Computer Architecture and Microprocessor (CSN-221)

SIMPLE RISC PROCESSOR ON LOGISM SIMULATOR

- Kancharla Nikhilesh Bhagavan (20114043)
- Kudikala Rishikesh (20114046)
- Madamanchi Ashok Chowdary (20114052)
- Mallamgari Nithin Reddy (20114053)
- Kyamaji Vedanth Vardhan (20114048)
- Murthati Mahi Babu (20114058)
- Baddam Venkat Praneeth Reddy (20114040)
- Macharla Sri Vardhan (20114051)

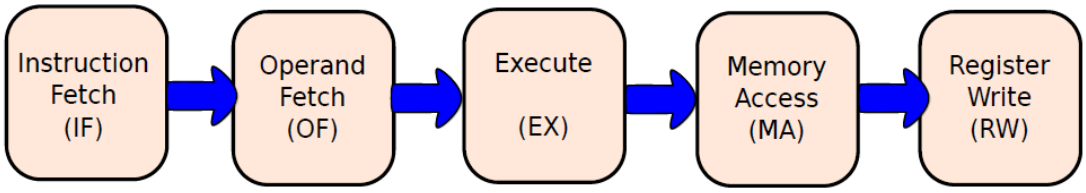
CONTEXT

- Problem statement
- Novelty of work done
- Individual Contributions
- Evaluation Parameters
- Results and discussion
- Instruction Format
- Conclusion
- Bibliography

PROBLEM STATEMENT

Designing a processor which has instruction sets that are simple and having simple addressing modes. And also the execution of the instructions are faster and take one clock cycle per instruction.

- Here we have to design a simple RISC processor (Reduced Instruction Set Computer) which streamline and accelerate data processing by minimising the number of instructions permanently stored in the microprocessor and by relying more on non-resident instruction.
- SimpleRisc is a simple and regular instruction set. It has three classes of instruction formats, branch, register, and immediate. The add, sub, mul, div, mod, and, or, cmp, not, lsl, lsr, asr, and mov instructions can have either the register or the immediate format. This is decided by the I bit (27th bit) in the instruction. The cmp instruction does not have a destination register. The mov and not instructions have only one source operand.
- We can broadly divide the operation of a processor into five stages.



NOVELTY OF THE WORK DONE

- We have successfully designed a hardwired processor that implements the entire SimpleRisc ISA.
- We have designed a processor(processor.circ) than can execute instructions faster i.e., within one clock cycle we can execute an instruction.
- We have aslo developed another circuit(extension of processor folder – extended processor(pipelined)) for the processor which executes an instruction in 5 stages in 5 clock cycles. This was a trial for making the circuit into a pipeline system, we can extend this circuit to make it a pipeline system.
- We have designed an assembler code in java that interprets and converts the assembly program into hexadecimal instructions that can be directly loaded into the instruction memory.
- Our processor will work efficiently for any sequence of instructiuons without any errors i.e., hazards are minimised.
- We have also added two extra instructions i.e., xor and xnor which are not in the SimpleRisc.

INDIVIDUAL CONTRIBUTIONS

- Kancharla Nikhilesh Bhagavan (20114043) – **Integration of all circuits + debugging.**
- Kudikala Rishikesh (20114046) **– Integration of all circuits + debugging.**
- Madamanchi Ashok Chowdary (20114052) – **Instruction Fetch + Memory Access**
- Mallamgari Nithin Reddy (20114053) – **Operand Fetch**
- Kyamaji Vedanth Vardhan (20114048) – **Execution Stage**
- Murthati Mahi Babu (20114058) – **Register Write**
- Baddam Venkat Praneeth Reddy (20114040) – **Assembly Code**
- Macharla Sri Vardhan (20114051) – **Branch units in different subcircuits + Evaluation and debugging**

EVALUATION PARAMETERS

Evaluation parameters and Approach:

We tested our Processor with different kind of SimpleRisc instructions such as branch, flag , arithmetic instructions individually and we also have tested with a programme which includes all kinds of instruction formats i.e.: “factorial of a number”. Below code is the implementation of factorial of a number:

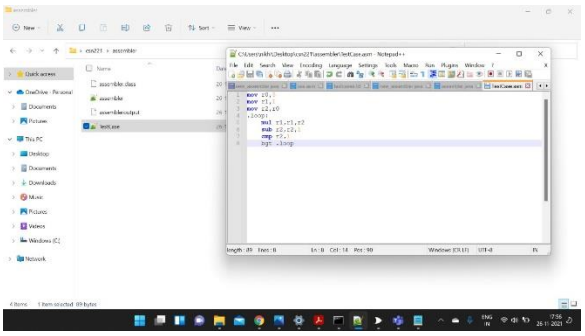
Test Case 1:

Factorial of a “5”:

High level language:

```
int num=5;
int prod=1;
int idx;
for  ( idx=num ; idx>1;idx--){
      prod=prod*idx;
}
```

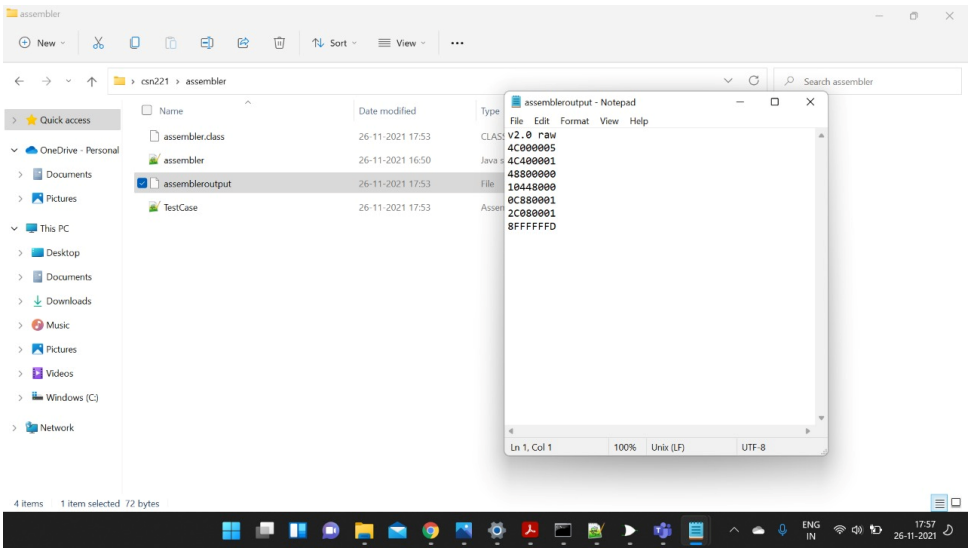
Assembly language:



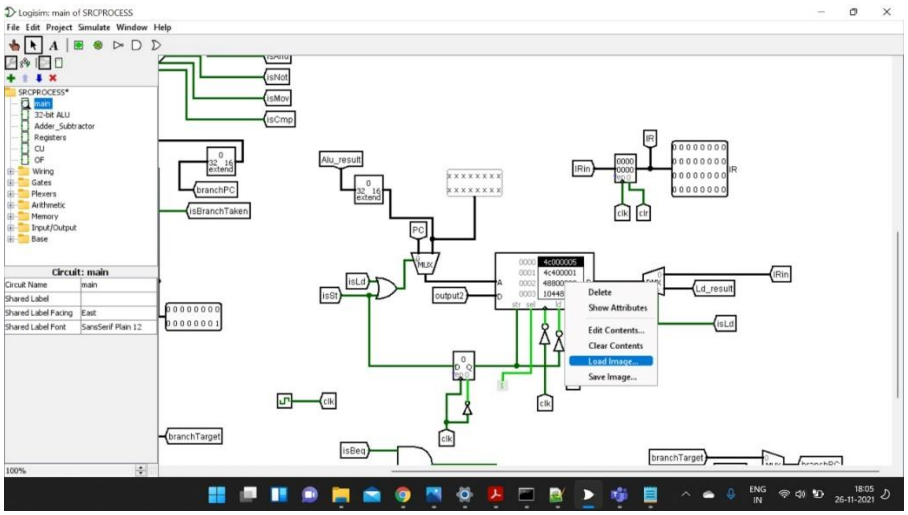
```
mov r0,5
mov r1,1
mov r2,r0
.loop:
    mul r1,r1,r2
    sub r2,r2,1
    cmp r2,1
    bgt .loop
```

ASSEMBLER

Hexadecimal instructions of the program:



- So, assembler converts the assembly program with instructions into machine executable instructions as shown.



