VLSI II - Assignment 5

Mehmet Eymen Ünay 040190218
Deniz Bashgoren 040180902

RV32I Instruction Set

The table of all RV32I instructions and a short description in pseudocode:

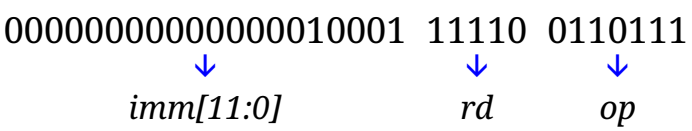
Opcode	Name	Pseudocode
LUI	Load upper immediate	`rd ← imm << 12`
AUIPC	Add upper immediate to PC	`rd ← pc + offset << 12`
JAL	Jump and link	`rd ← pc + length(inst)` `pc ← pc + offset`
JALR	Jump and link register	`rd ← pc + length(inst)` `pc ← (rs1 + offset) ∧ -2`
BEQ	Branch equal	`if rs1 = rs2 then pc ← pc + offset`
BNE	Branch not equal	`if rs1 ≠ rs2 then pc ← pc + offset`
BLT	Branch less than	`if rs1 < rs2 then pc ← pc + offset`
BGE	Branch greater	`if rs1 ≥ rs2 then pc ← pc + offset`
BLTU	Branch less than unsigned	`if rs1 < rs2 then pc ← pc + offset`
BGEU	Branch greater than unsigned	`if rs1 ≥ rs2 then pc ← pc + offset`
LB	Load byte	`rd ← s8[rs1 + offset]`
LH	Load halfword	`rd ← s16[rs1 + offset]`
LW	Load word	`rd ← s32[rs1 + offset]`
LBU	Load byte unsigned	`rd ← u8[rs1 + offset]`
LHU	Load halfword unsigned	`rd ← u16[rs1 + offset]`
SB	Store byte	`u8[rs1 + offset] ← rs2`
SH	Store halfword	`u16[rs1 + offset] ← rs2`
SW	Store word	`u32[rs1 + offset] ← rs2`
ADDI	Add immediate	`rd ← rs1 + sx(imm)`
SLTI	Set less than immediate	`rd ← sx(rs1) < sx(imm)`
SLTIU	Set less than immediate unsigned	`rd ← ux(rs1) < ux(imm)`
XORI	Exclusive or immediate	`rd ← ux(rs1) ^ ux(imm)`
ORI	Or immediate	`rd ← ux(rs1) | ux(imm)`
ANDI	And immediate	`rd ← ux(rs1) & ux(imm)`
SLLI	Shift left logic immediate	`rd ← ux(rs1) << ux(imm)`
SRLI	Shift right logic immediate	`rd ← ux(rs1) >> ux(imm)`
SRAI	Shift right arithmetic immediate	`rd ← sx(rs1) >> ux(imm)`
ADD	Add	`rd ← sx(rs1) + sx(rs2)`
SUB	Subtract	`rd ← sx(rs1) - sx(rs2)`
SLL	Shift left logical	`rd ← ux(rs1) << rs2`

Opcode	Name	Pseudocode
SLT	Set less than	`rd ← sx(rs1) < sx(rs2)`
SLTU	Set less than unsigned	`rd ← ux(rs1) < ux(rs2)`
XOR	Exclusive or	`rd ← ux(rs1) ^ ux(rs2)`
SRL	Shift right logic	`rd ← ux(rs1) >> rs2`
SRA	Shift right arithmetic	`rd ← sx(rs1) >> rs2`
OR	Or	`rd ← ux(rs1) | ux(rs2)`
AND	And	`rd ← ux(rs1) & ux(rs2)`

Encoding of RV32I Instructions

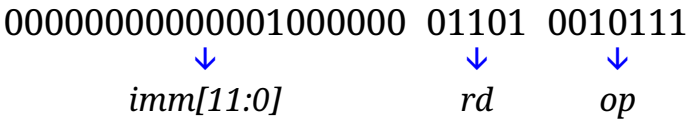
16 of the RV32I instructions have been selected and their encoding is analyzed below.

