Machine Architecture

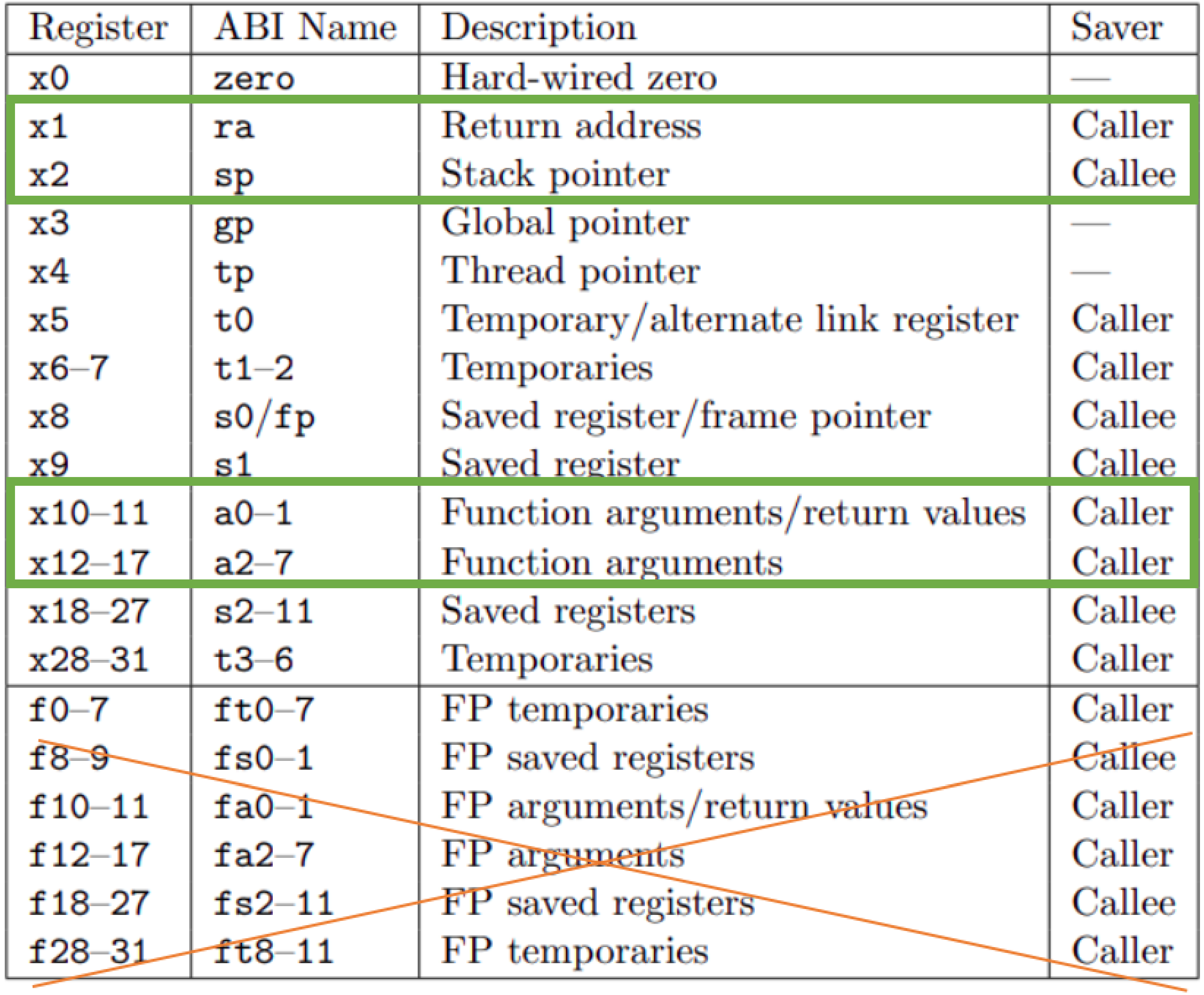
Assembler Programming

Fredag 10. Januar 2025

Benjamin Shultz

Computersystemer 2024/2025

Content



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
~~f8–9~~	~~fs0–1~~	~~FP saved registers~~	~~Callee~~
~~f10–11~~	~~fa0–1~~	~~FP arguments/return values~~	~~Caller~~
~~f12–17~~	~~fa2–7~~	~~FP arguments~~	~~Caller~~
~~f18–27~~	~~fs2–11~~	~~FP saved registers~~	~~Callee~~
~~f28–31~~	~~ft8–11~~	~~FP temporaries~~	~~Caller~~

Registers

X1 holds the address which will be jumped to when running *ret*

X10-17 contains function arguments. Caller writes to these registers, and callee reads from them.

X2/sp can be descreased to hold additional arguments on the stack.

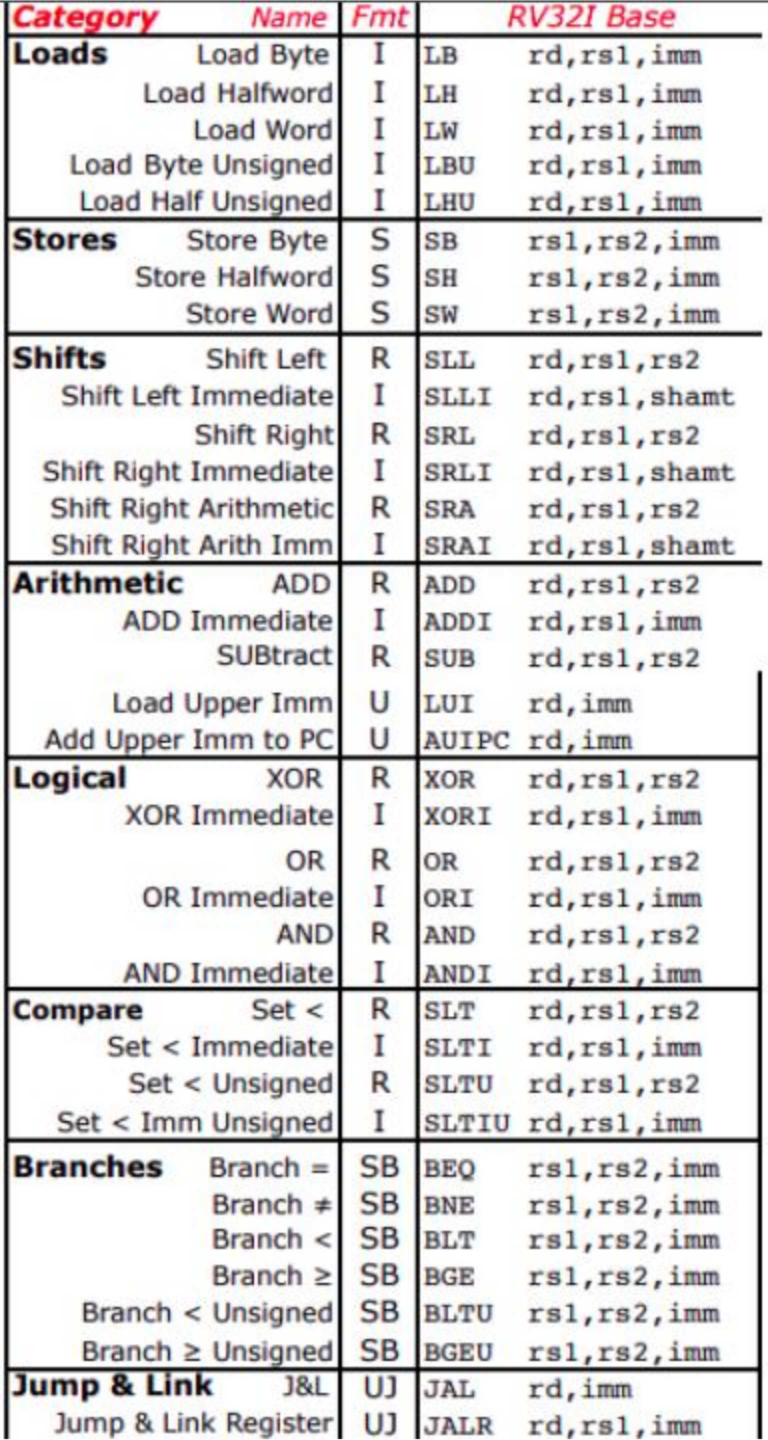
Source: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

