COA Project

DESIGNING AND EXECUTION

Team Members

1. Ansh Jain 19ucs059
2. Gaurav Niranjan 19ucs054
3. Neeraj Kumar Singhal 19ucs029
4. Aditya Raj 19ucs025

INSTRUCTION

The instruction is of 16 bits. Since the word length is 8 bits so the instruction is read from the memory in two parts. The first part reads 0 to 7 bits of instruction, the second part reads the 8 to 15 bits of instruction.

Our instruction is divided into 4 parts

0-3 bit Opcode

4th bit to decide the next 3 bit is register address or an immediate value

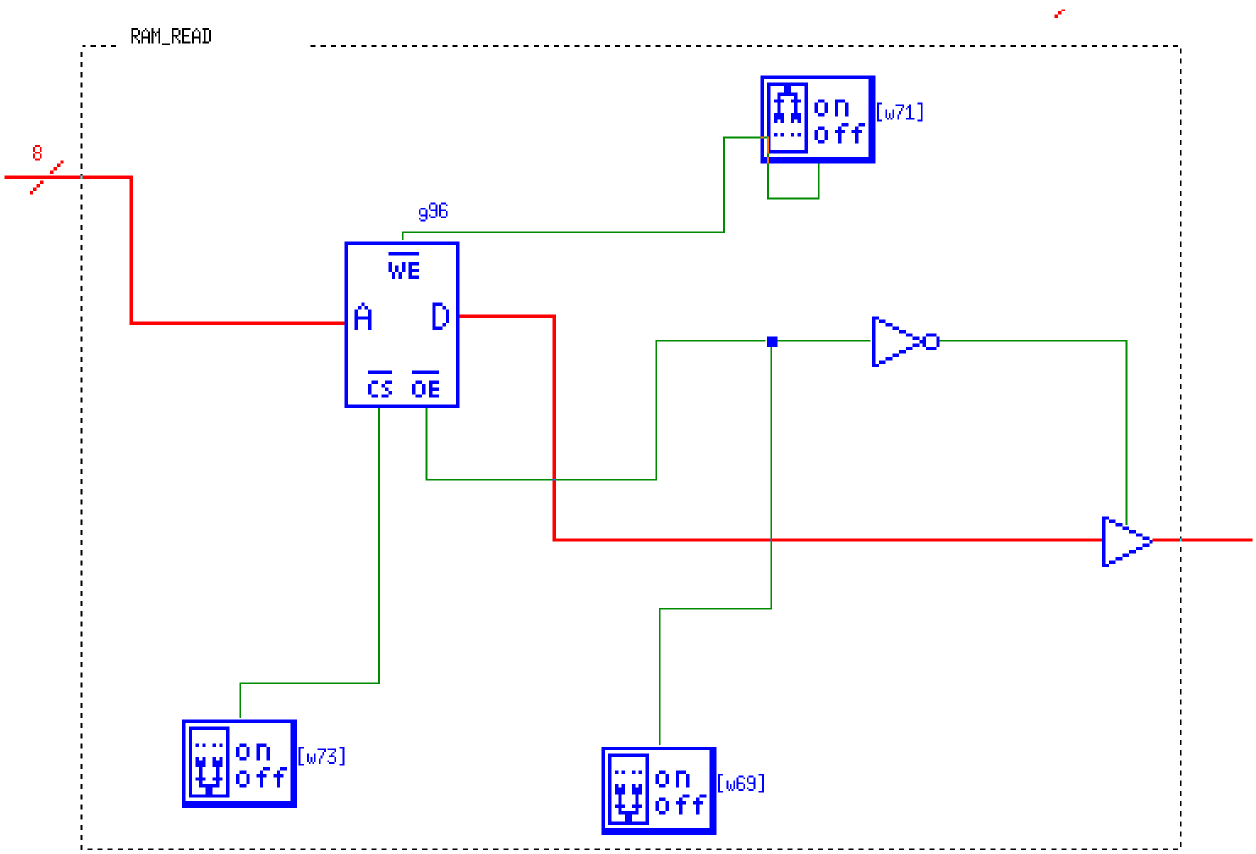
5-7 bit for register address or an immediate value

8-15 bit memory address

INSTRUCTION READ AND WRITE

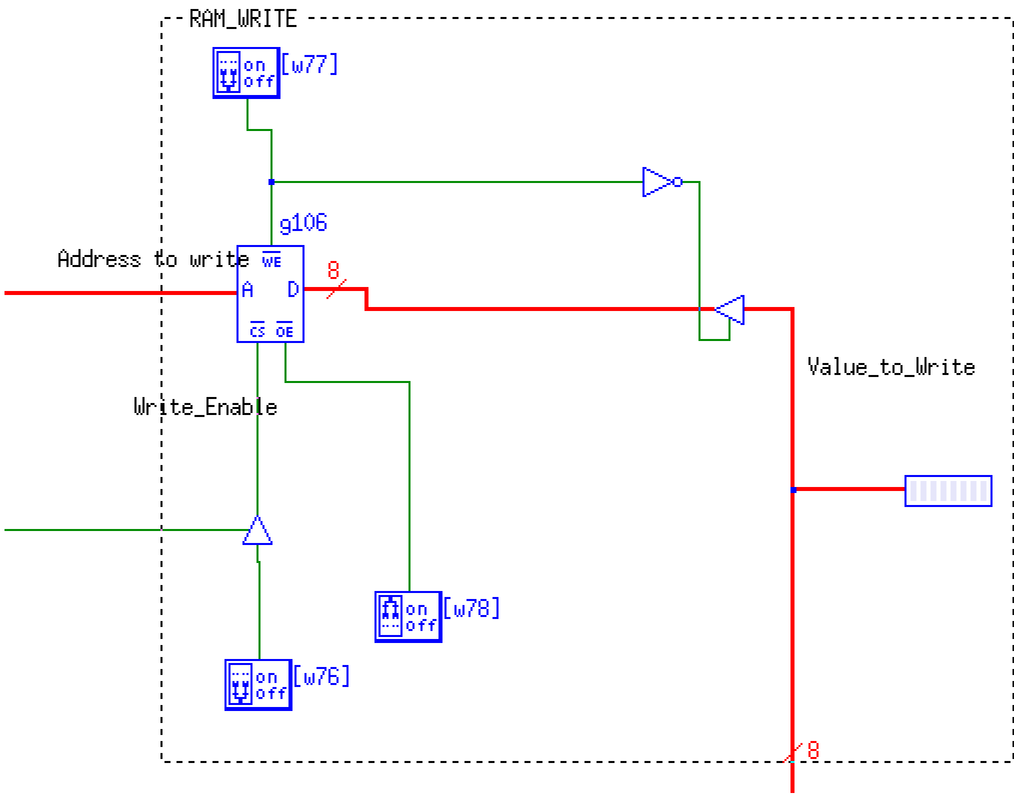
Instruction is read through RAM. Memory file (.mem) is loaded into the RAM. The address to be read is given by the Program counter module.

In the case of read, Chip Select is always enabled, OE is enabled and WE is disabled.



The data to be stored in memory is passed through the RAM with the memory address where it has to be stored. RAM Write is enabled when we have to store a value in memory.

In the case of Write, Chip Select is always enabled, OE is disabled and WE is enabled.



PROGRAM COUNTER

The Program Counter module stores the next instruction to be read in a register.

Clear pin enabled sets the program counter 0.

In the next clock cycle there are two possibilities:

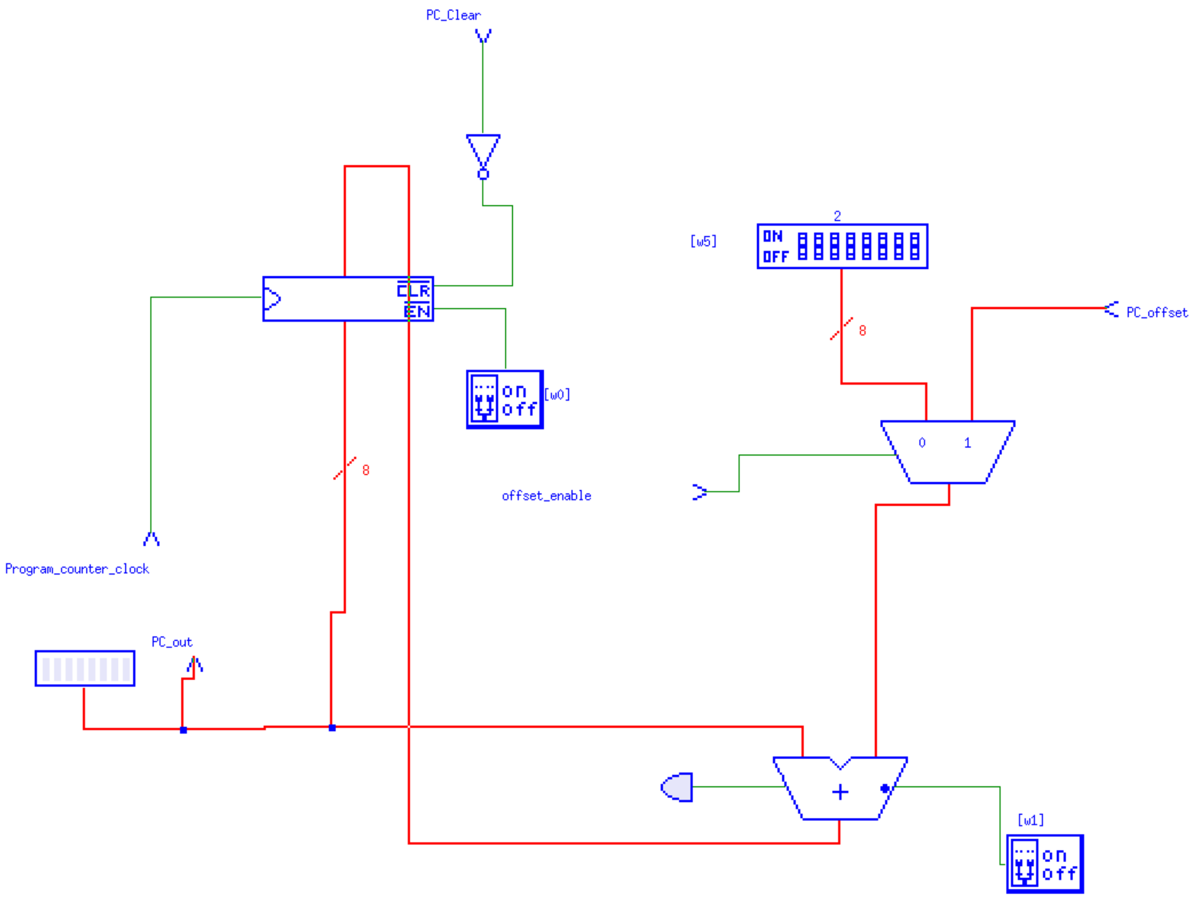
i) PC <- PC + 2: In case there is no Jump instruction and the next instruction has to be fetched and the instruction size is of 2 words.

