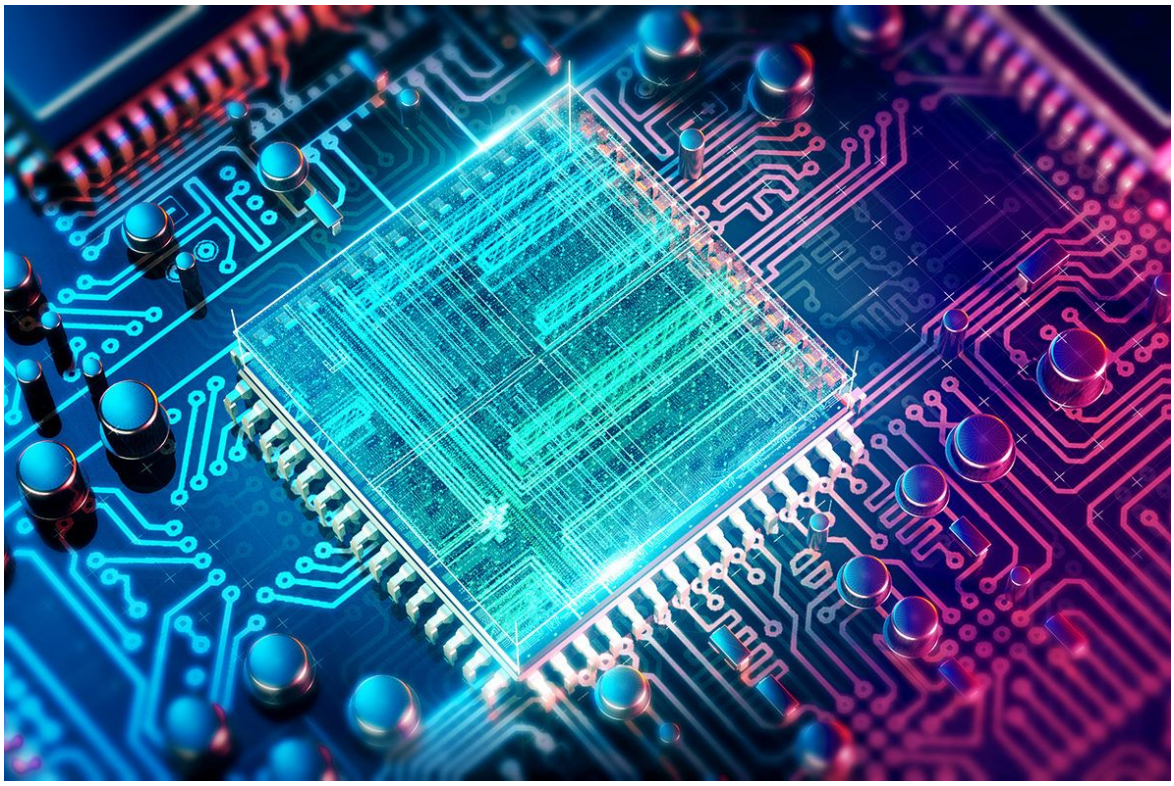
CSN-221 Project Report

Computer Architecture and Microprocessors (CSN-221), Autumn 2020



GUIDED BY: PROFESSOR SATISH KUMAR PEDDOJU

Aditya Rai	19114004
Gagan Sharma	19114032
Gajanan Gitte	19114033
Raghav Somani	19114068
Shlok Goyal	19114078
Gaurav Wasnik	19114090

BTech CSE 2nd year, IIT ROORKEE

The Project Statement

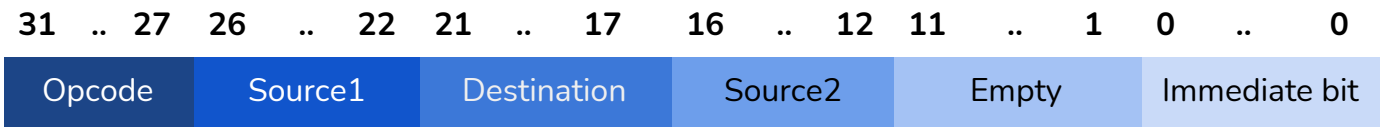
Design a CPU including ALUs and Register files, the Control circuitry, Instructions flow etc., on any simulator.

We have designed a 32-Bit RISC inspired Processor with Von Neumann Architecture using *Logisim*. We have also developed an assembler in python which converts the assembly code into machine language.

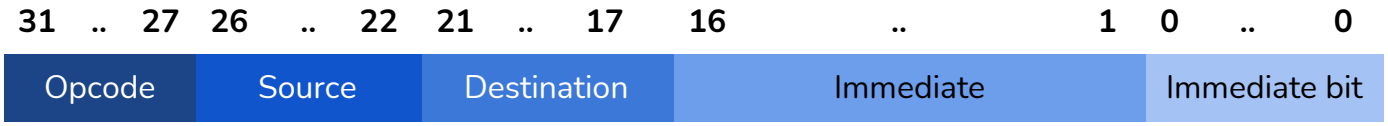
Instruction Set Architecture

The ISA used in this processor is inspired by RISC. The ISA consists of 16 main instructions. Given below is the instruction format of the ISA:

