smallest # bits needed to represent -9

why is ans 5?

---

9 in unsigned binary

$9 = 0b1001$

$8+1=9$

In signed binary

$0b1001 = 1\times(-2^3)+1$

$= -7$

$0x1001$

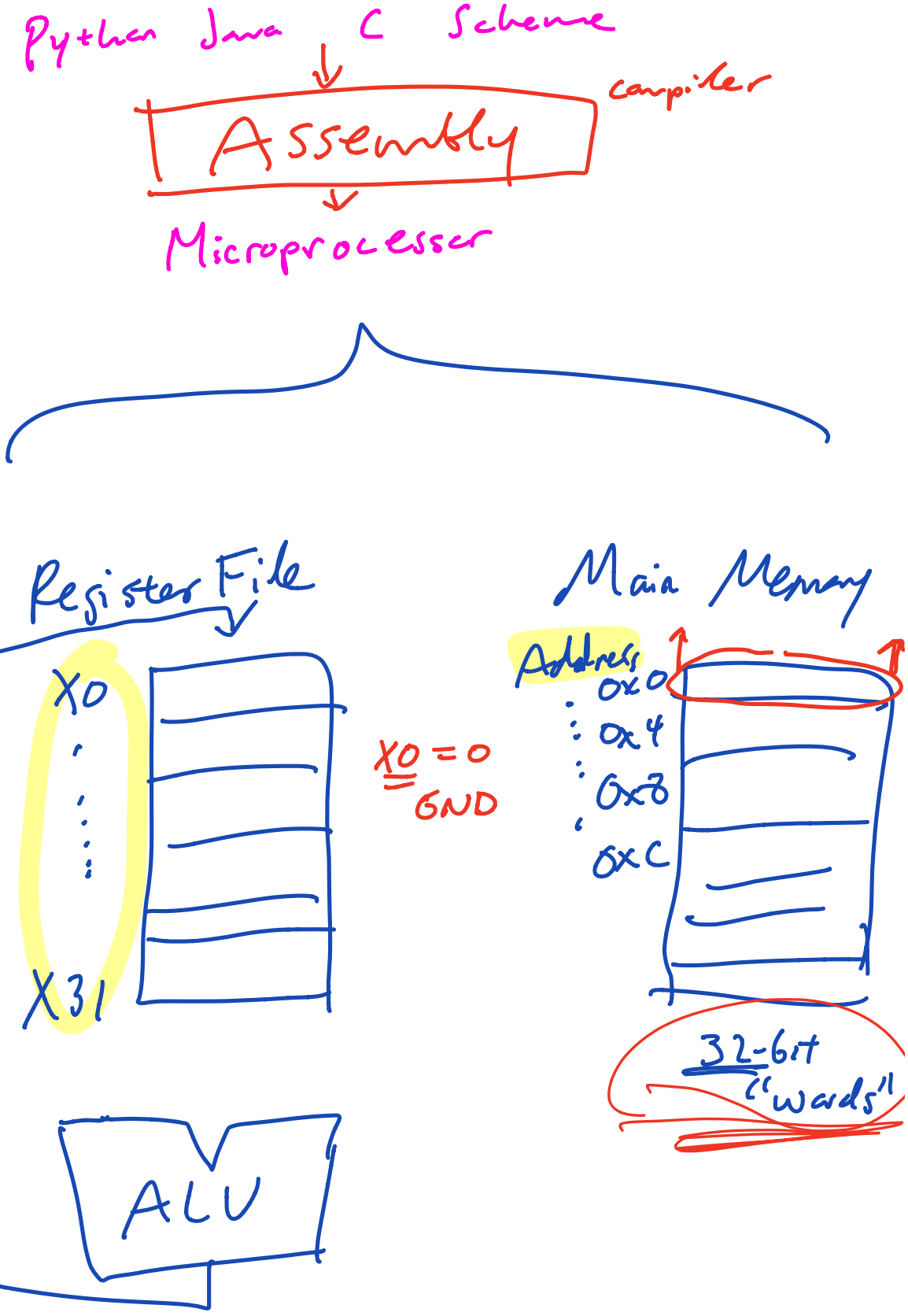
flip & add 1 $\Rightarrow$ $0b0110+1 = 0b0111$

