Lab 2 Report

CENG3420

Lam Kin Ho

1155158095

6 Apr, 2022

Lab 2.1

I-type

For lab 2.1, just starting with understanding the example code.

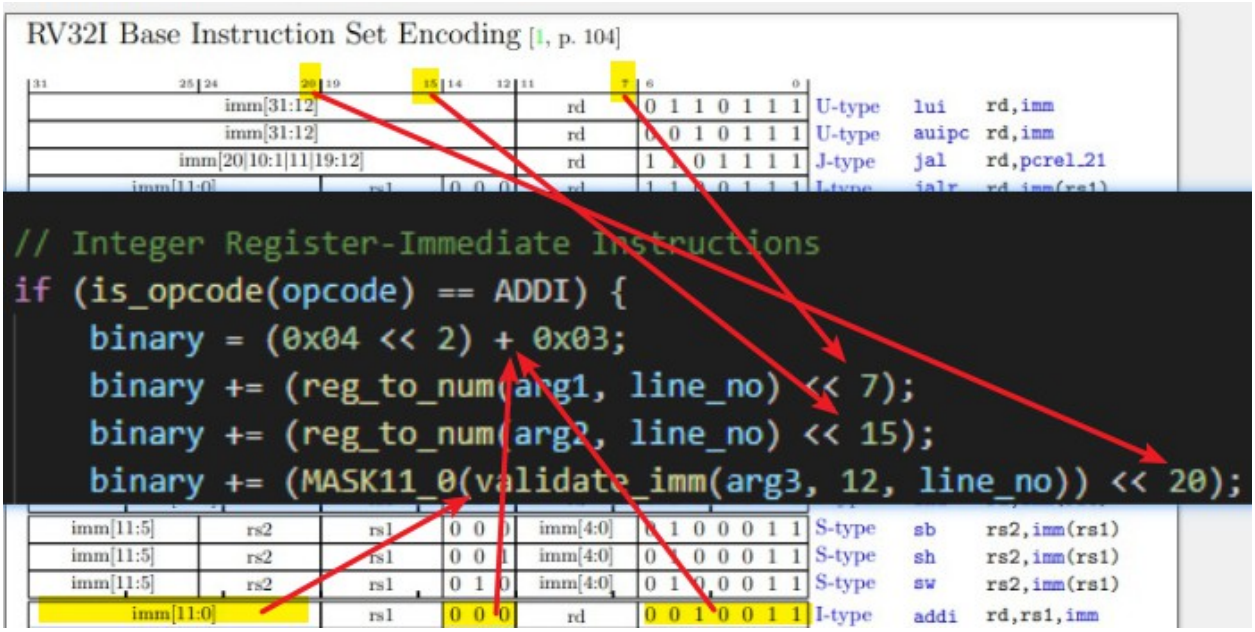
```
    if (is_opcode(opcode) == ADDI) {
    binary = (0x04 << 2) + 0x03;
    binary += (reg_to_num(arg1, line_no) << 7);
    binary += (reg_to_num(arg2, line_no) << 15);
    binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20);
}
```

Actually, we can see it is not hard to understand. The first line is just the **optcode** part. we can observe that 0x04 << 2 + 0x03 = 0001 0011. And the second line and third line are doing same job. It would first call reg_to_num function to convert the register(like t0, a1 etc.) to the 5-bit encoding (rx). Follow up by shifting 7 or 15 bit. The shift operation is set the rd and rs1 into suitable bits [7,11] or [15,19]. The final line is check the last parameter, the immediate, then take the lowest 12bit with MASK11_0 Marco, afterward shift to left by 20bit into [31,20].

And the following job will be fairly easy, just refer to the encoding and change the parameters, which are registers, imm, optcode, function code. All the other command, I just refer to the chart. We might take the example code to visualize once.

