Designing SimpleRISC processor on Logisim simulator

CSN-221 Group Project

TEAM MEMBERS

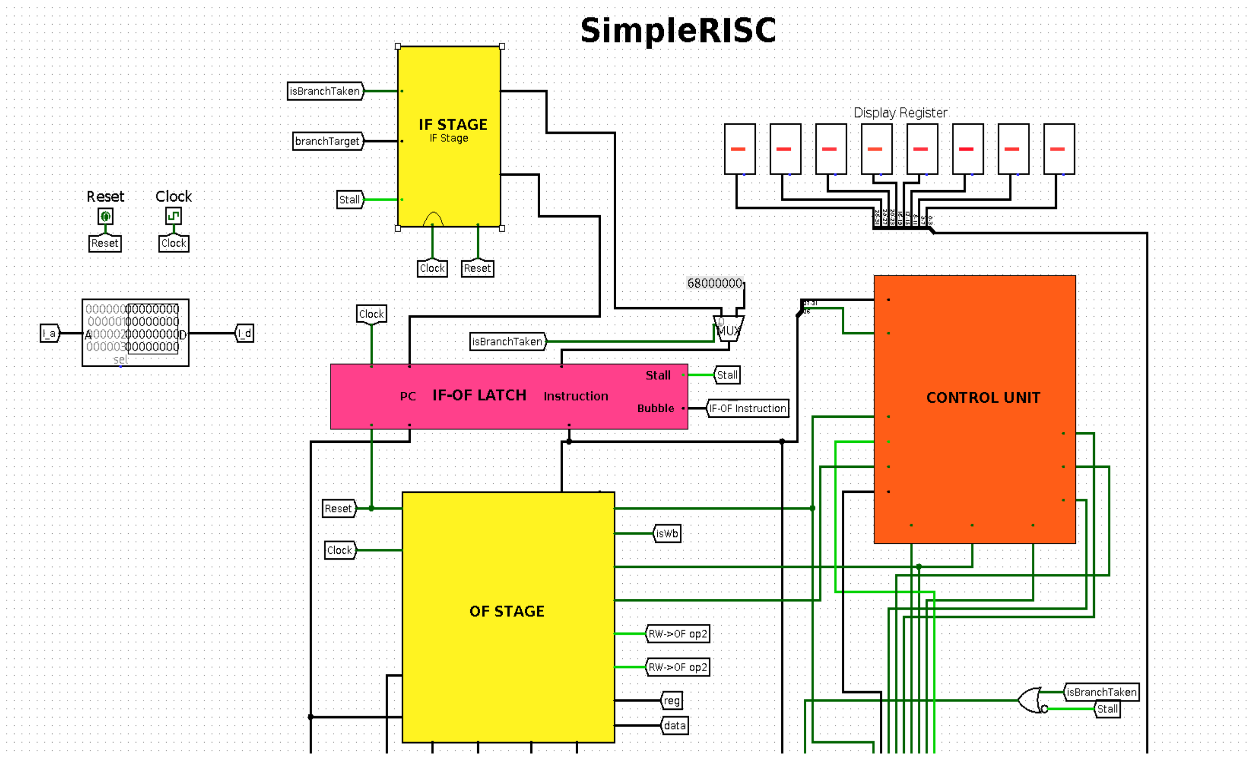
JAYATI SHRIVASTAVA	**18121012**
THARUNISWER R	**20114100**
HARSH KESARWANI	**18113052**
PRABHAV ROHILLA	**20114072**
JONTI KUMAWAT	**18115047**
TANYA GARG	**18122027**
LAXIT LALL	**20114049**
DEEPANSHU	**20114030**

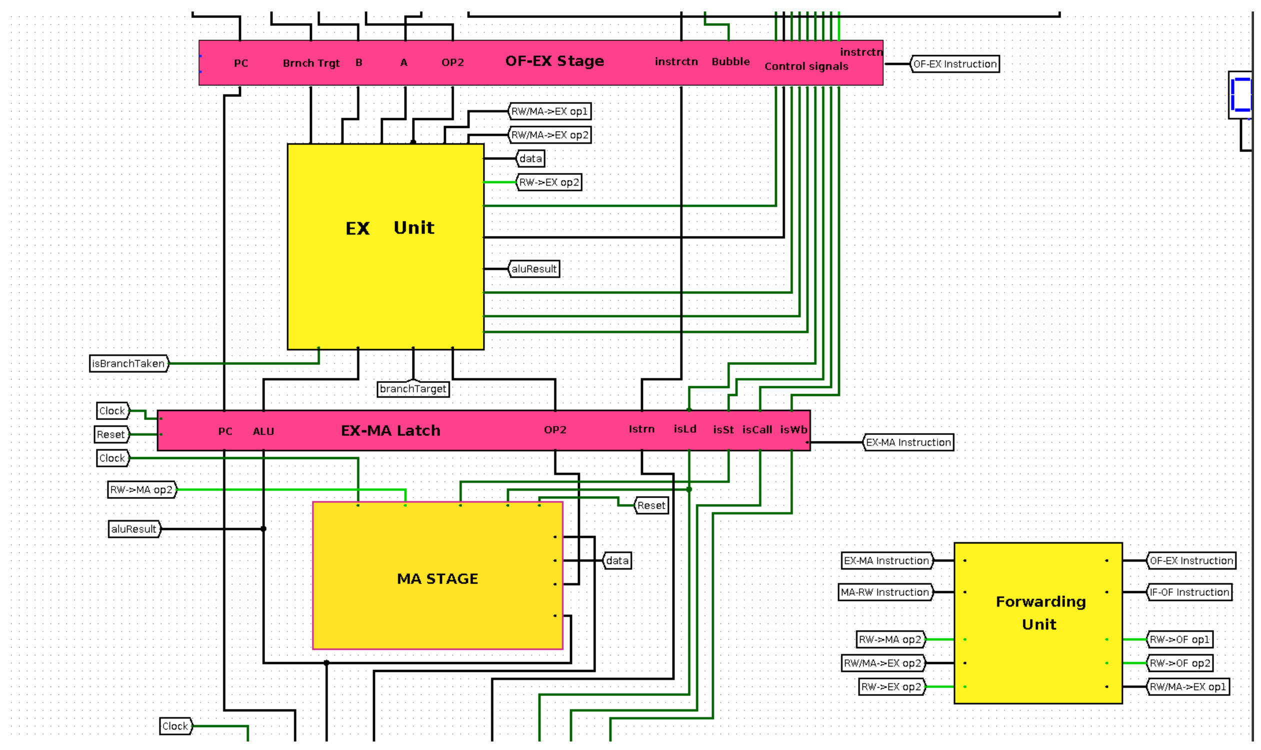
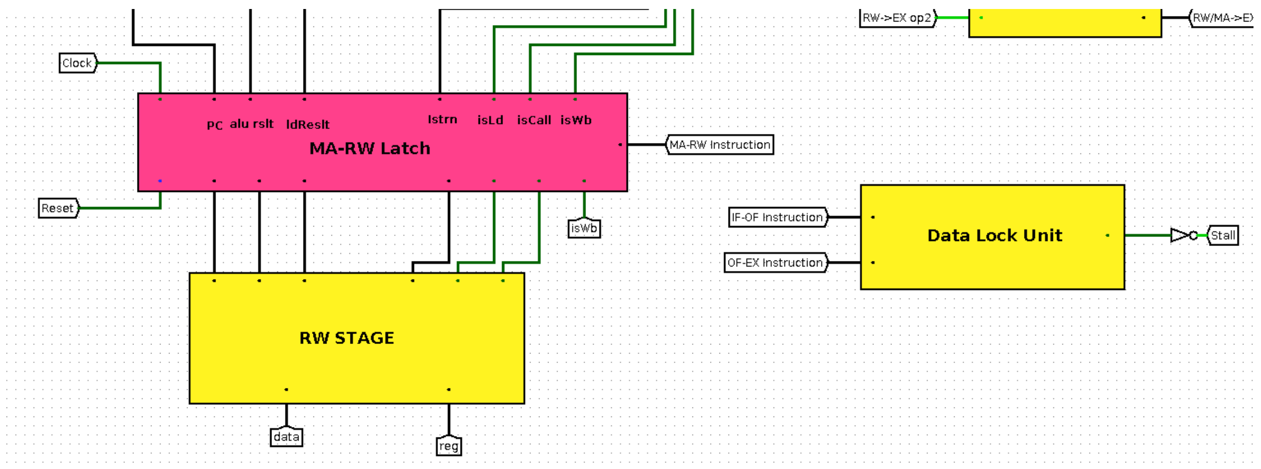
Table of Contents

1. Introduction

RISC stands for ‘Reduced instruction set computer’. It consists of a small number of instructions that have a simple and regular structure. Examples include ARM, MIPS, Alpha etc. SimpleRISC is a simple and concise assembly language, which has most of the features of full-scale assembly languages. It has 16 registers numbered r0 to r15 and contains only 21 instructions.

Here, we present   The code for the project can be found in this repo: https://github.com/THARUNISWER/32-bit_Simple_risc




PC
Brnch Trgt
B
A
OP2
OF-EX Stage
instrctn
Bubble
Control signals
instrctn
OF-EX Instruction
RW/MA->EX op1
RW/MA->EX op2
data
RW->EX op2
EX Unit
aluResult
isBranchTaken
branchTarget
Clock
Reset
Clock
PC
ALU
EX-MA Latch
OP2
Istrn
isLd
isSt
isCall
isWb
EX-MA Instruction
RW->MA op2
Reset
aluResult
MA STAGE
data
EX-MA Instruction
MA-RW Instruction
RW->MA op2
RW/MA->EX op2
RW->EX op2
Forwarding Unit
OF-EX Instruction
IF-OF Instruction
RW->OF op1
RW->OF op2
RW/MA->EX op1
Clock

RW->EX op2
RW/MA->EX
Clock
PC
alu rslt
ldReslt
Istrn
isLd
isCall
isWb
MA-RW Latch
MA-RW Instruction
Reset
isWb
IF-OF Instruction
OF-EX Instruction
Data Lock Unit
Stall
RW STAGE
data
reg

2. Problem Statement

Design the SimpleRISC processor discussed in the class on Logisim simulator. Extend the basic ideas discussed in the class with several other ideas that you think are important to be included. Evaluate rigorously. BENCHMARK evaluations are preferred.

3. Novelty of the work done

We have pipelined the simpleRISC processor so as to reduce the time taken to execute instructions. While implementing the pipeline we have also taken time to deal with hazards by designing a conflict detection unit, data lock unit and a forwarding unit that improves the efficiency of the processor as far as possible.

We also took time to design an assembler to convert our simpleRISC codes into machine codes in hex, so as to make our processor convenient to use and also make evaluation easier.

We have also evaluated rigorously using a minimum of about 14 benchmarks to test the various instructions and the performance of the processor.

To evaluate the benchmarks we have added a small extension to the processor (a separate logisim processor part in github repo for evaluation purposes) that can count the number of clock cycles it takes to execute the set of instructions we feed as input to the processor.

These are the novelties in our processor.

4. Individual Contributions

Jayati Shrivastava: Designed the high level architecture of the processor and its specifications. I then designed and implemented the Control Unit, the ALU and the Execute Stage. To realise the 5-stage pipeline, I designed and added the 4 buffer latches between successive stages. I also integrated the remaining components of the processor by adding the interconnections between the modules.

Tharuniswer R: Created benchmarks to test the performance, created an assembler to convert Simple RISC code to hex(machine language) and ran tests on processor to calculate speed-up of processor and tested if instructions are working.

Harsh Kesarwani: Designed and implemented the Memory Access (MA) Stage and the Register Writeback (RW) Stage, which are the last two stages of instruction processing. Also helped in fixing the errors during the integration of the stages.

Prabhav Rohilla: Designed and implemented the Instruction Fetch Stage of the processor. Helped in the testing of the processor and removed the errors that occurred while designing or interconnecting the various sub-circuits.

Jonti Kumawat: Designed and implemented data block unit of the processor. This unit was responsible for stalling the pipeline when load-use hazard conditions arose. It ensures the correctness of program execution.

