Project Report

Group members:
Armin Asgharifard
040190912

Behiç Erdem
040170212

Date

13.06.2023

Course title

VLSI II

Prof. Dr. Berna Örs Yalçın

Res. Assis. Fırat Kula

The Control Word

The control word in our processor is a **33-bit** word, for which the structure has been given as follows.

32 31 30 29 28	27 26 25 24 23	22 21 20 19 18	17	16 15 14 13	12	11 10 9	8	7 6 5 4 3	2	1	0
waddr	raddr0	raddr1	MB	FS	MD	wstrobe	we	shamnt	PL	JB	BC

waddr: the address to write data
raddr0: the address to read A from
raddr1: the address to read B from
MB: Mux B
FS: Function Select
MD: Mux D
wstrobe: whole, half, or byte select
we: write enable
shamnt: shift amount
PL: if set, it is either jump or branch.
JB: if set, it is a jump; otherwise, a conditional branch.
BC: if set, conditional branch on N; otherwise, conditional branch on Z

The encoding table for the control word is below:

waddr, raddr0, raddr1		MB		FS		MD		wstrobe		we		shamnt		PL		JB		BC	
Function	Code	Function	Code	Function	Code	Function	Code	Function	Code	Function	Code	Function	Code	Function	Code	Function	Code	Function	Code
R0	00000	Register	0	F ← B	0000	Function	0	Whole word	100	No write	0	Shift by 0	00000	No jump or branch	0	Conditional branch	0	Branch on Z	0
R1	00001	Constant	1	F ← B + 1	0001	Data in	1	Half word	010	Write	1	Shift by 1	00001	Either jump or branch	1	Unconditional jump	1	Branch on N	1
R2	00010			F ← A + B	0010			Byte	001			Shift by 2	00010						
R3	00011			F ← A + B + 1	0011							Shift by 3	00011						
R4	00100			F ← A + ~B	0100							Shift by 4	00100						
R5	00101			F ← A + ~B + 1	0101							Shift by 5	00101						
R6	00110			F ← B - 1	0110							Shift by 6	00110						
R7	00111			F ← B	0111							Shift by 7	00111						
R8	01000			F ← A & B	1000							Shift by 8	01000						
R9	01001			F ← A | B	1001							Shift by 9	01001						
R10	01010			F ← A ^ B	1010							Shift by 10	01010						
R11	01011			F ← 0	1011							Shift by 11	01011						
R12	01100			F ← B ≫ shamnt	1100							Shift by 12	01100						
R13	01101			F ← B ≪ shamnt	1101							Shift by 13	01101						
R14	01110			F ← B ⋙ shamnt	1110							Shift by 14	01110						
R15	01111			F ← B ⋘ shamnt	1111							Shift by 15	01111						
R16	10000											Shift by 16	10000						
R17	10001											Shift by 17	10001						
R18	10010											Shift by 18	10010						
R19	10011											Shift by 19	10011						
R20	10100											Shift by 20	10100						
R21	10101											Shift by 21	10101						
R22	10110											Shift by 22	10110						
R23	10111											Shift by 23	10111						
R24	11000											Shift by 24	11000						
R25	11001											Shift by 25	11001						
R26	11010											Shift by 26	11010						
R27	11011											Shift by 27	11011						
R28	11100											Shift by 28	11100						
R29	11101											Shift by 29	11101						
R30	11110											Shift by 30	11110						
R31	11111											Shift by 31	11111						

waddr, raddr0, raddr1 are addresses to read and write data.
MB is the multiplexer to select either loading data from a register or an immediate value.

