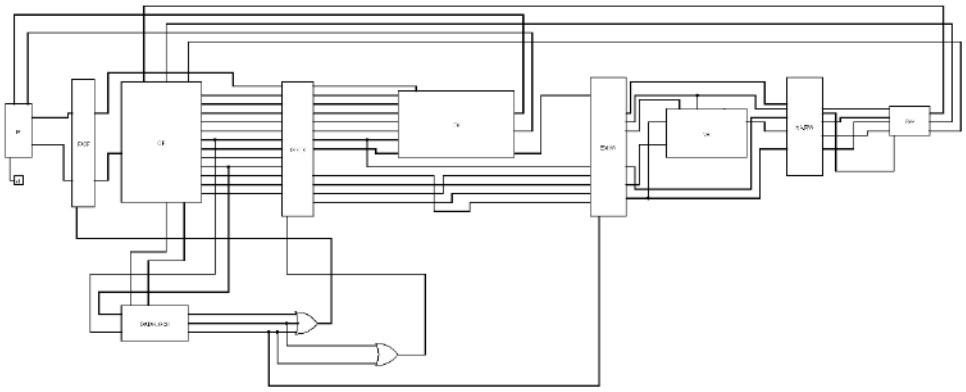


This circuit predicts weather the branch is taken or not. It also evaluates the target address corresponding to its prediction. This target is stored in the program counter and the execution continues without any stalls.

Further in the Alu, if it is found that the prediction was false, then the program counter is again set to the branch instruction and is again executed with the correct paradigm.

Probabilistically, the branch predictor predicts correctly around 50% of the time. Thus, we are saving a lot of time by not introducing stalls and going on with the prediction. Hence, the branch predictor helps in speeding up the execution rate of the architecture.

The final developed processor after interconnecting all parts



PREFETCHING

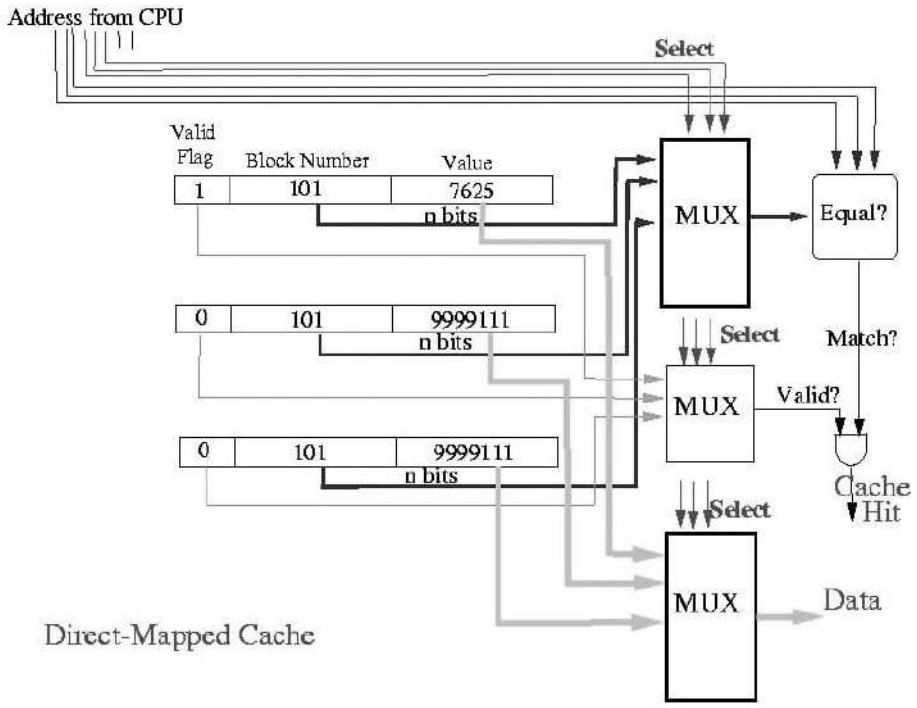
Cache prefetching is a technique used to boost the performance of processors by fetching instructions or data from the main memory to cache memory for faster access. Prefetching the data before it is required saves a lot of time for the processor, thereby able to execute larger number of instructions in a shorter time.

