Name: Khau Lien Kiet
Student ID: V202000119
Lab Instructor: Truong Tuan Vu
College: CECS

COMP2020 Design Documentation

Adopted from VinUniversity

November 3, 2021

I. Overview

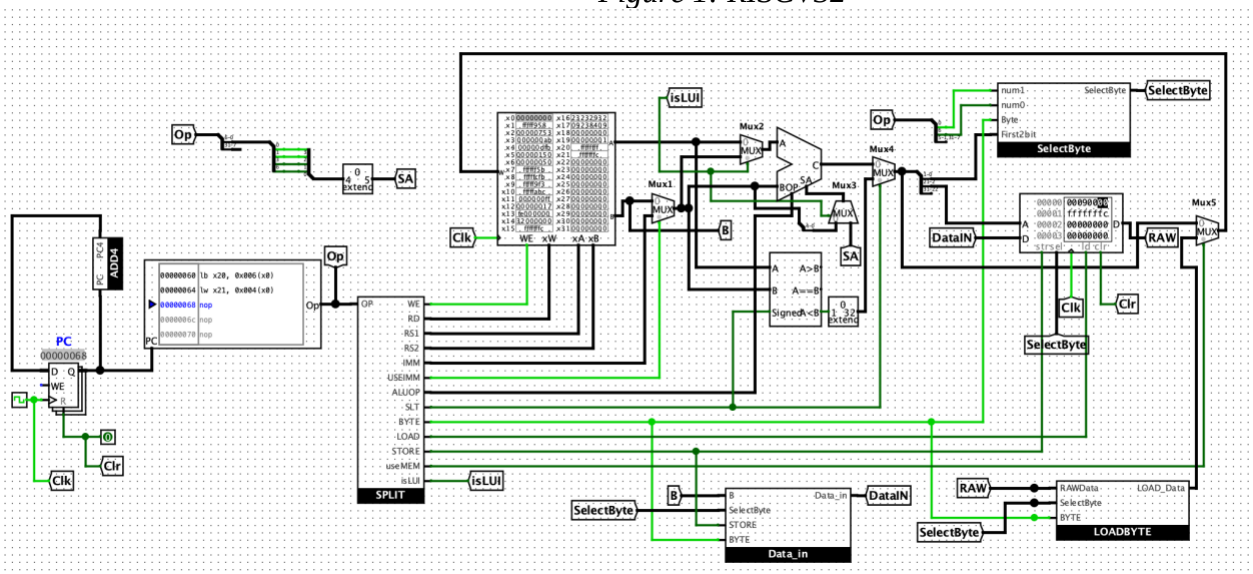
The project 2 is aimed to design a single-cycle processor in Logisim, which includes a small subset of the RISCV Instruction set (arithmetic & logic and memory operations)

The purpose of this project is to perform some dominant operations in the Arithmetic & Memory Logic. Those sections below would clearly state how the components of RISCV is arranged as well as the usage of the “boxing” principle.

II. Component Design Documentation

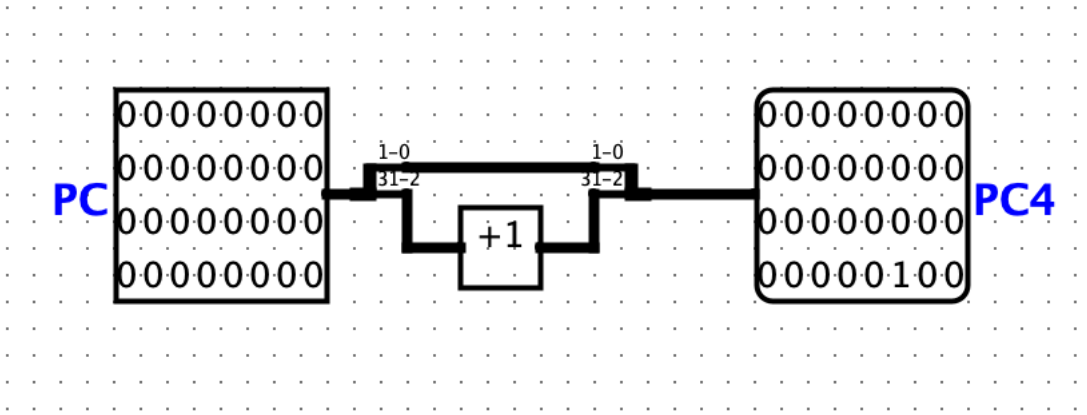
a. Diagram & Description

Figure 1: RISCV32



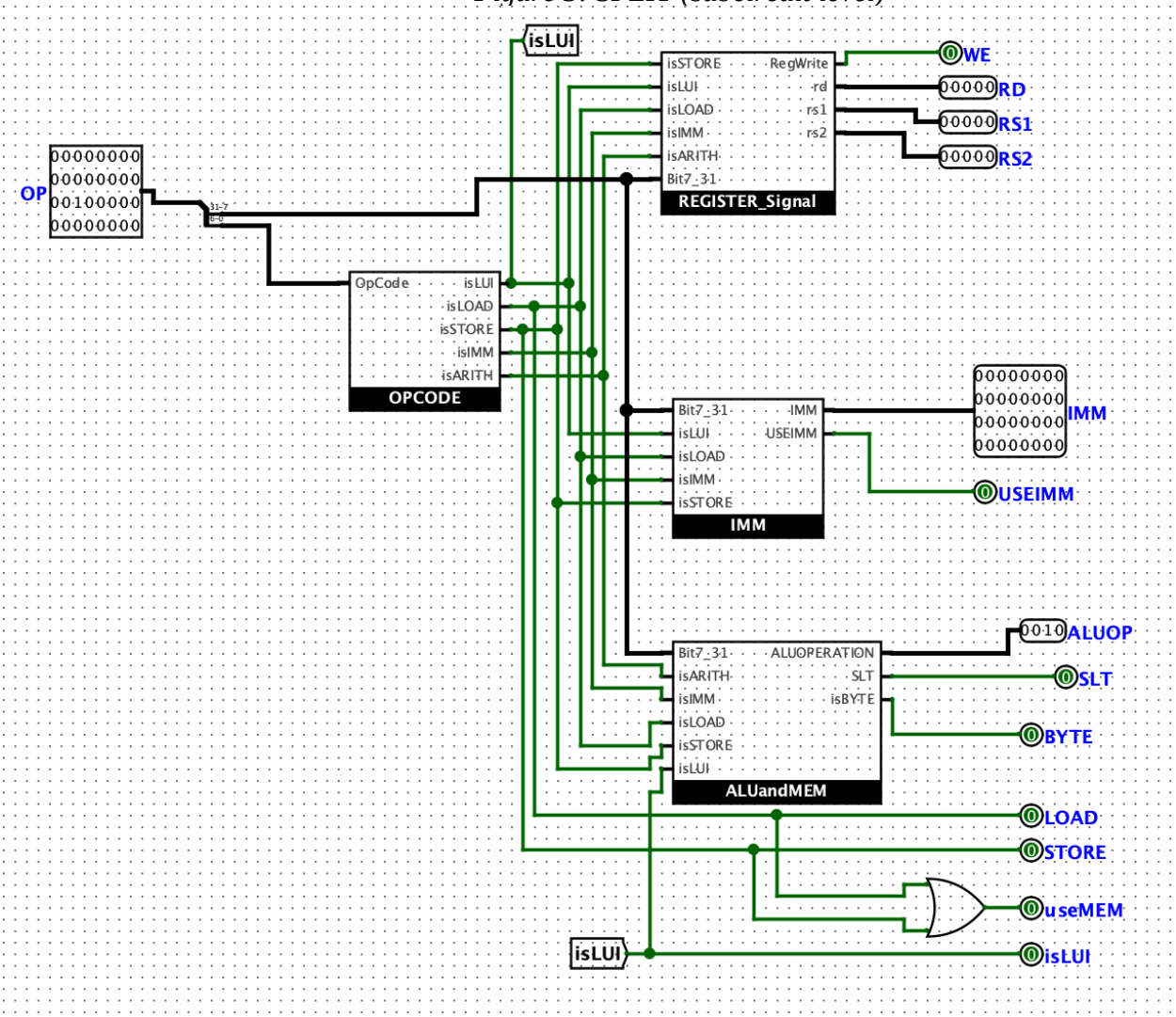
Description: The full RISCV is the combination of the three big subcircuits, which are ADD4, SPLIT, and SelectByte.

Figure 2: ADD4 (subcircuit level)



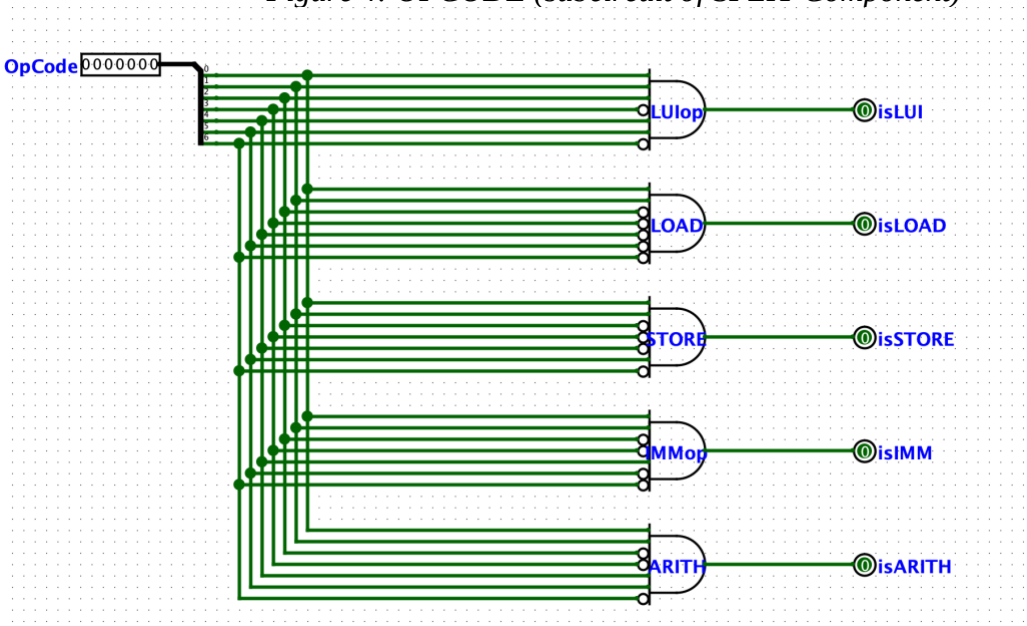
Description: This component is used to increment 4 byte after each program counter is performed

