CENG3420 Lab2.1 Report

Name: Ng Ethan SID:1155212674

Lab2.1

Step-by-step algorithm:

I-type:

First of all, we need to understand the example code.

```
if (is opcode(opcode) == ADDI) {
  binary = (0x04 << 2) + 0x03; //opcode
  binary += (reg_to_num(arg1, line_no) << 7); //rd
  binary += (reg_to_num(arg2, line_no) << 15); //rs1
  binary += (MASK11_0(validate_imm(arg3, 12, line_no)) << 20); //immediate num
}
```

In here, we could see that we are trying to change the assembley code to RV321 assembler. For the first line, it will equal to a 7bit opcode number. For the second line, it will equal to a 5-bit rd register. For the third line, it will equal to a 5-bit rs1 register and for the fourth line, it will equal to a 12bit immediate value. As a result, we could see that it actually same as the RV321 format below:

	31	27	26	25	24	20	19	15	14	12	11	7	6	0	
		func	:t7		rs	$^{\circ}2$	r	s1	fun	ct3		rd	opco	de	R-ty
≶	•	ir	nm[]	11:0)]		r	s1	fun	ct3		rd	opco	de	I-typ
ſ	ir	nm[1	1:5]		rs	\mathbf{s}^2	r	s1	fun	ct3	imn	n[4:0]	opco	de	S-typ
	imı	n[12	10:5	5]	rs	$^{\circ}2$	r	s1	fun	ct3	imm[4:1 11]	opco	de	B-typ
					im	m[31	:12]					rd	opco	de	U-tyj
ſ				imn	n[20	10:1	11 19:	12]				rd	opco	de	J-typ
_															

ре pe рe уpе pe

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0×20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0×00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0×3	0×00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
1b	Load Byte	I	0000011	0×0		rd - M[rs1+imm][0:7]	
1h	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
1bu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
Ibu	Load Byte (U)	1 1					
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
		_		0x5 0x0		rd = M[rs1+imm][0:15] M[rs1+imm][0:7] = rs2[0:7]	zero-extends
1hu	Load Half (U)	I	0000011				zero-extends
lhu sb	Load Half (U) Store Byte	I S	0000011 0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	zero-extends
lhu sb sh	Load Half (U) Store Byte Store Half	I S S	0000011 0100011 0100011	0x0 0x1		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15]	zero-extends
lhu sb sh sw	Load Half (U) Store Byte Store Half Store Word	S S S	0000011 0100011 0100011 0100011	0x0 0x1 0x2		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31]	zero-extends
1hu sb sh sw beq	Load Half (U) Store Byte Store Half Store Word Branch ==	S S S	0000011 0100011 0100011 0100011	0x0 0x1 0x2 0x0		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm	zero-extends
sb sh sw beq bne	Load Half (U) Store Byte Store Half Store Word Branch == Branch !=	S S S B	0000011 0100011 0100011 0100011 1100011	0x0 0x1 0x2 0x0 0x1		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm	zero-extends
sb sh sw beq bne blt	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch <	S S S B B	0000011 0100011 0100011 0100011 1100011 1100011	0x0 0x1 0x2 0x0 0x1 0x4		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 == rs2) PC += imm if(rs1 <= rs2) PC += imm	zero-extends
sb sh sw beq bne blt bge	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch < Branch <> Branch <> Branch <	S S S B B B	0000011 0100011 0100011 0100011 1100011 1100011 1100011	0x0 0x1 0x2 0x0 0x1 0x4 0x5		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm if(rs1 >= rs2) PC += imm	
sb sh sw beq bne blt bge bltu	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch < Branch ==	S S S B B B	0000011 0100011 0100011 0100011 1100011 1100011 1100011 1100011	0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 < rs2) PC += imm	zero-extends
sb sh sw beq bne blt bge bltu bgeu	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch <> Branch < Branch <> Branch <> U) Branch < (U) Branch > (U)	S S S B B B B	0000011 0100011 0100011 0100011 1100011 1100011 1100011 1100011 1100011	0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm if(rs1 >= rs2) PC += imm	zero-extends
sb sh sw beq bne blt bge bltu bgeu jal	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch < Branch < CU) Branch ≥ (U) Jump And Link Jump And Link Reg	S S S B B B B B B B B B B B B B B B B B	0000011 0100011 0100011 1100011 1100011 1100011 1100011 1100011 1100011 1101111	0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm rd = PC+4; PC += imm	zero-extends
sb sh sw beq bne blt bge bltu bgeu jal jalr lui	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch << Branch <> Branch U) Branch < (U) Jump And Link Jump And Link Reg Load Upper Imm</td <td>I S S S B B B B B J I</td> <td>0000011 0100011 0100011 1100011 1100011 1100011 1100011 1100011 1100011 1101111 1100111</td> <td>0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7</td> <td></td> <td>M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm rd = PC+4; PC += imm rd = PC+4; PC = rs1 + imm rd = mm << 12</td> <td>zero-extends</td>	I S S S B B B B B J I	0000011 0100011 0100011 1100011 1100011 1100011 1100011 1100011 1100011 1101111 1100111	0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm rd = PC+4; PC += imm rd = PC+4; PC = rs1 + imm rd = mm << 12	zero-extends
sb sh sw beq bne blt bge bltu bgeu jal jalr	Load Half (U) Store Byte Store Half Store Word Branch == Branch != Branch < Branch < CU) Branch ≥ (U) Jump And Link Jump And Link Reg	S S S B B B B B B B B B B B B B B B B B	0000011 0100011 0100011 1100011 1100011 1100011 1100011 1100011 1100011 1101111	0x0 0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7	imm=0x0	M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm if(rs1 >= rs2) PC += imm if(rs1 >= rs2) PC += imm if(rs1 >= rs2) PC += imm rd = PC+4; PC += imm	zero-extends