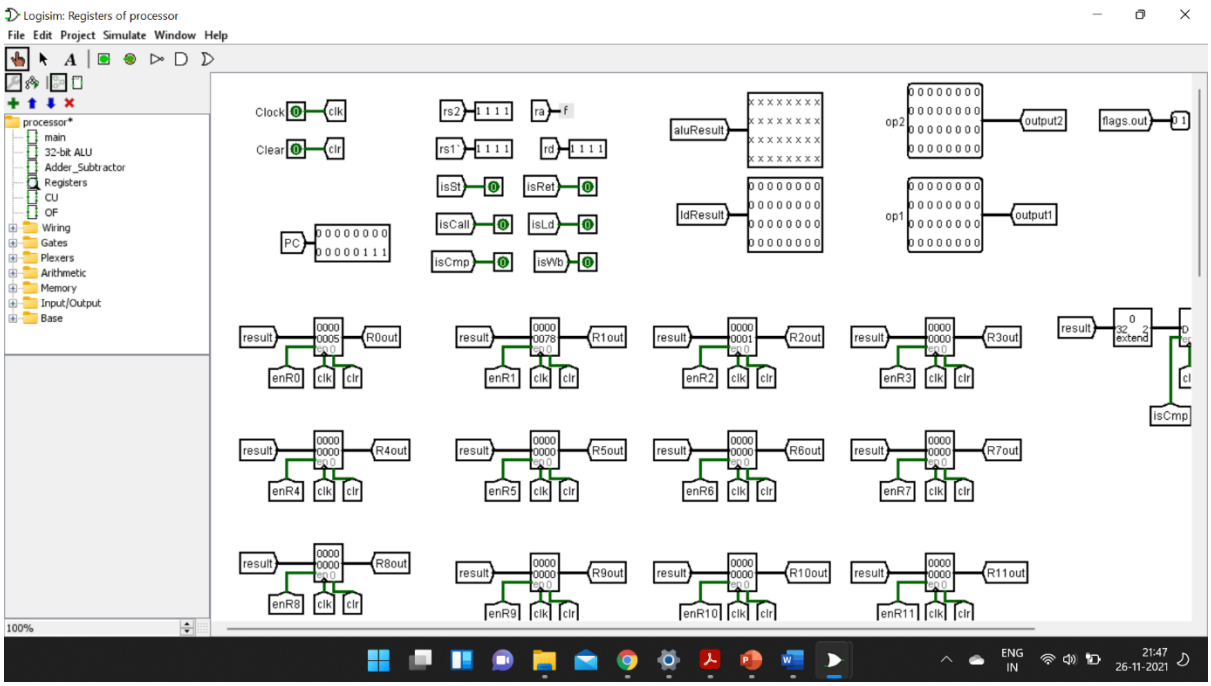
- These are loaded into the instruction memory as shown above and then executed.

- Initially we loaded the “r0”,”r2” registers with value “5” and “r1” register with value “1”.

- Register r0 contains input value i.e., 5. Register r1 is the required output i.e., we need to get factorial of 5 = 120 and register r2 is the index.

- Iteration takes place which decrements value of “r2” register (index) for every iteration of loop and loop stops executing once “r2” becomes “1”.

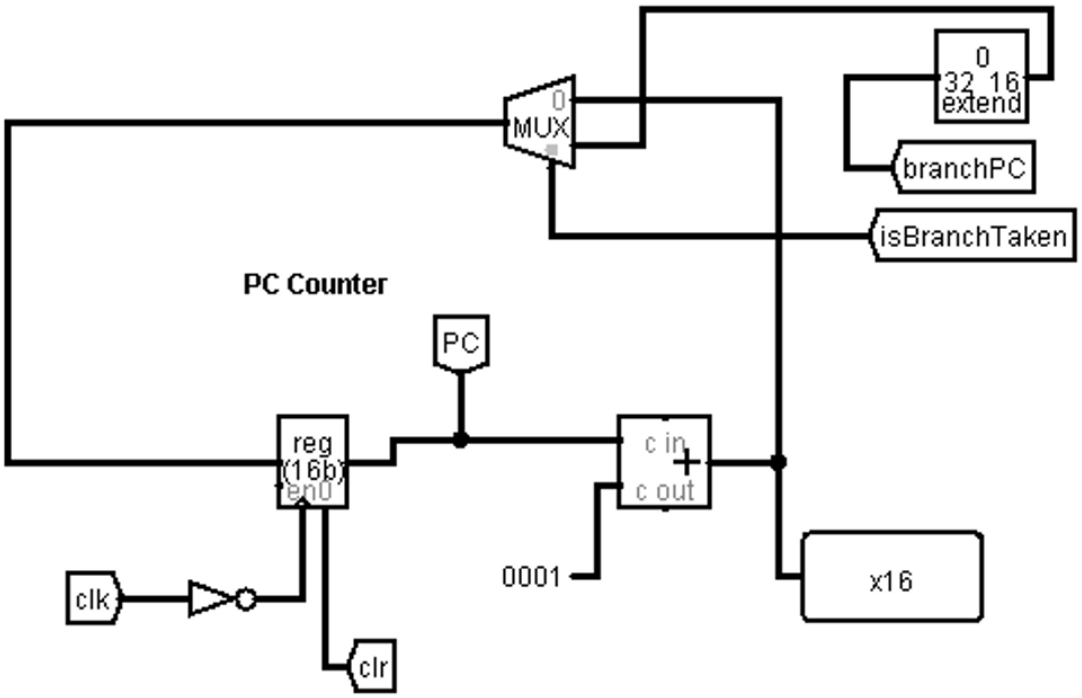
Output:



- Four iterations happened here where in each iteration r1 is multiplied with r2 value and stored in r1, r2 is reduced by 1, r2 compared with 1 and then check of loop condition.
- Here, as you can see the factorial of “5” is stored in “r1” register i.e., 00000078(in hexadecimal)->120(in decimal).

RESULTS AND DISCUSSION

➢ INSTRUCTION FETCH



- Instruction fetch unit fetches the instruction from the instruction memory which is currently stored in the PC and computes the address of the next instruction.
- At the end of the fetch operation, the PC points to the next operation which is to be read at the next cycle.
- The starting instruction ‘x16’ which is loaded from the instruction memory is sent to the multiplexer.
- The multiplexer has two inputs, one is the instruction ‘x16’ as mentioned above and the other is the branch target.
- This mux choses between them by using ‘isBranchTaken’ signal which we will get from control unit. That signal will be activated if there is any branch otherwise the mux choses the instruction from memory as shown in the above figure.