$= 7$

-2^{N-1} $\qquad$ $2^{N-1}-1$

$N=4 \quad -2^{4-1} = -8$

$N=5 \quad -2^{5-1} = -16$


Python Java C Scheme
compiler
Assembly
Microprocessor
Register File
X0
X31
XO = 0
GND
ALU
Main Memory
Address
0x0
0x4
0x8
0xC
32-bit
"words"

6.004 Spring 2021 Tutorial Problems
L02 – RISC-V Assembly

Computational Instructions

R-type: Register-register instructions: opcode = OP = 0110011

Arithmetic	Comparisons	Logical	Shifts
ADD, SUB	SLT, SLTU	AND, OR, XOR	SLL, SRL, SRA

Assembly instr: **oper rd, rs1, rs2**
Behavior: **reg[rd] <= reg[rs1] oper reg[rs2]**

- Rd = destination register (where result is saved)
- Rs1, rs2 = operand registers (their contents are used in the calculation)
- Example: "add x1, x2, x3" means "set x1 equal to x2 + x3"

SLT – Set less than
SLTU – Set less than unsigned
SLL – Shift left logical
SRL – Shift right logical
SRA – Shift right arithmetic

I-type: Register-immediate instructions: with opcode = OP-IMM = 0010011

Arithmetic	Comparisons	Logical	Shifts
ADDI	SLTI, SLTIU	ANDI, ORI, XORI	SLLI, SRLI, SRAI

Assembly instr: **oper rd, rs1, immI**

Behavior: **imm = signExtend(immI)** (sign extend to 32 bits)
reg[rd] <= reg[rs1] oper imm

- "Immediate" just means constant; immI is a 12-bit constant.
- Same functions as R-type except SUBI is not needed because immediate can be negative.
- Function is encoded in funct3 bits plus instr[30]. Instr[30] = 1 for SRAI. So SRLI and SRAI use same funct3 encoding.
- Example: "addi x1, x2, 0x4A7" means "set x1 equal to x2 + 1191"

U-type: opcode = LUI or AUIPC = (01|00)10111

LUI – load upper immediate
AUIPC – add upper immediate to PC (program counter)

Assembly instr: **lui rd, immU**

Behavior: **imm = {immU, 12'b0}** (immU concat. with 12 bits of 0)
Reg[rd] <= imm

- LUI is used to set a register equal to a number that is greater than 12 bits. A register can contain up to 32 bits, but "addi" only works with 12; LUI is used for the remaining 20 (32 – 12 = 20).
- immU = a 20 bit constant
- Example: "lui x2, 0x12345" would load register x2 with 0x12345000.
- In practice, it is simpler to use the pseudo-instruction "li" for loads of any size; see below for more details on pseudo-instructions.

Load Store Instructions

I-type: Load: with opcode = LOAD = 0000011

LW – load word

Assembly instr: **lw rd, immI(rs1)**

Behavior: **imm = signExtend(immI)** (to 32 bits)
Reg[rd] <= Mem[R[rs1] + imm]

- immI is a 12-bit constant known as the "offset;" this instruction will load the value located at an address given by the contents of rs1 + the offset. This is useful for accessing contiguous memory locations within the same program. The offset should be a multiple of 4 since a word (32 bits) in memory takes up 4 bytes and memory is byte-addressed.
- Example: If register x1 contains 0x1000, then "lw x2, 8(x1) will find the memory address 0x1008 and copy its contents into register x2.

S-type: Store: opcode = STORE = 0100011

SW – store word

Assembly instr: **sw rs2, immS(rs1)**

Behavior: **imm = signExtend(immS)**
Mem[R[rs1] + imm] <= R[rs2]

- immS is a 12-bit constant "offset" which works the same way as the offset described above for LW.
- Example: If register x3 contains 0x2000 and register x4 contains 0x3, the instruction "sw x4, 4(x3)" will store the value 0x3 into the memory location 0x2004.

Control Instructions

B-type: Conditional Branches: opcode = 1100011

Assembly instr:	**oper rs1, rs2, label**
Behavior:	**imm = distance to label in bytes =** **signExtend({immB[12:1],0})** **pc <= (R[rs1] comp R[rs2]) ? pc + imm : pc + 4**

Compares register rs1 to rs2. If comparison is true, then the program counter (PC) jumps to the instruction following the label specified; in other words, PC is updated with PC + imm. If the comparison is false, PC becomes PC + 4, aka the next instruction (no jumping). Comparison type is defined by operation.

