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

In this research, we focus on RISC V, an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles which was originally designed to support education and computer architecture research but now become an open-source 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.

I. INTRODUCTION

1. 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 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.

2. 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++.

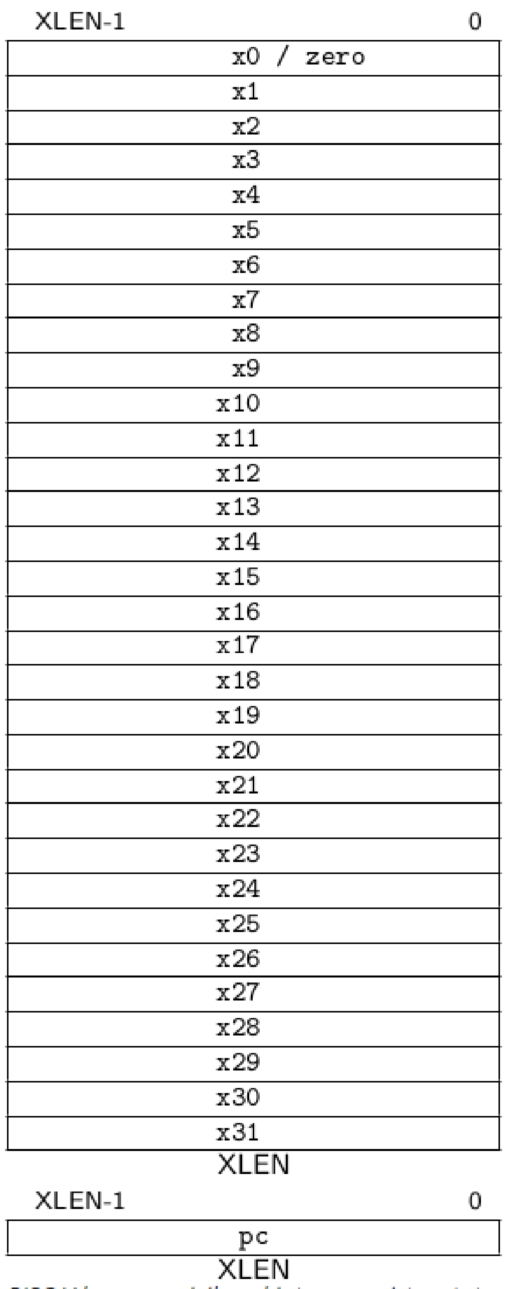
3. Member's work:

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	

II. THEORY

1. Programmers' Model for Base Integer ISA



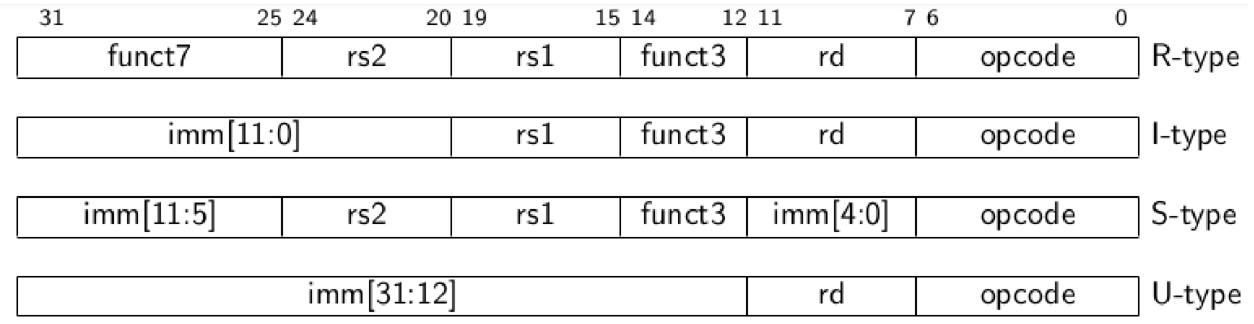
RISC-V base unprivileged integer register state.

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.

2. Base Instruction Format:

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure below.

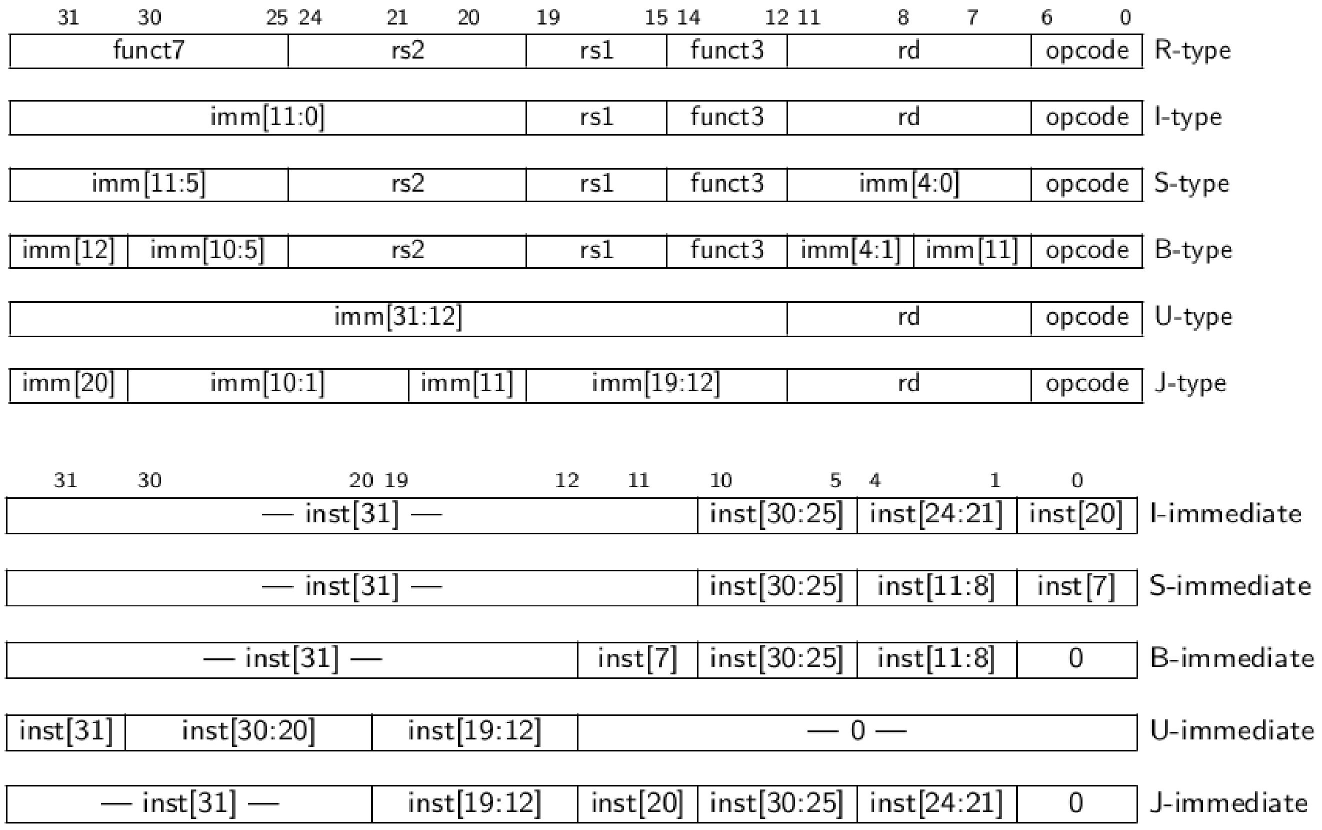

31 25 24 20 19 15 14 12 11 7 6 0
funct7 rs2 rs1 funct3 rd opcode R-type
imm[11:0] rs1 funct3 rd opcode I-type
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
imm[31:12] rd opcode U-type

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.

3. 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.

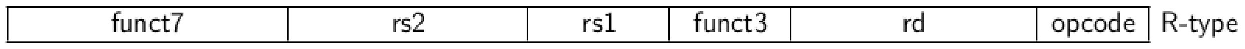

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type
31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

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.

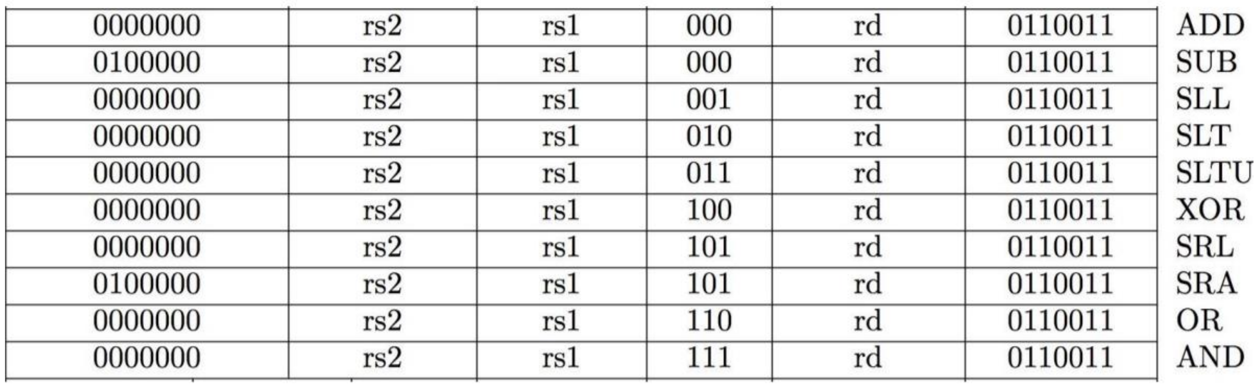
3.1. The Register to register arithmetic and logical operation / R-type:

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.



funct7	rs2	rs1	funct3	rd	opcode	R-type

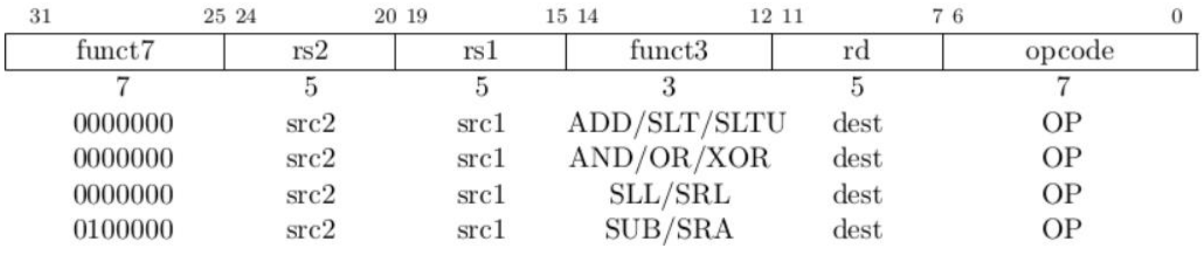
- **opcode** is an operation code or opcode that selects a specific operation
- **rs1** and **rs2** are the first and second source registers
- **rd** is the destination register
- **funct7** and **funct3** is used together with opcode to select an arithmetic instruction



0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

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.



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

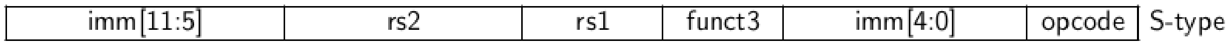
ADD performs the addition of rs1 and rs2. SUB performs the subtraction of rs2 from rs1. Overflows are ignored and the low XLEN bits of results are written to the destination rd. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rsc_1 < rsc_2, 0 otherwise. AND, OR, and 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.

3.2. 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

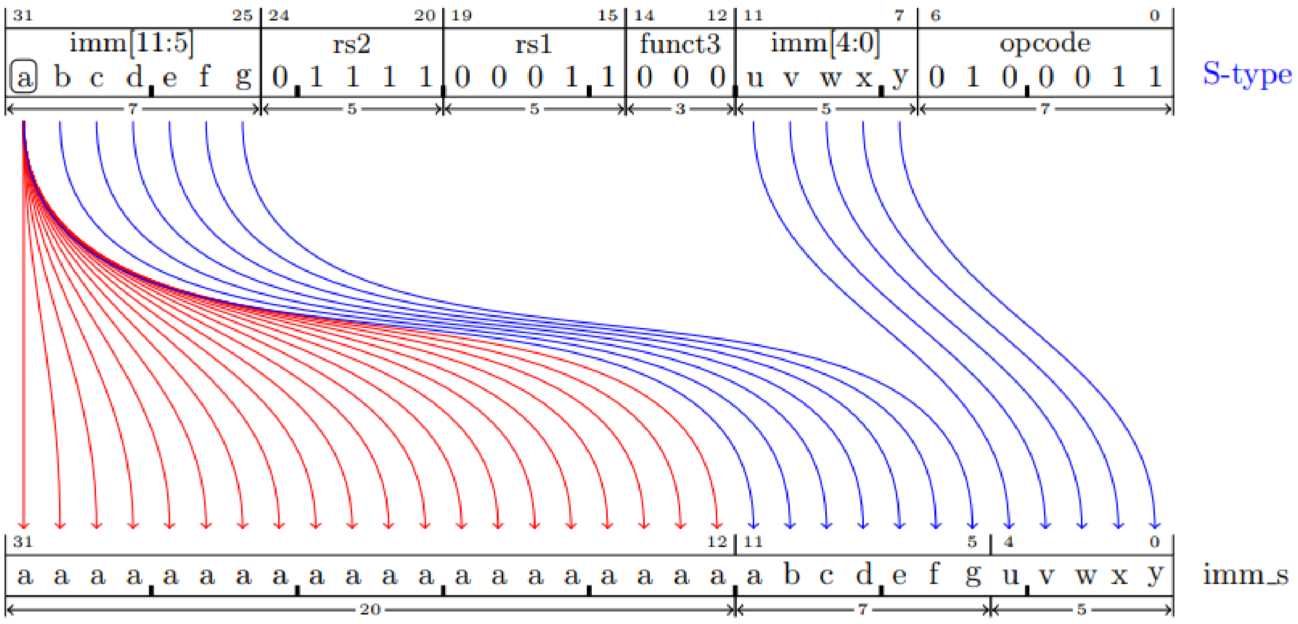


imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

All RV32i Store instruction

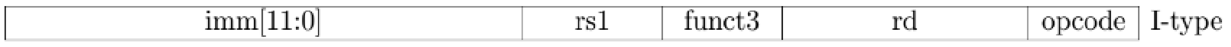
The detail format and also the way to generate immediate of S type are shown the below figures



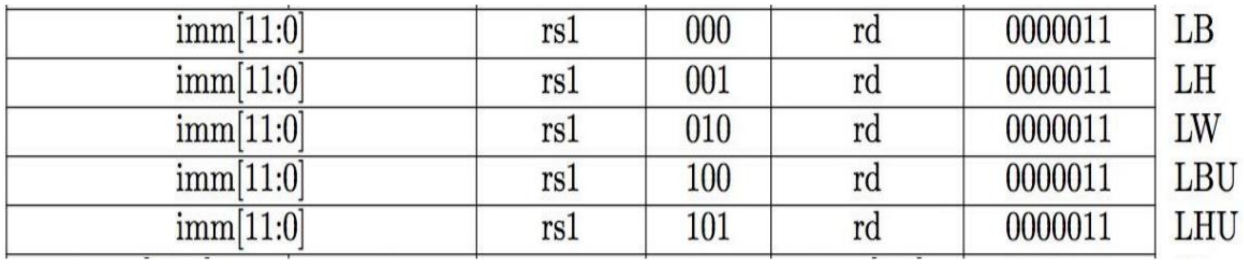
31 25 24 20 19 15 14 12 11 7 6 0
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode
a b c d e f g 0 1 1 1 1 0 0 0 1 1 0 0 0 u v w x y 0 1 0 0 0 1 1
S-type
7 5 5 3 5 7
31 12 11 5 4 0
a a a a a a a a a a a a a a a a a a a a a b c d e f g u v w x y
imm_s
20 7 5

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.



imm[11:0]	rs1	funct3	rd	opcode	I-type

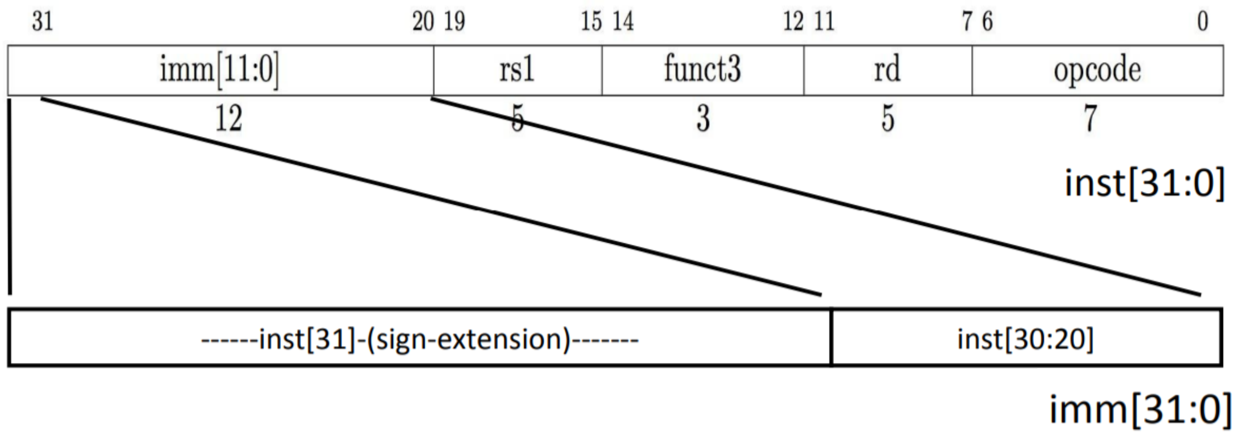


imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

All RV32i Load instruction

To generate I-type immediates:

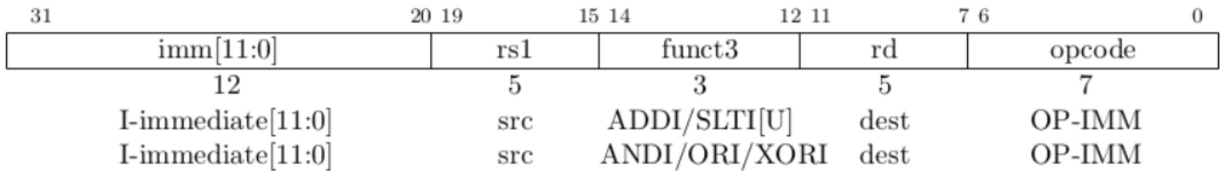
- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])



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.

