**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY**

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL ELECTRONIC ENGINEERING

**Advanced Program**

---------------o0o---------------

****

**CAPSTONE PROJECT II**

**EMULATING THE**

**RV32I CPU INSTRUCTION SET ARCHITECTURE**

**Instructor: Ph.D. Trần Hoàng Linh**

**Students: Lý Chí Học 1810930**

**Phạm Tấn Khải 1851081**

**HO CHI MINH CITY, DECEMBER, 2021**

**ACKNOWLEDGEMENT**

Thanks to Capstone II at Ho Chi Minh University of Technology, we got more experience and knowledge from many enthusiastic teachers and full-experienced people. Through this project, we have learned a lot ,the applications of the theory in the class, and the specific contributions that we can pursue.

We would like to thank Ph.D. Tran Hoang Linh, who gave us not only academic knowledge but also many precious advices. He provided us with invaluable advice and helped us in difficult periods. His motivation and help contributed tremendously to the successful completion of the project.

Besides, we would like to thank all the teachers in Electrical-Electronics Engineering faculty who helped us by giving us advice and providing the equipment which we needed. Also I would like to thank my family and friends for their support. Any attempt at any level can ‘t be satisfactorily completed without the support and guidance of my parents and friends.

At last but not in least, we would like to thank everyone who helped and motivated us to work on this project.

*Ho Chi Minh city, 23th December, 2021.*

**Lý Chí Học**

**Phạm Tấn Khải**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **ABSTRACT** | **………………………………………………………………………………………** | **5** |
| 1. **INTRODUCTION** | **………………………………………………………………………………………** | **6** |
| 1. **overview** | **………………………………………………………………………………………** | **6** |
| 1. **Project’s mission** | **………………………………………………………………………………………** | **7** |
| 1. **Member’s work** | **………………………………………………………………………………………** | **8** |
| 1. **THEORY** | **………………………………………………………………………………………** | **9** |
| 1. **Programmers’ Model for Base Integer ISA** | **………………………………………………………………………………………** | **9** |
| 1. **Base instruction format** | **………………………………………………………………………………………** | **10** |
| 1. **Immediate Encoding Variants** | **………………………………………………………………………………………** | **11** |
| * 1. **The Register to register arithmetic and logical operation / R-type** | **………………………………………………………………………………………** | **12** |
| * 1. **The load and store immediate instructions/ I-type & S-type** | **………………………………………………………………………………………** | **13** |
| * 1. **U-type** | **………………………………………………………………………………………** | **18** |
| * 1. **Conditional Branches /B-type** | **………………………………………………………………………………………** | **20** |
| * 1. **Unconditional Jump/J-type** | **………………………………………………………………………………………** | **21** |
| 1. **EMULATE RV32I STANDARD DESIGN** | **………………………………………………………………………………………** | **22** |
| 1. **Specification** | **………………………………………………………………………………………** | **22** |
| 1. **Design** | **………………………………………………………………………………………** | **22** |
| * 1. **Memory** | **………………………………………………………………………………………** | **23** |
| * 1. **CPU** | **………………………………………………………………………………………** | **24** |
| * 1. **Testing an example and result** | **………………………………………………………………………………………** | **40** |
| 1. **CONCLUSION** | **………………………………………………………………………………………** | **50** |
| 1. **REFERENCES** | **………………………………………………………………………………………** | **50** |

**ABSTRACT**

In this research, we focus on RISC V, an [open standard](https://en.wikipedia.org/wiki/Open_standard) [instruction set architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture) (ISA) based on established [reduced instruction set computer](https://en.wikipedia.org/wiki/Reduced_instruction_set_computer) (RISC) principles which was originally designed to support education and computer architecture research but now become an [open-source](https://en.wikipedia.org/wiki/Open_source_license) architecture for industry implementations. Due to our non-experience and time limitation, in this capstone II, we only focus on emulating RISC-V instruction RV32I. This simple architecture is designed to implement basic RISC-V instructions and may help us to develop in the future.

1. **INTRODUCTION**
2. **Overview:**

In 2010, Berkeley, an expert at the University of California, developed an open-source ISA RISC-V that may be utilized in a variety of applications at a low cost. The RISC-V instruction set has offered several chances for universities and individuals to construct their own CPU architectures. RISC-V is not over-optimized for a specific application, and it can be expanded to the user's application, that can help chip-development companies reduce expenses. RISC-V enables chipmakers to design a SOC that offers more power in a smaller space than previously conceivable with its simple instruction set. The major challenge with RISC-V is that the software ecosystem is too small to accommodate this instruction set format. RISC-V Computer Architecture has been the most studied topic in universities throughout the world by the majority of undergraduates, lecturers, and professors. RISC-V, on the other hand, is an open standard and platform, with a progressively rising share in the CPU core market estimated to reach 160 percent from 2018 to the end of 2025, according to Semico Research. Now, the next step the RISC-V evolution is developing RISC-V for high-performance compute (HPC). [RISC-V International announced that it would like to extend its reach into the world of data centers](https://riscv.org/blog/2021/06/risc-v-sig-hpc-enabling-risc-v-in-hpc-supercomputers-to-the-edge-and-emerging-ai-ml-dl-hpc-workloads/) with a focus on applications including machine learning. This makes RISC-V more economically appealing and also become a useful tool for implementation to produce RISC-V based devices

In Viet Nam, there are limited number of research about RISC-V because it is still something new in semiconductor industry in Viet Nam. However, in RISC-V Day Online Viet Nam 2020, Do Ngoc Huynh and Nguyen Hung Quan from VLSI TECHNOLOGY PAGE, they want to create the first open-source RISC-V SoC project in Vietnam that create education environment for young engineer and value for the VLSI community and encourage more people to learn more about the IC. University in Vietnam realized the promise of the semiconductor industry and created a training program for engineers specializing in chip and computer architecture design. To understand RISC-V ISA, we should achieve basic knowledge of RISC-V and method to emulate RISC-V instruction R32i.

1. **Project’s mission:**

In this Project, we focus on the implementation of the working functionality of RISCV instructions in this scope is the standard rv32i (integer base). The main mission is to emulate 40 instruction and put them through manual test to get the output from the acquire input, combining the knowledge of functional programing and the CPU architecture to produce the similar behavior of a real simulation on hardware description language. Verifying the outcome through monitoring the printed data that we get through the C/C++ program. And at the end of the project we would evaluate the method of adapting emulation of an ISA on such language as C/C++.

1. **Member’s work:**

We carry the work throughout 12 weeks, in the process we encounter many obstacle and gaining experiences also the ways to do solve the problem that we managed to discover.

|  |  |
| --- | --- |
| **Lý Chí Học** | **Phạm Tấn Khải** |
| Topic prepare and work load assigning | |
| * Research on theory of how the component Works in a data path of the rv32i. * Design the component as object in the C/C++ with multiple characteristics. * Planning ahead the work load for the group. * Code and check error or bugs in the execution function. | * Code components such as execution function, fetch function, the MEM, the CPU, research on matter that revolve around how data flow in the branch. * Checking the full manual pseudo ASM code in the main() function. |
| Both of the member done the writing and create presentation for the final | |

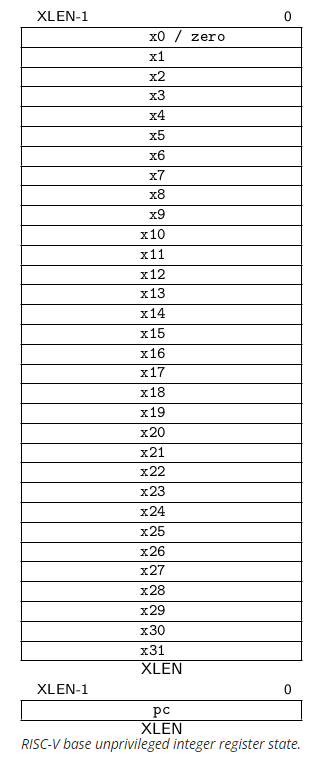
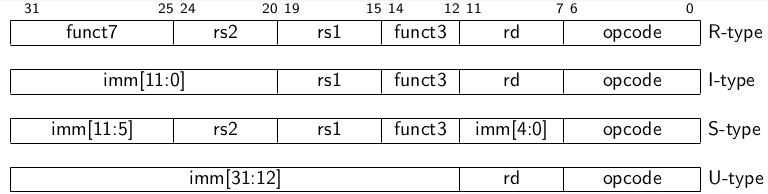
1. **THEORY**
2. ** Programmers’ Model for Base Integer ISA**

Figure above shows the unprivileged state for the base integer ISA. For RV32I, the 32 x registers are each 32 bits wide, i.e., XLEN=32. Register x0 is hardwired with all bits equal to 0. General purpose registers x1–x31 hold values that various instructions interpret as a collection of Boolean values, or as two’s complement signed binary integers or unsigned binary integers.

There is one additional unprivileged register: the program counter pc holds the address of the current instruction.

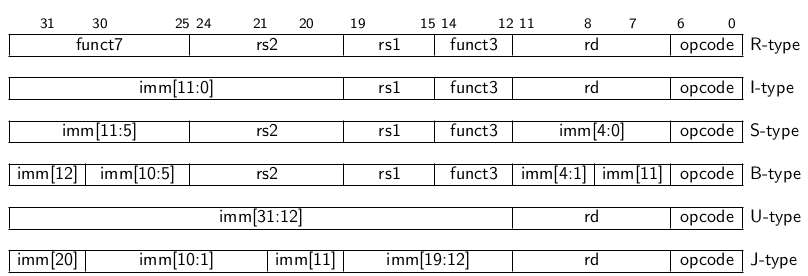
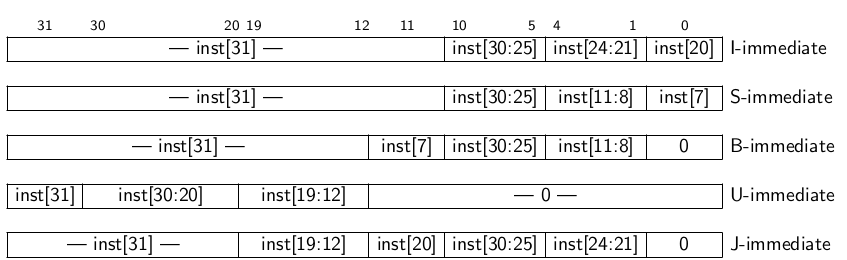
1. **Base Instruction Format:**

****In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure [below](https://five-embeddev.com/riscv-isa-manual/latest/rv32.html#fig:baseinstformats).