Figure 3: SPLIT (subcircuit level)



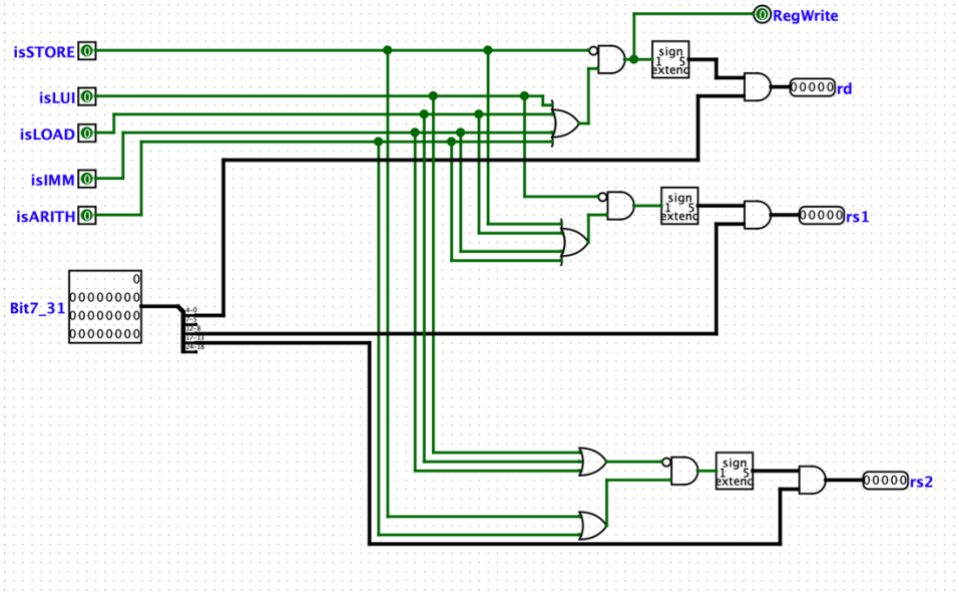
Description: The SPLIT Component is used to perform as the controller since it receives the Op from RISC-V Program ROM, thus its output will be determined by the Op and gives out the value outside SPLIT component. Then, its performance would go through the exact destination of other components.

Figure 4: OPCODE (subcircuit of SPLIT Component)



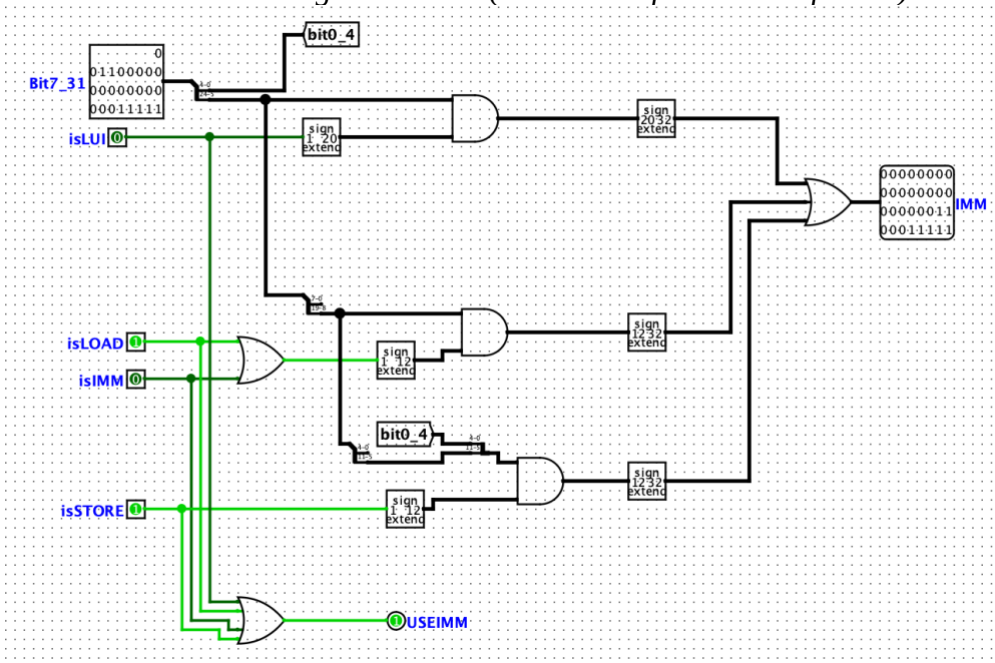
Description: The subcircuit is used to examine the Opcode to determine which group belongs to.

Figure 5: REGISTER_Signal (subcircuit of SPLIT component)



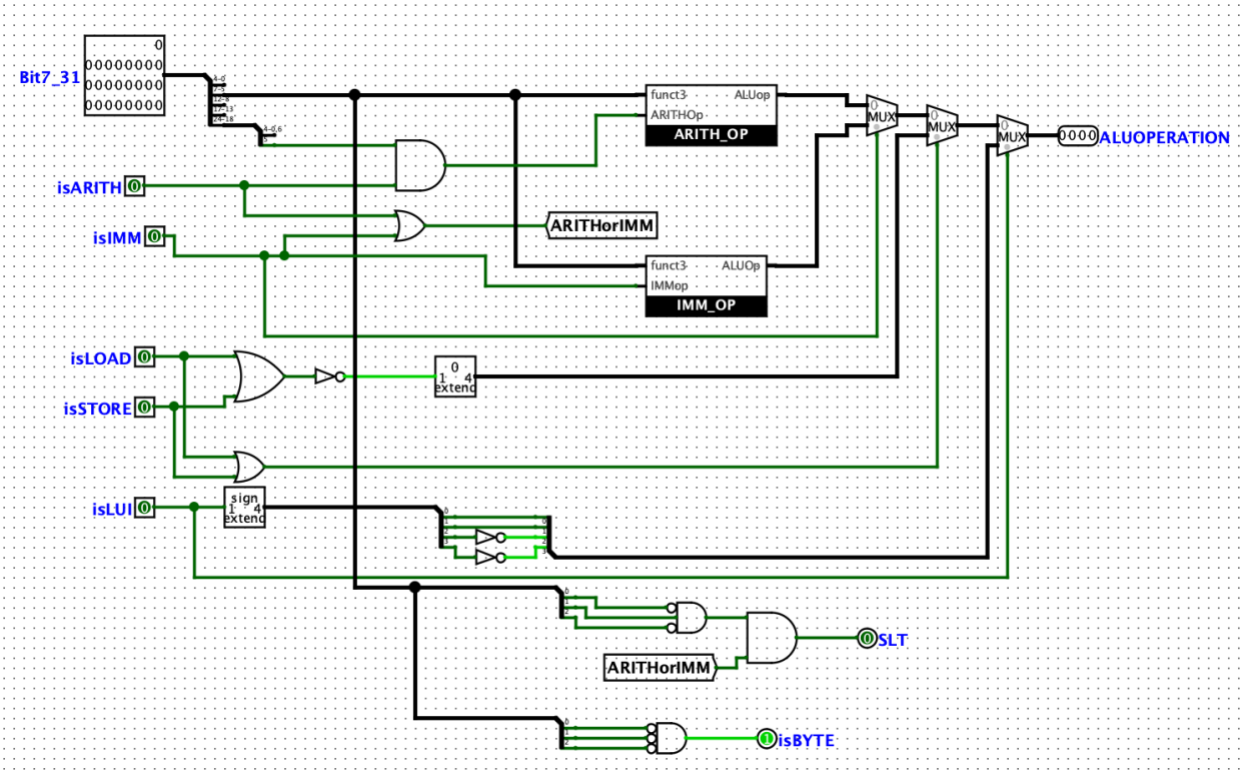
Description: The subcircuit is used for to give out the rd (register destination), rs1 (register source 1), rs2 (register source 2) as well as the RegWrite, which is used for the Write Enabled

Figure 6: IMM (subcircuit of SPLIT component)



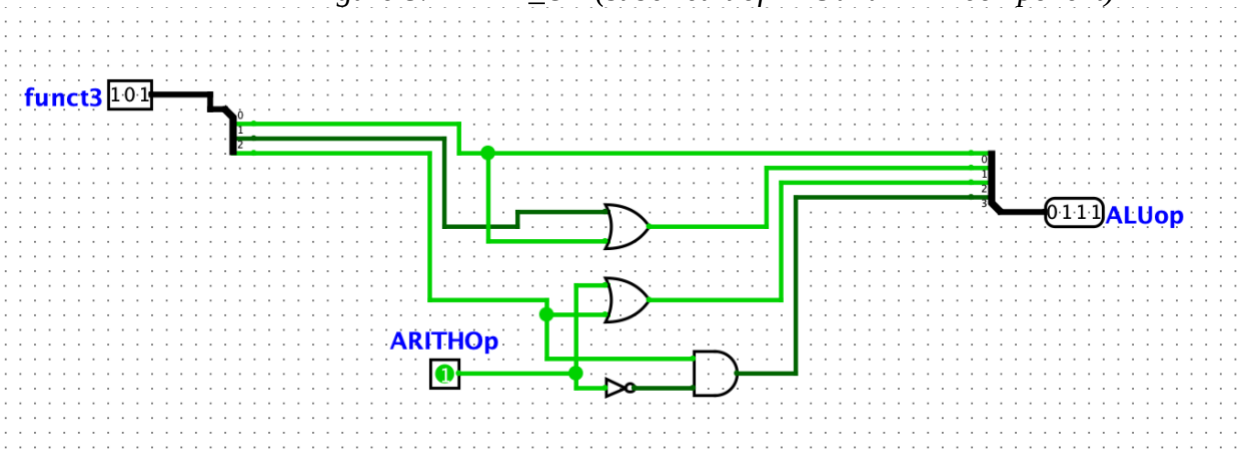
Description: The subcircuit is the combination of all the function consisting the IMM (immediate value) so that the output of this circuit is to generate the IMM (Immediate value) as well as the recognition of using MEM.