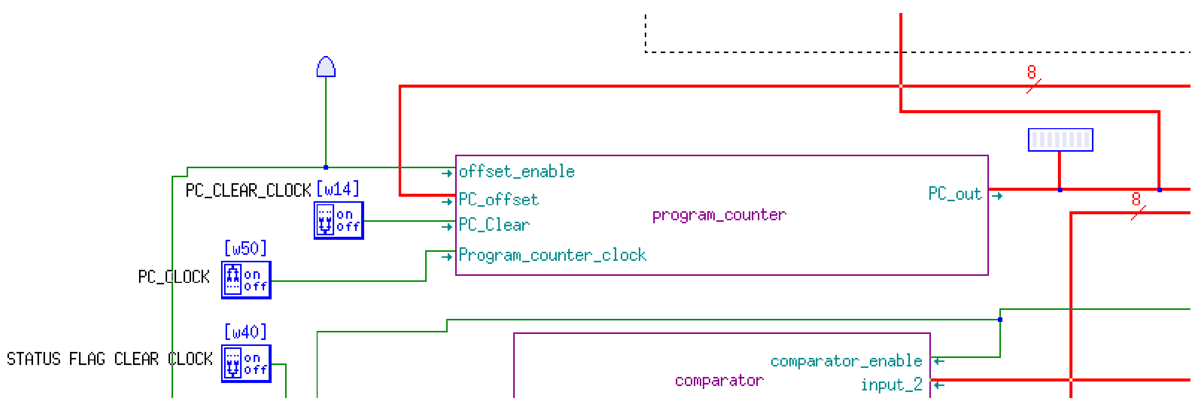
ii) PC <- PC + offset: In case there is a Jump instruction, the Jump offset is selected by the MUX and sent to the adder.

If there is JUMP instruction the offset_enable pin is enabled and the PC offset is added.

If not a jump instruction then 2 is added (Since instruction length is of 2 words) so that PC could store the address of the next instruction.

The PC clock could be used manually using a switch or we can use a clock.





INSTRUCTION DECODE

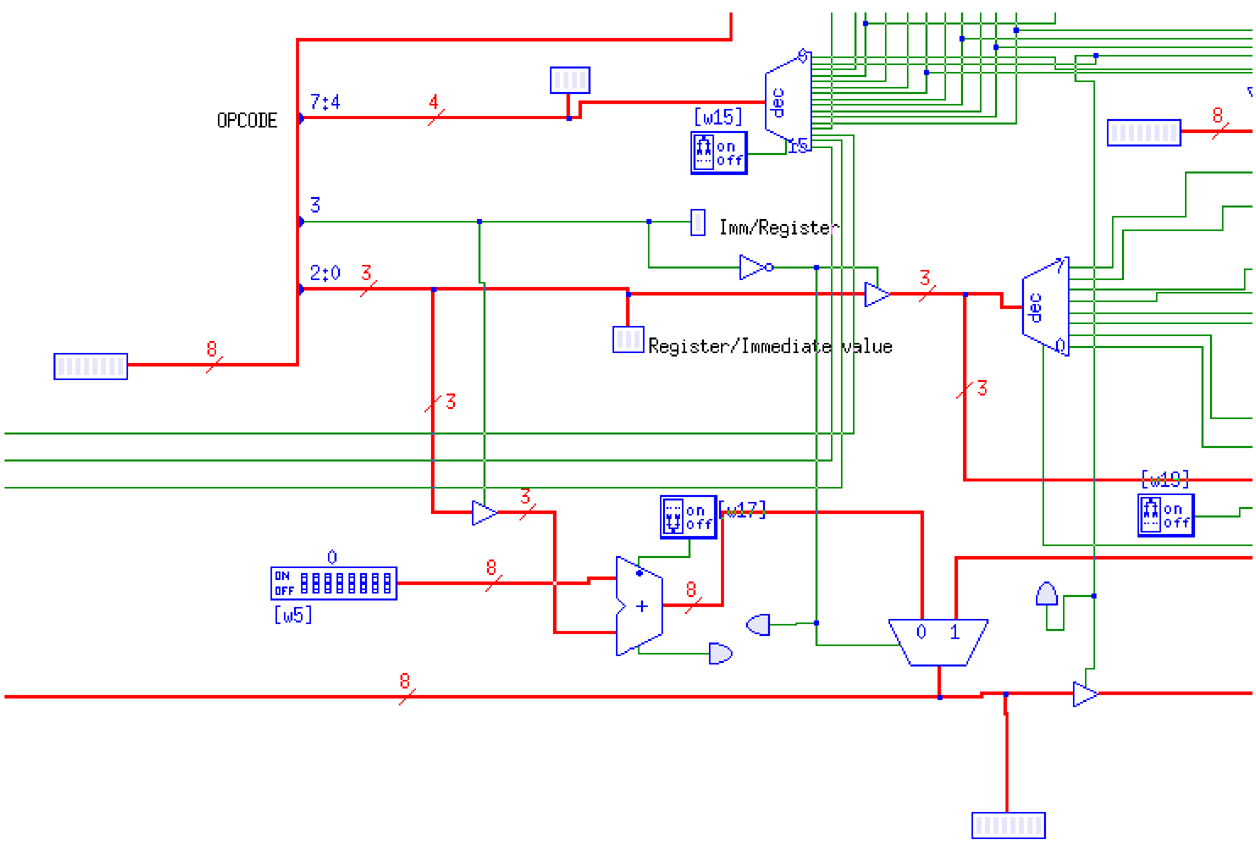
The instruction read is decoded in two parts.

The 4 bits for opcode are read and passed into the decoder. Decoder enables the pin based on the opcode. If the Instructions contain ALU operation then the enable is passed to the ALU so that the required ALU operation can be done. If the opcode is for Compare then the compare module is enabled from the decoder. If it's a jump command, then the jump module is enabled.

The 5th bit is to check whether to enable the register file or if it's an immediate value.

The next 3 bits of the instruction are read and if its Register address then it is passed to the Register file. If its immediate value then it is passed to the ALU as input for the operation.

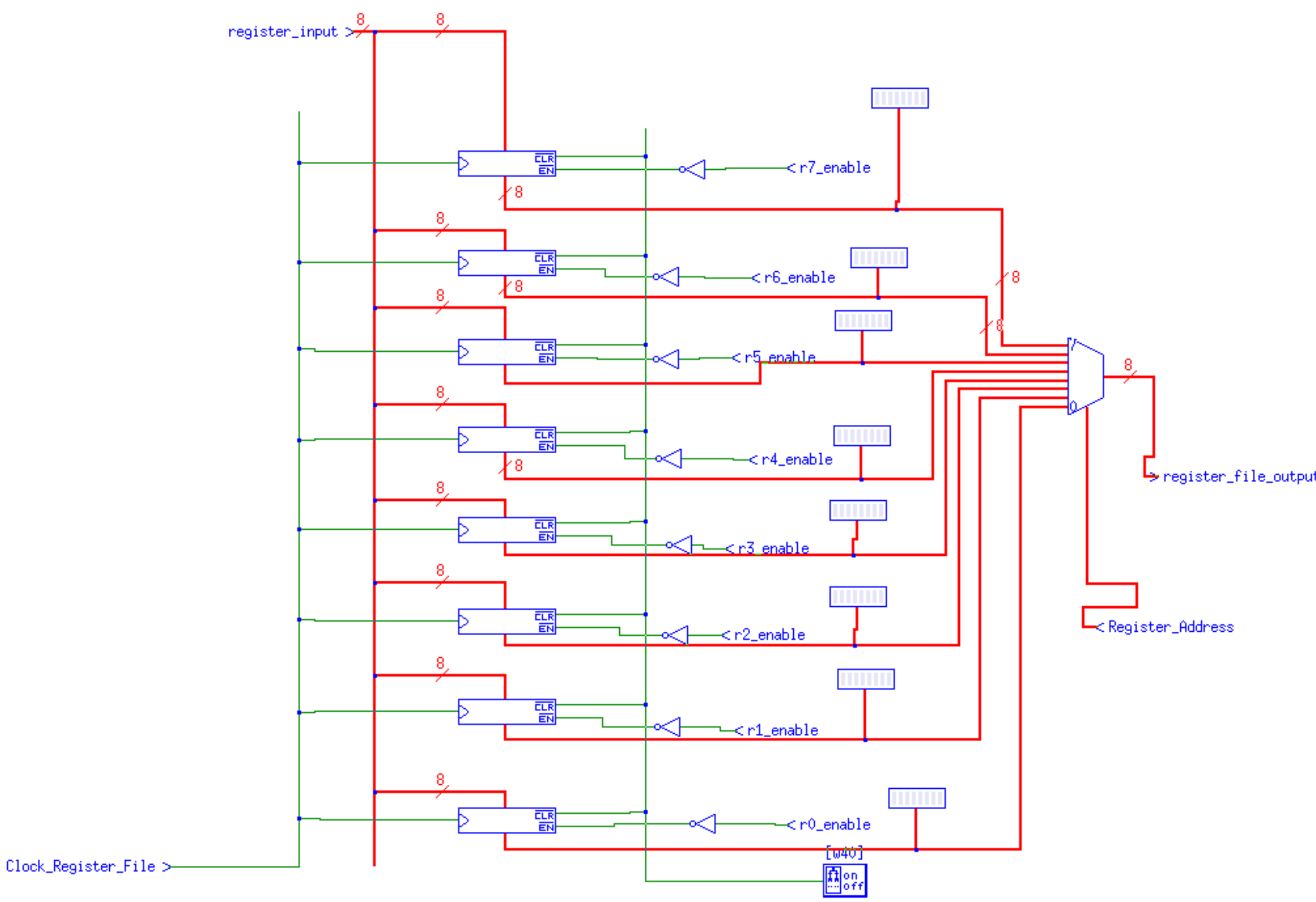
If it's an immediate value of 3 bits, 0 is added to the immediate value using an adder of 8 bits to change the number of bits of immediate value from 3 to 8.