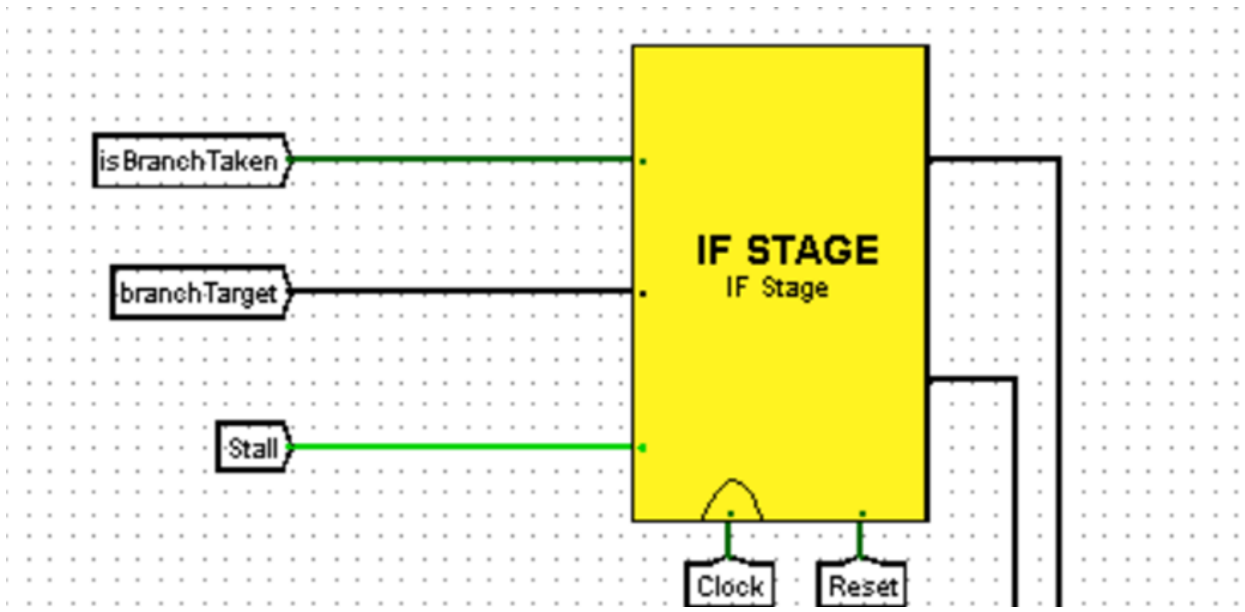
Tanya Garg: Wrote SimpleRisc codes to evaluate and benchmark the processor. These codes were converted to machine language and subsequently run on the processor to obtain the time taken by a pipeline processor. Further, calculated the speed-up for the benchmarking codes and helped in identifying and solving errors that arose during the execution of the codes.

Laxit lall: Designed and implemented forwarding unit of the processor.

Deepanshu: Designed and implemented the operand fetch unit and also helped in resolving errors in the interconnections of the circuit.

5. Methodology

<u>Instruction Fetch Unit</u>



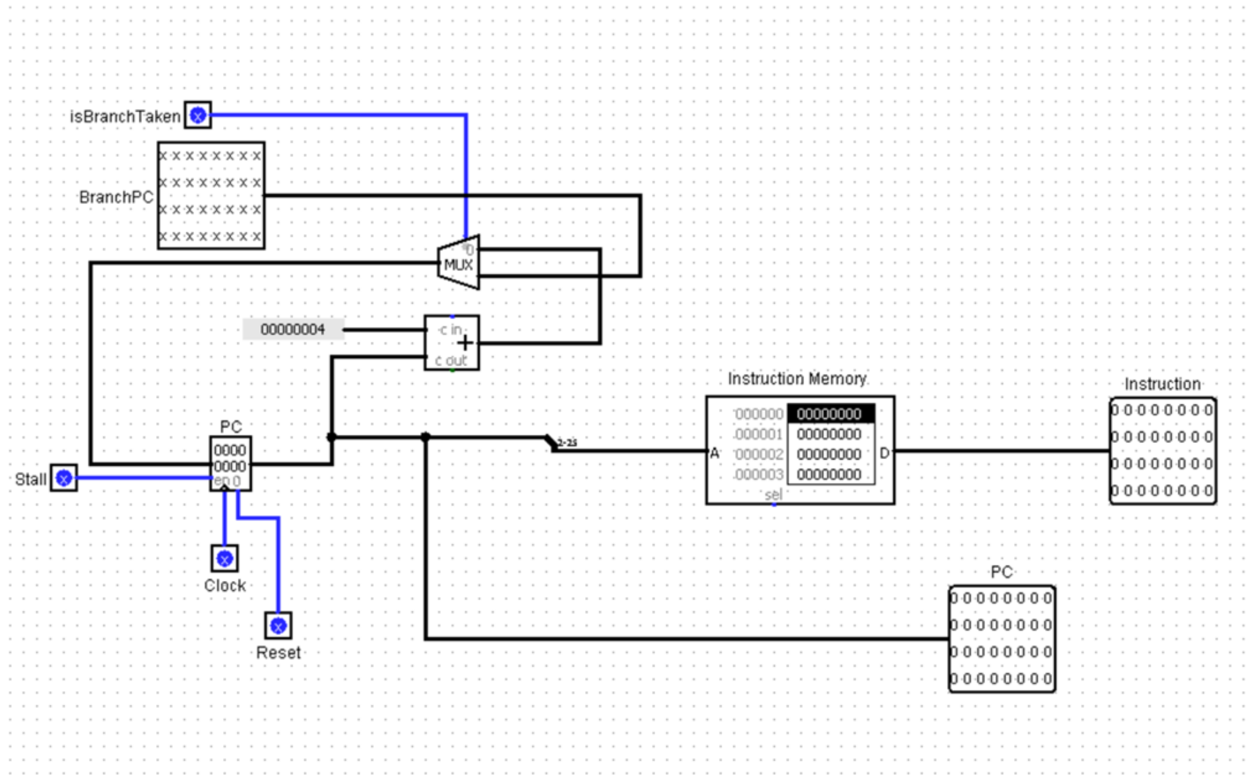
- This is the first stage in our processor where we fetch instruction from the memory.
- To access the memory and fetch the instruction we need to store the address of the location in a register, which we call the program counter(pc).
- After the instruction is fetched and proceeded to decoding, we need a mechanism to modify the PC so that it points to the next instruction. To point to the next instruction there can be two cases:-

    i. If the current instruction is a branch instruction which is taken, then we need to update the PC to make it equal to the address of the target branch.

    ii. Otherwise, we need to increment the PC by 4 because in our processor our instructions are 4 bytes long therefore the next instruction would be stored at the address (old PC+4).

- We are developing a pipelined processor thus we need a mechanism to deal with the stalls. So, we also maintain a stall bit to achieve this.

The circuit designed to achieve these functionalities is shown below:-



The external inputs required in the above circuit are listed below: -

1. isBranchTaken – This signal is used to determine whether the branch is taken or not. It is generated by the Branch unit in the EX Stage.

isBranchTaken signal is set to 1 on encountering call, b, ret, beq(when operands are equal) and bgt(when first operand greater than the second) and otherwise it is 0.

2. BranchPC – This is the target branch which is to be taken if isBranchTaken=1. This signal is generated by the Execution unit.

3. Stall – This signal is used to determine whether there is a stall or not. It is generated in the Data-Lock unit.

4. Clock – To keep the circuit synchronized.

5. Reset – In order to reset the whole program.

The following components are used while designing the circuit: -

- A register to store the PC. We connect Stall signal at the enable bit of the register. When Stall = 1, the clock edges are ignored as we have encountered a stall.

- A multiplexer is used to determine the next PC with isBranchTaken as the select bit, if isBranchTaken=1 then PC is BranchPC otherwise it is old PC+4.
- An adder to determine PC+4.
- A ROM which is our Instruction Memory. Having the address of size 24 bits, we fan out the PC from bit 2 to 25 which contains the address of the instruction.

We take two outputs from this circuit: -

i. PC – We use it in further stages.

ii. Instruction – We process it to the next stage for decoding.

The output of this stage is then sent to the IF-OF Latch.

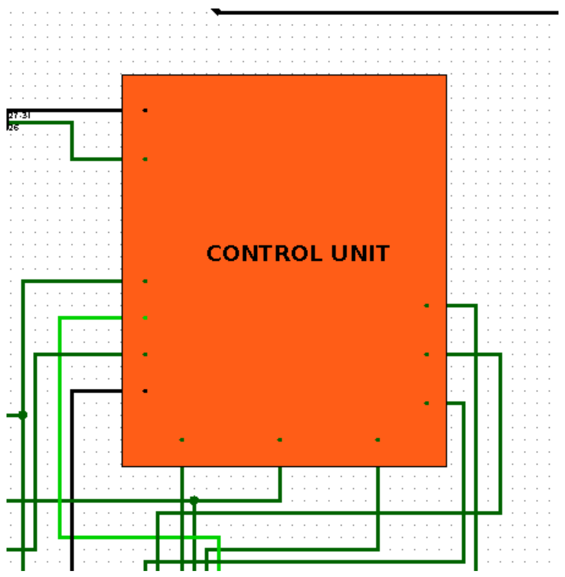
OPCODES

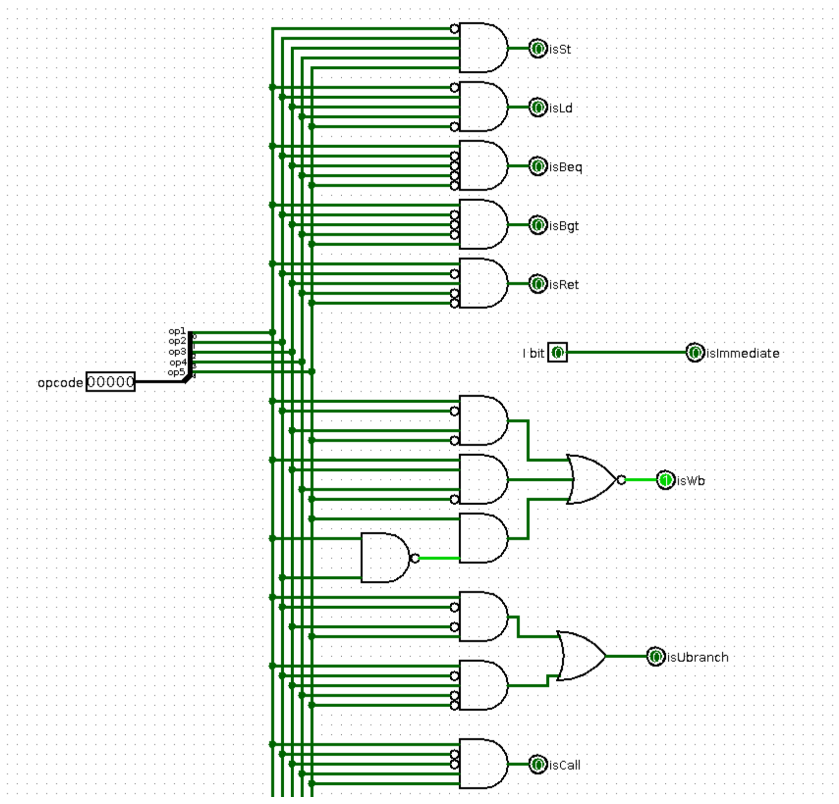
add	00000
sub	00001
mul	00010
div	00011
mod	00100
cmp	00101
and	00110
or	01111
not	01000
mov	01001
lsl	01010
lsr	01011
asr	01100