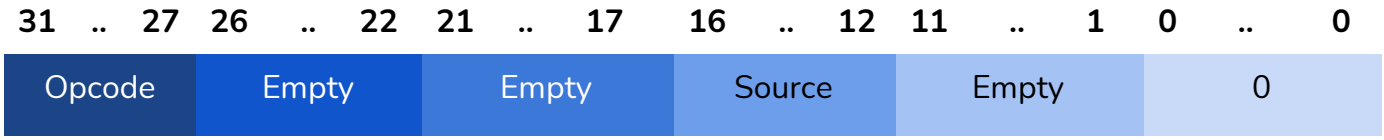
R-Format

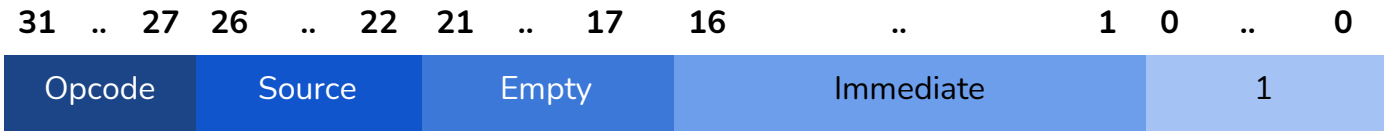


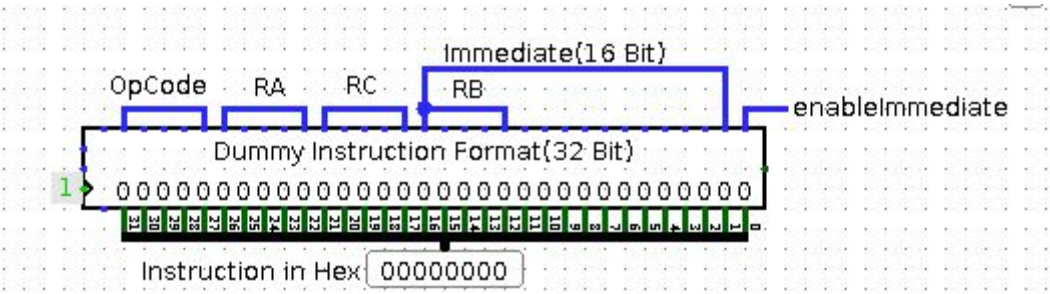
I-Format



J-Format







Register Instructions

OpCode	Immediate Bit	Operation	Assembly instruction
0000	0	Halt	halt
0001	0	Load	load
0010	0	Store	store
0011	0	Move	mov
0100	0	Jump	jump
0101	0	Jump If Zero	jz
0110	0	Jump If Carry	jc
0111	0	No Operation	nop

1000	0	Add	add
1001	0	Subtract	sub
1010	0	And	and
1011	0	OR	or
1100	0	NOT	not
1101	0	Multiply	mul
1110	0	Divide	div
1111	0	Compare	cmp

Immediate Instructions (If immediate bit is one)

OpCode	**ImmediateBit**	**Operation**	**Assembly Instruction**
0001	1	Load	loadi
0011	1	Move	movi

0100	1	Jump	jumpi
0101	1	Jump If Zero	jzi
0110	1	Jump If Carry	jci
1000	1	Add	addi
1001	1	Subtract	subi
1010	1	And	andi
1011	1	OR	ori
1101	1	Multiply	muli
1111	1	Compare	cmpi
1110	1	Divide	divi

Assembler

A python based assembler script is included with the processor. The assembler converts high level assembly language into machine instructions and outputs a Logisim memory image. Load this memory image into the simulator to run the program.

Assembler Documentation

Assembling Programs

1. To run programs on the processor, first assemble the memory image.
   `python assembler.py inputfilename [outputfilename]`
2. Load the memory image inside *Logism* by right clicking on the RAM module and selecting Load Image...
   - The programs are loaded at starting address 0x0000
   - Each instruction in the assembly language takes up one memory word of 32-bit.

Assembler Language Commands

Comments

Comments start with #. Any content after a # is ignored.

```
# This is a comment.
```

Halt

Halts the processor. To resume click the reset button.

```
halt
```

Load

Loads value from memory.

```
load [Destination] [Source]
load r3 r2

loadi [Destination] Immediate
loadi r3 0xabcd
```

Store

Store a value into memory.

```
store [Data] [Address]
store r3 r2
```

Mov

Move data from one register to another.

```
mov [Destination] [Source]
mov r3 r2

movi [Destination] ImmediateValue
movi r3 0xabcd
```

Jump

Jump to a particular address.

```
jump [Register with address]
jump r5

jumpi ImmediateAddress
jumpi 0xabcd
```

Jump if Zero

Jump to a particular address if zero flag is set.

```
jz [Register with address]
jz r6

jzi ImmediateAddress
jzi 0xabcd
```

Jump if Carry

Jump to a particular address if the carry flag is set.

```
jc [Register with address]
jc r7

jci ImmediateAddress
jci 0xabcd
```

Nop

Do Nothing

```
nop
```

Add, Subtract, And, Or, Multiply, Divide

Perform the arithmetic operation and store result to destination register

```
op [destination] [source1] [source2]

add r3 r1 r2
sub r3 r1 r2
and r3 r1 r2
or  r3 r1 r2
mul r3 r1 r2
div r3 r1 r2

opi [destination] [source1] Immediate Value
subi r3 r1 0x3
andi r3 r1 0x4
ori  r3 r1 0x5
muli r3 r1 0x6
divi r3 r1 0x2
```

Not

Invert bits and store them in the destination register.

```
not [destination] [source]
```

Cmp

Compares two values and sets flag register.

```
If Source1 > Source2 then the Carry flag is set.
If Source1 == Source2 then Zero flag is set.


cmp [source1] [source2]
cmp r1 r2

cmpi [source1] [source2]
cmpi r1 0xffff
```

CPU Design

CPU Features

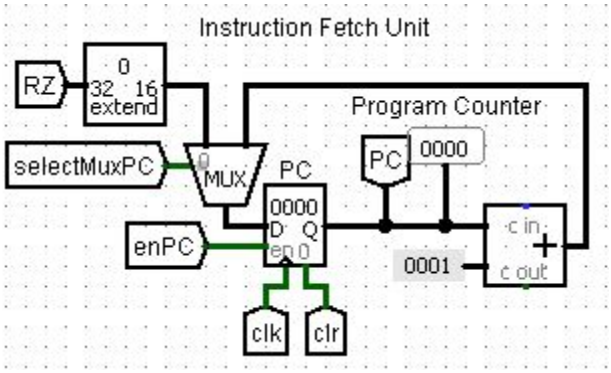
- The whole CPU is hardwired design with Von Neumann architecture.
- The RAM is 256KB in size with 16-Bit address bus and 32-Bit word length.
- There are 32 general purpose registers with 32-Bit word length.
- The CPU uses 16-Bit immediate.
- Each assembly instruction is executed in 5 stages separately.
- Instruction is fetched automatically at the start of the simulation and halts at the end of the program.
- An assembler is written based on Python which translates assembly instructions to machine language (in hexadecimal).