RV32I Base Instruction Set Encoding [1, p. 104]

31 25	24 20	19 15	14 12	11 7	6 0			
imm[31:12]				rd	0 1 1 0 1 1 1	U-type	lui	rd,imm
imm[31:12]				rd	0 0 1 0 1 1 1	U-type	auipc	rd,imm
imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1	J-type	jal	rd,pcrel_21
imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1	I-type	jalr	rd,imm(rs1)
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	beq	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bne	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	blt	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bge	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bltu	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bgeu	rs1,rs2,pcrel_13
imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1	I-type	lb	rd,imm(rs1)
imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1	I-type	lh	rd,imm(rs1)
imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1	I-type	lw	rd,imm(rs1)
imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1	I-type	lbu	rd,imm(rs1)
imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1	I-type	lhu	rd,imm(rs1)
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sb	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	S-type	sh	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sw	rs2,imm(rs1)
imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1	I-type	addi	rd,rs1,imm
imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1	I-type	slti	rd,rs1,imm
imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1	I-type	sltiu	rd,rs1,imm
imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1	I-type	xori	rd,rs1,imm
imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1	I-type	ori	rd,rs1,imm
imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1	I-type	andi	rd,rs1,imm
0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	I-type	slli	rd,rs1,shamt
0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli	rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai	rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	add	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	sub	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type	sll	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type	slt	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type	sltu	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type	xor	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	srl	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	sra	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type	or	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type	and	rd,rs1,rs2
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1			ecall	
0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1			ebreak	
csr[11:0]		rs1	0 0 1	rd	1 1 1 0 0 1 1	I-type	csrrw	rd,csr,rs1
csr[11:0]		rs1	0 1 0	rd	1 1 1 0 0 1 1	I-type	csrrs	rd,csr,rs1
csr[11:0]		rs1	0 1 1	rd	1 1 1 0 0 1 1	I-type	csrrc	rd,csr,rs1
csr[11:0]		zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1	I-type	csrrwi	rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1	I-type	csrrsi	rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1	I-type	csrrci	rd,csr,zimm



For the other instructions, the first line of other instructions will also include function code calculation. (which is 0 in the case addi, so there is no need in example code).

For slli, srli, and srai,
Only the lowest 5 bit of imm will be used as shamt(shift at most 32 bits). I get this with simple binary and with 11111(the prefix 0b meaning binary format). Also, when the 30th bit is 1, it will be srai, when it is 0, it will be srli. I also done this job in first line.

For lui,
It is quite different with other instruction, instead of 3, it only take 2 parameter. Just the rd and imm(or label). To complete this, first we setup the optcode with 0b110111. then set the rd like other i-type instructions. Finally, call handle_label_or_imm function, following by binary and with 0b11111111111111111111000000000000 (0xFFFFF000 also works) to get the [31:12] part and shift to corresponding bits.

R-type

We can observe that it is similar job with i-type, just instead of worrying imm[11:0], we now need to code the third register into 5-bit (rx) in the last line of each instructions. The optcode is changed as well.

```
    else if (is_opcode(opcode) == ADD) {
    binary = (0x0C << 2) + 0x03;
    binary += (reg_to_num(arg1, line_no) << 7);
    binary += (reg_to_num(arg2, line_no) << 15);
    binary += (reg_to_num(arg3, line_no) << 20);
}
```

For sub and sra, we can see the 30bit is used(like srai, srli) to distinguish from and(this and refer to instruction) and srl. I also do this job in first line in corresponding instruction.

J-type

Now, we need to handle J-type, it is a bit more complicate than previous. Starting with the easier one, jalr, it actually encoding like i-type, only the optcode is different, so simply replace first line with suitable optcode.

The harder one, jal, need more work to encode. The first two line is trivial, just the optcode and rd. However, the imm part is divided into four part and we have to rearrange it. We first call handle_label_or_imm function to get whole imm, then binary and operation to get corresponding part, [19:12] [11] [10:1] [20] (I also comment in asm.c), Finally shift to right place, it can be calcuated by -imm[lower bit] + 12 + bit before it. Take imm[10:1] as example, 1 is lower bit, and imm[19:12](8bits) + imm[11](1bits) are before it. so it should be shift, -1+12+(8+1)=20 bits. After we shift the imm correctly, we are done for j-type.

B-type

Now we have to deal with b-type, the optcode, rs1 , rs2, func is trival. Only part we have to consider is imm. We first call label_to_num function to get the imm, then divide it and shift to right bits. all the technique for imm are already introduced in j-type. So it is not a tough job.

S-type

Finally, we work with s-type,
Again, lb, lh, lw just like i-type, so we should first change the optcode and func. Then we need to call parse_regs_indirect_addr function, it help us to chop down imm(rs1) into ret-¿imm and ret-¿rs1. Then the subsequence job is same with i-type, just imm and rs1 will be replaced.
Instead of i-type, sb, sh, sw are more like B-type, we just need to set optcode, func, rs2 right and like load-instructions, call parse_regs_indirect_addr function to get rs1 and imm, and chop down imm and shift to correct bits, then job are done.