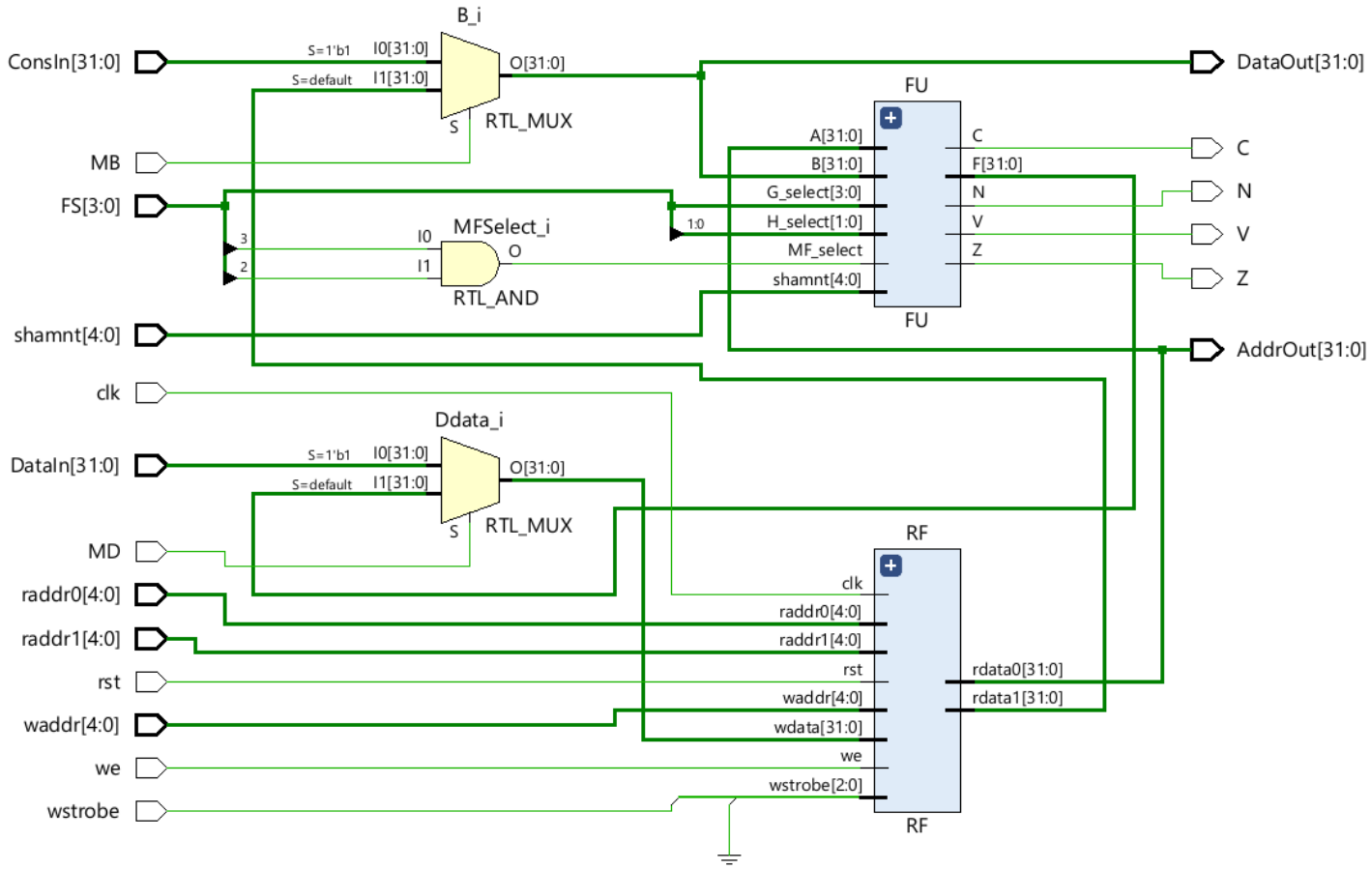
FS is the function select which which changes **MF, GS,** and **HS.** As you know, MF is the selection for ALU output or Shifter output. GS is the selection for ALU operations, and HS is the selection for the shifter operations. The relations between FS and those selection bits are:

- $MF_i = FS_3 \& FS_2$
- $GS_i [3:0] = FS_i$
- $HS_i [1:0] = FS_i$

MD is the multiplexer to either load data into a register from the data memory or the FU output.
wstrobe is for choosing a whole word, half word, or a byte load/store.
we is the enable bit for writing to RF.
shamnt is the amount by which shifting a number is requested.
PL, JB, and **BC** are the control signals of the program counter.

Datapath

The datapath is formed by making appropriate connections between RF and FU. The following is the resulting schematic of the datapath.



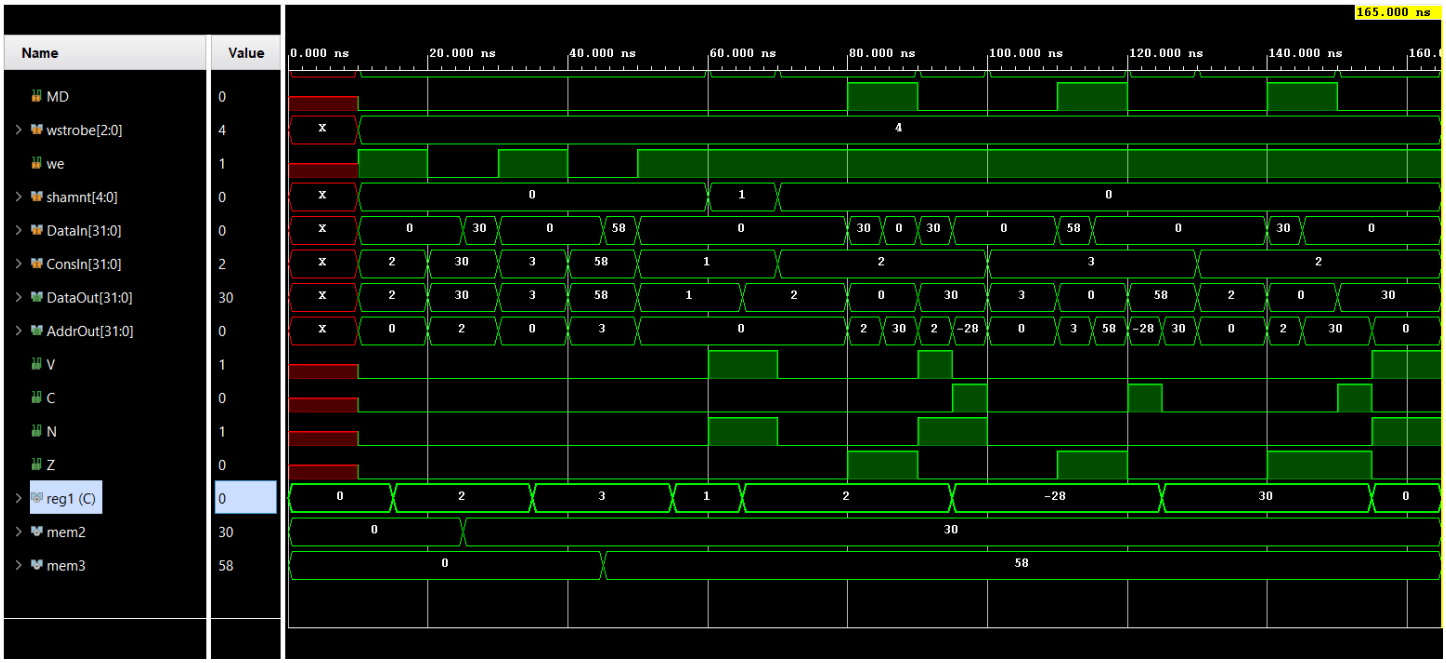
As an example, we want to perform the instructions below in order:

$C = 1$
$C = 2C$
$C = C - N$
$C = C + A$
$C = C - N$

Assuming that N and A are already initialized in the data memory address 2 and 3, respectively, the assembly codes for the above operations would be: (r1 is the assigned register for C)

```
ADDI r1, r0, #1      // Move number 1 to r1
SLLI r1, r1, #1      // Multiply r1 by 2
ADDI r2, r0, #2      // Move number 2 to r2
LOAD r2, (r2)        // Load address 2 of memory into r2
SUB r1, r1, r2       // Store r1 - r2 into r1
ADDI r2, r0, #3      // Move number 3 to r2
LOAD r2, (r2)        // Load address 3 of memory into r2
ADD r1, r1, r2       // Store r1 + r2 into r1
ADDI r2, r0, #2      // Move number 2 to r2
LOAD r2, (r2)        // Load address 2 of memory into r2
SUB r1, r1, r2       // Store r1 - r2 into r1
```

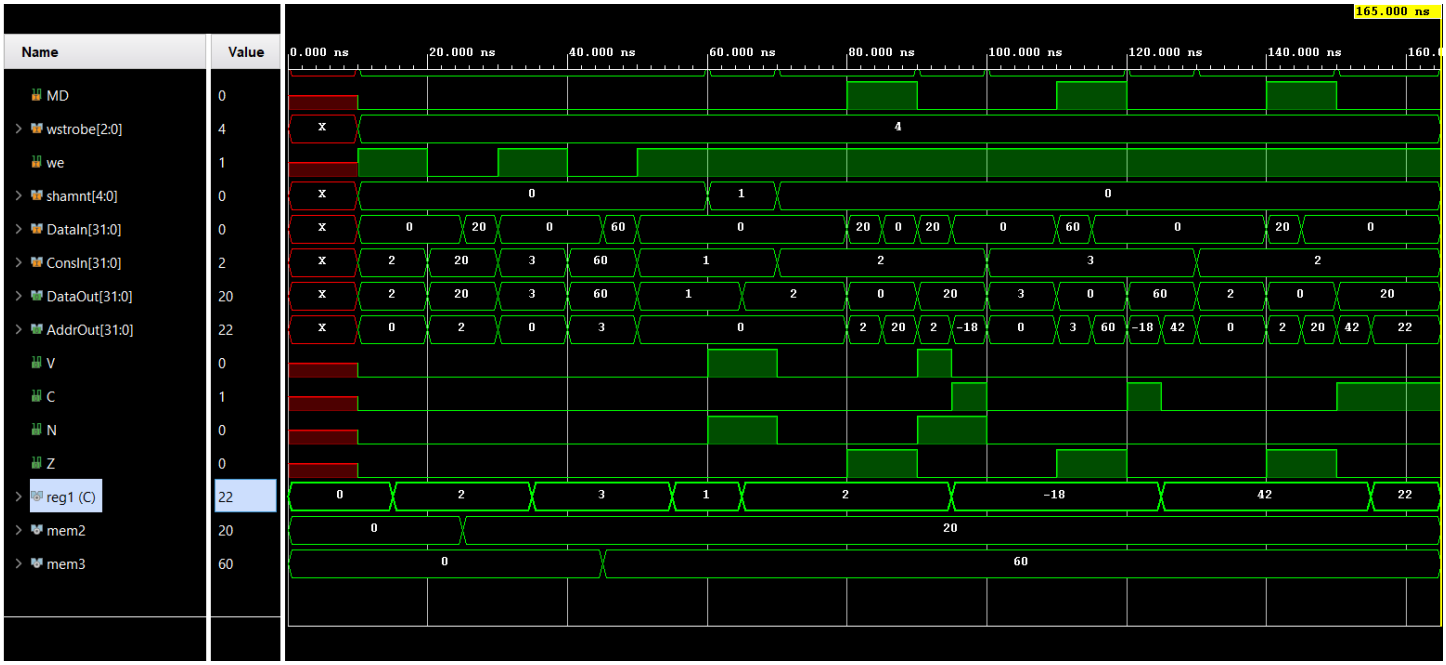
To try this in testbench, we translated this into their corresponding words and applied them in **Datapath_tb.v**. In the test code, instead of using $readmemb, the data memory were also initialized using required control words beforehand. The resulting waveform is as below.