All have a fixed 32-bit length and must be aligned in memory on a four-byte boundary. If the target address is not four-byte aligned, an instruction-address-misaligned exception is generated on a taken branch or unconditional jump. This exception is detected on the branch or jump instruction rather than the target instruction. If a conditional branch is not taken, no instruction-address-misaligned exception is generated.

To facilitate decoding, the RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers in the same place in all formats. Except for the 5-bit immediates used in CSR instructions , immediates are usually sign-extended, are often packed towards the instruction's leftmost available bits, and were allocated to decrease hardware complexity. To speed up sign-extension circuitry, the sign bit for all immediates is always in bit 31 of the instruction.

1. **Immediate Encoding Variants:**

There are a more two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure [below](https://five-embeddev.com/riscv-isa-manual/latest/rv32.html#fig:baseinstformatsimm).

The only difference between the S and B formats is that the 12-bit immediate field in the B format is used to encode branch offsets in multiples of 2. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware, the middle bits (imm[10:1]) and sign bit remain constant, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

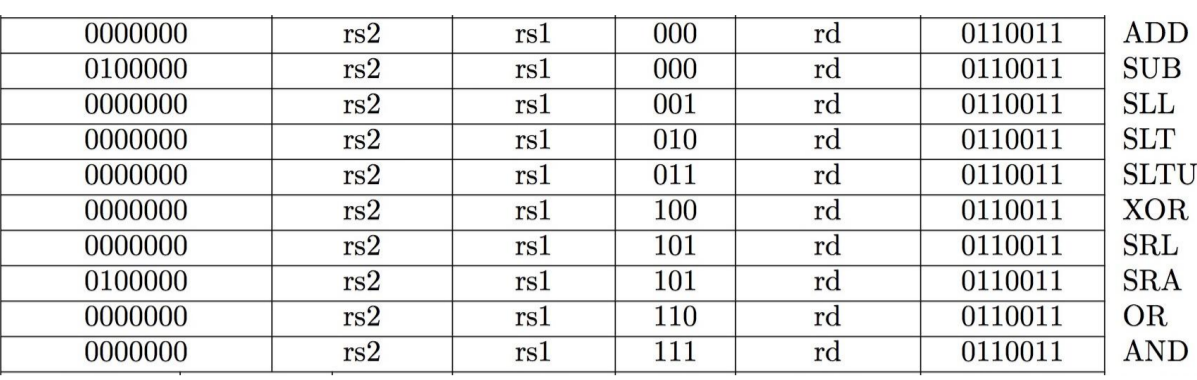
The only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to generate U immediates and right by 1 bit to form J immediates. The position of instruction bits in U and J format immediates is selected to maximize overlap with other formats and with each other.

* 1. **The Register to register arithmetic and logical operation / R-type:**

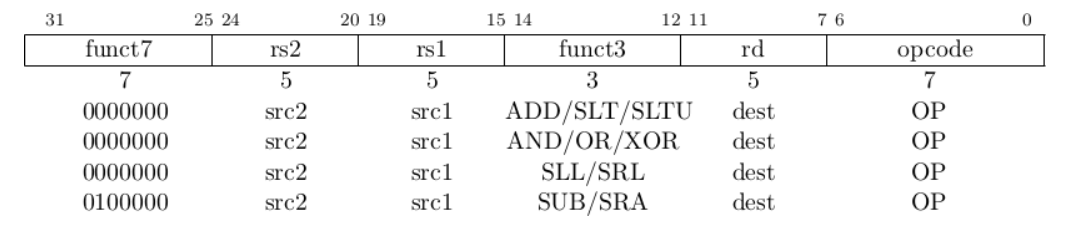
R-type instructions are used to assign a destination register rd to the result of an arithmetic, logical, or shift operation performed on source registers rs1 and rs2.



* **opcode** is an operation code or opcode that selects a specific operation
* **rs1** and **rs2** are the first and second source registers
* **rd** is the destination register
* **funct7** and **funct3** is used together with opcode to select an arithmetic instruction

*Implementing other R-Format instructions*

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



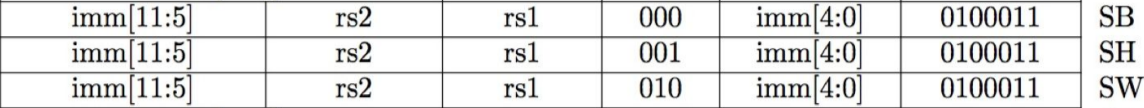
ADD performs the addition of rs1 and rs2. SUB performs the subtraction of rs2 from rs1. Overflows are ignored and the low XLEN bits of results are written to the destination rd. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rsc\_1 < rsc\_2, 0 otherwise. AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

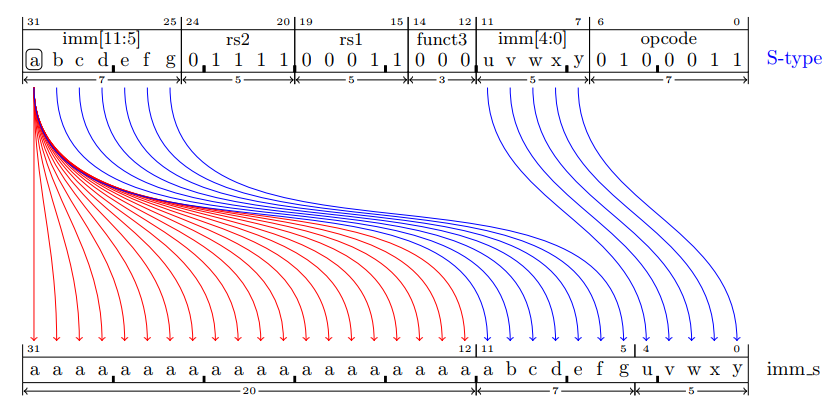
* 1. **The load and store immediate instructions/ I-type & S-type:**

***S-type instruction:***

S-type instruction is used for store instruction to encode instructions with a signed 12-bit immediate operand, an rs1 register, and an rs2 register

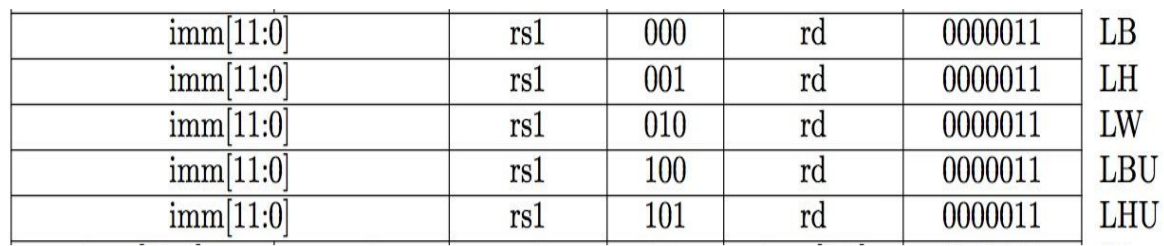
*****All RV32i Store instruction*

The detail format and also the way to generate immediate of S type are shown the below figures

****

***I-type instruction:***

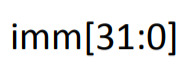
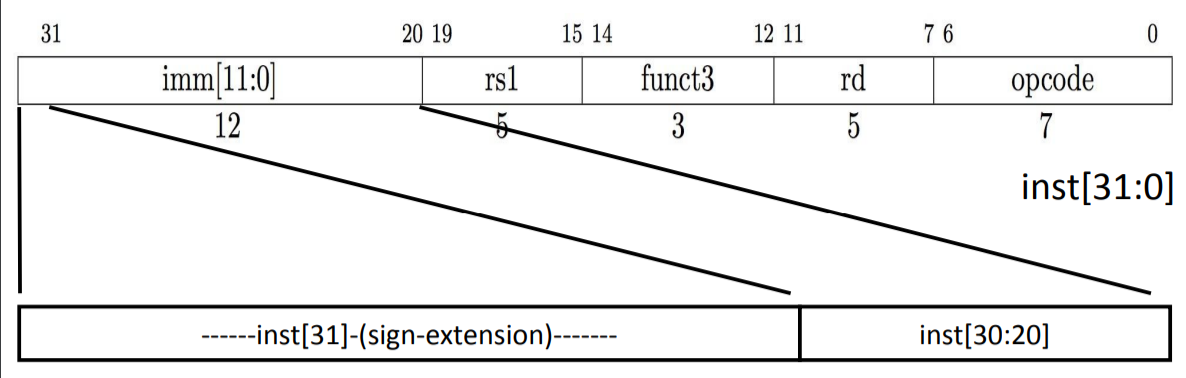
I-type instruction is used for load instruction to encode instructions with a signed 12-bit immediate operand, rd register and rs1 register.



*All RV32i Load instruction*

To generate I-type immediates:

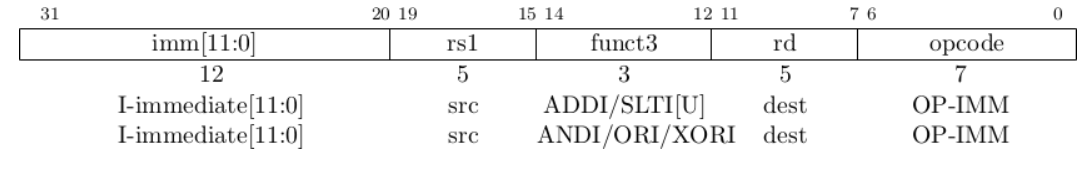
* High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
* Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

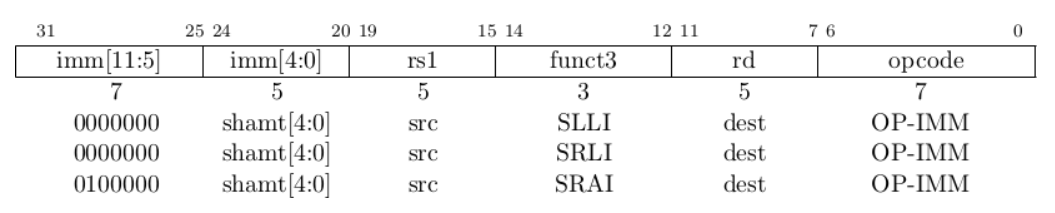


ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudo instruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1, 1* sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudo instruction SEQZ *rd, rs*).