Figure 7: ALUandMEM (subcircuit of SPLIT component level)



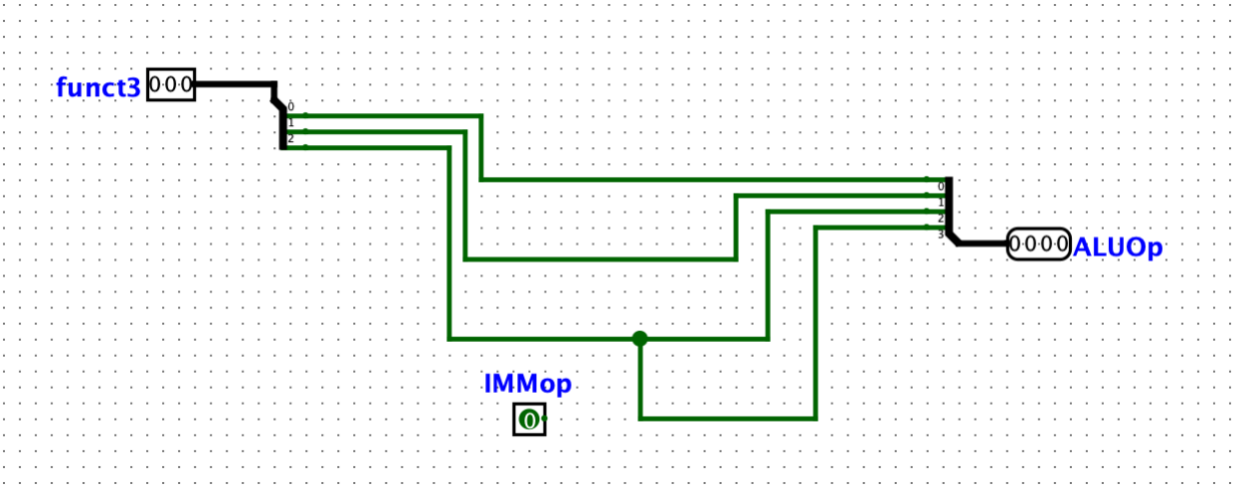
Description: From this circuit, for each different type of function, we can generate ALUOperation, which is used for inputting the ALU, and then perform its given operation from the ALU.

Figure 8: ARITH_OP (subcircuit of ALUandMEM component)



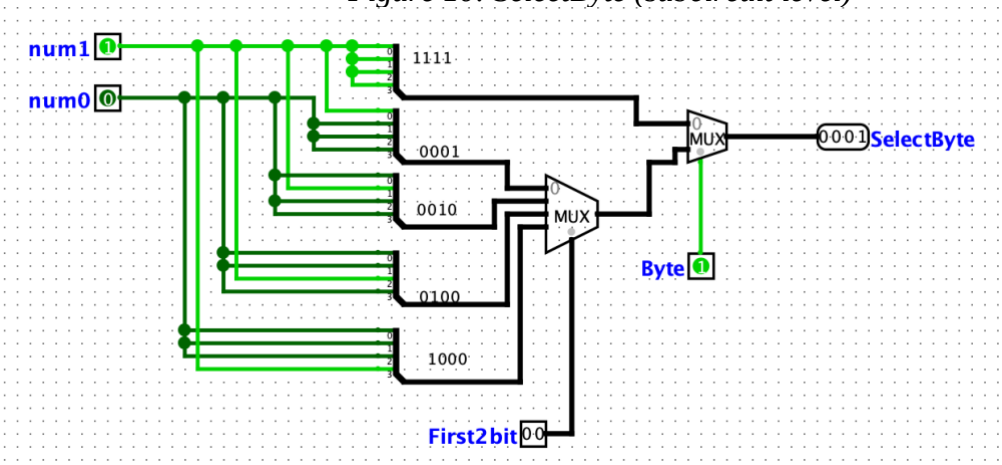
Description: The subcircuit is used for determining the ALUOp when having an input from the funct3 combined with the ARITHOp

Figure 9: IMM_OP (subcircuit of ALUandMEM component)



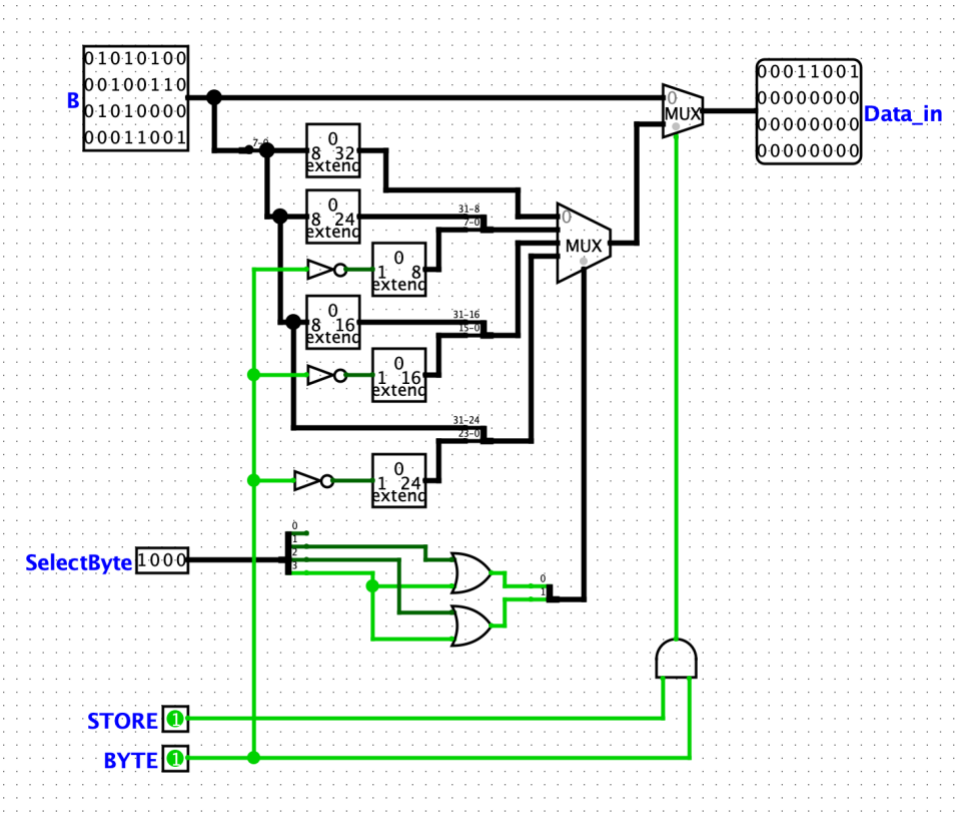
Description: Its purpose is the same as the Figure 9. However, it is used for creating the ALUOp for which function contained the immediate

Figure 10: SelectByte (subcircuit level)



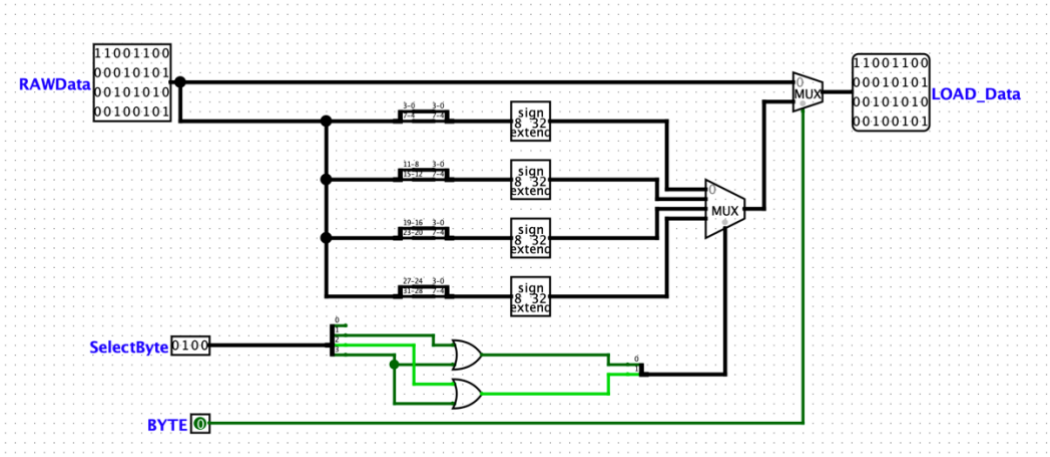
Description: The purpose of the SelectByte is to generate the 4-bit of SelectByte that goes in the RAM (Memory).