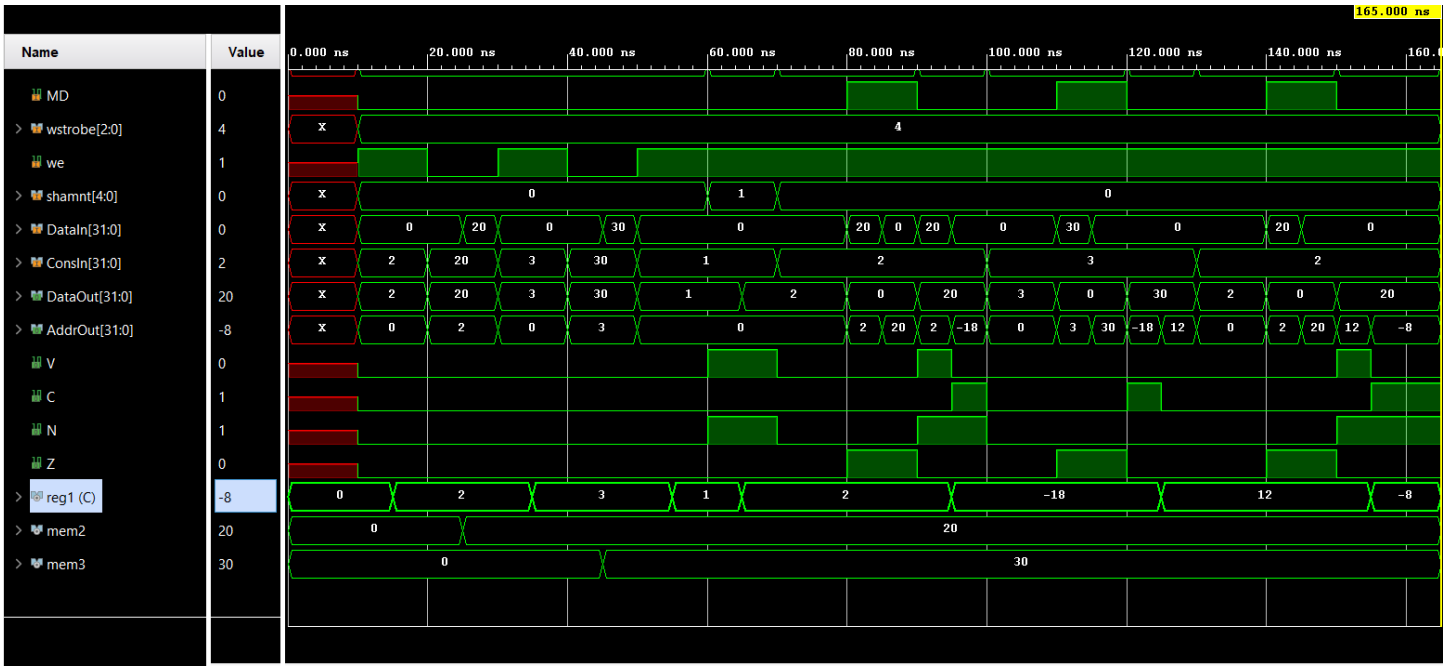


In the example above, N and A are chosen as 30 and 58, respectively. As you can see, after the initialization of the data memory numbers, register1, which is assigned for holding C value, has been set to 1. Multiplication by

2 follows it. Then, we see subtraction of N from C, which results in -28. After that, it is added to A, which becomes 30. By subtracting N from C again, zero is obtained, as it is clear.

The two following examples demonstrate conditions in which a positive and a negative value for C is obtained at the end.