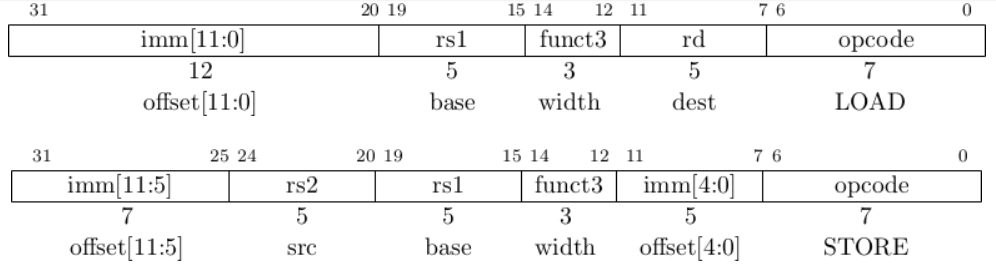
ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1* (assembler pseudo instruction NOT *rd, rs*).



Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

***Load and store instructions***

RV32i is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32i provides a 32-bit address space that is byte-addressed. Loads with a destination of x0 must still raise any exceptions and cause any other side effects even though the load value is discarded.



Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

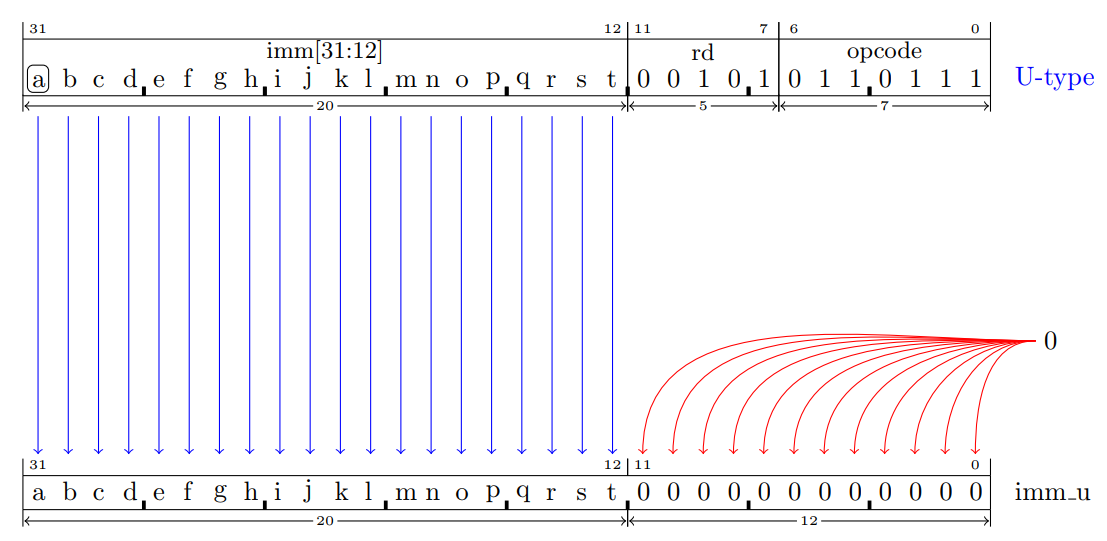
The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

* 1. **U-type:**

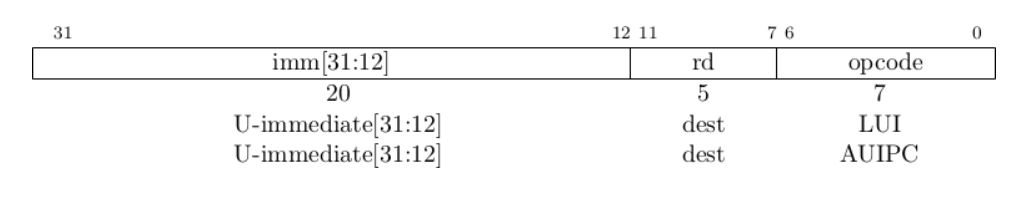
U-Type instruction is used for instructions which have a 20-bit immediate operand and an rd destination register.



The detail format and also the way to generate immediate of U type are shown the below figures



***LUI and AUIPC***

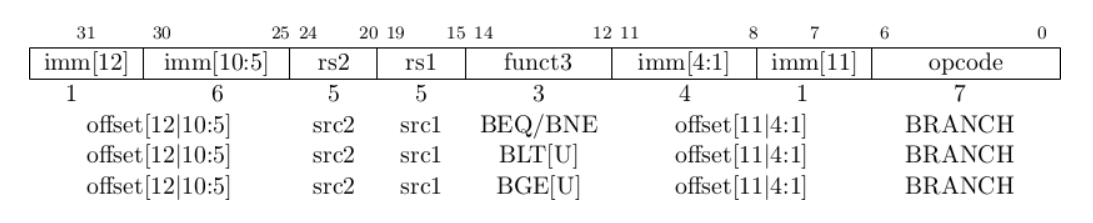


LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the 32-bit U-immediate value into the destination register rd, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register rd.

* 1. **Conditional Branches /B-type:**

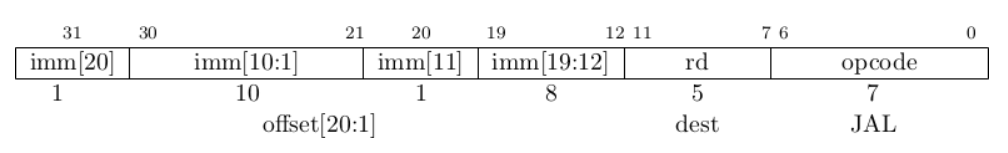
All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.



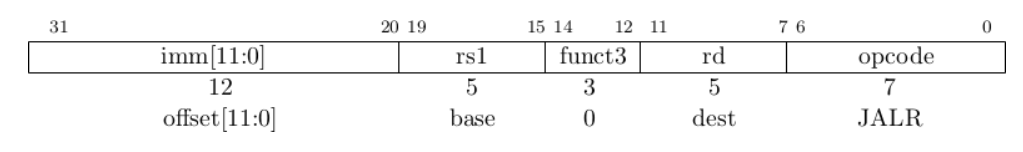
Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

* 1. **Unconditional Jump/J-type:**

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register *rd*. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.



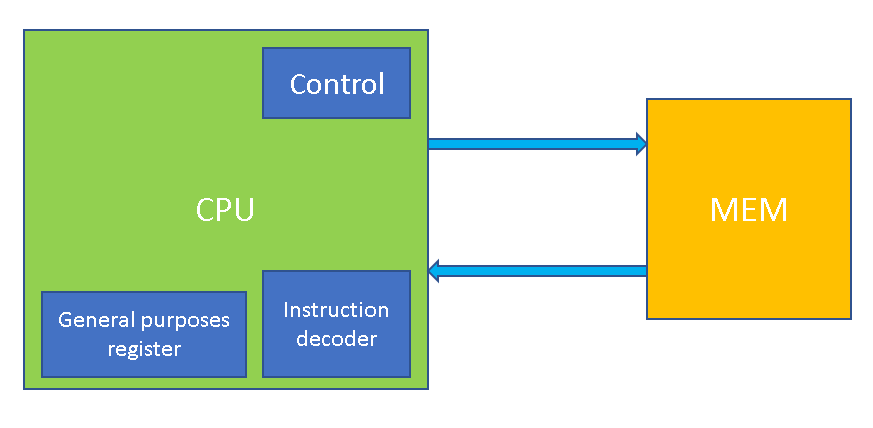
1. **EMULATE RV32I STANDARD INTEGER BASE DESIGN:**
2. **Specification.**

Our RV32I emulator characteristic:

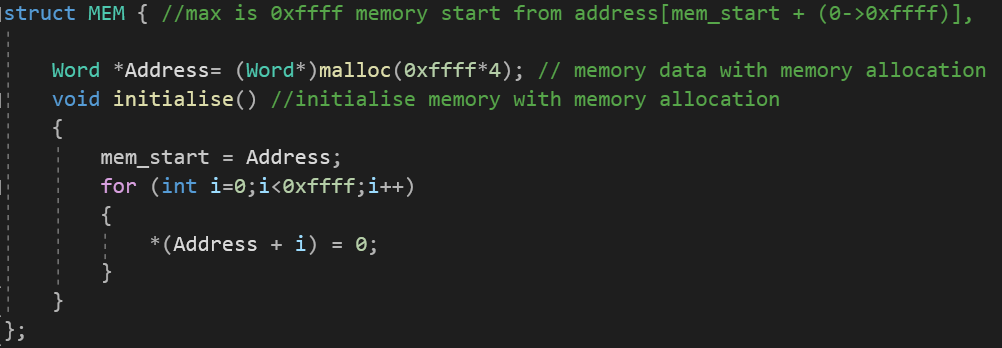
+ 32bits instruction set architecture (standard integer base).

+ **R**, **I**, **S**, **SB**, **U**, **UJ** instruction types are included in the emulator program.

Software used in this project is **Visual studio 2019**.

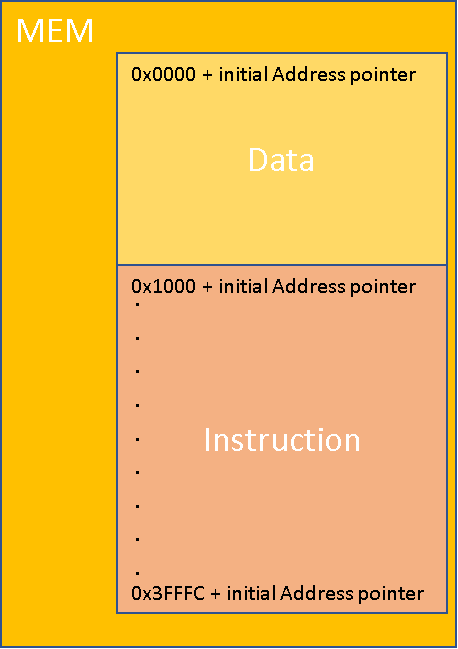
1. **Design of the program.**
   1. **Memory.**

Memory is built by creating a struct named “MEM” with allocated memory



The MEM contain an address pointer with “*4-bytes* wide *x 0xffff*” allocated memory, a function named “initialize” to reset and fill all the memory block to 0x0.

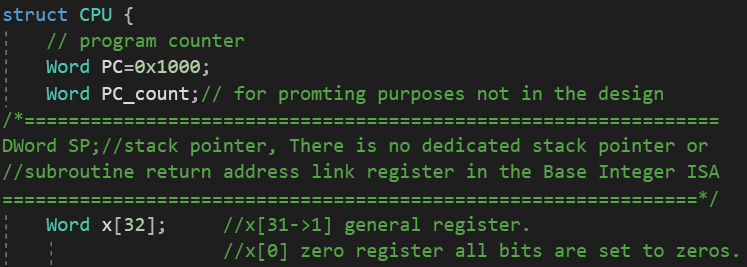
This memory will store the instruction needed for the program to load, also the value that we want to store or load data from it. The instruction will be store at the allocated Address with the address at 0x1000 + initial allocated address pointer.