Register File

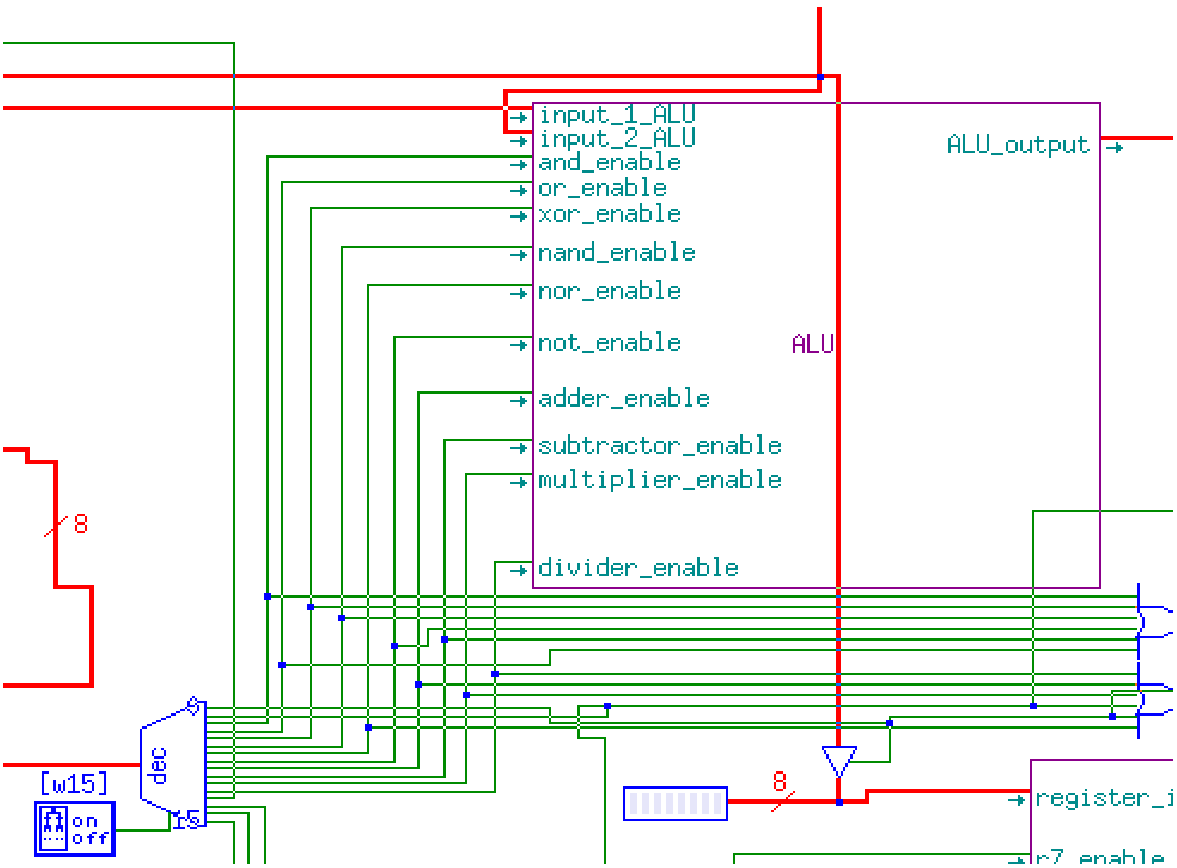
Register File has 8 registers. Every register has its own enable pin. The address of the register in the instruction determines which register will be enabled.

The data in the register as input comes from the register_input line and enters the register which is enabled. The MUX selects the desired register output based on the register address. The clock is taken outside this register file module to control the changes in each register.

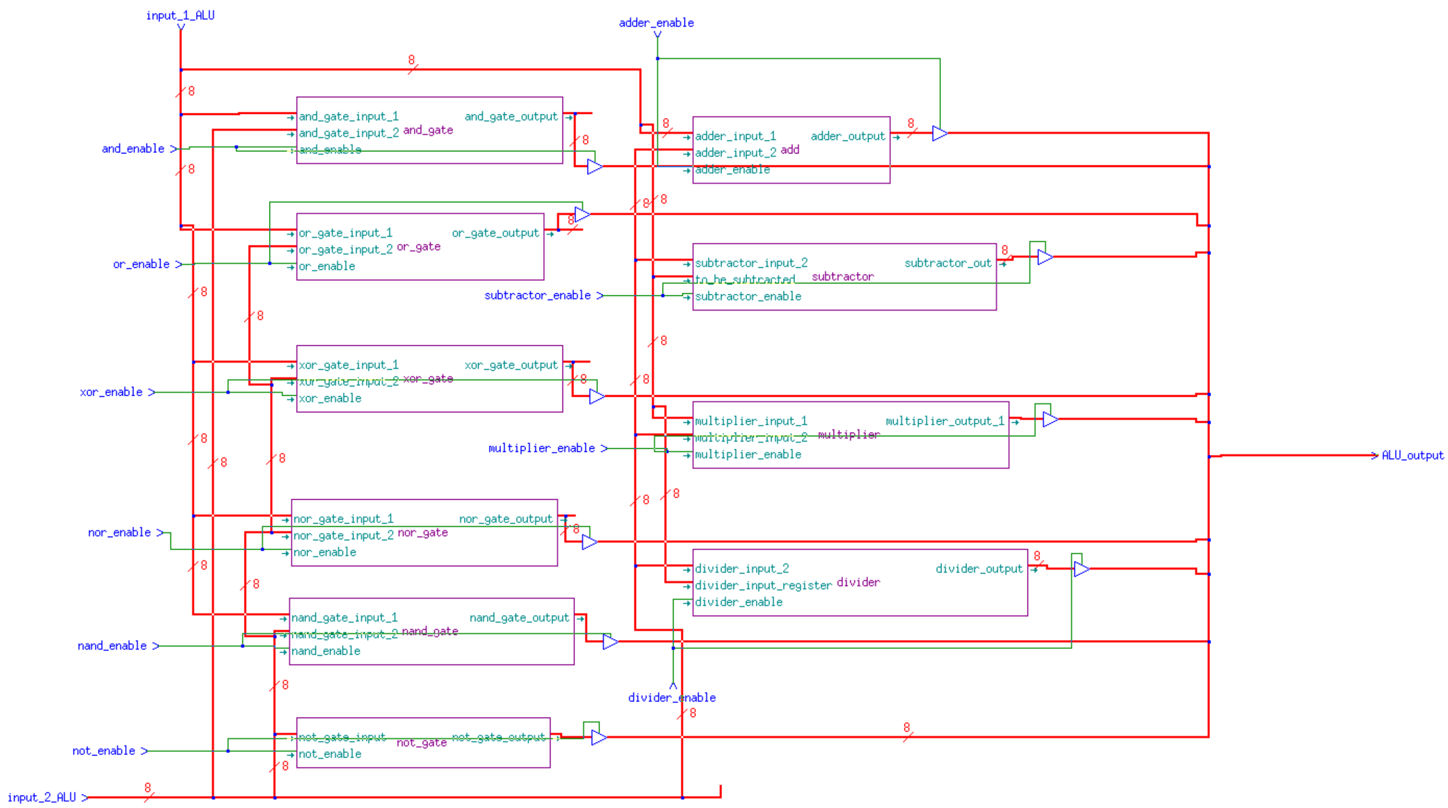


ALU

The ALU is made as a module consisting of several other modules each performing its task like adding, AND operation, multiplication, etc.
Each module in the ALU has its own enable pin. The two input data are passed to all the ALU modules but the operation will only occur in the module which was enabled based on the OPCode decoded in the decode phase.
The output of ALU operation is taken by the ALU_output line to store in memory.



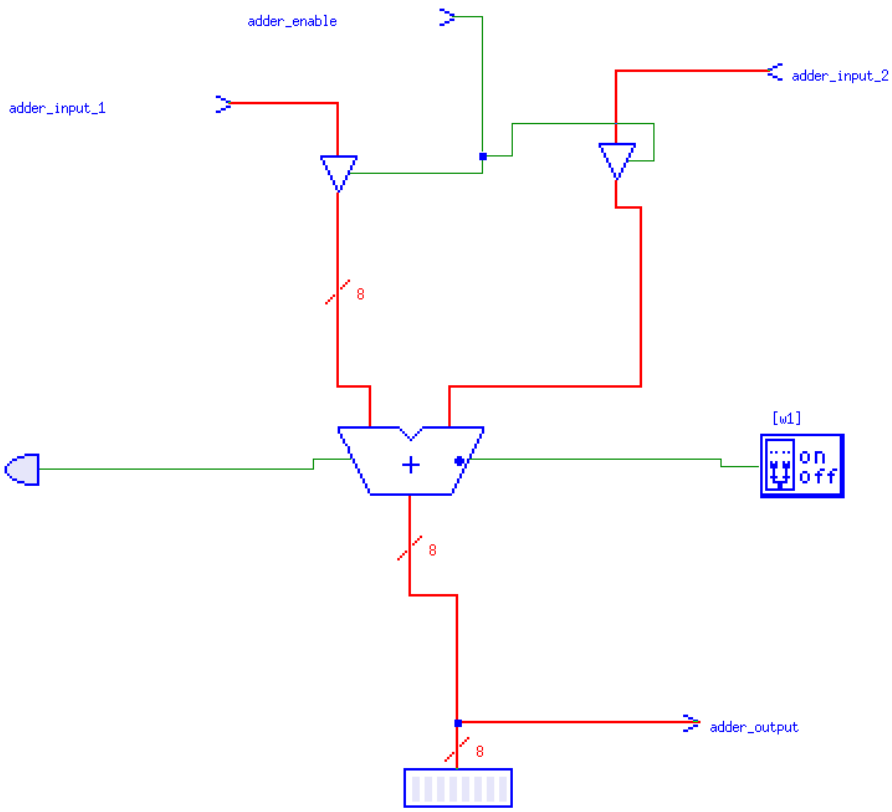
ALU Module:



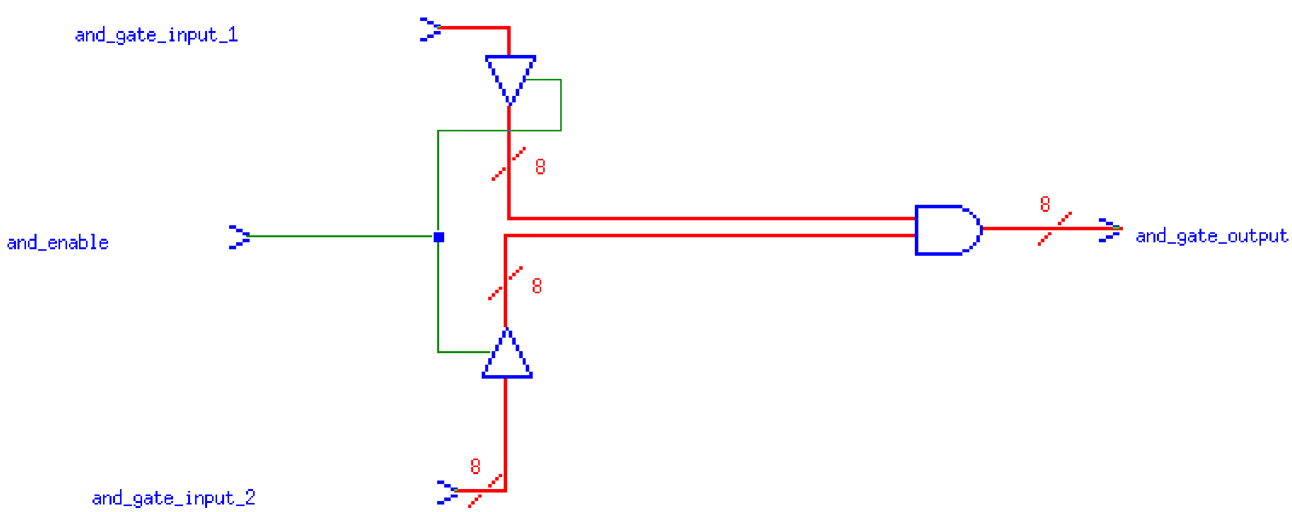
ALU Operations -

1. ADDER
2. AND
3. OR
4. NOR
5. NOT
6. NAND
7. XOR
8. SUBTRACTOR
9. MULTIPLIER
10. DIVIDER

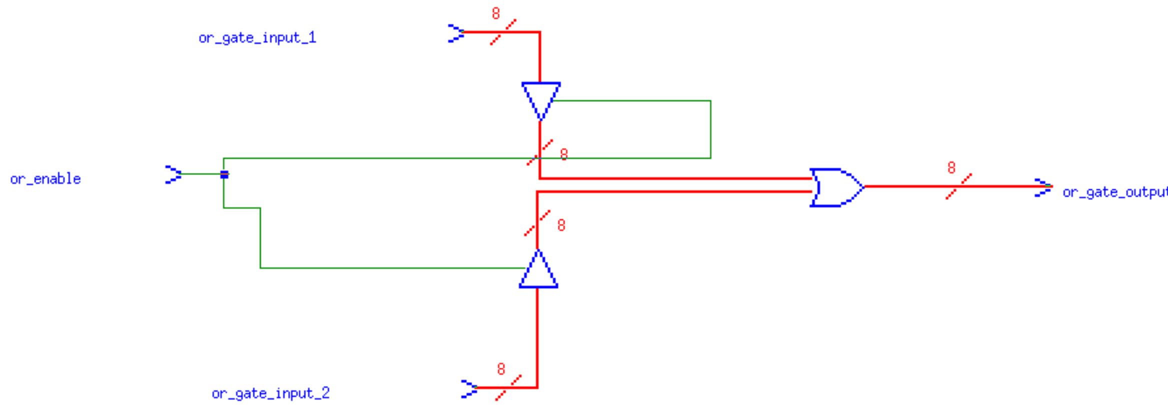
ADDER


adder_enable
adder_input_1
adder_input_2
8
[w1]
on
off
8
adder_output
8

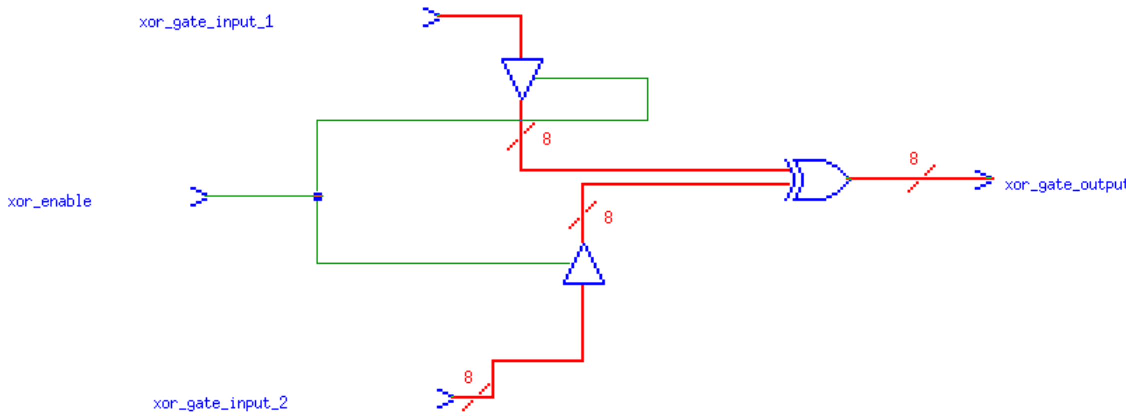
AND


and_gate_input_1
8
and_enable
8
and_gate_output
8
8
and_gate_input_2

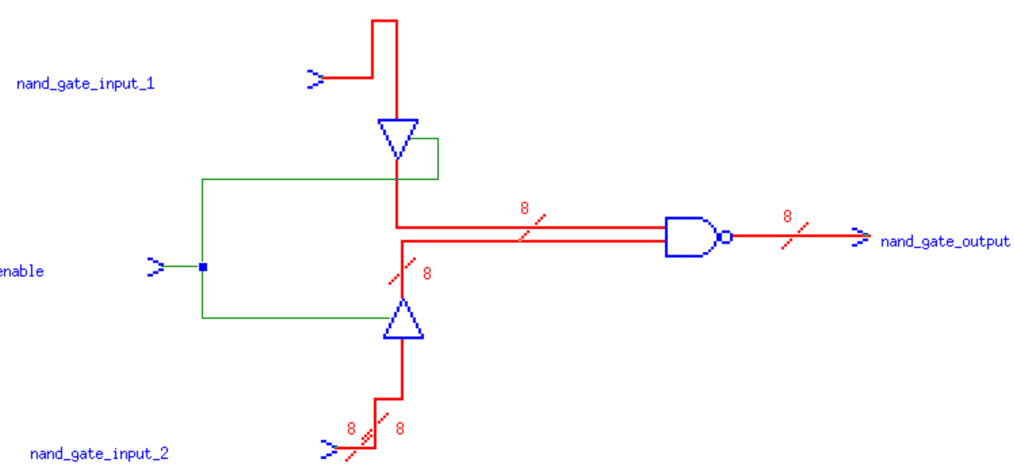
OR


or_gate_input_1
or_enable
or_gate_output
or_gate_input_2
8

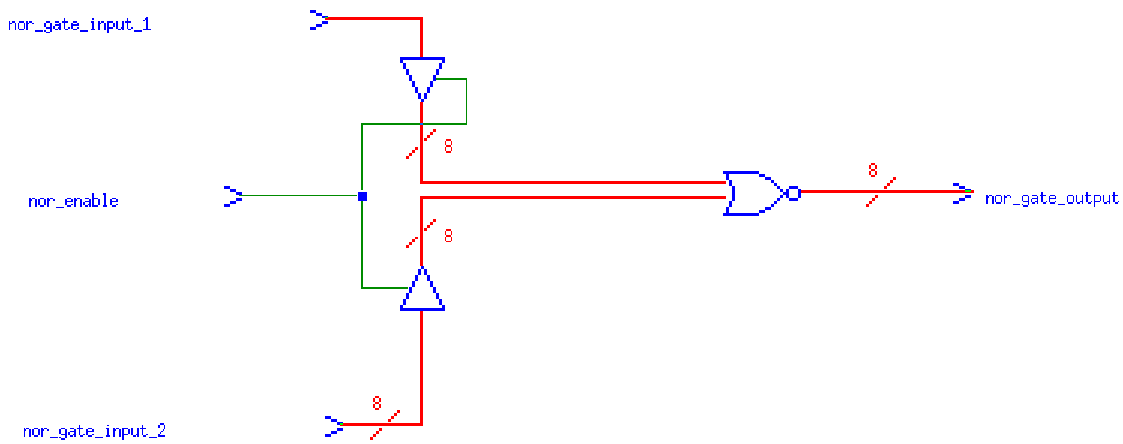
XOR


xor_gate_input_1
xor_enable
xor_gate_output
xor_gate_input_2
8

NAND

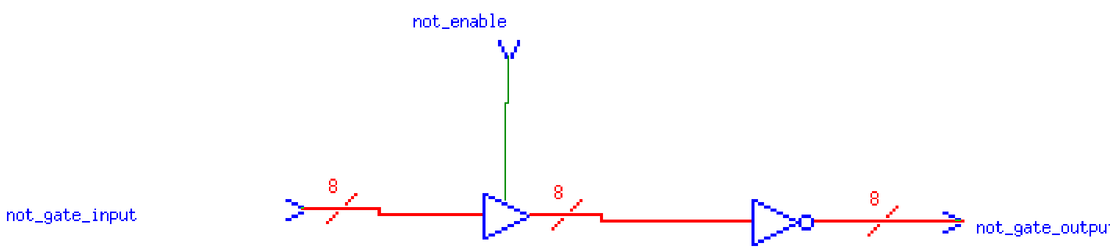

nand_gate_input_1
nand_enable
nand_gate_output
nand_gate_input_2
8

NOR



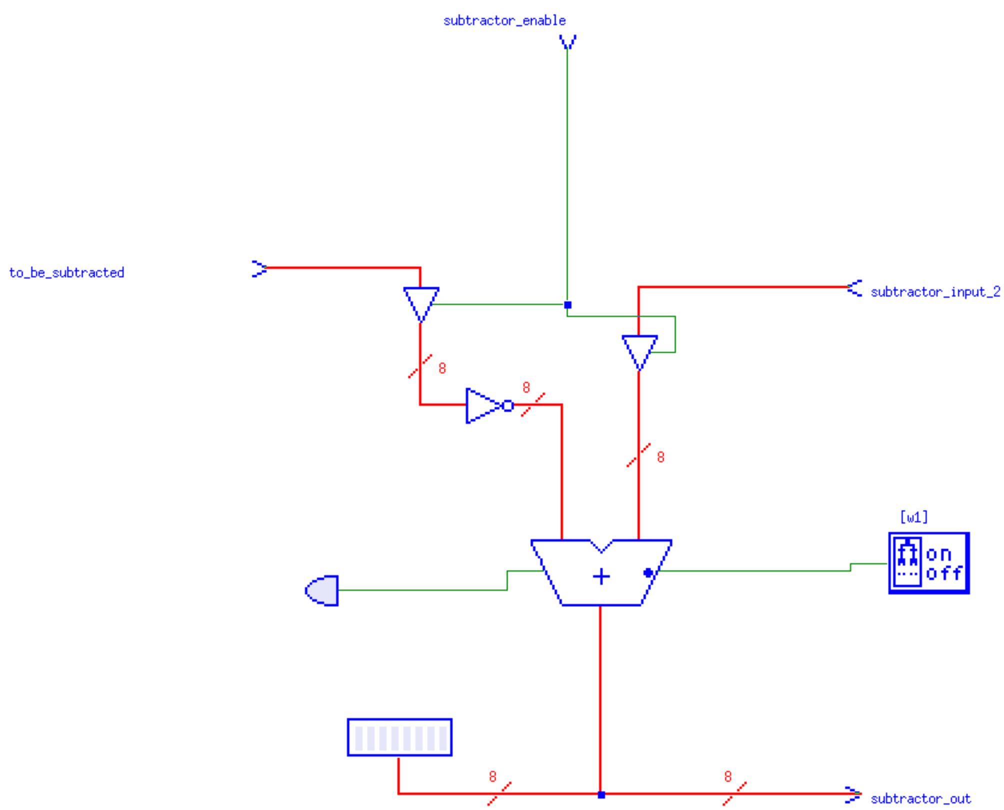
nor_gate_input_1
8
nor_enable
8
8
nor_gate_output
8
nor_gate_input_2