Figure 11: DATA_IN (subcircuit of the RISCV)



Description: This specialized circuit is to function the STORE Operation and select whether it is the SW or SB

Figure 12: LOADBYTE (subcircuit of the RISCV)

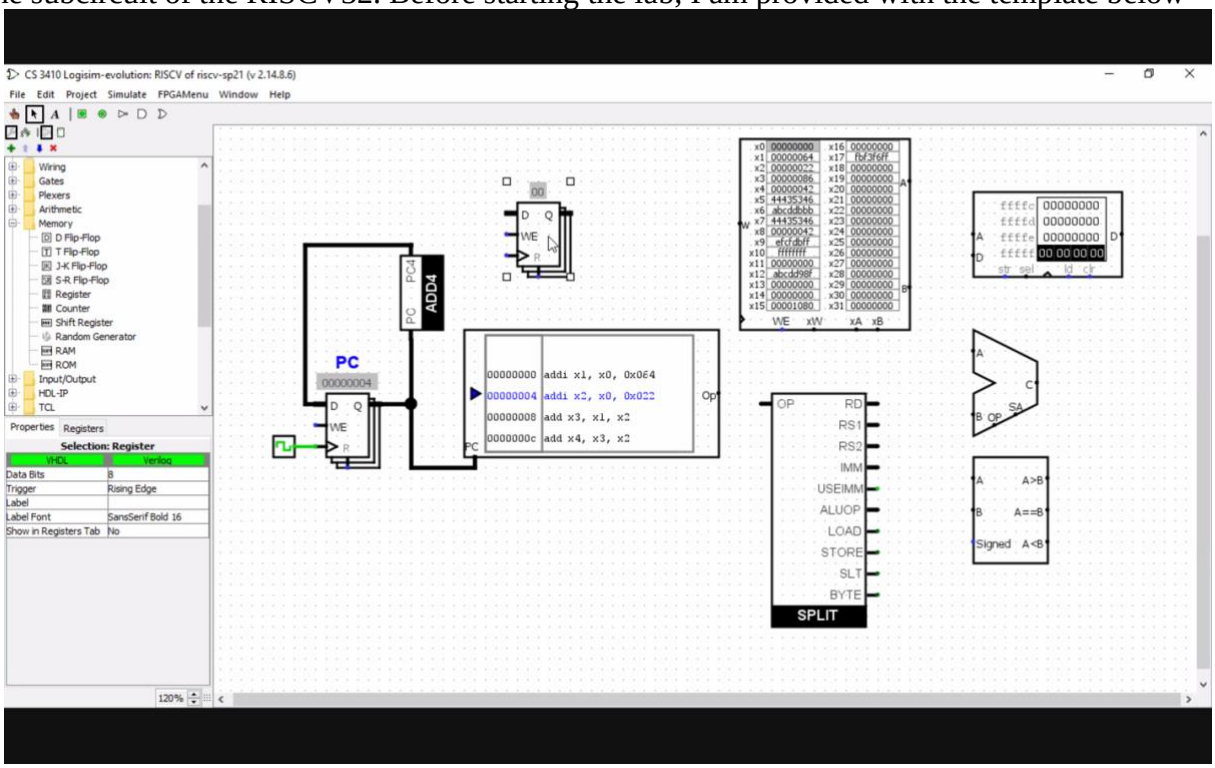


Description: This specialized circuit is to function the LOAD Operation and select whether it is the LW or LB

Evaluation: Those above figure and their description are the big ideas to show the connection of each component and the link between the RISCV32 and the small subcircuit.

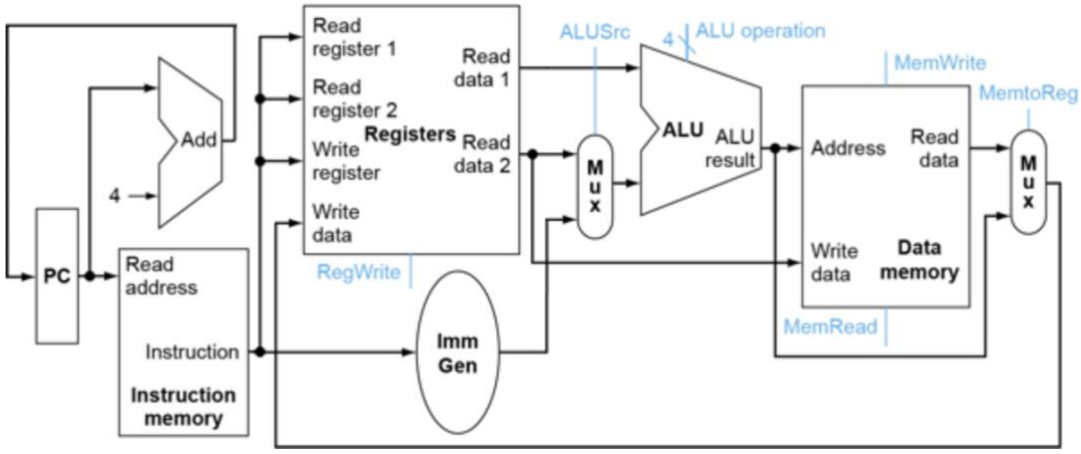
b. <u>Design Justification</u>

To approach this establishment, my design decision is boxing the similar features of RISCV32 or the subcircuit of the RISCV32. Before starting the lab, I am provided with the template below



- The main part we have to do is to create the subcircuit of the ADD4 as well as the SPLIT in order to complete the full single-cycle RISCV.

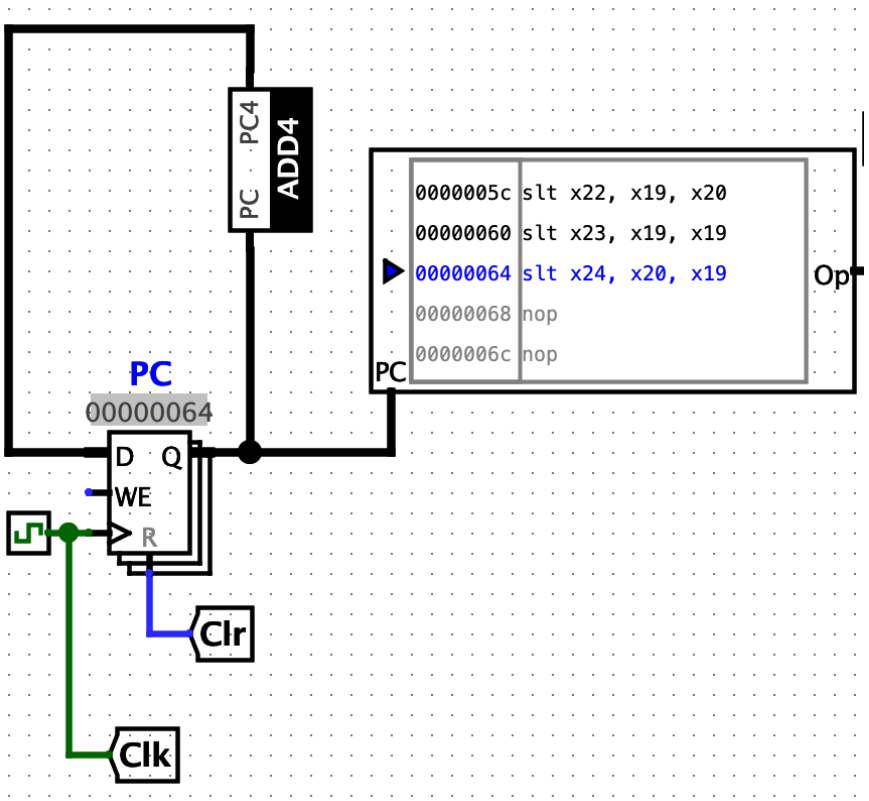
- The visualization of the Processor is provided in the Project 2 content in order to understand how the relationship of each component works



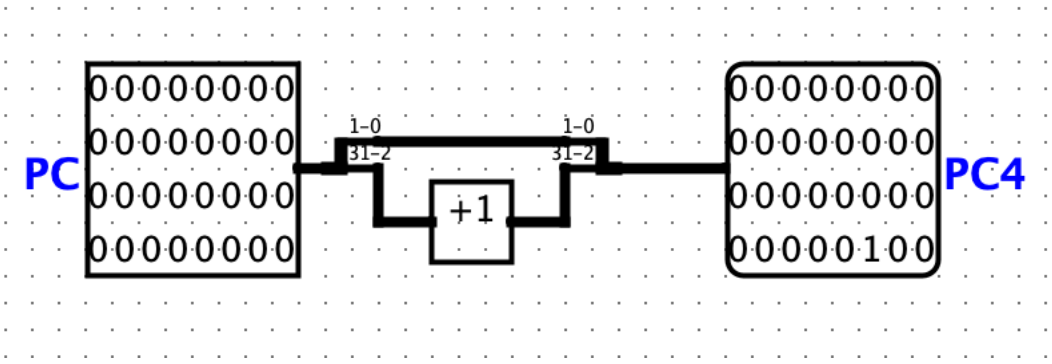
- In the lecture, we have learned 5 stages of the RISCV (Fetch, Decode, Execute, Memory Access, Write Back). From each stage, we have to perform each kind of different logical building. Thus, in the section III, I will deliver each stage in order aligned with the order of the diagram above.

III. Implementation Details

a. <u>FETCH</u>

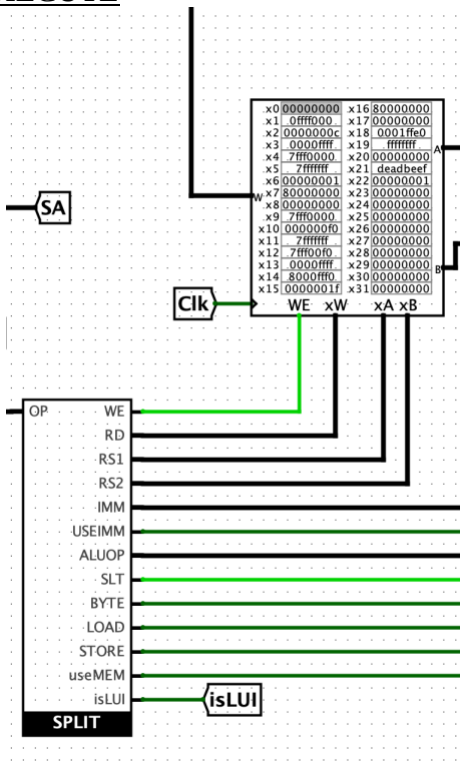


- During the Fetch Stage/Cycle, according to the lecture, the idea is to increment the number of the program counter by 4. Thus, I create the “ADD4” component is used to add 4 from each instruction
- The visualization below is the building of increment by 4 (in number).