- BEQ – branch if equal (==)
- BNE – branch if not equal (!=)
- BLT – branch if less than (<)
- BGE – branch if greater than or equal (>=)
- BLTU – branch if less than using unsigned numbers (< unsigned)
- BGEU – branch if greater than or equal using unsigned numbers (>= unsigned)

J-type: Unconditional Jump: opcode = JAL = 1101111

Assembly instr:	**JAL rd, label**
Behavior:	**imm = distance to label in bytes =** **signExtend({immJ{20:1},0})** **pc[rd] <= pc + 4; pc <= pc + imm**

- JAL = "jump and link"
- Saves PC+4 (the return address) into rd
- Sets PC = PC + jump distance (to label specified)
- immJ is a 20 bit constant; therefore, the jump distance must be <= 20 bits, aka within 2^{18} instructions of the PC (because there are 4 addresses per instruction)
- Use it for functions: "jal ra, FuncName" (will be discussed in more detail later)

I-type: Unconditional Jump: opcode = JALR = 1100111

Assembly instr:	**JALR rd, rs1, immI**
Behavior:	**imm = signExtend(immI)** **pc[rd] <= pc + 4; pc <= (R[rs1]+imm) & ~0x00000001** **(zero out the bottom bit of pc)**

- JALR = "jump and link register"
- Writes PC+4 (return address) to rd and sets PC = rs1 + immI
- immI is a 12-bit constant

Common pseudoinstructions:

Jump:
j label = jal x0, label (ignore return address)

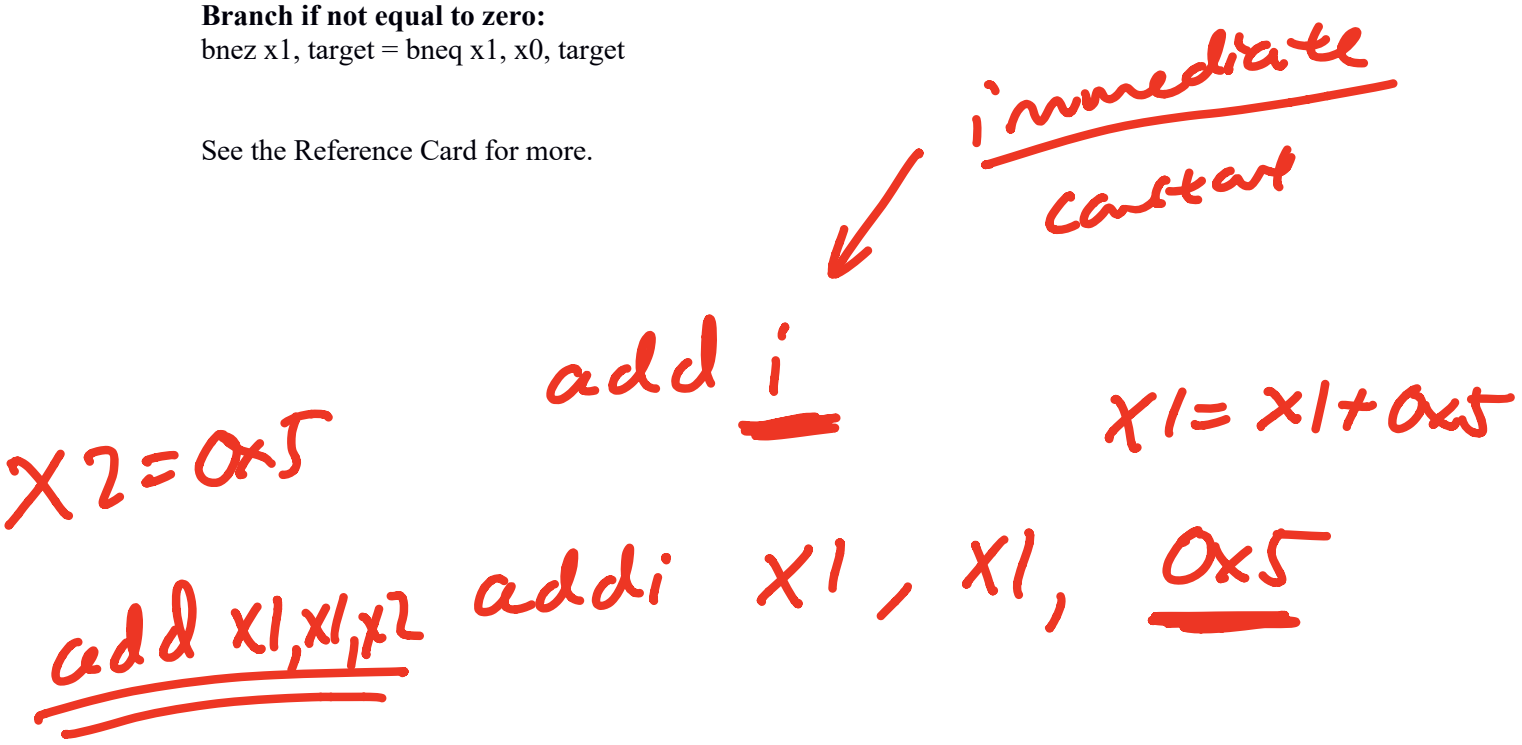
Load Immediate:
li x1, 0x1000 = lui x1, 1
li x1, 0x1100 = lui x1, 1; addi x1, x1, 0x100
li x4, 3 = addi x4, x0, 3