Where (0x04<<2)+0x03 = 0010011, << is the instucrions to shift left by n unit. So we could following the above table to finish out code, by changing the all the instructions into binary number and convert it to RV321.

```
if (is_opcode(opcode) == ADDI) {
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += (reg_to_num(arg2, line_no) << 7); //rs1
    binary += (MASK11_0(validate_imm(arg3, width: 12, line_no)) << 20); //immediate num
} else if (is_opcode(opcode) == SLLI) {
    /* Lab2-1 assignment */
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x1<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (validate_imm(arg3, width: 12, line_no)&8x1F) << 20; //immediate num
    //warn("Lab2-1 assignment: SLLI instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == XORI) {
    /* Lab2-1 assignment */
    binary += (0x04 << 2) + 0x03; //opcode
    binary += (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += (x4SK11_0(validate_imm(arg3, width: 12, line_no)) << 20); //immediate num
    // warn("Lab2-1 assignment: XORI instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SRLI) {
    /*
         * Lab2-1 assignment
         * tip: you may need the function `lowerSbit`
         */
         binary += (0x04 << 2) + 0x03; //opcode
         binary += (0x04 <
```

```
} else if (is_opcode(opcode) == SRAI) {
    /*
        * Lab2-1 assignment
        * tip: you may need the function `lowerSbit`
        */
        binary = (0x94 << 2) + 0x03; //opcode
        binary += (reg_to_num(arg1, line_no) << 7); //rd
        binary += (reg_to_num(arg2, line_no) << 15); //rs1
        binary += (validate_imm(arg3, width: 12, line_no)&0x1F) << 20 ;//immediate num
        binary += (0x20e<25); //funct7
        // warn("Lab2-1 assignment: SRAI instruction\n");
        //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == DRI) {
            /* Lab2-1 assignment */
            binary += (reg_to_num(arg1, line_no) << 7); //rd
            binary += (reg_to_num(arg1, line_no) << 7); //rd
            binary += (MASK11_0(validate_imm(arg3, width: 12, line_no)) << 20); //immediate num
            // warn("Lab2-1 assignment: ORI instruction\n");
            //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == ANDI) {
                 /* kab2-1 assignment */
            binary += (0x94 << 2) + 0x03; //opcode
            binary += (0x94 << 2) + 0x03; //opcode
            binary += (0x94 << 2) + 0x03; //opcode
            binary += (reg_to_num(arg1, line_no) << 7); //rd
            binary += (reg_to_num(arg1, line_no) << 7); //rd
            binary += (reg_to_num(arg1, line_no) << 7); //rd
            binary += (reg_to_num(arg2, line_no) << 15); //rs1
            binary += (reg_t
```

Here are the code in I-type, each of the line have specify which instuctions is it, while other code will add a funct3 binary number for following the RV321 format.

In addition, for slli, srli and srai, since we only need the lowest 5 bit of imm, so I have convert it with &0x1F for choosing the lowest 5bit only, and in imm[11:5], we just have to follow the table(which has been specify in row funct7) to change the value.