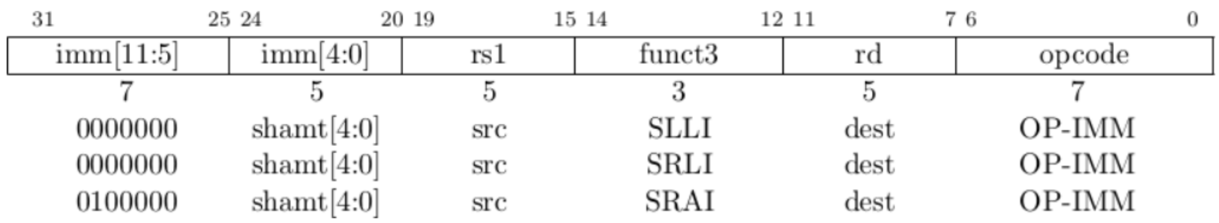
SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 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*).



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

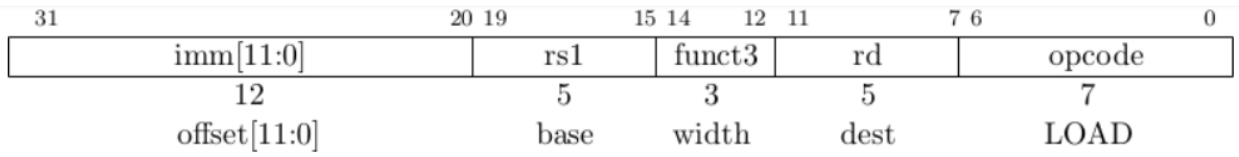
Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits);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



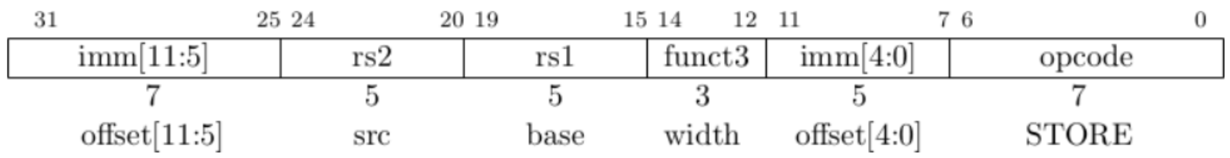
31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

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.



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD



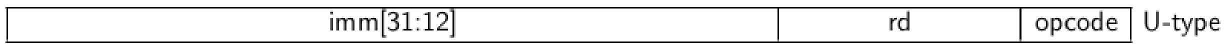
31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

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.

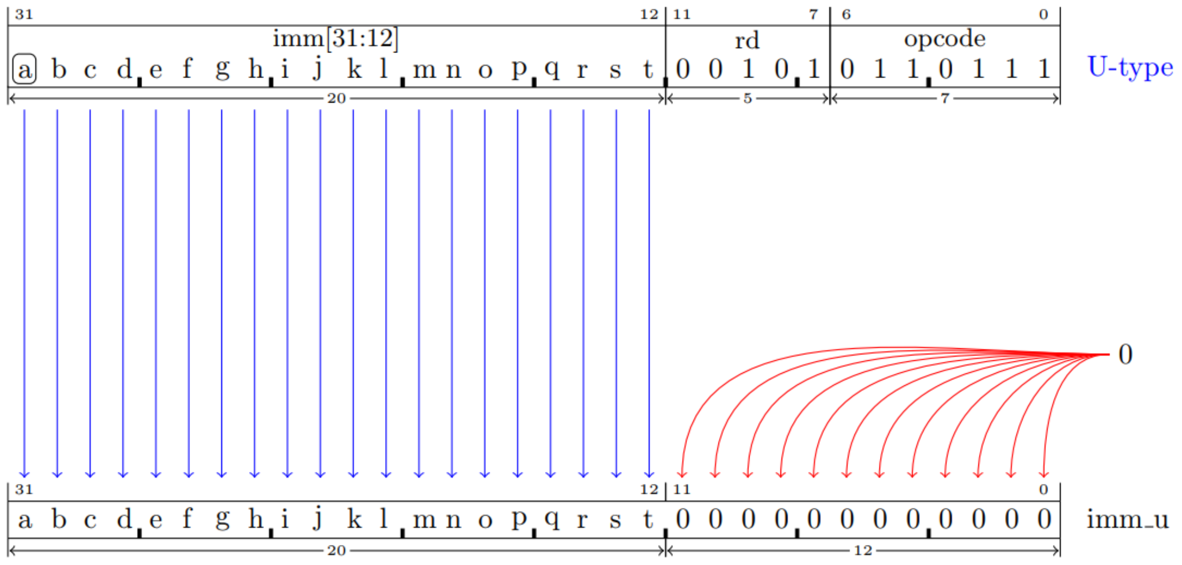
The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The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

3.3. U-type:

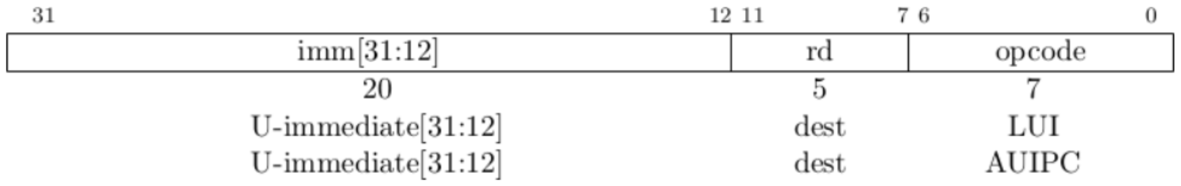
U-Type instruction is used for instructions which have a 20-bit immediate operand and an rd destination register.



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


imm[31:12]
rd
opcode
U-type
a b c d e f g h i j k l m n o p q r s t
0 0 1 0 1
0 1 1 0 1 1 1
0
imm_u
a b c d e f g h i j k l m n o p q r s t 0 0 0 0 0 0 0 0 0 0 0 0

__LUI and AUIPC__



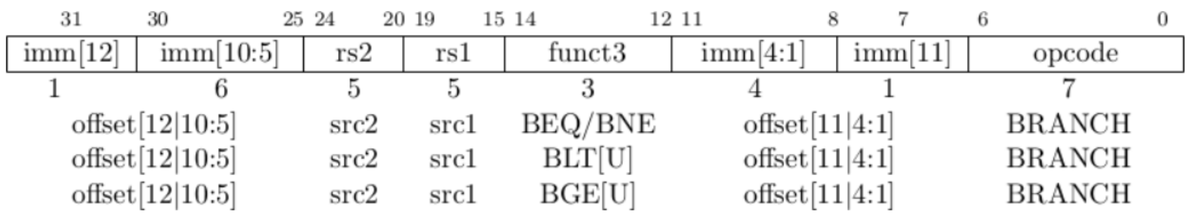
31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

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.

3.4. 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.

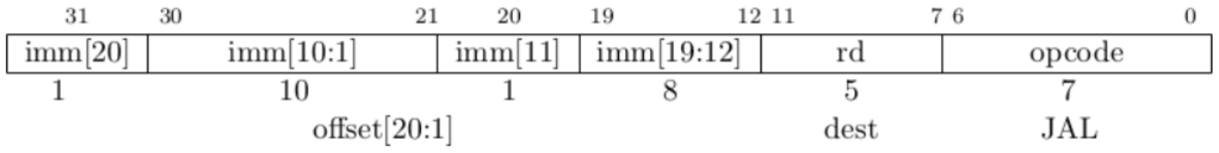


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

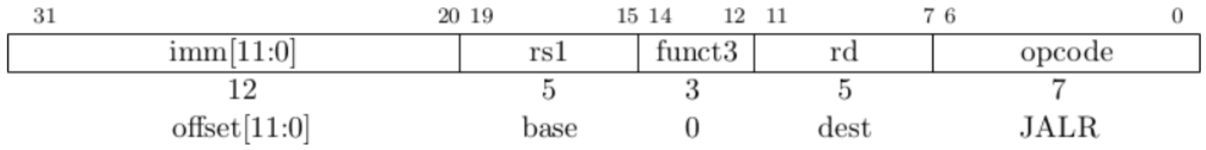
Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or 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.

3.5. 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.



III. EMULATE RV32I STANDARD INTEGER BASE DESIGN:

1. Specification.

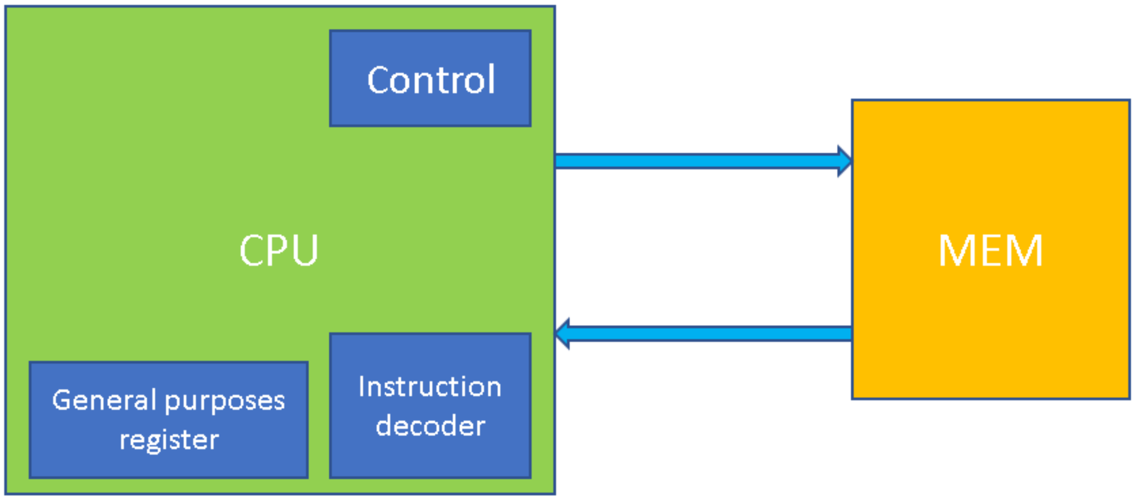
Our RV32I emulator characteristic:

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

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

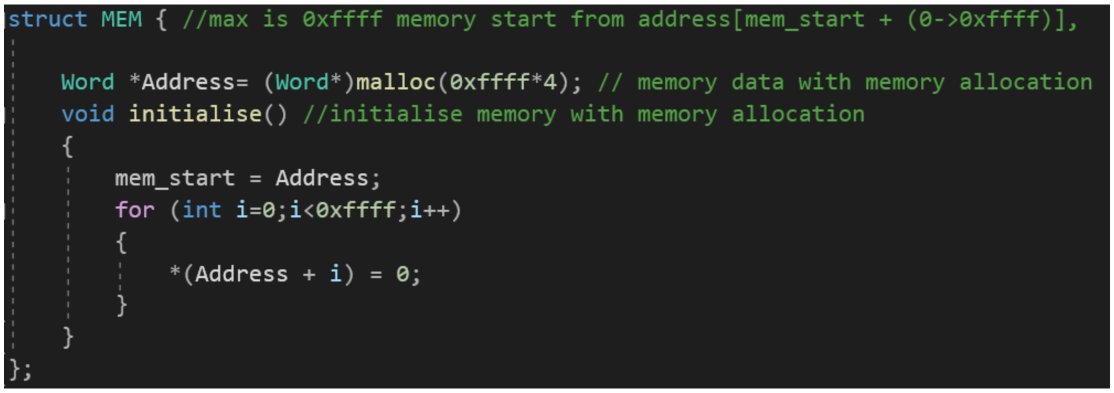
Software used in this project is **Visual studio 2019**.

2. Design of the program.



2.1 Memory.

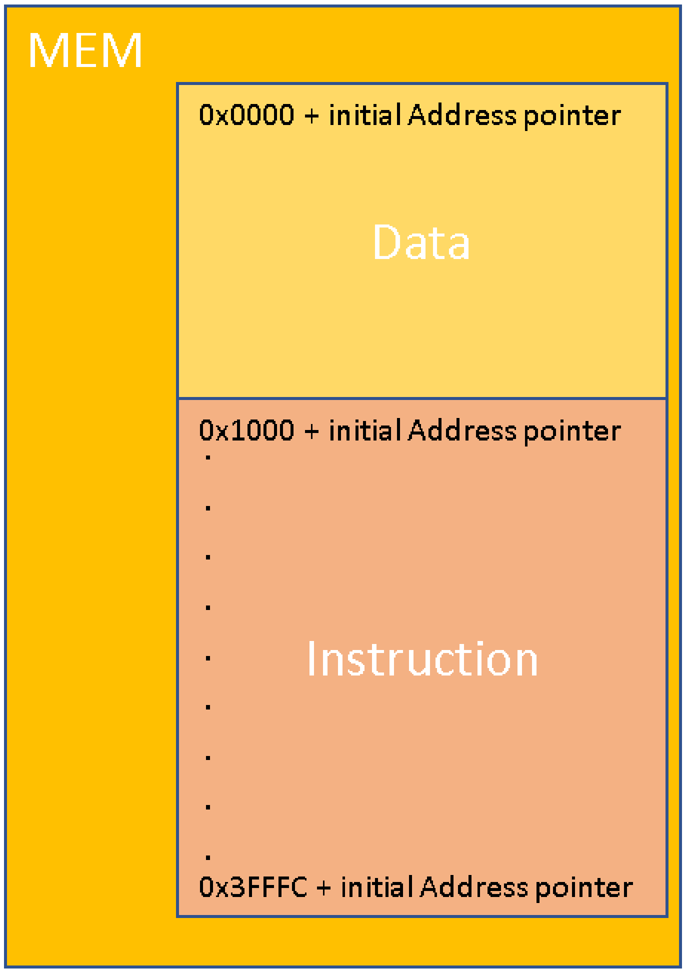
Memory is built by creating a struct named "MEM" with allocated memory



```
struct MEM { //max is 0xffff memory start from address[mem_start + (0->0xffff)],

    Word *Address= (Word*)malloc(0xffff*4); // memory data with memory allocation
    void initialise() //initialise memory with memory allocation
    {
        mem_start = Address;
        for (int i=0;i<0xffff;i++)
        {
            *(Address + i) = 0;
        }
    }
};
```

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.

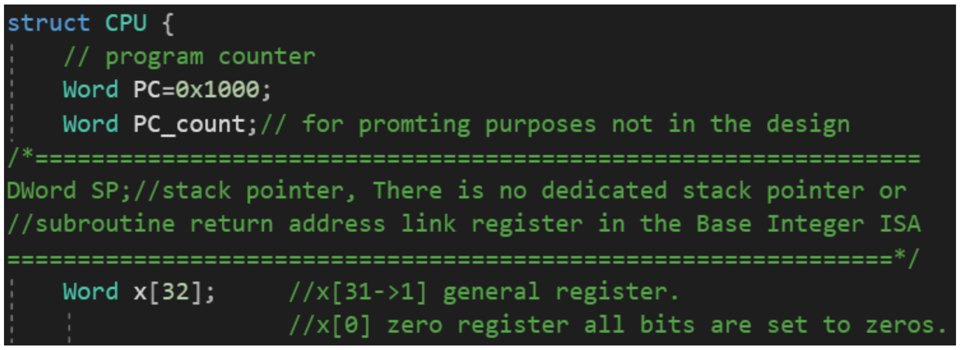
This memory will store the instruction needed for the program to load, also the value that we want to store or load data from it. The instruction will be store at the allocated Address with the address at 0x1000 + initial allocated address pointer.