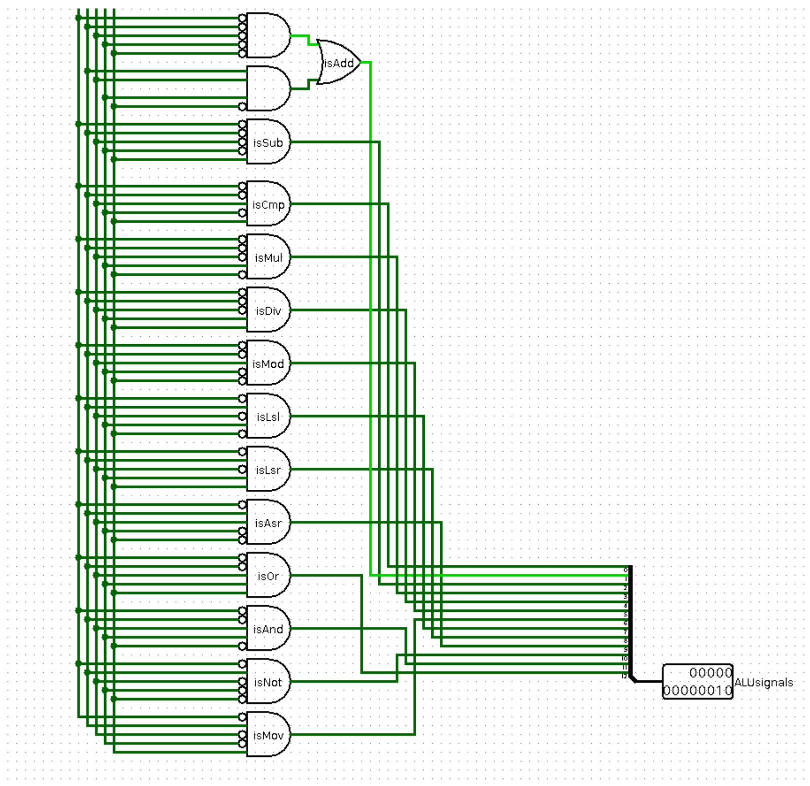
nop	01101
ld	01110
st	01111
beq	10000
bgt	10001
b	10010
call	10011
ret	10100

CONTROL UNIT

The processor uses a hardwired Control Unit which takes the 5-bit opcode from the fetched instruction as an input. On every rising edge of the clock, the control unit decodes this opcode to generate the correct control and alu signals for that instruction. Our Control Unit is a straightforward hardwire mapping of the opcodes and signals given in the following tables.


27-31
26
CONTROL UNIT


isSt
isLd
isBeq
isBgt
isRet
I bit
isImmediate
op1
op2
op3
op4
op5
opcode
00000
isWb
isUbranch
isCall



Control Signals

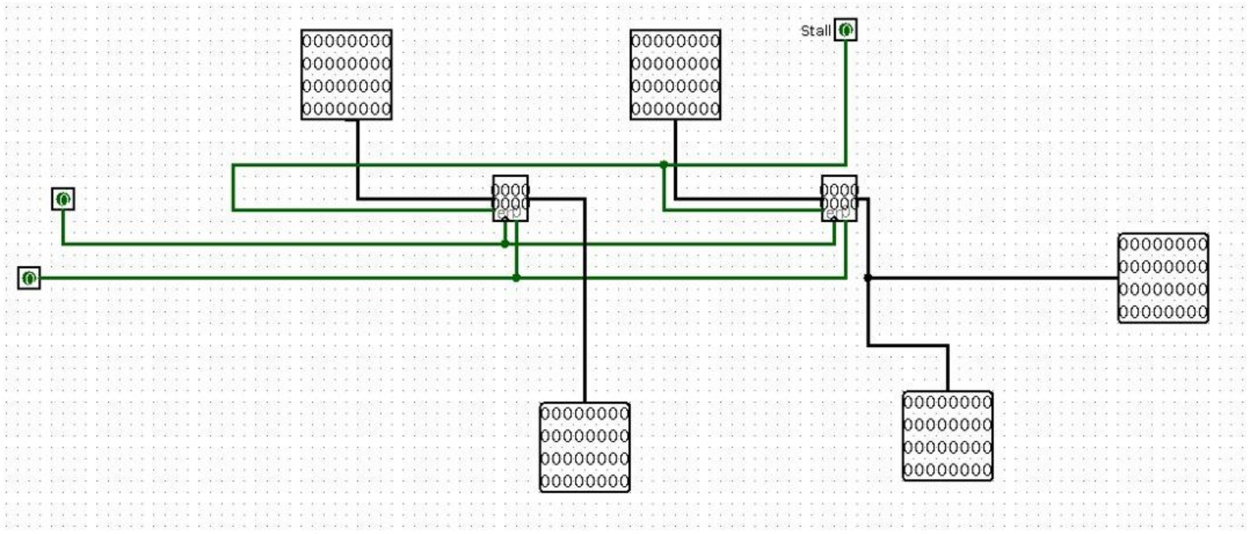
Serial No.	Signal	Condition
1	isSt	$\overline{op5}\,.op4.op3\,.op2\,.op1$
2	isLd	$\overline{op5}\,.op4\,.op3\,.op2\,.\overline{op1}$
3	isBeq	$op5\,.\overline{op4}\,.\overline{op3}\,.\overline{op2}\,.\overline{op1}$
4	isBgt	$op5\,.\overline{op4}\,.\overline{op3}\,.\overline{op2}\,.op1$
5	isRet	$op5\,.\overline{op4}\,.op3\,.\overline{op2}\,.\overline{op1}$
6	isImmediate	I
7	isWb	$\sim(op5\ +\ \overline{op5}\,.op3\,.op1.(op4+\overline{op2}\,))+op5\,.\overline{op4}\,.\overline{op3}\,.op2\,.op1$
8	isUbranch	$op5\,.\overline{op4}\,.(\overline{op3}\,.op2.+op3\,.\overline{op2}\,.\overline{op1}\,)$
9	isCall	$op5\,.\overline{op4}\,.\overline{op3}\,.op2\,.op1$

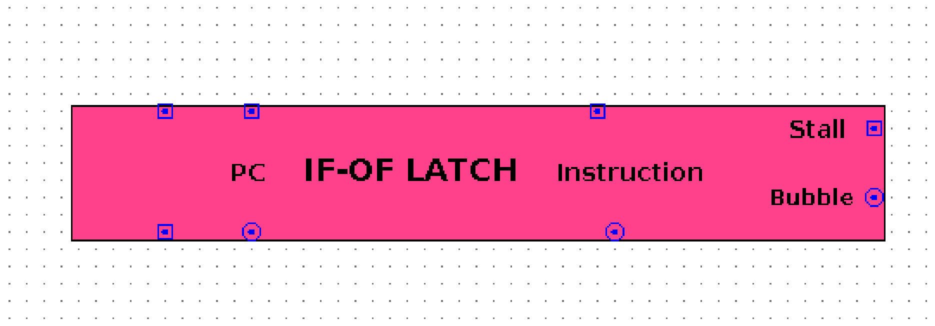
ALUSignals		
10	isAdd	$\overline{op5}\ .\overline{op4}\ .\overline{op3}\ .\overline{op2}\ .\overline{op1} + \overline{op5}\ .\mathrm{op4}.op3\ .op2$
11	isSub	$\overline{op5}\ .\overline{op4}\ .\overline{op3}\ .\overline{op2}\ .op1$
12	isCmp	$\overline{op5}\ .\overline{op4}\ .op3\ .\overline{op2}\ .op1$
13	isMul	$\overline{op5}\ .\overline{op4}\ .\overline{op3}\ .op2\ .\overline{op1}$
14	isDiv	$\overline{op5}\ .\overline{op4}\ .\overline{op3}\ .op2\ .op1$
15	isMod	$\overline{op5}\ .\overline{op4}\ .op3\ .\overline{op2}\ .\overline{op1}$
16	isLsl	$\overline{op5}\ .op4\ .\overline{op3}\ .\mathrm{op2}.\overline{op1}$
17	isLsr	$\overline{op5}\ .op4\ .\overline{op3}\ .op2\ .op1$
18	isAsr	$\overline{op5}\ .op4\ .op3\ .\overline{op2}\ .\overline{op1}$
19	isOr	$\overline{op5}\ .\overline{op4}\ .op3\ .op2\ .op1$
20	isAnd	$\overline{op5}\ .\overline{op4}\ .op3\ .op2\ .\overline{op1}$
21	isNot	$\overline{op5}\ .op4\ .\overline{op3}\ .\overline{op2}\ .\overline{op1}$
22	isMov	$\overline{op5}\ .op4\ .\overline{op3}\ .\overline{op2}\ .op1$

<u>LATCHES</u>

We have implemented the 5-stage pipeline in the processor by the use of latches between the stages. Instead of directly sending the output of one stage as input to the next stage, we have introduced buffer latches to hold the intermediate values between two stages.There are a total of 4 such latches, i.e - IF-OF Latch, OF-EX Latch, EX-MA Latch and the MA-RW Latch. In a pipelined processor, each stage handles a different instruction, so an instruction moves from one stage to the next with its complete instruction packet that contains its PC, 32-bit instruction, the control signals and intermediate results such as branch target and alu result.

There are some data paths that are not latched such as RW stage is writing directly to the register file in the OF stage, similarly the branch target is passed directly by the EX stage to IF stage.

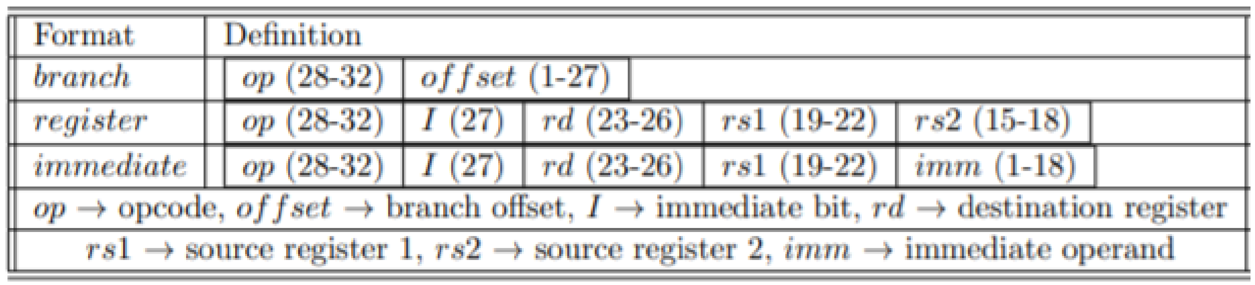




Operand Fetch Unit

OF - Operand Fetch, as the name suggests is used to decode the instructions and break them into respective fields. Simple RICS has a simple and regular instruction set. There are mainly three types of instruction formats: Branch, register, and immediate.
The operand fetch unit has two important functions -
1) Calculate the values of the immediate operand and the branch target by unpacking the offset embedded in the instruction.
2) Read the source register file.

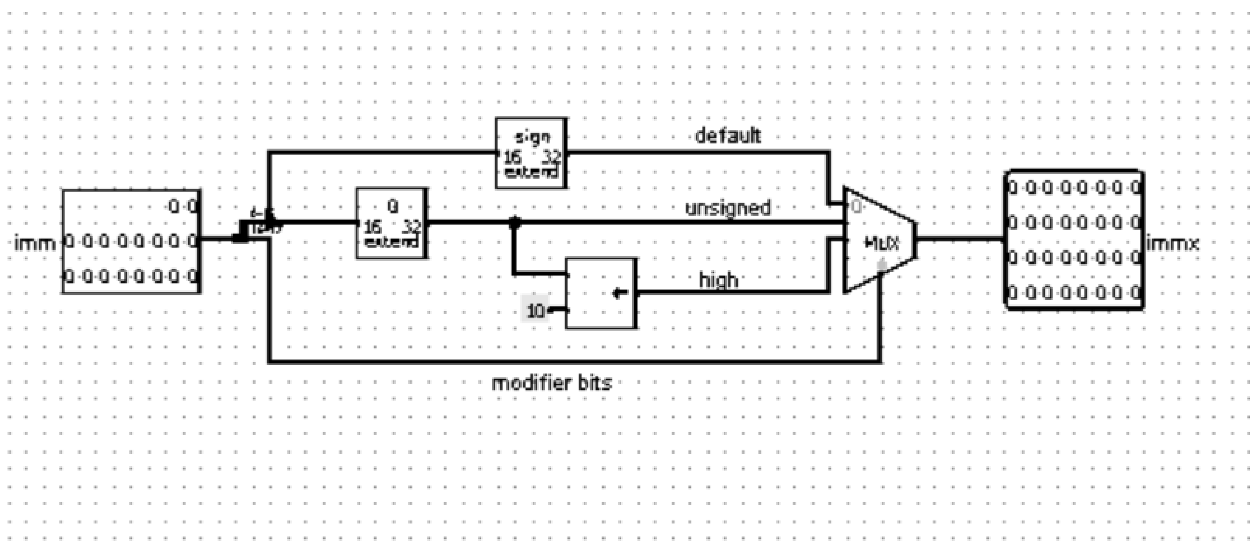
There are 27 instructions in simple RICS and hence 5 bits are used to represent these instructions. The instruction set consists of 32 bits out of which 5 are used to represent the opcode. The register is represented using 4 bits and an immediate value (extended immediate) is of 18 bits, with the first 2 bits representing the 3 modifiers (d - default, u - unsigned with MSBs 0's, h - unsigned with LSBs 0's). The three types of instruction formats have the following format for the instruction received from the IF unit. Each of these types needs to be handled separately.