lui x30, 17
x30 ← 17 << 12
U-TYPE



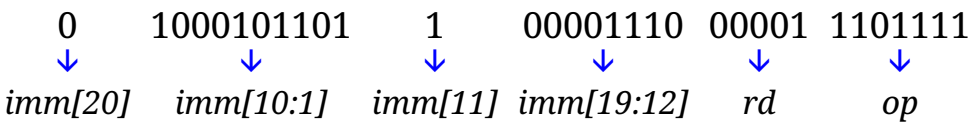
Fill x30's upper 20 bits with 17.

auipc x13, 64
x13 ← pc + 64 << 12
U-TYPE



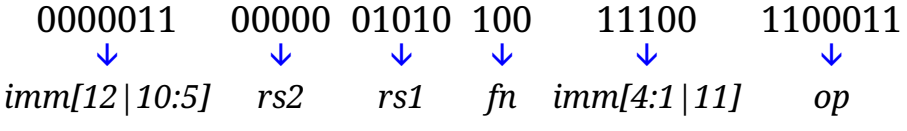
Fill x13 with pc+64<<12.

jal x1, 60506
x1 ← pc + 32
pc ← pc + 60506
J-TYPE



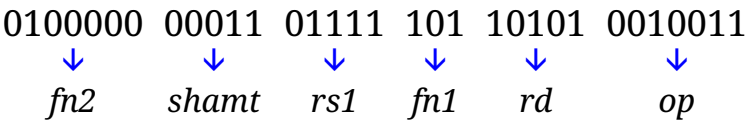
Save address of next instruction to x1, and jump 60506 addresses ahead.

blt x10, x0, 62
if x10 < x0 then
pc ← pc + 62
B-TYPE



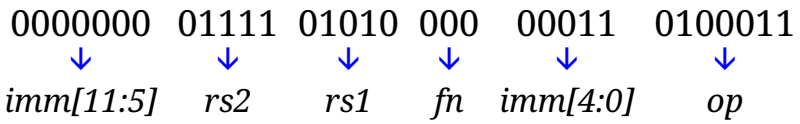
If the value in x10 is less than the value in x0, jump 62 addresses ahead.

srai x21, x15, 3
x21 ← sx(x15) >> 3
I-TYPE



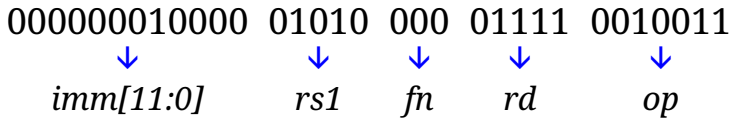
Shift bits stored in x15 3 bits to the right, while filling 3 new empty bits with the leftmost bit's value, and put it inside x21.

sb x15, 3(x10)
u8[x10 + 3] ← x15
S-TYPE



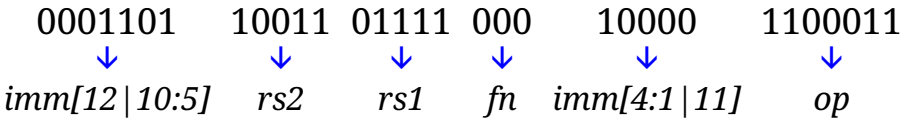
Take the lowest byte from x15's value, and store it in the address x10+3.

addi x15, x10, 16
x15 ← x10 + 16
I-TYPE



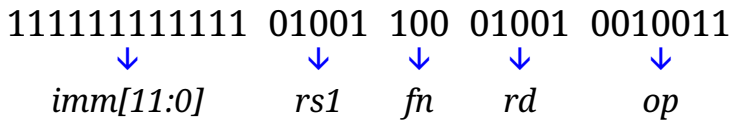
Add x10 with 16, put in x15.

beq x15, x19, 432
if x15 = x19 then
pc ← pc + 432
B-TYPE



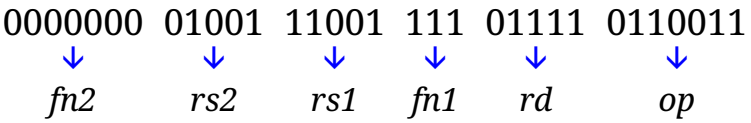
If the values in x15 and x19 are equal, jump 432 addresses ahead.