MEM
0x0000 + initial Address pointer
Data
0x1000 + initial Address pointer
Instruction
0x3FFFC + initial Address pointer

2.2 CPU.

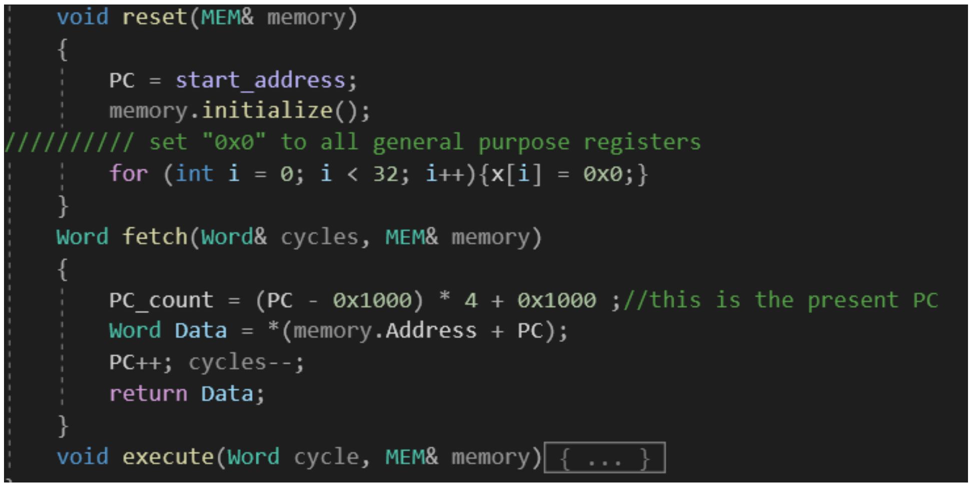
The ***cpu*** design contain:

- ***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.



```
struct CPU {
    // program counter
    Word PC=0x1000;
    Word PC_count;// for promting purposes not in the design
/*=================================================================
DWord SP;//stack pointer, There is no dedicated stack pointer or
//subroutine return address link register in the Base Integer ISA
===============================================================*/
    Word x[32];     //x[31->1] general register.
                    //x[0] zero register all bits are set to zeros.
```

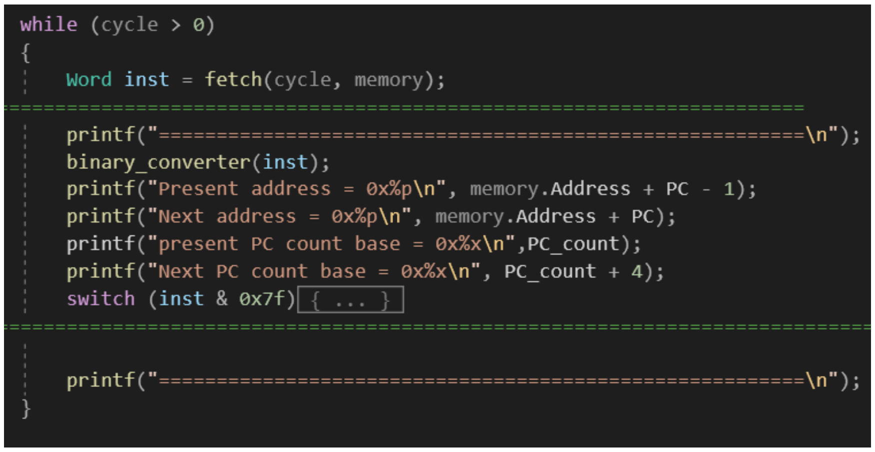
- A "***reset***" function to purposely to assign start-address value which is 0x1000, run the ***initialize*** sequence and a for loop to set all General Purposes Register to 0x0.
- ***Fetch*** function that returned ***instructions*** to load to the ***control*** which is the ***execute*** function 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.



```
    void reset(MEM& memory)
    {
        PC = start_address;
        memory.initialize();
////////// set "0x0" to all general purpose registers
        for (int i = 0; i < 32; i++){x[i] = 0x0;}
    }
    Word fetch(Word& cycles, MEM& memory)
    {
        PC_count = (PC - 0x1000) * 4 + 0x1000 ;//this is the present PC
        Word Data = *(memory.Address + PC);
        PC++; cycles--;
        return Data;
    }
    void execute(Word cycle, MEM& memory){ ... }
```

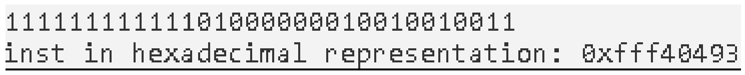
- ***Fetch*** function:
  - We use the "***PC_count***" for monitoring the real address of the loaded present memory block whom content fed to the execution stage, because when we allocated the memory we cast the 32bits unsigned integer on the address so whenever we increase the "***PC***" varial by 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:



```
while (cycle > 0)
{
    Word inst = fetch(cycle, memory);
===========================================================================
    printf("=========================================================\n");
    binary_converter(inst);
    printf("Present address = 0x%p\n", memory.Address + PC - 1);
    printf("Next address = 0x%p\n", memory.Address + PC);
    printf("present PC count base = 0x%x\n",PC_count);
    printf("Next PC count base = 0x%x\n", PC_count + 4);
    switch (inst & 0x7f){ ... }
===========================================================================

    printf("=========================================================\n");
}
```

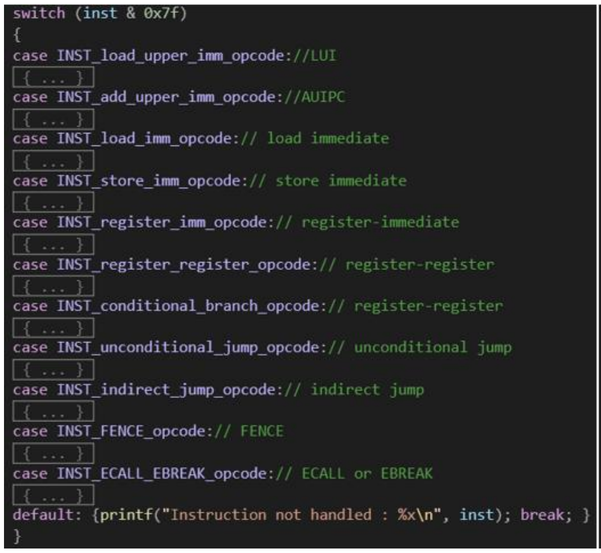
- Variable "***inst***" is the 32bits instruction register that hold the present instruction fed by the ***fetch*** function and later decoded and then executed.
- ***Binary_converter*** is the function that prompt every single bits in the instruction for monitoring purposes.



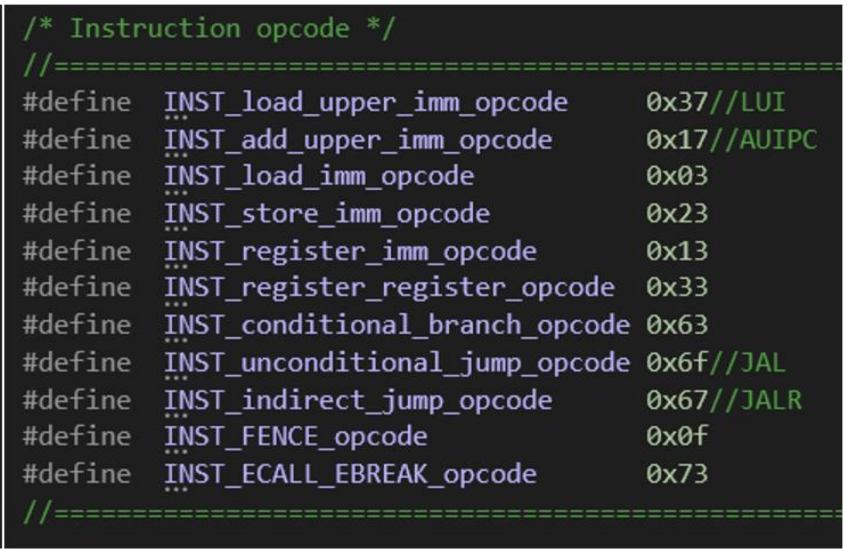
```
11111111111101000000010010010011
inst in hexadecimal representation: 0xfff40493
```

- 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.



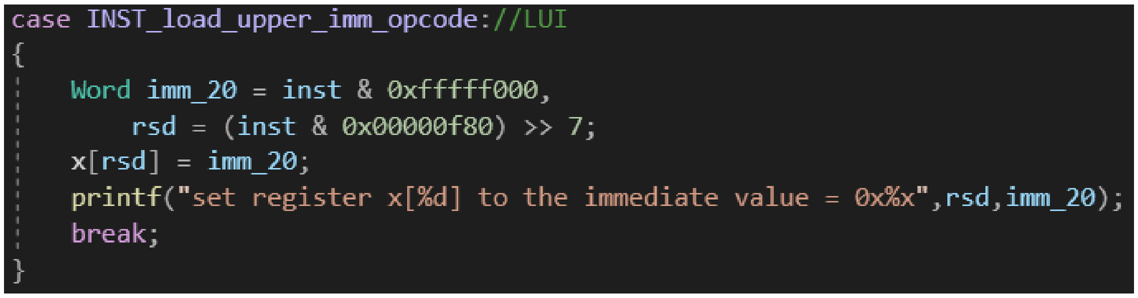
```
switch (inst & 0x7f)
{
case INST_load_upper_imm_opcode://LUI
{ ... }
case INST_add_upper_imm_opcode://AUIPC
{ ... }
case INST_load_imm_opcode:// load immediate
{ ... }
case INST_store_imm_opcode:// store immediate
{ ... }
case INST_register_imm_opcode:// register-immediate
{ ... }
case INST_register_register_opcode:// register-register
{ ... }
case INST_conditional_branch_opcode:// register-register
{ ... }
case INST_unconditional_jump_opcode:// unconditional jump
{ ... }
case INST_indirect_jump_opcode:// indirect jump
{ ... }
case INST_FENCE_opcode:// FENCE
{ ... }
case INST_ECALL_EBREAK_opcode:// ECALL or EBREAK
{ ... }
default: {printf("Instruction not handled : %x\n", inst); break; }
}
```



```
/* Instruction opcode */
//==================================================
#define  INST_load_upper_imm_opcode      0x37//LUI
#define  INST_add_upper_imm_opcode       0x17//AUIPC
#define  INST_load_imm_opcode            0x03
#define  INST_store_imm_opcode           0x23
#define  INST_register_imm_opcode        0x13
#define  INST_register_register_opcode   0x33
#define  INST_conditional_branch_opcode  0x63
#define  INST_unconditional_jump_opcode  0x6f//JAL
#define  INST_indirect_jump_opcode       0x67//JALR
#define  INST_FENCE_opcode               0x0f
#define  INST_ECALL_EBREAK_opcode        0x73
//==================================================
```

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**.

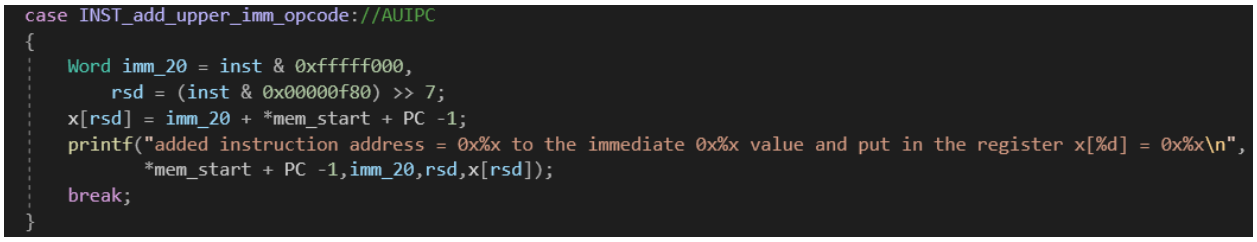
- The "**INST_load_upper_imm_opcode"** is the opcode for the function LUI (load upper- immediate) which is used to build 32bits constants and it is a U-type instruction format. This function places the 32bits U-immediate value into the destination register ***rsd***, the most significant 20bits are took from the immediate then the lower bits are set to 0s.



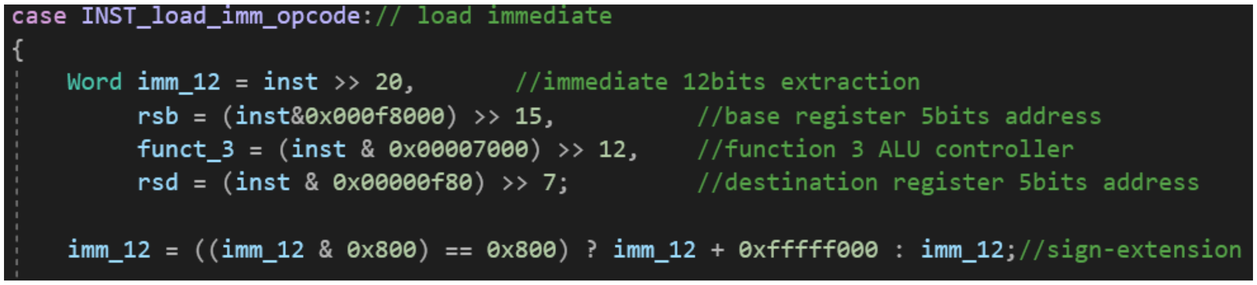
```
case INST_load_upper_imm_opcode://LUI
{
    Word imm_20 = inst & 0xfffff000,
        rsd = (inst & 0x00000f80) >> 7;
    x[rsd] = imm_20;
    printf("set register x[%d] to the immediate value = 0x%x",rsd,imm_20);
    break;
}
```

- 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

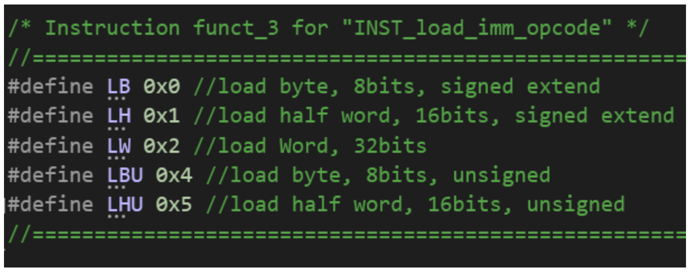
AUIPC instruction then put it in the destination register.


```
case INST_add_upper_imm_opcode://AUIPC
{
    Word imm_20 = inst & 0xfffff000,
        rsd = (inst & 0x00000f80) >> 7;
    x[rsd] = imm_20 + *mem_start + PC -1;
    printf("added instruction address = 0x%x to the immediate 0x%x value and put in the register x[%d] = 0x%x\n",
           *mem_start + PC -1,imm_20,rsd,x[rsd]);
    break;
}
```

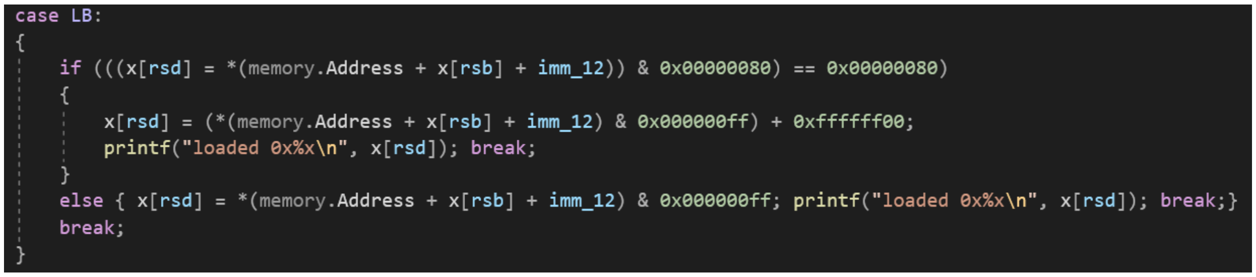
- The "**INST_load_imm_opcode**" is use for loading data from a memory whom address is comprise by the base register and the immediate value. Below is the instruction decoding.


```
case INST_load_imm_opcode:// load immediate
{
    Word imm_12 = inst >> 20,        //immediate 12bits extraction
         rsb = (inst&0x000f8000) >> 15,         //base register 5bits address
         funct_3 = (inst & 0x00007000) >> 12,   //function 3 ALU controller
         rsd = (inst & 0x00000f80) >> 7;        //destination register 5bits address

    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
```

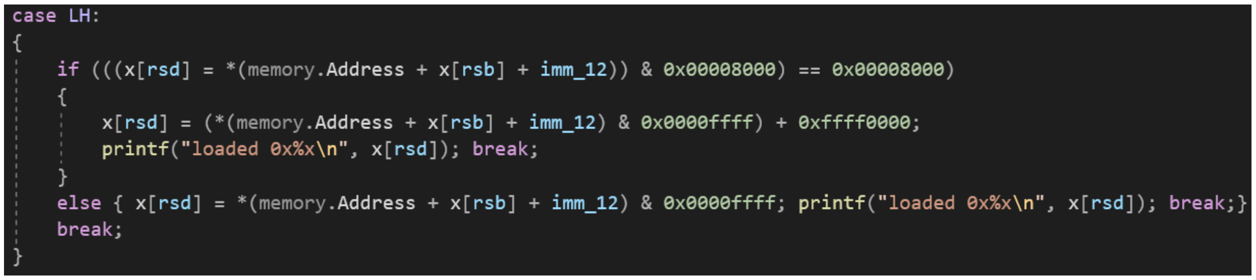
the load immediate opcode has 5 sub functions (**funct_3**).


```
/* Instruction funct_3 for "INST_load_imm_opcode" */
//====================================================
#define LB 0x0 //load byte, 8bits, signed extend
#define LH 0x1 //load half word, 16bits, signed extend
#define LW 0x2 //load Word, 32bits
#define LBU 0x4 //load byte, 8bits, unsigned
#define LHU 0x5 //load half word, 16bits, unsigned
//====================================================
```

- **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.

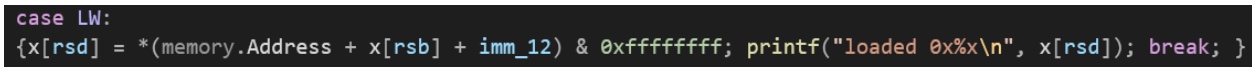

```
case LB:
{
    if (((x[rsd] = *(memory.Address + x[rsb] + imm_12)) & 0x00000080) == 0x00000080)
    {
        x[rsd] = (*(memory.Address + x[rsb] + imm_12) & 0x000000ff) + 0xffffff00;
        printf("loaded 0x%x\n", x[rsd]); break;
    }
    else { x[rsd] = *(memory.Address + x[rsb] + imm_12) & 0x000000ff; printf("loaded 0x%x\n", x[rsd]); break;}
    break;
}
```

- **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.