- The conditions for isBranchTaken signal are as follows:

Instruction	Value of $isBranchTaken$
non-branch instruction	0
call	1
ret	1
b	1
beq	branch taken – 1 branch not taken – 0
bgt	branch taken – 1 branch not taken – 0

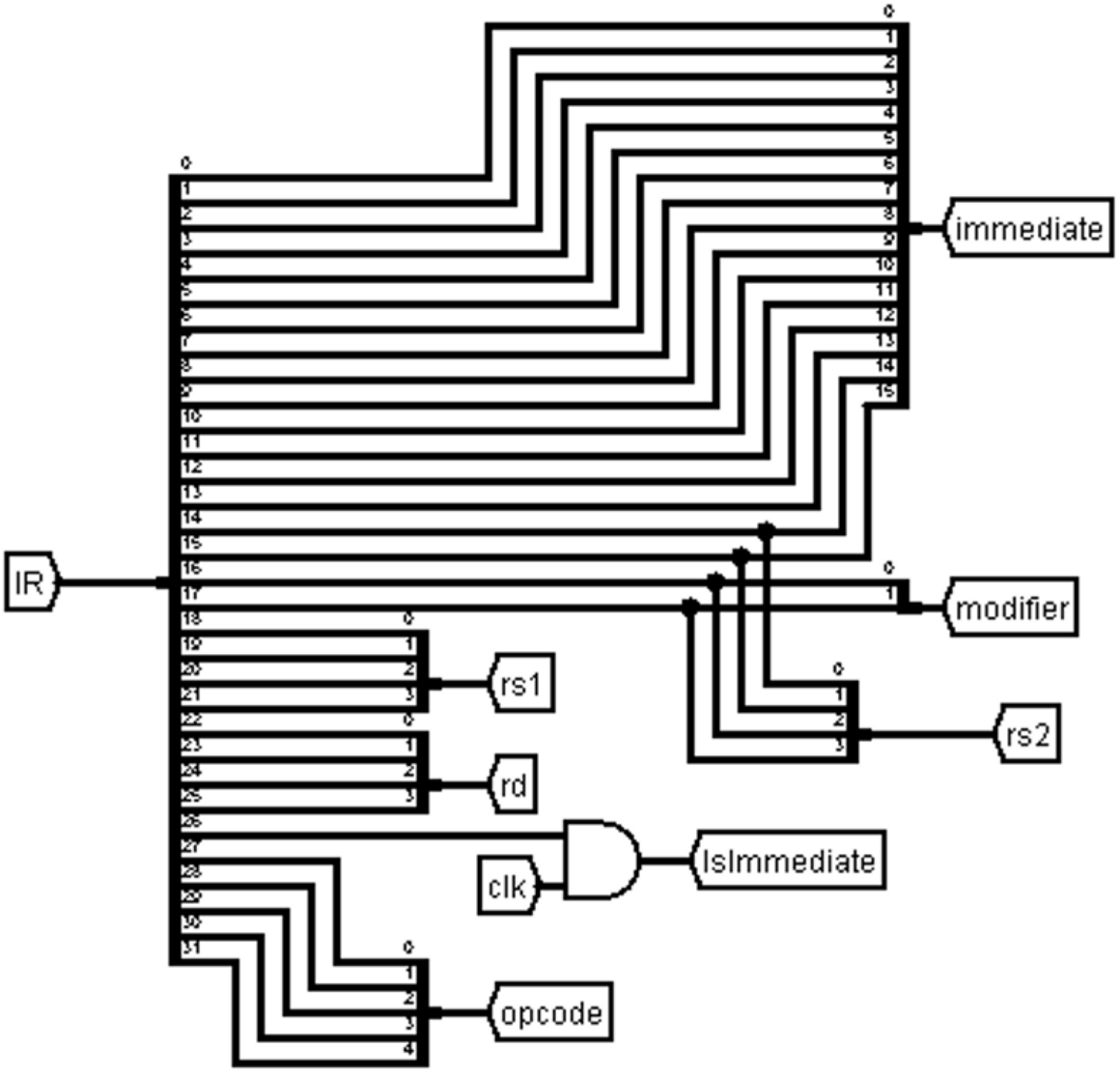
- The result from mux is passed to the pc register which contains the program counter (negative edge triggered) and is passed to the tunnel named 'PC' and 4 is added to this address for calculating address of next instruction (PC new = PC+4) and sent to the mux as shown in the figure. In Logisim we will be adding one instead of four because the memory gap between two instructions is already set as 4.
- In this way, for every clock edge the new instruction will be fetched, and the address of next instruction is computed.

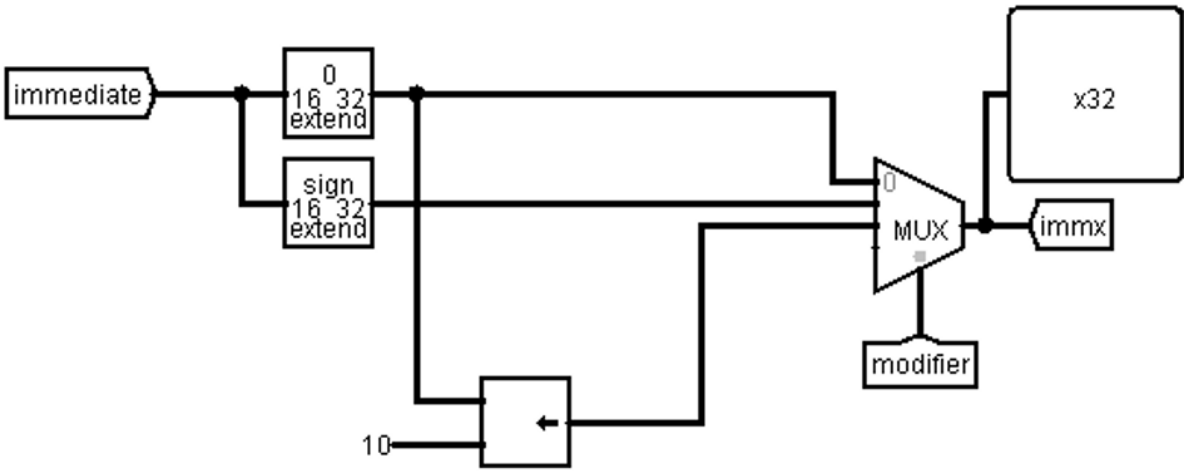
➢ OPERAND FETCH

The Operand Fetch unit has two important functionalities:

- **Calculating the value of the immediate operand and the branch target by encoding the offset embedded in the instruction.**
- **Reading the source registers.**

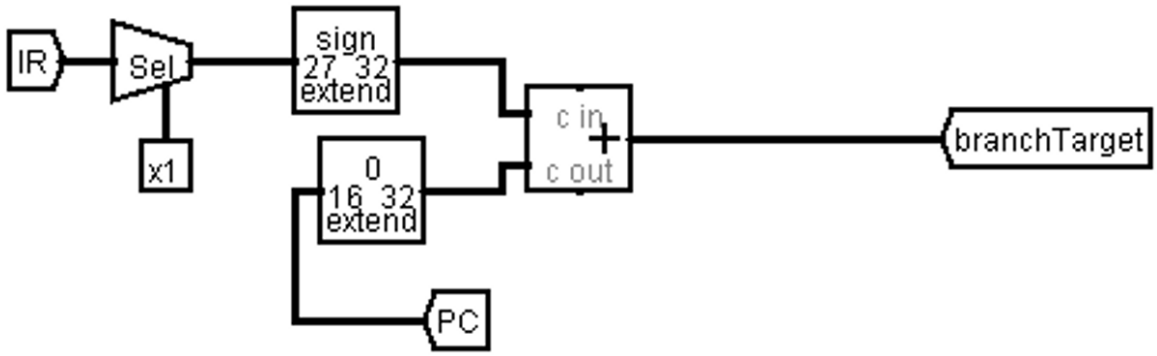
<u>Calculation of Immediate operand and the Branch Target:</u>
<u>IMMEDIATE: -</u>





- To calculate the Immediate operand, we have extracted the immediate bits(1-18) from the instruction using the splitter as shown in the first figure. Using that splitter we have not only seperated 18 bits , also we have seperated the modifier bits(17,18) and immediate bits(1-16).
- Now with the help of multiplexer, we have extracted 32 bit format of the immediate value i.e., if the modifier bits are 00 then we extend the sign of the 16 bit number to make it 32 bit number, if the modifier bits are 01 then we have filled the top 16 bits with 0, and if the modifier bits are 10 then we have shifted the 16 bit number, 16 positions to the left.
- The final 32 bit format of the immediate is represented using ‘immx’.