- To perform this, I split the 32 bit into 2 first bits and 30 remaining bits. In order to increment by 4, just add 1 on the bit 3 since 100 equals 4 (in number).

b. <u>DECODE & former-EXECUTE</u>



- During the Decode Stage, the focus is to create the SPLIT Component since it serves as the controller instead. Thus, I have to analyze the SPLIT Component.
- From the previous section, I notice that there are some hint outputs provided in (IIb) such as RD, RS1, RS2, IMM, USEIMM, ALUOP, LOAD, STORE, SLT, BYTE.
- The reason I combine EXECUTE stage in this subsection because in the SPLIT Component, there exists the ALUOP is to determine which function performed in the ALU.

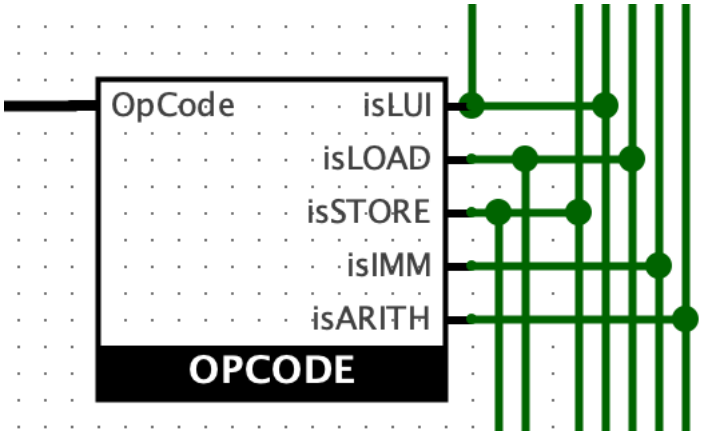
- Considering the ALU Op, there are 7 operations (OP) such as AND, OR, SHIFT LEFT LOGICAL, XOR, SHIFT RIGHT ARITHMETIC, SUBTRACT, ADD.

- On the other hand, during the DECODE STAGE, there are 18 different functions are needed to perform. Thus, looking up to the RISCV Manual on the page 104, we have this table

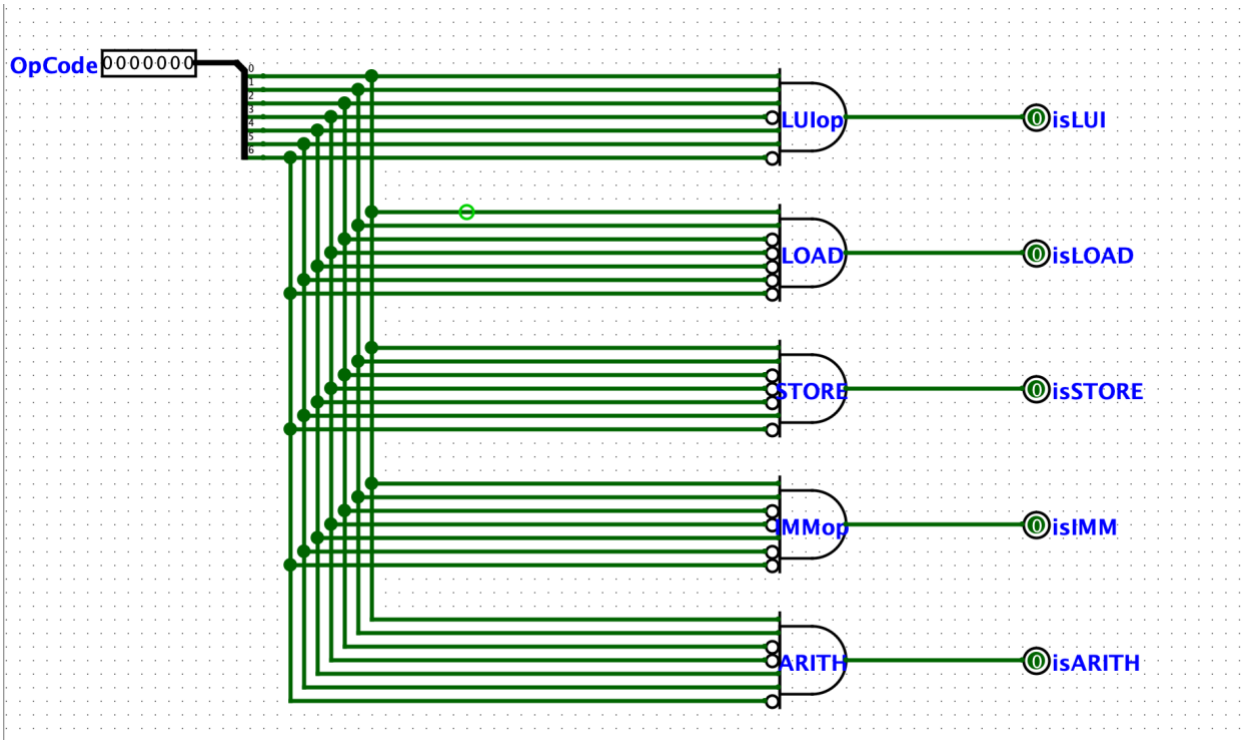
Instruction Name	imm/funct7	rs2	rs1	funct3	Imm/rd	Op
LUI	IMM[31:12]				RD	0110111
LB	IMM[11:0]		RS1	000	RD	0000011
LW	IMM[11:0]		RS1	010	RD	0000011
SB	IMM[11:5]	RS2	RS1	000	IMM[4:0]	0100011
SW	IMM[11:5]	RS2	RS1	010	IMM[4:0]	0100011
ADDI	IMM[11:0]		RS1	000	RD	0010011
XORI	IMM[11:0]		RS1	100	RD	0010011
ORI	IMM[11:0]		RS1	110	RD	0010011
ANDI	IMM[11:0]		RS1	111	RD	0010011
SLTI	IMM[11:0]		RS1	010	RD	0010011
ADD	0000000	RS2	RS1	000	RD	0110011
SUB	0100000	RS2	RS1	000	RD	0110011
AND	0000000	RS2	RS1	111	RD	0110011
OR	0000000	RS2	RS1	110	RD	0110011
XOR	0000000	RS2	RS1	100	RD	0110011
SLT	0000000	RS2	RS1	010	RD	0110011
SLL	0000000	RS2	RS1	001	RD	0110011
SRA	0100000	RS2	RS1	101	RD	0110011

Table 1: The Instruction Binary Number (Adapted from RISCV Manual)

- Recognition of instructions: With different Op (Operation Code), I divide those into 5 categories based on the difference of the OpCode. Thus, I create a subcircuit to output the recognition of those instructions.

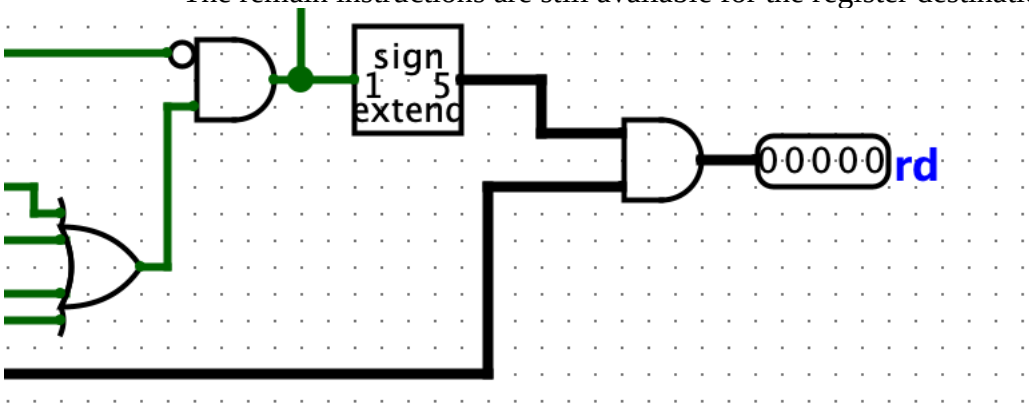


  - To visualize each detail parts, go deeper into its subcircuit



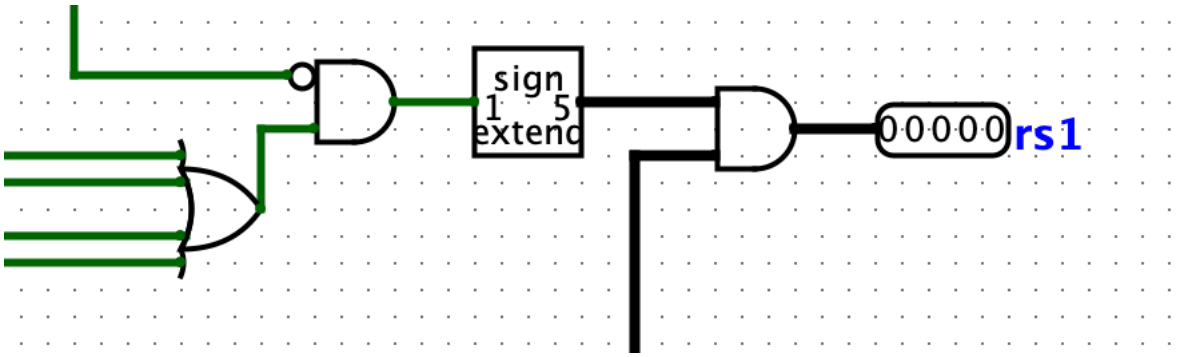
- Based on each Opcode, to generate its recognition, I used all AND Gates (combine the input and inverse its input). Later, if we click the right OpCode, it will generate 1, which means for the notification of that instruction.