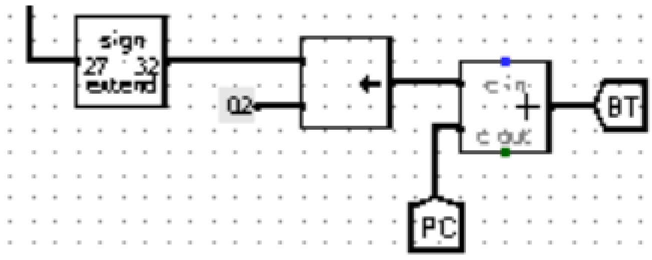
Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*rs*2 (15-18)
immediate	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs*1 (19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
*rs*1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

(pic ref: notes)
rd represents the destination address rs1 and/or rs2 are the registers containing source register address.

Calculation of immediate operand and branch target:



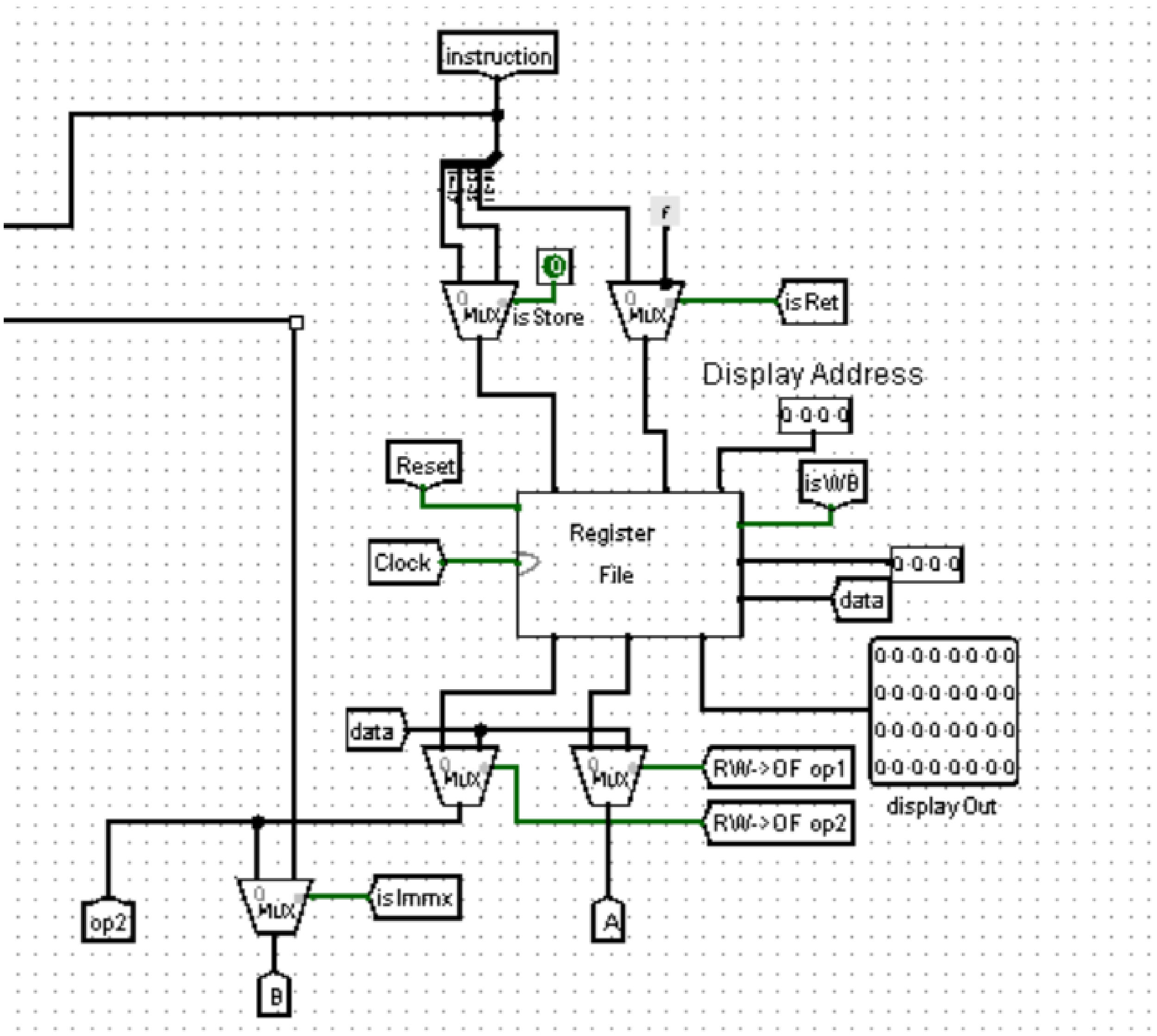
The circuit shown is used to calculate the immx(immediate extended) value. The three types of modifiers, default(d) which is the signed value and extends the sign to 32 bits. Unsigned(u) which add 0 at the beginning of the 16 bit immediate and high(h) which adds 0 at the end of the 16 bit immediate.



To calculate the target branch, we take the instruction and remove the opcode. The left 27 bits, which represent the offset of the memory word, is extended by 2 bit to make it 29 bits which is again extended in sign to make it 32 bits. This shifted offset is then added to the PC(Program counter) to get the branch target. Hence we use PC+ Offset addressing mode to calculate the branch target.

Here an important thing to note is that we calculate branch target and immx for all the instructions, and we mostly require only one of these, hence we are doing extra work here. But it doesn’t interfere with the speed of the processor and moreover, we want the correct values which may be useful for the future processes.

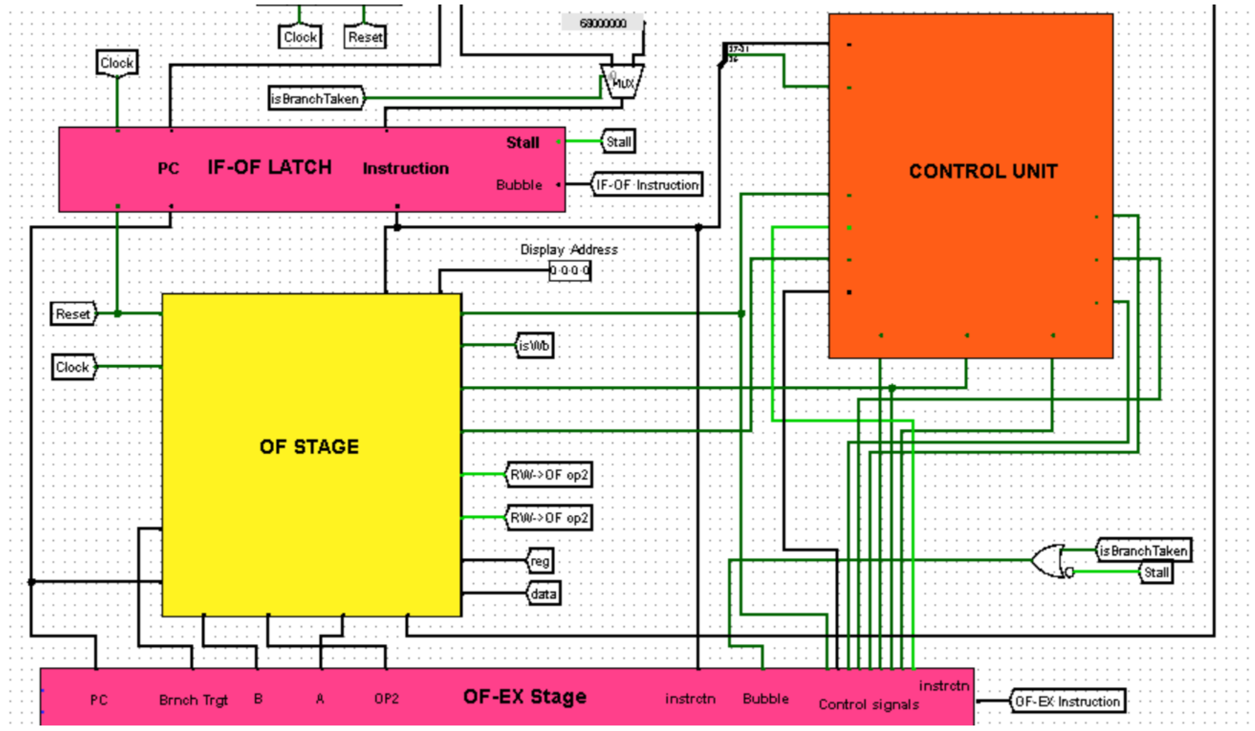
Reading the registers:



The register file has 16 registers, r0 to r15, 2 read ports, and a write port. For the first register operand, we have 2 choices. For ALU and memory instructions we read the first register source, rs1(19 - 22 bits). If it's a return instruction then the MUX output for the 1st operand would be the return address. This is controlled by the isRet control signal.

The second input to the register read port is either the destination address register, rd(23: 26 bits), or the second source register rs2(15: 18 bits). The isSt control signal controls the MUX for this. A store instruction doesn't have the 3rd field(rs2) and hence we use the rd as the 2nd output.
The register also has a write-back unit, which is controlled by the WB control signal and is used as input for writing back to the register file.
We can note that here also we do some extra work which sometimes may become power inefficient but it's better since the revised circuit would become slower.

The OF stage of the pipeline is shown below.



When we add a pipeline to the processor the only difference is the two new pipelines added above and below the OF unit - IF-OF latch and the OF-EX latch. The control unit generates the signals - isRet , isSt, and isImmx. The rest of the signals are not useful for the OF unit and hence are passed onto the OF-EX latch for use in further stages.
All the outputs generated by the OF unit are carried to the OF-EX latch. The two inputs to the ALU unit(A and B) , the branch target and the value to be written to the memory for store instruction , op2. And hence 4 packets are allotted in the OF-EX latch.