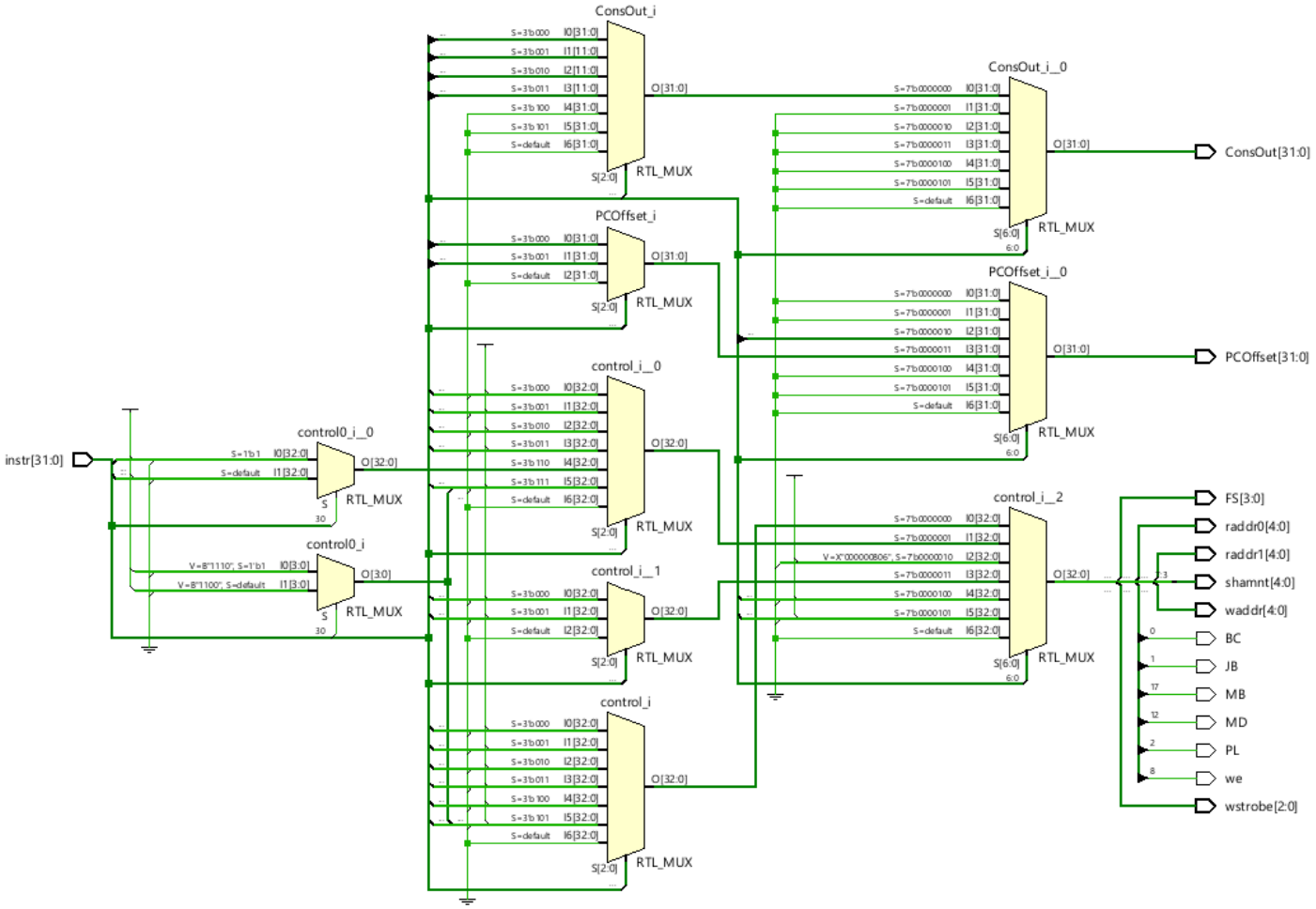


Instruction Decoder

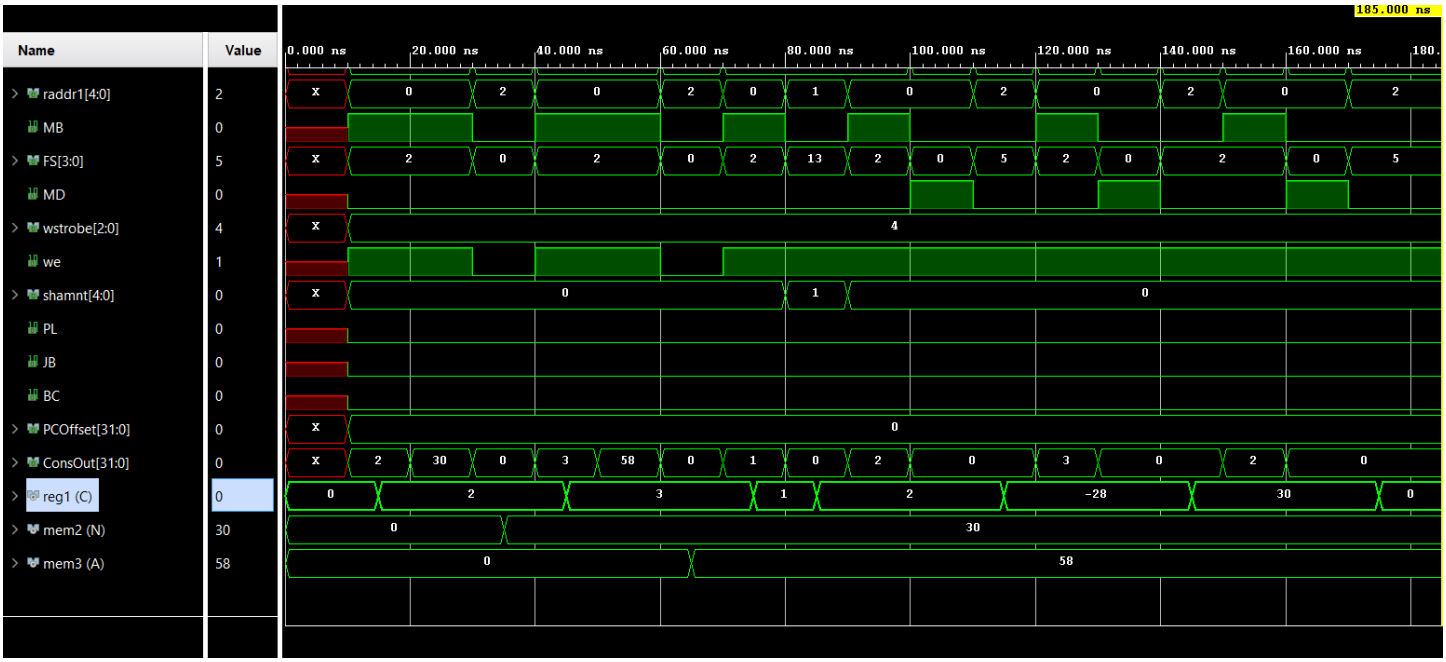
The instruction decoder that we designed receives the 32-bit instruction as the input, and provides the control word, the program counter offset (is there is a jump or branch), and the constant input to the datapath.
The instructions that our processor supports are the following:

- ADDI
- ANDI
- ORI
- XORI
- SLLI
- SRLI
- SRAI
- ADD
- AND
- OR
- XOR
- SLL
- SRL
- SRA
- SUB
- JUMP
- BEQ
- BLT
- LOAD
- STORE

According to RV32I manual, we have designated a unique binary opcode, funct3, and funct7 values for each instruction. The design, unsurprisingly, consists of only multiplexers in the high-level schematic.