RV32I Base Integer Instructions

rd is destination register

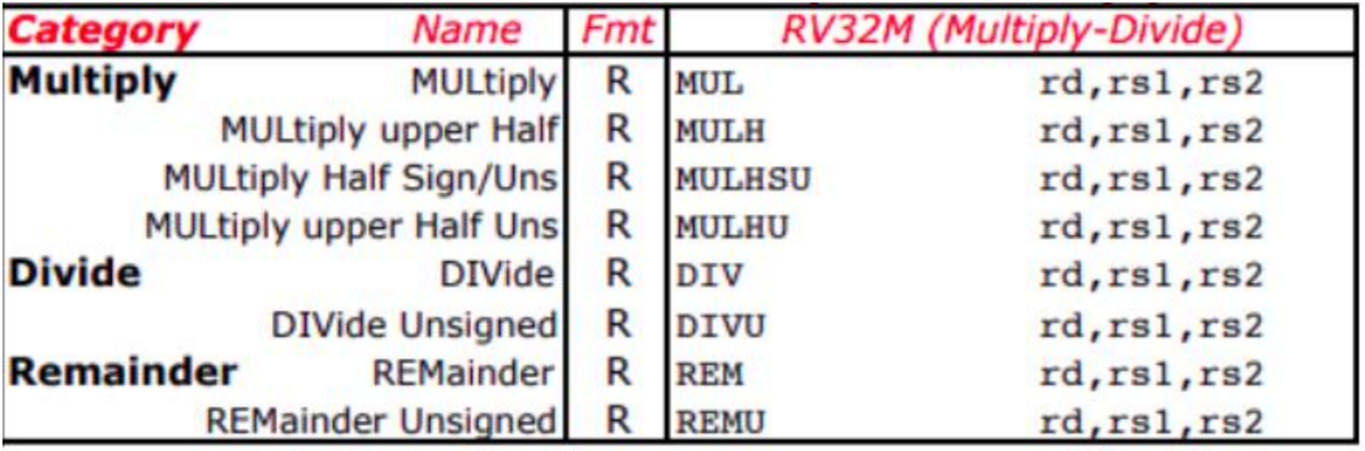
rs1 and *rs2* are source registers

Source: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf



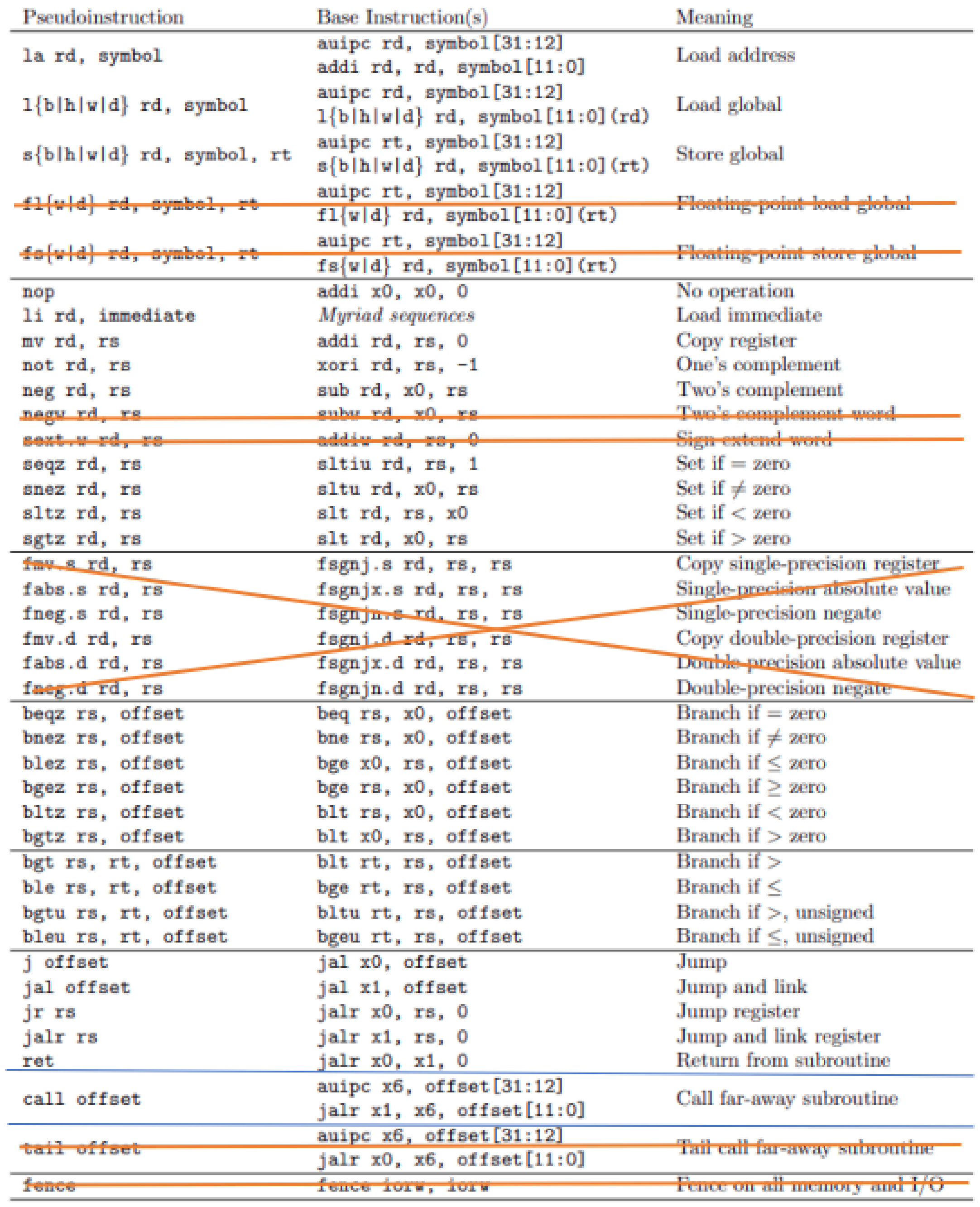
Category	Name	Fmt	RV32I Base	
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link	J&L	UJ	JAL	rd,imm
	Jump & Link Register	UJ	JALR	rd,rs1,imm

RV32M Integer Multiplication and Division



Category	Name	Fmt	RV32M (Multiply-Divide)	
Multiply	MULtiply	R	`MUL`	`rd,rs1,rs2`
	MULtiply upper Half	R	`MULH`	`rd,rs1,rs2`
	MULtiply Half Sign/Uns	R	`MULHSU`	`rd,rs1,rs2`
	MULtiply upper Half Uns	R	`MULHU`	`rd,rs1,rs2`
Divide	DIVide	R	`DIV`	`rd,rs1,rs2`
	DIVide Unsigned	R	`DIVU`	`rd,rs1,rs2`
Remainder	REMainder	R	`REM`	`rd,rs1,rs2`
	REMainder Unsigned	R	`REMU`	`rd,rs1,rs2`

Source: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf



Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address
l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	Load global
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	Store global
~~fl{w|d} rd, symbol, rt~~	auipc rt, symbol[31:12] fl{w|d} rd, symbol[11:0](rt)	~~Floating-point load global~~
~~fs{w|d} rd, symbol, rt~~	auipc rt, symbol[31:12] fs{w|d} rd, symbol[11:0](rt)	~~Floating-point store global~~
nop	addi x0, x0, 0	No operation
li rd, immediate	*Myriad sequences*	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
~~negw rd, rs~~	~~subw rd, x0, rs~~	~~Two's complement word~~
~~sext.w rd, rs~~	~~addiw rd, rs, 0~~	~~Sign extend word~~
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
~~fmv.s rd, rs~~	~~fsgnj.s rd, rs, rs~~	~~Copy single-precision register~~
~~fabs.s rd, rs~~	~~fsgnjx.s rd, rs, rs~~	~~Single-precision absolute value~~
~~fneg.s rd, rs~~	~~fsgnjn.s rd, rs, rs~~	~~Single-precision negate~~
~~fmv.d rd, rs~~	~~fsgnj.d rd, rs, rs~~	~~Copy double-precision register~~
~~fabs.d rd, rs~~	~~fsgnjx.d rd, rs, rs~~	~~Double-precision absolute value~~
~~fneg.d rd, rs~~	~~fsgnjn.d rd, rs, rs~~	~~Double-precision negate~~
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if ≠ zero
blez rs, offset	bge x0, rs, offset	Branch if ≤ zero
bgez rs, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if ≤, unsigned
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, rs, 0	Jump register
jalr rs	jalr x1, rs, 0	Jump and link register
ret	jalr x0, x1, 0	Return from subroutine
call offset	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]	Call far-away subroutine
~~tail offset~~	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	~~Tail call far-away subroutine~~
~~fence~~	~~fence iorw, iorw~~	~~Fence on all memory and I/O~~

Table 20.2: RISC-V pseudoinstructions.

Pseudo Instructions

A pseudo instruction is an instruction handled by the assembler by translating it into one or more base (non-pseudo) instructions.

li is usually replaced by *addi* and/or *lui*.

Source: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

Calling Functions

- Functions can be called with the instructions:
  - *jal* (jump and link).
  - *jalr* (jump and link register)
- OBS: *jalr x0, x1, 0* does NOT call functions it returns from a function. (x1 contains the return address)

- Functions can be called with the following pseudo instructions:
  - *j offset* :jump
  - *jal offset* :jump and link (return addr is stored in x1)
  - *jr rs* :jump register
  - *jalr rs* :jump and link register (return addr is stored in x1)
  - *call offset* :jump and link to far away address (return addr is stored in x1)
  - OBS: the pseudo instruction *ret* (instruction *jalr x0, x1, 0*) does NOT call other functions, it returns from a function. (can also be written *jalr x0, ra, 0)*

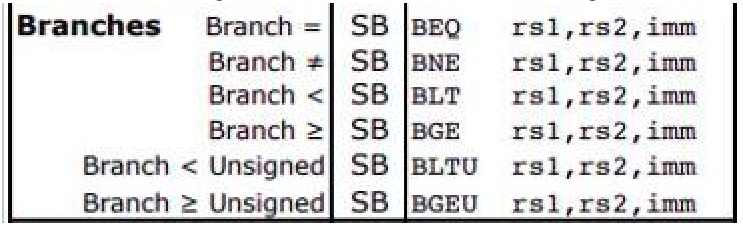
Function Arguments

- Registers a0-7 are used. (if more arguments, the stack is used)
- Function arguments can be stored and loaded from the stack.
- Argument registers are recognized by using the register in a function without first having written to it in that function.