NOT

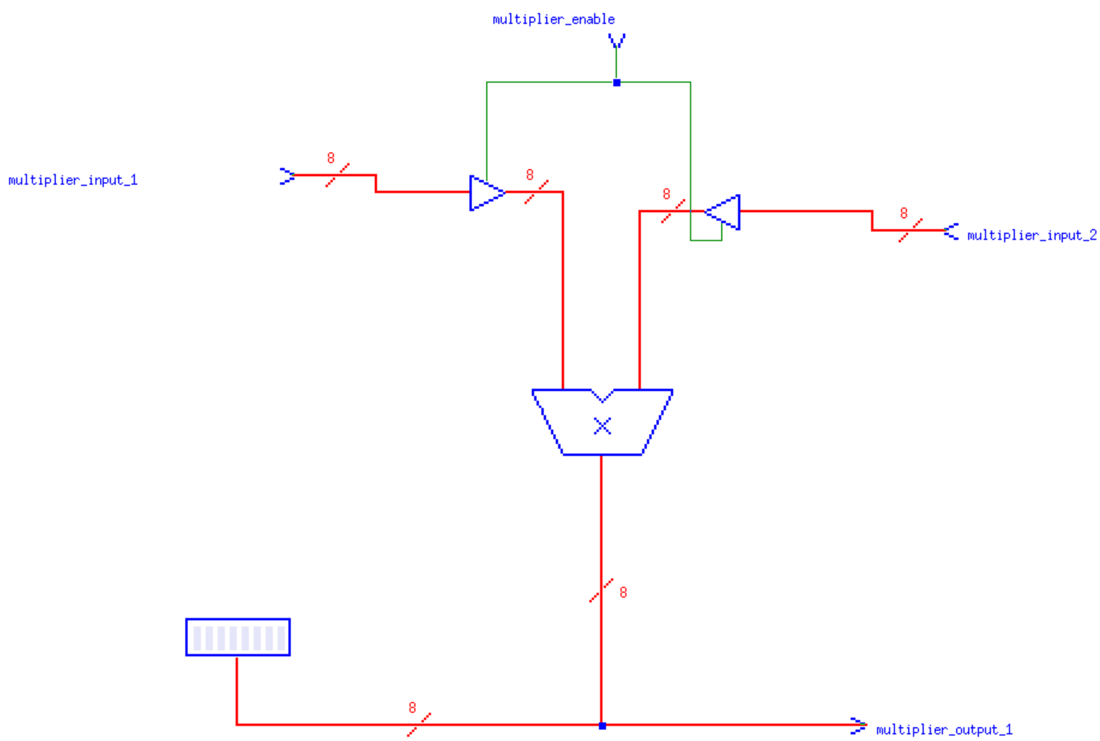


not_enable
8
8
8
not_gate_input
not_gate_output

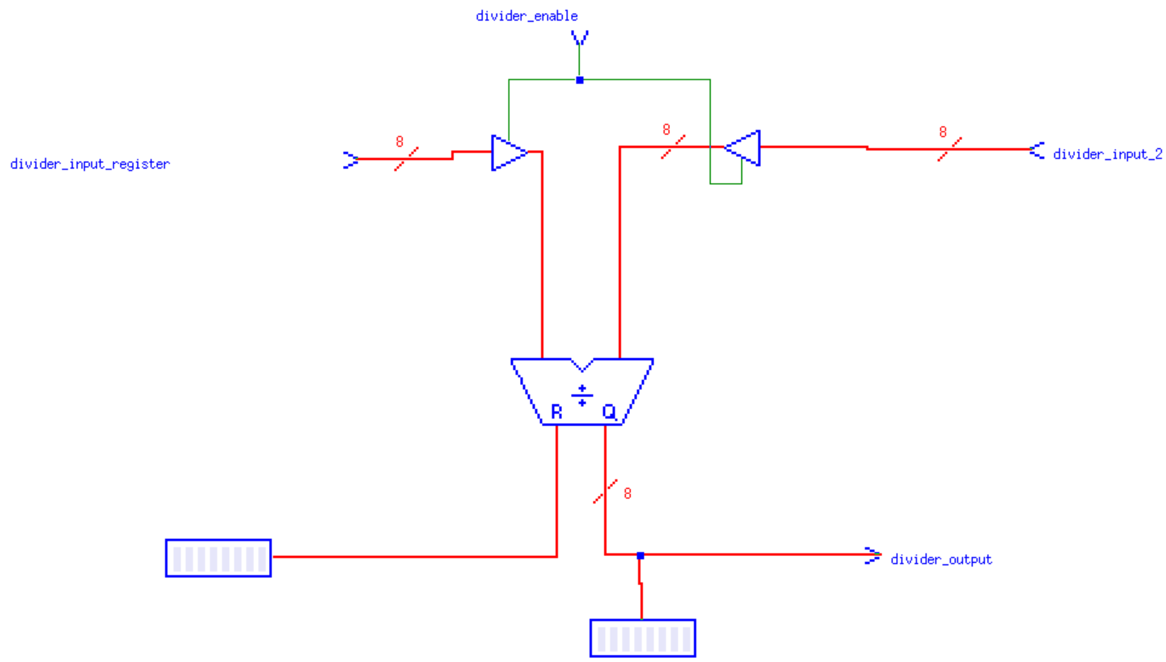
SUBTRACTOR



MULTIPLIER



DIVIDER

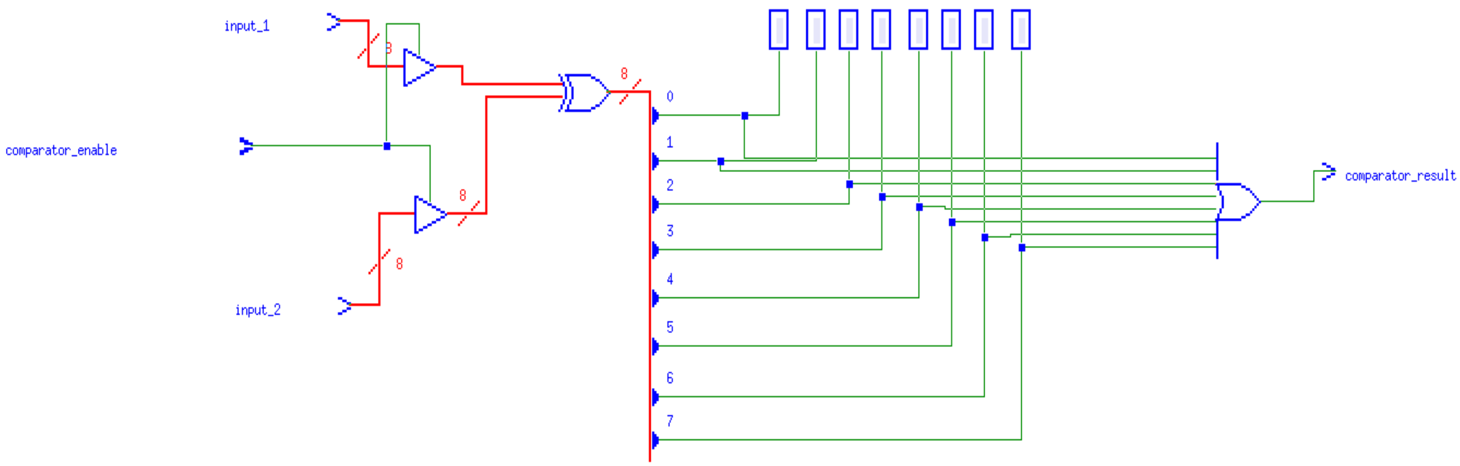


divider_enable
divider_input_register
8
8
8
divider_input_2
R
Q
8
divider_output

Comparator

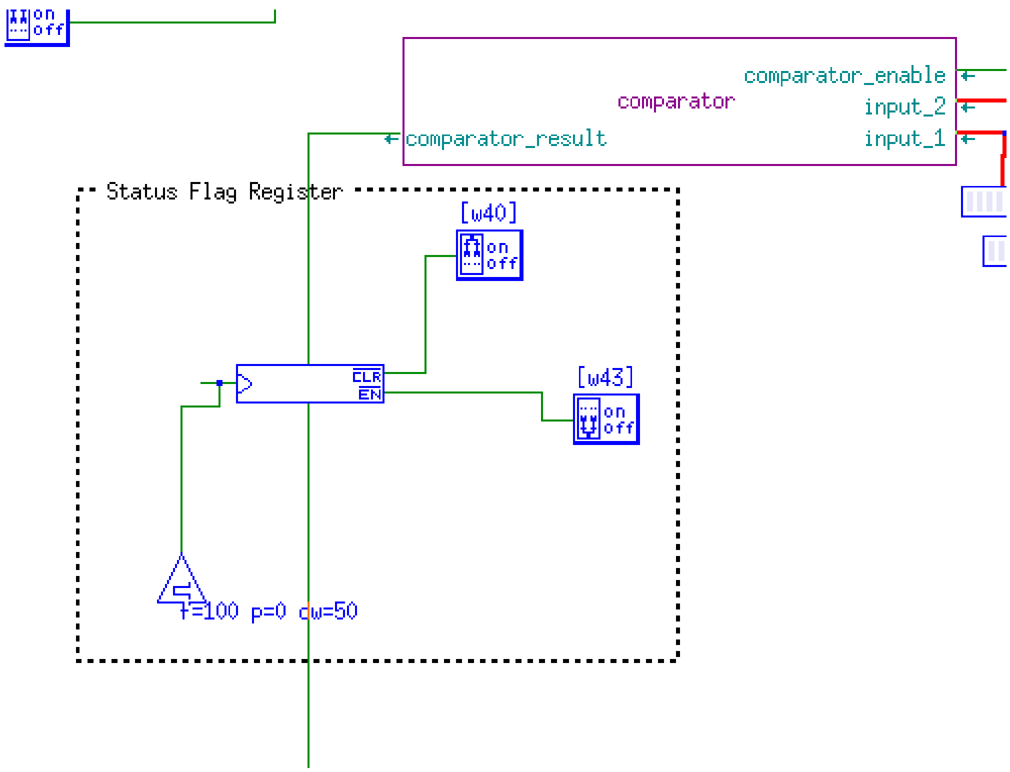
Comparator checks if two values taken as input from input_1 line and input_2 line are equal or not. The result of the comparator is taken from the compartaor_result line to the Status flag.