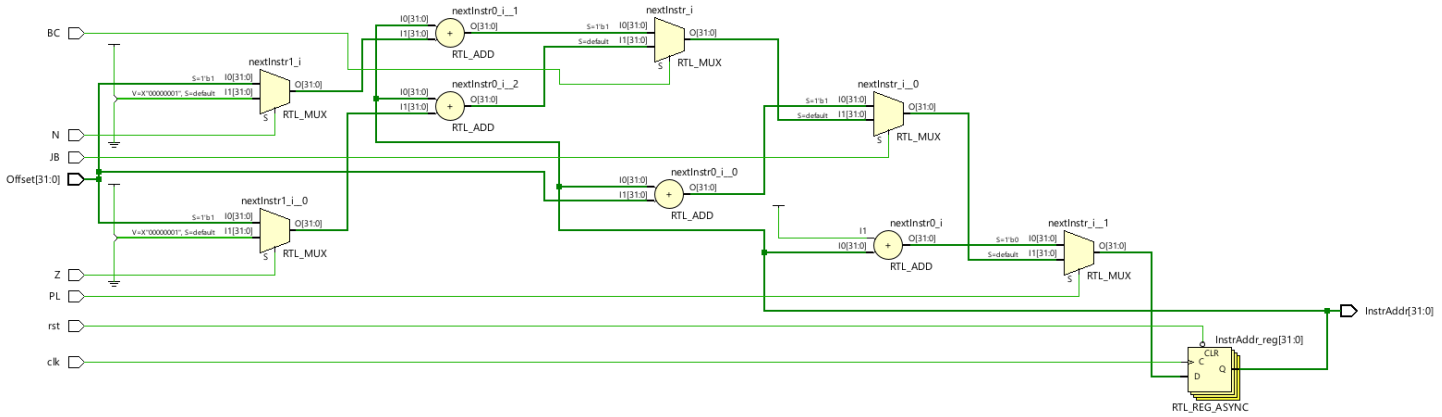
The exact same operations in the previous section are performed here using the instruction decoder. Instead of giving the control words in the test code, instruction bits were given to let the circuit generate the control words and the constant inputs on its own. The filename is **ID_tb.v**. Please take a look at the resulting waveform.



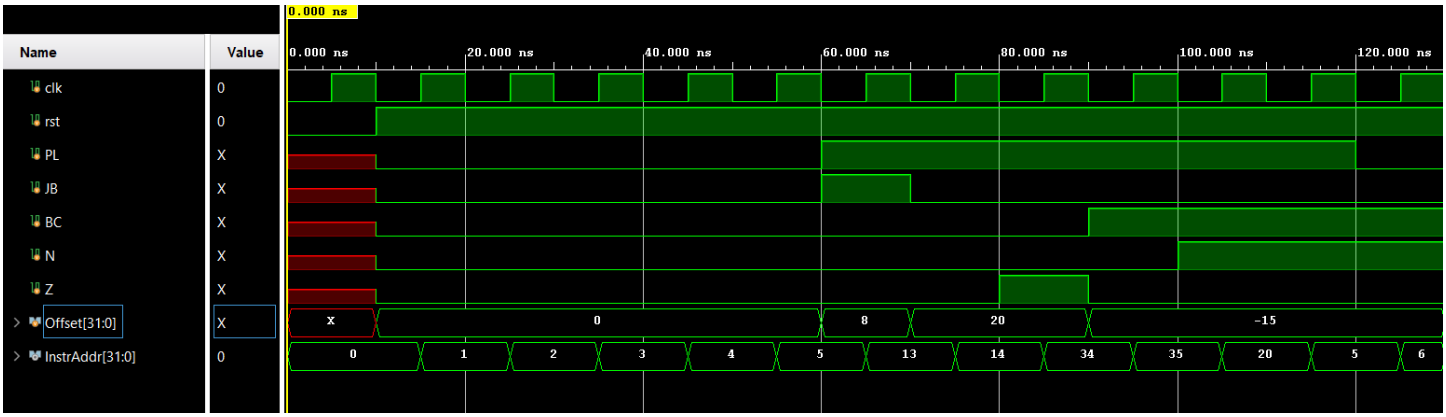
Since successful decoding has been performed, the same operations as the previous section has been done in the same way.

Program Counter

The program counter in our design receives Negative, Zero inputs from the Datapath, and PL, JB, BC from the instruction decoder to decide on the next instruction address to give out to the instruction memory. The following schematic represents the design.