- Caller loads arguments into registers a0-a7.
- Caller calls function (*jal*, *jalr*, etc.)
- Callee uses arguments
- Callee stores result (if any) in a0-1
- Callee returns (ret)
- Caller uses results (if any) from a0-1

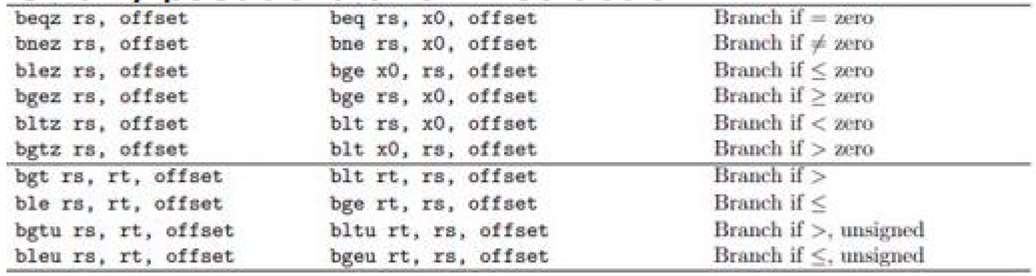
While-loop, For-loop, If-Statement

- Are usually denoted with any branch instruction:



Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm

- Or any pseudo branch instruction:



`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned

- While-loop, For-loop optionally have a *jalr* or *jal* instruction (*j* pseudo instruction)

- OBS: Infinite while-loop can use pseudo instruction *j offset* (instruction *jal x0, offset* , negative offset) with no branch instruction.

Reverse engineering RISC-V to C

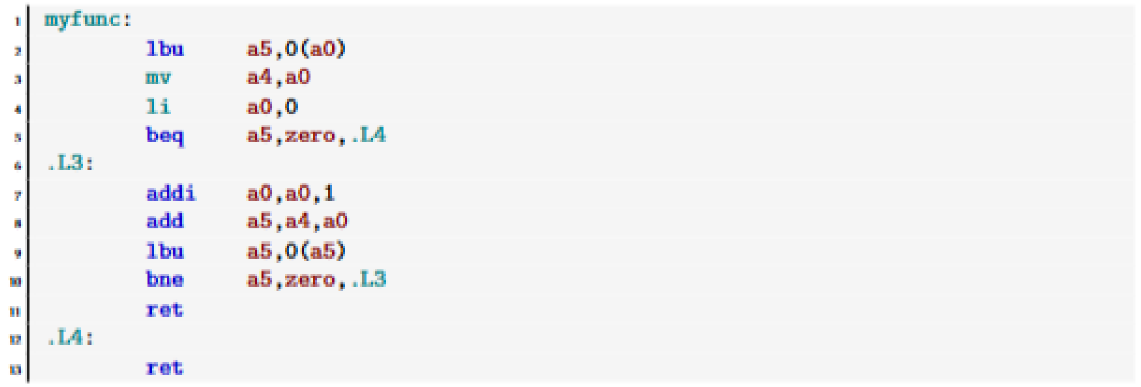
- RISC-V programs are usually written with go-to style
- C programs are usually written with NON go-to style (which is usually a requirement in exams)

- Godbolt can be used to practice translation: https://godbolt.org/z/eh69GT8qW

Example: exam-2023-24

Assembler programming

Consider the following program written in RISC-V assembler.



```
myfunc:
        lbu     a5,0(a0)
        mv      a4,a0
        li      a0,0
        beq     a5,zero,.L4
.L3:
        addi    a0,a0,1
        add     a5,a4,a0
        lbu     a5,0(a5)
        bne     a5,zero,.L3
        ret
.L4:
        ret
```

Question 1.3.1: The code snippet is a function. Is this function calling other functions? Argue for your answer

Question 1.3.2: Which registers hold the functions arguments (if any)? Argue for your answer.

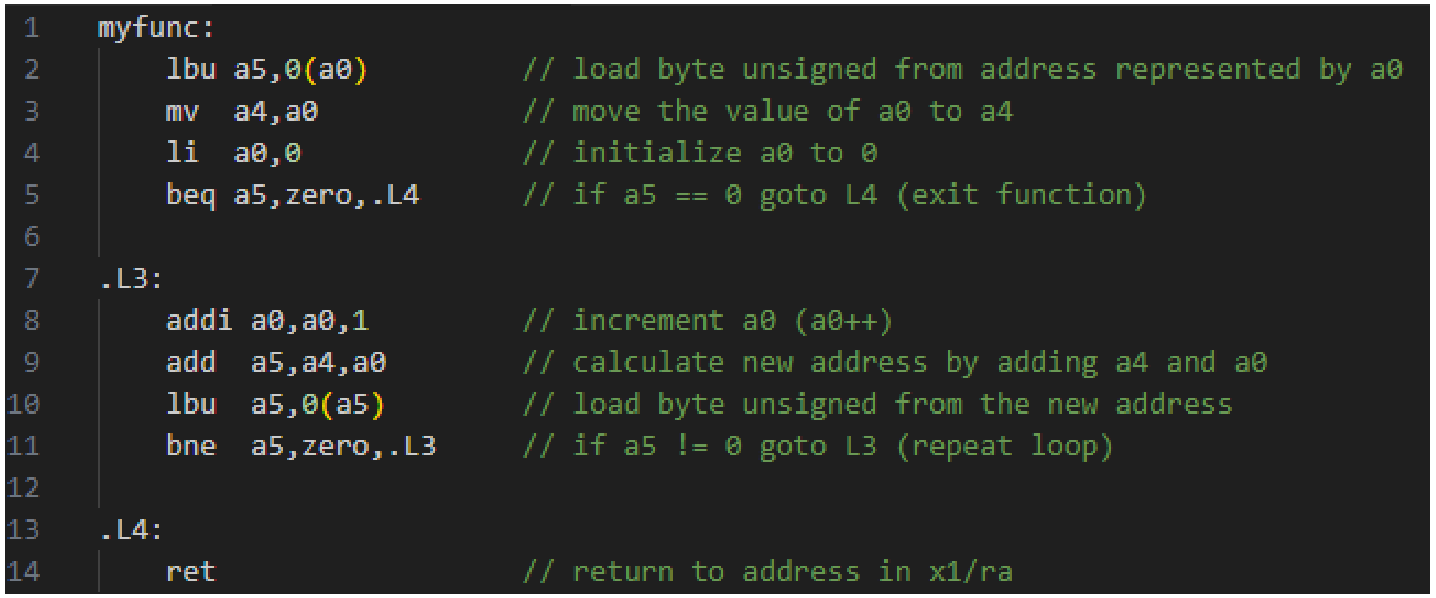
Question 1.3.3: The function contains a loop. Which instructions form the loop? Describe how you identified this.

Question 1.3.4: Rewrite the above RISC-V assembler program to a C program. The resulting program must not have a goto-style and minor syntactical mistakes are acceptable..

Question 1.3.5: Describe shortly the functionality of the program.

Question 1.3.6: What is The purpose of the “lui” instruction and how does it work?