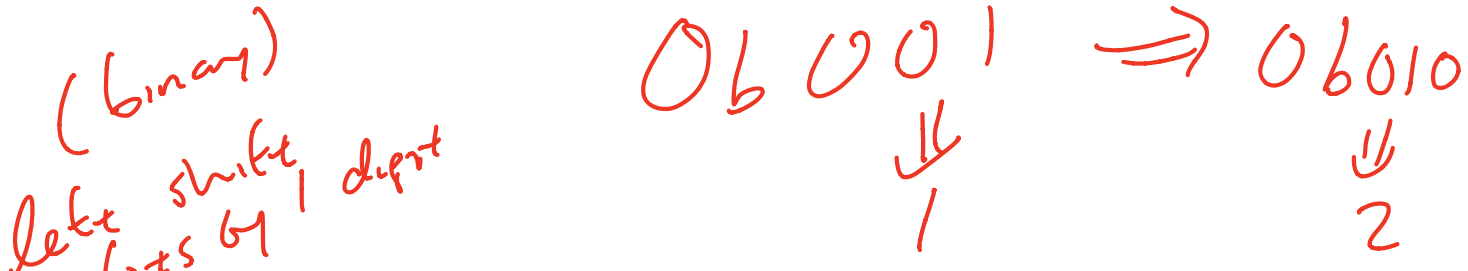
Move:
mv x3, x2 = addi x3, x2, 0

Branch if equal to zero:
beqz x1, target = beq x1, x0, target

Branch if not equal to zero:
bnez x1, target = bneq x1, x0, target

See the Reference Card for more.





Note: A small subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1.

Compile the following expressions to RISCV assembly. Assume a is stored at address 0x1000, b is stored at 0x1004, and c is stored at 0x1008. Assume that all values are 32-bit signed integers.

1. `a = b + 3*c;` ★

```
li  x5  0x1000
lw  x2  0(x5)
lw  x3  4(x5)
lw  x4  8(x5)
```

```
// x5=0x1000
// a=x2
// b=x3
// c=x4
```

```
slli x6, x4, 1
add  x6, x6, x4
add  x2, x6, x3
sw   x2, 0(x5)
```

2. `if (a > b) { c = 17; }` ★

```
li  x1, 0x1000
lw  x2, 0(x1)   // a=x2
lw  x3, 4(x1)   // b=x3
bge x3, x2, ennnd   // if x3≥x2, jump to end
li  x4, 17      // c=x4
sw  x4 8(x1)
```

ennnd:

3.
```
sum = 0;
for (i = 0; i < 10; i = i+1) { sum += i; }
```

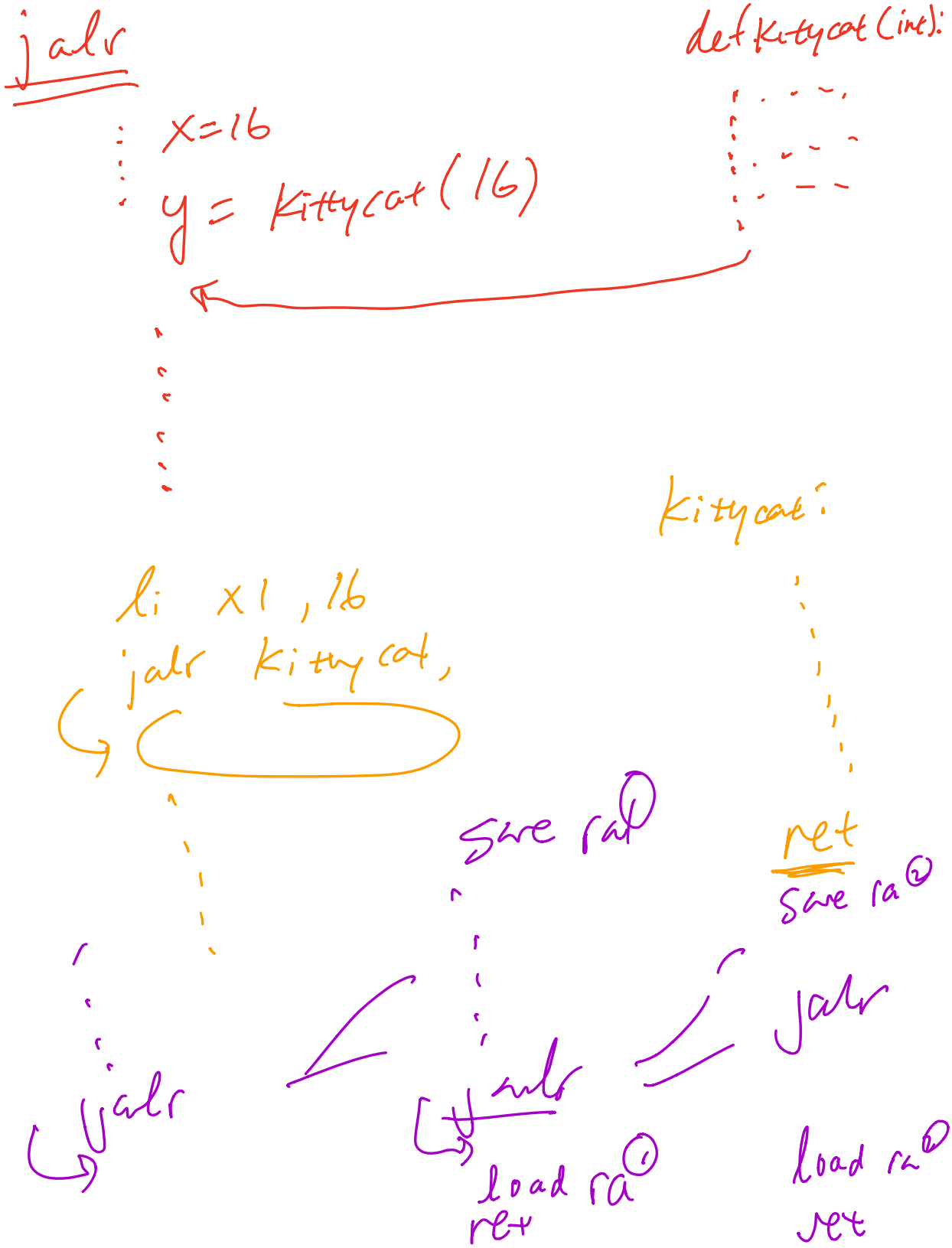
```
addi x1, x0, 0   // x1=0 (sum)
addi x2, x0, 0   // x2=0 (i)
addi x3, x0, 10  // x3=10
looooop:
    add  x1, x1, x2   // x1=x1+x2  sum=sum+i
    addi x2, x2, 1    // x2=x2+1   i=i+1
    blt  x2, x3, looooop   // if x2<x3, jump to looooop
```

Problem 2. ★

Compile the following expression assuming that a is stored at address 0x1100, and b is stored at 0x1200, and c is stored at 0x2000. Assume a, b, and c are arrays whose elements are stored in consecutive memory locations. Assume that all values are 32-bit signed integers.

```
for (i = 0; i < 10; i = i+1) { c[i] = a[i] + b[i]; }
```

```
addi  x1, x0, 0  //x1=0 (sum)
addi  x2, x0, 0  // x2=0  (i)
addi  x3, x0, 10  //x3=10
loop420:
        bge  x2, x3, end420
        add  x1 , x1 , x2
        addi  x2, x2, 1
        jal  loop420
end420:
```


jalr
def kittycat(int):
x=16
y = kittycat(16)
kittycat:
li x1, 16
jalr kittycat,
save ra
ret
save ra
jalr
jalr
jalr
load ra
ret
load ra
ret

