CENG3420 Lab2.1 Report

Name: Ng Ethan SID:1155212674

Lab2.1

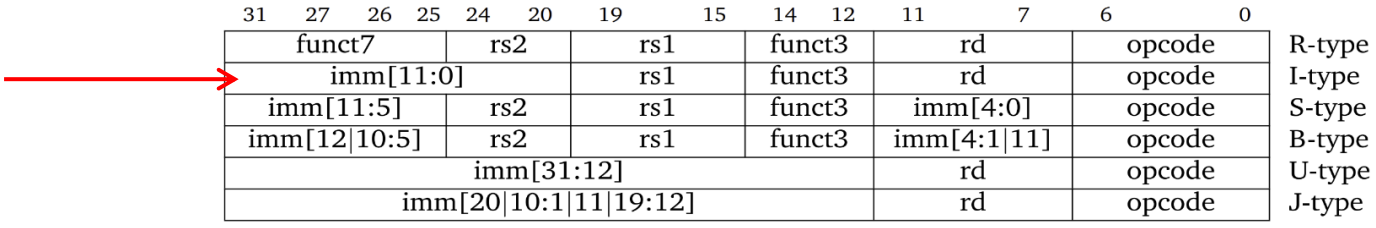
Step-by-step algorithm:

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	
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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	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]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	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 ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	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 (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) << 15);  //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)&0x1F) << 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 += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x4<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (MASK11_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 `lower5bit`
     */
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x5<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (validate_imm(arg3, width: 12,line_no)&0x1F) << 20; //immediate num
    //warn("Lab2-1 assignment: SRLI instruction\n");
```

```
} else if (is_opcode(opcode) == SRAI) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `lower5bit`
     */
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x5<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (validate_imm(arg3, width: 12,line_no)&0x1F) << 20 ;//immediate num
    binary += (0x20<<25); //funct7
   // warn("Lab2-1 assignment: SRAI instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == ORI) {
    /* Lab2-1 assignment */
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x6<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    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) {
    /* Lab2-1 assignment */
    binary = (0x04 << 2) + 0x03; //opcode
    binary += (reg_to_num(arg1, line_no) << 7); //rd
    binary += 0x7<<12; //funct3
    binary += (reg_to_num(arg2, line_no) << 15); //rs1
    binary += (MASK11_0(validate_imm(arg3, width: 12, line_no)) << 20); //immediate num
    //warn("Lab2-1 assignment: ADDI instruction\n");
    //exit(EXIT_FAILURE);
```

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	
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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	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]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	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 ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	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 (0x0C << 2)+0x03 = 0110011, << 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.

```
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);
} else if (is_opcode(opcode) == SUB) {
    /* Lab2-1 assignment */
    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 += (0x20 << 25); //funct7
   // warn("Lab2-1 assignment: SUB instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SLL) {
    /* Lab2-1 assignment */
    binary = (0x0C << 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 += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (0x0 << 25);//funct7
    //warn("Lab2-1 assignment: SLL instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == XOR) {
    /* Lab2-1 assignment */
    binary = (0x0C << 2) + 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    binary += 0x4 <<12;//funct3
    binary += (reg_to_num(arg2, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (0x0 << 25);//funct7
    //warn("Lab2-1 assignment: XOR instruction\n");
    //exit(EXIT_FAILURE);
```