Source: exam-2023-24-form.pdf

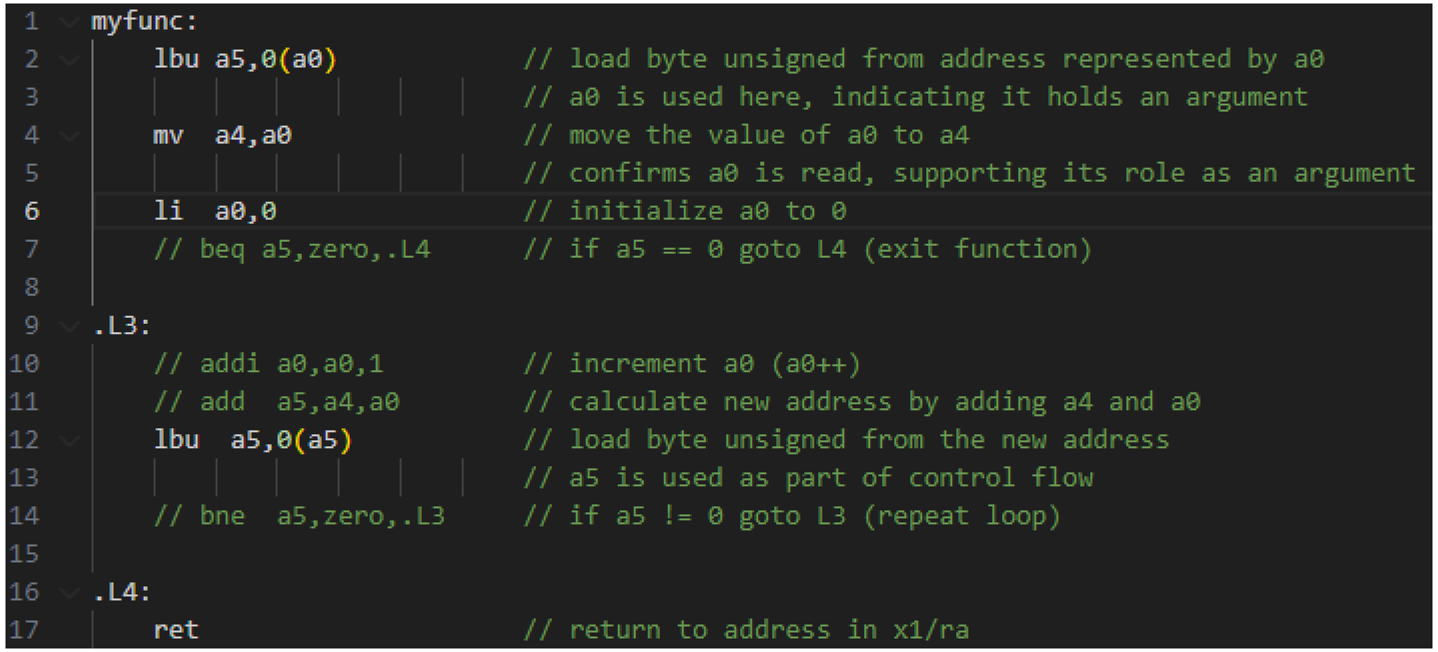


```
myfunc:
    lbu a5,0(a0)          // load byte unsigned from address represented by a0
    mv  a4,a0             // move the value of a0 to a4
    li  a0,0              // initialize a0 to 0
    beq a5,zero,.L4       // if a5 == 0 goto L4 (exit function)

.L3:
    addi a0,a0,1          // increment a0 (a0++)
    add  a5,a4,a0         // calculate new address by adding a4 and a0
    lbu  a5,0(a5)         // load byte unsigned from the new address
    bne  a5,zero,.L3      // if a5 != 0 goto L3 (repeat loop)

.L4:
    ret                   // return to address in x1/ra
```

Question 1.3.1: The code snippet is a function. Is this function calling other functions? Argue for your answer

It is **not** calling other functions.

If it were, it would have JAL or a JALR instruction that is **SAVING** the return address in x1/ra, and not only return pseudo-instructions



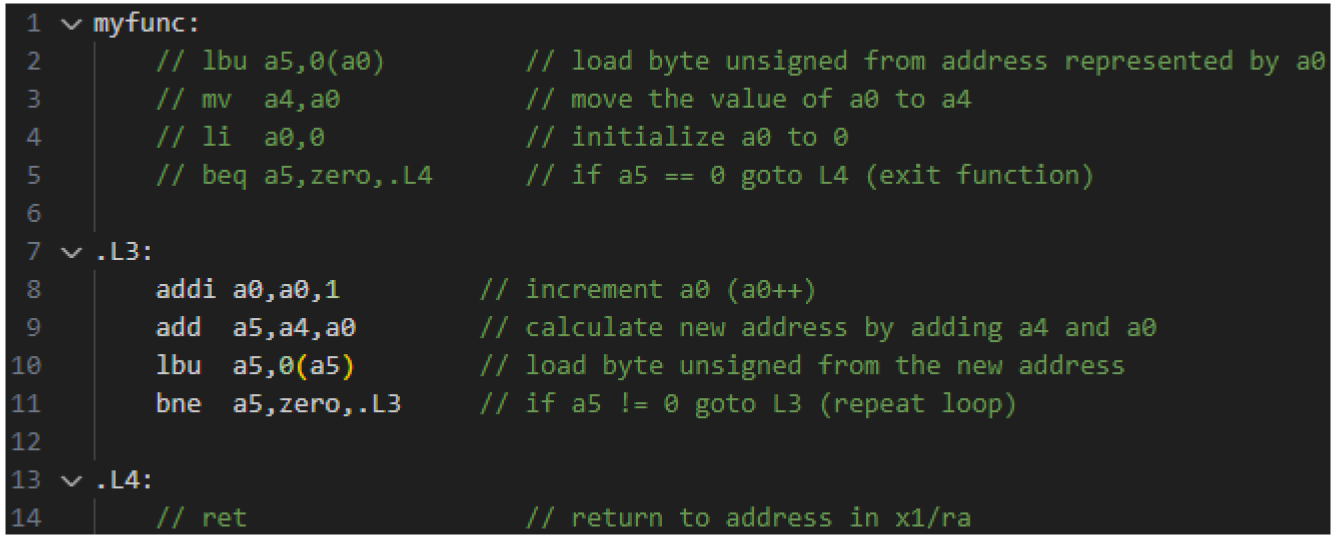
```
 1  myfunc:
 2      lbu a5,0(a0)            // load byte unsigned from address represented by a0
 3                              // a0 is used here, indicating it holds an argument
 4      mv  a4,a0               // move the value of a0 to a4
 5                              // confirms a0 is read, supporting its role as an argument
 6      li  a0,0                // initialize a0 to 0
 7      // beq a5,zero,.L4      // if a5 == 0 goto L4 (exit function)
 8
 9  .L3:
10      // addi a0,a0,1         // increment a0 (a0++)
11      // add  a5,a4,a0        // calculate new address by adding a4 and a0
12      lbu  a5,0(a5)           // load byte unsigned from the new address
13                              // a5 is used as part of control flow
14      // bne  a5,zero,.L3     // if a5 != 0 goto L3 (repeat loop)
15
16  .L4:
17      ret                     // return to address in x1/ra
```

Question 1.3.2: Which registers hold the functions arguments (if any)? Argue for your answer.

Look for registers that are read before they are written to.

The argument is in a0, since this register is used in the function, before it is written indicating it holds an input value passed to the function

In the example, a0 is used on line 2 and line 4 (before being written to on line 6).

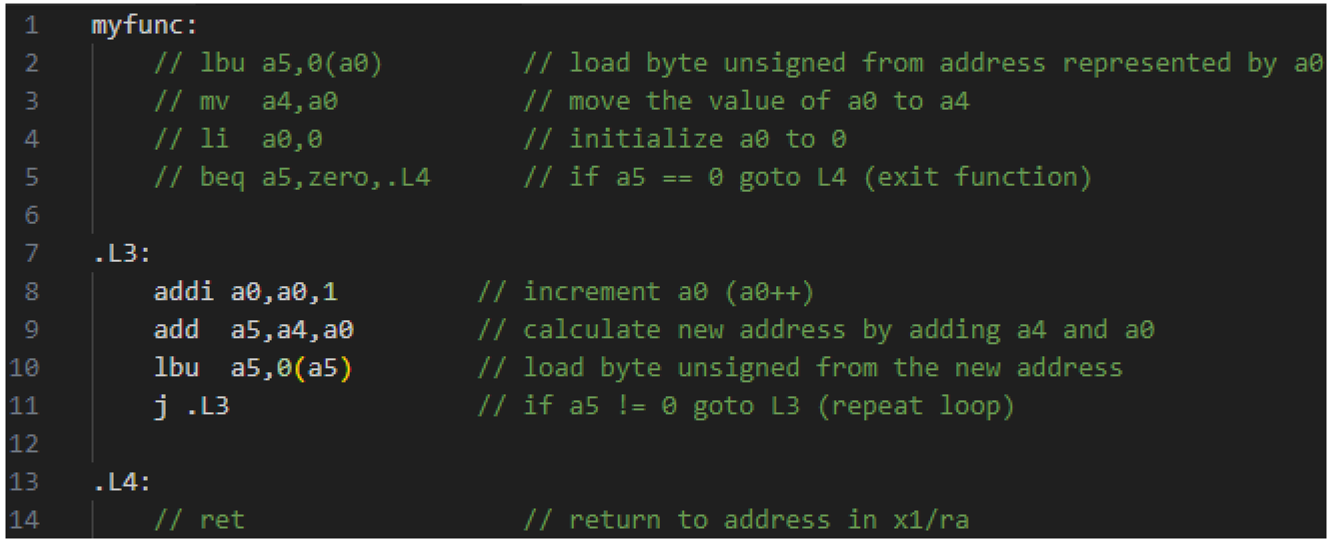


```
myfunc:
    // lbu a5,0(a0)         // load byte unsigned from address represented by a0
    // mv  a4,a0            // move the value of a0 to a4
    // li  a0,0             // initialize a0 to 0
    // beq a5,zero,.L4      // if a5 == 0 goto L4 (exit function)

.L3:
    addi a0,a0,1         // increment a0 (a0++)
    add  a5,a4,a0        // calculate new address by adding a4 and a0
    lbu  a5,0(a5)        // load byte unsigned from the new address
    bne  a5,zero,.L3     // if a5 != 0 goto L3 (repeat loop)

.L4:
    // ret                  // return to address in x1/ra
```

Question 1.3.3: The function contains a loop. Which instructions form the loop? Describe how you identified this.

Look for any branch instructions (slide 9), and optionally a jal or jalr instruction with a negative offset (or a label at a line less than the jump instruction).