Lab2.2

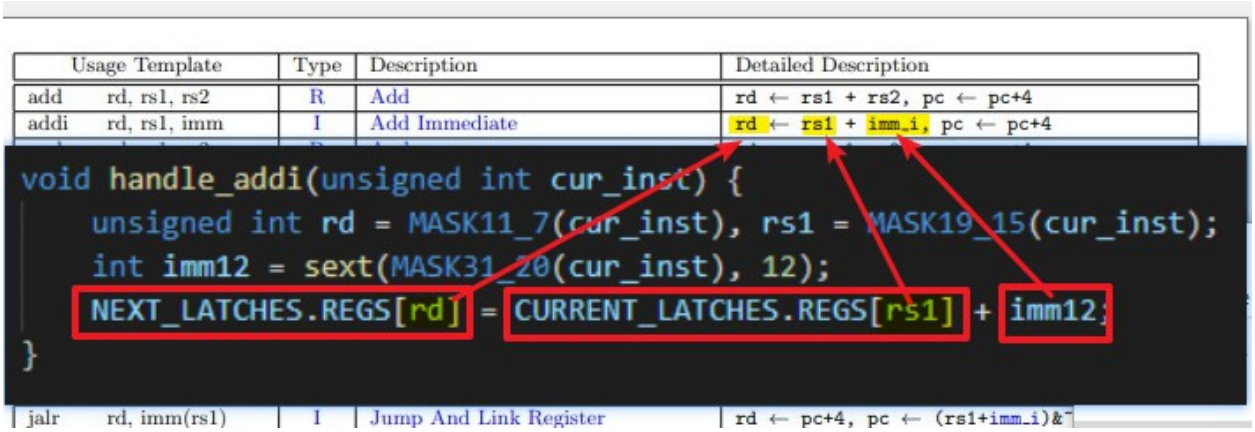
I-type

For lab 2.2, it is a bit complicate, but we can sort it out by observing and understanding the given code.

```
    void handle_addi(unsigned int cur_inst) {
        unsigned int rd = MASK11_7(cur_inst), rs1 = MASK19_15(cur_inst);
        int imm12 = sext(MASK31_20(cur_inst), 12);
        NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.REGS[rs1] + imm12;
}
```

The first line is get the 5 bit register code (rx) from cur_inst with the aid of MASK11_7 and MASK19_15. The second line fetch the imm first and then sign-extend it by calling sext function. The final line update the register(rd) with current register(rs1) plus imm. We can decode I-type like this, the optcode and func are handled in another function, so we need to consider rs1 rd and imm. Take addi as example, We can see that it is referring to these part (see the figure). So we just follow the chart to change the operator to +,-,<<,>>,^ etc. then jobs are done.

Usage Template	Type	Description	Detailed Description
add rd, rs1, rs2	R	Add	rd ← rs1 + rs2, pc ← pc+4
addi rd, rs1, imm	I	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4
and rd, rs1, rs2	R	And	rd ← rs1 & rs2, pc ← pc+4
andi rd, rs1, imm	I	And Immediate	rd ← rs1 & imm_i, pc ← pc+4
auipc rd, imm	U	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4
beq rs1, rs2, pcrel_13	B	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)
bge rs1, rs2, pcrel_13	B	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
bgeu rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
blt rs1, rs2, pcrel_13	B	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bltu rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bne rs1, rs2, pcrel_13	B	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)
jal rd, pcrel_21	J	Jump And Link	rd ← pc+4, pc ← pc+imm_j
jalr rd, imm(rs1)	I	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i)&~1
lb rd, imm(rs1)	I	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4
lbu rd, imm(rs1)	I	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4
lh rd, imm(rs1)	I	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4
lhu rd, imm(rs1)	I	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4
lui rd, imm	U	Load Upper Immediate	rd ← imm_u, pc ← pc+4
lw rd, imm(rs1)	I	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4
or rd, rs1, rs2	R	Or	rd ← rs1 | rs2, pc ← pc+4
ori rd, rs1, imm	I	Or Immediate	rd ← rs1 | imm_i, pc ← pc+4
sb rs2, imm(rs1)	S	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4
sh rs2, imm(rs1)	S	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4
sll rd, rs1, rs2	R	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4
slli rd, rs1, shamt	I	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4
slt rd, rs1, rs2	R	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
slti rd, rs1, imm	I	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltiu rd, rs1, imm	I	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltu rd, rs1, rs2	R	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
sra rd, rs1, rs2	R	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srai rd, rs1, shamt	I	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
srl rd, rs1, rs2	R	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srli rd, rs1, shamt	I	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
sub rd, rs1, rs2	R	Subtract	rd ← rs1 - rs2, pc ← pc+4
sw rs2, imm(rs1)	S	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4
xor rd, rs1, rs2	R	Exclusive Or	rd ← rs1 ^ rs2, pc ← pc+4
xori rd, rs1, imm	I	Exclusive Or Immediate	rd ← rs1 ^ imm_i, pc ← pc+4