R-type:

As what we have done before, we first look for the example code:

```
else if (is_opcode(opcode) == ADD) {
    binary = (0x0C << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += (reg_to_num(arg2, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (0x0 << 25);//funct7
}
```

In here, we could see that we are trying to change the assembley code to RV321 assembler. For the first line, it will equal to a 7bit **opcode** number. For the second line, it will equal to a 5-bit **rd** register. For the third line, it will equal to a 5-bit **rs1** register. For the fourth line, it will equal to a 5bit **rs2** register and for the fifth line, it will equal to a 5bit **funct7** number. As a result, we could see that it actually same as the RV321 format below:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
$ \longrightarrow $	func	ct7		rs	²	rs1		fun	ct3		rd	opcode	
	imm[11:0]					rs1	fun	funct3		rd	opcode		
	imm[:	l1:5]		rs	2	rs1	-	fun	ct3	im	m[4:0]	opcode	
in	nm[12	10:5	5]	rs	52	rs1		fun	ct3	imm	[4:1 11]	opcode	
				im	m[31	:12]					rd	opcode	
			imn	ո[20	10:1	11 19:1	2]				rd	opcode	

R-type I-type S-type B-type U-type J-type

RV32I Base Integer Instructions

Inet	Name	EMT	Opeodo	funct?	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0×00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
s11	Shift Left Logical	R	0110011	0x1	0×00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0×00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0×00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
1b	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
1h	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
1bu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
1hu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
SW	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	В	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	В	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	В	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch 🦓 >=	В	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	В	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch \geq (U)	В	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0×0	imm=0×0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Where (0x0C << 2)+0x03 = 0110011, << is the instuctions to shift left by n unit. So we could following the above table to finish out code, by changing the all the instructions into binary number and convert it to RV321.

```
pelse if (is_opcode(opcode) == ADD) {
    binary = (0x00 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (reg_to_num(arg3, line_no) << 28); //rs2
    binary += (0x0 << 25);

    else if (is_opcode(opcode) == SUB) {
        /* Lab2-1 assignment */
        binary += (0x0 << 2) + 0x03; //opcode
        binary += (reg_to_num(arg2, line_no) << 7); //rd
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (reg_to_num(arg3, line_no) << 29); //rs2
        binary += (reg_to_num(arg3, line_no) << 29); //rs2
        binary += (0x00 << 25); //funct7
        // warn(*Lab2-1 assignment: SUB instruction\n*);
        //exit(EXIT_FAILURE);
    } else if (is_opcode(opcode) == SLL) {
        /* Lab2-1 assignment */
        binary += (0x00 << 2) + 0x03; //opcode
        binary += (reg_to_num(arg1, line_no) << 7); //rd
        binary += (reg_to_num(arg2, line_no) << 15); //rs1
        binary += (reg_to_num(arg2, line_no) << 20); //rs2
        binary += (reg_to_num(arg3, line_no) << 20); //rs2
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (0x0 << 2) + 0x03; //opcode
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (0x0 << 2) + 0x03; //opcode
        binary += (reg_to_num(arg3, line_no) << 7); //rd
        binary += (reg_to_num(arg3, line_no)
```

```
} else if (is_opcode(opcode) == SRL) {
    /* Lab2-1 assignment */
    binary == (0x80 < 2) + 0x83;//opcode
    binary += (reg_to_num(argl, line_no) << 7);//rd
    binary += (reg_to_num(argl, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (0x80 << 25);//funct7
    //marn("Lab2-1 assignment: SRL instruction\n");
//exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SRA) {
    /* Lab2-1 assignment */
    binary += (0x80 << 2) + 0x83;//opcode
    binary += (0x80 << 2) + 0x83;//opcode
    binary += (0x80 << 2) + 0x83;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    binary += (reg_to_num(arg3, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);///rs2
    binary += (reg_to_num(arg3, line_no) << 20);///rs2
    binary += (0x80 << 25);//funct7
    //marn("tab2-1 assignment: SRA instruction\n");
//exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == OR) {
    /* Lab2-1 assignment */
    binary += (0x80 << 2) + 0x803;
    binary += (reg_to_num(arg2, line_no) << 7);
    binary += (reg_to_num(arg2, line_no) << 15);
    binary += (reg_to_num(arg3, line_no) << 20);
    //warn("tab2-1 assignment: OR instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == AND) {
    /* Lab2-1 assignment */
    binary += (0x80 << 2) + 0x803;
    binary += (0x80 << 2) + 0x803;
```

```
} else if (is_opcode(opcode) == AND) {
    /* Lab2-1 assignment */
    binary = (0x00 << 2) + 0x03;
    binary += (reg_to_num(arg1, line_no) << 7);
    binary += 0x7 <<12;
    binary += (reg_to_num(arg2, line_no) << 15);
    binary += (reg_to_num(arg3, line_no) << 20);
    binary += (0x0 << 25);
    //warn("lab2-1 assignment: AND instruction\n");
    //exit(EXIT_FAILURE);
}</pre>
```

Here are the code in I-type, each of the line have specify which instuctions is it, while other code will add a funct3 binary number for following the RV321 format.