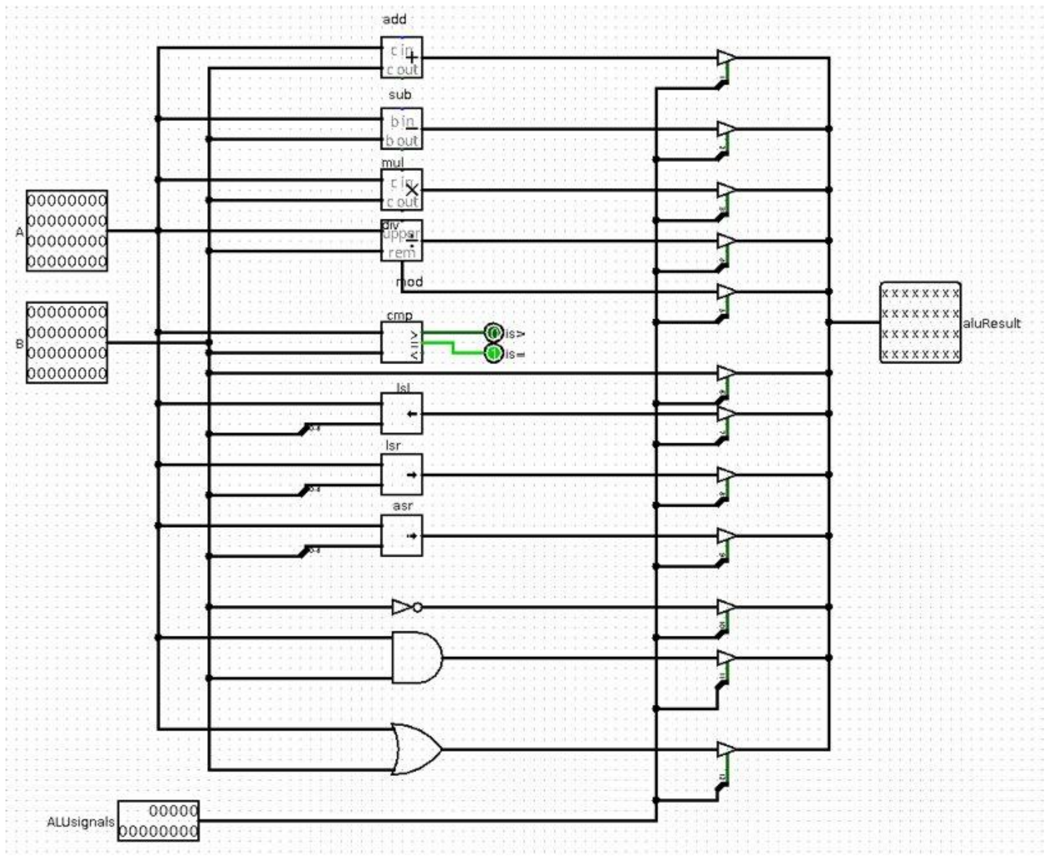
EXECUTE STAGE

The EX stage produces the aluresult and also updates the flags (isGT, isEQ) after result computation. The ALU takes the operands A and B as the input and performs the required operation as per the alu signals received by the EX stage from the OF-EX latch.

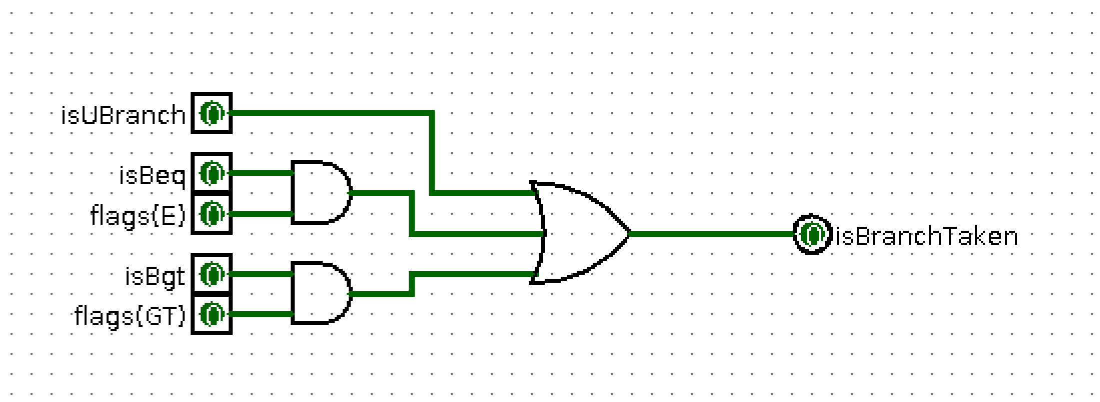
The branch unit receives the control signals (isBeq, isBgt, is UBranch) and flags and computes the branch target which is sent directly to the IF stage. The EX stage then passes the complete instruction packet to the MA stage.

INPUTS => operands A, B, op2, set of all control signals, PC, instruction

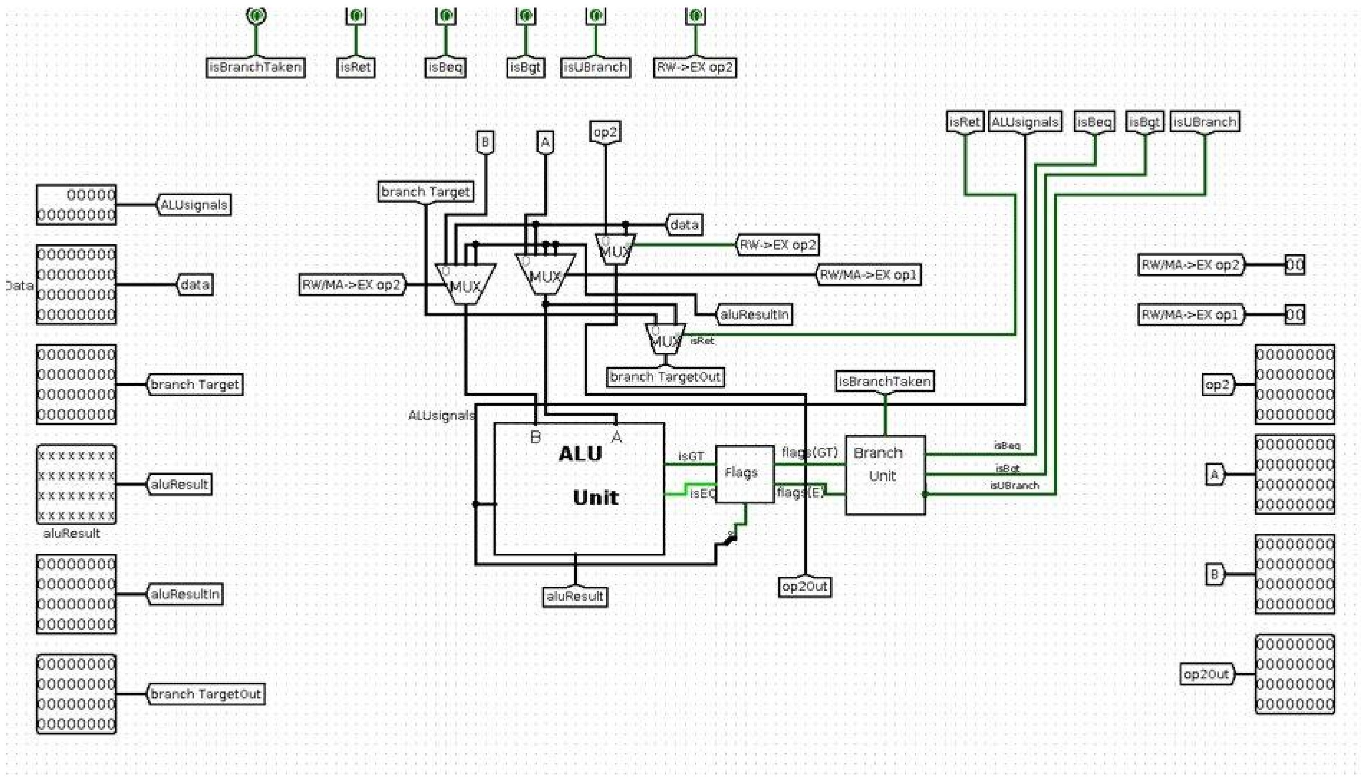
OUTPUTS => aluresult, branch target, set of all control signals, PC, instruction (passed directly from OF-EX latch)



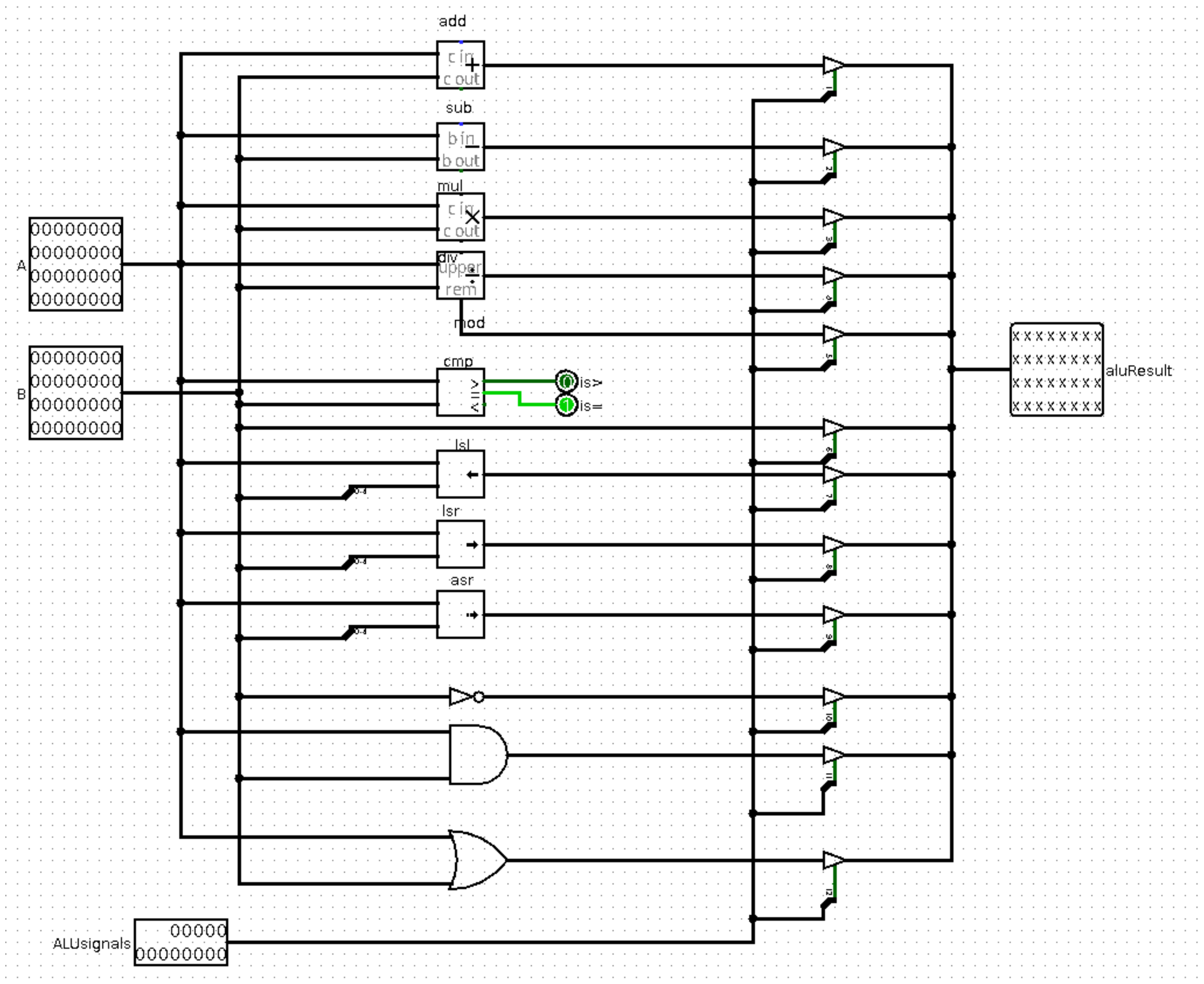
ALU



BRANCH UNIT



EX UNIT

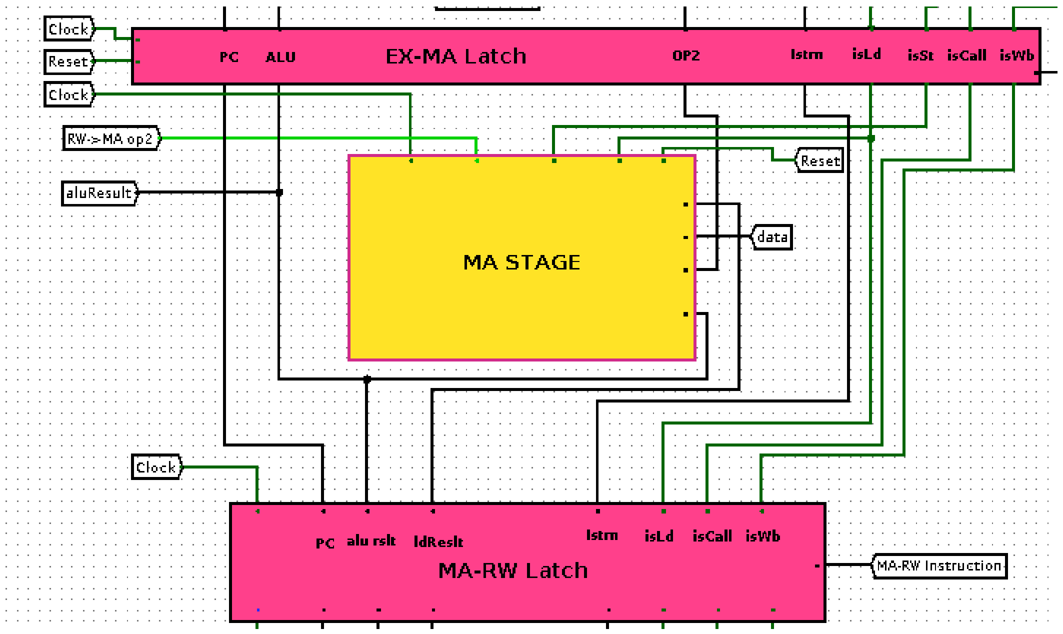

add
sub
mul
div
mod
cmp
lsl
lsr
asr
A
B
is>
is=
aluResult
ALUsignals
00000
00000000