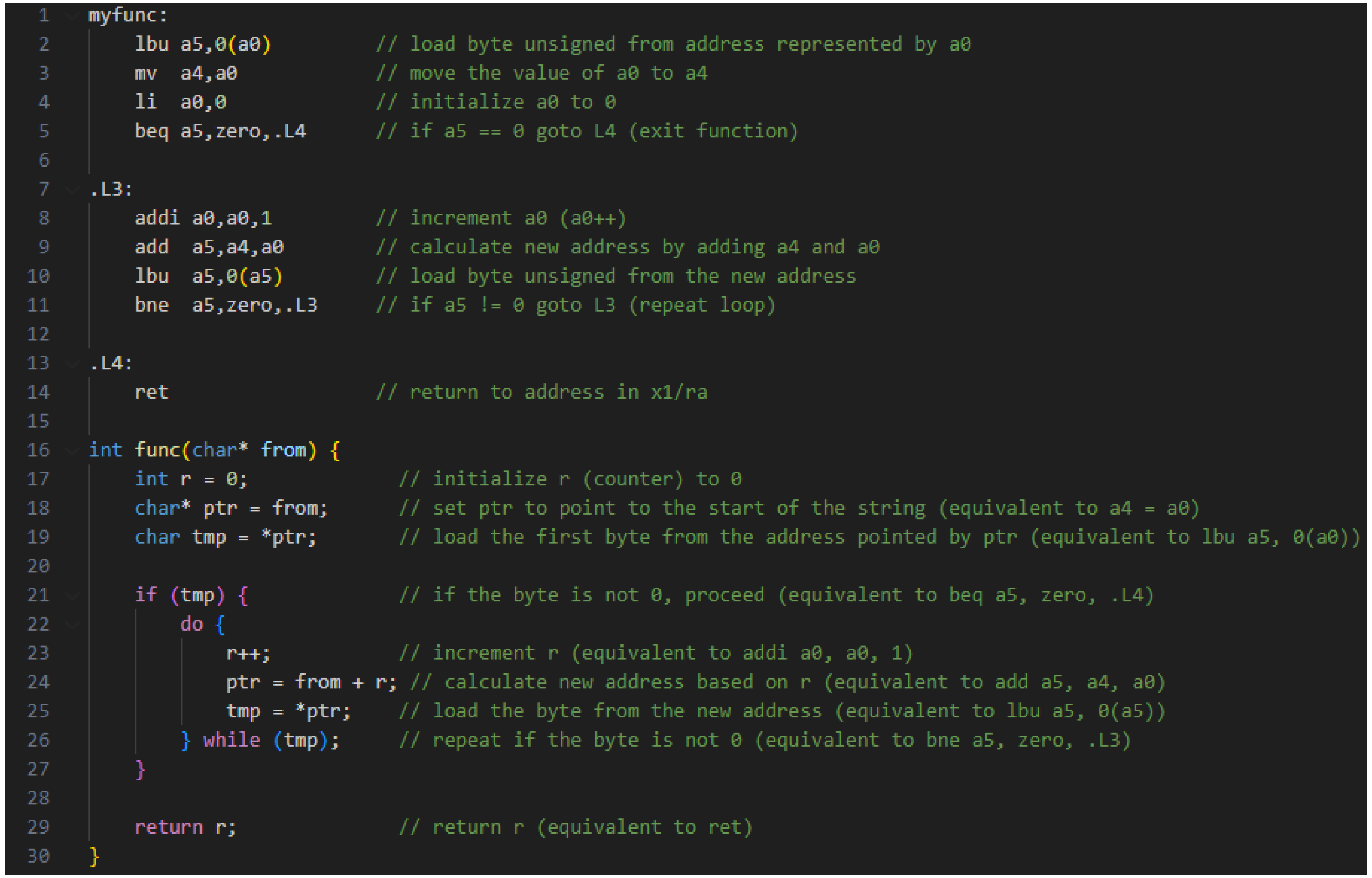
The loop are the 5 instructions from .L3 to the bne instruction which branches to .L3. We can see it forms a loop, since the control flow passes from the last instruction to the first.



```
myfunc:
    // lbu a5,0(a0)         // load byte unsigned from address represented by a0
    // mv  a4,a0            // move the value of a0 to a4
    // li  a0,0             // initialize a0 to 0
    // beq a5,zero,.L4      // if a5 == 0 goto L4 (exit function)

.L3:
    addi a0,a0,1         // increment a0 (a0++)
    add  a5,a4,a0        // calculate new address by adding a4 and a0
    lbu  a5,0(a5)        // load byte unsigned from the new address
    j .L3                // if a5 != 0 goto L3 (repeat loop)

.L4:
    // ret                  // return to address in x1/ra
```

The code could be written with j instruction.

<------

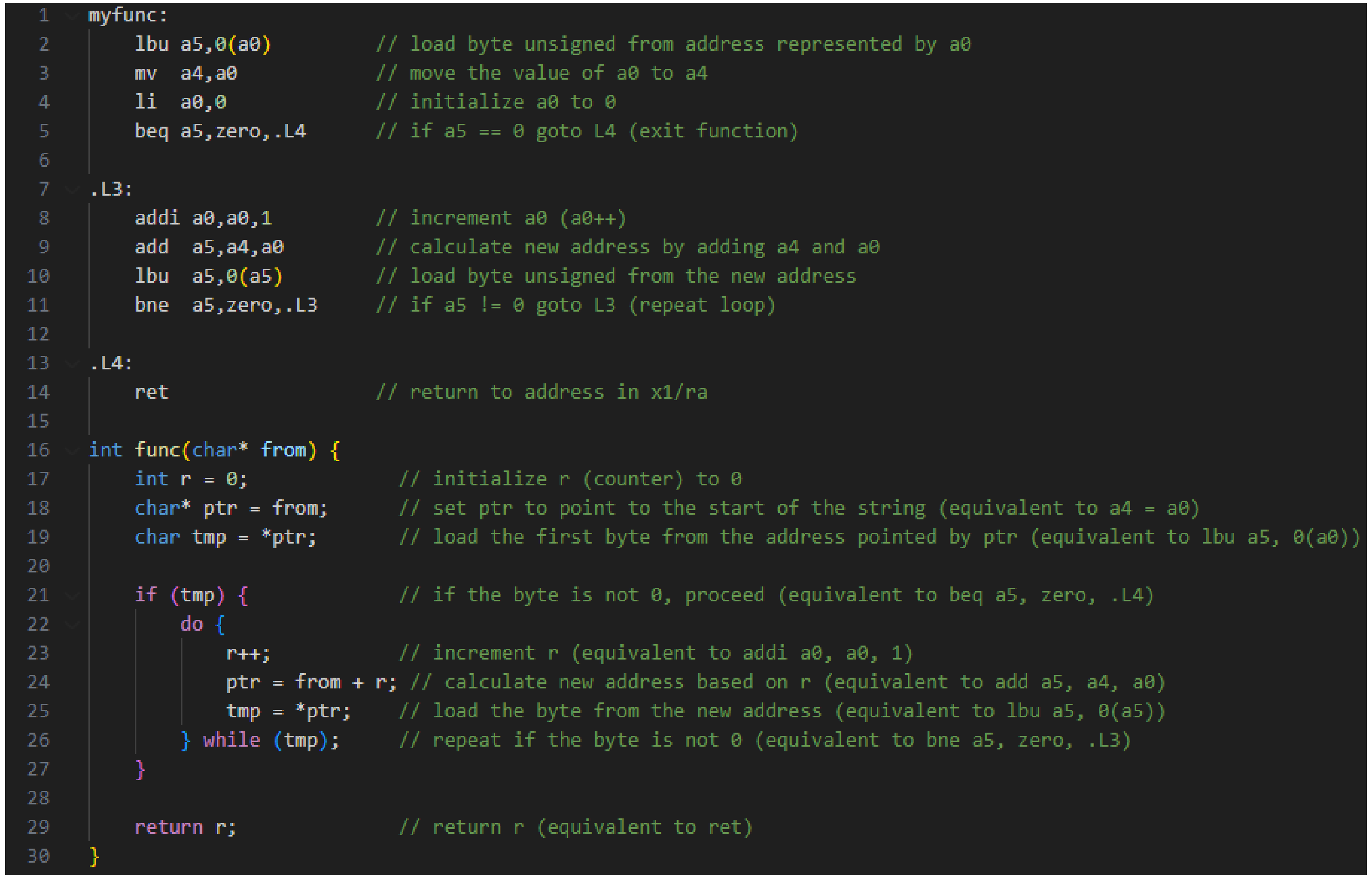


```
myfunc:
    lbu a5,0(a0)         // load byte unsigned from address represented by a0
    mv  a4,a0            // move the value of a0 to a4
    li  a0,0             // initialize a0 to 0
    beq a5,zero,.L4      // if a5 == 0 goto L4 (exit function)

.L3:
    addi a0,a0,1         // increment a0 (a0++)
    add  a5,a4,a0        // calculate new address by adding a4 and a0
    lbu  a5,0(a5)        // load byte unsigned from the new address
    bne  a5,zero,.L3     // if a5 != 0 goto L3 (repeat loop)

.L4:
    ret                  // return to address in x1/ra

int func(char* from) {
    int r = 0;             // initialize r (counter) to 0
    char* ptr = from;      // set ptr to point to the start of the string (equivalent to a4 = a0)
    char tmp = *ptr;       // load the first byte from the address pointed by ptr (equivalent to lbu a5, 0(a0))

    if (tmp) {             // if the byte is not 0, proceed (equivalent to beq a5, zero, .L4)
        do {
            r++;           // increment r (equivalent to addi a0, a0, 1)
            ptr = from + r; // calculate new address based on r (equivalent to add a5, a4, a0)
            tmp = *ptr;    // load the byte from the new address (equivalent to lbu a5, 0(a5))
        } while (tmp);     // repeat if the byte is not 0 (equivalent to bne a5, zero, .L3)
    }

    return r;              // return r (equivalent to ret)
}
```

Question 1.3.4: Rewrite the above RISC-V assembler program to a C program. The resulting program must not have a goto-style and minor syntactical mistakes are acceptable..