```
} else if (is_opcode(opcode) == SRL) {
    /* Lab2-1 assignment */
    binary = (0x0C << 2) + 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    binary += 0x5 <<12;//funct3
    binary += (reg_to_num(arg2, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);//rs2
    binary += (0x0 << 25);//funct7
    //warn("Lab2-1 assignment: SRL instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SRA) {
    /* Lab2-1 assignment */
    binary = (0x0C << 2) + 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    binary += 0x5 <<12;//funct3
    binary += (reg_to_num(arg2, line_no) << 15);//rs1
    binary += (reg_to_num(arg3, line_no) << 20);///rs2
    binary += (0x20 << 25);//funct7
    //warn("Lab2-1 assignment: SRA instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == OR) {
    /* Lab2-1 assignment */
    binary = (0x0C << 2) + 0x03;
    binary += (reg_to_num(arg1, line_no) << 7);
    binary += 0x6 <<12;
    binary += (reg_to_num(arg2, line_no) << 15);
    binary += (reg_to_num(arg3, line_no) << 20);
    binary += (0x0 << 25);
    //warn("Lab2-1 assignment: OR instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == AND) {
    /* Lab2-1 assignment */
    binary = (0x0C << 2) + 0x03;
    binary += (reg_to_num(arg1, line_no) << 7);
    binary += 0x7 <<12;
    binary += (reg_to_num(arg2, line_no) << 15);
```

```
} else if (is_opcode(opcode) == AND) {
    /* Lab2-1 assignment */
    binary = (0x0C << 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);
}
```

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	
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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	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]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	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 ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch >=	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch > (U)	B	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	

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.

```
else if (is_opcode(opcode) == JALR) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `parse_regs_indirect_addr`
     * e.g., parse_regs_indirect_addr(arg2, line_no)
     */
    binary = (0x19 <<2) + 0x03;//opcode
    binary += 0x0 <<12;//funct3
    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 += MASK11_0(ret->imm) << 20;//imm
    //warn("Lab2-1 assignment: JALR instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == JAL) {
    /*
     * Lab2-1 assignment
     * tip: you may need the function `handle_label_or_imm`
     * e.g., handle_label_or_imm(arg2, label_table, cmd_no, line_no)
     */
    binary = (0x1B << 2) + 0x03;//opcode
    binary += (reg_to_num(arg1, line_no) << 7);//rd
    int val = handle_label_or_imm(line_no,arg2,label_table,number_of_labels);//change label to num
    int offset= val-addr;
    binary += offset & 0xFF000;//imm[19:12]
    binary += (offset & 0x800) << 9;//imm[11]
    binary += (offset & 0x7FE) << 20;//imm[10:1]
    binary += (offset & 0x100000) <<11;//imm[20]
    //warn("Lab2-1 assignment: JAL instruction\n");
    //exit(EXIT_FAILURE);
}
```

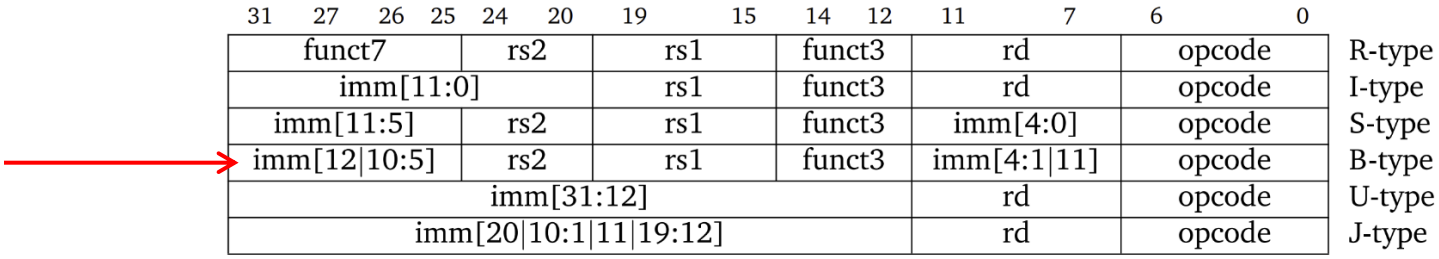
Here is the code of J-type. Since in jal , there will be a label (the function name), so we need to change it to immediate number by using handle_label_or_imm.

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);
}
```