In comparator, the XOR the gate of 8 bit is used to compare each bit of the values. If the values will be equal the XOR gate will return 0 and the and OR of each bit will give result 0.
So the comparator will give result 0 if the two values are equal and 1 if the values are not equal.



Status Flag Register

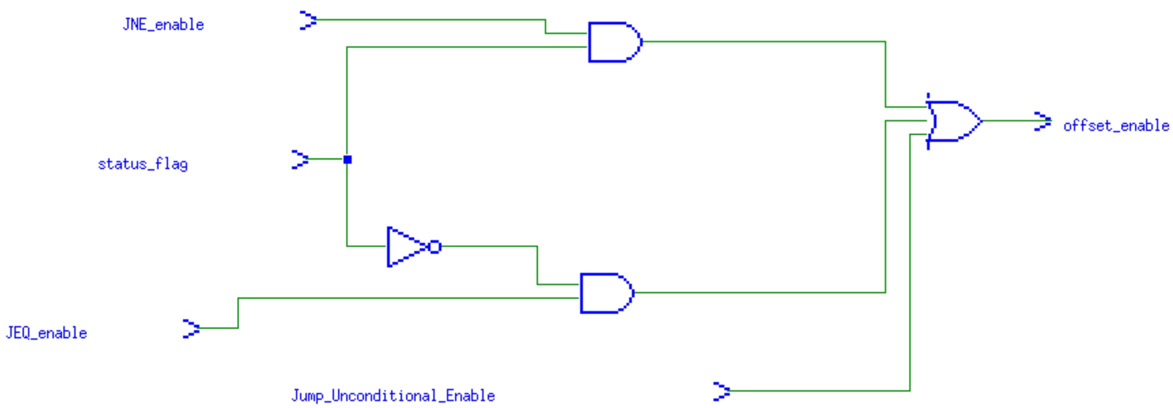
The status flag register holds only 1 bit to signify the comparison result. It is set to 1 if the two inputs in the comparator are NOT EQUAL and 0 if they are EQUAL.



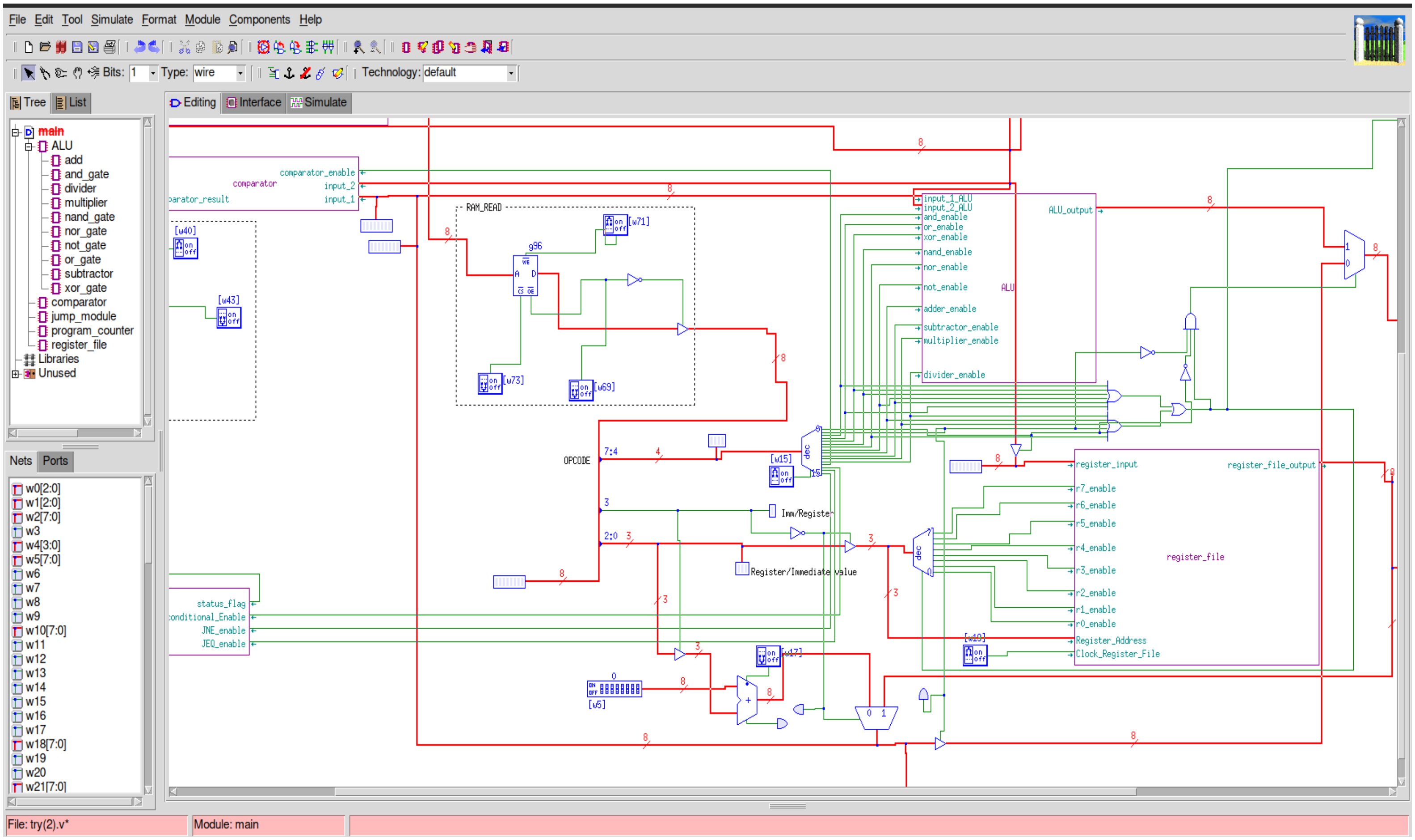
Jump Module:

The status flag is set to 1 when the comparison result is NOT EQUAL. The Jump offset will be sent only if there is a jump instruction and the comparison is satisfied. Jump if not equal and status flag set to 1 or Jump if equal and status flag set to 0 or if there is an unconditional jump.

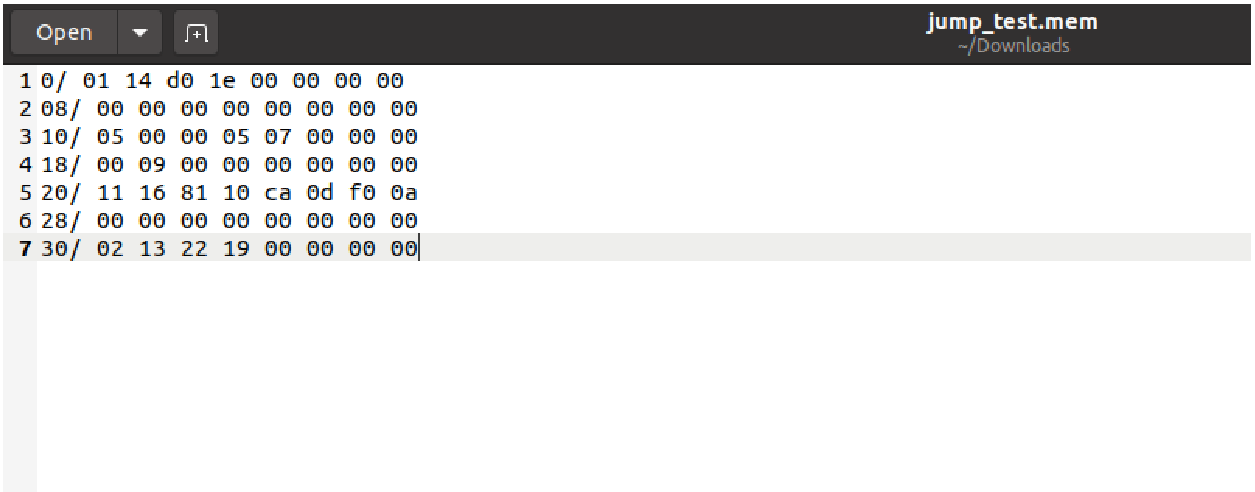
Offset_Enable = JNE.StatusFlag + JEQ.(StatusFlag)' + JUC



Designing of the main file which connects different modules:



INSTRUCTION EXECUTION

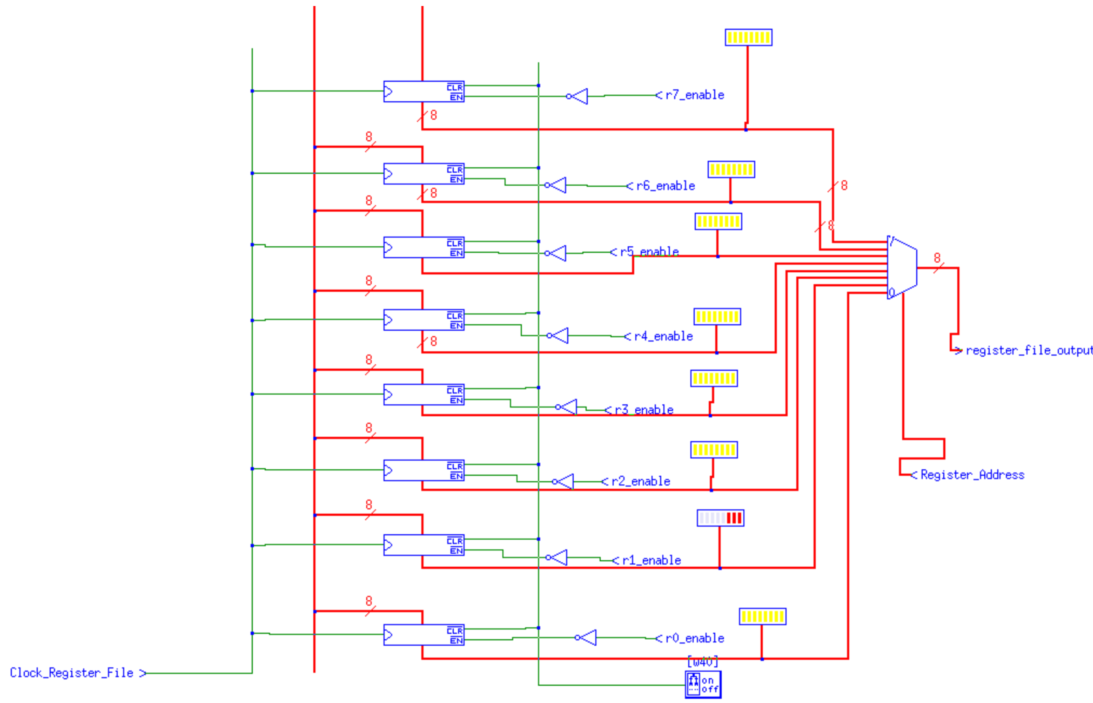



```
0/ 01 14 d0 1e 00 00 00 00
08/ 00 00 00 00 00 00 00 00
10/ 05 00 00 05 07 00 00 00
18/ 00 09 00 00 00 00 00 00
20/ 11 16 81 10 ca 0d f0 0a
28/ 00 00 00 00 00 00 00 00
30/ 02 13 22 19 00 00 00 00
```

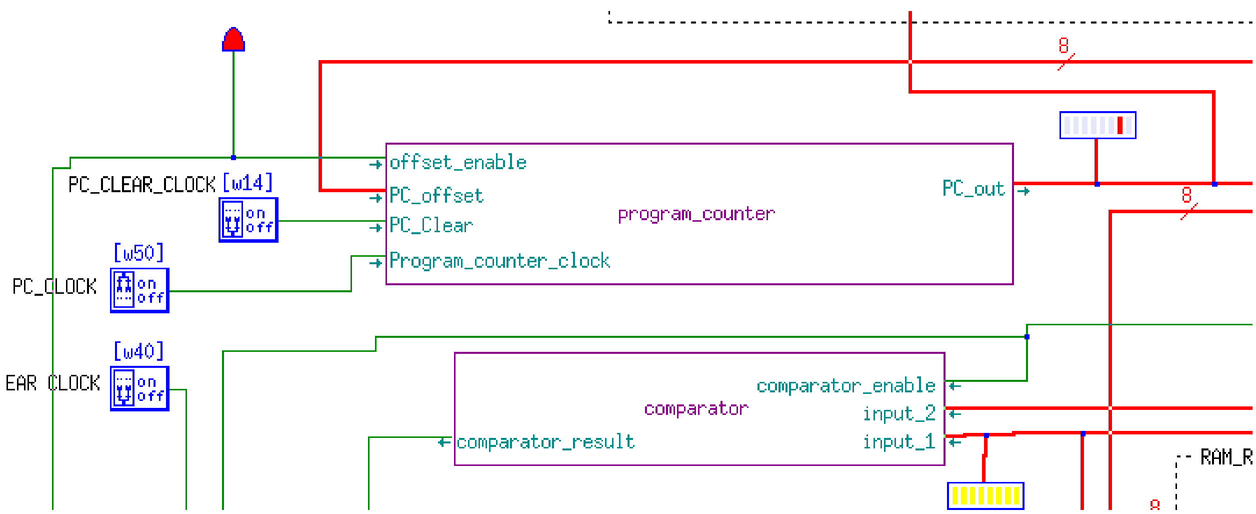
INSTRUCTION	BINARY	HEXADECIMAL
LOAD R1 (20)	0000 0001 0000 1010	01 14
JMP (#20)	1101 0000 0000 1010	d0 1e
STORE R1 (22)	0001 0001 0001 0110	11 16
ADD R1 (16)	1000 0001 0001 0000	81 10
CMP #2 (13)	1100 1010 0000 1101	ca 0d
JNE (#10)	1111 0000 0000 1010	f0 0a
LOAD R2 (19)	0000 0010 0001 0011	02 13
AND R2 (25)	0010 0010 0001 1001	22 19