Problem 3. ★

Hand assemble the following sequence of instructions into its equivalent binary encoding. (Hint: use the ISA Reference Card at the end of this worksheet to parse and encode the instruction)

```
addi x1, x1, -1
```

MIT 6.004 ISA Reference Card: Instruction Encodings

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[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 Instruction Set (MIT 6.004 subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Problem 4.

A) Assume that the registers are initialized to: x1=8, x2=10, x3=12, x4=0x1234, x5=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. `SLL x6, x4, x5` **Value of x6: _____** ★
2. `ADD x7, x3, x2` **Value of x7: _____**
3. `ADDI x8, x1, 2` **Value of x8: _____**
4. `SW x2, 4(x4)` **Value stored: _____ at address: _____** ★

B) Assume X is at address 0x1CE8

```
    li x1, 0x1CE8
    lw x4, 0(x1)
    blt x4, x0, L1
    addi x2, x0, 17
    beq x0, x0, L2
L1: srai x2, x4, 4
L2:

X:  .word 0x87654321
```

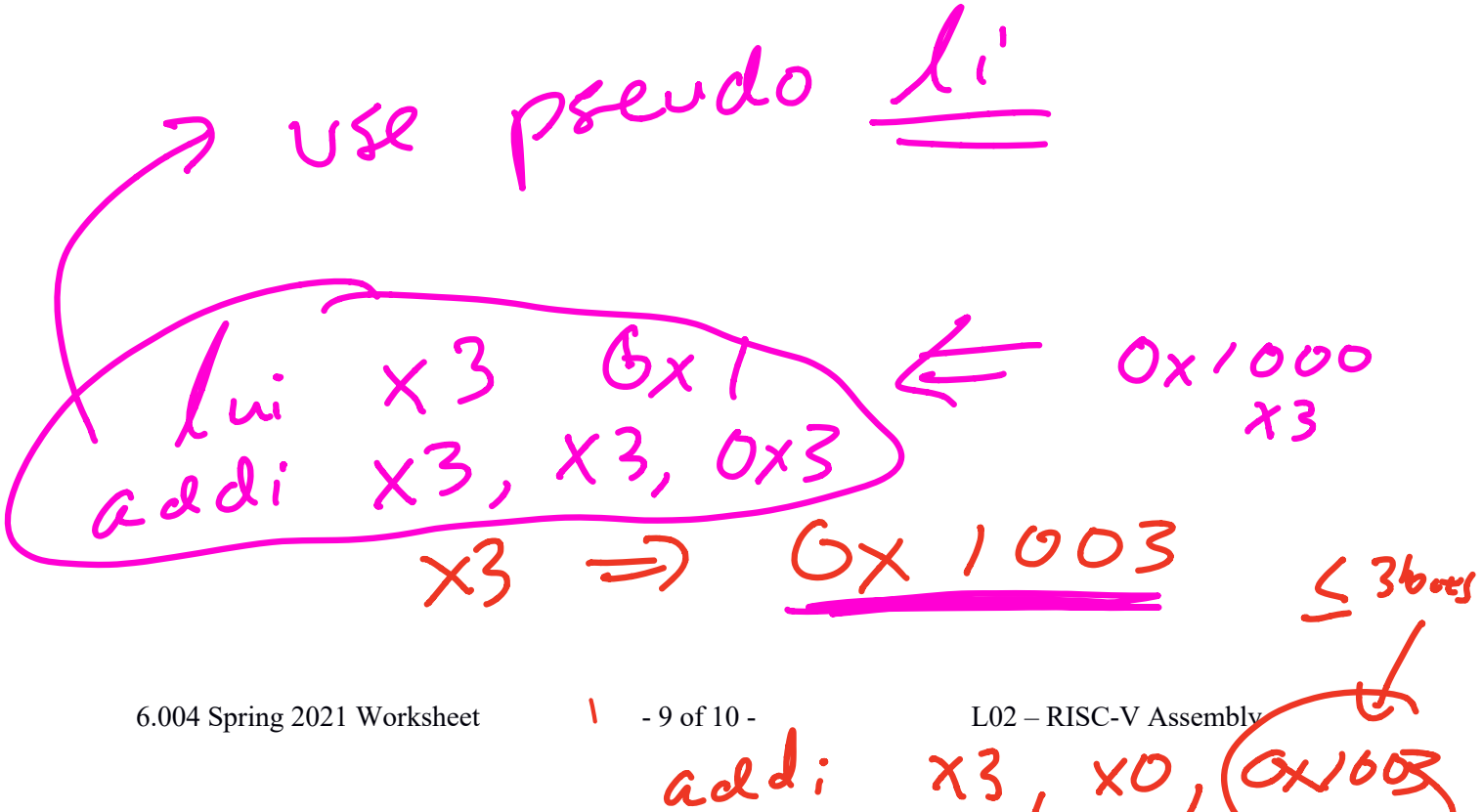
Value left in x4? _____

Value left in x2? _____

Problem 5.

Compile the following Fibonacci implementation to RISCV assembly.

```
# Reference Fibonacci implementation in Python
def fibonacci_iterative(n):
    if n == 0:
        return 0
    n = n - 1
    x, y = 0, 1
    while n > 0:
        # Parallel assignment of x and y
        # The new values for x and y are computed at the same time, and
        # then the values of x and y are updated afterwards
        x, y = y, x + y
        n = n - 1
    return y
```



MIT 6.004 ISA Reference Card: Instructions

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label : pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label : pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label : pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label : pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label : pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label : pc + 4`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + offset]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[reg[rs1] + offset] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