The argument is a char* because lbu loads a byte (size of a char)

The return value in r is an integer, which represents the count of non-zero characters in the string. Hence, the return type of the function is most likely int.


```
myfunc:
    lbu a5,0(a0)        // load byte unsigned from address represented by a0
    mv  a4,a0           // move the value of a0 to a4
    li  a0,0            // initialize a0 to 0
    beq a5,zero,.L4     // if a5 == 0 goto L4 (exit function)

.L3:
    addi a0,a0,1        // increment a0 (a0++)
    add  a5,a4,a0       // calculate new address by adding a4 and a0
    lbu  a5,0(a5)       // load byte unsigned from the new address
    bne  a5,zero,.L3    // if a5 != 0 goto L3 (repeat loop)

.L4:
    ret                 // return to address in x1/ra

int func(char* from) {
    int r = 0;              // initialize r (counter) to 0
    char* ptr = from;       // set ptr to point to the start of the string (equivalent to a4 = a0)
    char tmp = *ptr;        // load the first byte from the address pointed by ptr (equivalent to lbu a5, 0(a0))

    if (tmp) {              // if the byte is not 0, proceed (equivalent to beq a5, zero, .L4)
        do {
            r++;            // increment r (equivalent to addi a0, a0, 1)
            ptr = from + r; // calculate new address based on r (equivalent to add a5, a4, a0)
            tmp = *ptr;     // load the byte from the new address (equivalent to lbu a5, 0(a5))
        } while (tmp);      // repeat if the byte is not 0 (equivalent to bne a5, zero, .L3)
    }

    return r;               // return r (equivalent to ret)
}
```

Question 1.3.5: Describe shortly the functionality of the program.

The program calculates the length of a zero-terminated string

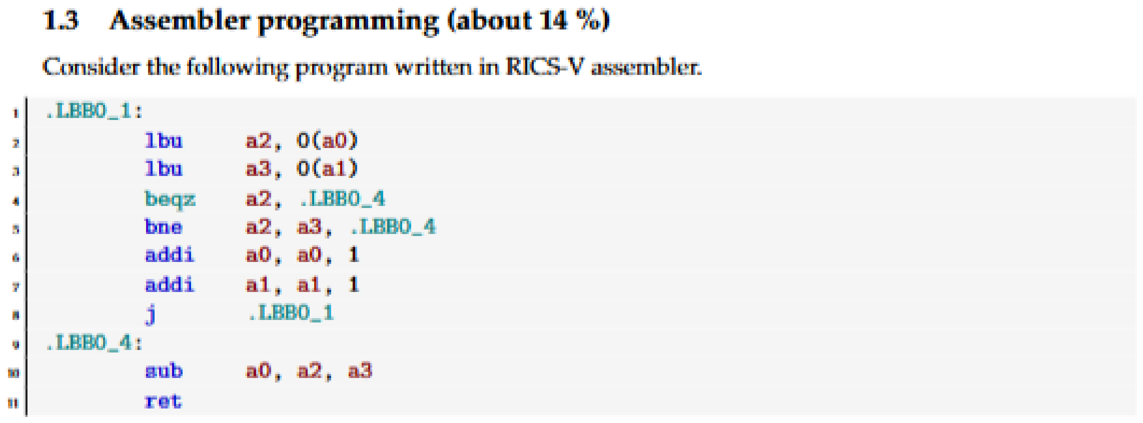
Question 1.3.6: What is The purpose of the “lui” instruction and how does it work?

The lui instruction is basically used to handle really big numbers (more than 12 bits) in RISC-V.

It works by loading a 20-bit value into the top 20 bits of a register, while the bottom 12 bits are set to zero. This gives you the "upper half" of a 32-bit number.

But to get the full 32-bit number, you usually follow it up with an addi instruction, which adds the "lower half" (the remaining 12 bits). Together, these two instructions let you work with full 32-bit constants, even though individual instructions have limits on the size of the numbers they can handle.

It's like building a big number in two steps: first setting up the top part (lui), then filling in the bottom part (addi).



1.3 Assembler programming (about 14 %)

Consider the following program written in RICS-V assembler.

```
.LBB0_1:
        lbu     a2, 0(a0)
        lbu     a3, 0(a1)
        beqz    a2, .LBB0_4
        bne     a2, a3, .LBB0_4
        addi    a0, a0, 1
        addi    a1, a1, 1
        j       .LBB0_1
.LBB0_4:
        sub     a0, a2, a3
        ret
```

Example: reexam-2023-24

Question 1.3.1: The code snippet is a function. Is this function calling other functions? Argue for your answer

Question 1.3.2: Which registers hold the functions arguments (if any)? Argue for your answer.

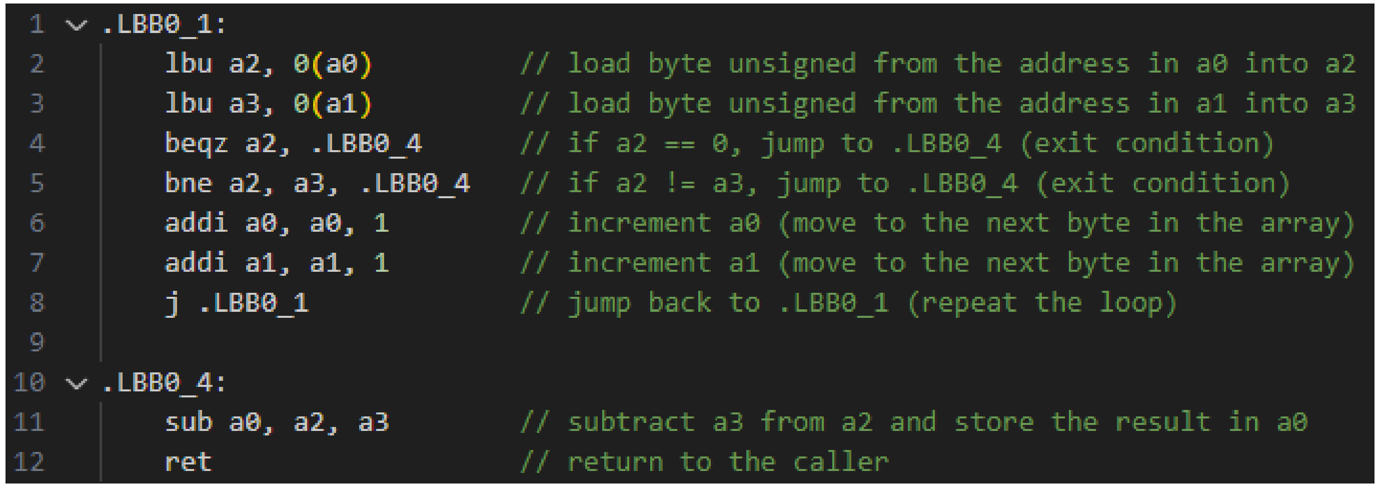
Question 1.3.3: The function contains a loop. Which instructions form the loop? Describe how you identified this.

Question 1.3.4: Rewrite the above RISC-V assembler program to a C program. The resulting program must not have a goto-style and minor syntactical mistakes are acceptable..

Question 1.3.5: Describe shortly the functionality of the program.

Question 1.3.6: What are the purpose of the “lb” and “lbu” instructions, how do they work and what is the difference between them?