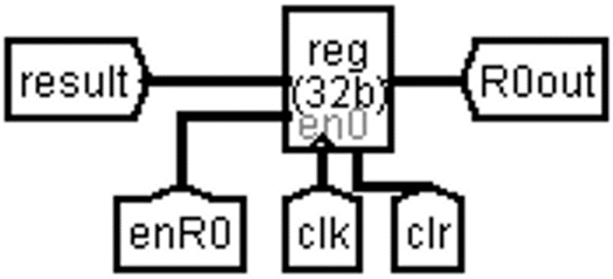
BRANCH TARGET:-



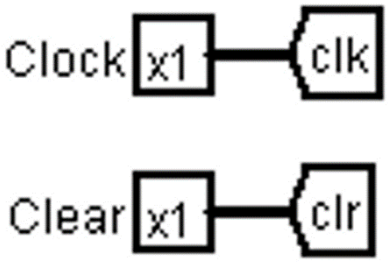
- For the calculation of branch target, first we need to extract the offset(1 to 27 bits) from the instruction and extend its sign to make it 32 bit number.
- We use PC-relative addressing mode in SimpleRISC, so we have to add PC to this 32 bit offset to get the branch target.
- This branch target does not work in the case of ret instruction as the branch target in that case is the contents of the ra registerwhich comes from the register file.

Reading the registers:

The register file contains 16 registers and two read ports and 1 write port. A port is a point of connection i.e., an interface, and is used for the purpose of either entering inputs, or reading outputs. We can have a read port (for reading data), a write port (for writing data).

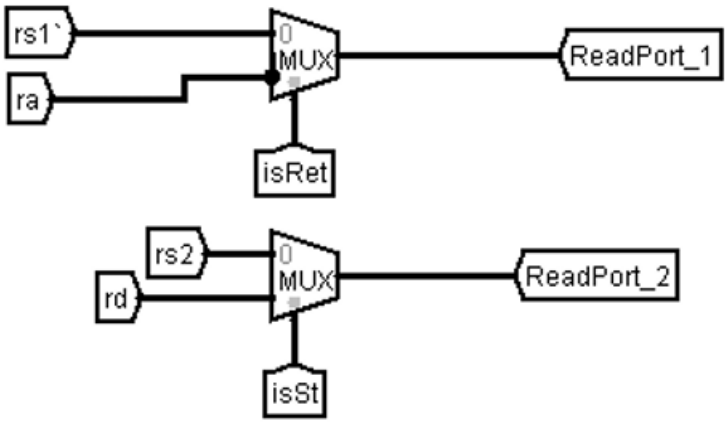


This is a basic register which is used to store information.



These symbols represent the clock and clear in the above register.

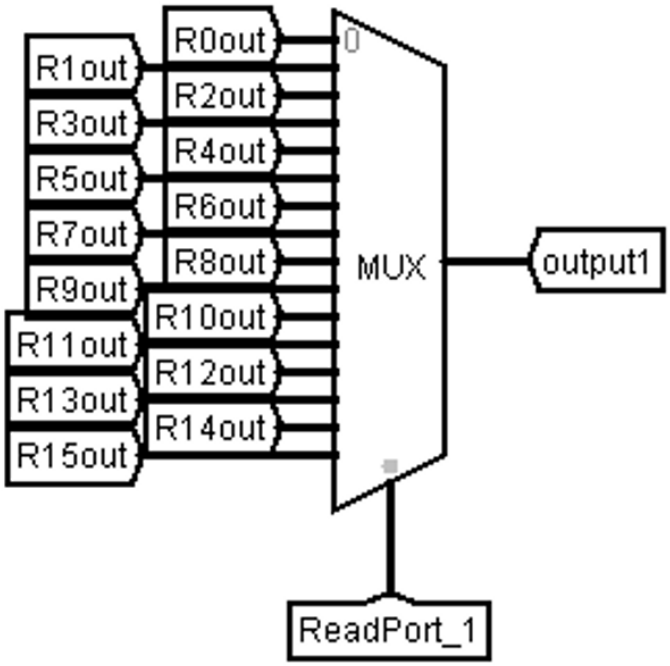
Registers are the fastest type of memory available; they are being used for the data to be operated on by the ALU.



In this, the first input is either rs1 or rs (ret instruction)

The second input is either rs2 or rd (store instruction)

And it gives the address as output (ReadPort_1 and ReadPort_2).



Here, the multiplexer takes address from the Read port and gives the output which is stored are stored in the registers.

The following table shows the list of instruction opcodes:

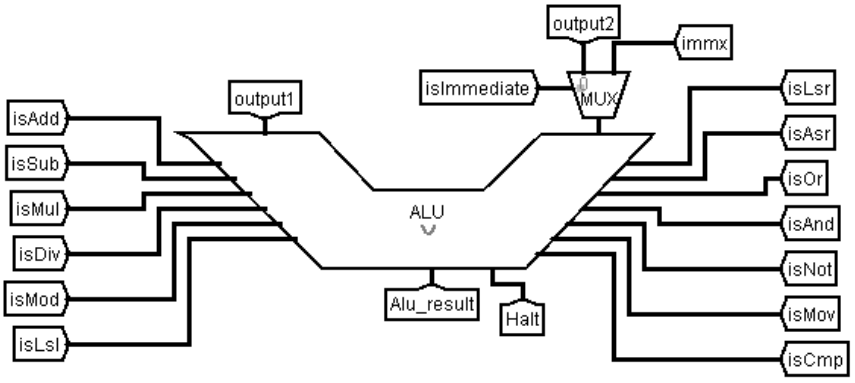
Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd, rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd, rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd, imm[rs1]
cmp	00101	cmp rs1, (rs2/imm)	st	01111	st rd, imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd, rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

And the following table shows the summary of the instruction formats

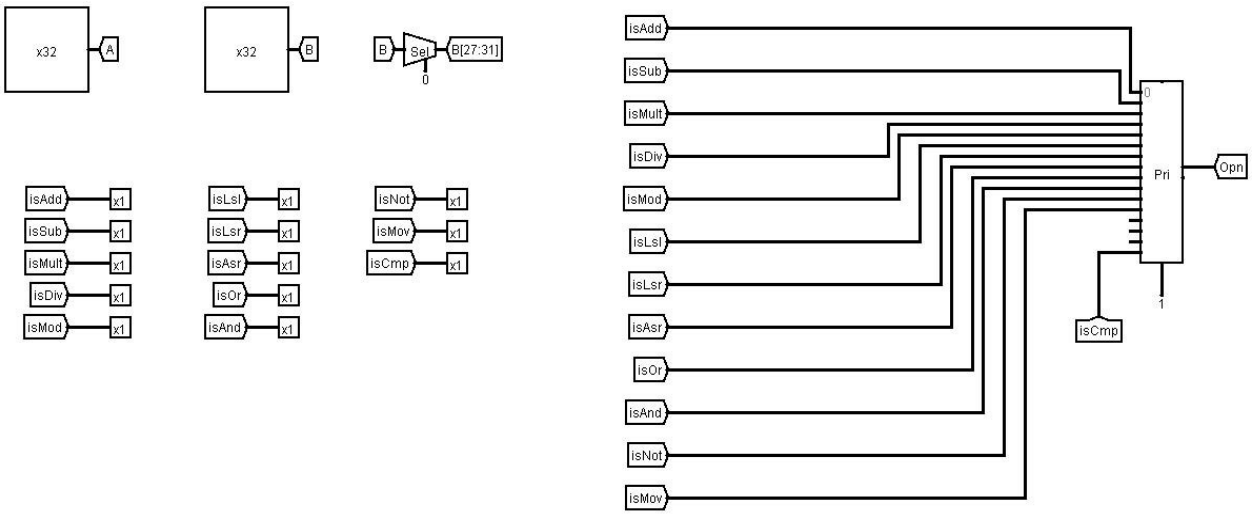
Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*rs*2 (15-18)
immediate	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
*rs*1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

➢ EXECUTION UNIT

Arithmetic Logic Unit (ALU)



- In above circuit we can see an input-output view of ALU(black-box view).
- Inputs being isAdd, isSub.....output1&output2/immediate.
- Output of ALU being Alu_result