5 stages of execution

- Instruction Fetch stage (IF) : fetch next instruction
- Operand Fetch stage (OF) : decode the instruction and fetch operands
- Execution stage (EX) : execute ALU operations and generate results
- Memory Access unit (MA) : Access memory for load and store operations
- Register Write Stage (RW) : Writeback to the register if needed

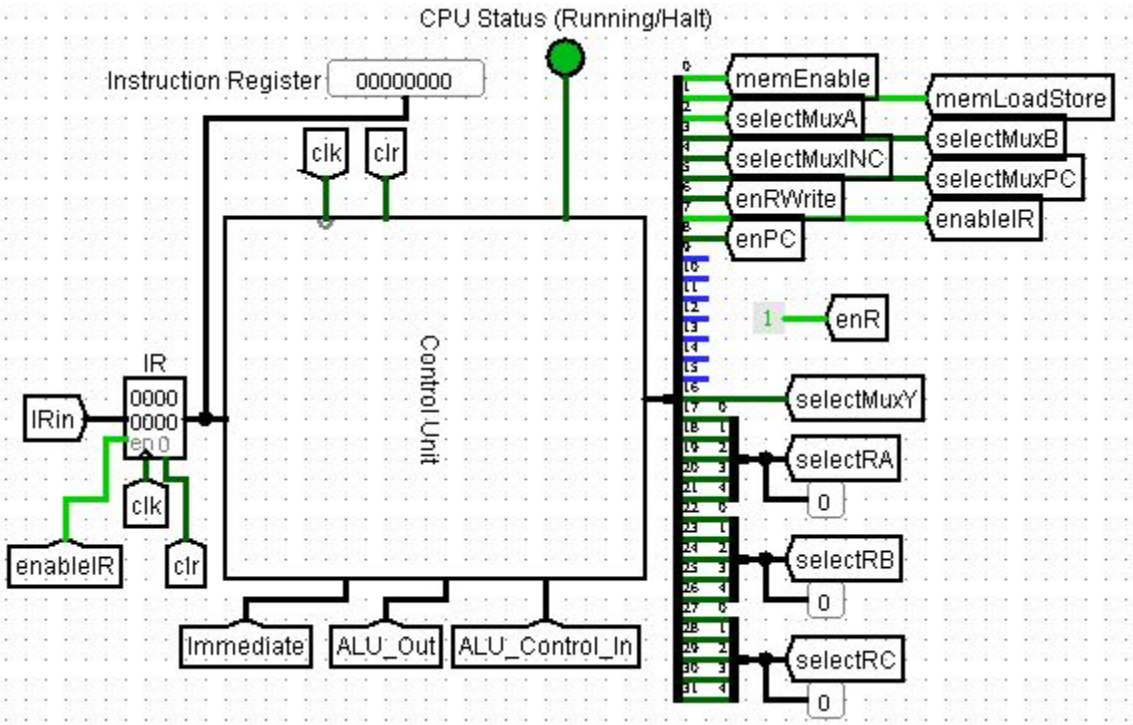
Working Components

1. **Instruction Fetch Unit**



- **PC** holds the address of the instruction that has to be executed.
- The **adder** increments the PC to fetch the next instruction for execution.
- **RZ** holds the address of the instruction that has to be executed in case of jump instruction.
- The **selectMuxPC** is 0 when jump has to be taken and 1 when the next instruction needs to be executed.
- The **enPC** is 1 when the PC is allowed to update and 0 when it's not.
- The clock is represented by **clk**.
- If the PC needs to be reset the **clr** is set to 1 else 0.

2. **Control Unit Signals**



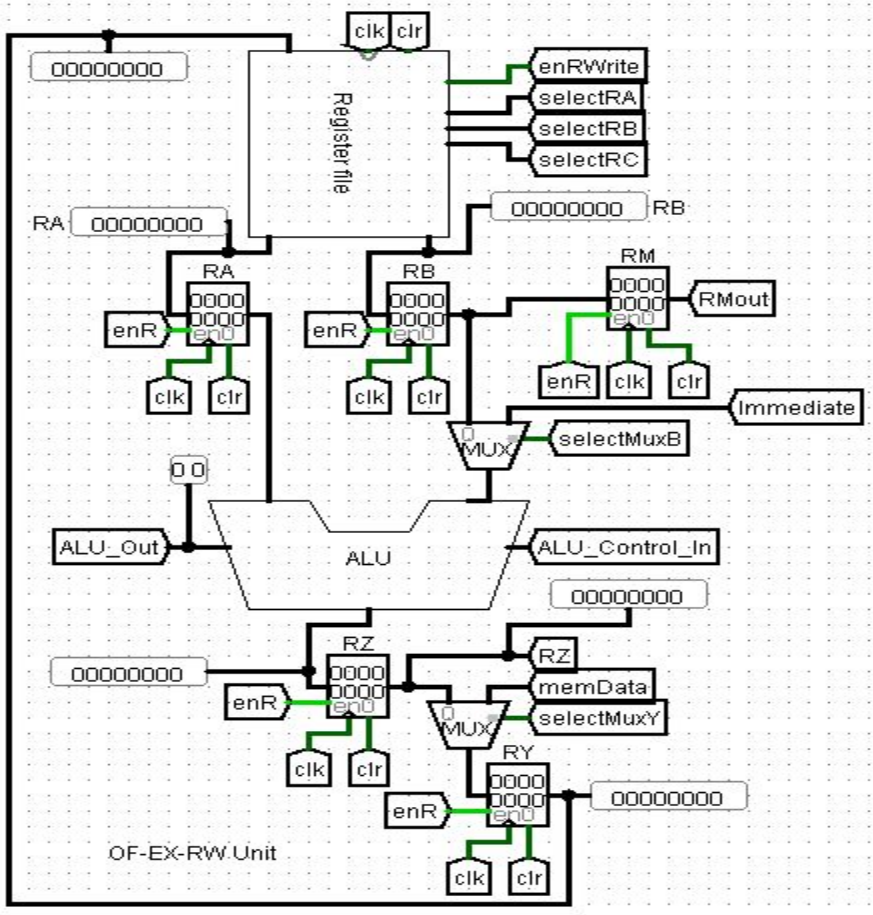
- IRin contains the instruction that has to be executed and stores it in **IR**(Instruction Register).
- The signal **enableIR** is 1 when updating IR is allowed else 0.
- The clock is represented by **clk**.
- To reset the IR **clr** is set to 1.

Control Signal	Function
memEnable	Memory access is allowed-1 Otherwise-0
memLoadStore	Store instruction-0

	Otherwise-1
selectMuxA	Fetch next instruction-1 Load/Store instruction-0
selectMuxB	2nd operand is immediate-1 2nd operand is register-0
selectMuxINC	Allow incrementing PC-1 Otherwise-0
selectMuxPC	Next instruction-1 Jump instruction taken-0
enRWrite	Enable Register write-back-1 Otherwise-0
enableIR	Allow update of IR-1 Otherwise-0
enPC	Allow update of PC-1 Otherwise-0
enR	Enable temporary result to update to be later stored in a register-1
selectMuxY	store load result in a register-1 Store aluResult in a register-0
selectRA	Select bits for 1st operand-5 bits
selectRB	Select bits for 2nd operand-5 bits
selectRC	Select bits for destination-5 bits
Immediate	Calculated immediate-32 bits

ALU_Out	Values of C (greaterThan) and Z (equalTo) flags-2bits
ALU_Control_In	Contains information about aluOp and if comparing flags are required or not-8 bits

3. **OF-EX-MA-RW stage**
    - Operand Fetch, Execution and Register write are depicted in 1 circuit.
    - Memory Access Unit is a separate circuit which helps in load/store instruction and fetching new instructions for execution. It will be discussed after this section.