- Register Generation: From the Table 1, with different grouping, we have different RD or RS1 or RS2. Hence, we must consider case by case for each instruction.
  - The input received is bit from 7 to 31 (exclude the Op)
  - RD:
    - Only isStore does not have the register destination (RD).
    - The remain instructions are still available for the register destination (RD)

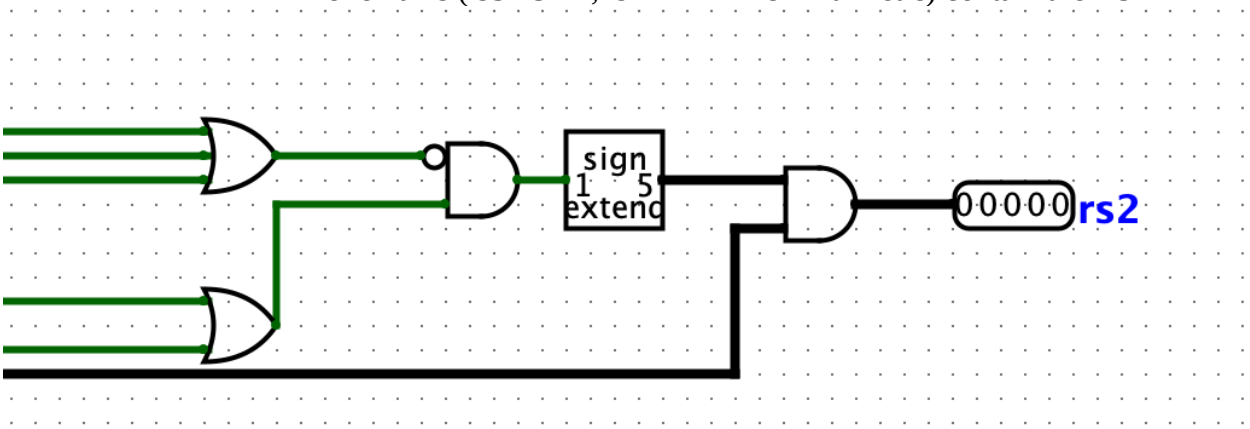


    - Whenever the instruction is not STORE, the output for extending 1 to 5 is 11111. Then, the inputs for the last AND gate are the previous condition (considering whether is STORE or not) and the address of the register destination.
    - If the input above is 11111, it will take the value contained the address in the RD. Otherwise, there is no value towards the output RD.

- RS1:
  - Only isLUI does not have the RS1 (Register Source 1)
  - The remain instructions do have address for RS1.



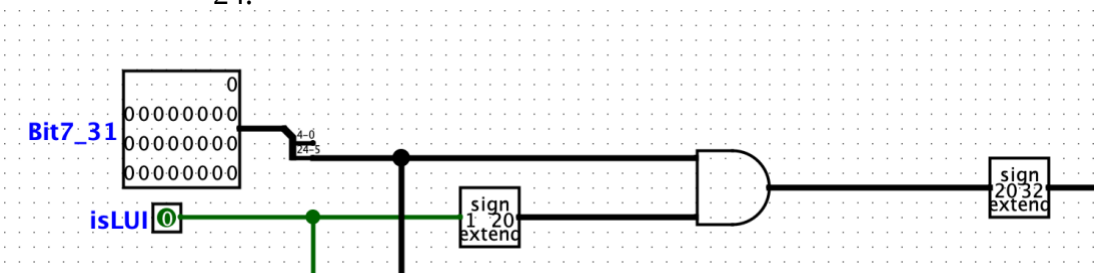
  - Same idea for building the logic gates as RD
  - The output is received if its instruction is not LUI
- RS2:
  - Instruction isLUI, isLOAD, isIMM (is Immediate instruction) do not have the RS2 (Register Source 2)
  - The remains (isSTORE, isARITH – is Arithmetic) contain the RS2



  - Same idea building as RD, RS1
  - With the above condition, we will generate RS2 in the case of need to have RS2
- RegWrite: Condition is the same as RD since if we do not write in the register file, we do not need to enable the RegWrite (Register Write), which has the same function as WE (Write Enabled)

- Immediate Notification and Generation (IMM):
  - The input into the IMM component is from the bit of 7 to 31 (excluded the 7 Op)
  - *Evaluation:* There are 3 different types of immediate based on the number of bit in immediate values. For instance, isLUI has 20-bit immediate (IMM[31:12]) or isLOAD & isIMM have 12-bit immediate (IMM[11:0]) or isSTORE contains 7-bit immediate (IMM[11:5]). Therefore, we must consider case by case each category for different number-in-bit of the immediate
  - isLUI (20-bit immediate)

- Taken the last 20-bit from 25-bit inputs, which means take the bit 5 to bit 24.



- If the instruction is LUI, we extend number 1 to 20-bit of 1. Hence, from the AND Gate, if there is all number 1 in the input, the value depends of all other bits input. Thus, the result would be the 20-bit from the Bit7_31. As we all need the immediate to generate outside the SPLIT, we have to extend it into 32-bit immediate.

  - isLOAD or isIMM (12-bit immediate)
    - The same idea as isLUI
    - Furthermore, two of the operations share the same instruction so that we have to use OR gate and extend to 12-bit afterwards to determine whether taken the last 7 bit from bit 7 to bit 31.
    - Lastly, extend to 32-bit for the immediate for the need outside the SPLIT
  - isSTORE (7-bit immediate)
    - The same idea as isLOAD and isIMM
  - useIMM: is the generation bit for the usage in the RISCV to notice whether use immediate or not
- LOAD: is the generation bit for the distinguishing whether load or store in the RAM (Memory), which base is on the isLOAD from the OPCODE
- STORE: is the generation bit for the distinguishing whether load or store in the RAM (Memory), which base is on the isSTORE from the OPCODE
- useMEM: is the generation bit for the statement usage of LOAD or STORE
- BYTE: is the generation bit for the distinguishing LW or LB whether SW or SB
- isLUI: is the generation bit for the usage of the LUI Instruction

c. EXECUTE

- The connection of the DECODE stage and the EXECUTE stage is the ALUandMEM since there are ALUOP inside the SPLIT Compon

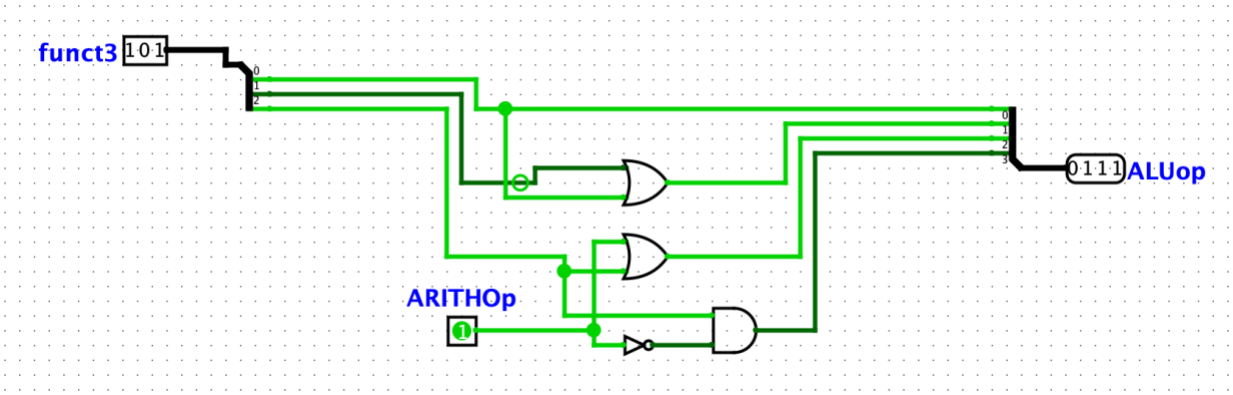
Op	name	C	V
1111	and	C = A & B	V = 0
1110	or	C = A | B	V = 0
001x	shift left logical	C = A << Sa	V = 0
1100	xor	C = A ^ B	V = 0
0111	shift right arithmetic	C = A >> Sa	V = 0
010x	subtract	C = A - B	V = overflow
000x	add	C = A + B	V = overflow

Table 2: ALU Table

- To design the ALUandMEM subcircuit, I have to consider every function in the ARITH, LUI, LOAD, STORE, IMM
  - ARITH: generate the ALUOP as in the ALU table
  - LUI: generate the ALUOP as the Shift Left Logical
  - LOAD: generate the ALUOP as in the ADD operation in the ALU Table since need to address the content of RS and the IMM
  - STORE: generate the ALUOP as in the ADD operation in the ALU Table since need to address the content of RS and the IMM
  - IMM: generate the ALUOP as in the ALU Table
- ARITH_OP

	Funct3_0	Funct3_1	Funct3_2	ARITH	ALU_0	ALU_1	ALU_2	ALU_3
ADD	0	0	0	0	x	0	0	0
SUB	0	0	0	1	x	0	1	0
AND	1	1	1	0	1	1	1	1
OR	0	1	1	0	0	1	1	1
XOR	0	0	1	0	0	0	1	1
SLL	1	0	0	0	x	1	0	0
SRA	1	0	1	1	1	1	1	0

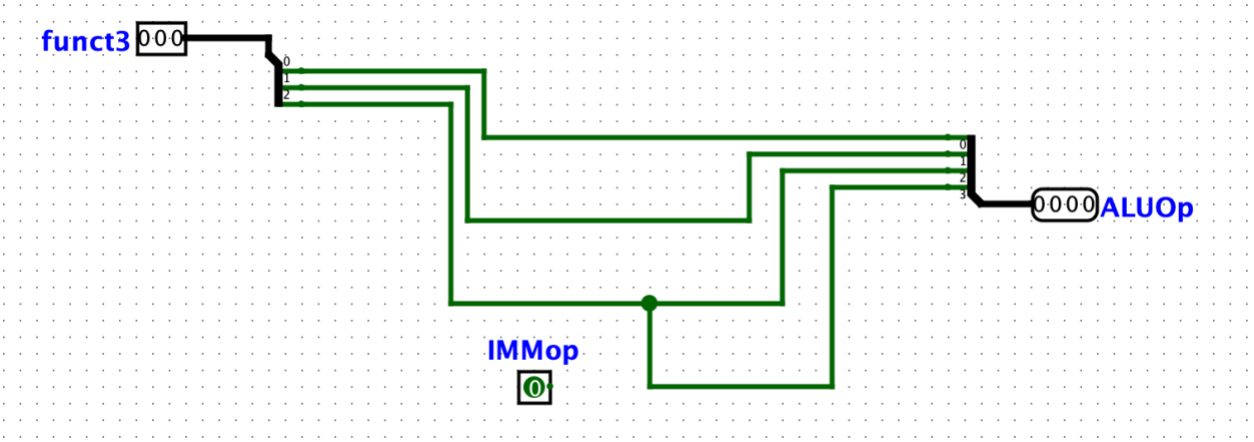
From the Truth Table, we can generate the subcircuit for the ARITH_OP by using the Combinational Analysis to build the circuit