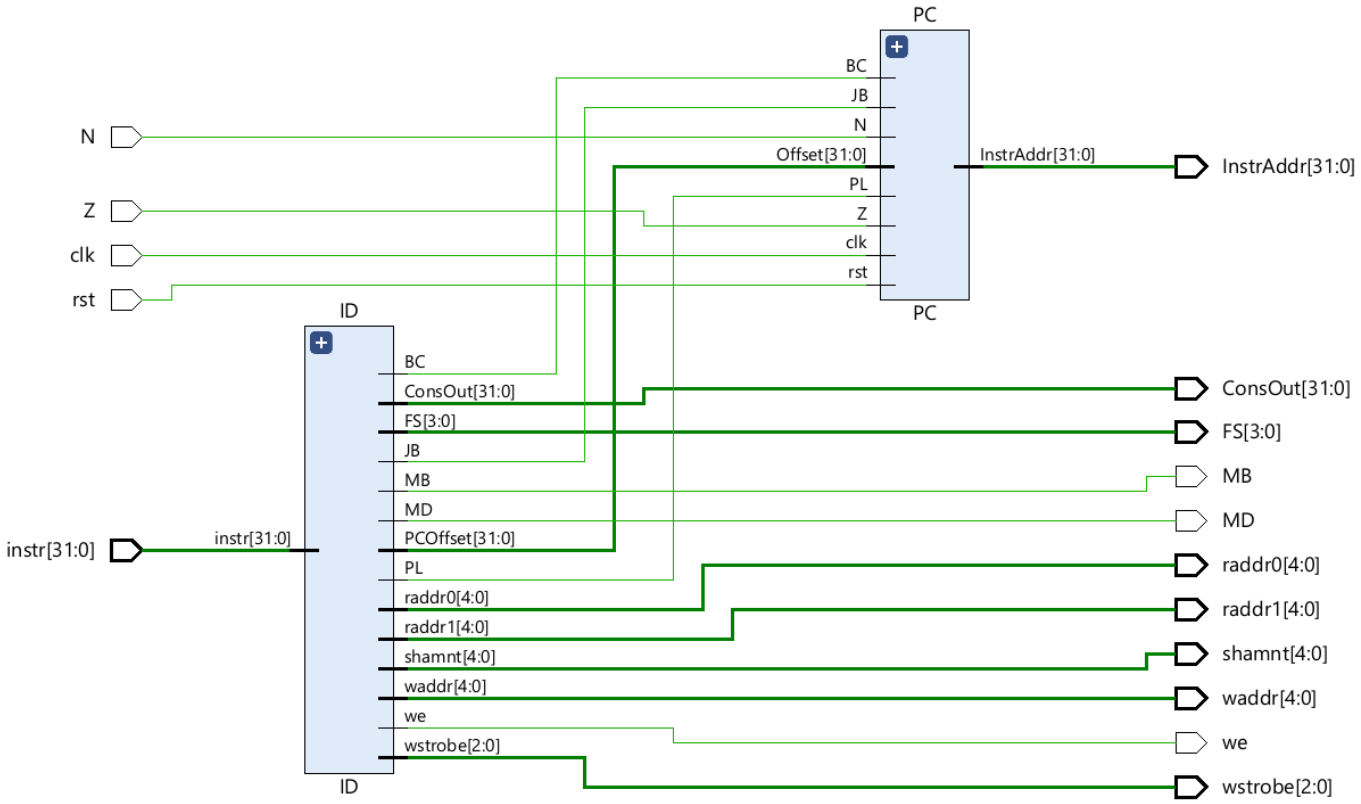
To test this module, all possible input sets were tested.



Whenever PL = 0, InstrAddr is incremented by one at each clock cycle. When PL = 1, and JB = 1, InstrAddr will be incremented or decremented by whatever the offset value is at the moment. If PL = 1, JB = 0, depending on BC value, N or Z will be checked to be decided on whether perform the offset addition, or still increment the address by one. BC = 0 will make the circuit dependent on Z, and BC = 1 will make the circuit dependent on N.

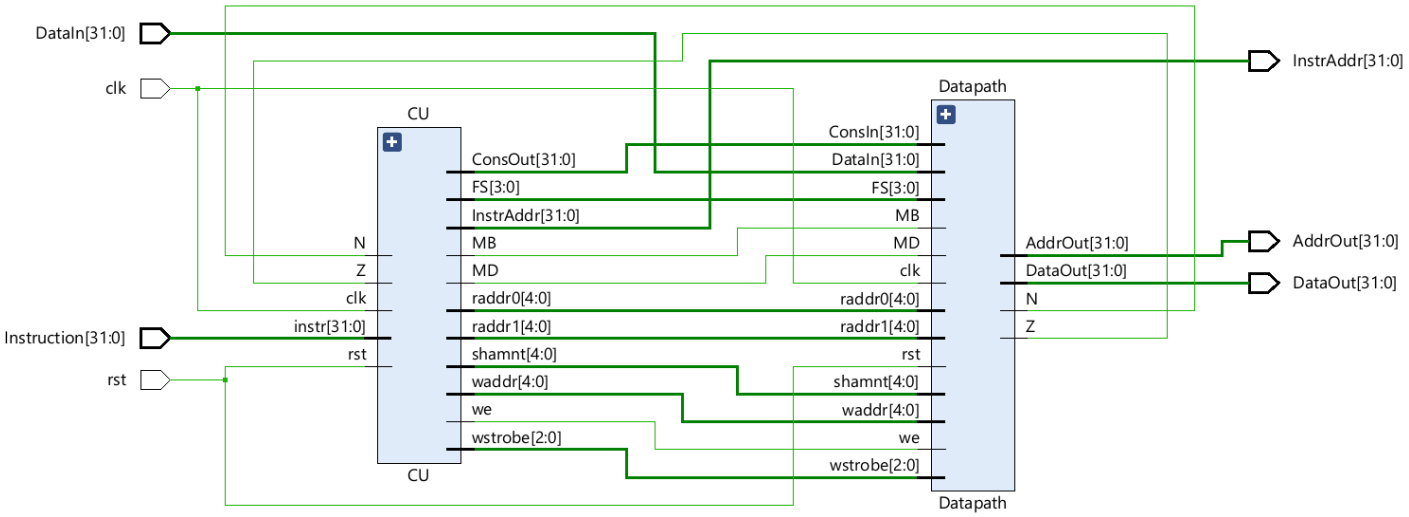
Control Unit

The Instruction Decoder and Program Counter are connected together to form the control unit.