Memory Access Unit

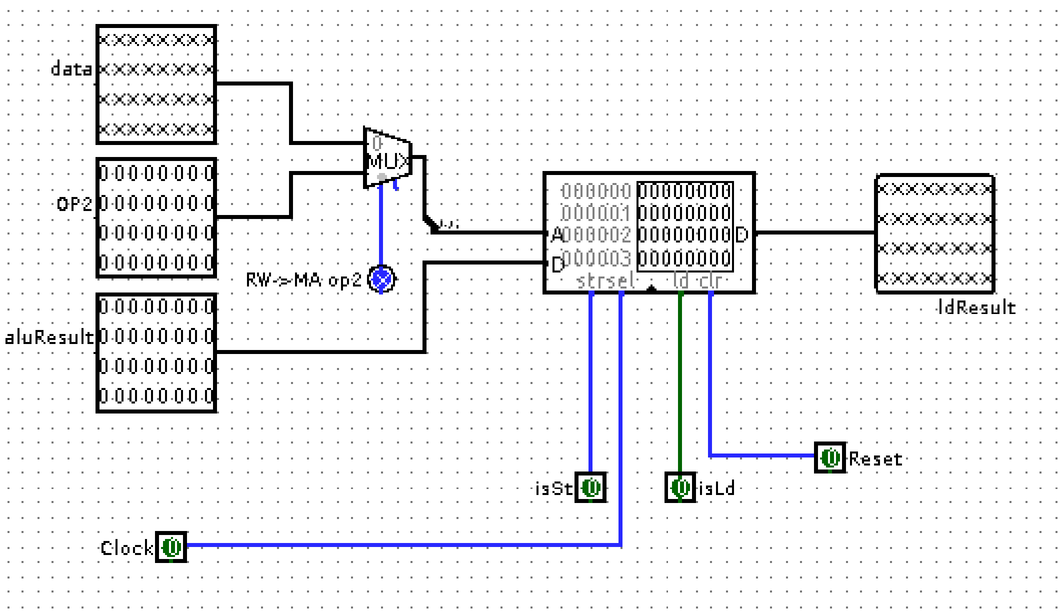
Memory Access Unit or the MA Unit stands between the Execute (EX) stage and the Register Writeback (RW) stage. It takes data as well as address as the inputs.
There are two sources of registers in the memory unit.

1. Memory Address Register (MAR)
2. Memory Data Register (MDR)

The aluResult field from the EX stage contains the address, which is calculated by ALU and it goes into MAR. op2 field contains the data that needs to be stored, which is written to MDR by the memory unit.

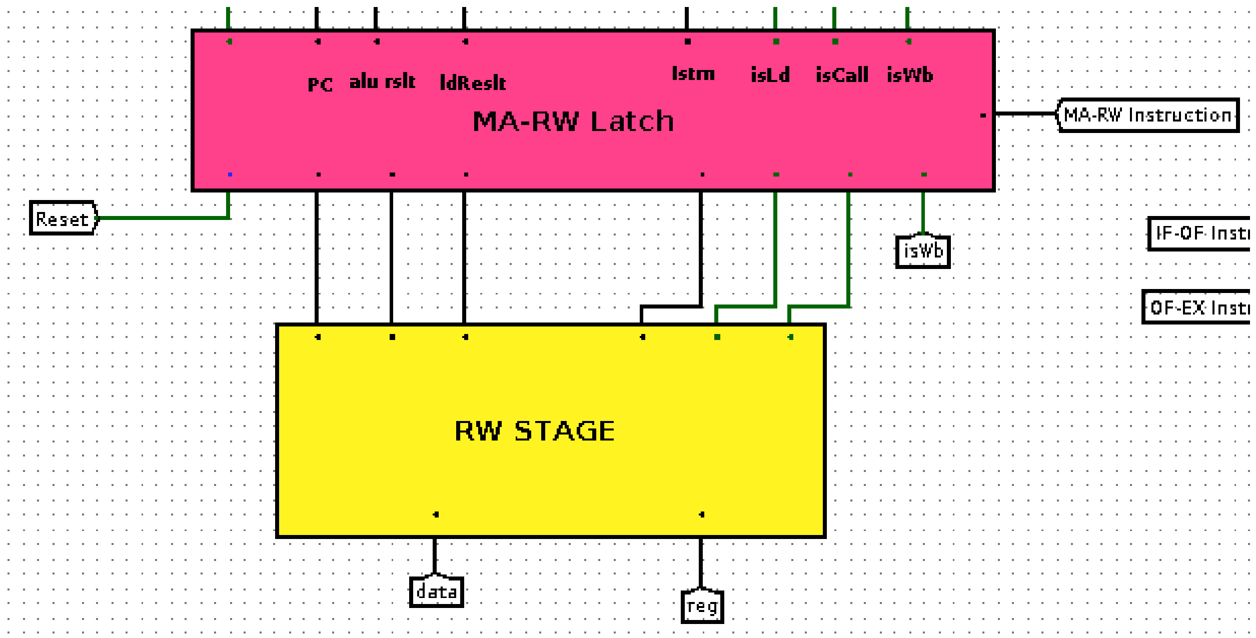


The memory unit also takes two control signals - isLd and isSt. At a given time, atmost one of these signals can be active. The memory unit is disabled if none of the signals is active which implies that the the instruction is not a memory instruction. In case of load instruction, the result of memory unit is stored in field called ldResult. There is no output in case of store instruction. So, the ldResult contains whatever we loaded from memory and rest of the fields like program counter, aluResult, instructions and control signals continues to the next stage, which is RW stage.

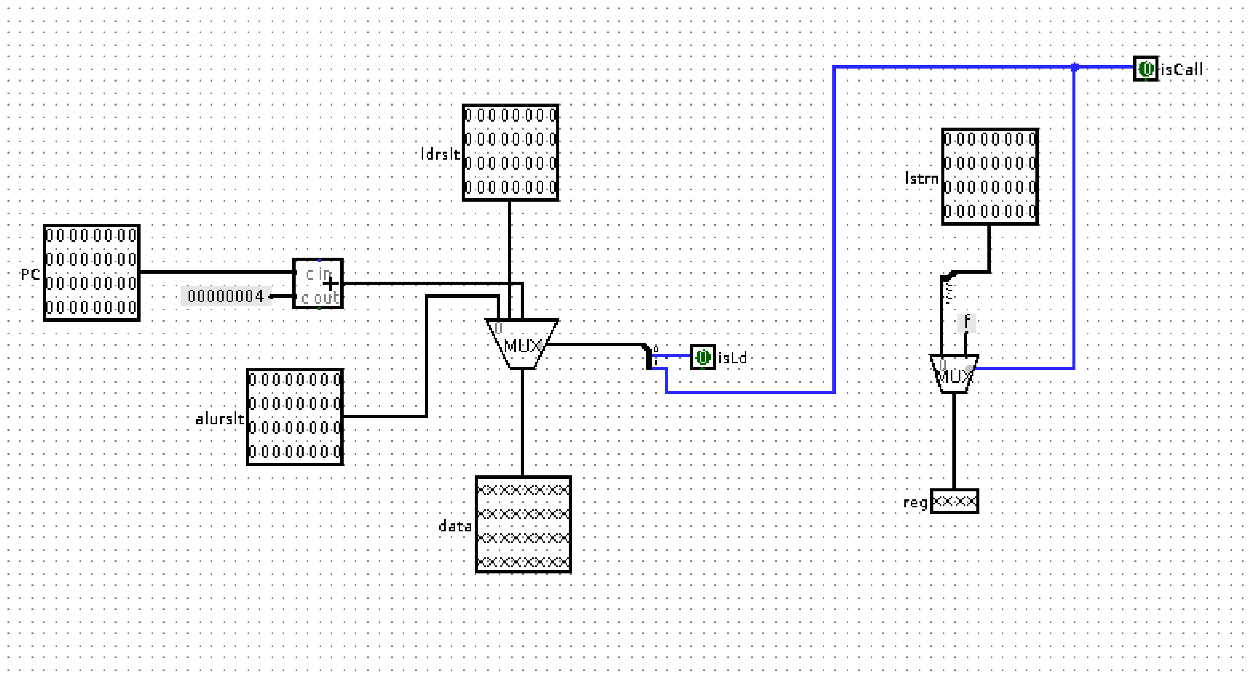


MA UNIT

Register Writeback Unit



Register Writeback unit or the RW stage is the last step of instruction processing. Here the computed values are written back to the register file. The value which is to be written can be the output of the load instruction, the output of the ALU or the return address which is equal to the PC of call instruction plus 4. I have implemented a three input multiplexer to chose between these three values - aluResult, ldResult and return address. The multiplexer is controlled by two control signals ‘isLd’ and ‘isCall’. When isCall signal is active, return address is chosen,which is PC+4. When isCall signal is inactive, ldResult and aluResult are chosen based on whether isLd signal is active or inactive respectively.



RW UNIT

One of the three inputs become the data that needs to be written to the register file and it goes directly to the register file. The register is located in RW stage but it can also be located in the OF stage. Then, the second multiplexer is implemented to choose the address, which is either the ra field (id of the return address) or the rd field (destination register). It is chosen by isCall control signal. The isWb control signal is used to enable the write.

FORWARDING UNIT

Output from MA-RW latch, EX-MA latch, OF-EX latch, IF-OF latch are inputs to the forwarding unit. We need to check if there is a conflict between 2 instructions in different stages. So, conflict is checked for instructions of MA-RW and EX-MA, MA-RW and OF-EX, MA-RW and IF-OF, EX-MA and OF-EX. Conflict is a read after write dependency.

Conflict in first operand:-

If the opcode of A belongs to beq, bgt, b, nop, not, call, mov then it means it does not use its operand so it will give no conflict. To check if any of beq, bgt, b, nop, not, call, mov is true or not, modified AND gates are used. Each output of AND gates is input to NOR gate which will give 0 (i.e no conflict) if any of the input is 1 (i.e output of AND gate). Let this operation be named CHECK OPCODE A1.

If the opcode of B belongs to beq, bgt, b, nop, st, cmp, ret then it means that we are not writing to any register, so there is RAW dependency so no conflict. Similarly, to check if any of beq, bgt, b, nop, st, cmp, ret is true or not, modified AND gates are used and its outputs are connected to inputs of NOR gate. Let this operation be named CHECK OPCODE B1.

If none of the above satisfies the condition then, we set a src1 variable which is equal to rs1 of operand of A (i.e 18-21 bits of A) but if opcode of A is equal to ret instruction then src1 is changed to ra. To transform this, an AND gate is modified such that it gives 1 if digits of opcode are equal to ret instruction. For selection of src1 value, MUX is used with output of AND gate as selector and rs1 as input0 and ra as input1. A variable dest is set to rd of operand B (i.e 22-25 bits of B) but if opcode B is equal to call instruction then dest is changed to ra. Similarly, to implement this, AND gate and MUX is introduced in an orderly manner.