In addition, for slli, srli and srai, since we only need the lowest 5 bit of imm, so I have convert it with &0x1F for choosing the lowest 5bit only, and in imm[11:5], we just have to follow the table(which has been specify in row funct7) to change the value.

J-type and JAL

First of all, we look at the RV321 J-type format:

	31	27	26	25	24	20	19	15	14	12	11	7	6		0
		func	t7		rs	2				ct3	1	rd	opcode		
_	1	ir	nm[11:0)]		rs	1	fun	ct3	rd		ope	code	
	in	nm[1	1:5]		rs	2	rs	1	fun	ct3	imn	1[4:0]	ope	code	
	imı	n[12	10:5	5]	rs	2	rs	1	fun	ct3	imm[4:1 11]	ope	code	
						m[31						rd	ope	code	
imm[20 10:1 11 19:12]										1	rd	ope	code		

R-type I-type S-type B-type U-type J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0×00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
1h	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
1bu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
1hu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
SW	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	В	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	В	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	В	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch 🦥 >=	В	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	В	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch > (U)	В	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	4
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
IUI	Load Opper Imm	U	וווטווט			ra = 1mm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Since we notice that one of them is I-type and one of them is J-type, we just have to following to the instruction to finished the code.

Here is the code of J-type. Since in jal, there will be a label (the function name), so we need to change it to immediate number by using handle_label_or_imm.

B-type

As what we have done before, we first look for the example code:

```
else if (is_opcode(opcode) == BEQ) {
  binary = (0x18 << 2) + 0x03;//opcode
  binary += (reg_to_num(arg1, line_no) << 15);//rs1
  binary += (reg_to_num(arg2, line_no) << 20);//rs2
  int val = label_to_num(
    line no, arg3, 12, label table, number of labels
  ),offset = val - addr;//change label to num
  // imm[11]
  binary += ((offset & 0x800) >> 4);
  // imm[4:1]
  binary += ((offset & 0x1E) << 7);
  // imm[10:5]
  binary += ((offset & 0x7E0) << 20);
  // imm[12]
  binary += ((offset & 0x1000) << 19);
}
```

In here, we could see that we are trying to change the assembley code to RV321 assembler. For the first line, it will equal to a 7bit **opcode** number. For the second line, it will equal to a 5-bit **rs1** register. For the third line, it will equal to a 5-bit **rs2** register. For the remaining line, it will equal to a 12bit **immedaite value.** As a result, we could see that it actually same as the RV321 format below:

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
Γ		func	:t7		rs	2	rs.	fun	ct3		rd	opc	ode	
		ir	nm[î	11:0)]		rs.	funct3		rd		opc	ode	
Γ		nm[1	_		rs	2	rs.	fun	ct3	imn	n[4:0]	opc	ode	
¥	imr	m[12	10:5	5]	rs	2	rs.	fun	ct3	imm[4:1 11]	opc	ode	
Γ					im	m[31	:12]					rd	opc	ode
imm[20 10:1 11 19:12]												rd	opc	ode

R-type I-type S-type B-type U-type J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
1h	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
1bu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
SW	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	В	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	В	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	В	1100011	0x4		if(rs1 < rs2) PC += imm	4
bge	Branch 🦓 >=	В	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	В	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch \geq (U)	В	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Where (0x18 << 2)+0x03 = 1100011, << is the instuctions to shift left by n unit. So we could following the above table to finish out code, by changing the all the instructions into binary number and convert it to RV321.

```
binary += (reg_to_num(arg2, line_no) << 20);//rs2
  binary += ((offset & 0x800) >> 4);
  binary += ((offset & 0x7E0) << 20);
binary += (reg_to_num(arg2, line_no) << 20);//rs2
int val = label_to_num(</pre>
line_no, arg3, width 12, label_table, number_of_labels
),offset = val - addr;
binary = (0x18 << 2) + 0x03;//opcode
binary += 0x5 << 12;//funct3
\label{eq:binary += (reg_to_num(arg1, line_no) << 15);//rs1 \\ \binary += (reg_to_num(arg2, line_no) << 20);//rs2
```

Here is the code in B-type, each of the line have specify which instuctions is it, while other code will add a funct3 binary number for following the RV321 format. Also, since there may have a label in B-type, so we have use 'label_to_num' to help us change the label to a number.