* 1. **CPU.**

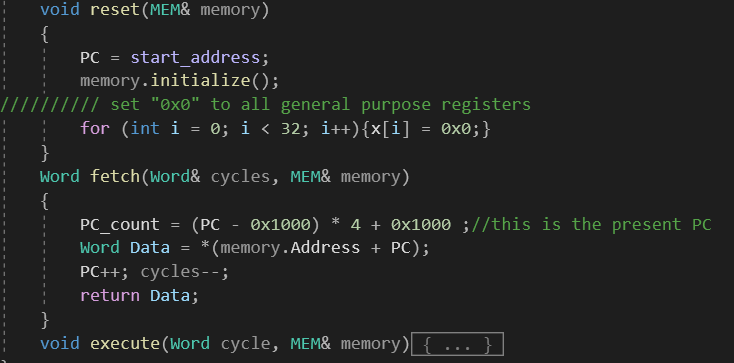
The ***cpu*** design contain:

* ***Program Counter***: defined by Word (unsigned integer), set to start at the value 0x1000.
* ***General purposes registers***: defined by Word (unsigned integer), with 32 32bits registers.

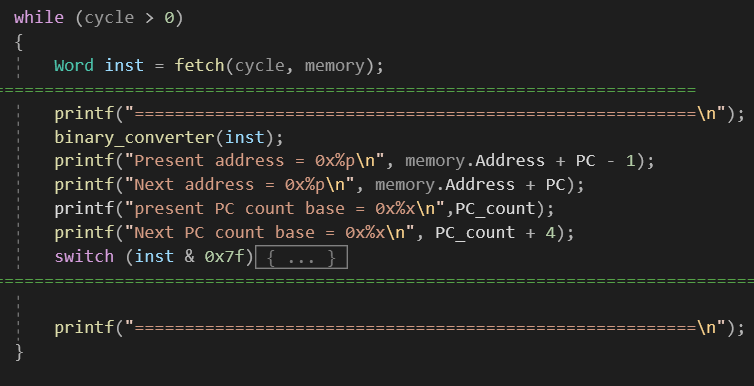


* A “***reset***” function to purposely to assign start-address value which is 0x1000, run the ***initialize*** sequence and a for loop to set all General Purposes Register to 0x0.
* ***Fetch*** function that returned ***instructions*** to load to the ***control*** which is the ***execute*** function also created inside the object ***CPU***.
* ***Execute*** function to execute the instructions fed by the ***fetch*** function.

All the function are using pass by reference to link to memory.



* ***Fetch*** function:
* We use the “***PC\_count***” for monitoring the real address of the loaded present memory block whom content fed to the execution stage, because when we allocated the memory we cast the 32bits unsigned integer on the address so whenever we increase the “***PC***” varial by 1 its mean the address get read by 4-bytes or the address increase by 4 so the “***PC\_count***” is for monitoring and follow the debug process of the program.
* The Data we declare here is the variable that will returned by the ***fetch*** function as the instruction for the execution stage which start at the pointer ***Address*** + ***PC***.
* Then the ***PC*** get increase by “1” (+ 4 in memory address).
* The ***cycles*** variable is then decrease, the number of cycles is set by manual for easy debug purposes and easy to control the underdeveloped program, mainly is to control the desired number of instructions to be executed.
* ***Execution*** function:
* As we mentioned above the ***cycles*** control the number of the instructions to be executed:

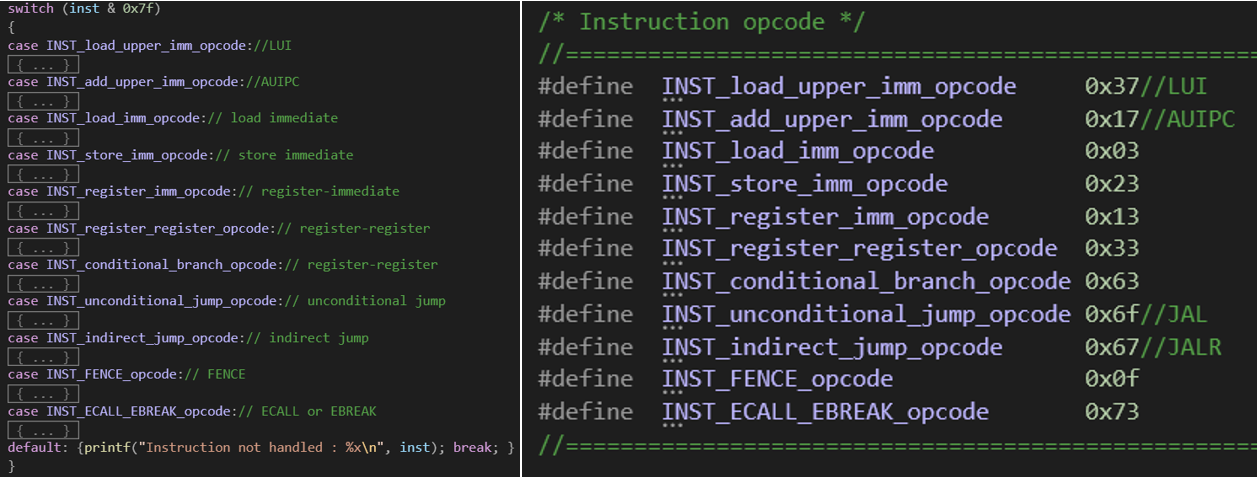


* Variable “***inst***” is the 32bits instruction register that hold the present instruction fed by the ***fetch*** function and later decoded and then executed.
* ***Binary\_converter*** is the function that prompt every single bits in the instruction for monitoring purposes.

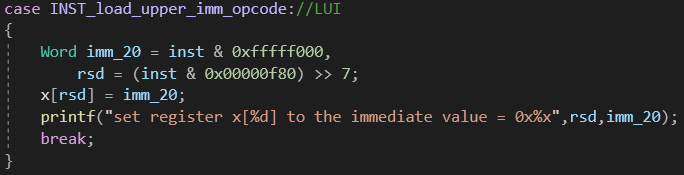
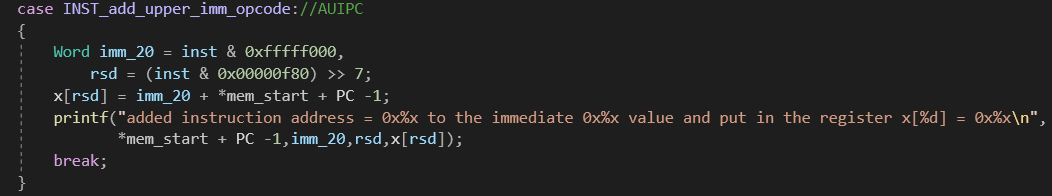
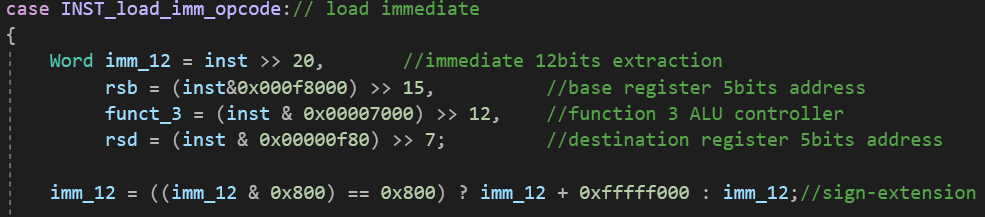


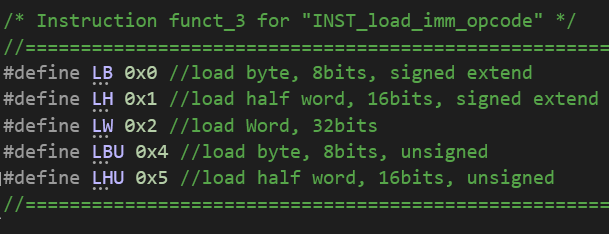
* The decoding stage: the main switch function.

- The **opcode** is extracted by masking the instruction (multiply the 32bits instruction with the number ***0x7f***) with the first 7 least significant bits **opcode** and get selected by the main switch function choosing opcode to execute.

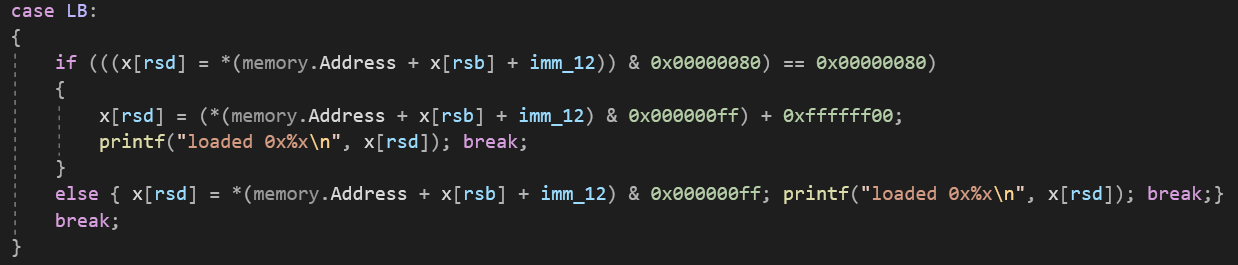


There are 11 **opcode** that covered all 40 instructions of the standard integer base and out of 40 we covered 37 functions except **ECALL**, **EBREAK**, **FENCE**.

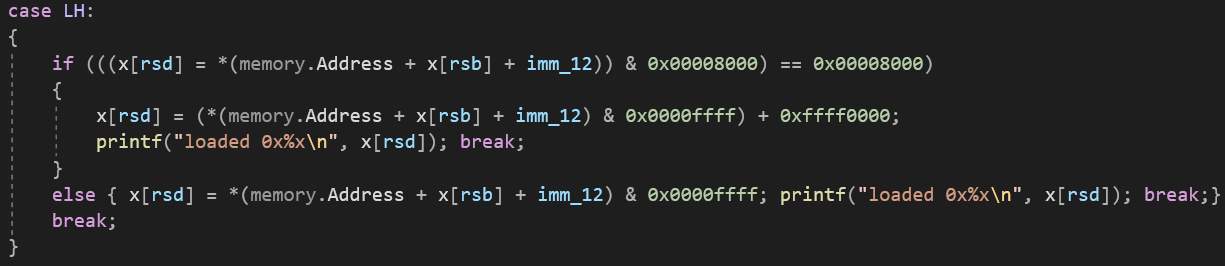
* The “**INST\_load\_upper\_imm\_opcode”** is the opcode for the function LUI (load upper- immediate) which is used to build 32bits constants and it is a U-type instruction format. This function places the 32bits U-immediate value into the destination register ***rsd***, the most significant 20bits are took from the immediate then the lower bits are set to 0s. 
* The “**INST\_add\_upper\_imm\_opcode**” is the opcode for the function **AUIPC** (add upper immediate) is used to build the pc-relative addresses. The instruction is a U-type format, the function form a 32bits U-immediate value that fill the lower bit with zeros then add to the current address of the **AUIPC** instruction then put it in the destination register. 
* The “**INST\_load\_imm\_opcode**” is use for loading data from a memory whom address is comprise by the base register and the immediate value. Below is the instruction decoding.the load immediate opcode has 5 sub functions (**funct\_3**).