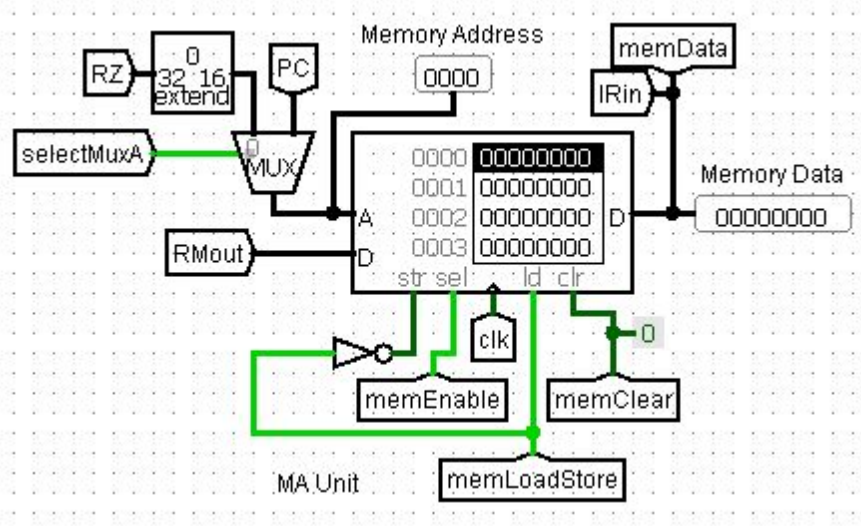


- The selectRA and selectRB give the addresses of operand

registers and the values of these registers are stored in RA and RB respectively.

- RB is also stored in RM which is the value stored in the memory in case of store instruction.
- The selectMuxB signal helps in deciding if we need immediate or RB for the ALU operation.
- The ALU operation takes place in the ALU Unit with RA and RB as operands, ALU_Out for information needed for C and Z flags, and ALU_Control_In to decide the ALU operation.
- The aluResult is stored in the RZ register.
- The memData signal contains the value from load operation (ldResult).
- The selectMuxY decides if the operation is load operation or not and hence choose from RZ and memData. The value is stored in RZ.
- The destination is calculated using selectRC bits and the result (RY) is stored in the destination register.

4. **MA Unit**



- The **RZ** stores the address of the memory unit required for load/store operation.
- **RMout** contains the data to be stored from the RM register for store-operation.
- The **selectMuxA** signal decides if the operation is load/store or not.
- The **memLoadStore** signal decides if the operation is load or store.
- If it is a **store-operation** then RMout is stored in a memory unit with address RZ.
- If it is a **load-operation** then the value stored at RZ is fetched and stored in **memData**.

The Register File

A register file is an array of processor registers. In most computers register operations can be performed faster than corresponding memory operations. Therefore, the most heavily used local variables and temporaries are placed in registers. The registers used in Register File are processor registers which are quickly accessible locations available to a computer's processor.

The Register File

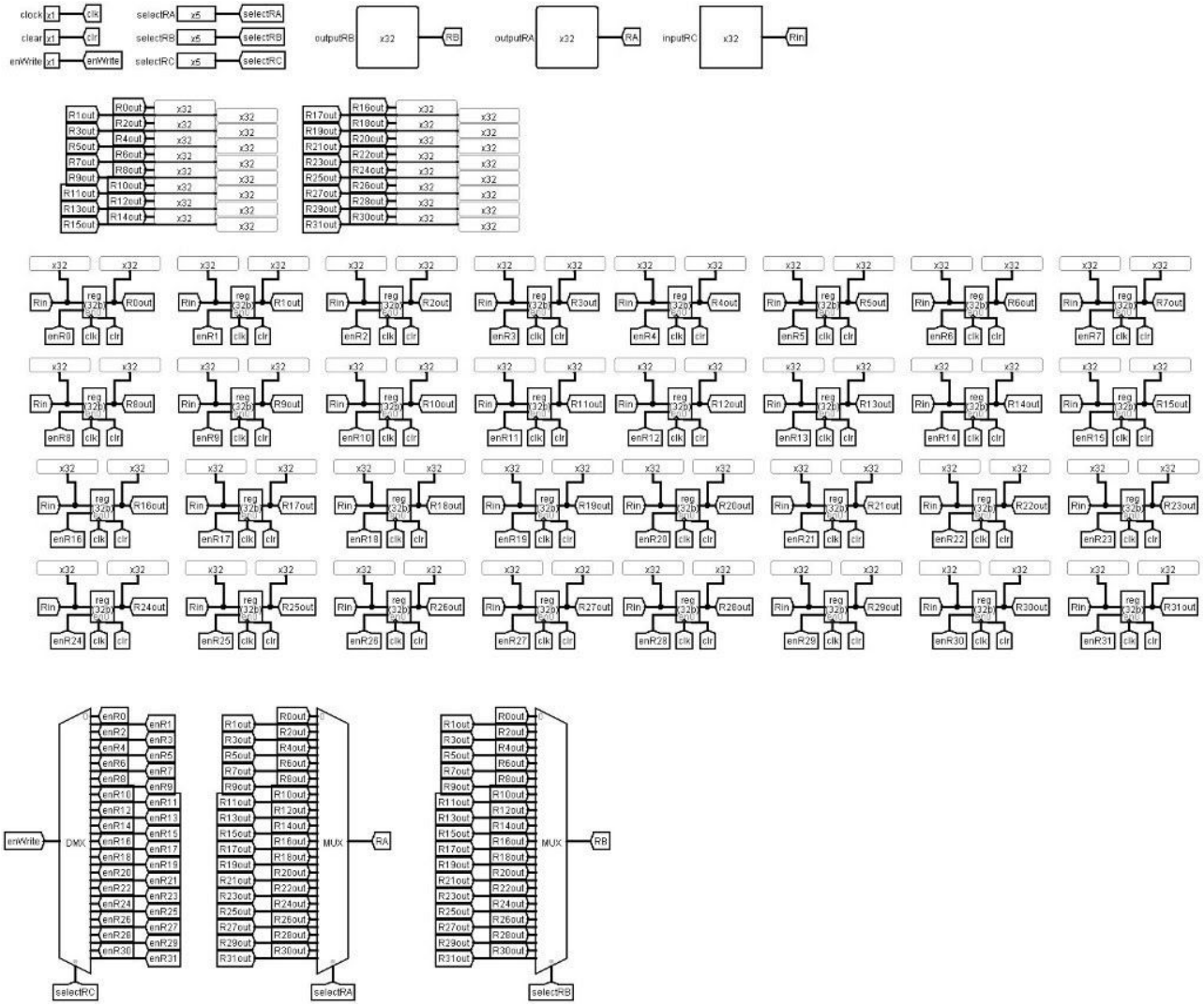
The Register File in this processor has read and write capability. It receives a set of inputs specifying whether to read or to write and in which register to perform that operation.
There are 32 registers from r0 to r31 which store information.
The register file also consists of two Multiplexers MUX A and MUX B, and a Demultiplexer DEMUX C.
MUX A selects one of the registers based on selectRA and outputs the register value. Similarly, MUX B selects one of the registers based on selectRB and outputs the register value.
DEMUX C writes the input Rin to the register specified by selectRC.