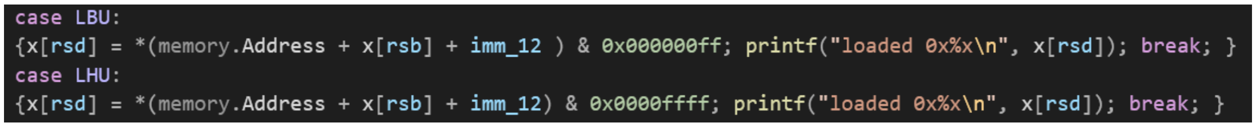
```
case LH:
{
    if (((x[rsd] = *(memory.Address + x[rsb] + imm_12)) & 0x00008000) == 0x00008000)
    {
        x[rsd] = (*(memory.Address + x[rsb] + imm_12) & 0x0000ffff) + 0xffff0000;
        printf("loaded 0x%x\n", x[rsd]); break;
    }
    else { x[rsd] = *(memory.Address + x[rsb] + imm_12) & 0x0000ffff; printf("loaded 0x%x\n", x[rsd]); break;}
    break;
}
```

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



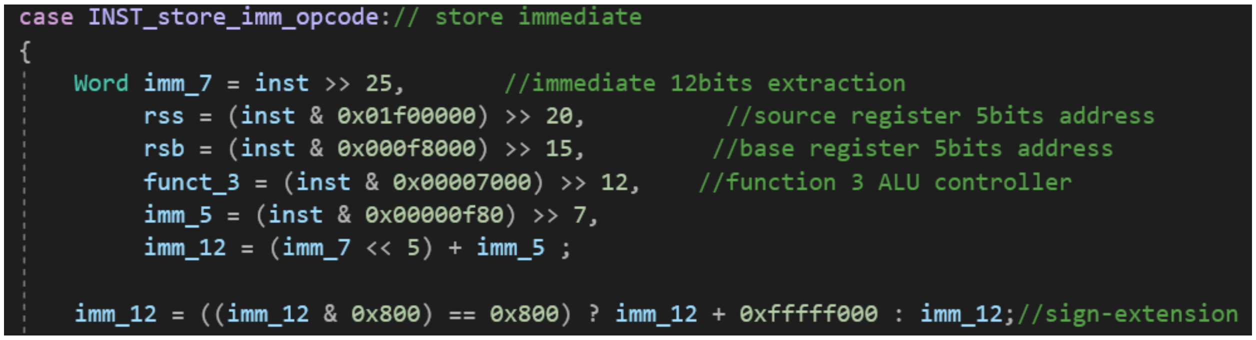
```
case LW:
{x[rsd] = *(memory.Address + x[rsb] + imm_12) & 0xffffffff; printf("loaded 0x%x\n", x[rsd]); break; }
```

- **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.



```
case LBU:
{x[rsd] = *(memory.Address + x[rsb] + imm_12 ) & 0x000000ff; printf("loaded 0x%x\n", x[rsd]); break; }
case LHU:
{x[rsd] = *(memory.Address + x[rsb] + imm_12) & 0x0000ffff; printf("loaded 0x%x\n", x[rsd]); break; }
```

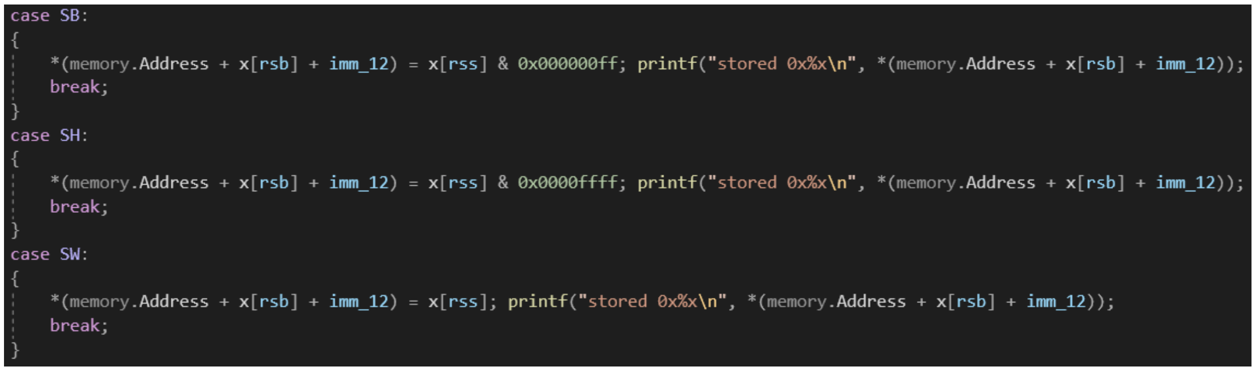
- The “**INST_store_imm_opcode**” is use for storing data from a source register to a memory whom address is comprise by the base register and the immediate value. Below is the instruction decoding.



```
case INST_store_imm_opcode:// store immediate
{
    Word imm_7 = inst >> 25,        //immediate 12bits extraction
         rss = (inst & 0x01f00000) >> 20,          //source register 5bits address
         rsb = (inst & 0x000f8000) >> 15,         //base register 5bits address
         funct_3 = (inst & 0x00007000) >> 12,    //function 3 ALU controller
         imm_5 = (inst & 0x00000f80) >> 7,
         imm_12 = (imm_7 << 5) + imm_5 ;

    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
```

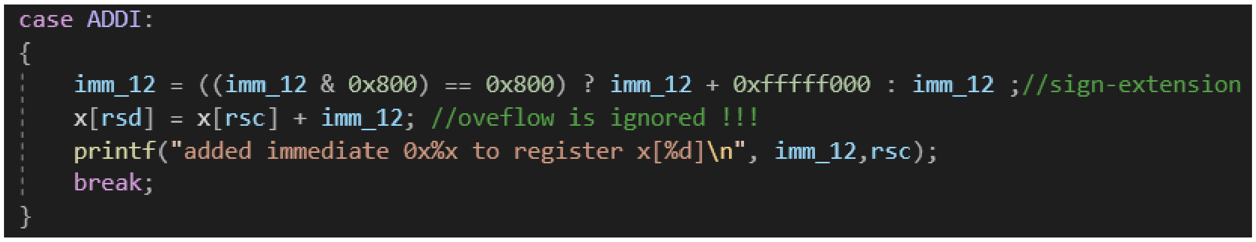
the load immediate opcode has 3 sub functions (**funct_3**).

- 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.



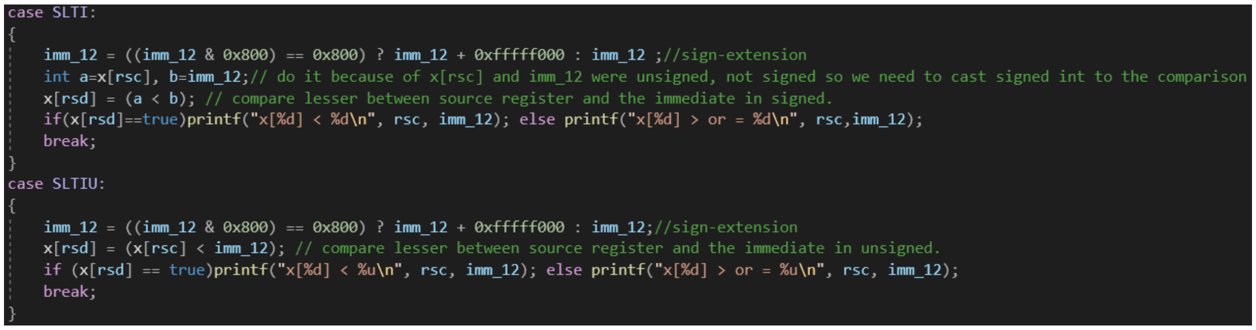
```
case SB:
{
    *(memory.Address + x[rsb] + imm_12) = x[rss] & 0x000000ff; printf("stored 0x%x\n", *(memory.Address + x[rsb] + imm_12));
    break;
}
case SH:
{
    *(memory.Address + x[rsb] + imm_12) = x[rss] & 0x0000ffff; printf("stored 0x%x\n", *(memory.Address + x[rsb] + imm_12));
    break;
}
case SW:
{
    *(memory.Address + x[rsb] + imm_12) = x[rss]; printf("stored 0x%x\n", *(memory.Address + x[rsb] + imm_12));
    break;
}
```

- 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.



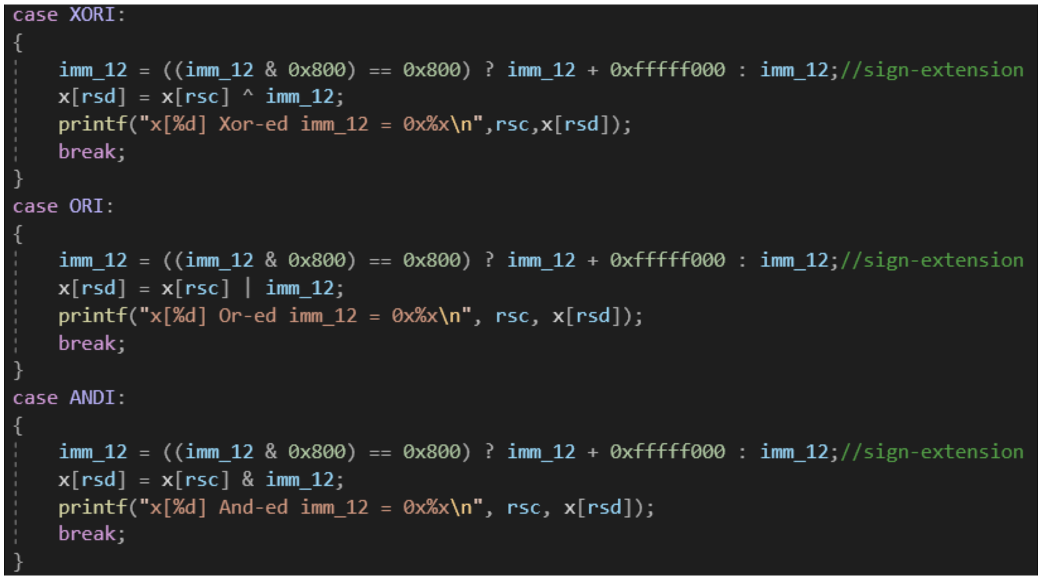
```
case ADDI:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12 ;//sign-extension
    x[rsd] = x[rsc] + imm_12; //oveflow is ignored !!!
    printf("added immediate 0x%x to register x[%d]\n", imm_12,rsc);
    break;
}
```

- **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].



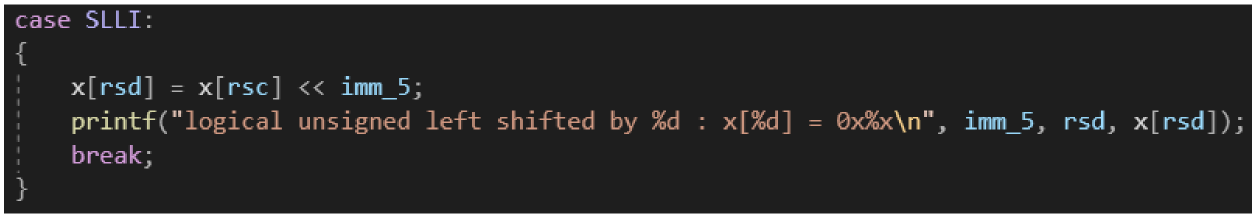
```
case SLTI:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12 ;//sign-extension
    int a=x[rsc], b=imm_12;// do it because of x[rsc] and imm_12 were unsigned, not signed so we need to cast signed int to the comparison
    x[rsd] = (a < b); // compare lesser between source register and the immediate in signed.
    if(x[rsd]==true)printf("x[%d] < %d\n", rsc, imm_12); else printf("x[%d] > or = %d\n", rsc,imm_12);
    break;
}
case SLTIU:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
    x[rsd] = (x[rsc] < imm_12); // compare lesser between source register and the immediate in unsigned.
    if (x[rsd] == true)printf("x[%d] < %u\n", rsc, imm_12); else printf("x[%d] > or = %u\n", rsc, imm_12);
    break;
}
```

- **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.



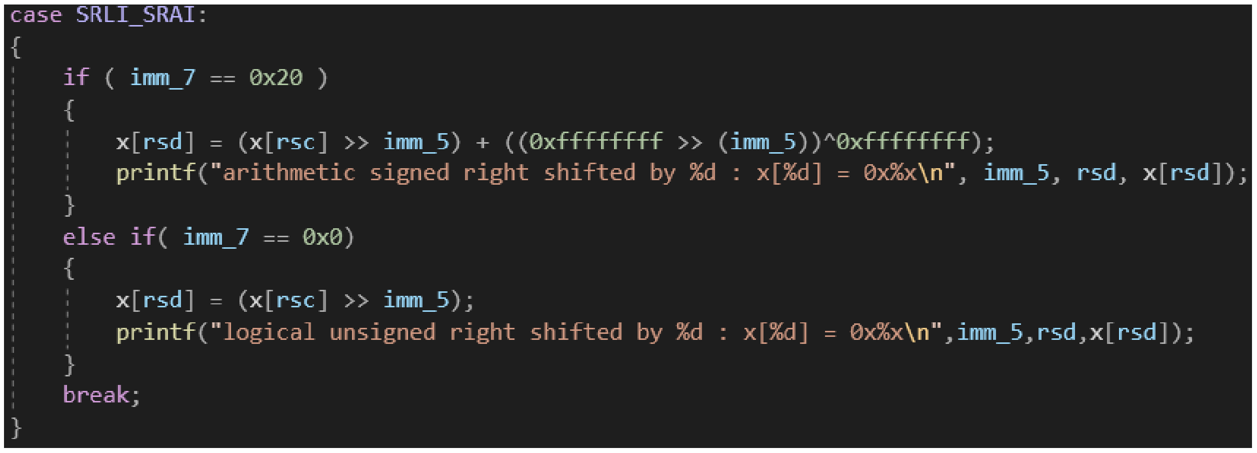
```
case XORI:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
    x[rsd] = x[rsc] ^ imm_12;
    printf("x[%d] Xor-ed imm_12 = 0x%x\n",rsc,x[rsd]);
    break;
}
case ORI:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
    x[rsd] = x[rsc] | imm_12;
    printf("x[%d] Or-ed imm_12 = 0x%x\n", rsc, x[rsd]);
    break;
}
case ANDI:
{
    imm_12 = ((imm_12 & 0x800) == 0x800) ? imm_12 + 0xfffff000 : imm_12;//sign-extension
    x[rsd] = x[rsc] & imm_12;
    printf("x[%d] And-ed imm_12 = 0x%x\n", rsc, x[rsd]);
    break;
}
```

- **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.



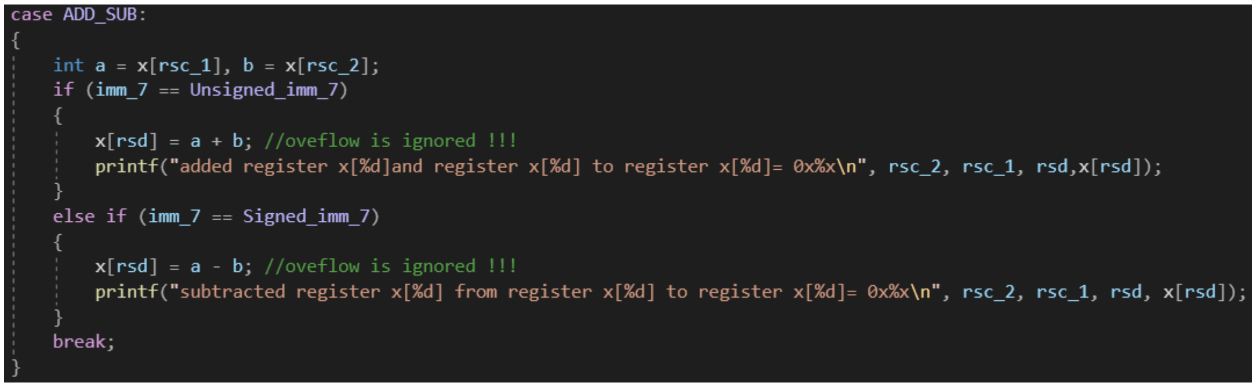
```
case SLLI:
{
    x[rsd] = x[rsc] << imm_5;
    printf("logical unsigned left shifted by %d : x[%d] = 0x%x\n", imm_5, rsd, x[rsd]);
    break;
}
```

- **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.



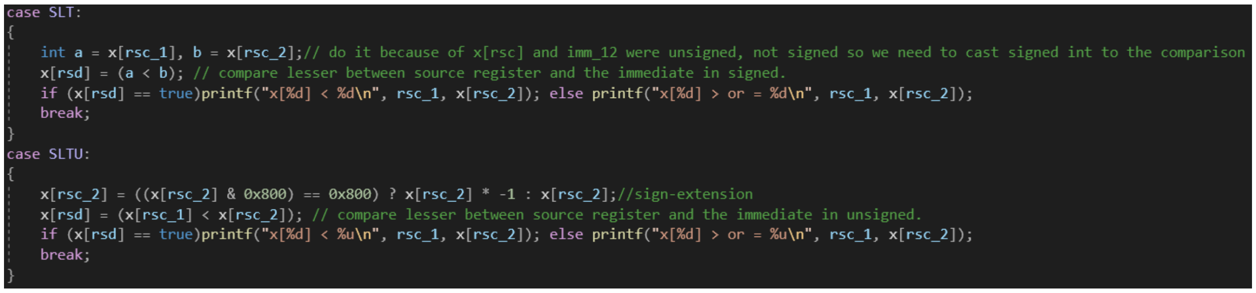
```
case SRLI_SRAI:
{
    if ( imm_7 == 0x20 )
    {
        x[rsd] = (x[rsc] >> imm_5) + ((0xffffffff >> (imm_5))^0xffffffff);
        printf("arithmetic signed right shifted by %d : x[%d] = 0x%x\n", imm_5, rsd, x[rsd]);
    }
    else if( imm_7 == 0x0)
    {
        x[rsd] = (x[rsc] >> imm_5);
        printf("logical unsigned right shifted by %d : x[%d] = 0x%x\n",imm_5,rsd,x[rsd]);
    }
    break;
}
```

- 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.