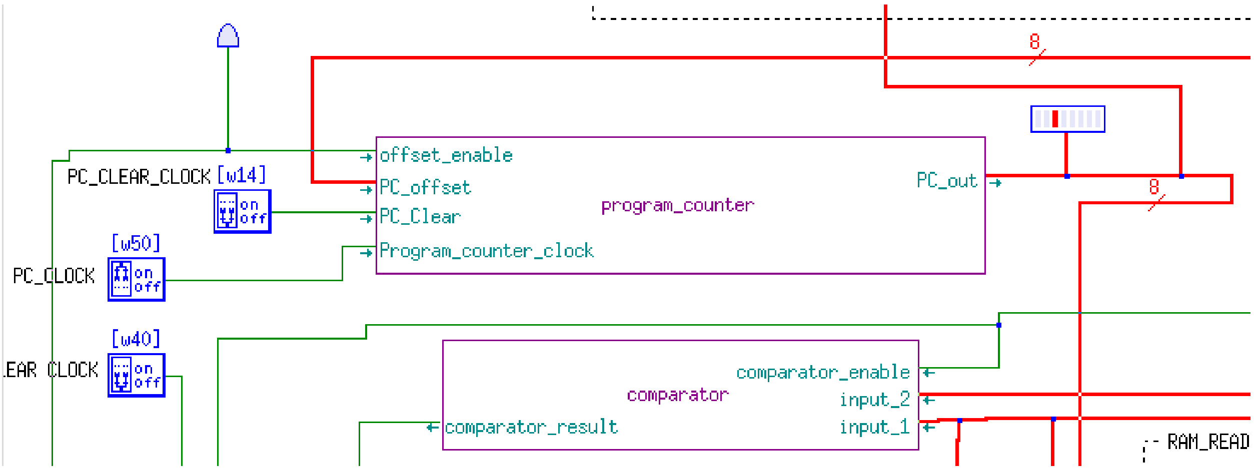
EXECUTION

LOAD R1 (20) -> 01 14

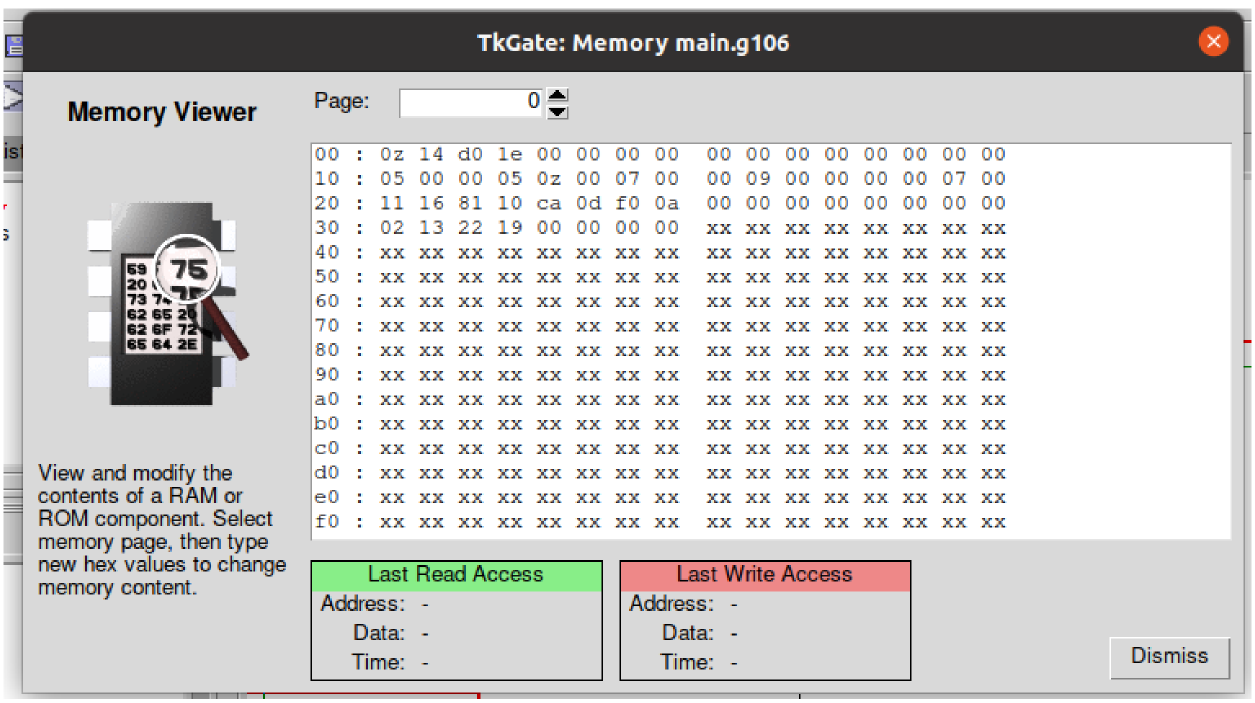


JMP #20 -> d0 1e

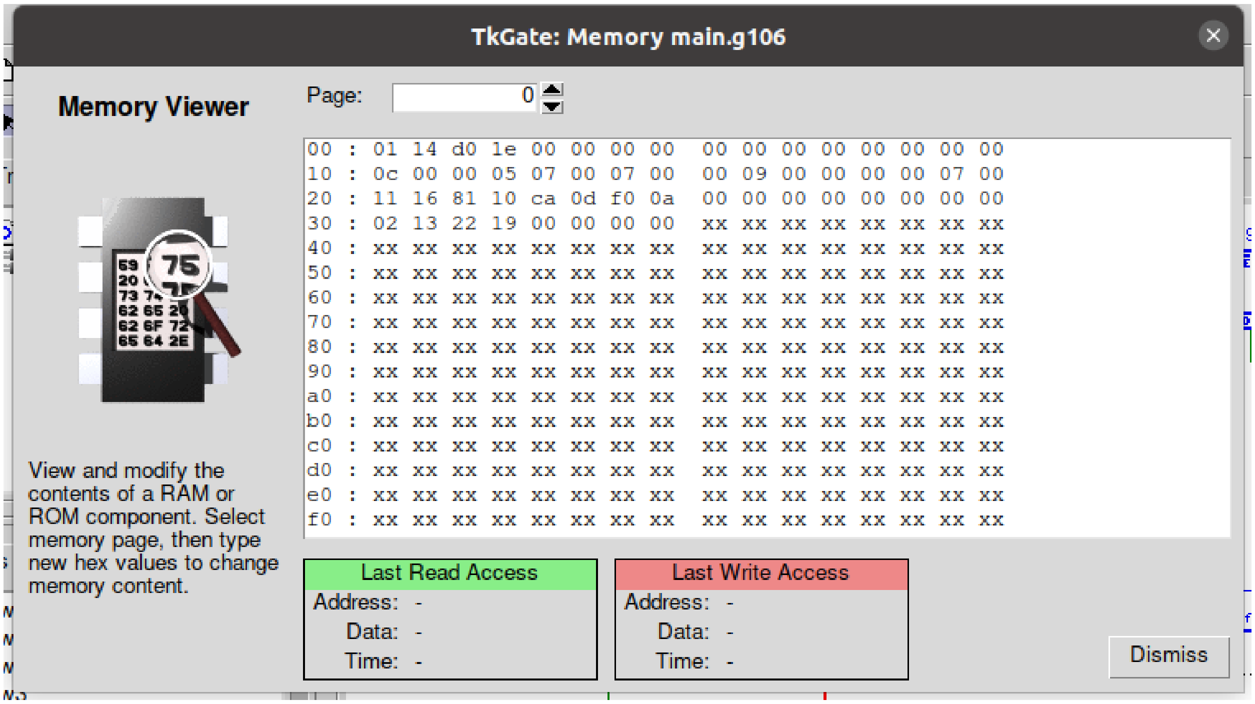




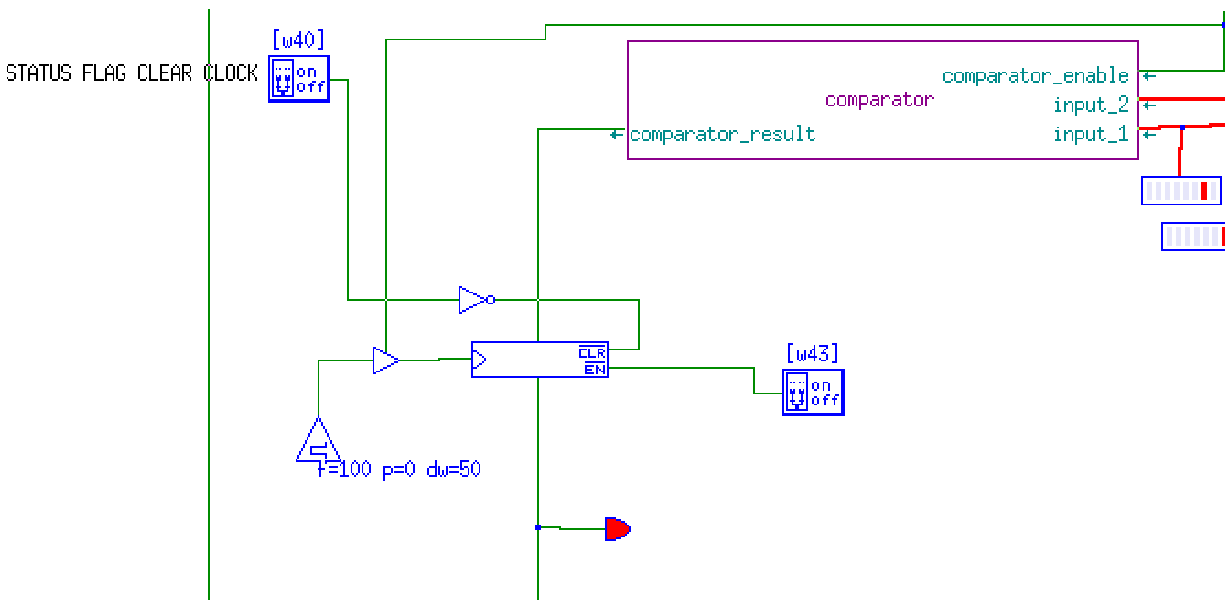
STORE R1 (22)   -> 11 16



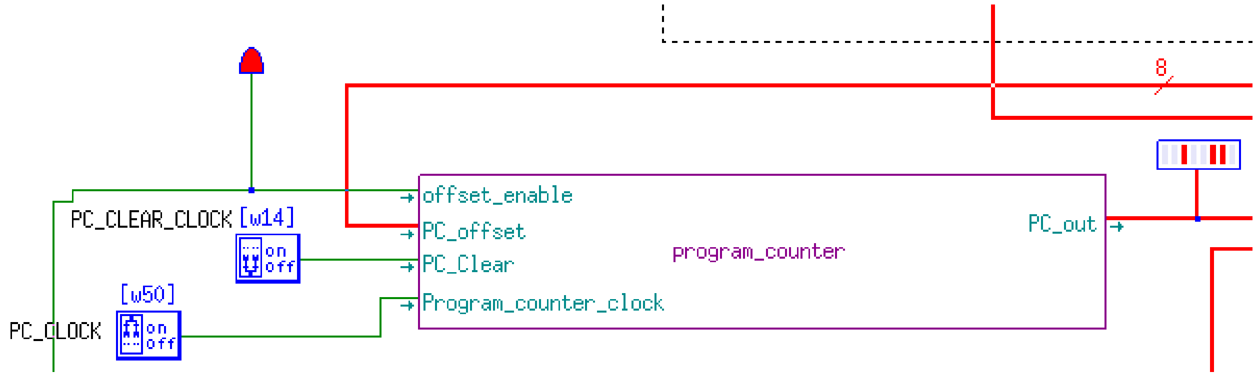
ADD R1 (16)   -> 81 10

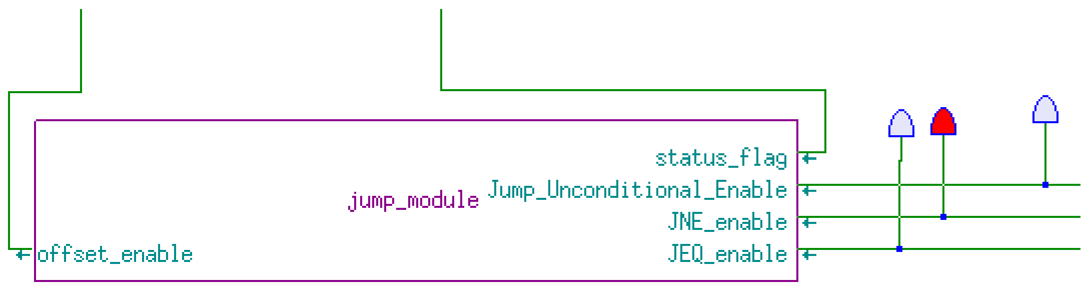


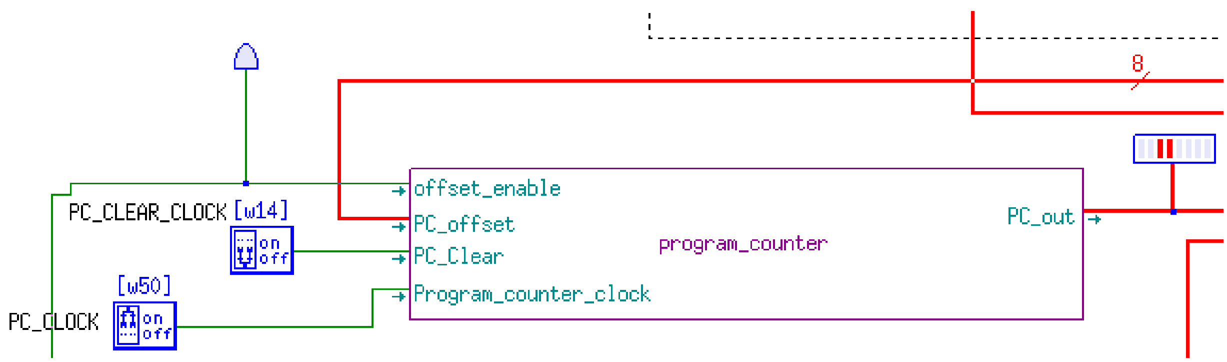
CMP #2 (13)   -> ca 0d



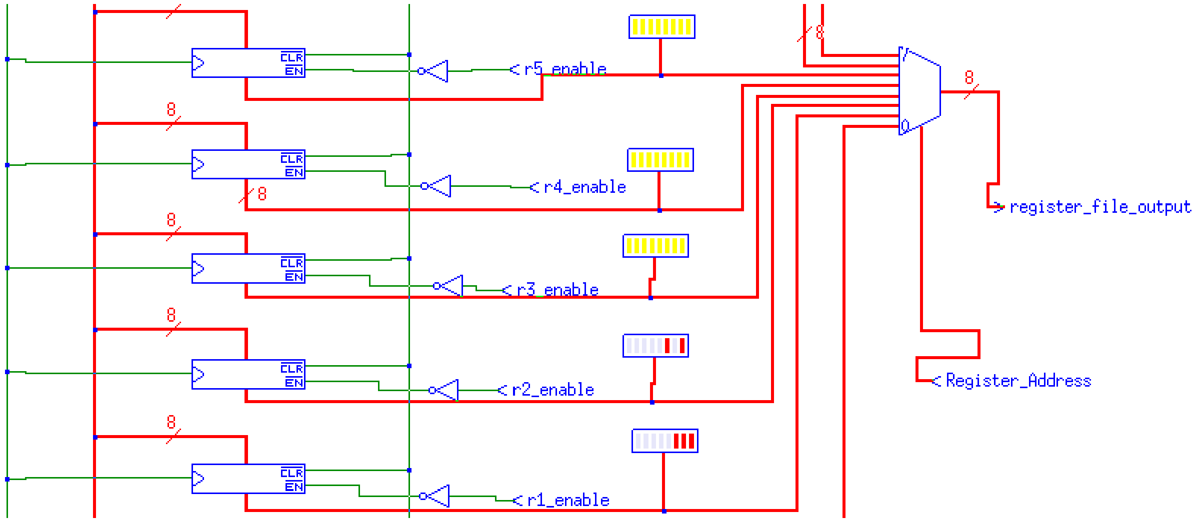
JNE (#10)   ->   f0 0a







LOAD R2 (19) -> 2 13



AND R2 (25)   -> 22 19