In this implementation of ARM processor, we are trying to prefetch instructions/data by:

1. Checking whether the address that the instruction/data is trying to access, is present in cache memory or not, using direct cache mapping.
2. If the address is present, it results in a cache hit. If not, it results in a cache miss.
3. When there is cache miss, the concept of stream buffer is used.

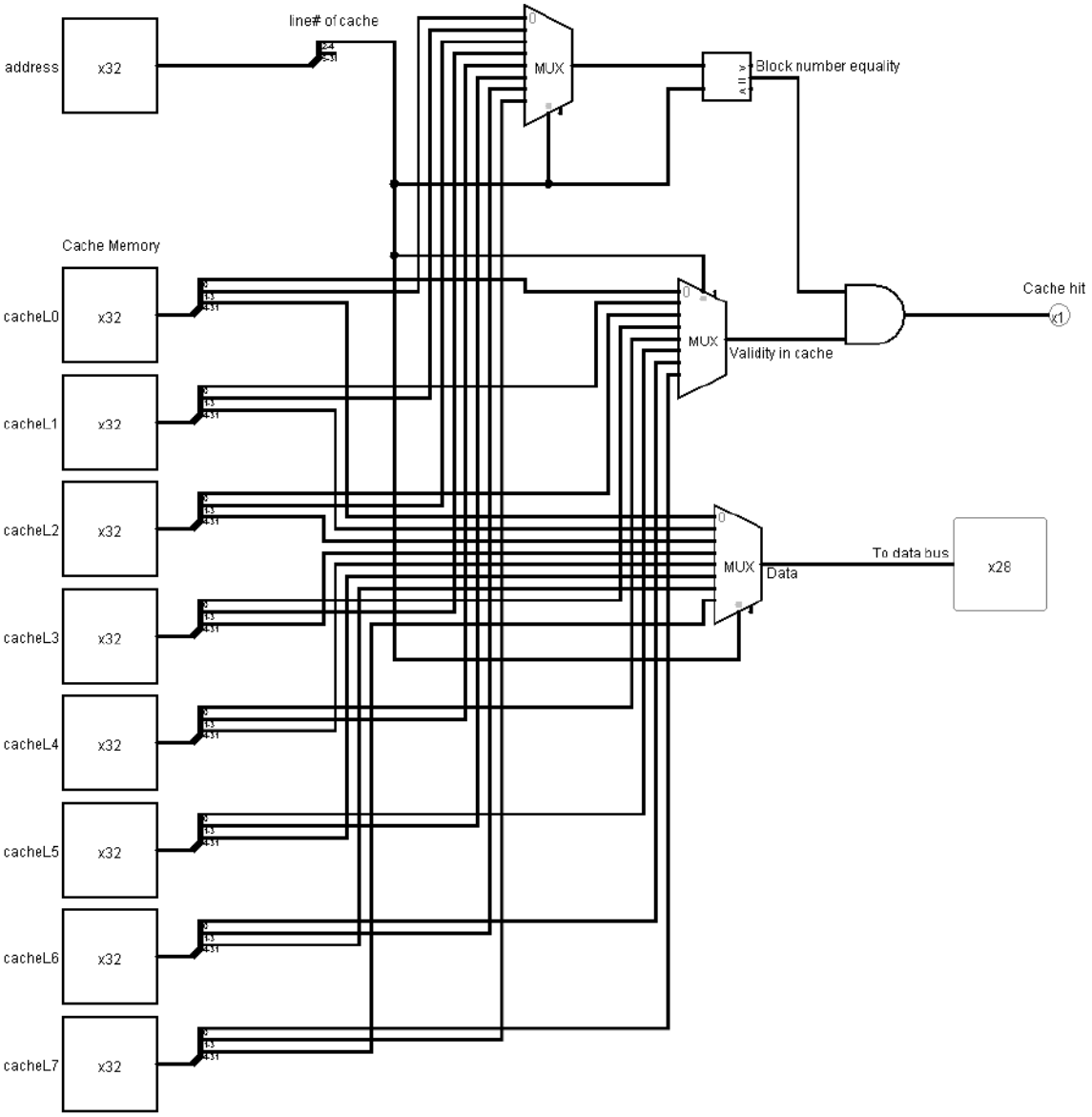
Direct cache mapping:

In direct cache mapping, we compare the address required by the instruction/data in the cache memory. We check the whether the line number (as stated in the instruction/data address) of cache contains the same block number as specified by the address. If both the block numbers are same, then it results in a cache hit, and the value stored in that block number is provided to the data bus. A pictorial representation of direct cache mapping is given below:

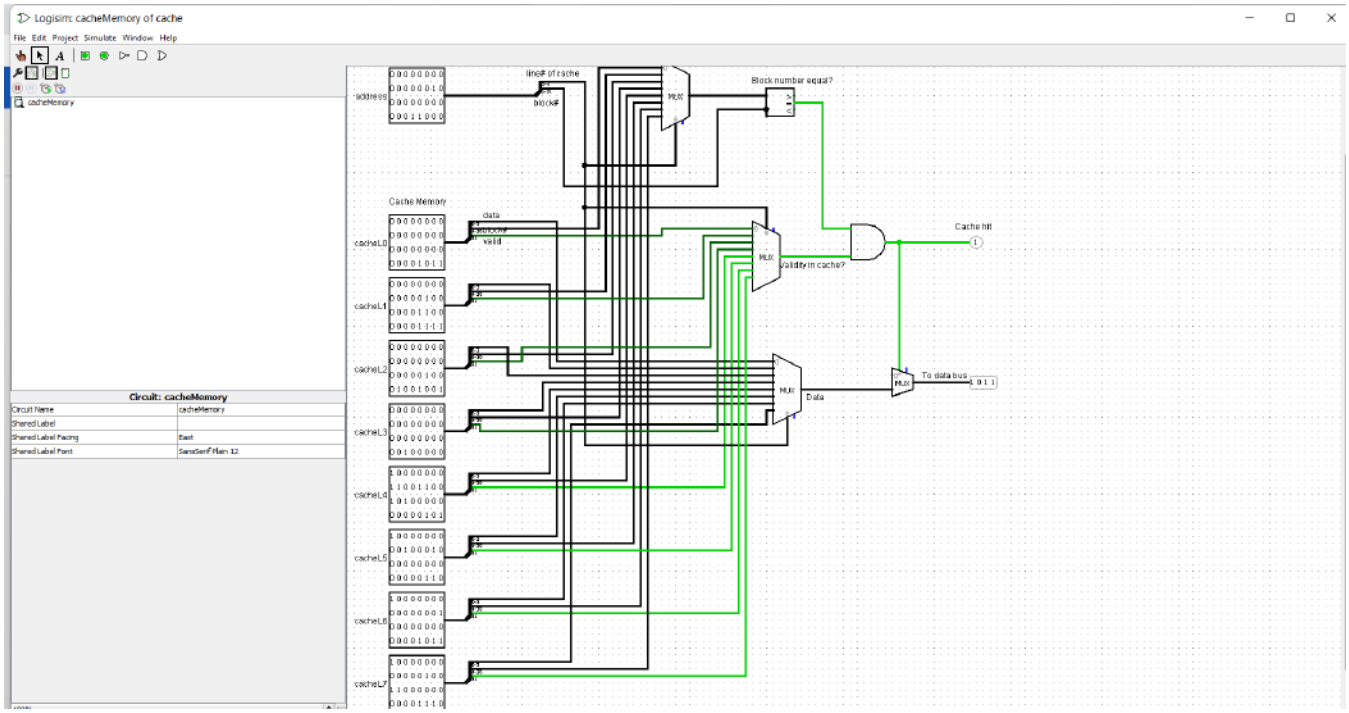


[Source: stated in references]

The implementation of direct cache mapping and cache hit implemented in logisim:



Working can be seen as:



If the cache mapping results in a cache miss, we proceed to use stream buffer. Using stream buffer, we allocate the succeeding cache blocks into a stream buffer of size, let's say, 4. We start searching for the required address in the stream. If address is found, value is returned, and this cache block is placed up as "most recently used". If not, then the search goes on, and if not found in cache memory, then main memory is accessed.