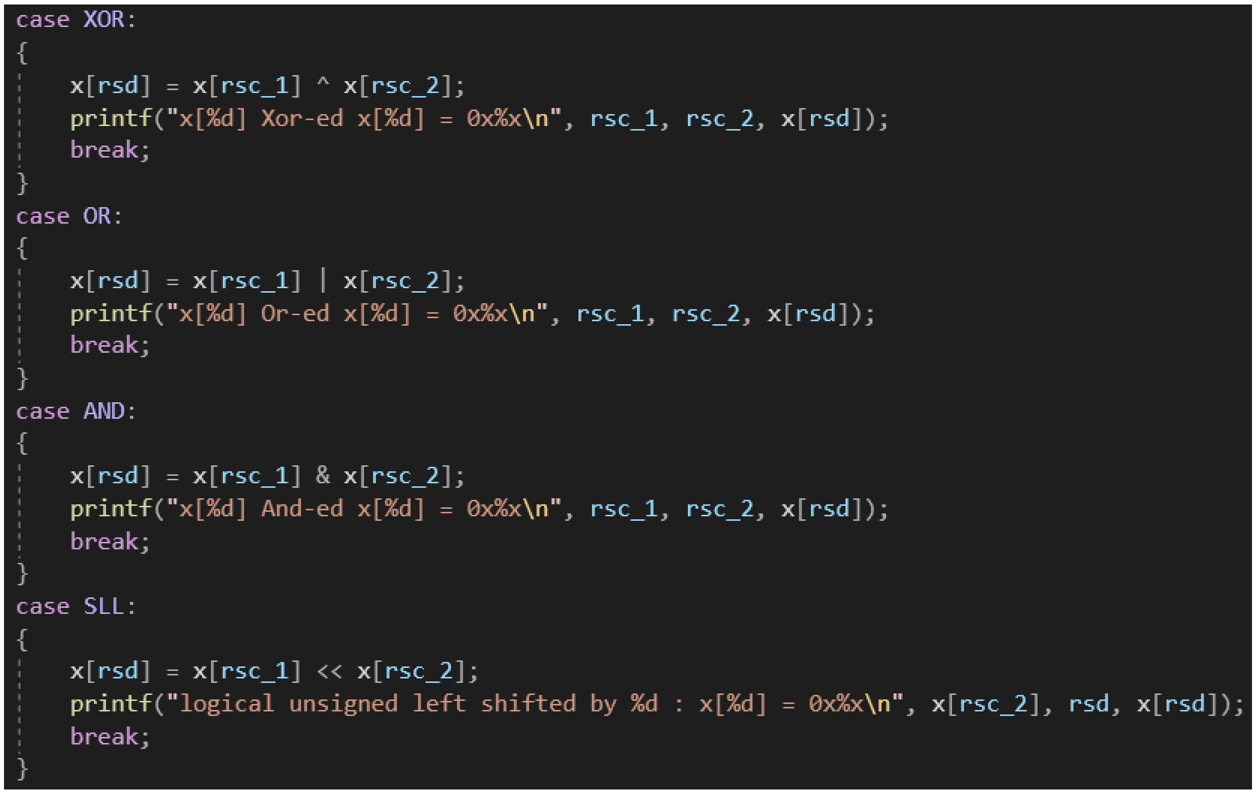
```
case ADD_SUB:
{
    int a = x[rsc_1], b = x[rsc_2];
    if (imm_7 == Unsigned_imm_7)
    {
        x[rsd] = a + b; //oveflow is ignored !!!
        printf("added register x[%d]and register x[%d] to register x[%d]= 0x%x\n", rsc_2, rsc_1, rsd,x[rsd]);
    }
    else if (imm_7 == Signed_imm_7)
    {
        x[rsd] = a - b; //oveflow is ignored !!!
        printf("subtracted register x[%d] from register x[%d] to register x[%d]= 0x%x\n", rsc_2, rsc_1, rsd, x[rsd]);
    }
    break;
}
```

  - 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.

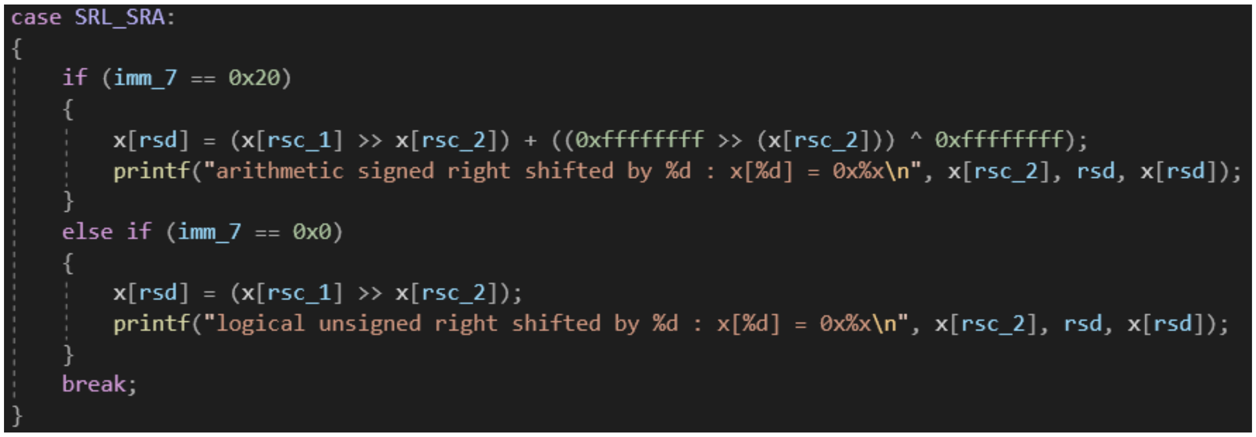


```
case SLT:
{
    int a = x[rsc_1], b = x[rsc_2];// do it because of x[rsc] and imm_12 were unsigned, not signed so we need to cast signed int to the comparison
    x[rsd] = (a < b); // compare lesser between source register and the immediate in signed.
    if (x[rsd] == true)printf("x[%d] < %d\n", rsc_1, x[rsc_2]); else printf("x[%d] > or = %d\n", rsc_1, x[rsc_2]);
    break;
}
case SLTU:
{
    x[rsc_2] = ((x[rsc_2] & 0x800) == 0x800) ? x[rsc_2] * -1 : x[rsc_2];//sign-extension
    x[rsd] = (x[rsc_1] < x[rsc_2]); // compare lesser between source register and the immediate in unsigned.
    if (x[rsd] == true)printf("x[%d] < %u\n", rsc_1, x[rsc_2]); else printf("x[%d] > or = %u\n", rsc_1, x[rsc_2]);
    break;
}
```

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



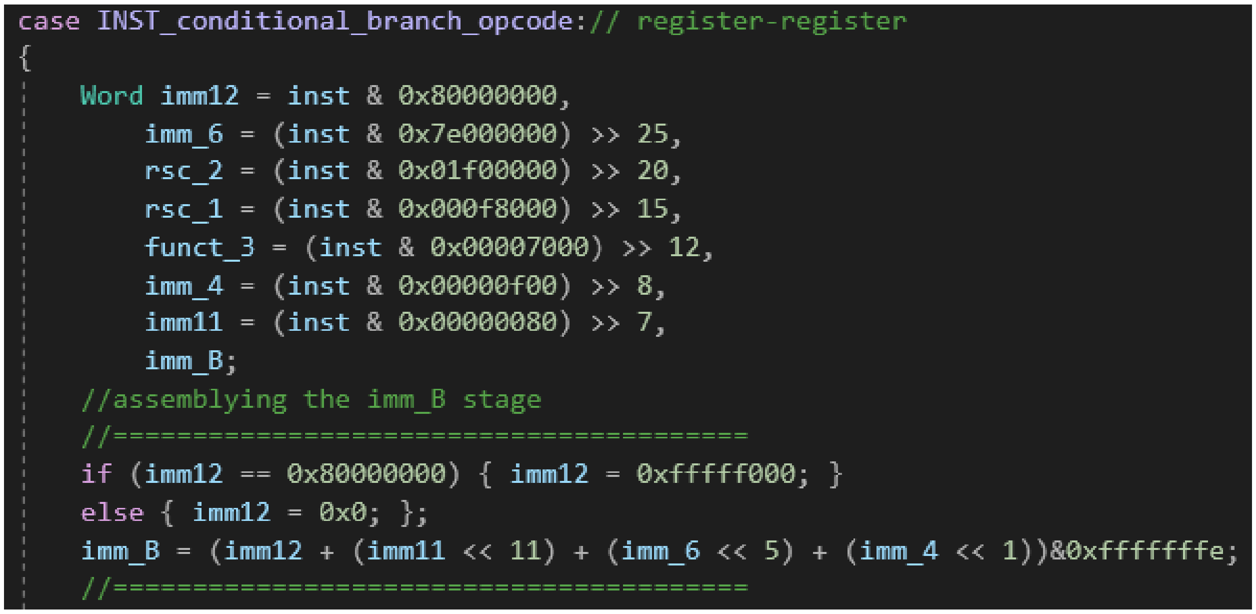
```
case XOR:
{
    x[rsd] = x[rsc_1] ^ x[rsc_2];
    printf("x[%d] Xor-ed x[%d] = 0x%x\n", rsc_1, rsc_2, x[rsd]);
    break;
}
case OR:
{
    x[rsd] = x[rsc_1] | x[rsc_2];
    printf("x[%d] Or-ed x[%d] = 0x%x\n", rsc_1, rsc_2, x[rsd]);
    break;
}
case AND:
{
    x[rsd] = x[rsc_1] & x[rsc_2];
    printf("x[%d] And-ed x[%d] = 0x%x\n", rsc_1, rsc_2, x[rsd]);
    break;
}
case SLL:
{
    x[rsd] = x[rsc_1] << x[rsc_2];
    printf("logical unsigned left shifted by %d : x[%d] = 0x%x\n", x[rsc_2], rsd, x[rsd]);
    break;
}
```



```
case SRL_SRA:
{
    if (imm_7 == 0x20)
    {
        x[rsd] = (x[rsc_1] >> x[rsc_2]) + ((0xffffffff >> (x[rsc_2])) ^ 0xffffffff);
        printf("arithmetic signed right shifted by %d : x[%d] = 0x%x\n", x[rsc_2], rsd, x[rsd]);
    }
    else if (imm_7 == 0x0)
    {
        x[rsd] = (x[rsc_1] >> x[rsc_2]);
        printf("logical unsigned right shifted by %d : x[%d] = 0x%x\n", x[rsc_2], rsd, x[rsd]);
    }
    break;
}
```

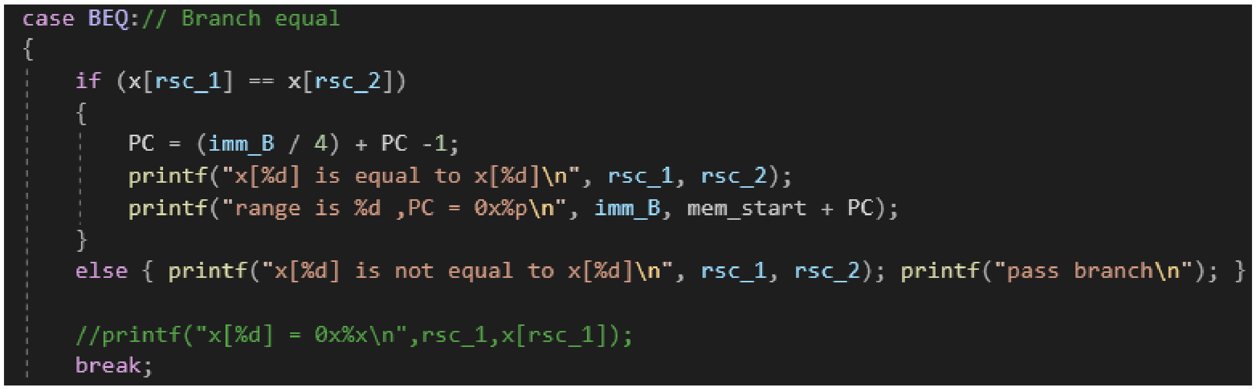
- The “**INST_conditional_branch_opcode**” is use for branching with condition that rely on the comparison of the two source registers 1 and 2. The immediate value of this type of function is also special because it is an immediate that has the least significant bit set to 0 because of this function 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:



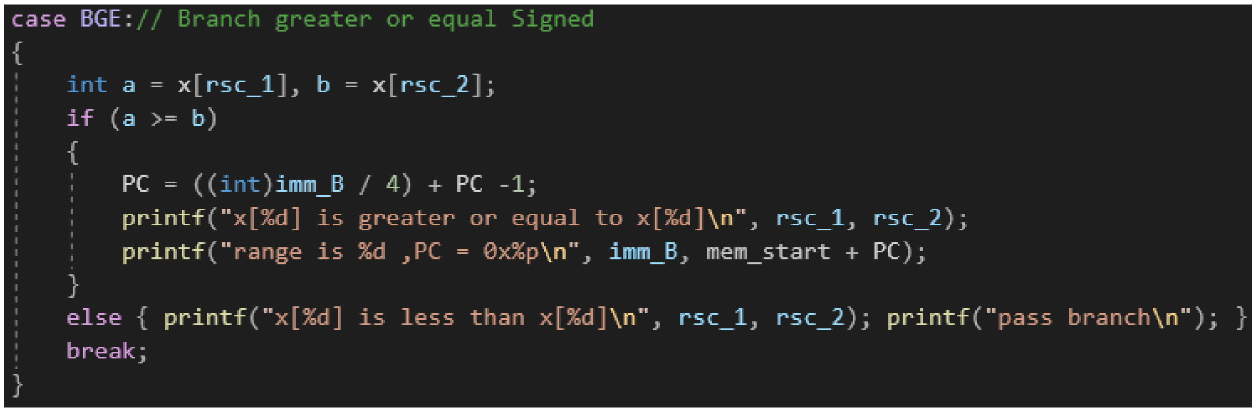
```
case INST_conditional_branch_opcode:// register-register
{
    Word imm12 = inst & 0x80000000,
        imm_6 = (inst & 0x7e000000) >> 25,
        rsc_2 = (inst & 0x01f00000) >> 20,
        rsc_1 = (inst & 0x000f8000) >> 15,
        funct_3 = (inst & 0x00007000) >> 12,
        imm_4 = (inst & 0x00000f00) >> 8,
        imm11 = (inst & 0x00000080) >> 7,
        imm_B;
    //assemblying the imm_B stage
    //========================================
    if (imm12 == 0x80000000) { imm12 = 0xfffff000; }
    else { imm12 = 0x0; };
    imm_B = (imm12 + (imm11 << 11) + (imm_6 << 5) + (imm_4 << 1))&0xfffffffe;
    //========================================
```

- **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)



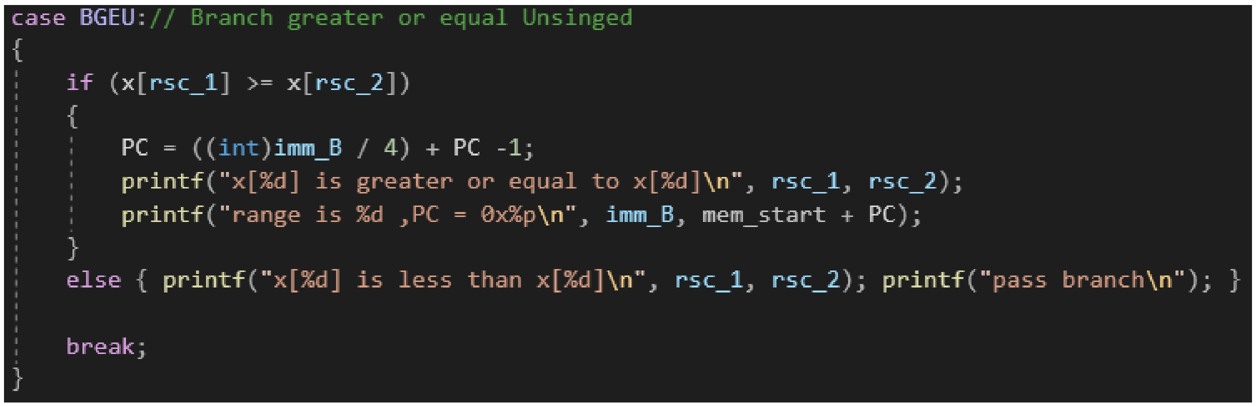
```c
case BEQ:// Branch equal
{
    if (x[rsc_1] == x[rsc_2])
    {
        PC = (imm_B / 4) + PC -1;
        printf("x[%d] is equal to x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is not equal to x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }

    //printf("x[%d] = 0x%x\n",rsc_1,x[rsc_1]);
    break;
```

- **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)



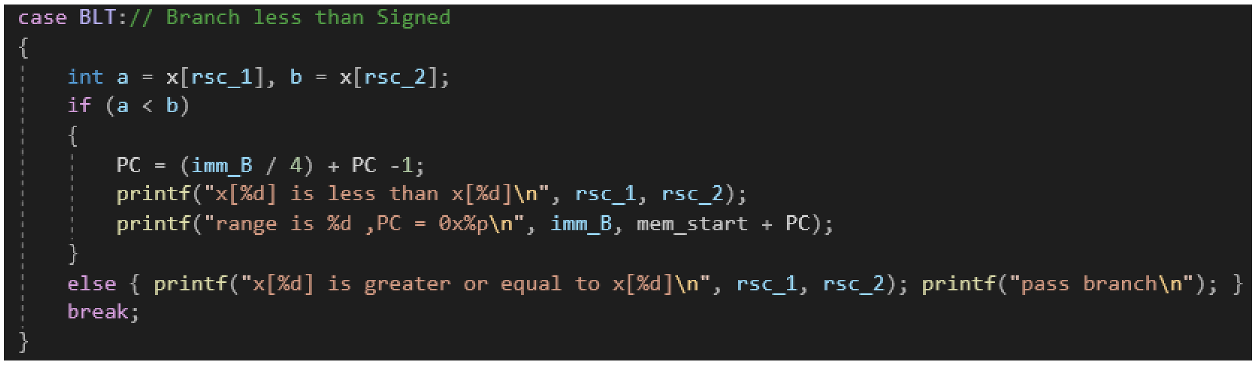
```c
case BGE:// Branch greater or equal Signed
{
    int a = x[rsc_1], b = x[rsc_2];
    if (a >= b)
    {
        PC = ((int)imm_B / 4) + PC -1;
        printf("x[%d] is greater or equal to x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is less than x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }
    break;
}
```

- **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)



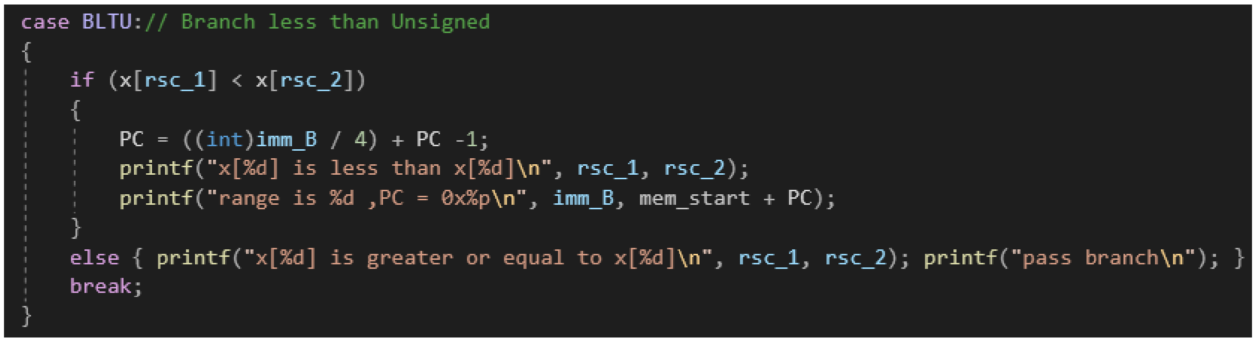
```c
case BGEU:// Branch greater or equal Unsinged
{
    if (x[rsc_1] >= x[rsc_2])
    {
        PC = ((int)imm_B / 4) + PC -1;
        printf("x[%d] is greater or equal to x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is less than x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }

    break;
}
```