* **LB** is for loading the least significant byte out of the content of the acquire memory block then get sign-extend and put in the destination register.



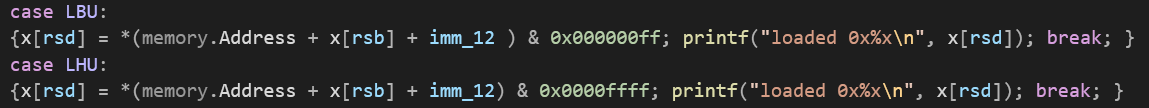
* **LH** is for loading the least significant half word out of the content of the acquire memory block then get sign-extend and put in the destination register.



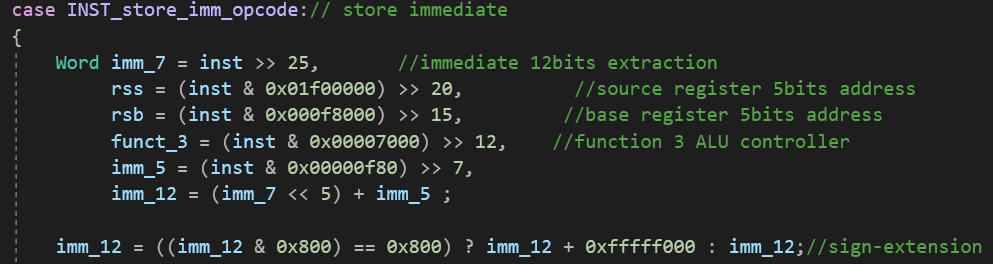
* **LW** is for loading the full word out of the content of the acquire memory block then put in the destination register.



* **LBU** is for loading the unsigned least significant half word out of the content of the acquire memory block then put in the destination register.
* **LHU** is for loading the unsigned least significant half word out of the content of the acquire memory block then put in the destination register.

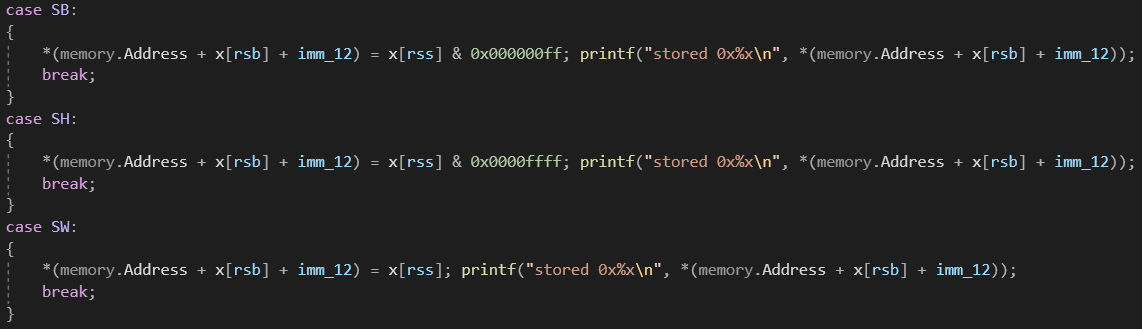


* The “**INST\_store\_imm\_opcode**” is use for storing data from a source register to a memory whom address is comprise by the base register and the immediate value. Below is the instruction decoding.

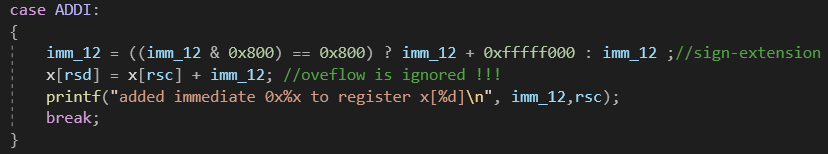


the load immediate opcode has 3 sub functions (**funct\_3**).

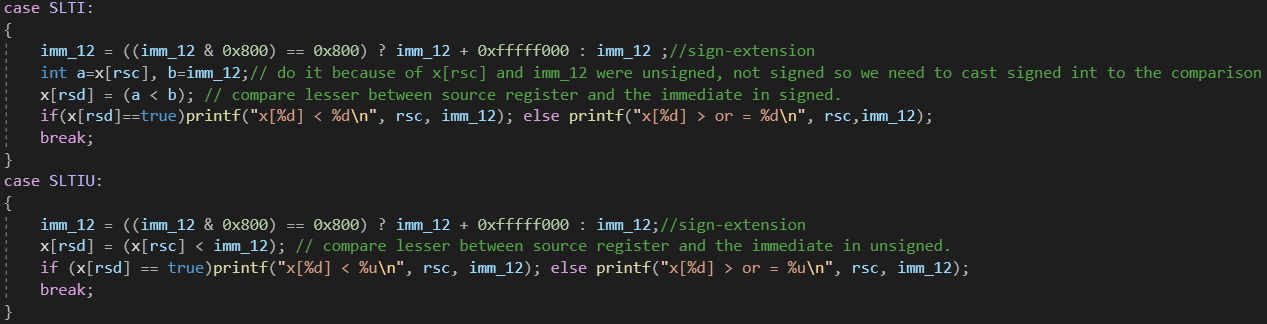
* Like the load sub functions, storing functions also have the capability to store the value in the source register in least significant byte and half word, word. But the value does not sign-extending.



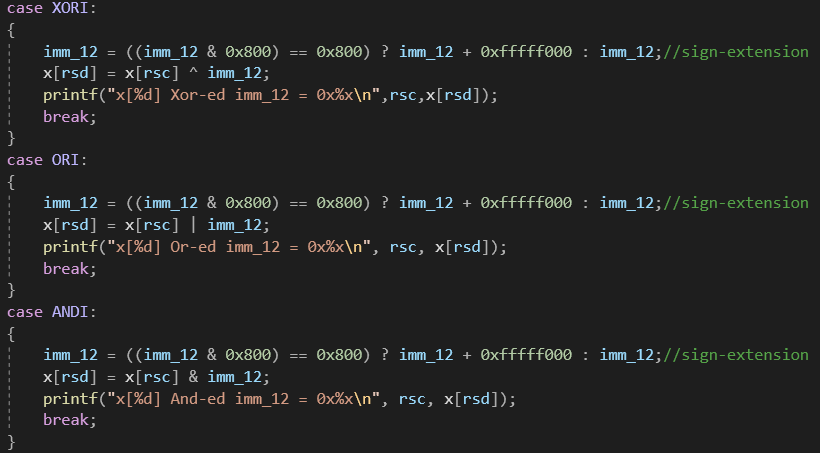
* The “**INST\_register\_imm\_opcode**” is the function that provide operations such as addition, compare , logical , shift logical, shift arithmetic (unsigned and signed).
* **ADDI** is for the addition between signed immediate and the register source then put in the destination register.



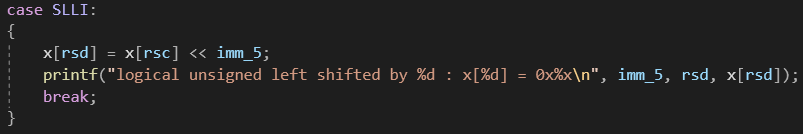
* **STLI** is for compare if the source register is less than the signed integer value in immediate then set the destination register to 1 or else set to 0. **STLIU** compare if the source register is less than the unsigned integer value in immediate then set the destination register to 1 or else set to 0. The immediate is always created with the sign-extending the imm value even though it is then later used as an unsigned integer for the purposes of comparing its magnitude to the unsigned value in source register. Therefore, this instruction provides a method to compare source register to a value in the range of [0->0x7FF] and [ 0xFFFFF800 -> 0xFFFFFFFF].



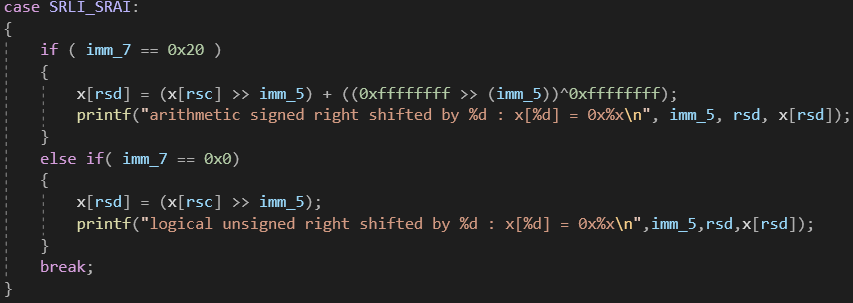
* **XORI, ORI, ANDI** also takes in the sign-extended immediate value and do logical operation between the immediate value and value stored in the source register then put in the destination register.



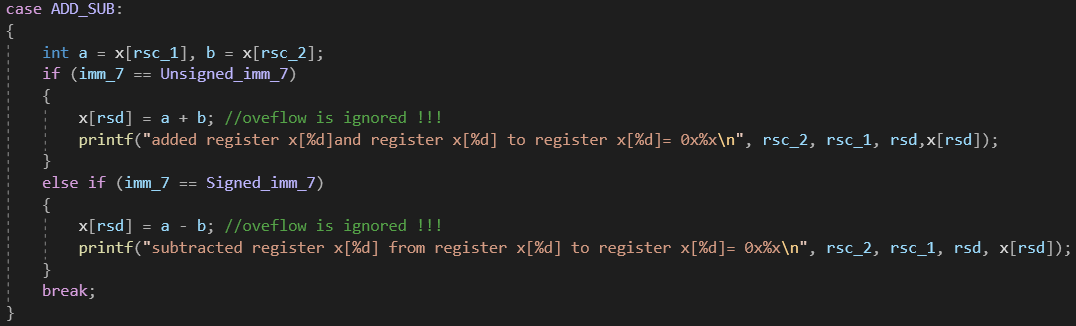
* **SLLI** is the function for logic left shift the value stored in source register with an amount (**shamt**) of 5bits unsigned value that we get from the **imm\_5** by putting 0s to the right match the shift amount.