The input enWrite consists of a single bit specifying whether to perform write operation or not. The inputs selectRA, selectRB, selectRC consist of 5 bits each, specifying the address (0 to 31) of the register in the Register File.

Design of the Register File

The Register File has been designed keeping in mind the various underlying intricacies of the SimpleRISC processor.
The Register File works in close conjunction with the Arithmetic and Logic Unit (ALU) and the register storage can be read and written based on the calls of ALU.
Logisim has been used to design the Register File as it provides a simple interface for efficient designing.

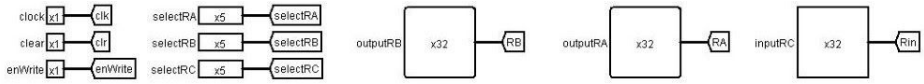
Final Design of the Register File



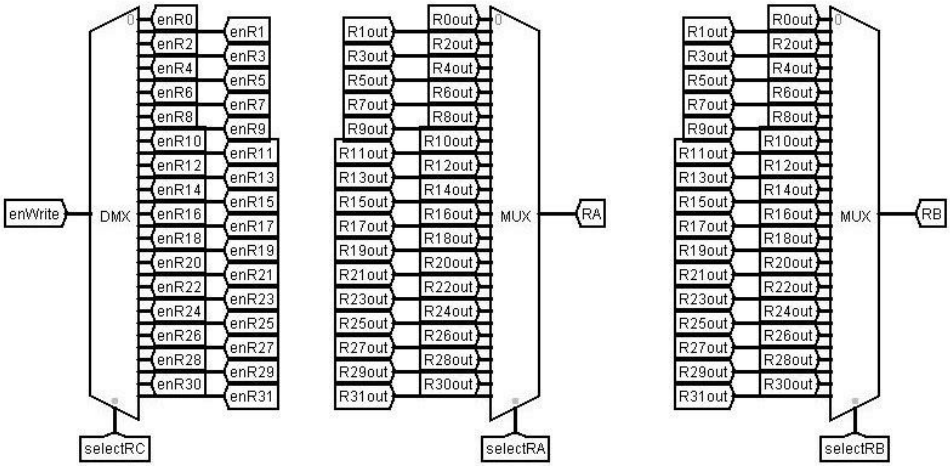
As we can observe in the final design that all the components essential to the Register File have been implemented. Let's go over each of the components for a detailed understanding.

Input / Output

(a)



(b)



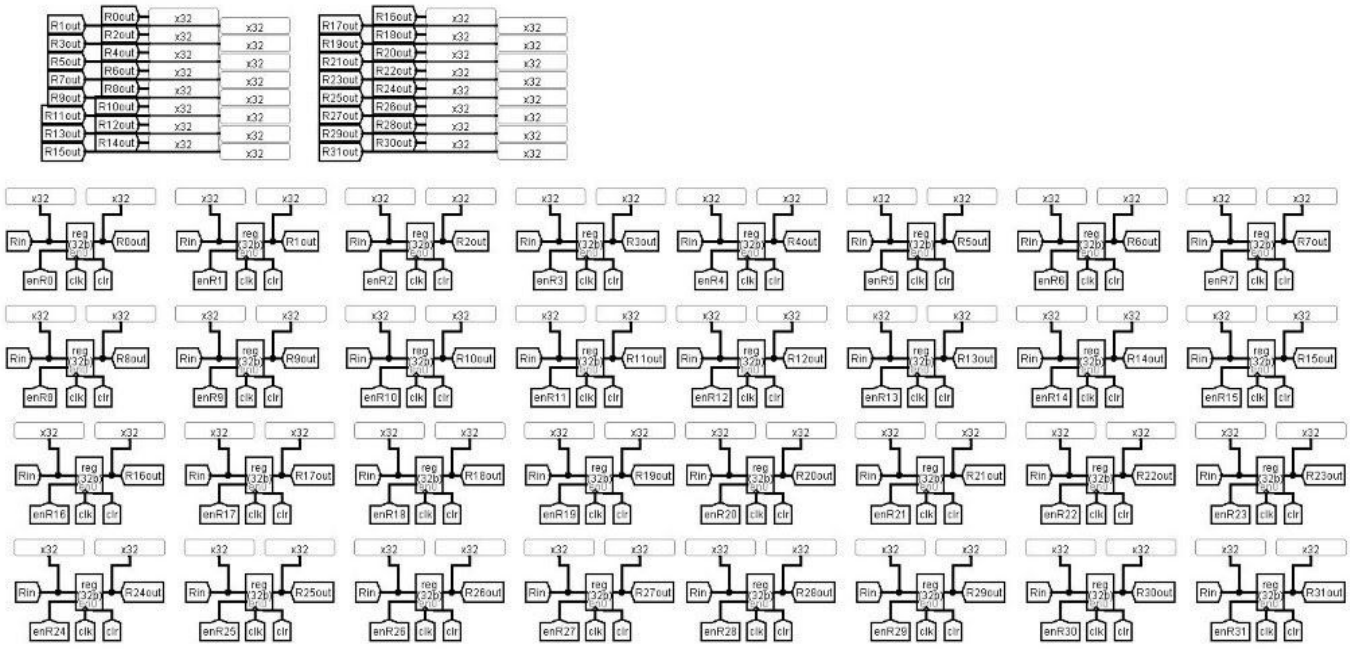
The Inputs to the Register File are

1. **Clock**: the clock signal
2. **Clear**: the clear input
3. **EnWrite**: enable write to the register
4. **SelectRA**: select register A
5. **SelectRB**: select register B
6. **SelectRC**: select register C
7. **Rin**: the value that will be stored in a register in write operation.