If src1 = dest, then conflict occurs. As we know, this output depends on the output of CHECK OPCODE A1 and CHECK OPCODE B1 as the conflict occurs on rejection of these 2 conditions. So, a MUX is used with a selector as output of the AND gate whose inputs are CHECK OPCODE A1 and CHECK OPCODE B1’s output. 0 is input0 and comparator output as input1. Comparator with input src1 and dest will give 1 iff src1 = dest. MUX output gives conflict result of the first operand.

Conflict in second operand:-

If the opcode of A is equal to any of beq, bgt, b, nop, call then it does not read from any register so it will give no conflict. To implement this, each instruction is compared with opcode A using

AND gate. Outputs of AND gate are input to inputs to a NOR gate which will give 0 (no conflict) if all inputs are 1. Let this operation be named CHECK OPCODE A2.

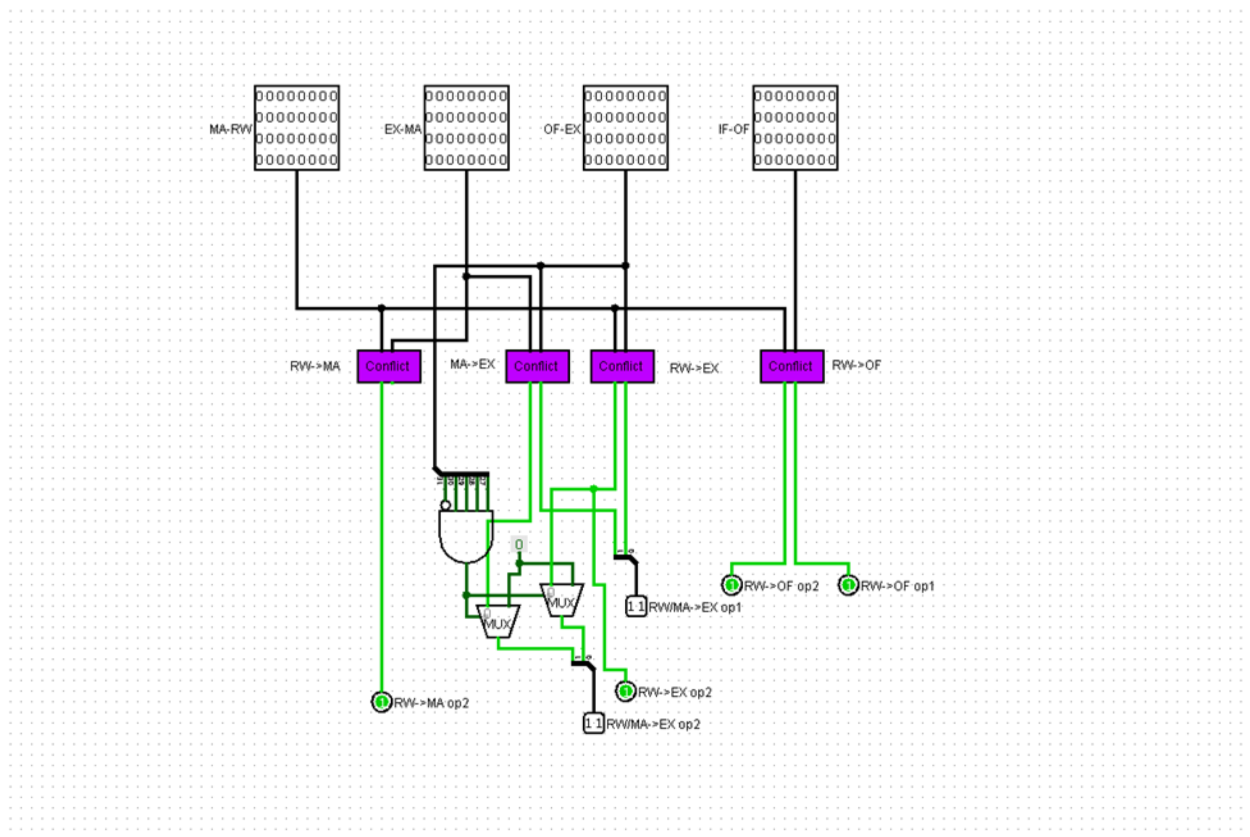
The condition for opcode B here is the same as in CHECK OPCODE B1.

Now see if the second operand to see if it is a register. If opcode A is not a store instruction and if A's immediate (i.e bit number 26) is 1 then it gives no conflict. That means if opcode equal to st = 0 and A's immediate = 1 the result is 0 else result is 1. So, a MUX is used. An AND gate named 'CheckSt' is implemented to check equality between opcode and st. MUX is used with selector 'CheckSt' and A's immediate as input0 and 1 as input1. Let this operation be named SPLIT A.

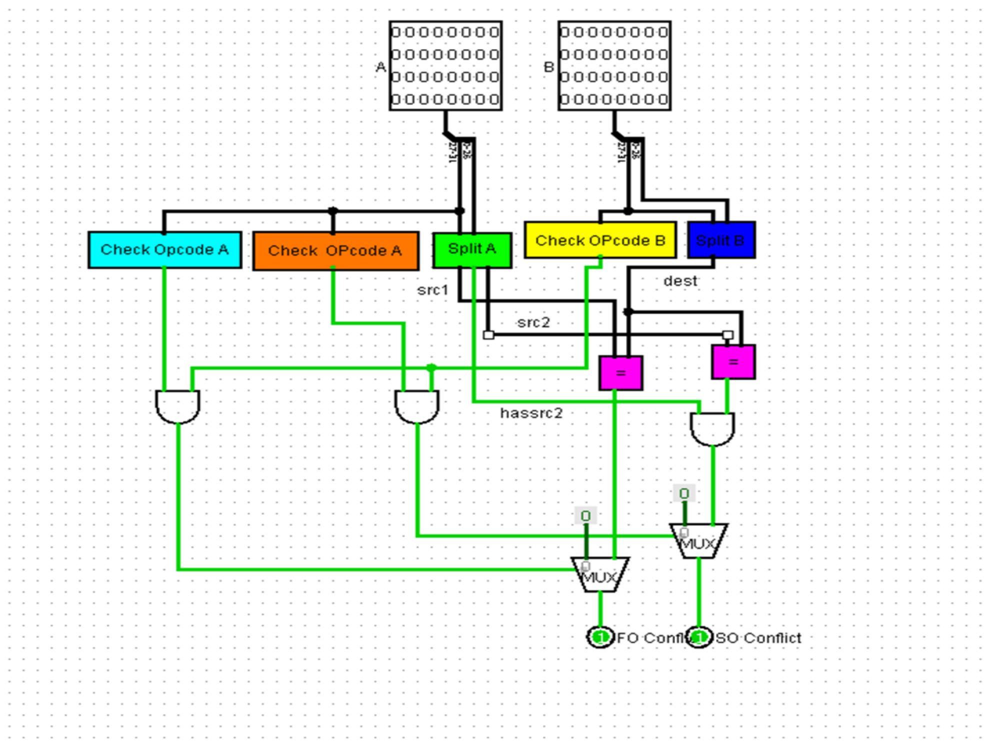
Now that second register exist, a variable src2 is set to rs2 of A (14-17 bits of A). If opcode A is st instruction then src2 is set to rd of A (22-25 bits of A). To implement this, MUX is used with selector CheckSt and rs2 as input0 and rd as input1.

Then a variable dest is set which undergoes same conditions as in first operant condition. To check final condition src2 = dest. But final condition is not only dependent on src2 and dest, it also depends on output of SPLIT A, CHECK OPCODE A2, CHECK OPCODE B1. To check src2 = dest, a comparator is used. AND gate(a1) is used with inputs - SPLIT A and comparator's output. Implementation of the effect of CHECK OPCODE A2 and CHECK OPCODE B1 is displayed using AND gate(a2). MUX is introduced to implement the final condition, selector – a2 and 0 as input0 and a1 as input1. This give conflict result for second operand.

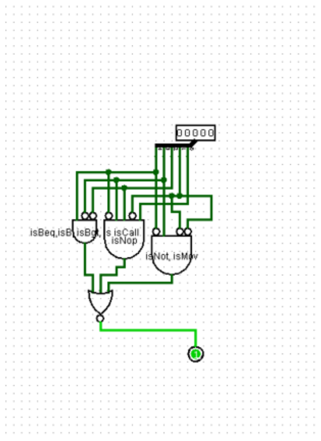
Outputs from 4 conflict units will help in choosing between default input and forwarded input. 1st conflict unit's output will used in MA stage, first operand conflict will not be used here as operand 1 input will be used in ALU and only operand 2 is passed without operation in EX stage. 4th conflict unit outputs will be used in OF stage. EX stage used 2 to 4 MUX which will be used in ALU. So, 2nd and 3rd conflict unit outputs are merged. If opcode of OF-EX instruction is st instruction, then for operand 2 - RW/MA to EX is 00, else RW/MA to EX value is unchanged. For operand 1 - RW/MA to EX, bits are merged accordingly.



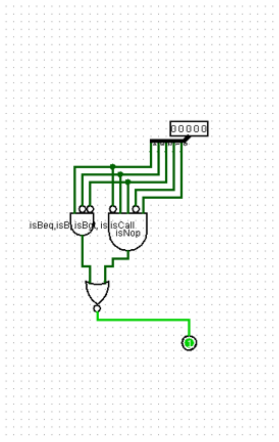
Forwarding unit



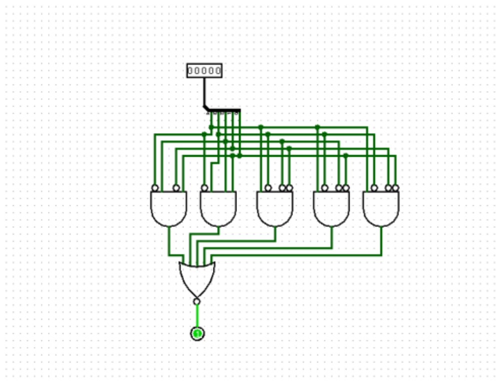
Conflict unit



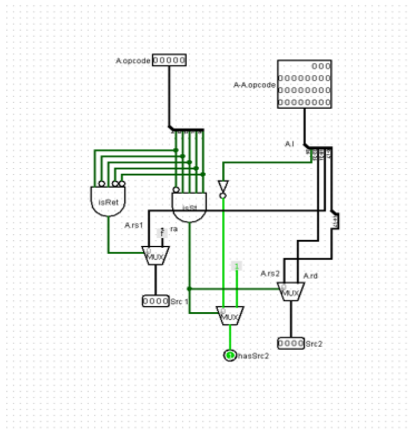
CHECK OPCODE A1
(beq,bgt,b,nop,not,call,mov)



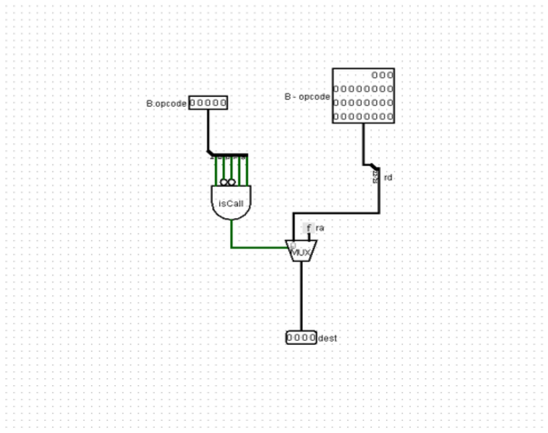
CHECK OPCODE A2
(nop,b,beq,bgt,call)



CHECK OPCODE B1
(beq,bgt,b,nop,st,cmp)



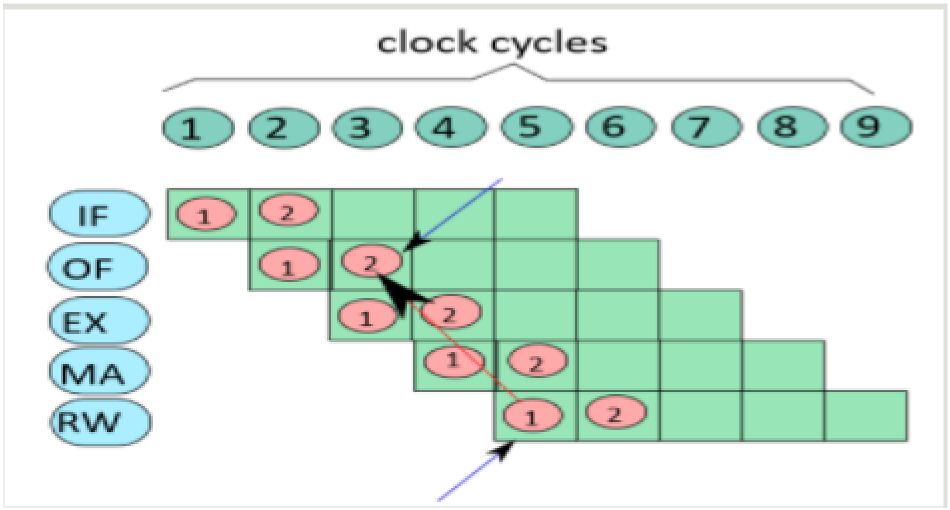
SPLIT A



SPLIT B

Pipeline hazards:

Data Hazards : It occurs due to unavailability of data, or the availability of incorrect data in pipeline stages.