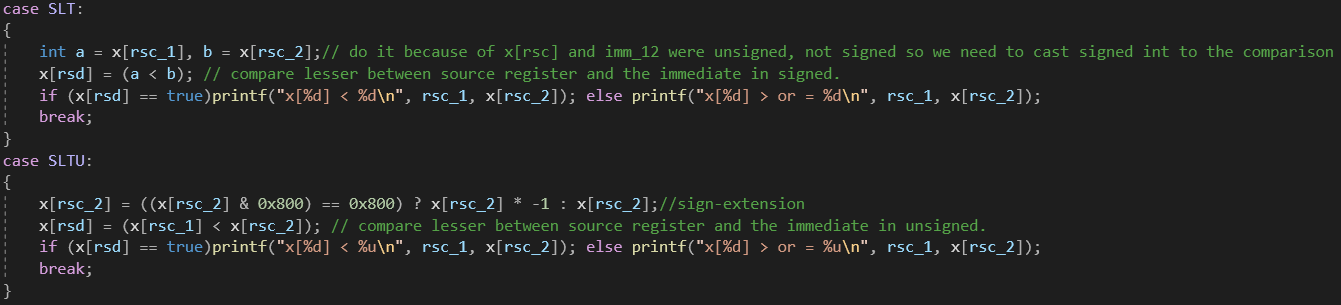
* **SRLI** is the logical right shift, shift the value stored in source register with an amount (**shamt**) of 5bits unsigned value that we get from the **imm\_5** by putting 0s to the left match the shift amount. SRAI is also shifting the source register value to the right but instead of putting unsigned 0s to the left its put signed 1s.



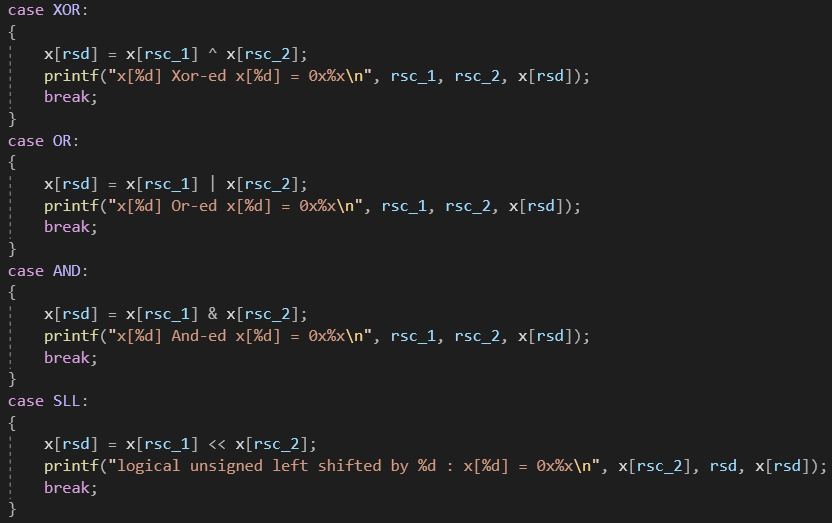
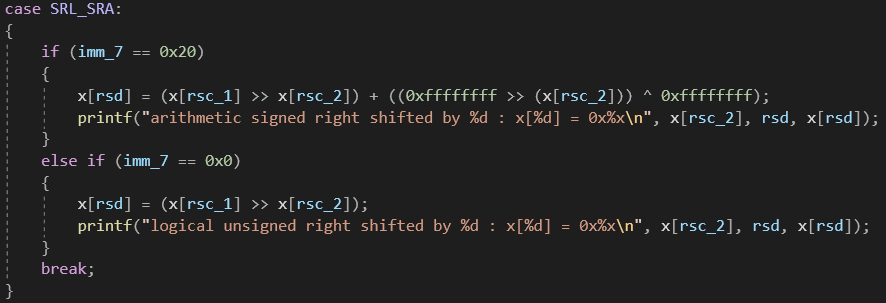
* The “**INST\_register\_register\_opcode**” is the function that provide operations such as addition, subtraction, compare , logical , shift logical, shift arithmetic.
* **ADD\_SUB** is the function that operate arithmetic calculation between source register 1 and source register 2, to determine the addition or subtraction property we use the most significant 7bits to let the program know whether to do addition or subtraction, for subtraction the value of the imm\_7 is 0x20 which is the label **Signed\_imm\_7**, and for addition is 0x0 which is the label **Unsigned\_imm\_7** then the result is put into destination register.



* The **SLT** and **SLTU** have the same functional purpose like the immediate version but the different is the comparing between the source register 1 to source register 2 and then store 1 in the destination register if the statement is true or else put 0.

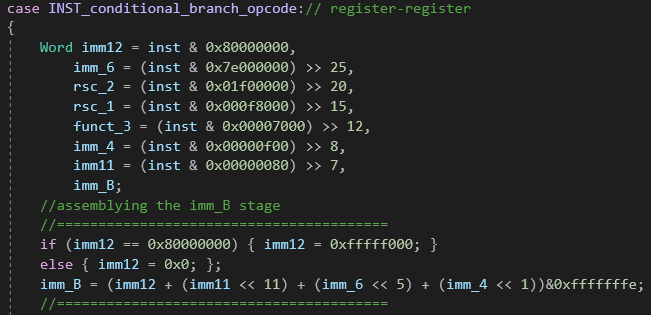


* The logical operation is also the same for this registers operating opcode.

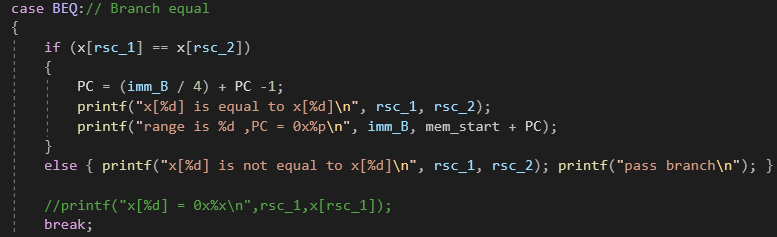
 

* The “**INST\_conditional\_branch\_opcode**” is use for branching with condition that rely on the comparison of the two source registers 1 and 2. The immediate value of this type of function is also special because it is an immediate that has the least significant bit set to 0 because of this function does not allow the immediate jump address amount to be an odd number and because of the jumping address that took the full 4-byte so the second least significant bits also set to 0, so when we jump with condition like this we could only do from 4 to 4\*n (n is the amount of jumping to n full 4 byte address).

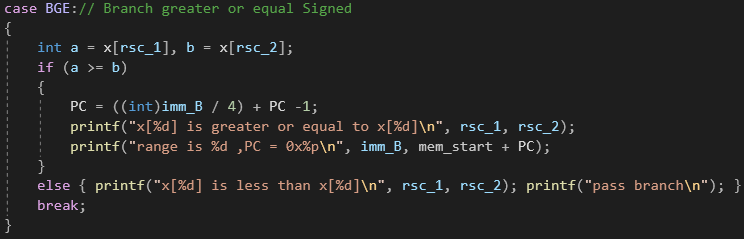
Below is the code of deconstructing the instruction and assembly of immediate value:



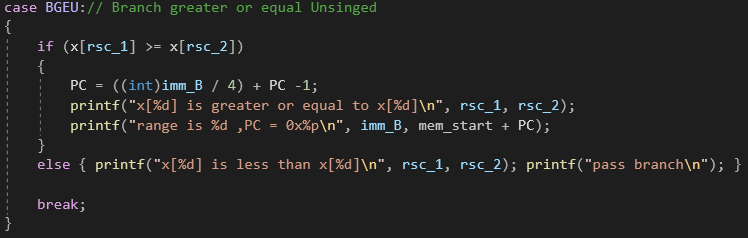
* **BEQ** is the function checking the equality of the source register 1 to the source register 2, if this is true then the “present” **PC** is then added to the immediate value (signed or unsigned), the reason we divided the **imm\_B** value by 4 and add to the **PC** is because increasing or decreasing 1 is a full 4-byte address, and we then subtract the **PC** by 1 is because the **PC** variable is storing the next **PC** that was done by the **fetch** stage.( source registers are both compared in signed value)



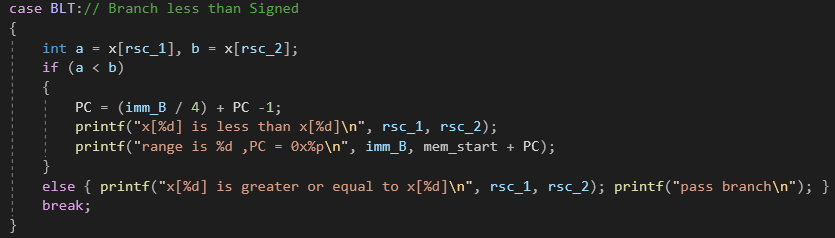
* **BGE** is the function to check if the source register 1 is greater/equal source register 2 and then branch if the statement is true. (source registers are both compared in signed value)



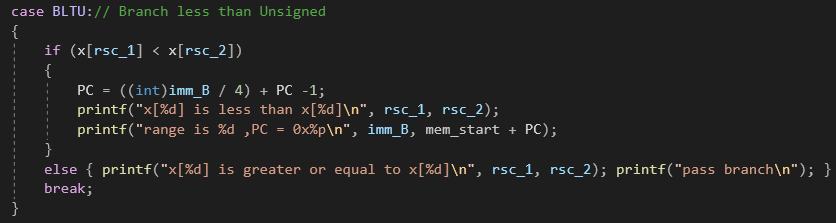
* **BGEU** is the function to check if the source register 1 is greater/equal source register 2 and then branch if the statement is true. (source registers are both compared in unsigned value)



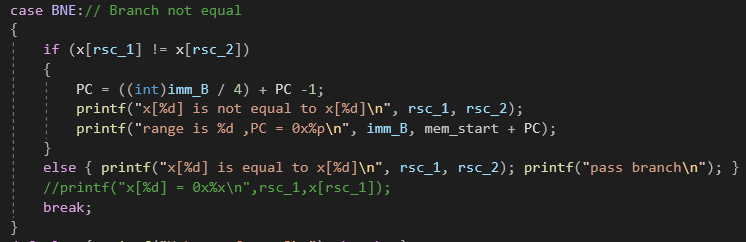
* **BLT** is the function to check if the source register 1 is less than source register 2 and then branch if the statement is true. (source registers are both compared in signed value)



* **BLTU** is the function to check if the source register 1 is less than source register 2 and then branch if the statement is true. (source registers are both compared in unsigned value)