The Outputs from the Register File are

1. **OutputRA**: value stored in register A
2. **OutputRB**: value stored in register B

The Registers



The Registers are the fundamental components of the Register File.
In the above design, the upper part represents the register bank defining the 32 registers that have been implemented (from r0 to r31).
The lower part represents all the 32 registers from r0 to r31 and their defining attributes, i.e., their input, stored value and output.

Working of the Register File

The workflow is as follows

(a) When data from RA and RB is to be read

i. On the next clock edge when selectRA and selectRB are enabled
ii. selectRA goes as the select input to MUX A and selectRB goes as the select input to MUX B

iii. The corresponding register (from r0 to r31) is then selected from the respective MUX and the value stored in that register reaches to the output of the MUX
iv. The output of MUX A is outputRA and the output of MUX B is output RB

(b) When data is to be written to RC

i. On the next clock edge when selectRC and enWrite are enabled
ii. SelectRC goes as the select input to the DMUX C with enWrite as the input
iii. The corresponding register (from r0 to r31) is then selected and Rin reaches the input of that register
iv. Rin is then stored in that register

The Control Unit

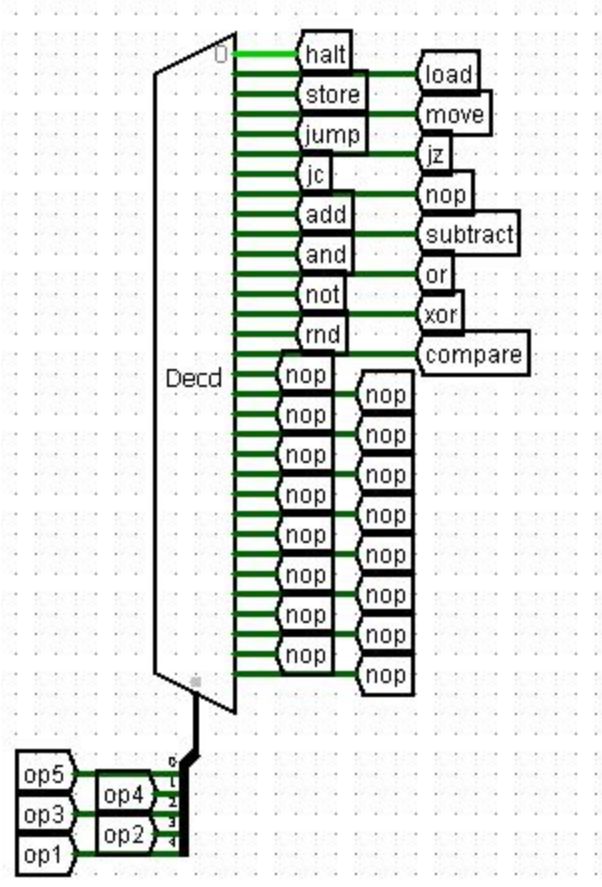
Most these wires/pins have been already explained while some are described below:

- **Clock:** This is how the control unit “keeps time.” The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time or the clock cycle time.
- **Instruction Register:** The opcode and addressing mode of current instruction concludes the micro-operation to be performed during EX-Cycle.
- **Flags:** Flags are to determine the state of the processor and the outcome of previous ALU operations.

The outputs of the control unit are the **Control Signals** within the processor.

The Decoder

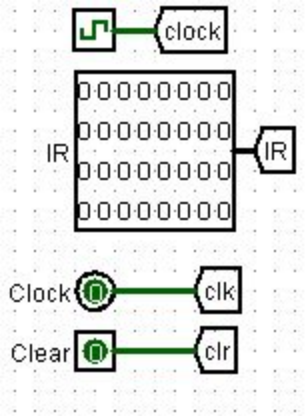
1. The instruction has an op-code of 5 bits.
2. These 5 op-code bits are put



into a MUX which generates Control Signals

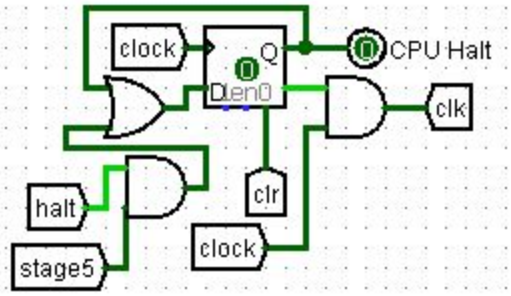
3. Shown here, are halt, load, store, etc. control signals

The Clock, Clear and Instruction Register

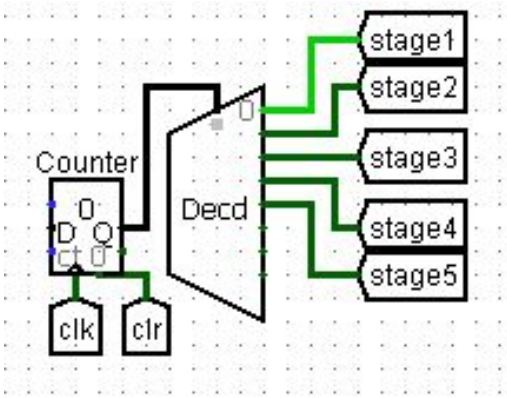


The Clock and clear pins are integral in the Control and Execution part of the processor.
We are using a falling edge triggered D-flip-flop as we aim to

1. Reduce active power consumption
2. Data-lockout issue
3. Multi-Transition Problem

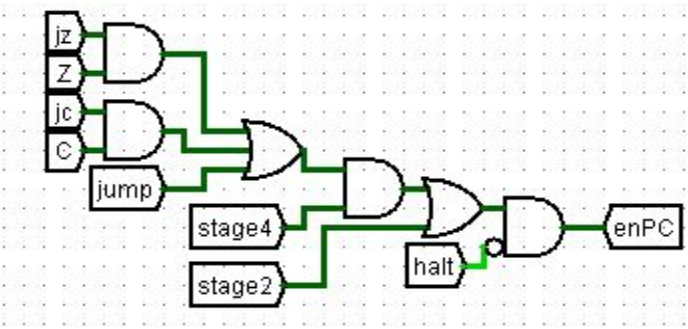


The D-flop counter here attached with the clk and clr pins, when inserted into the decoder gives us 5 stages executing an instruction.
These 5 stages (fetch decode execute read write) have been already explained and these stages involve:

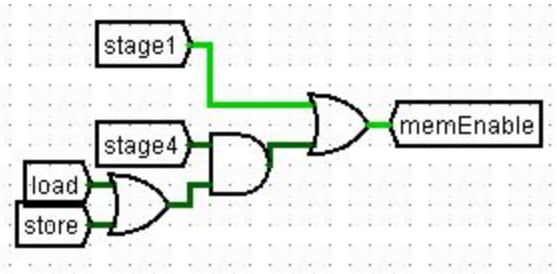


- **selectMuxA**
- **selectMuxY (has load/store op)**
- **enableIR(instruction register)**
- **selectMuxINC**
- **selectMuxPC**
- **‘enImmediate’ = selectMuxB**