Note: *luiConstant* is a 20-bit value. *offset* and *constant* are signed 12-bit values that are sign-extended to 32-bit values. *label* is a 32-bit memory address or its alias name.

MIT 6.004 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
`li rd, constant`	Load Immediate	`reg[rd] <= constant`
`mv rd, rs1`	Move	`reg[rd] <= reg[rs1] + 0`
`not rd, rs1`	Logical Not	`reg[rd] <= reg[rs1] ^ -1`
`neg rd, rs1`	Arithmetic Negation	`reg[rd] <= 0 - reg[rs1]`
`j label`	Jump	`pc <= label`
`jal label` `call label`	Jump and Link (with ra)	`reg[ra] <= pc + 4` `pc <= label`
`jr rs`	Jump Register	`pc <= reg[rs1] & ~1`
`jalr rs`	Jump and Link Register (with ra)	`reg[ra] <= pc + 4` `pc <= reg[rs1] & ~1`
`ret`	Return from Subroutine	`pc <= reg[ra]`
`bgt rs1, rs2, label`	Branch > (Signed)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`ble rs1, rs2, label`	Branch ≤ (Signed)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`bgtu rs1, rs2, label`	Branch > (Unsigned)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`bleu rs1, rs2, label`	Branch ≤ (Unsigned)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`beqz rs1, label`	Branch = 0	`pc <= (reg[rs1] == 0) ? label : pc + 4`
`bnez rs1, label`	Branch ≠ 0	`pc <= (reg[rs1] != 0) ? label : pc + 4`
`bltz rs1, label`	Branch < 0 (Signed)	`pc <= (reg[rs1] <s 0) ? label : pc + 4`
`bgez rs1, label`	Branch ≥ 0 (Signed)	`pc <= (reg[rs1] >=s 0) ? label : pc + 4`
`bgtz rs1, label`	Branch > 0 (Signed)	`pc <= (reg[rs1] >s 0) ? label : pc + 4`
`blez rs1, label`	Branch ≤ 0 (Signed)	`pc <= (reg[rs1] <=s 0) ? label : pc + 4`

MIT 6.004 ISA Reference Card: Calling Convention

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

MIT 6.004 ISA Reference Card: Instruction Encodings

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[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 Instruction Set (MIT 6.004 subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

- For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., $pc + imm = label$).
- Not all immediate bits are encoded. Missing lower bits are filled with zeros and missing upper bits are sign-extended.

lui x3, 3

x3 = 0x3000

li x3, 3 {
lui x3, 0
addi x3, x3, 3
}

addi x3, x3, 0x3000

x3 = 0x3