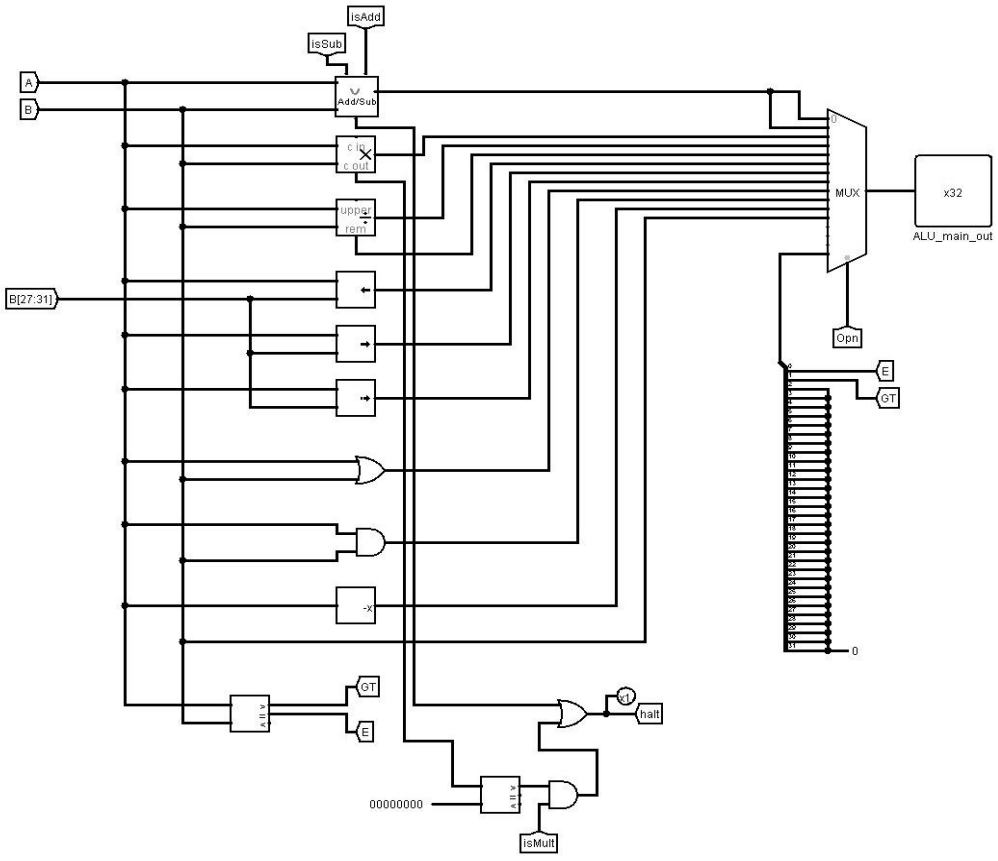


- Let's understand detailed design of ALU in 2parts.
- Here we can see 2 inputs A, B of ALU which are

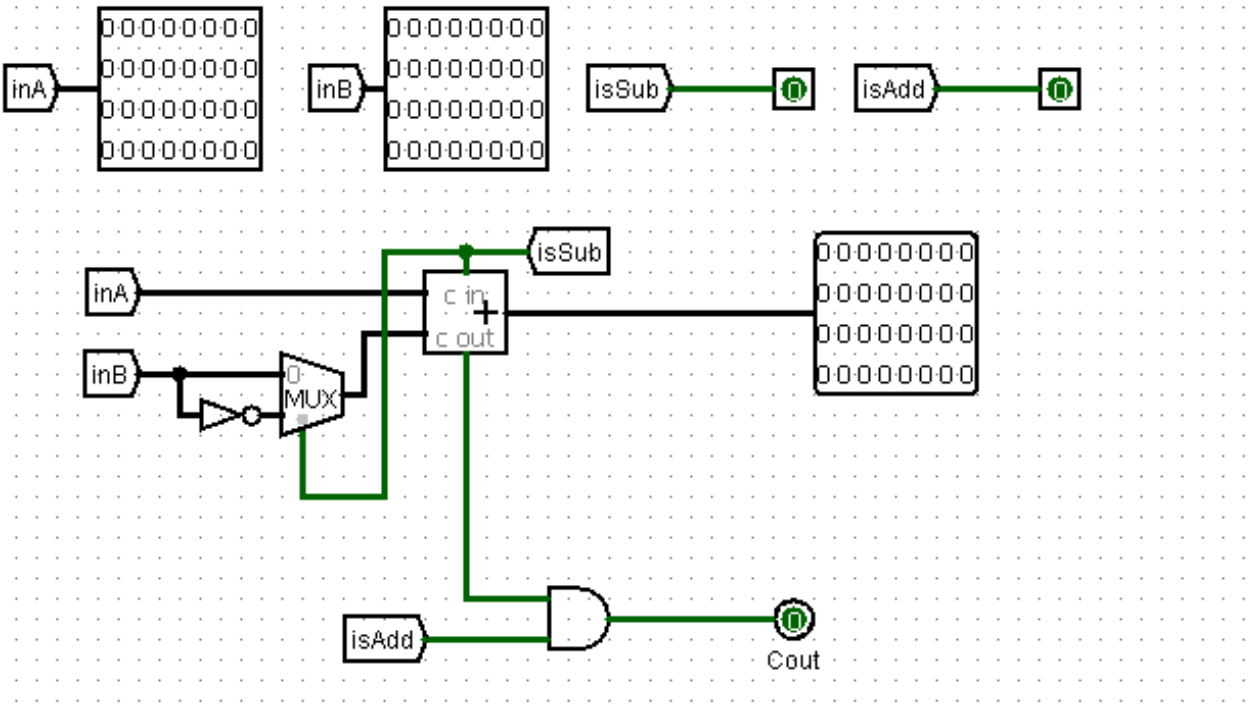
  A-rs1.

  B-rs2/imm.
- We also can see many 1bit signals like isAdd, isSub...

- IsAdd-to perform A+B
- IsSub-to perform A-B
- IsMult-to perform A*B
- IsDiv, isMod-to perform A/B, A%B
- IsOr, IsAnd-to perform A|B, A&B
- IsCmp- to compare A&B
- IsMov-to move B into destination register or here it simply does result = B
- IsNot-to negate B that is output is ~B
- IsLsl-to shift A by B units left.
- IsLsr, IsAsr-to shift A by B units right(unsigned, signed).
- We choose between these diff instructions(13) by encoding these into 4-bit opn.

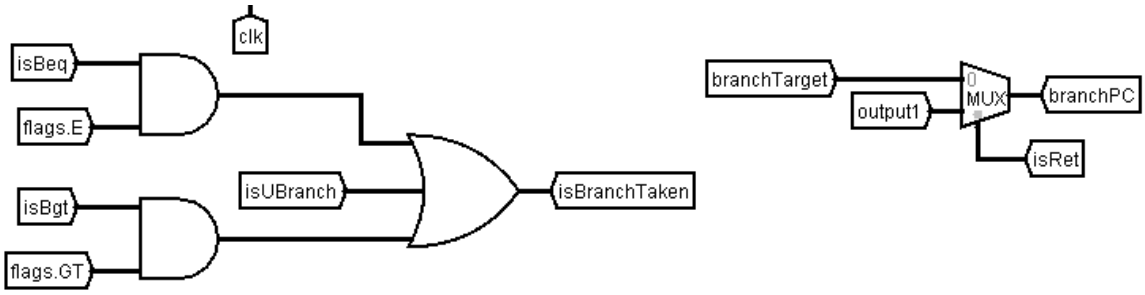


- We can see from with A&B as inputs we perform addition and subractrion using adder/subtracter and its design is



- As subtraction is also a 2's complement addition we complement 2 if its subtraction.
- Below adder/subtracter are multiplier and divider (outputs are A/B and A%B)
- Even below we can see lsl, lsr, asr operators.
- After these we can see and,or,not gates for their respective operations
- At bottom is comparator with 2 outputs being E, GT flags representing equality and greater than output.
- The results of all these operations are sent into a multiplexer and depending on value of Opn particular result is choosen as ALU_main_out that is as ALU result.