- IMM_OP

	Funct3_0	Funct3_1	Funct3_2	IMM	ALU_0	ALU_1	ALU_2	ALU_3
ADDI	0	0	0	1	x	0	0	0
XORI	0	0	1	1	0	0	1	1
ORI	1	1	1	1	0	1	1	1
ANDI	1	1	1	1	1	1	1	1

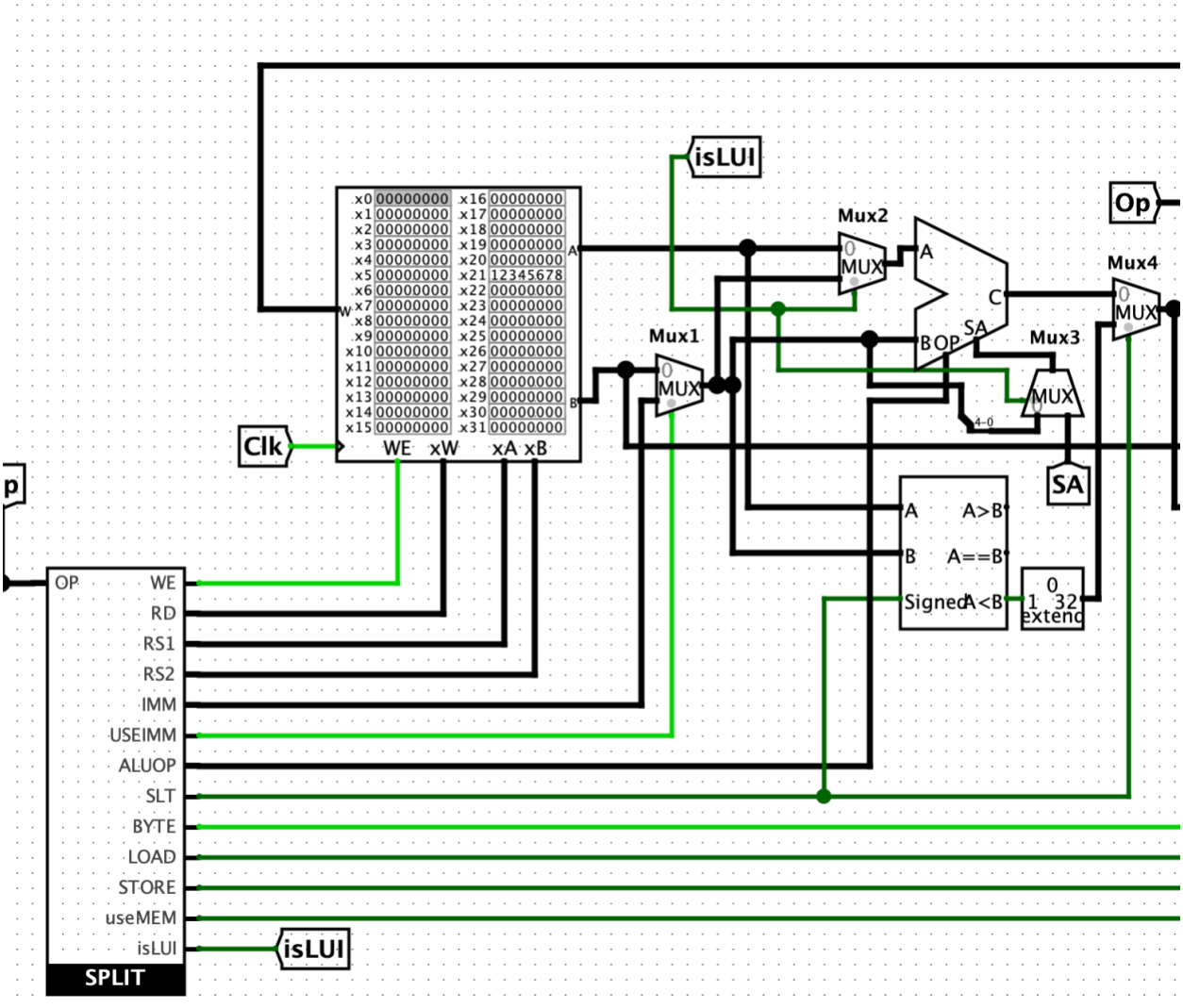
From the Truth Table, we can generate the subcircuit for the IMM_OP by using the Combinational Analysis to build the circuit



- LOAD & STORE Operation: Since its operation perform the ADD Operation, I must generate it in the ALUOP the same as the ALU Table of ADD (000x)
- LUI: Since its operation perform the SLL Operation, I must generate in the ALUOP the same as the ALU Table of SLL (001x)
- The same idea for choosing which would be chosen to generate the ALUOPERATION, I use MUX to differentiate whether the using one
    - ⇨ Lastly, its results will be the ALUOP based on the type of instructions from the input at the beginning

- SLT Operation: As SLT does not contain in the AlU Table, I have to split it out in another operation to consider in the main RISCV, which is input into the Comparator Blackbox
- isBYTE: I use the funct3 to examine whether it is Word or Byte. If it is a Byte, its funct3 would be 000. Otherwise, it is 010. Hence, its output to generate the bit for notification of using Byte or Word

COMBINATION FOR THE SPLIT & ALU & COMPARATOR BLACKBOX

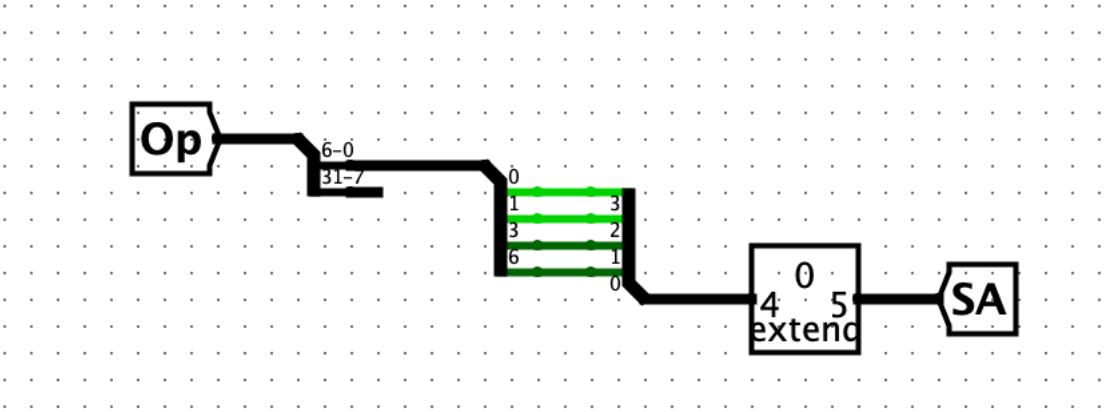


- Among the DECODE and EXECUTE stages, we connect all the output from the SPLIT Component towards the Register File (DECODE) & ALU + Blackbox Comparator

SPLIT Component	Another component	Purpose
WE	WE (Register File)	To enable the write enable (WE) in the Register File (isStore does not notify the WE)
RD	xW (Register File)	Execute the address of the RD to the xW in the Register File
RS1	xA (Register File)	Execute the address of the RS1 to the xA in the Register File
RS2	xB (Register File)	Execute the address of the RS2 to the xB in the Register File

IMM	The input for MUX1	IMM is the 32-bit number to use in the MUX1 as there are some instructions above
USEIMM	The select bit for MUX1	The usage for the select bit to notify it contains immediate or not
ALUOP	The ALU	Give out 4-bit of ALUOP (detailed explanation above of generating the ALUOp)
SLT	The Signed for Blackbox Comparator & The select bit for the MUX4	The notification for the Signed in the Blackbox Comparator and its select bit for the MUX4
BYTE	Use for SelectByte Component (description below)	To differentiate whether Word or Byte to input in the SelectByte and Data-IN and LOADBYTE
LOAD	Use in the RAM Memory	The notification for the RAM of having LOAD instruction
STORE	Use in the RAM Memory	The notification for the RAM of having STORE instruction
useMEM	The Select Bit for the MUX5	The notification for accessing into the RAM Memory
isLUI	The Select Bit for the MUX2	To recognize it is used for the LUI Instruction

- isLUI: When going into the ALU, its function is different from the Shift Left Logical as it needs to shift 12 binary bit whether it is not any 5 last bit for the original Shift Left Logical.
    - Establishing the MUX2 in order to let the immediate for the place of input A of ALU with the notification of select bit from isLUI in the SPLIT Component
    - Esatablishing the MUX3 in order to let the SA with the constant 12 (as LUI has to be shifted 12 bit = 3 hexdecimal bits) instead of 5 hexadecimal in general
    - To build the constant for the SA in the LUI operation