For slli, srli, srai,

These three doesnt take imm[11:0] like other i-type, it rather take shamt_i which only 5 bit. So just call MASK24_20 MACRO to extract the corresponding part. Also, the 30th bit used to distinguish srli and srai are also handled by the other function.

The << operator behave the same no matter arithmetic or logically so we don't need to worry about it. While the >> do have the difference, for signed-int it does arithmetic shift, unsigned-int it does logically shift. (Compiler dependent, while gcc, which I'm using, behave like this) So just convert the int into unsigned int in srli, then everything work fine.

For lui,
it is even more easier, we only need to take rd and imm[31:12], and update register[rd] to imm[31:12].

R-type

R-type, this type is similar to i-type, just in stead of taking imm[11:0], it take rs2. And undate the register[rd] to register[rs1] (operand) register[rs2].
Like i-type, the handle of 30th bit of sra and srl are done by another function. Also convert rs1 into unsigned int in srl.

J-type

J-type, or jalr, we can see it can be decoded like i-type so we dont need to worry this part. Take a look to the chart, we have to update rd to pc+4, and pc to (rs1+imm_i) 1. It can be done with these two line of code

```
NEXT_LATCHES.PC = CURRENT_LATCHES.REGS[rs1] + imm12;
NEXT_LATCHES.REGS[rd] = CURRENT_LATCHES.PC + 4;
```

Which is very trivial, however I didn't handle the **1**, what it mean is set the LSB of pc to 0 because pc will never ever point to odd address. If desired, can handle by adding

```
NEXT_LATCHES.PC = NEXT_LATCHES.PC & 0xFFFFFFFE;
```

B-type

For B-type, rs1, rs2, can be easily handled like any other instructions done before. The tough part is imm, as it is divided into four part. Fortunately, the example code actually done the job for us. So the actually work we have to do is just copying the code and change the comparison operator in the if-statement to <, !=, etc.

S-type

S-type is the most difficult part in this lab. rd,rs1,imm12 is trivial but the tough part is reading mem, as each memory is 1byte only, so for lb, lh, lw, we need to read 1mem, 2mem, 4mem respectively. Rather than bother with reading MEMORY[base + 1or2or3], I just call the read_mem function, it will return a int (4Byte). So for lw, work are done, for lh, just take the lower 2Byte by 0xFFFF and sign-ext it. While for sb, sh, sw, we have to do it our self.

Take sw code

```
unsigned int rs1 = MASK19_15(cur_inst), rs2 = MASK24_20(cur_inst);
int imm_s = (sext(MASK31_25(cur_inst), 7) << 5 ) + MASK11_7(cur_inst);
MEMORY[CURRENT_LATCHES.REGS[rs1]+imm_s] = MASK7_0(CURRENT_LATCHES.REGS[rs2]);
MEMORY[CURRENT_LATCHES.REGS[rs1]+imm_s+1] = MASK15_8(CURRENT_LATCHES.REGS[rs2]);
MEMORY[CURRENT_LATCHES.REGS[rs1]+imm_s+2] = MASK23_16(CURRENT_LATCHES.REGS[rs2]);
MEMORY[CURRENT_LATCHES.REGS[rs1]+imm_s+3] = MASK31_24(CURRENT_LATCHES.REGS[rs2]);
```

we can note that, we write to MEM[CURRENT_LATCHES.REGS[rs1]+imm+i], each summach are referring to

CURRENT_LATCHES.REGS[rs1]	base address
imm_s	offset
i	1st 2nd 3rd 4th MEM

opcode & funct3

Finally we handle the opcode and funct3. We first detect the opcode then then funct3. when dueling with

srai	srli
add	sub
sra	srl

After funct3, we have moreover to check the 30th bit to check which instruction it refer to. Instead of only checking 30th bit, we get 25th bit to 31st bit and compare to zero. All the other instruction are done similarly, however I stick with binary format rather than hex so I don't need to bother the conversion.