S-type , LB, LH and LW

As what we have done before, we first look for the example code:

```
else if (is_opcode(opcode) == LB) {
  binary = 0x03;//opcode
  binary += (reg_to_num(arg1, line_no) << 7);//rd
  struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
  binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
  binary += (ret->imm << 20);//imm
}</pre>
```

This is for the example of I-type,In here, we could see that we are trying to change the assembley code to RV321 assembler. For the first line, it will equal to a 7bit **opcode** number. For the second line, it will equal to a 5-bit **rd** register. For the third line, it is going to sperate the register and immediate value. For the fourth line, it will equal to a 5bit **rs1** register and for the last is equal to a 12bit **immediate value.** As a result, we could see that it actually same as the RV321 format below:

	31	27	26	25	24	20	19	15	14	12	11	7	6	0	
		func	t7		rs	2	rs1	L	fun	ct3	1	rd	opco	de	R-
-		in	nm[:	11:0)]		rs1	L	fun	ct3	1	rd	opco	de	I-1
	in	nm[1	1:5]		rs	2	rs]	L	fun	ct3	imm	[4:0]	opco	de	S-
	imı	n[12	10:5	5]	rs	2	rs]	L	fun	ct3	imm[4	4:1 11]	opco	de	B-
						m[31	_				1	:d	opco	de	U-
			:	imn	ı[20	10:1	11 19:1	2]			1	:d	opco	de	J-

R-type I-type S-type B-type U-type J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0×0	0×20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0×00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0×7	0×00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0×00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0×00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0×2	0×00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0×00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0×4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0×7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0×5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
1b	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
1h	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
1w	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
1bu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
1hu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb							
	Store Byte	S	0100011	0×0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Byte Store Half	S S	0100011 0100011	0x0 0x1		M[rs1+imm][0:7] = rs2[0:7] M[rs1+imm][0:15] = rs2[0:15]	
sh sw							
	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
SW	Store Half Store Word	S S	0100011 0100011	0x1 0x2		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31]	
sw beq	Store Half Store Word Branch ==	S S	0100011 0100011 1100011	0x1 0x2 0x0		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm	
sw beq bne	Store Half Store Word Branch == Branch !=	S S B	0100011 0100011 1100011 1100011	0x1 0x2 0x0 0x1		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm	
sw beq bne blt	Store Half Store Word Branch == Branch != Branch <	S S B B	0100011 0100011 1100011 1100011 1100011	0x1 0x2 0x0 0x1 0x4		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 < rs2) PC += imm	zero-extends
sw beq bne blt bge	Store Half Store Word Branch == Branch != Branch < Branch =	S S B B	0100011 0100011 1100011 1100011 1100011	0x1 0x2 0x0 0x1 0x4 0x5		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 >= rs2) PC += imm	zero-extends zero-extends
sw beq bne blt bge bltu	Store Half Store Word Branch == Branch != Branch < Branch < Branch < Color of the	S S B B B	0100011 0100011 1100011 1100011 1100011 1100011	0x1 0x2 0x0 0x1 0x4 0x5 0x6		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] if(rs1 = rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 > rs2) PC += imm if(rs1 >= rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 < rs2) PC += imm if(rs1 > rs2) PC += imm	
sw beq bne blt bge bltu bgeu	Store Half Store Word Branch == Branch != Branch < Branch < Branch < U) Branch < (U) Branch \geq (U)	S S B B B B	0100011 0100011 1100011 1100011 1100011 1100011 1100011	0x1 0x2 0x0 0x1 0x4 0x5 0x6		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] If(rs1 == rs2) PC += imm if(rs1 != rs2) PC += imm if(rs1 <= rs2) PC += imm if(rs1 >= rs2) PC += imm if(rs1 <= rs2) PC += imm	
beq bne blt bge bltu bgeu jal	Store Half Store Word Branch == Branch != Branch < Branch < Branch < CU) Branch ≥ (U) Jump And Link Jump And Link Reg	S S B B B B	0100011 0100011 1100011 1100011 1100011 1100011 1100011 1100011	0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7		M[rs1+imm][0:15] = rs2[0:15] M[rs1+imm][0:31] = rs2[0:31] H[rs1=rs2] PC == inm if(rs1 = rs2) PC == inm if(rs1 = rs2) PC == inm if(rs1 > rs2) PC == inm rd = PC+4; PC += inm	
beq bne blt bge bltu bgeu jal jalr	Store Half Store Word Branch == Branch!= Branch << Branch <	S S B B B B	0100011 0100011 1100011 1100011 1100011 1100011 1100011 1101111 1100111	0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7		M[rs1+imm][e:15] = rs2[e:15] M[rs1+imm][e:31] = rs2[e:31] if(rs1 == rs2) PC == imm if(rs1 = rs2) PC == imm if(rs1 >= rs2) PC == imm if(rs1 >= rs2) PC == imm if(rs1 >= rs2) PC == imm rf(rs1 >= rs2) PC == imm rd = PC+4; PC += imm rd = PC+4; PC == s1 == imm	
beq bne blt bge bltu bgeu jal jalr	Store Half Store Word Branch == Branch != Branch < Branch < Branch < U) Branch >= U) Jump And Link Jump And Link Reg Load Upper Imm	S S S B B B B I I	0100011 0100011 1100011 1100011 1100011 1100011 1100011 1101111 1100111	0x1 0x2 0x0 0x1 0x4 0x5 0x6 0x7	imm=0x0	M[rs1+imm][6:15] = rs2[6:15] M[rs1+imm][6:31] = rs2[6:31] M[rs1+imm][6:31] = rs2[6:31] M[rs1] = rs2] PC += imm if(rs1 rs2) PC += imm if(rs1 rs2) PC += imm if(rs1 > rs2) PC += imm if(rs1 > rs2) PC += imm rd = PC:44; PC += imm rd = PC:44; PC = rs1 + imm rd = tmm << 12	