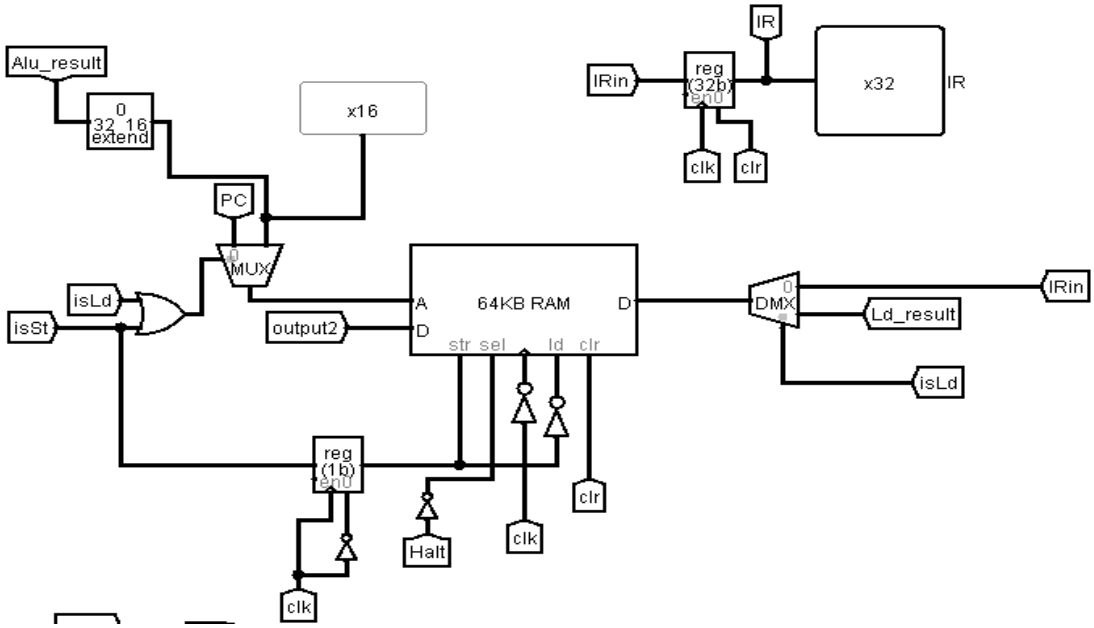
Branch signals



- IsBranchTaken is generated when isBeq/isBgt are 1 and required comparisons or isUBranch are met.
- BranchPC chooses branchTarget in case of ret instruction
- This branchPC is loaded into PC when isBranchTaken is 1.

➢ MEMORY ACCESS

RAM

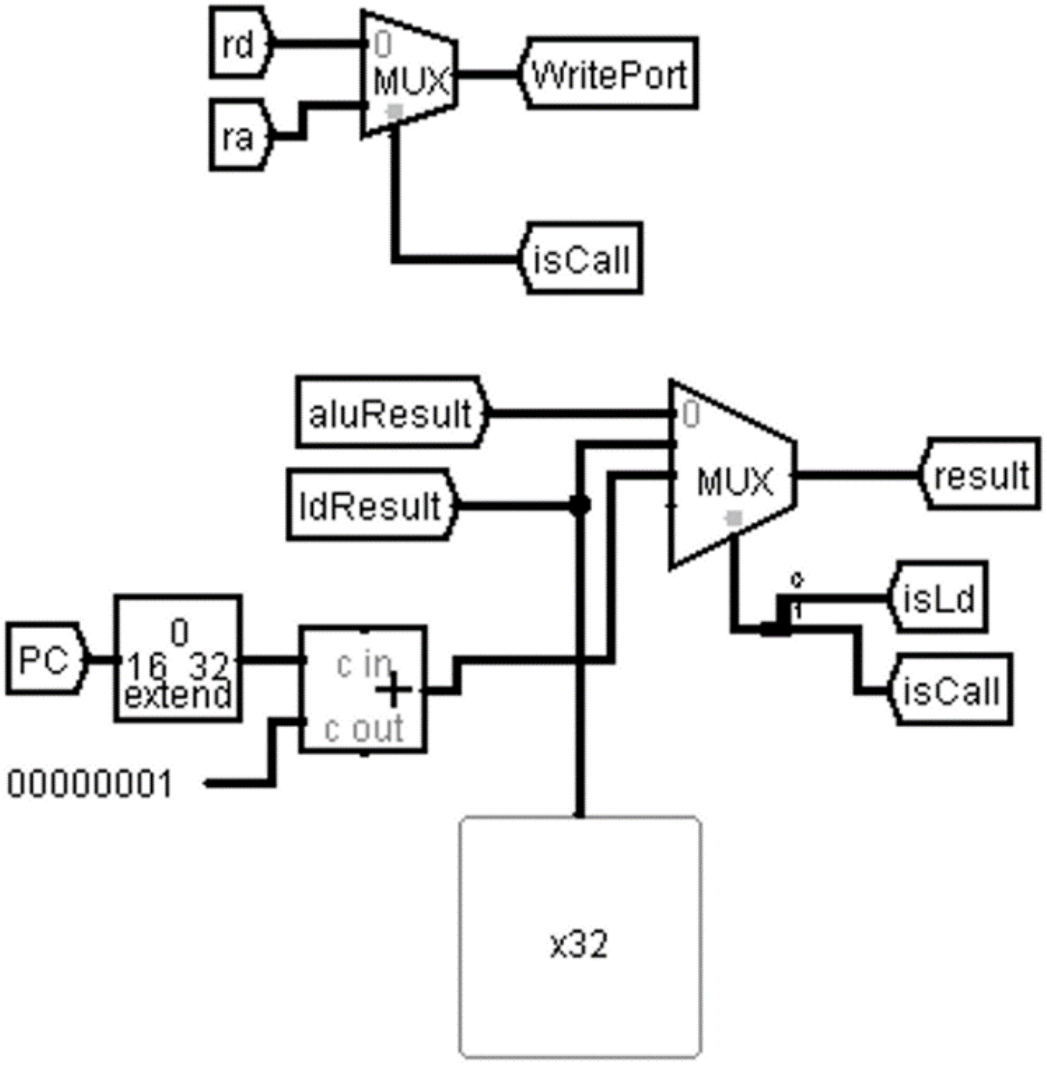


- This part of the circuit is memory (RAM).
- It is used to store instructions that have been loaded into it after the assembly code converts assembly language to hexadecimal instruction code.
- We fetch the instructions from this unit and load them into the Instruction register (IR).
- We also use this unit to store and load data.
- As you can see, there is a multiplexer which selects the Alu_result input instead of PC which is for instruction fetch, when there is a load or store instruction.
- Alu_result is basically the address into/from which the data is being stored/loaded.
- Output 2 is generally the data from the register (which has the data that we need to store in the memory).

- Halt if 1 deselects this unit which means the instructions are no more fetched and the processor is halted.
- And this unit requires basic necessary connections like clock and clear.

➢ REGISTER WRITEBACK UNIT

This is the last step of instruction processing. In this step, we will write back the calculated values back to the register file. That calculated value can be ALU result or output of load instruction or the return address in the case of call instruction. We will call this process as register writeback.