- **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)



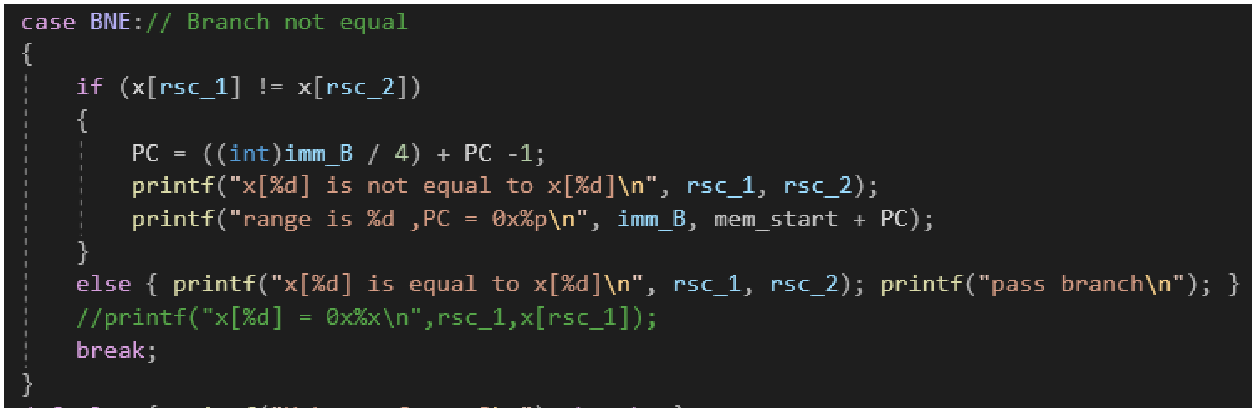
```
case BLT:// Branch less than Signed
{
    int a = x[rsc_1], b = x[rsc_2];
    if (a < b)
    {
        PC = (imm_B / 4) + PC -1;
        printf("x[%d] is less than x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is greater or equal to x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }
    break;
}
```

- **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)



```
case BLTU:// Branch less than Unsigned
{
    if (x[rsc_1] < x[rsc_2])
    {
        PC = ((int)imm_B / 4) + PC -1;
        printf("x[%d] is less than x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is greater or equal to x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }
    break;
}
```

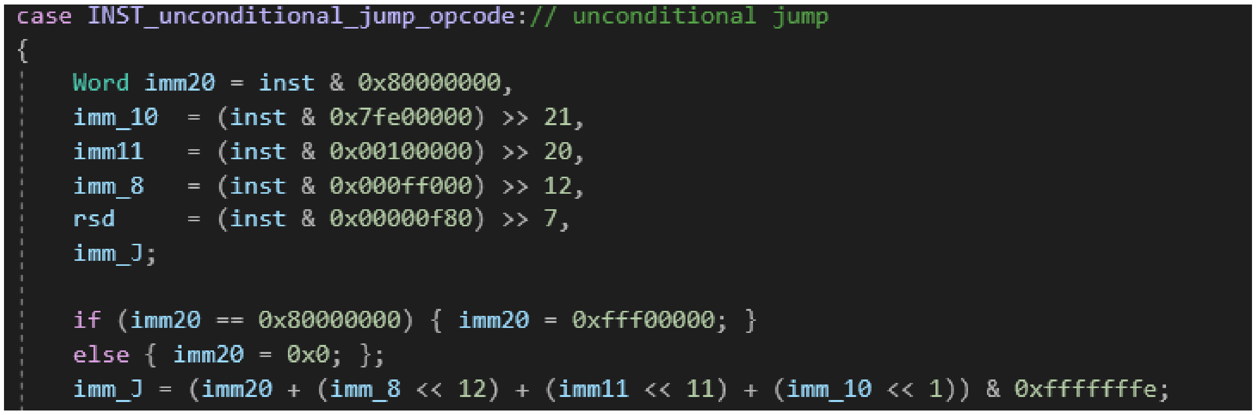
- **BNE** is the function to check if the source register 1 is not equal to the source register 2 and then branch if the statement is true. (source registers are both compared in signed value)



```
case BNE:// Branch not equal
{
    if (x[rsc_1] != x[rsc_2])
    {
        PC = ((int)imm_B / 4) + PC -1;
        printf("x[%d] is not equal to x[%d]\n", rsc_1, rsc_2);
        printf("range is %d ,PC = 0x%p\n", imm_B, mem_start + PC);
    }
    else { printf("x[%d] is equal to x[%d]\n", rsc_1, rsc_2); printf("pass branch\n"); }
    //printf("x[%d] = 0x%x\n",rsc_1,x[rsc_1]);
    break;
}
```

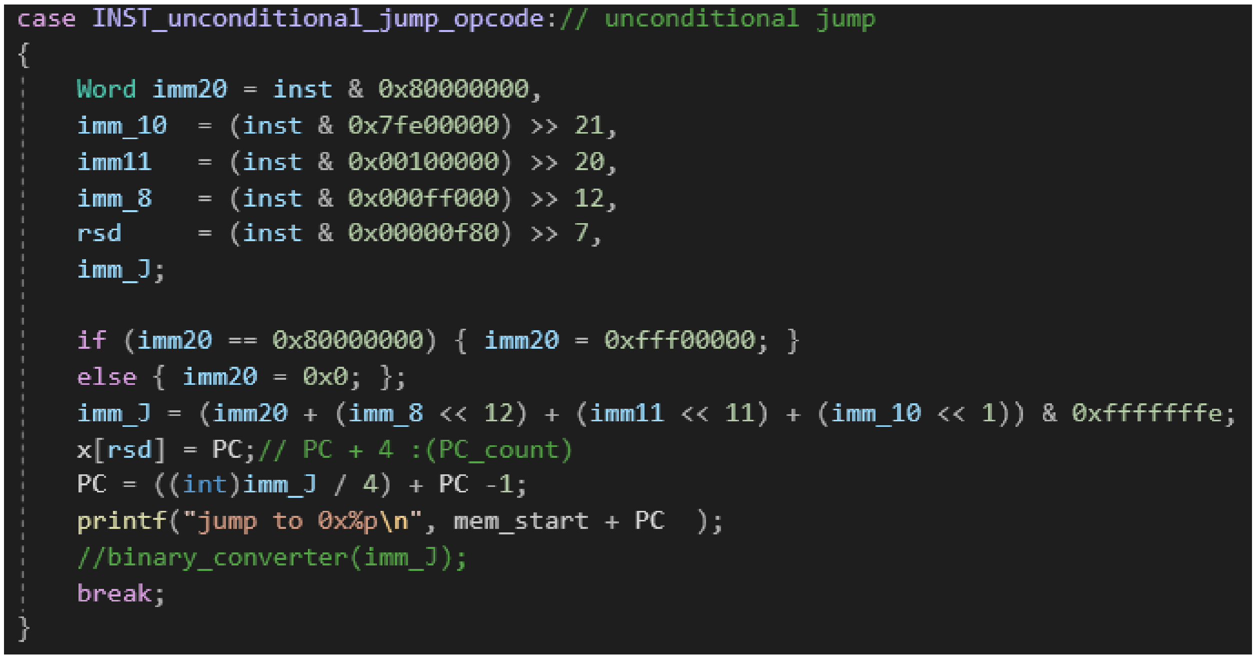
▪ The "**INST_unconditional_jump_opcode**" is to jump unconditionally to another address the characteristic of this function is similar to branch functions . But the different is it jump without comparing or checking any statement and also retain the (**PC**+4) address in a destination register for jumping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

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



```
case INST_unconditional_jump_opcode:// unconditional jump
{
    Word imm20 = inst & 0x80000000,
    imm_10   = (inst & 0x7fe00000) >> 21,
    imm11    = (inst & 0x00100000) >> 20,
    imm_8    = (inst & 0x000ff000) >> 12,
    rsd      = (inst & 0x00000f80) >> 7,
    imm_J;

    if (imm20 == 0x80000000) { imm20 = 0xfff00000; }
    else { imm20 = 0x0; };
    imm_J = (imm20 + (imm_8 << 12) + (imm11 << 11) + (imm_10 << 1)) & 0xfffffffe;
```

- **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.



```
case INST_unconditional_jump_opcode:// unconditional jump
{
    Word imm20 = inst & 0x80000000,
    imm_10  = (inst & 0x7fe00000) >> 21,
    imm11   = (inst & 0x00100000) >> 20,
    imm_8   = (inst & 0x000ff000) >> 12,
    rsd     = (inst & 0x00000f80) >> 7,
    imm_J;

    if (imm20 == 0x80000000) { imm20 = 0xfff00000; }
    else { imm20 = 0x0; };
    imm_J = (imm20 + (imm_8 << 12) + (imm11 << 11) + (imm_10 << 1)) & 0xfffffffe;
    x[rsd] = PC;// PC + 4 :(PC_count)
    PC = ((int)imm_J / 4) + PC -1;
    printf("jump to 0x%p\n", mem_start + PC  );
    //binary_converter(imm_J);
    break;
}
```

- **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.

2.3 Testing an example and result

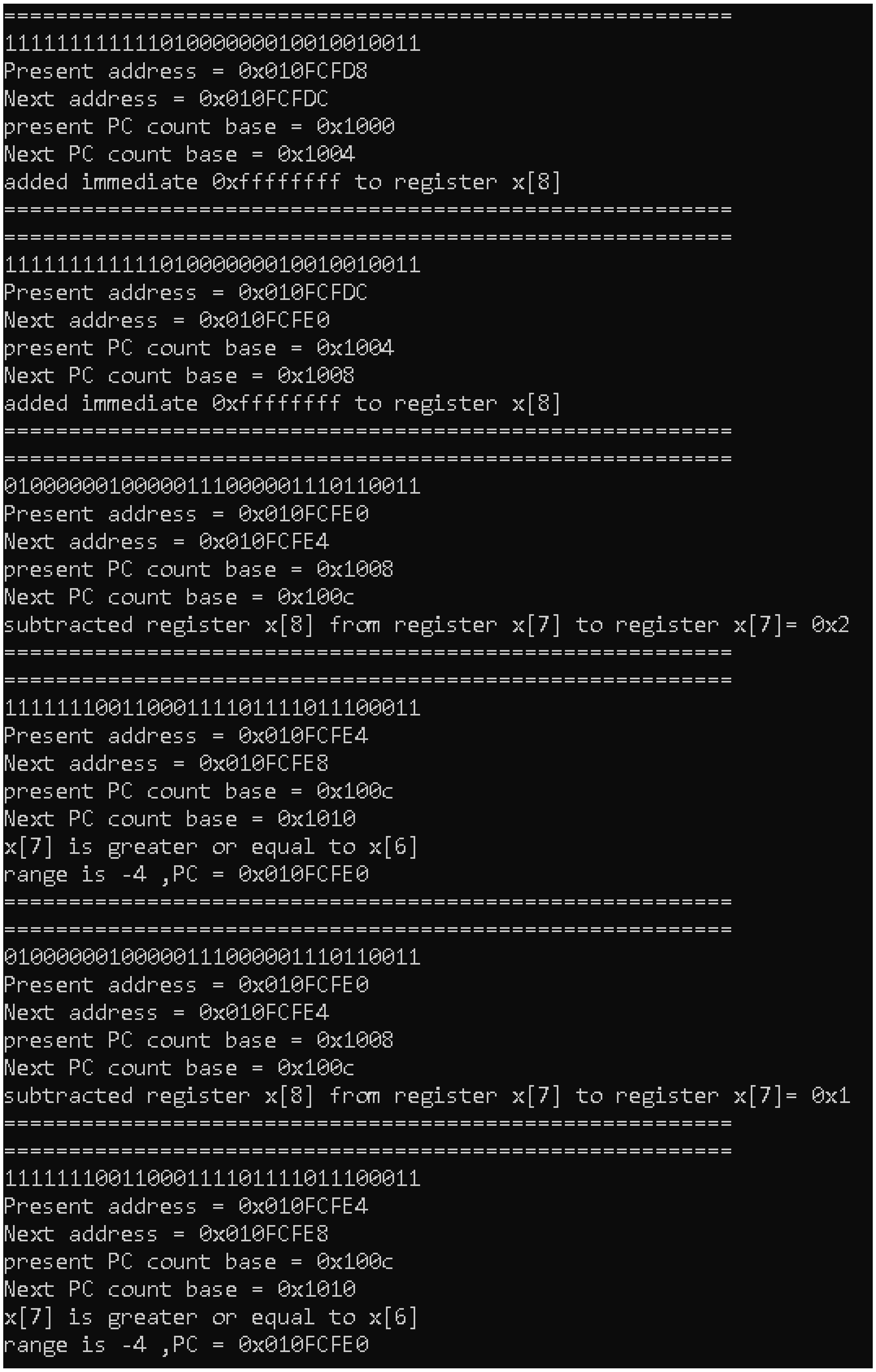
In other to test the program we manual put instruction into the memory by hand (we haven't achieve create an assembler to translate asm code to machine code).

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

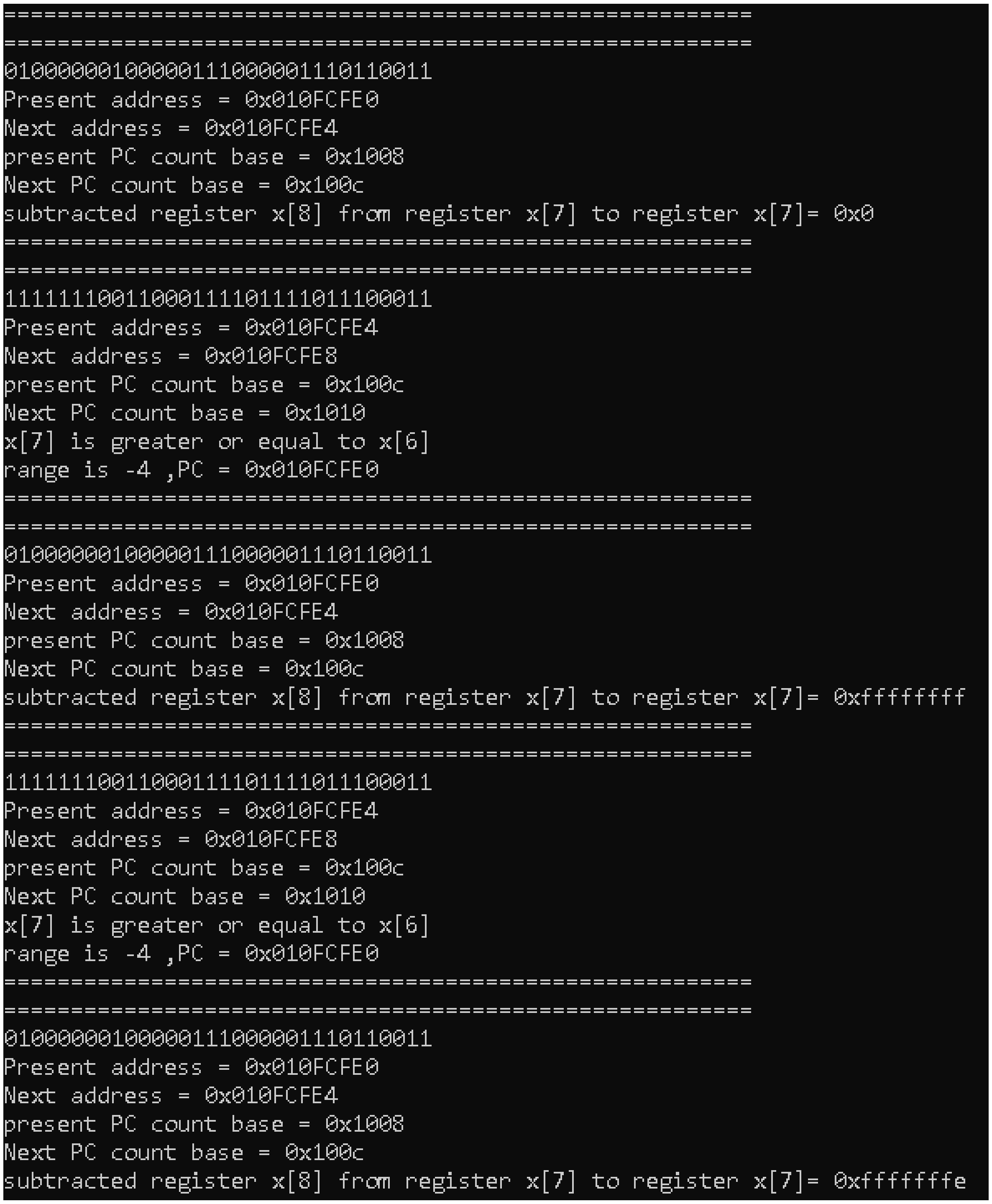
Initial condition: **x6** = -1; **x7** = 3; **x8** = 1;

```
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;
}
```