While S-type have the same doing method as B-type, so we could get a reference in there.

```
// Load and Store Instructions
else if (is_opcode(opcode) == LB) {
    binary = 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
    binary += (ret->inm << 20);//inm
} else if (is_opcode(opcode) == LH) {
    /* Lab2-1 assignment */
    binary += 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);//parse the register indirect addressing syntax
    binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
    binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
    binary += (RASK11_0(ret->inm) << 20);//imm
    //warn("Lab2-1 assignment : LH instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == LW) {
    /* Lab2-1 assignment */
    binary += 0x2 <<12;//funct3
    binary += 0x2 <<12;//funct3
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
    binary += (reg_to_num(ret->reg, line_no) << 5);//rs1
    binary += (reg_to_num(ret->reg, line_no) << 5);//rs1
    binary += (RASK11_0(ret->rem) << 20);//imm
    //warn("Lab2-1 assignment: LW instruction\n");
    //exit(EXIT_FAILURE);</pre>
```

```
| else if (is_opcode(opcode) == SB) {
    /* Lab2-1 assignment */
    binary = (Bx88<<2) +0x85;//opcode
    binary += (reg_to_num(argl, line_no) << 20);//rs2
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);//parse the register indirect addressing syntax
    binary += (reg_to_num(ret->reg, line_no) << 15);//rs1
    binary += ((ret->imm60x6TE0) << 20);//imm[4:0]
    binary += ((ret->imm60xFE0) << 20);//imm[4:0]
    binary += ((ret->imm60xFE0) << 20);//imm[11:5]
    //warn("Lab2-1 assignment */
    //exit(EXIT_FALLURE);
} else if (is_opcode(opcode) == SH) {
    /* Lab2-1 assignment */
    binary += (0x88<<2) +0x85;
    binary += (reg_to_num(argl, line_no) << 20);
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += ((ret->imm60x6TE0) << 20);
    //warn("Lab2-1 assignment: SH instruction\n");
    //exit(EXIT_FALLURE);
} else if (is_opcode(opcode) == SW) {
    /* Lab2-1 assignment */
    binary += (xeg_to_num(argl, line_no) << 20);
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += 0x2 << 12;//funct3
    binary += 0x2 << 12;//funct3
    binary += (reg_to_num(argl, line_no) << 20);
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (xeg_to_num(argl, line_no) << 15);
    binary += (reg_to_num(xeg_t, line_no) << 15);
    binary += ((ret->imm60xFE0) << 20);
    //warn("Lab2-1 assignment: SW instruction\n");
    //exit(EXIT_FALLURE);
</pre>
```

Here are the code of S-type and the loading type. Since both of them have a register indirect addressing sytanx, so we need to sperate it by using 'parse regs indirect addr' function.