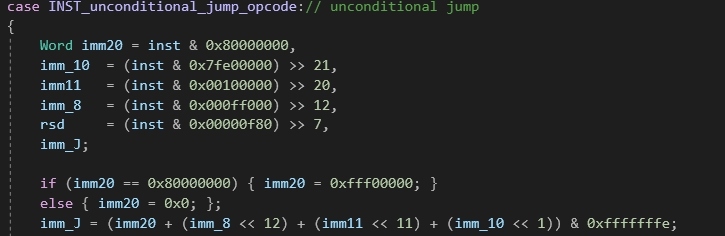


* **BNE** is the function to check if the source register 1 is not equal to the source register 2 and then branch if the statement is true. (source registers are both compared in signed value)

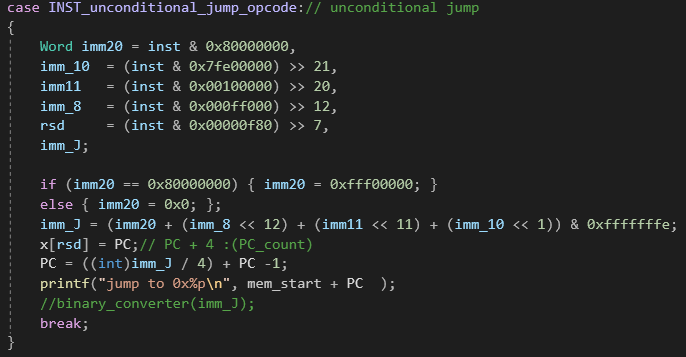


* The “**INST\_unconditional\_jump\_opcode**” is to jump unconditionally to another address the characteristic of this function is similar to branch functions . But the different is it jump without comparing or checking any statement and also retain the (**PC**+4) address in a destination register for jumping purposes.

Below is the decoding the instruction and assembly of the immediate value:



* **JAL** Set register rd to the address of the next instruction that would otherwise be executed (the 1559 address of the **JAL** instruction + 4) and then jump to the address given by the sum of the pc 1560 register and the immediate value as decoded from the instruction.



* **JALR** Set register rd to the address of the next instruction that would otherwise be executed (the 1656 address of the **JALR** instruction + 4) and then jump to an address given by the sum of the rs1 1657 register and the immediate value as decoded from the instruction.
* For the **ECALL, EBREAK, FENCE** we have not manage to implement these function into the program successfully.
  1. **Testing an example and result**

In other to test the program we manual put instruction into the memory by hand (we haven’t achieve create an assembler to translate asm code to machine code).

* ADDI x9,x8,0xfff
* ADDI x9,x8,0xfff
* ADD x7,x7,x8
* BGE x7,x6,0x4
* AUIPC x30,0x0

Initial condition: **x6** = -1; **x7** = 3; **x8** = 1;

int main()

{

//create the cpu core and RAM

CPU cpu; MEM mem;

//reset the memory for initial run

cpu.reset(mem);

//set up memory and register to run some example

cpu.x[6] = -1;

cpu.x[7] = 0x3;

cpu.x[8] = 0x1;

\*(mem.Address + start\_address ) = Reg\_imm(0xfff, x\_8, ADDI, x\_9, INST\_register\_imm\_opcode);

\*(mem.Address + start\_address + 1) = Reg\_imm(0xfff, x\_8, ADDI, x\_9, INST\_register\_imm\_opcode);

\*(mem.Address + start\_address + 2) = Reg\_reg(Signed\_imm\_7, x\_8, x\_7, ADD\_SUB, x\_7, INST\_register\_register\_opcode);

\*(mem.Address + start\_address + 3) = Branches(1, 0x3f, x\_6, x\_7, BGE, 0xe, 1, INST\_conditional\_branch\_opcode);

\*(mem.Address + start\_address + 4) = AUIPC(0x0,x\_30,INST\_add\_upper\_imm\_opcode);

/\*executation of the cpu\*/

//==================

cpu.execute(13, mem);

//==================

//promting register

for (int i = 0; i < 32; i++)

{

if (i <= 9) {

printf("x[%d ] = 0x%x | %d | ", i, cpu.x[i], cpu.x[i]); //}

binary\_converter(cpu.x[i]);

}

else {

printf("x[%d] = 0x%x | %d | ", i, cpu.x[i], cpu.x[i]); //}

binary\_converter(cpu.x[i]);

}

}

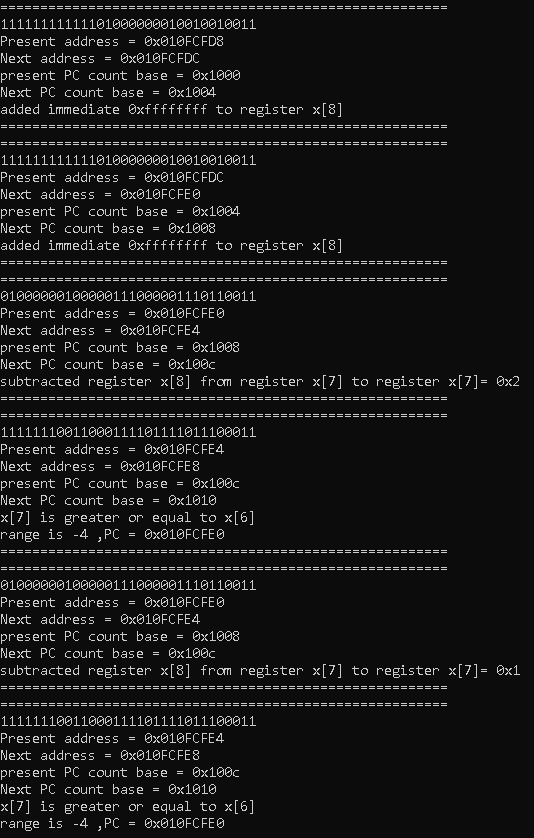
//showing the stop address

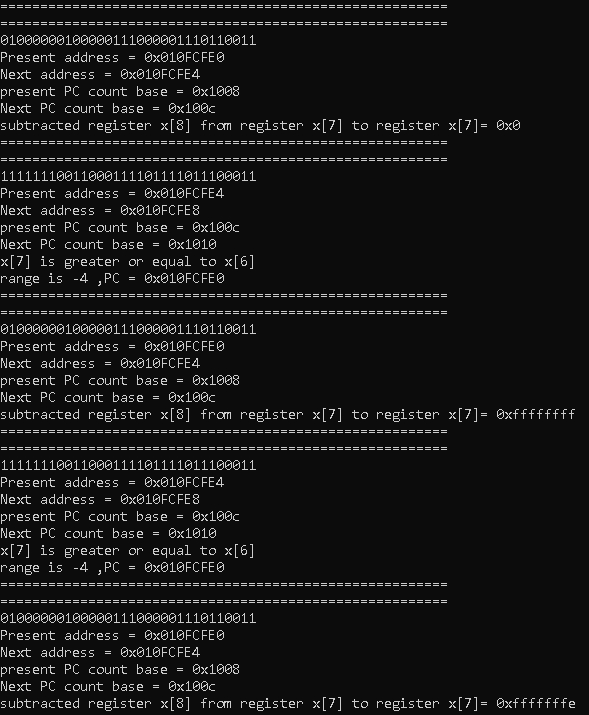
printf("\nPC = 0x%p\n",mem.Address + cpu.PC);

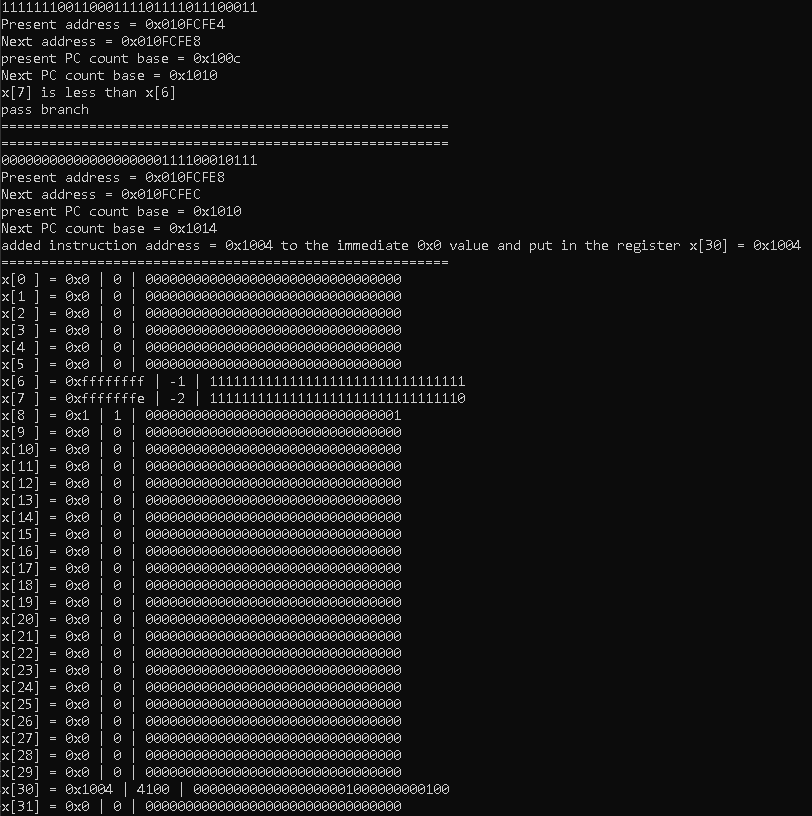
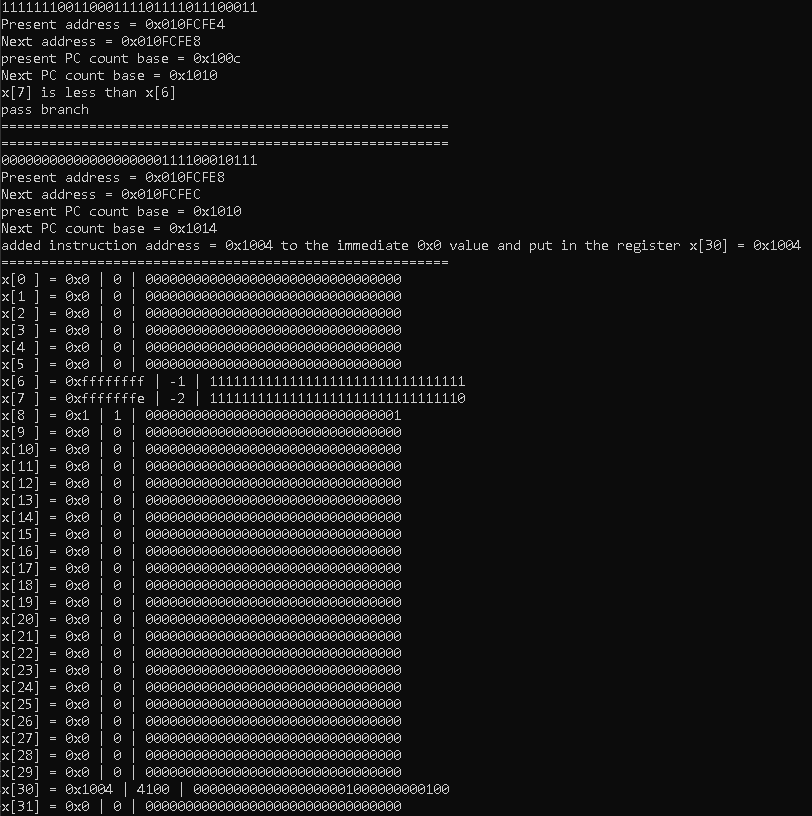
return 0;

}

We then run the program and get the result below.