Source: reexam-2023-24-form.pdf

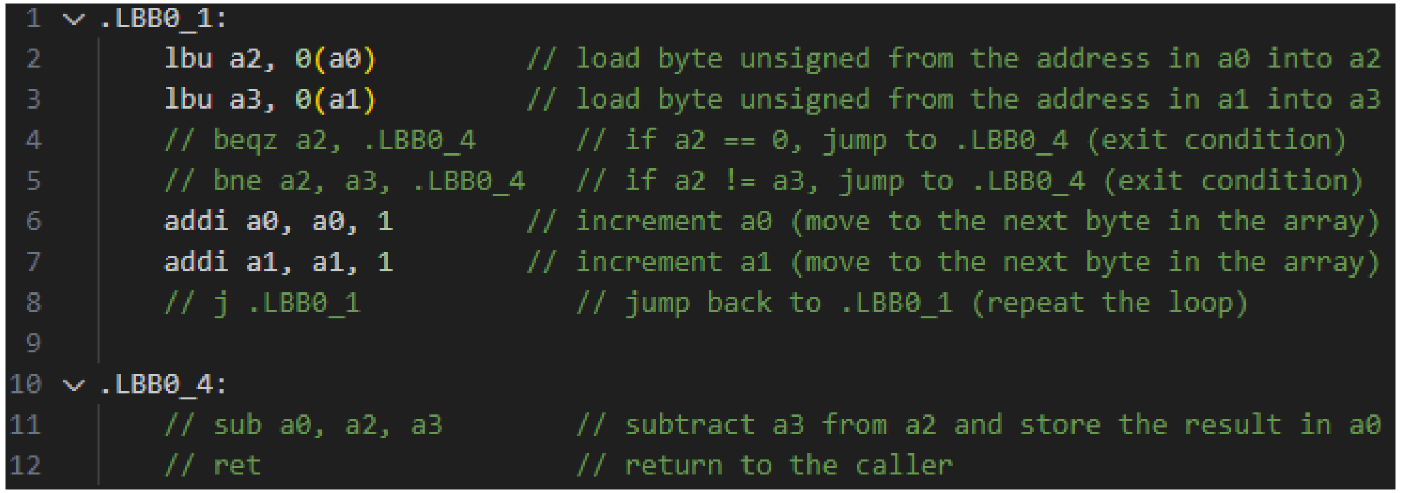


```
.LBB0_1:
    lbu a2, 0(a0)        // load byte unsigned from the address in a0 into a2
    lbu a3, 0(a1)        // load byte unsigned from the address in a1 into a3
    beqz a2, .LBB0_4     // if a2 == 0, jump to .LBB0_4 (exit condition)
    bne a2, a3, .LBB0_4  // if a2 != a3, jump to .LBB0_4 (exit condition)
    addi a0, a0, 1       // increment a0 (move to the next byte in the array)
    addi a1, a1, 1       // increment a1 (move to the next byte in the array)
    j .LBB0_1            // jump back to .LBB0_1 (repeat the loop)

.LBB0_4:
    sub a0, a2, a3       // subtract a3 from a2 and store the result in a0
    ret                  // return to the caller
```

Question 1.3.1: The code snippet is a function. Is this function calling other functions? Argue for your answer

It is **not** calling other functions.

If it were, it would have JAL or a JALR instruction that is **SAVING** the return address in x1/ra, and not only return pseudo-instructions

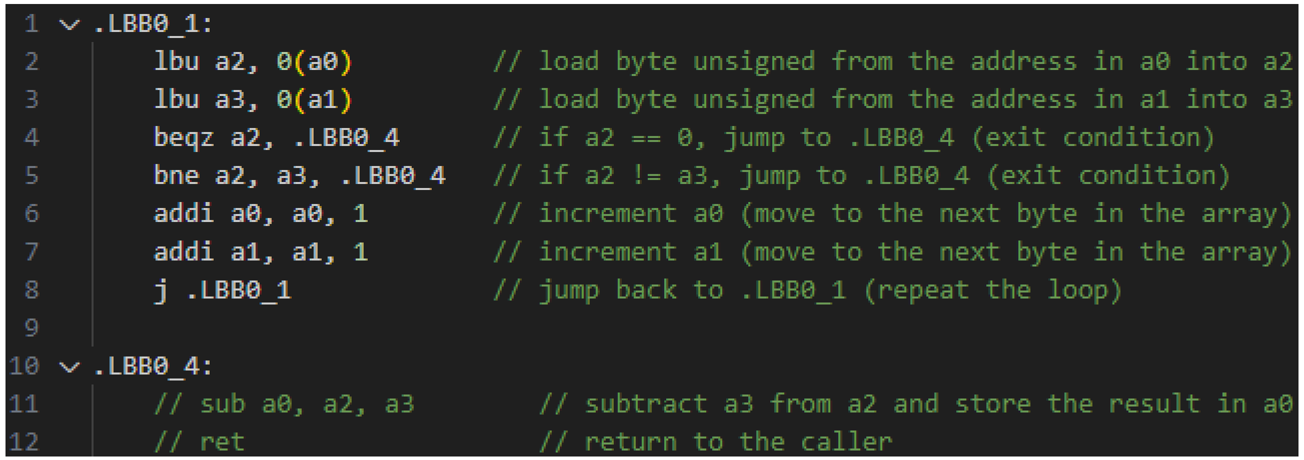


```
 1  .LBB0_1:
 2      lbu a2, 0(a0)          // load byte unsigned from the address in a0 into a2
 3      lbu a3, 0(a1)          // load byte unsigned from the address in a1 into a3
 4      // beqz a2, .LBB0_4       // if a2 == 0, jump to .LBB0_4 (exit condition)
 5      // bne a2, a3, .LBB0_4    // if a2 != a3, jump to .LBB0_4 (exit condition)
 6      addi a0, a0, 1         // increment a0 (move to the next byte in the array)
 7      addi a1, a1, 1         // increment a1 (move to the next byte in the array)
 8      // j .LBB0_1              // jump back to .LBB0_1 (repeat the loop)
 9
10  .LBB0_4:
11      // sub a0, a2, a3         // subtract a3 from a2 and store the result in a0
12      // ret                    // return to the caller
```

Question 1.3.2: Which registers hold the functions arguments (if any)? Argue for your answer.

Look for registers that are read before they are written to.

In the example, both a0 and a1 is used on line 2 and 3 before being written to on line 6 and 7.



```
.LBB0_1:
    lbu a2, 0(a0)        // load byte unsigned from the address in a0 into a2
    lbu a3, 0(a1)        // load byte unsigned from the address in a1 into a3
    beqz a2, .LBB0_4     // if a2 == 0, jump to .LBB0_4 (exit condition)
    bne a2, a3, .LBB0_4  // if a2 != a3, jump to .LBB0_4 (exit condition)
    addi a0, a0, 1       // increment a0 (move to the next byte in the array)
    addi a1, a1, 1       // increment a1 (move to the next byte in the array)
    j .LBB0_1            // jump back to .LBB0_1 (repeat the loop)

.LBB0_4:
    // sub a0, a2, a3       // subtract a3 from a2 and store the result in a0
    // ret                  // return to the caller
```

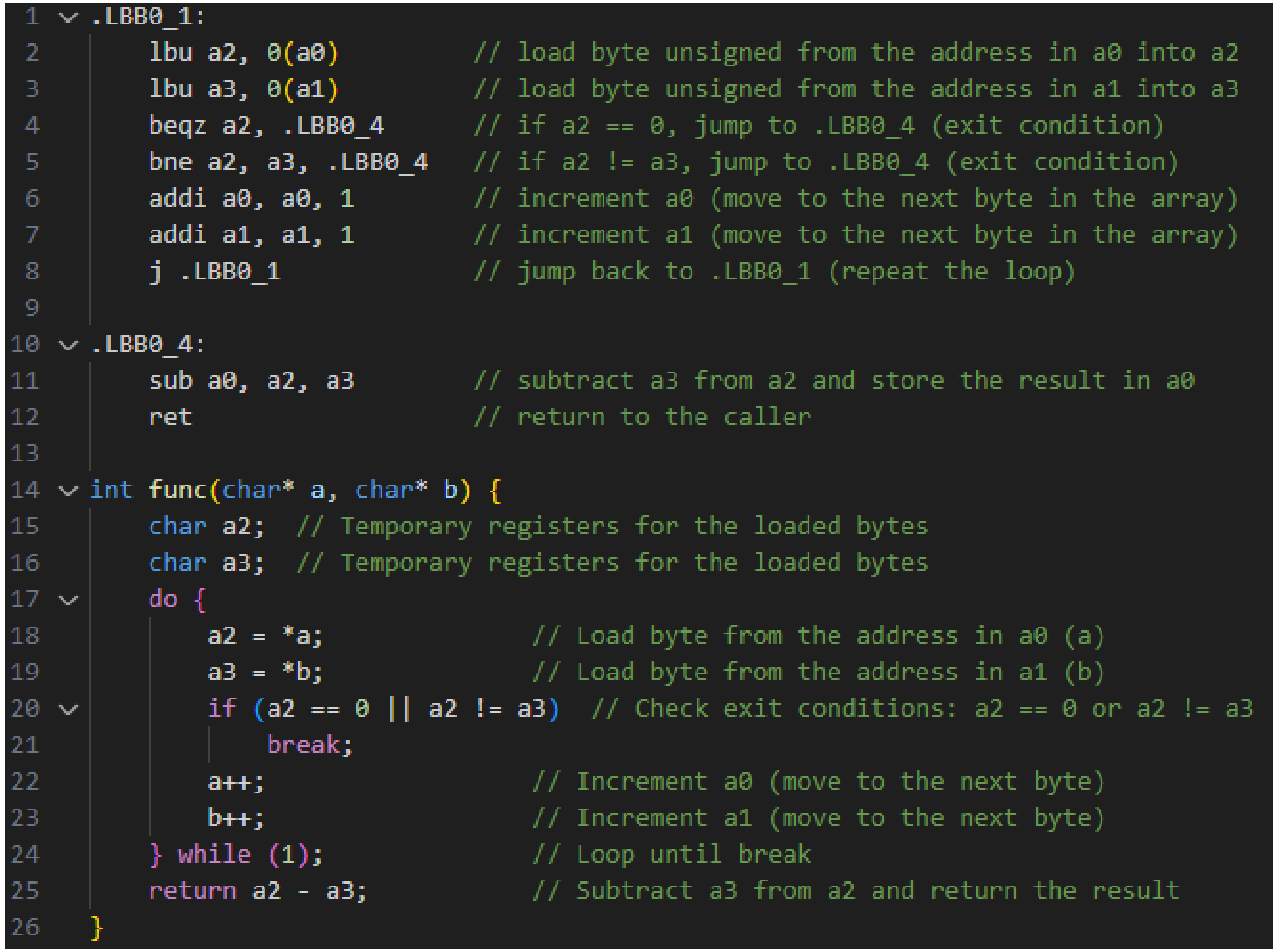
Question 1.3.3: The function contains a loop. Which instructions form the loop? Describe how you identified this.