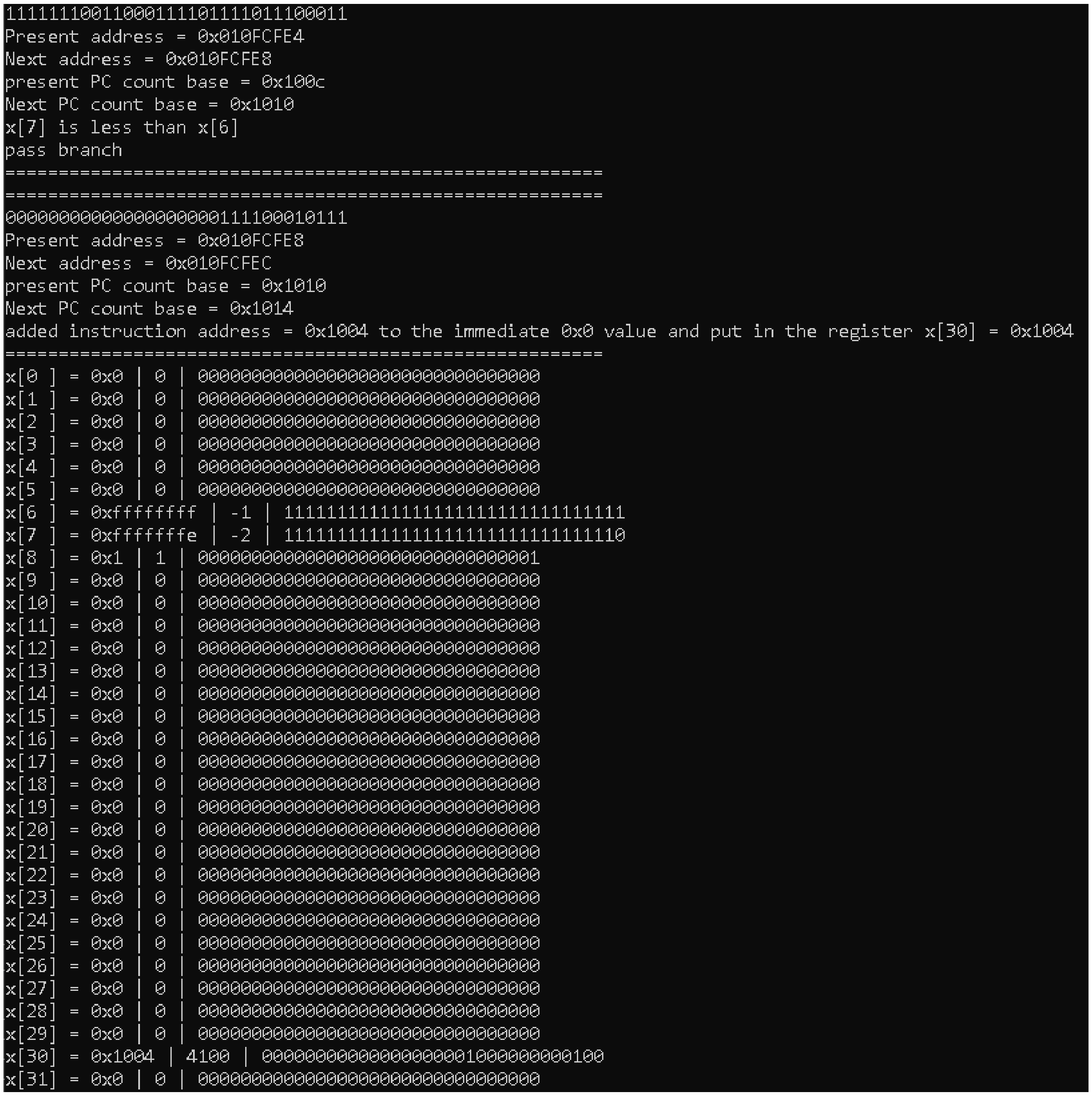
We then run the program and get the result below.



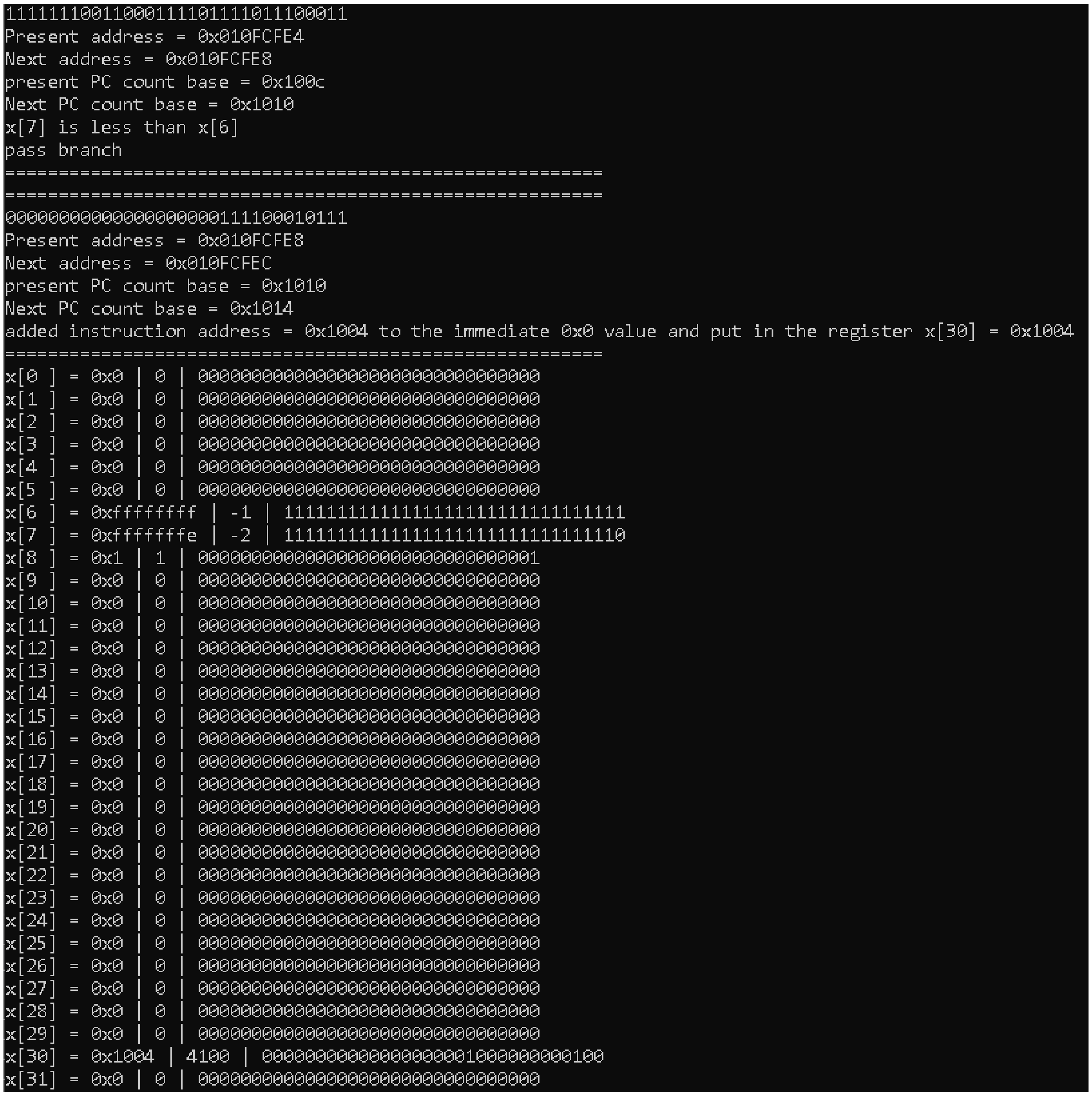
```
=======================================================
11111111111101000000010010010011
Present address = 0x010FCFD8
Next address = 0x010FCFDC
present PC count base = 0x1000
Next PC count base = 0x1004
added immediate 0xffffffff to register x[8]
=======================================================
=======================================================
11111111111101000000010010010011
Present address = 0x010FCFDC
Next address = 0x010FCFE0
present PC count base = 0x1004
Next PC count base = 0x1008
added immediate 0xffffffff to register x[8]
=======================================================
=======================================================
01000000100000111000001110110011
Present address = 0x010FCFE0
Next address = 0x010FCFE4
present PC count base = 0x1008
Next PC count base = 0x100c
subtracted register x[8] from register x[7] to register x[7]= 0x2
=======================================================
=======================================================
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is greater or equal to x[6]
range is -4 ,PC = 0x010FCFE0
=======================================================
=======================================================
01000000100000111000001110110011
Present address = 0x010FCFE0
Next address = 0x010FCFE4
present PC count base = 0x1008
Next PC count base = 0x100c
subtracted register x[8] from register x[7] to register x[7]= 0x1
=======================================================
=======================================================
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is greater or equal to x[6]
range is -4 ,PC = 0x010FCFE0
```



```
=========================================================
=========================================================
01000000100000111000001110110011
Present address = 0x010FCFE0
Next address = 0x010FCFE4
present PC count base = 0x1008
Next PC count base = 0x100c
subtracted register x[8] from register x[7] to register x[7]= 0x0
=========================================================
=========================================================
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is greater or equal to x[6]
range is -4 ,PC = 0x010FCFE0
=========================================================
=========================================================
01000000100000111000001110110011
Present address = 0x010FCFE0
Next address = 0x010FCFE4
present PC count base = 0x1008
Next PC count base = 0x100c
subtracted register x[8] from register x[7] to register x[7]= 0xffffffff
=========================================================
=========================================================
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is greater or equal to x[6]
range is -4 ,PC = 0x010FCFE0
=========================================================
=========================================================
01000000100000111000001110110011
Present address = 0x010FCFE0
Next address = 0x010FCFE4
present PC count base = 0x1008
Next PC count base = 0x100c
subtracted register x[8] from register x[7] to register x[7]= 0xfffffffe
```



```
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is less than x[6]
pass branch
========================================================
========================================================
00000000000000000000111100010111
Present address = 0x010FCFE8
Next address = 0x010FCFEC
present PC count base = 0x1010
Next PC count base = 0x1014
added instruction address = 0x1004 to the immediate 0x0 value and put in the register x[30] = 0x1004
========================================================
x[0 ] = 0x0 | 0 | 00000000000000000000000000000000
x[1 ] = 0x0 | 0 | 00000000000000000000000000000000
x[2 ] = 0x0 | 0 | 00000000000000000000000000000000
x[3 ] = 0x0 | 0 | 00000000000000000000000000000000
x[4 ] = 0x0 | 0 | 00000000000000000000000000000000
x[5 ] = 0x0 | 0 | 00000000000000000000000000000000
x[6 ] = 0xffffffff | -1 | 11111111111111111111111111111111
x[7 ] = 0xfffffffe | -2 | 11111111111111111111111111111110
x[8 ] = 0x1 | 1 | 00000000000000000000000000000001
x[9 ] = 0x0 | 0 | 00000000000000000000000000000000
x[10] = 0x0 | 0 | 00000000000000000000000000000000
x[11] = 0x0 | 0 | 00000000000000000000000000000000
x[12] = 0x0 | 0 | 00000000000000000000000000000000
x[13] = 0x0 | 0 | 00000000000000000000000000000000
x[14] = 0x0 | 0 | 00000000000000000000000000000000
x[15] = 0x0 | 0 | 00000000000000000000000000000000
x[16] = 0x0 | 0 | 00000000000000000000000000000000
x[17] = 0x0 | 0 | 00000000000000000000000000000000
x[18] = 0x0 | 0 | 00000000000000000000000000000000
x[19] = 0x0 | 0 | 00000000000000000000000000000000
x[20] = 0x0 | 0 | 00000000000000000000000000000000
x[21] = 0x0 | 0 | 00000000000000000000000000000000
x[22] = 0x0 | 0 | 00000000000000000000000000000000
x[23] = 0x0 | 0 | 00000000000000000000000000000000
x[24] = 0x0 | 0 | 00000000000000000000000000000000
x[25] = 0x0 | 0 | 00000000000000000000000000000000
x[26] = 0x0 | 0 | 00000000000000000000000000000000
x[27] = 0x0 | 0 | 00000000000000000000000000000000
x[28] = 0x0 | 0 | 00000000000000000000000000000000
x[29] = 0x0 | 0 | 00000000000000000000000000000000
x[30] = 0x1004 | 4100 | 00000000000000000001000000000100
x[31] = 0x0 | 0 | 00000000000000000000000000000000
```

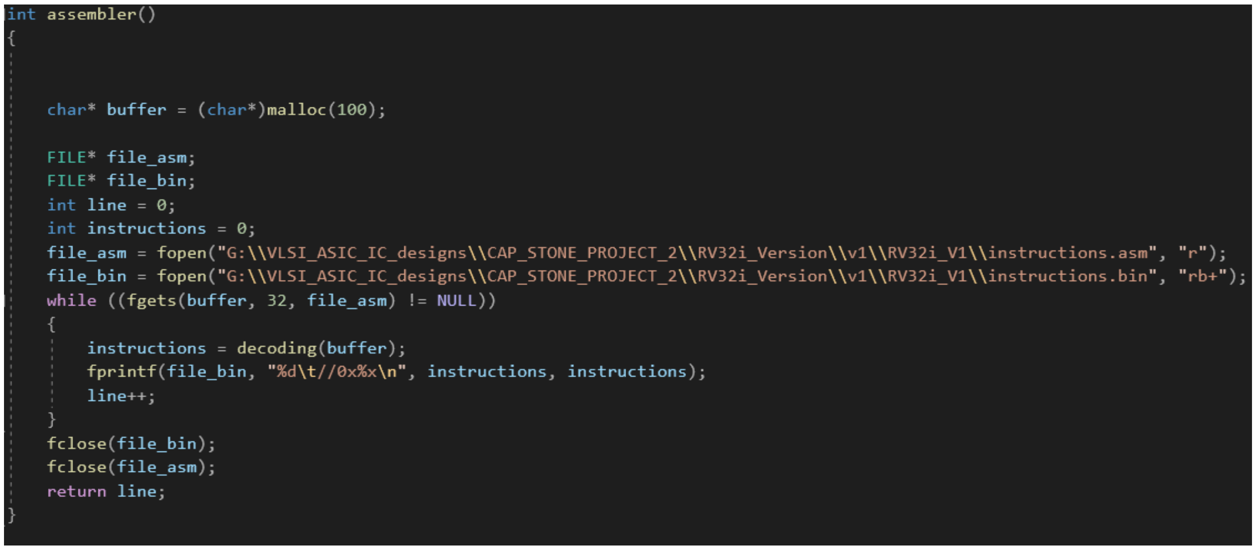


```
11111110011000111101111011100011
Present address = 0x010FCFE4
Next address = 0x010FCFE8
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is less than x[6]
pass branch
========================================================
========================================================
00000000000000000000111100010111
Present address = 0x010FCFE8
Next address = 0x010FCFEC
present PC count base = 0x1010
Next PC count base = 0x1014
added instruction address = 0x1004 to the immediate 0x0 value and put in the register x[30] = 0x1004
========================================================
x[0 ] = 0x0 | 0 | 00000000000000000000000000000000
x[1 ] = 0x0 | 0 | 00000000000000000000000000000000
x[2 ] = 0x0 | 0 | 00000000000000000000000000000000
x[3 ] = 0x0 | 0 | 00000000000000000000000000000000
x[4 ] = 0x0 | 0 | 00000000000000000000000000000000
x[5 ] = 0x0 | 0 | 00000000000000000000000000000000
x[6 ] = 0xffffffff | -1 | 11111111111111111111111111111111
x[7 ] = 0xfffffffe | -2 | 11111111111111111111111111111110
x[8 ] = 0x1 | 1 | 00000000000000000000000000000001
x[9 ] = 0x0 | 0 | 00000000000000000000000000000000
x[10] = 0x0 | 0 | 00000000000000000000000000000000
x[11] = 0x0 | 0 | 00000000000000000000000000000000
x[12] = 0x0 | 0 | 00000000000000000000000000000000
x[13] = 0x0 | 0 | 00000000000000000000000000000000
x[14] = 0x0 | 0 | 00000000000000000000000000000000
x[15] = 0x0 | 0 | 00000000000000000000000000000000
x[16] = 0x0 | 0 | 00000000000000000000000000000000
x[17] = 0x0 | 0 | 00000000000000000000000000000000
x[18] = 0x0 | 0 | 00000000000000000000000000000000
x[19] = 0x0 | 0 | 00000000000000000000000000000000
x[20] = 0x0 | 0 | 00000000000000000000000000000000
x[21] = 0x0 | 0 | 00000000000000000000000000000000
x[22] = 0x0 | 0 | 00000000000000000000000000000000
x[23] = 0x0 | 0 | 00000000000000000000000000000000
x[24] = 0x0 | 0 | 00000000000000000000000000000000
x[25] = 0x0 | 0 | 00000000000000000000000000000000
x[26] = 0x0 | 0 | 00000000000000000000000000000000
x[27] = 0x0 | 0 | 00000000000000000000000000000000
x[28] = 0x0 | 0 | 00000000000000000000000000000000
x[29] = 0x0 | 0 | 00000000000000000000000000000000
x[30] = 0x1004 | 4100 | 00000000000000000001000000000100
x[31] = 0x0 | 0 | 00000000000000000000000000000000
```

***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.

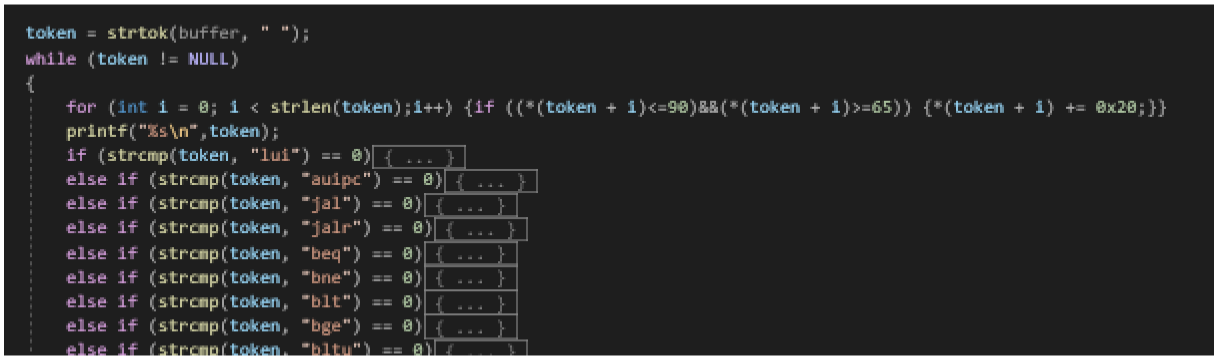
We start from making file pointer for the ".asm" file and a file pointer for ".bin"



```
int assembler()
{


    char* buffer = (char*)malloc(100);

    FILE* file_asm;
    FILE* file_bin;
    int line = 0;
    int instructions = 0;
    file_asm = fopen("G:\\VLSI_ASIC_IC_designs\\CAP_STONE_PROJECT_2\\RV32i_Version\\v1\\RV32i_V1\\instructions.asm", "r");
    file_bin = fopen("G:\\VLSI_ASIC_IC_designs\\CAP_STONE_PROJECT_2\\RV32i_Version\\v1\\RV32i_V1\\instructions.bin", "rb+");
    while ((fgets(buffer, 32, file_asm) != NULL))
    {
        instructions = decoding(buffer);
        fprintf(file_bin, "%d\t//0x%x\n", instructions, instructions);
        line++;
    }
    fclose(file_bin);
    fclose(file_asm);
    return line;
}
```

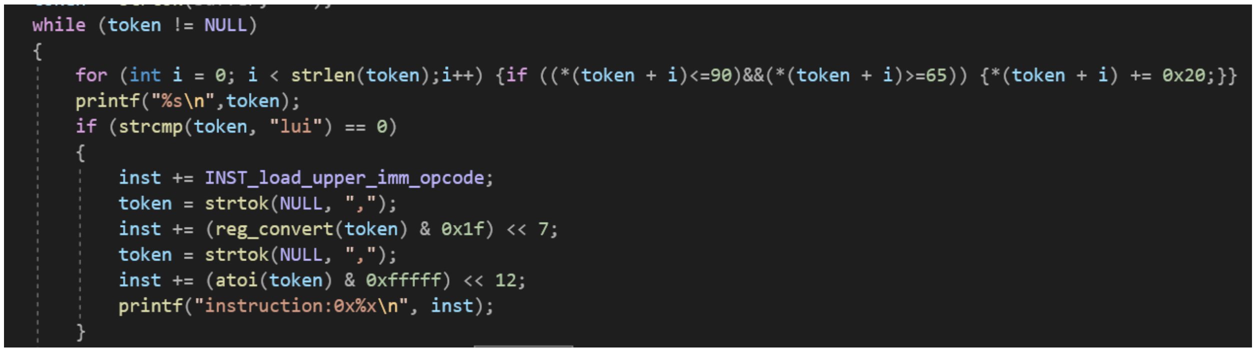
The buffer is memory allocated for reading strings from ".asm" file code lines. The instruction is to store the binary instruction not the field code in strings. The "fgets()" function read every single line in the file at each interval of the while loop then fetch the string to the decoding() function.