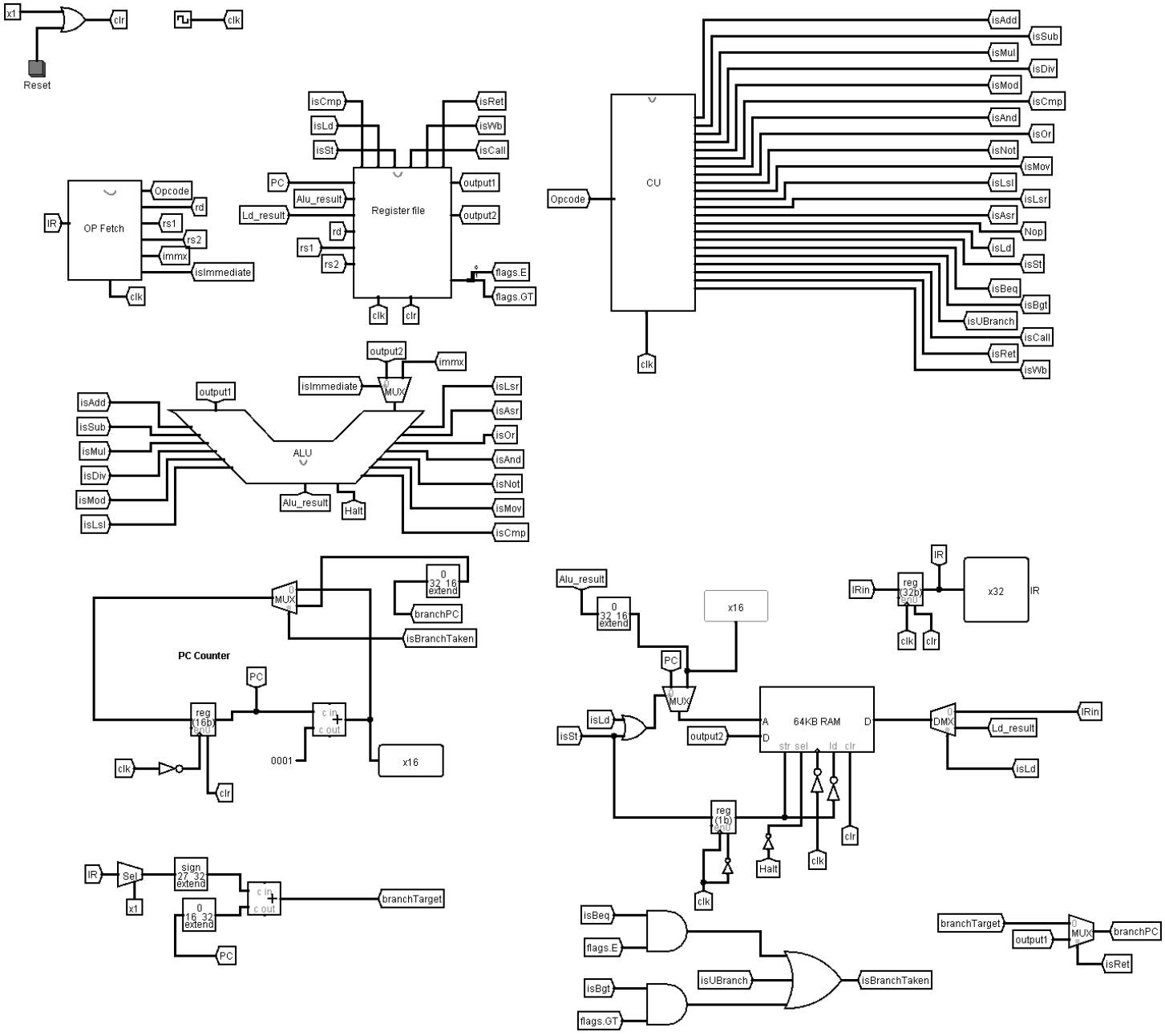


We will choose the right source operand for the write port using the multiplexer. Here we use the control signals isLd and isCall for controlling the multiplexer. We have three inputs for the multiplexer i.e., aluResult, ldResult and return address as shown in the figure. When both isLd and isCall are 0 then aluResult will be sent, if isLd is 1 and isCall is 0 then ldResult will be the output and if isLd is 0 and isCall is 1 then the return address is sent to the writeport.

- The register file contain 2 read ports and 1 writeport, the read ports are already shown in the operand fetch unit.
- The writeport contains three inputs i.e., enable (isWriteback-isWb), address (using the first multiplexer shown in the figure) and data from the multiplexer as discussed above.

➢ THE DATA PATH

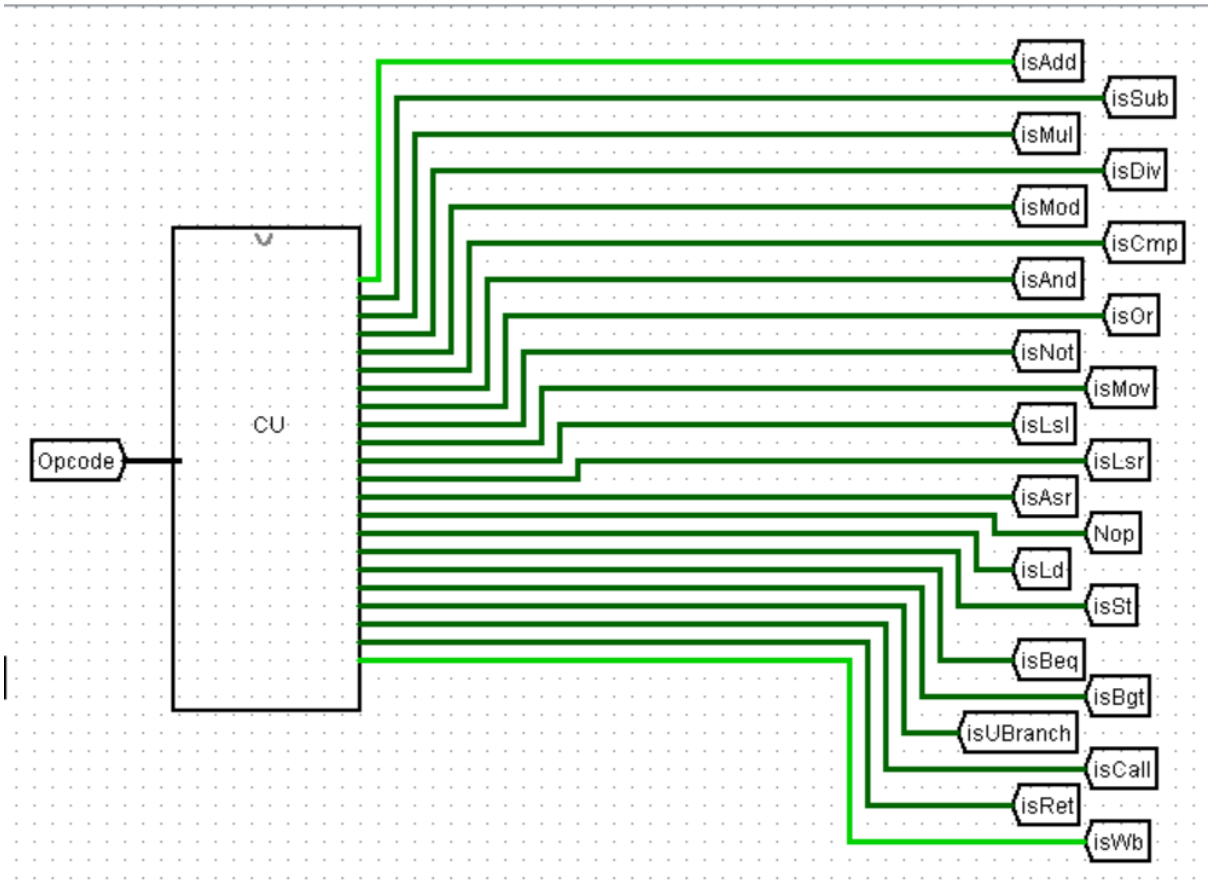
Till now we have divided the processor into 5 units, now it's time to form the whole by joining all the parts. For every clock cycle, the processor fetches an instruction from a new PC fetches the operands, executes the instruction, and writes the results back to the data memory and register file. The main state elements of the processor are these registers, pc, flag registers and the register file.



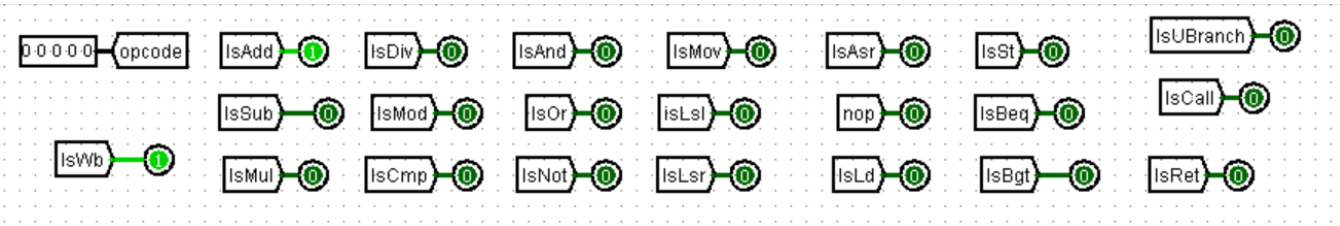
➢ THE CONTROL UNIT

The control unit (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor.