RESULTS AND DISCUSSION -

The functioning of any CPU based on van numen model can be seen to implement any particular instruction in five stages viz,

Instruction fetch, Operand Fetch, Execute, Memory access and Register writeback. In a traditional processor, all five stages are executed in a single clock cycle Hence when an instruction is in one of the stages, the rest four-stage remain idle. Hence in a traditional processor, the efficiency at any given instant is around 20%. This hinders with the performance of the CPU.

To overcome this idle hog, we introduce the principles of pipelining in the traditional processor. Now all the five components are made mutually independent and operate on the instruction separately.

The instruction is passed through each stage sequentially and gets operated by it. All the stages are separated by latches and the instruction is passed from one stage to the next at every clock edge. This approach provides that no part of the

CPU is idle at any given instance (unless the CPU itself adds a stall). Hence the performance of the CPU is increased exponentially.

For example, if a single cycle CPU has a clock cycle of 50ns, then an instruction is executed at every 50ns. Now if we implement the same architecture using pipelining, then assuming that every stage takes equal time the clock cycle would be (50/5) ns + latch delay, which is nearly one-fifth compared to the traditional processor. Hence the efficiency is increased by almost 400%.

The introduction of pipelining also introduces certain hazards in the architecture. Like, it may happen that an instruction tries to fetch the value stored in a register, but the register value was changed if the preceding instruction. Hence the OF of second instruction cannot be executed until RWB of the first is done. In such cases the processor has to introduce stalls between such instructions so that the correct value is read. Hazards also arris in branching statements where at certain times unwanted statements may get executed. In this case also the CPU introduces stalls to manage the hazard.

Also, a two-bit branch predictor is used to predict the direction of branch before it is determined by the ALU. Here the processor goes forward with execution without stalling. When this prediction matches the evaluated result from ALU the processor continues its execution, otherwise, the pc returns to the branching statement and the instructions are repeated to counterpart any error that may have incurred. This branch predictor provides a significant boost to the programme execution as it is helping the processor avoid stalls.

We tried to implement cache prefetching using direct cache mapping, which helps the processor to access values from memory faster, thereby enabling it to perform many tasks in lesser time.

CONCLUSION-

We have implemented a five-stage pipelined processor based on ARM architecture. The processor executes any instruction through a five-stage pipeline consisting of the following stages:

a) Instruction Fetch

b) Operand Fetch

c)Execute

d)Memory Access

e) Register write back

This processor provides for better performance than a single cycle processor by irradicating the idle hog.

Hazard controls are implemented, which stall the certain stages of the processor for certain cycles whenever there is a possibility of a misread of data.

A two-bit branch predictor is implemented which predicts the direction of branch statements. This helps in reducing the execution time as in this case the processor will need not have to stall to manage hazards.

Cache prefetching using direct cache mapping, which helps the processor to access values from memory faster, thereby enabling it to perform many tasks in lesser time.

REFERENCES AND BIBLIOGRAPHY-

Arm Cortex-R52 Processor Technical Reference Manual r1p1
Basic Computer Architecture

Logisim Documentation (Church)

Pipelined Processor

Memory Unit in a Processor

Instruction Memory

Direct Cache Mapping

INDIVIDUAL CONTRIBUTION

Ansita Behera (20114014)- Prefetching

Chinayush Waman Wasnik (20114027)- Pipeline Data Stalls, RegisterWrite Unit

Deepak Agarwal (20114028)- Operand Fetch

Nikhil (20114063)- Instruction Fetch, Control Unit

Pranjal Singh (20114074) - Register file, Memory unit

Pratyush Kumar (20114076) - 2-bit Branch Predictor

Priyansh Bhandari (20111032)- Branch Unit, Flags

Puneet (20114078)- ALU