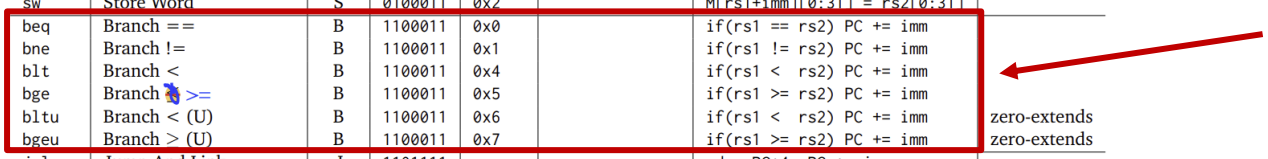
In here, we could see that we are trying to change the assembley code to RV321 assembler. For the first line, it will equal to a 7bit **opcode** number**.** For the second line, it will equal to a 5-bit **rs1** register. For the third line, it will equal to a 5-bit **rs2** register. For the remaining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



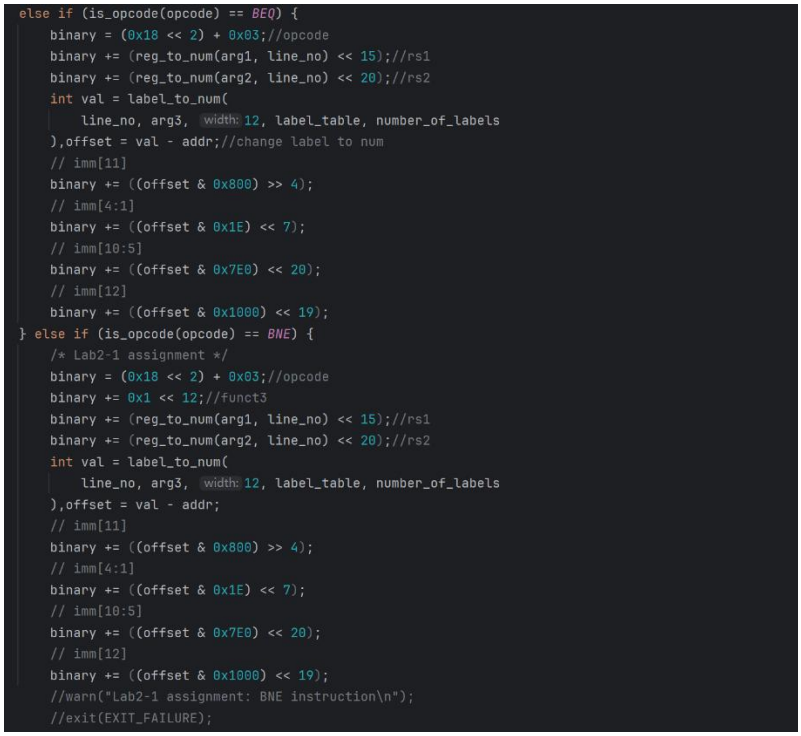
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