It takes in the “Opcode” as input and gives out control signals which are further used in different stages like register write/read, execution and operand fetch unit.

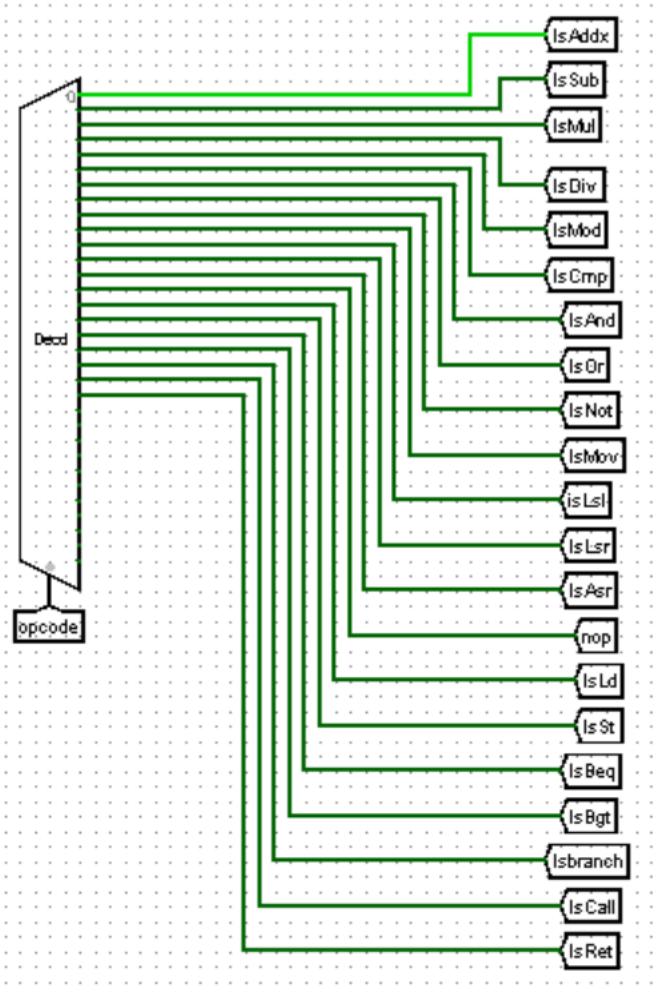


Inside of Control Unit Sub-Circuit



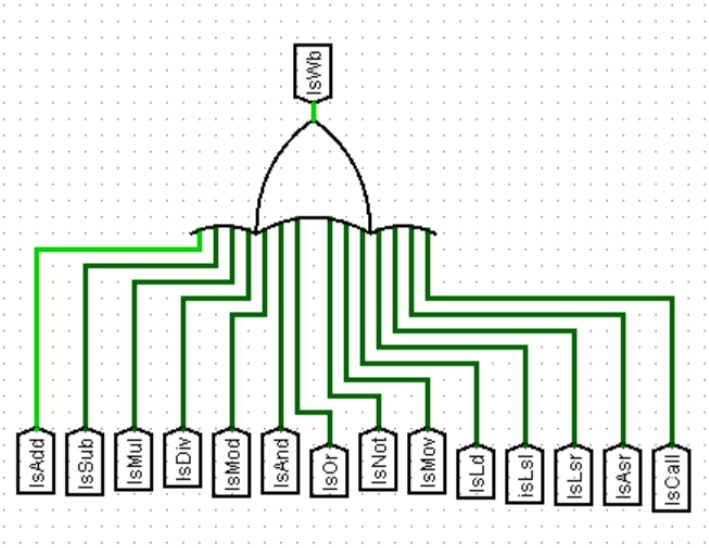
Here the opcode is 5-bit input signal and remaining are 22 control signals which are 1-bit each.

- The value of IsAdd becomes 1 when the instruction type is add, ld or store
- The value of IsSub becomes 1 when the instruction type is sub.
- The value of IsMul becomes 1 when the instruction type is mul.
- The value of IsDiv becomes 1 when the instruction type is div.
- The value of IsMod becomes 1 when the instruction type is mod.
- The value of IsCmp becomes 1 when the instruction type is cmp.
- The value of IsAnd becomes 1 when the instruction type is and.
- The value of IsOr becomes 1 when the instruction type is or.
- The value of IsNot becomes 1 when the instruction type is not.
- The value of IsMov becomes 1 when the instruction type is mov.
- The value of IsLsl becomes 1 when the instruction type is lsl.
- The value of IsLsr becomes 1 when the instruction type is lsr.
- The value of IsAsr becomes 1 when the instruction type is asr.
- The value of nop becomes 1 when the instruction type is nop.
- The value of IsLd becomes 1 when the instruction type is ld.
- The value of IsSt becomes 1 when the instruction type is st.
- The value of IsBeq becomes 1 when the instruction type is beq.
- The value of IsBgt becomes 1 when the instruction type is bgt.
- The value of IsUBranch becomes 1 when the instruction type is b, call, ret.
- The value of IsCall becomes 1 when the instruction type is call.
- The value of IsRet becomes 1 when the instruction type is ret.
- The value of IsWb becomes 1 when the instruction type is add, sub, mul, div, mod, and, or, not, mov, ld, lsl, lsr, asr, call.

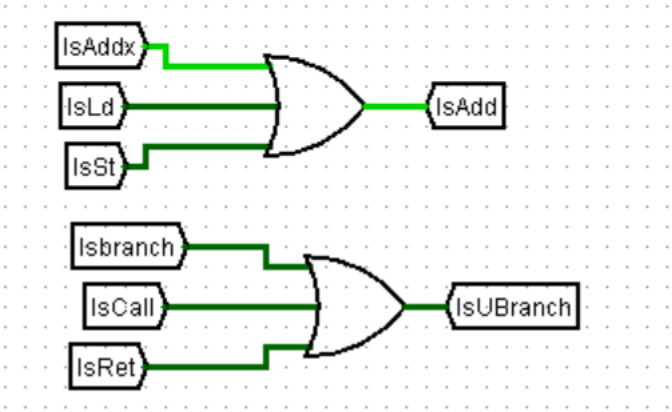


Opcode passed as select for the above decoder which performs the action of identifying which output should 1. If the opcode is 0, the value of signal IsAddx is made 1. If it is 1, then the value of signal IsSub is assigned 1. Similarly the remaining signals are also assigned based on the value of opcode.

IsWb signal is activated, whenever one of the IsAdd, IsSub, IsMul, IsDiv, IsMod IsAnd, IsOr, IsNot, IsMov, IsLd, IsLsl, IsLsr, IsAsr, IsCall is activated. So to achieve this, we have used a ‘nor’ gate to implement this whose inputs are the above mentioned signals and output is IsWb.



Similarly, IsAdd is activated whenever one of the IsAddx, IsLd, IsSt is activated. And signal IsUBranch is activated whenever one of the Isbranch, IsCall , IsRet is 1. We have again used nor gates to implement this logic.



CONCLUSION

We have made the SimpleRisc and simulated it on the logisim.We have designed various stages of the processor like Instruction Fetch, Operand Fetch, Execute, Memory, Register Write and also integrated them with the assembly code. We have also developed a circuit which executes an instruction in 5 stages in 5 clock cycles. This is basis for future extension of this the project i.e multicore processing.The whole project is simulated and is also checked upon test cases like factorial of a number etc and thus was made sure that there are no errors i.e hazards are minimized. There are certain areas like multicore processing in which the project can be extended for future purposes.

Bibliography :-

- Basic Computer Architecture Version 2.1 - Smruti R. Sarangi.
- http://www.cburch.com/logisim/docs/2.7/en/html/guide/tutorial/index.html - Logism Tutorial.
- https://www.researchgate.net/publication/281100703 - Reference for multicore environment.

GITHUB LINK:

https://github.com/NikhileshBhagavan/32bit-SimpleRisc-Processor