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



```
token = strtok(buffer, " ");
while (token != NULL)
{
    for (int i = 0; i < strlen(token);i++) {if ((*(token + i)<=90)&&(*(token + i)>=65)) {*(token + i) += 0x20;}}
    printf("%s\n",token);
    if (strcmp(token, "lui") == 0){ ... }
    else if (strcmp(token, "auipc") == 0){ ... }
    else if (strcmp(token, "jal") == 0){ ... }
    else if (strcmp(token, "jalr") == 0){ ... }
    else if (strcmp(token, "beq") == 0){ ... }
    else if (strcmp(token, "bne") == 0){ ... }
    else if (strcmp(token, "blt") == 0){ ... }
    else if (strcmp(token, "bge") == 0){ ... }
    else if (strcmp(token, "bltu") == 0){ ... }
```

An example of decoding the field code:

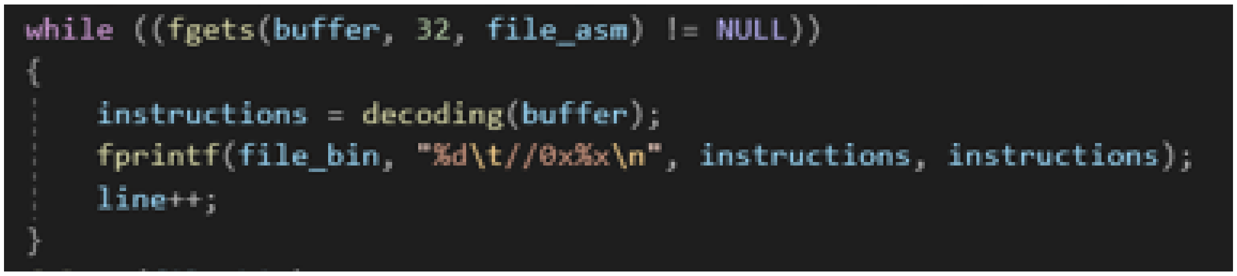


```
while (token != NULL)
{
    for (int i = 0; i < strlen(token);i++) {if ((*(token + i)<=90)&&(*(token + i)>=65)) {*(token + i) += 0x20;}}
    printf("%s\n",token);
    if (strcmp(token, "lui") == 0)
    {
        inst += INST_load_upper_imm_opcode;
        token = strtok(NULL, ",");
        inst += (reg_convert(token) & 0x1f) << 7;
        token = strtok(NULL, ",");
        inst += (atoi(token) & 0xfffff) << 12;
        printf("instruction:0x%x\n", inst);
    }
```

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.

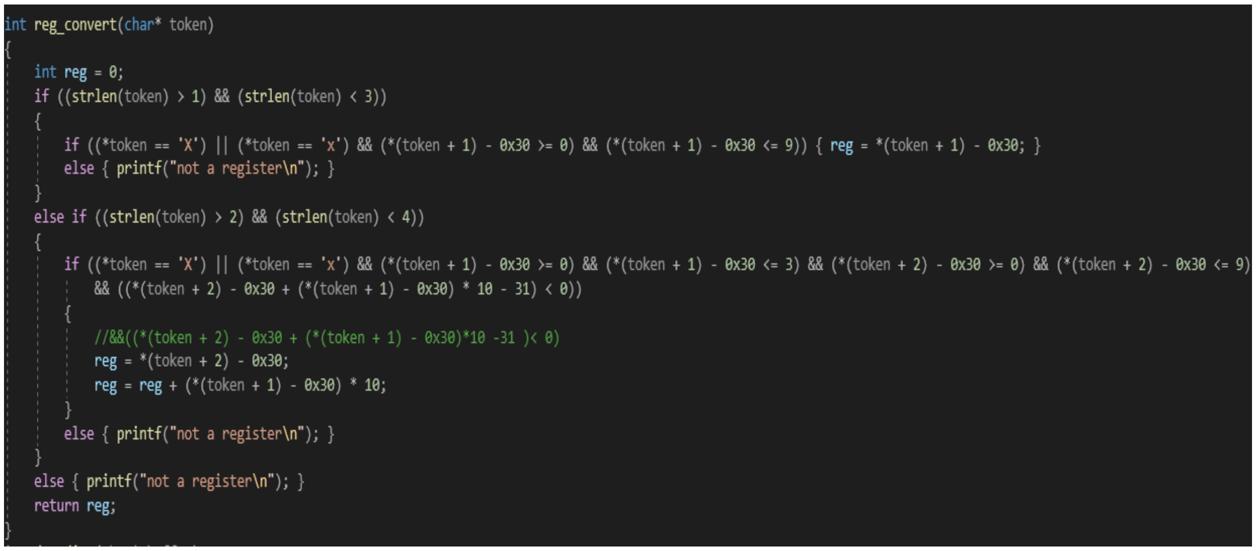
The instruction decoding is build by extracting “token” from the line read by the buffer which read from “.asm” file. Because of the sequential-ness in the reading of the token by the function “strtok()” we need to patch the instruction piece by piece because the strtok would increment to another field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



```
while ((fgets(buffer, 32, file_asm) != NULL))
{
    instructions = decoding(buffer);
    fprintf(file_bin, "%d\t//0x%x\n", instructions, instructions);
    line++;
}
```

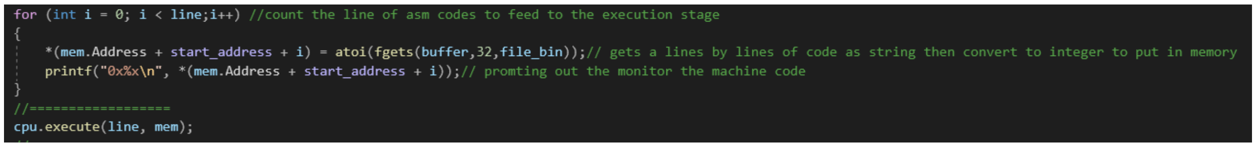
The reg_convert is to convert “x1” or any register coded field to numbers to be embedded into the instruction.



```
int reg_convert(char* token)
{
    int reg = 0;
    if ((strlen(token) > 1) && (strlen(token) < 3))
    {
        if ((*token == 'X') || (*token == 'x') && (*(token + 1) - 0x30 >= 0) && (*(token + 1) - 0x30 <= 9)) { reg = *(token + 1) - 0x30; }
        else { printf("not a register\n"); }
    }
    else if ((strlen(token) > 2) && (strlen(token) < 4))
    {
        if ((*token == 'X') || (*token == 'x') && (*(token + 1) - 0x30 >= 0) && (*(token + 1) - 0x30 <= 3) && (*(token + 2) - 0x30 >= 0) && (*(token + 2) - 0x30 <= 9)
            && ((*(token + 2) - 0x30 + (*(token + 1) - 0x30) * 10 - 31) < 0))
        {
            //&&((*(token + 2) - 0x30 + (*(token + 1) - 0x30)*10 -31 )< 0)
            reg = *(token + 2) - 0x30;
            reg = reg + (*(token + 1) - 0x30) * 10;
        }
        else { printf("not a register\n"); }
    }
    else { printf("not a register\n"); }
    return reg;
}
```

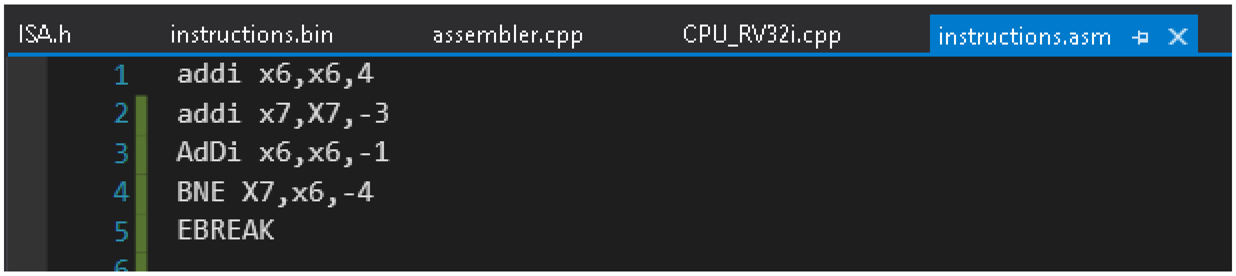
And this function also care none about capitalized character or not it will still accept the register field if wrote in the appropriate format.

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.



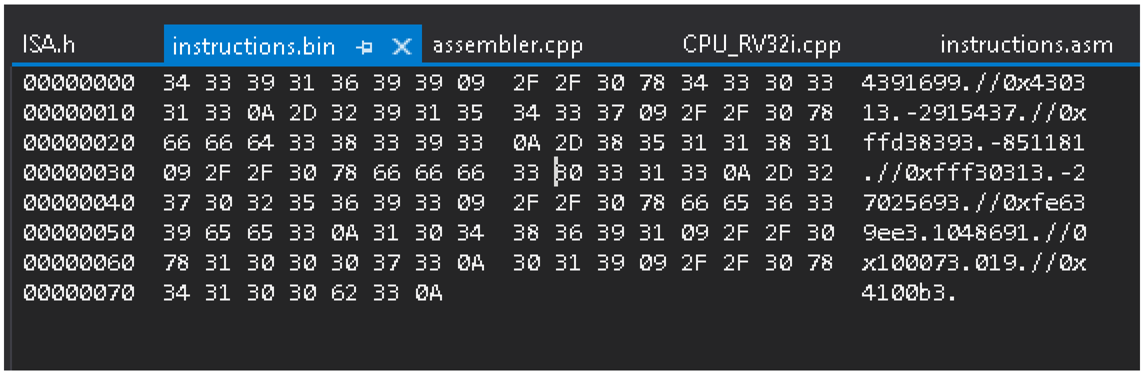
```
for (int i = 0; i < line;i++) //count the line of asm codes to feed to the execution stage
{
    *(mem.Address + start_address + i) = atoi(fgets(buffer,32,file_bin));// gets a lines by lines of code as string then convert to integer to put in memory
    printf("0x%x\n", *(mem.Address + start_address + i));// promting out the monitor the machine code
}
//==================
cpu.execute(line, mem);
```

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.



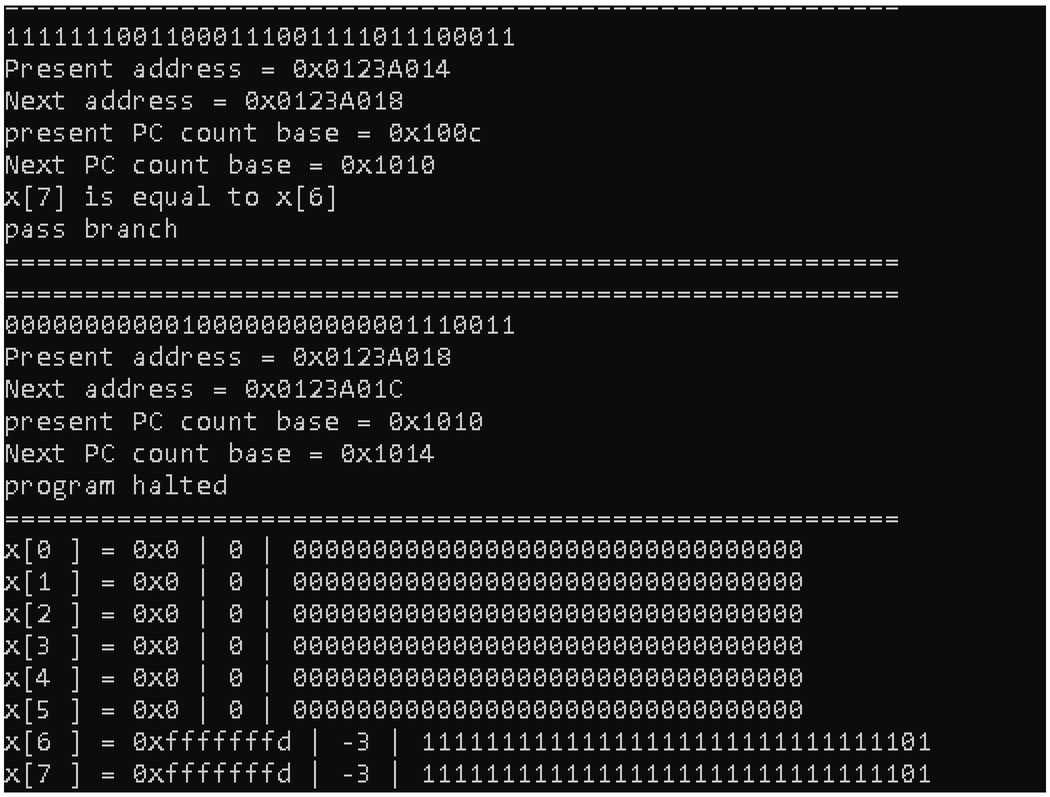
```
addi x6,x6,4
addi x7,X7,-3
AdDi x6,x6,-1
BNE X7,x6,-4
EBREAK
```

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



```
ISA.h    instructions.bin    assembler.cpp    CPU_RV32i.cpp    instructions.asm
00000000  34 33 39 31 36 39 39 09  2F 2F 30 78 34 33 30 33  4391699.//0x4303
00000010  31 33 0A 2D 32 39 31 35  34 33 37 09 2F 2F 30 78  13.-2915437.//0x
00000020  66 66 64 33 38 33 39 33  0A 2D 38 35 31 31 38 31  ffd38393.-851181
00000030  09 2F 2F 30 78 66 66 66  33 30 33 31 33 0A 2D 32  .//0xfff30313.-2
00000040  37 30 32 35 36 39 33 09  2F 2F 30 78 66 65 36 33  7025693.//0xfe63
00000050  39 65 65 33 0A 31 30 34  38 36 39 31 09 2F 2F 30  9ee3.1048691.//0
00000060  78 31 30 30 30 37 33 0A  30 31 39 09 2F 2F 30 78  x100073.019.//0x
00000070  34 31 30 30 62 33 0A                              4100b3.
```

And the result of the program:



```
========================================================
11111110011000111001111011100011
Present address = 0x0123A014
Next address = 0x0123A018
present PC count base = 0x100c
Next PC count base = 0x1010
x[7] is equal to x[6]
pass branch
========================================================
========================================================
00000000000100000000000001110011
Present address = 0x0123A018
Next address = 0x0123A01C
present PC count base = 0x1010
Next PC count base = 0x1014
program halted
========================================================
x[0 ] = 0x0 | 0 | 00000000000000000000000000000000
x[1 ] = 0x0 | 0 | 00000000000000000000000000000000
x[2 ] = 0x0 | 0 | 00000000000000000000000000000000
x[3 ] = 0x0 | 0 | 00000000000000000000000000000000
x[4 ] = 0x0 | 0 | 00000000000000000000000000000000
x[5 ] = 0x0 | 0 | 00000000000000000000000000000000
x[6 ] = 0xfffffffd | -3 | 11111111111111111111111111111101
x[7 ] = 0xfffffffd | -3 | 11111111111111111111111111111101
```

IV. 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.

V. REFERENCES:

[1] CS Division, EECS Department, University of California, Berkeley, August 24, 2021, "*The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*".

[2] John Winans, June 29, 2021, *" RISC-V 2 Assembly Language Programming"*

[3] Steven Ho,*" RISC-V CPU Datapath, Control Intro"*, https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture11.pdf?fbclid=IwAR3308_-BuzA7rz2_QQQwf1aiF1IBweUWdUzWobR9DPFypsnL6Xh2mROLA4

[4] Lê Quang Hưng, “RENAS MCU A Microcontroller using RISC-V ISA, AMBA Bus and SPI peripheral”.

[5] 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

[6] 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

[7] 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