xori x9, x9, -1
x9 ← ux(x9) ^ -1
I-TYPE



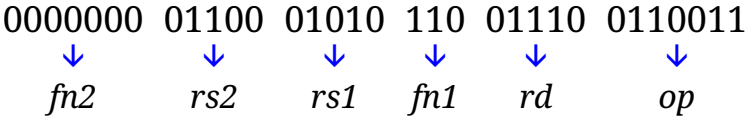
Perform an xor operation bit-by-bit of the value in x9 and -1 (all 1s), then put back in x9.

and x15, x25, x9
x15 ← ux(x25) &
ux(x9)
R-TYPE



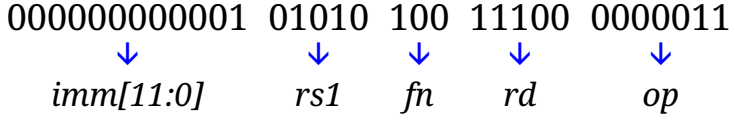
Perform a bitwise and operation on x25 and x9 then store the result in x15.

or x14, x10, x12
x14 ← ux(x10) |
ux(x12)
R-TYPE



Perform a bitwise or operation on x10 and x12 then store the result in x14.

lbu x28, 1(x10)
x28 ← u8[x10 + 1]
I-TYPE



Get the 8-bit value in the memory at the atddress x10+1, then put it on the lowest byte of x28, while filling the upper 3 bytes with zeros.

slli x15, x14, 5
x15 ← ux(x14) << 5
I-TYPE

0000000	00101	01110	001	01111	0010011
fn2	*shamt*	*rs1*	*fn1*	*rd*	*op*

Shift bits stored in x14 5 bits to the left, while filling the 5 new empty bits with zeros (hence 'logic' shift, and put it inside x15.

sw x16, 4(x25)
u32[x25 + 4] ← x16
S-TYPE

0000000	10000	11001	010	00100	0100011
imm[11:5]	*rs2*	*rs1*	*fn*	*imm[4:0]*	*op*

Take the low 4 bytes from x16's value, put them in the 4 addresses starting from x25+4. The endianness is set by EEI during the execution, so it's not guaranteed, but in either case, loading the same bytes using LW instruction should give bytes in the same order.

sub x15, x14, x8
x15 ← sx(x14) - sx(x8)
R-TYPE

0100000	01000	01110	000	01111	0110011
fn2	*rs2*	*rs1*	*fn1*	*rd*	*op*

Calculate the value x14-x8 by treating both registers as signed, and store the result in x15.

bne x15, x14, -26
if x15 ≠ x14 then pc ← pc - 26
B-TYPE

1111111	01110	01111	001	00111	1100011
imm[12|10:5]	*rs2*	*rs1*	*fn*	*imm[4:1|11]*	*op*

If the values in registers x15 and x14 are different, move 26 addresses backwards.

The NOP and HINT instructions

NOP is a pseudoinstruction that does nothing, but occupy space. This is desired in cases like:

- When we want to make sure there is enough space between certain instructions
- In jump tables (such as in switch statements in C), where the distance in instructions needs to be precise
- At the end of the program, to make sure that the program doesn't trigger a trap

In practice, any instruction that doesn't change the state of the registers and the memory can be used as a NOP, however, the RISC-V specification defines a canonical encoding for the NOP: `addi x0, x0, 0` (in hexadecimal: 0x00000013).

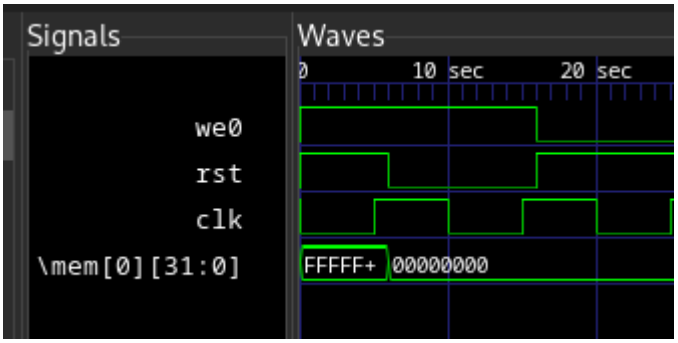
Other encodings of NOP have been named as HINT instructions in the standard. These have no effect on the visible state but may be used to communicate with the RISC-V