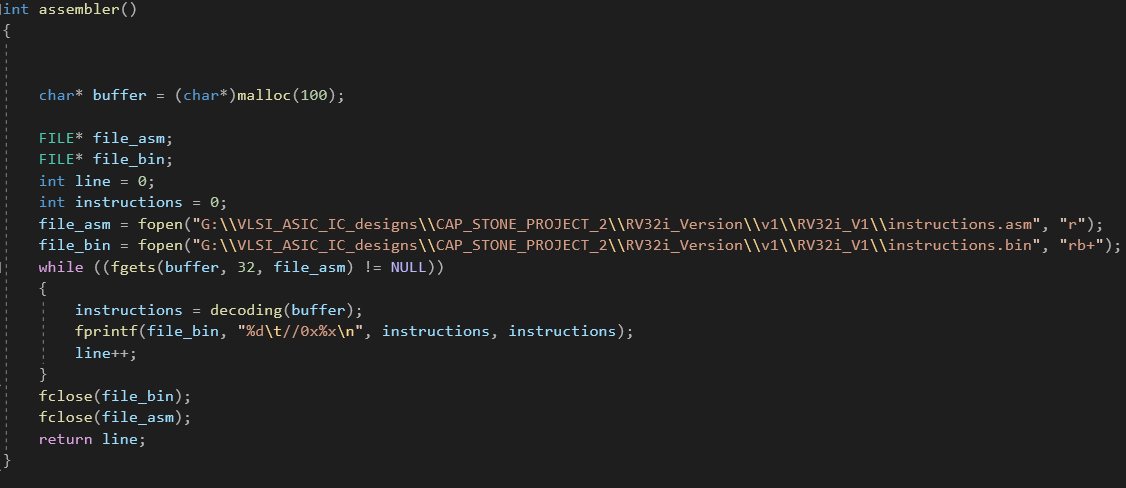




***Result:* x7 = -2; x6 = -1; x7 < x6 (-2 < -1)**

Addition to the project is the assembler from field code to machine code which is turning field code (assembly code) to binary code for the program to process the instruction in binary code not strings.

We start from making file pointer for the “.asm” file and a file pointer for “.bin”



The buffer is memory allocated for reading strings from “.asm” file code lines. The instruction is to store the binary instruction not the field code in strings. The “fgets()” function read every single line in the file at each interval of the while loop then fetch the string to the decoding() function.

The decoding function is destine to turn string coded operations to binary instructions.



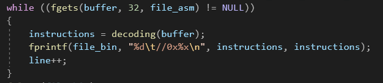
An example of decoding the field code:



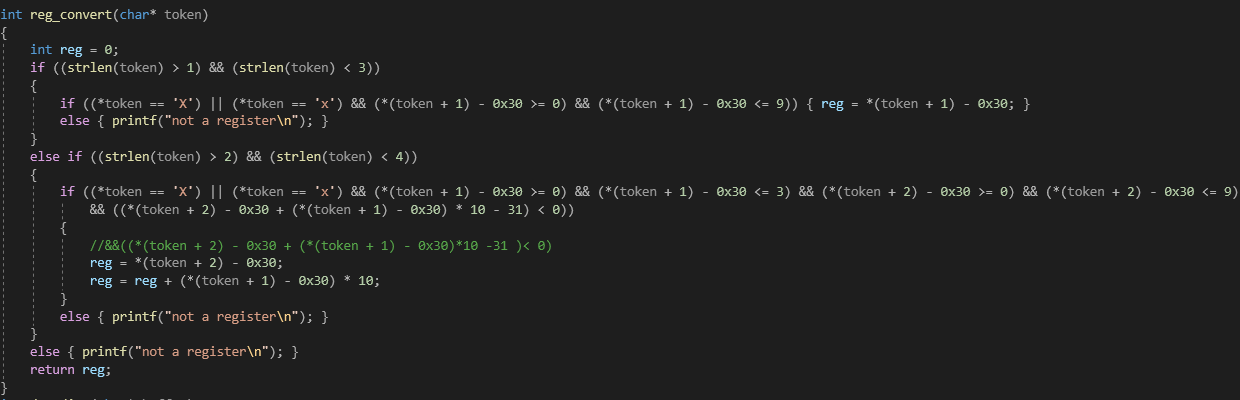
The for loop at the beginning is to let the field code to care none for the capitalize field code or not it will still accept the coded field and turn the input into indexes to be execute by the field code encode into binary.

The instruction decoding is build by extracting “token” from the line read by the buffer which read from “.asm” file. Because of the sequential-ness in the reading of the token by the function “strtok()” we need to patch the instruction piece by piece because the strtok would increment to another field code from every time we call it.

Then when the instruction get decoded and assembled it then returned to the function decoding() then pass the the instruction variable in the assembler() function.

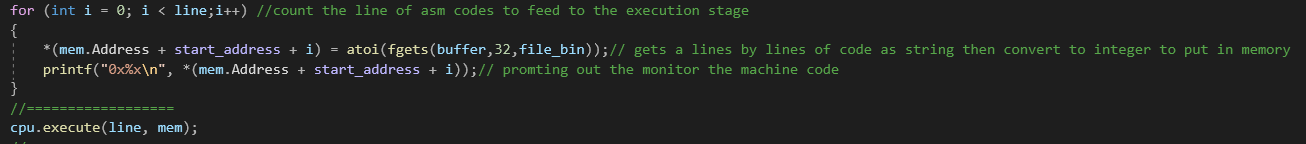


The reg\_convert is to convert “x1” or any register coded field to numbers to be embedded into the instruction.

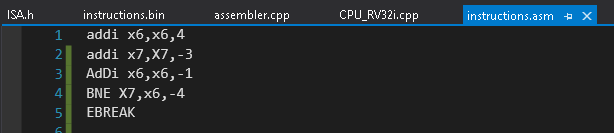


And this function also care none about capitalized character or not it will still accept the register field if wrote in the appropriate format.

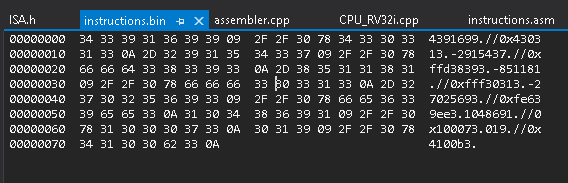
And instead of fabricate function and put instruction into the memory manually, we read the instruction from the .bin file then put it into the memory and let the program do all the execution to the code.



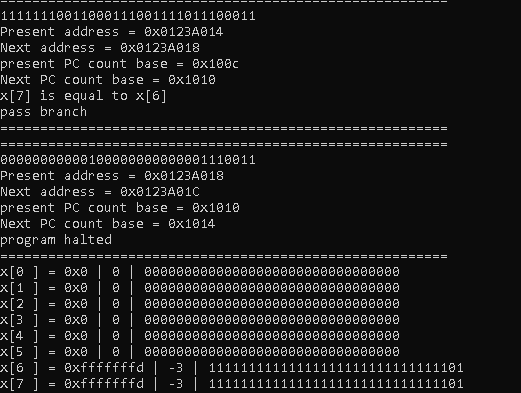
For example, below is the asm code (in .asm file) for an example of setting up two register with different value and try get the second number equal to the first.



Then we ran through the program we got this in this file .bin



And the result of the program:



1. **CONCLUSION.**

At the end, our emulation of the rv32i meet all the required criteria except the 3 function that are **ECALL**, **EBREAK**, **FENCE** is in the process of debugging. The program can perform **37** out of **40** instructions of 6-types of instruction. In the process of building the emulation, we all experience hardship from knowledge to communicating since covid-19 still on the loose, but we managed to turn in the project with good result and spirit of eager to learn more, work together more efficient.

Through the project we could learn a lot from implementing the RISCV architecture with C/C++ programming and we found out that System-C might be our next aim if we ever continue to build RISC-V with another extension for fast ISA implementation purposes. We also figure and be more clearer in the understanding of building data path for a cpu, and in the near future we could use this program to aid our work in doing research in Multicore risc-v cpu.

1. **REFERENCES:**
2. CS Division, EECS Department, University of California, Berkeley, August 24, 2021, “*The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*”.
3. John Winans, June 29, 2021*, ” RISC-V 2 Assembly Language Programming”*
4. Steven Ho,*” RISC-V CPU Datapath, Control Intro”*, <https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture11.pdf?fbclid=IwAR3308_-BuzA7rz2_QQQwf1aiF1IBweUWdUzWobR9DPFypsnL6Xh2mROLA4>
5. Lê Quang Hưng, “RENAS MCU A Microcontroller using RISC-V ISA, AMBA Bus and SPI peripheral”.
6. Semico Research & Consulting Group, RISC-V Market Analysis: The New Kid on the Block ,September. 22, 2021,<https://semico.com/content/risc-v-cores-cagr-approach-160-2025-says-semico-research>
7. Do Ngoc Quynh, Nguyen Quang Hung, 2020*,“VANGUARD (VG) – The First Open Source RISC-V SoC Project in Vietnam”*, <https://riscv.or.jp/wp-content/uploads/Vanguard-The-first-opensource-RISC-V-SoC-Project-in-VietNam_draft.pdf>
8. Simon Rokicki, Davide Pala, Joseph Paturel, Olivier Sentieys, Nov 2019*,” What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications”*, https://hal.archives-ouvertes.fr/hal-02303453/document?fbclid=IwAR12tRln7NbFMCS2kuGTh4jlLvzkKp8gZZcXEvCCVvlxqOgyC8Gbg6yrJj8