Look for any branch instructions (slide 9), and optionally a jal or jalr instruction with a negative offset (or a label at a line less than the jump instruction).

The loop consists of the 7 instructions from .LBB0_1 to the j instruction.

We can see it forms a loop because the **j instruction** unconditionally jumps back to .LBB0_1, restarting the sequence of instructions.

The **bne & beqz instructions** are used to **break out of the loop** under certain conditions, and they do not form the loop.

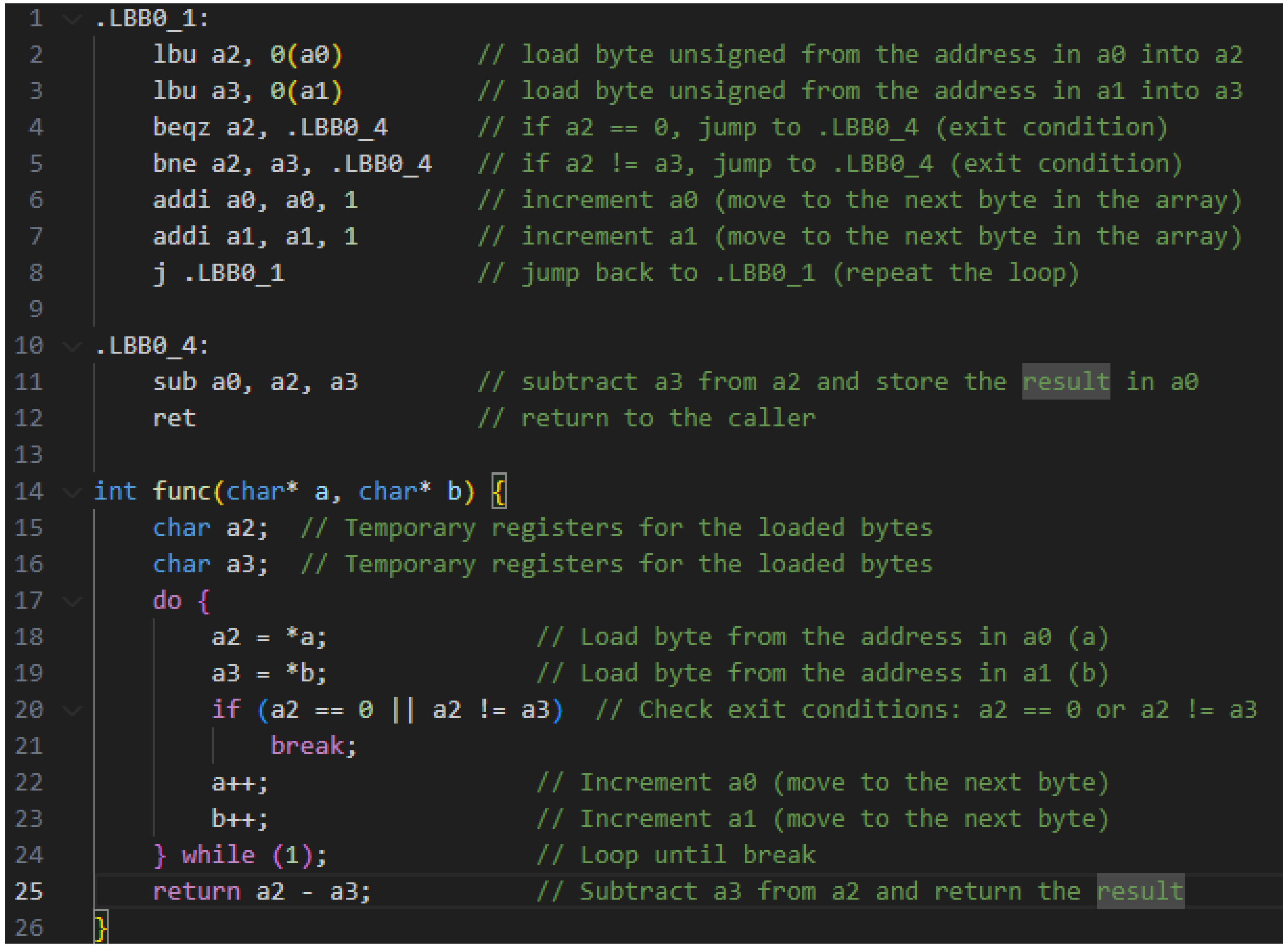


```
.LBB0_1:
    lbu a2, 0(a0)         // load byte unsigned from the address in a0 into a2
    lbu a3, 0(a1)         // load byte unsigned from the address in a1 into a3
    beqz a2, .LBB0_4      // if a2 == 0, jump to .LBB0_4 (exit condition)
    bne a2, a3, .LBB0_4   // if a2 != a3, jump to .LBB0_4 (exit condition)
    addi a0, a0, 1        // increment a0 (move to the next byte in the array)
    addi a1, a1, 1        // increment a1 (move to the next byte in the array)
    j .LBB0_1             // jump back to .LBB0_1 (repeat the loop)

.LBB0_4:
    sub a0, a2, a3        // subtract a3 from a2 and store the result in a0
    ret                   // return to the caller

int func(char* a, char* b) {
    char a2;  // Temporary registers for the loaded bytes
    char a3;  // Temporary registers for the loaded bytes
    do {
        a2 = *a;              // Load byte from the address in a0 (a)
        a3 = *b;              // Load byte from the address in a1 (b)
        if (a2 == 0 || a2 != a3)  // Check exit conditions: a2 == 0 or a2 != a3
            break;
        a++;                  // Increment a0 (move to the next byte)
        b++;                  // Increment a1 (move to the next byte)
    } while (1);              // Loop until break
    return a2 - a3;           // Subtract a3 from a2 and return the result
}
```

Question 1.3.4: Rewrite the above RISC-V assembler program to a C program. The resulting program must not have a goto-style and minor syntactical mistakes are acceptable..

The arguments are char* because lbu loads a byte (size of a char) from memory.

The return value in a0 is the result of subtracting, which makes it an integer.



```
.LBB0_1:
    lbu a2, 0(a0)         // load byte unsigned from the address in a0 into a2
    lbu a3, 0(a1)         // load byte unsigned from the address in a1 into a3
    beqz a2, .LBB0_4      // if a2 == 0, jump to .LBB0_4 (exit condition)
    bne a2, a3, .LBB0_4   // if a2 != a3, jump to .LBB0_4 (exit condition)
    addi a0, a0, 1        // increment a0 (move to the next byte in the array)
    addi a1, a1, 1        // increment a1 (move to the next byte in the array)
    j .LBB0_1             // jump back to .LBB0_1 (repeat the loop)

.LBB0_4:
    sub a0, a2, a3        // subtract a3 from a2 and store the result in a0
    ret                   // return to the caller

int func(char* a, char* b) {
    char a2;  // Temporary registers for the loaded bytes
    char a3;  // Temporary registers for the loaded bytes
    do {
        a2 = *a;                // Load byte from the address in a0 (a)
        a3 = *b;                // Load byte from the address in a1 (b)
        if (a2 == 0 || a2 != a3)  // Check exit conditions: a2 == 0 or a2 != a3
            break;
        a++;                    // Increment a0 (move to the next byte)
        b++;                    // Increment a1 (move to the next byte)
    } while (1);                // Loop until break
    return a2 - a3;             // Subtract a3 from a2 and return the result
}
```

Question 1.3.5: Describe shortly the functionality of the program.

The program compares two zero-terminated strings. It is the strcmp function from the C standard library

Returns:
0 if the two strings are equal.

A **negative value** if the first string is lexicographically less than the second string.

A **positive value** if the first string is lexicographically greater than the second string.

Question 1.3.6: What is The purpose of the “lb” and “lbu” instructions and how do they work, and what is the difference between them?

The lb and lbu instructions are used to load a single byte from memory into a register in RISC-V. They both calculate the memory address by adding a 12-bit immediate constant to a base address stored in a register (rs1). Once the address is computed, they fetch the byte at that location and place it in the lower 8 bits of the destination register (rd).

The difference between them is how they handle the upper 24 bits of the destination register:

lbu: This is ”load byte **unsigned**." It fills the upper 24 bits of the register with zeros, giving you a **zero-extended** value.

lb: This is "load byte **signed**." It copies the most significant bit (sign bit) of the loaded byte into the upper 24 bits, giving you a **sign-extended** value.

Recap

You can use the reference tables for registers, instructions, and pseudo instructions, or make your own notes.