The Complete Processor

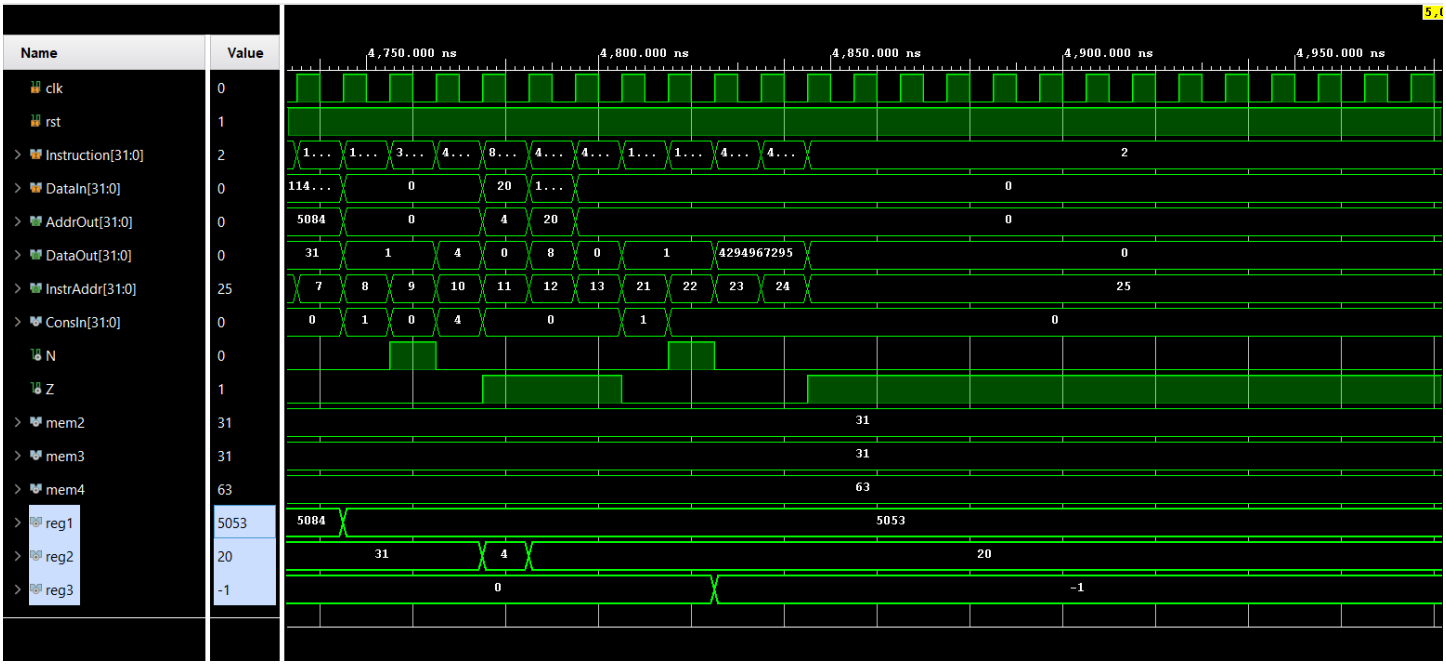
The control unit and the datapath are connected together to form the complete processor. Take a look at the following schematic.



To test this processor, two register files will also be instantiated along with the processor; one for data memory, and one for instruction memory. Both of them are initialized using $readmemb function. Memory initialization files are included as .mem file. The assembly code to run the whole algorithm is the following code:

```
ADDI r1, r0, #0
ADDI r3, r0, #31
for_loop: SLLI r1, r1, #1
ADDI r2, r0, #2
LOAD r2, (r2)
BLT r1, r2, CLN1
SUB r1, r1, r2
CLN1: ADDI r4, r0, #1
SLL r4, r4, r3
ADDI r2, r0, #4
LOAD r2, (r2)
AND r4, r4, r2
BEQ r0, r4, CLN2
ADDI r2, r0, #3
LOAD r2, (r2)
ADD r1, r1, r2
ADDI r2, r0, #2
LOAD r2, (r2)
BLT r1, r2, CLN2
SUB r1, r1, r2
CLN2: ADDI r4, r0, #1
SUB r3, r3, r4
BLT r0, r3, for_loop
BEQ r0, r3, for_loop
exit: Jump exit
```

N, A, and B are initialized as 31, 31, and 63 in addresses 2, 3, and 4 respectively. The instruction machines coded are initialized in the instruction memory. Register1 is again reserved for C value, and register3 is assigned for handling the for loop. The waveform is too long but a portion of it is given here.



This is the ending portion of the waveform. Look at InstrAddr. You can see that the program ends at the 25th instruction and sticks in there because the last line of our assembly code is self-jumping instruction. This necessary in a microprocessor code to end in a self-jumping line, and this is obvious here. On around 4840ns, we can see that a conditional branch has occurred and InstrAddr has branched from 13 to 21.
You can checkout the whole simulation to verify the instructions performing correctly. The testbench filename is **Processor_tb.v**.