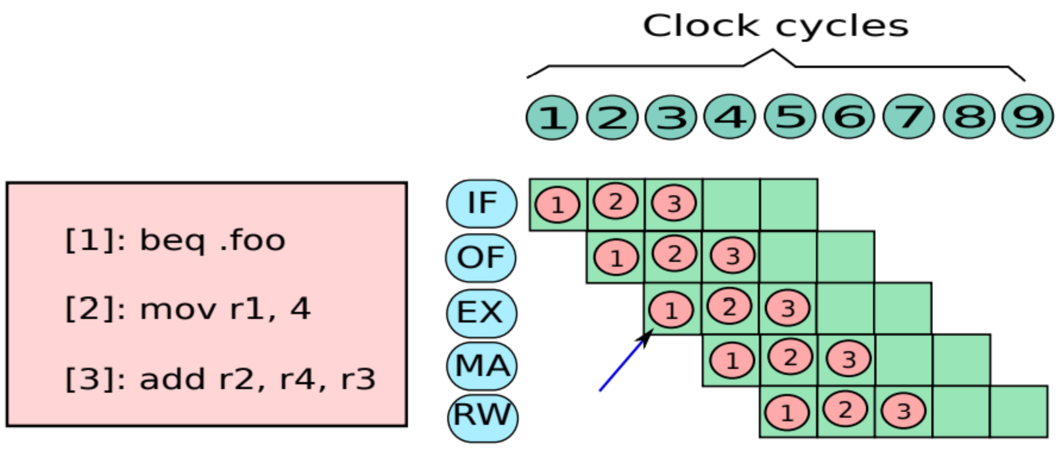


(pic ref: Computer Organisation and Architecture, by Smruti Ranjan Sarangi)

Ex :- [1] add r1 r2 r3
    [2] sub r3 r1 r4
In this example, r1 is read before the 1st instruction is completed . Therefore the OF unit will read the wrong value of r1 and hence we will get an incorrect result.

Control Hazards :

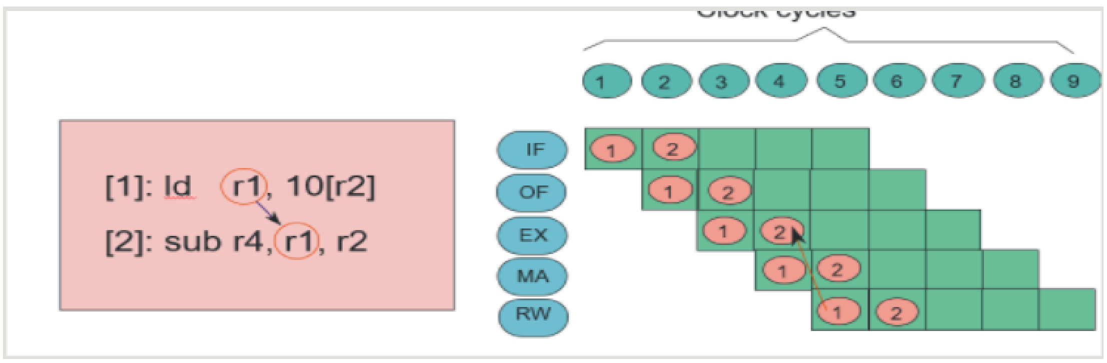
```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
...
...
.foo:
[100]: add r4, r1, r2
```



(pic ref: Computer Organisation and Architecture, by Smruti Ranjan Sarangi)

In this example, outcome of the branch is decided in cycle 3,there is no way of knowing in cycles 2 and 3, about the outcome of the branch. If the branch is taken, then there is a possibility that instructions 2 and 3 might corrupt the state of the program, and consequently introduce an error. This is called control Hazards.

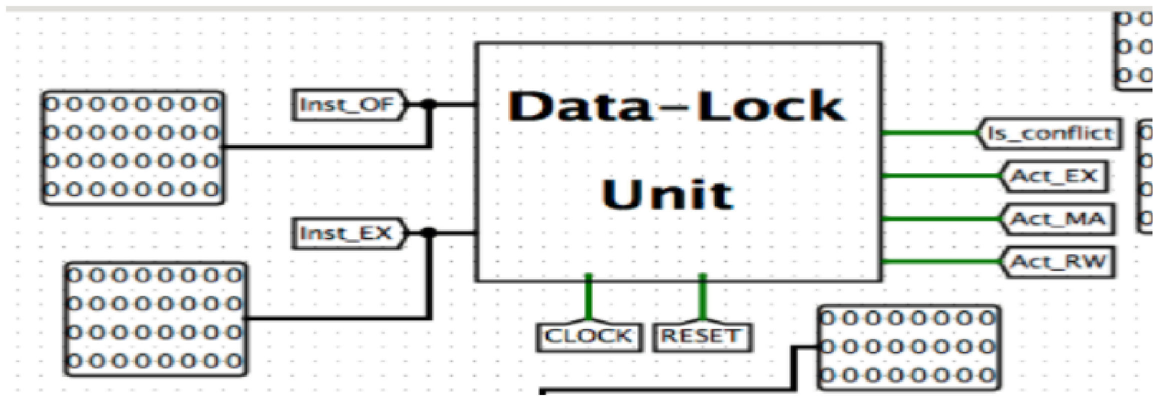
Load-use Hazard in forwarding :



(pic ref: Computer Organisation and Architecture, by Smruti Ranjan Sarangi)

Here, in this example, there is no way that RW can forward it's data to EX of second instruction. That's why it will cause Load-use hazard. Here, we need to stall (add a bubble) and then use RW -> EX forwarding.

<u>Data lock unit</u>



If the instruction in EX stage is a "LOAD" and the instruction in OF stage uses It's loaded value, then it stalled for 1 cycle. Inst_OF and Inst_EX is used to determine whether there is a hazard or not. If there is hazard, data-lock unit will raise the flags to stalling a clock cycle for stages EX, MA and RW.

6. Evaluation parameters

We evaluate the processor using benchmarks to ascertain the capacity of the pipelined simple RISC processor. We measure how much time it takes for us to run specific benchmarks and see the speed-up factor of the pipelined processor to a non-pipelined one in various cases. This gives a rough idea of the power of our processor in execution.

Below is the table listing the processor's specifications:

Table-1:

Benchmark Name	No of instructions	No of effective instructions	Time taken as estimated in a non-pipelined processor	Time taken in pipelined processor	Speed up factor	Remarks
Arithmetic	14	14	70	21	3.33	Simple arithmetic operations
Or-and-not	8	8	40	11	3.64	Calculate the XOR
Load-store	8	8	40	14	2.85	Storing and loading values from a 1D array
Modifiers test	3	3	15	7	2.14	Check working of modifiers h and u
Shift test	15	96	480	144	3.33	Test for lsl,lsr,asr
Call test	18	18	90	40	2.25	Test for call and ret
Branch test	24	18	90	27	3.33	Test for b,beq,bgt
Extensive cmp and branch test	21	40	200	62	3.22	Extensive test for b,beq,bgt
Data hazard test	30	30	150	35	4.28	Test if pipeline handles hazards
Ideal pipeline test	14	14	70	18	3.89	Test if processor doesn't introduce unnecessary

						stalls or flushes

The above benchmarks are for fundamental test purposes. They don't convey the full power of our processor accurately as the number of instructions are small. To accurately measure the performance of our processor we try to run larger programs. The drawback in this case is that in larger programs it becomes difficult to estimate time taken for execution of the program in a non-pipelined processor.

List of larger programs we tried:

Table-2:

Program name	No of instructions	Time taken in pipelined processor	Time as estimated in non-pipelined processor	Speed up factor	Remarks
Bubble-sort	36	244	1073	4.39	No. of effective instructions varies depending on inputs
Bubble-sort (Variation 2)	36	134	590	4.4	Best case
Bubble-sort(Variation 3)	36	140	611	4.36	Another variation
Factorial	9	54	155	2.87	Computes the factorial of a number
Least Common Multiple	25	131	410	3.13	Computes LCM of 3 numbers
Highest Common Factor	24	101	355	3.51	Computes HCF of 3 numbers

Average speed up				3.77	

Our processor is an average 3.77 times faster than a normal simpleRISC processor in executing some common programs

7. Results and Discussion

The processor was built by dividing the system into the following stages:

- Instruction Fetch (IF) stage: Fetches instruction from memory and increments the program counter methodically to point to the next desired instruction only.
- Operand Fetch (OF) stage: Calculates the appropriate branch target, extends the immediate value to make it suitable for the 32-bit processor and reads registers for data that will be used for further operations.
- Execute (EX) stage: Uses the ALU to perform arithmetic and logical operations on the operands received from the OF stage
- Memory Access (MA) stage: Accesses memory as required by the current instruction
- Read-Write (RW) stage: Updates the value of the registers as required by the current instruction

After rigorously evaluating the processor against a range of programs, the functioning of the various components was verified and the following inferences were drawn:

- The processor offers appreciable speed-up in all the operations due to the use of pipeline and thus improves the overall performance of the processor.
- A relatively high speed-up of around 4.4 was achieved for bubble sort, one of the most trivial sorting algorithms. But, when another complex program, factorial, was computed the speed-up was not as high. This could convey that the algorithm used does not exploit the use of a pipeline significantly. Further, some errors could have been introduced due to the estimation of the time taken in a non-pipelined processor and thus the values might not convey the whole picture.

The speed-up for each execution can be seen in Table-1 and Table-2.

Overall, the processor does a good job in dealing with the data hazards. An average speed-up of 3.77 was obtained. We plan to further integrate our processor with hardware prefetchers, branch predictors and cache management units to make it more reliable and improve its efficiency. Cache management will not only give us a structured management system to deal with the data used and generated during various processes but also provide in-memory storage to increase data retrieval performance. This integration will make our processor a well-rounded system.

8. Conclusion

By Building on our knowledge of a SimpleRISC processor and combining it with the essential 5-stage pipeline structure, we have been able to successfully design and implement a 32-bit SimpleRisc processor with forwarding and lock units . The integration with the pipeline provided significant speed-up in implementation of programs with polynomial complexity. The working of the processor has been fully verified through test-programs and benchmark evaluations. Further, the project allowed us to delve deeper into the nitty-gritty of basic computer architectures and understand the importance of individual components. In the end, we were able to build a processor which is capable of performing many common instructions that are typically required by processors used in small embedded systems.

9. References

- Computer Organization and Architecture, by William Stallings
- Computer Organization and Design, The hardware and software interface, by David A Patterson & John L Hennessy
- Advanced Computer Architecture, by Smruti Ranjan Sarangi
- Computer Organisation and Architecture, by Smruti Ranjan Sarangi
- https://www.youtube.com/watch?v=uCOrNJzXcDU&list=PL1iLu2CSC9EWAo0ysorNI_nebwF6Rwkr0
- https://github.com/yxwangcs/MIPS-CPU/tree/main/Benchmarks
- https://en.wikipedia.org/wiki/Operand_forwarding
- https://www.youtube.com/watch?v=iat7ucqD7-M
- https://ieeexplore.ieee.org/abstract/document/7019240