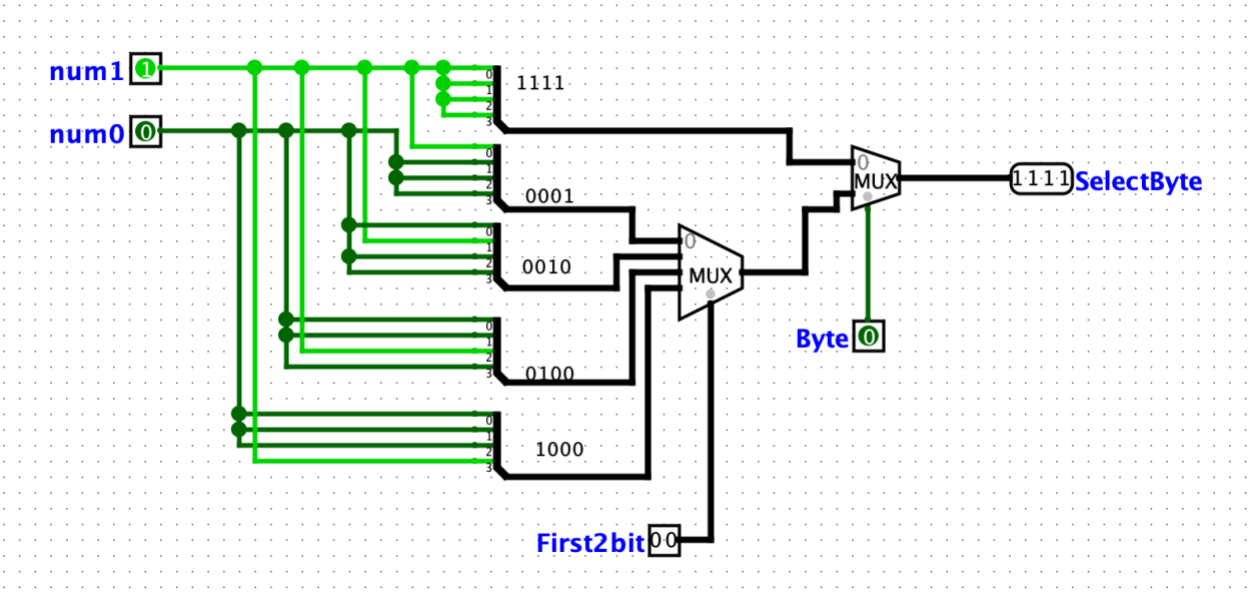


- SLT: This instruction has to go through the Blackbox Comparator as the ALU does not operate this function
    - Easily applied the Blackbox Comparator from the COMP2020 Component
    - Its output is needed to extend through 32-bit
    - Establishing the MUX4 to enable the select bit from SLT output from the SHIFT Component

d. <u>MEMORY ACCESS</u>

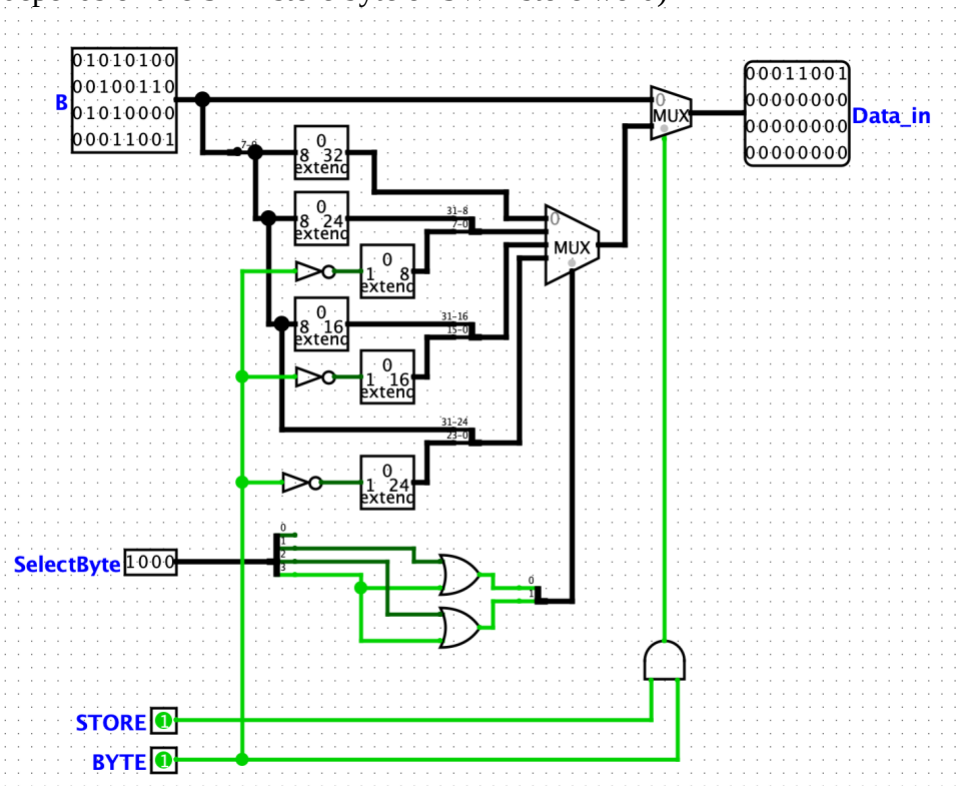
During the Memory Access stage, the focus is to put in all the inputs through RAM Memory (SelectByte, LOAD, STORE, Address, Data, Clock, Clear) and the output would be the 32-bit data from the Memory

- Clock: The same clock as in the Program Counter in the FETCH STAGE
- Clear: Input 1-bit if we want to all clear the Program Counter and the RAM Memory
- LOAD: Use the output from the SPLIT Component
- STORE: Use the output from the SPLIT Component
- Address: From the result computed in the MUX, I take the bit-2 to the bit-21 for giving in the address to the RAM Memory by using splitter
- SelectByte: From the Op in the Program Memory, I use the bit 0 as number is always 1 and bit 6 is always 0 to generate the select byte. There are four cases in the Select Byte (1111 – full word, 0001-the first 2 bytes, 0010-the second 2 bytes, 0100-the third 2 bytes, 1000-the fourth 2 bytes)
    - To determine the selection for choosing which order to use the SelectByte, I use the first 2 bits in the output from the MUX4.
    - The input for the SelectByte Component is num1 (always 1), num0(always 0), First2bit, BYTE from the SPLIT Component

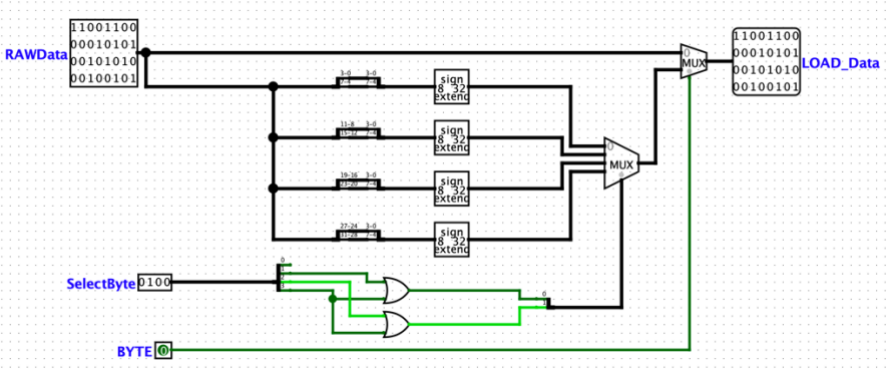


- DATA:
    - Data-IN: The function for the STORE is needed to select which 2 bytes for storing in the Memory
        - The idea is to split the 8 bits once at a time since 8 bits equals 2 bytes. Then, for each 8 bits from the B (RS2), we have to test whether it does not have the BYTE. Then , it can get the output into the MUX.
        - To select which 8 bits in the B, use the SELECTBYTE from the main circuit
        - Then, if there is a BYTE usage, it can enable the last MUX in order to take the value of each 8 bit into the DATA_IN. Otherwise, it stores the whole word (32 bit = 4 bytes)

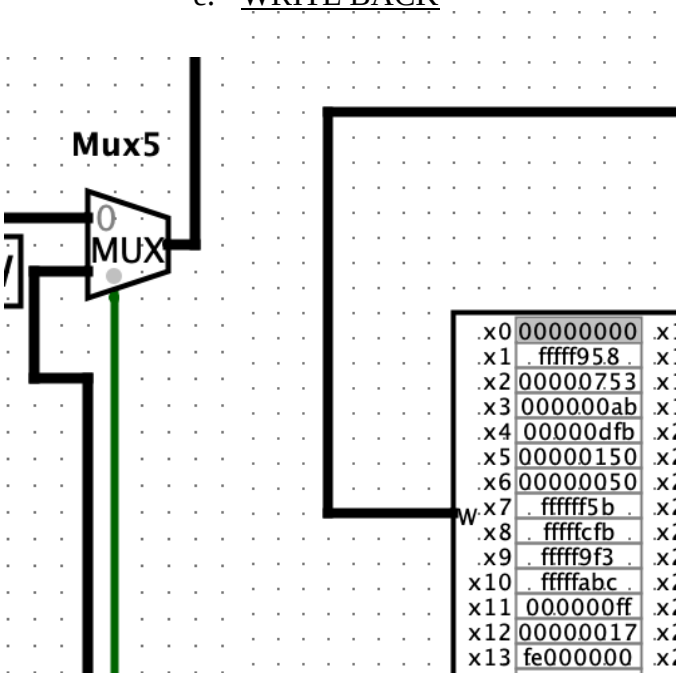
- When going to the DATA_IN, it will go to the RAM Memory to store in its address and value of storing based on whether the SelectByte is (which depends on the SB – store byte or SW – store word)



- LOAD_BYTE
  - After accessing in the RAM, if the function is LOAD, there would be output from the Data in the RAM.
  - The RAW DATA would be accessed into the LOADBYTE Component.
  - If there is not any notification of the BYTE, all 32-bit would be given out to the register.
  - Otherwise, they will split each 8 bit (same idea as of DATA_IN). Thus, based on different SELECTBYTE, we can select which 8 bit in order to write in the Register File



e. WRITE BACK



- The MUX5 is to determine whether which operation is used whether the Arithmetic or the Memory Access by having the notification bit of “useMEM”
- If the “useMEM”=1, they will take the value LOAD into the Register. However, the STORE could not be written since “WE” can not enabled into the Register File
- If the useMem = 0, there would be the ordinary ARITHMETIC, thus, all the values can be written on the Register File.

IV. Reference

[1] P2: RISCV on Canvas Page
[2] Lectures in COMP2020