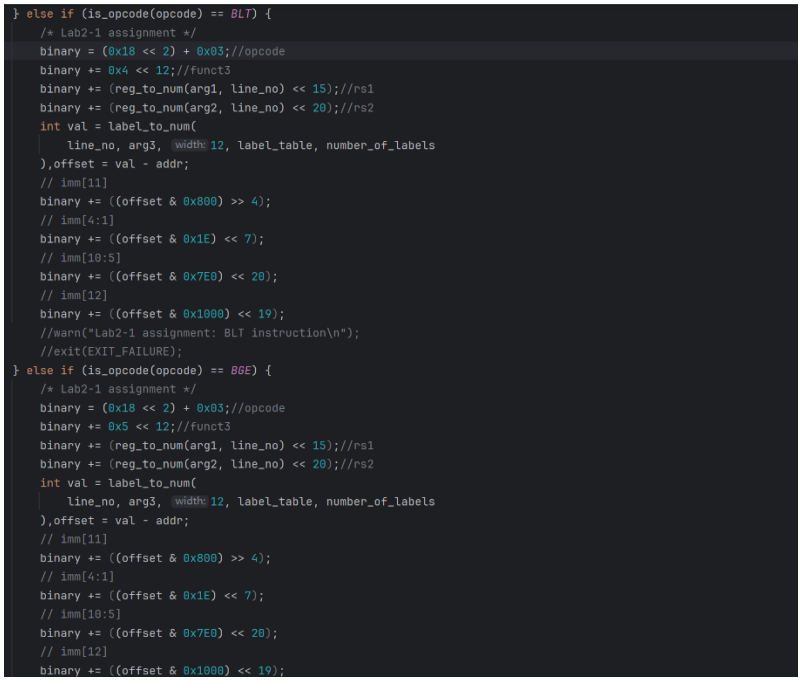
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	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]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	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 ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch >=	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch > (U)	B	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 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.



```
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, width: 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);
} else if (is_opcode(opcode) == BNE) {
    /* Lab2-1 assignment */
    binary = (0x18 << 2) + 0x03;//opcode
    binary += 0x1 << 12;//funct3
    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, width: 12, label_table, number_of_labels
    ),offset = val - addr;
    // 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);
    //warn("Lab2-1 assignment: BNE instruction\n");
    //exit(EXIT_FAILURE);
```



```
} else if (is_opcode(opcode) == BLT) {
    /* Lab2-1 assignment */
    binary = (0x18 << 2) + 0x03;//opcode
    binary += 0x4 << 12;//funct3
    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, width: 12, label_table, number_of_labels
    ),offset = val - addr;
    // 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);
    //warn("Lab2-1 assignment: BLT instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == BGE) {
    /* Lab2-1 assignment */
    binary = (0x18 << 2) + 0x03;//opcode
    binary += 0x5 << 12;//funct3
    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, width: 12, label_table, number_of_labels
    ),offset = val - addr;
    // 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);
```

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
}
```

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	
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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	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]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	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 ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥ >=	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	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	

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->imm << 20);//imm
} else if (is_opcode(opcode) == LH) {
    /* Lab2-1 assignment */
    binary = 0x03;//opcode
    binary += 0x1 <<12;//funct3
    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 += (MASK11_0(ret->imm) << 20);//imm
    //warn("Lab2-1 assignment: LH instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == LW) {
    /* Lab2-1 assignment */
    binary = 0x03;//opcode
    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 += (MASK11_0(ret->imm) << 20);//imm
    //warn("Lab2-1 assignment: LW instruction\n");
    //exit(EXIT_FAILURE);
```

```
} else if (is_opcode(opcode) == SB) {
    /* Lab2-1 assignment */
    binary = (0x08<<2) +0x03;//opcode
    binary += (reg_to_num(arg1, 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->imm&0x01F) << 7);//imm[4:0]
    binary += ((ret->imm&0xFE0) << 20);//imm[11:5]
    //warn("Lab2-1 assignment: SB instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SH) {
    /* Lab2-1 assignment */
    binary = (0x08<<2) +0x03;
    binary += 0x1 << 12;//funct3
    binary += (reg_to_num(arg1, line_no) << 20);
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);
    binary += ((ret->imm&0x01F) << 7);
    binary += ((ret->imm&0xFE0) << 20);
    //warn("Lab2-1 assignment: SH instruction\n");
    //exit(EXIT_FAILURE);
} else if (is_opcode(opcode) == SW) {
    /* Lab2-1 assignment */
    binary = (0x08<<2) +0x03;
    binary += 0x2 << 12;//funct3
    binary += (reg_to_num(arg1, line_no) << 20);
    struct_regs_indirect_addr* ret = parse_regs_indirect_addr(arg2, line_no);
    binary += (reg_to_num(ret->reg, line_no) << 15);
    binary += ((ret->imm & 0x01F) << 7);
    binary += ((ret->imm&0xFE0) << 20);
    //warn("Lab2-1 assignment: SW instruction\n");
    //exit(EXIT_FAILURE);
```

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.