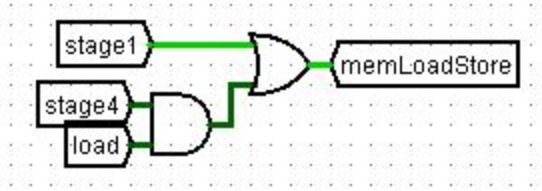
- **enRWrite**
- **selectRA, selectRB and selectRC**
- **enableFlags, memLoadStore and memEnable**



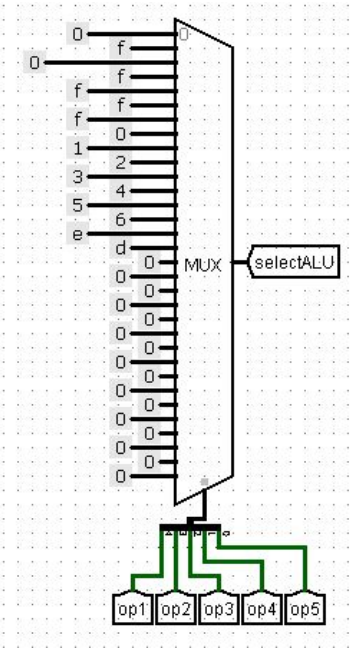
Shown besides, is a gate level implementation generates **enPC** control signal; during Instruction Address generation, it controls whether to enable **Program Counter** and process to the next instruction. We can easily understand the inputs required for the flag generation.



The Control Signal generated besides is **memEnable**; This control signal defines whether direct memory access is allowed for load or store operations or not.

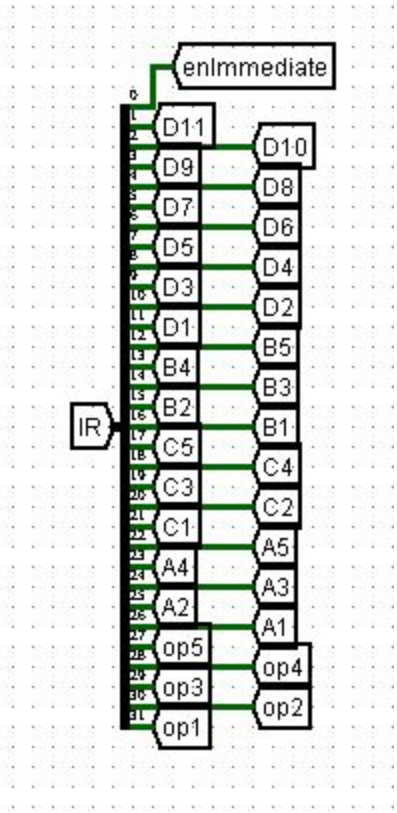


The Control Signal generated above is **memLoadStore**



selectALU: 4 bit wide wires present to communicate across the CU and the ALU, which is passed on into the MUX of ALU to choose the desired operation result/output.

ALU_Control: It is made up of clk, clr, C(in), selectALU and enableFlag wires and is a 8 bit wide wire showing unchanged pull behaviour.



The most important OpCode, address regs, Immediate signal all, are insourced from the Instruction Register.

OpCode has 5 bits

Addresses: each register location occupies 5 bits.

Immediate: has 16 bits reserved for it overlapping with the 5 bits occupied by the 3rd address location

Enable bit for immediate value.

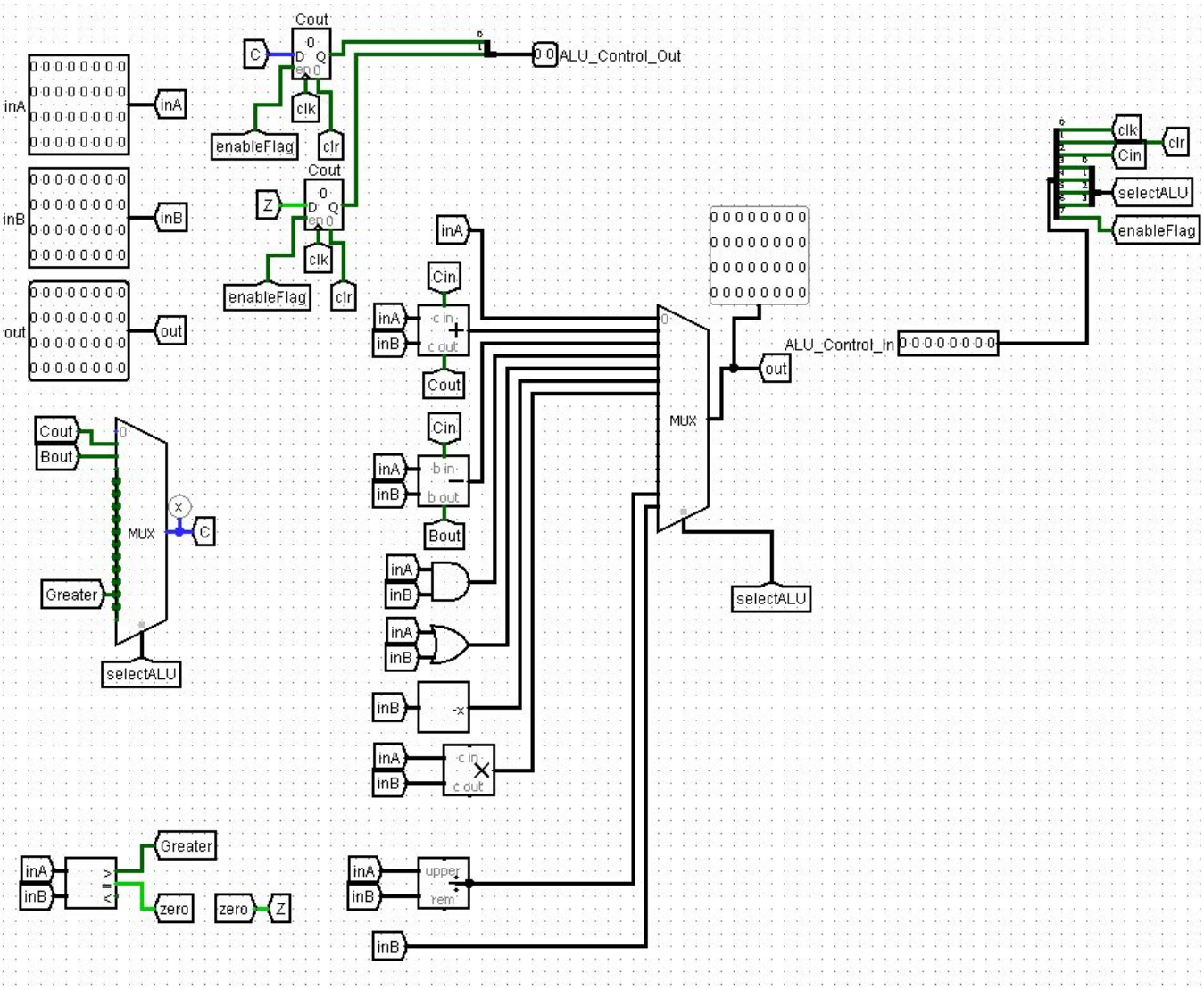
Calculation for instruction extraction:

5(opcode) + 5 + 5(2 registers) + 16 + 1 (immediate and enable) = 32bits instructions from IR (Instruction Register).

ALU

ALU is used for performing arithmetic, bitwise or logical operation in CPU.

It is one of the main components of the CPU apart from the Control Unit.



Inside ALU, there are 3 registers involved (in general):

1. **inA** :- This register contains the first operand and contains 32 bits (not present in case of negation operation).
2. **inB** :- This register contains the second operand and contains 32

bits.

3. **out** :- This register stores the value of the final result and contains 32 bits.

Apart from these 3 registers, we also have a clock signal and various other control signals.

1. **enableFlag** :- It contains one data bit which is 1 if we have to perform some operation.
2. **selectALU** :- It contains 4 data bits which represents the type of operation to be performed in ALU (like addition, subtraction, etc).
3. **Cin** :- It contains 1 data bit which represents whether we have carry or not in input.
4. **Cout** :- It contains 1 data bit which represents whether we have carry or not in the result.
5. **Clr** :- It is 1 if we have to reset all the registers else 0.
6. **ALU_Control_in** :- It is of 8 bits which is a combination of (1 clk bit + 1clr bit + 1 Cin bit + 4 selectALU bit + 1 enableFlag bit).
7. **C, Z** :- If operation is of logical type then these will be set according to the result. Like if two elements are equal then Z is set to 1 and C to 0. And if the first operand is greater than the second operand then C to 1 and Z to 0.

Some of the operations supported in ALU are :-

1. **Addition** :- It involves three inputs (i.e. inA, inB, Cin) and two outputs (i.e. out, Cout). Out is of size 32 bits which gives the addition of inA and inB along with Cin (if it is 1) and out is of size 1 bit and contains

output carry bit.

2. **Subtraction** :- It involves three inputs (i.e. inA, inB, Cin) and two outputs (i.e. out, Cout). Out is of size 32 bits which gives the subtraction of inA and inB along with Cin (if it is 1) and out is of size 1 bit and contains output carry bit.
3. **Multiplication** :- It involves two inputs (i.e. inA, inB) and one output (i.e. out). Out is of size 32 bits which gives the multiplication of inA and inB.
4. **Division :-** It involves two inputs (i.e. inA, inB) and one output (i.e. out). Out is of size 32 bits which gives the division of inA and inB.
5. **AND** :- It involves two inputs (i.e. inA, inB) and one output (i.e. out). Out is of size 32 bits which gives the (inA & inB).
6. **OR** :- It involves two inputs (i.e. inA, inB) and one output (i.e. out). Out is of size 32 bits which gives the (inA | inB).
7. **NOT** :- It involves one input (i.e. inB) and one output (i.e. out). Out is of size 32 bits which gives the (~inB).
8. **Compare** :- It involves two inputs (i.e. inA, inB) and compares them and sets the value of C, Z flags accordingly.

Example Programs

1. Example Program 1

```
# Add first 10 natural numbers and store in r1
movi r1 0x0
movi r2 0xa
add r1 r1 r2
subi r2 r2 0x1
cmpi r2 0x0
jci 0x2
```

2. Example program 2

```
# Put r1 = n for calculating n! and store it in r2
movi r1 0x5
movi r2 0x1
cmpi r1 0x0
jzi 0x9
jumpi 0x7
mul r2 r2 r1
subi r1 r1 0x1
cmpi r1 0x1
jci 0x5
```

3. Example program 3

```
# divide two numbers
movi r1 0x2
movi r2 0x6
div r3 r2 r1
```

Testing and Software

Features of the Platform:

- One-touch-away windows software
- 3 themes for easing user interface
- Easy to use and user friendly
- Designed using batch-file scripting
- Uses Logisim for the simulation of the processor circuit

Testing on the Platform:

- Earlier testing was done manually
- We have semi-automated the process
- Use of logging-simulation mode of Logisim

Innovation: Logisim provides a logging feature along with its 'poke-a-wire' functionality. If the states of the register file are properly documented, we can find the errors in the program, if any.

The RISC-R Software comes with some basic test cases for a given example program.

Results and Discussion

- The processor is designed to execute every instruction in 5 stages and every stage takes 1 clock cycle to execute.
- The processor is positive edge triggered by design.
- So for complete execution of 1 instruction it will take 5 clock cycles.

Individual Contributions

Aditya Rai – 19114004

1. Worked on Control Unit testing and implementation of various flags
2. Mobilised resources for implementation and helped in report compilation.

Gagan Sharma – 19114032

1. Made the main circuit of the CPU.
2. Made the report for the main CPU design.
3. Example programs
4. Made Assembler

Gajanan Gitte – 19114033

1. Made and designed the Control Unit and Instruction Register of the Processor
2. Wrote and designed the benchmark tests for RISC-R
3. Deployed the RISC-R application on Windows platform
4. Coded and Designed the .exe library for RISC, design ready for linux platform
5. Compiled report for the Control Unit part

Raghav Somani – 19114068

1. Made memory access circuit
2. Designed ISA
3. Documentation of ISA and Assembler
4. Designed Assembler

Shlok Goyal – 19114078

1. Made the design for the ALU in *Logisim*.
2. Made the report for the ALU part.
3. *Done the testing of ALU components.*
4. *Integrate ALU with other components.*

Gaurav Wasnik – 19114090

1. Made the design for the Register File in *Logisim*
2. Made the report for Register File
3. Ensured the proper working of Register File and related inputs/outputs

References

1. http://www.cse.iitd.ernet.in/~srsarangi/archbooksoft.html
2. Computer Organisation and Architecture (By Smruti Ranjan Sarangi)
3. The Guide to Being a Logisim User (cburch.com)
4. RiSC-ISA (umd.edu) A 16-bit architecture