hardware in an implementation-specific way. For example, a hardware implementer might define that `slti x0, x1, 2` can be used to signal the hardware that the next branch is not likely to be taken. This permits the ISA users to mark branches unlikely to be taken in this way, thereby allowing the hardware to optimize its internal circuitry. The table of all reserved and non-reserved hint instructions:

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
FENCE	*pred*=0 or *succ*=0	$2^{5}-1$	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

Instruction Memory

In the testbench, we first load the instructions from the file `instInit.txt` into a register internal to the testbench, named `instructions`. The instInit file holds the instructions in hexadecimal format, however, if its binary equivalent is also provided as a seperate file `instInitBin.txt`. In the testbench we first demonstrate that rst works asynchronously. As seen in the waveform below, the contents of the memory change to 0 right when rst is activated.



Next, we demonstrate that while we=0, the write operations won't work. First we try to fill up the memory while we0=0, and then while we0=1.

```
Address:   0 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   1 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   2 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   3 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   4 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   5 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   6 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address:   7 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
...

Address:   0 Data_Hex: 00011f37 Data_Bin: 00000000000000010001111100110111
Address:   1 Data_Hex: 00040697 Data_Bin: 00000000000001000000011010010111
Address:   2 Data_Hex: 45b0e0ef Data_Bin: 01000101101100001110000011101111
Address:   3 Data_Hex: 06054e63 Data_Bin: 00000110000001010100111001100011
Address:   4 Data_Hex: 4037da93 Data_Bin: 01000000001101111101101010010011
Address:   5 Data_Hex: 00f501a3 Data_Bin: 00000000111101010000000110100011
Address:   6 Data_Hex: 01050793 Data_Bin: 00000001000001010000011110010011
Address:   7 Data_Hex: 1b378863 Data_Bin: 00011011001101111000100001100011
...
```

Running the testbench repeatedly emits the warning: `VCD warning: array word InstMem_Tb.m.mem[0] will conflict with an escaped identifier.` . It can be suppressed as explained in `https://stackoverflow.com/questions/20317820/`. We decided to keep the code simpler at the cost of this warning.

Data Memory

`wr_strb` is encoded as following:

```
input wire [2:0] wr_strb;
// wr_strb = 000 -> store word
// wr_strb = 001 -> store lower halfword
// wr_strb = 010 -> nop
// wr_strb = 011 -> store higher halfword
// wr_strb = 100 -> store lowest byte
// wr_strb = 101 -> store 2nd lowest byte
// wr_strb = 110 -> store 3rd lowest byte
// wr_strb = 111 -> store highest byte
```

The state 010 is chosen to be nop to simplify the hardware: the msb bit controls the byte-level stores; the lsb bit controls the halfword-level stores and the combination 000 is for word-level stores. The enum-like encoding of wr_strb allows us increment it in the testbench as if it was an integer. In the testbench, a value of 0xAABBCCDD is placed in the addresses starting from 0, but each time with a new wr_strb mode. The values are stored in little endian order. In the testbench we've redefined DEPTH=8 so that the console output is more consice, however it can easily be changed to its original value of 128. The end result is as following:

```
...
Address: 0 Data_Hex: aabbccdd Data_Bin: 10101010101110111100110011011101
Address: 1 Data_Hex: 0000ccdd Data_Bin: 00000000000000001100110011011101
Address: 2 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 3 Data_Hex: ccdd0000 Data_Bin: 11001100110111010000000000000000
Address: 4 Data_Hex: 000000dd Data_Bin: 00000000000000000000000011011101
Address: 5 Data_Hex: 0000dd00 Data_Bin: 00000000000000001101110100000000
Address: 6 Data_Hex: 00dd0000 Data_Bin: 00000000110111010000000000000000
Address: 7 Data_Hex: dd000000 Data_Bin: 11011101000000000000000000000000
```