Assembler Programming

Mandag 5. Januar 2026

Computersystemer 2025/2026

Tobias Andersen

Contents

- Basics of RISC-V Assembler Programming
- How To Identify Control Flow Structures
- Exam Example

Basics of RISC-V Assembler Programming

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Registers Calling Conventions

Helps us read and understand RISC-V code.

x1(ra) holds address which will be jumped to when returning.
x2(sp) be decreased to allocate stack space for additional arguments, especially important for recursive functions.

x10-x17(a0-a7) contains function arguments. Caller writes to these registers, and Callee reads from them.

x5-x7(t0-t2) and x28-x31(t3-t6) are used to hold intermediate values during computations.

NOTE: These are calling CONVENTIONS, not rules, they are typically followed, but not mandatory.

RV32I Base Integer Instructions

Inst	Name	Description (C)
add	ADD	rd = rs1 + rs2
sub	SUB	rd = rs1 - rs2
xor	XOR	rd = rs1 ^ rs2
or	OR	rd = rs1 | rs2
and	AND	rd = rs1 & rs2
sll	Shift Left Logical	rd = rs1 << rs2
srl	Shift Right Logical	rd = rs1 >> rs2
sra	Shift Right Arith*	rd = rs1 >> rs2
slt	Set Less Than	rd = (rs1 < rs2)?1:0
sltu	Set Less Than (U)	rd = (rs1 < rs2)?1:0
addi	ADD Immediate	rd = rs1 + imm
xori	XOR Immediate	rd = rs1 ^ imm
ori	OR Immediate	rd = rs1 | imm
andi	AND Immediate	rd = rs1 & imm
slli	Shift Left Logical Imm	rd = rs1 << imm[0:4]
srli	Shift Right Logical Imm	rd = rs1 >> imm[0:4]
srai	Shift Right Arith Imm	rd = rs1 >> imm[0:4]
slti	Set Less Than Imm	rd = (rs1 < imm)?1:0
sltiu	Set Less Than Imm (U)	rd = (rs1 < imm)?1:0
lb	Load Byte	rd = M[rs1+imm][0:7]
lh	Load Half	rd = M[rs1+imm][0:15]
lw	Load Word	rd = M[rs1+imm][0:31]
lbu	Load Byte (U)	rd = M[rs1+imm][0:7]
lhu	Load Half (U)	rd = M[rs1+imm][0:15]
sb	Store Byte	M[rs1+imm][0:7] = rs2[0:7]
sh	Store Half	M[rs1+imm][0:15] = rs2[0:15]
sw	Store Word	M[rs1+imm][0:31] = rs2[0:31]
beq	Branch ==	if(rs1 == rs2) PC += imm
bne	Branch !=	if(rs1 != rs2) PC += imm
blt	Branch <	if(rs1 < rs2) PC += imm
bge	Branch ≥	if(rs1 >= rs2) PC += imm
bltu	Branch < (U)	if(rs1 < rs2) PC += imm
bgeu	Branch ≥ (U)	if(rs1 >= rs2) PC += imm
jal	Jump And Link	rd = PC+4; PC += imm
jalr	Jump And Link Reg	rd = PC+4; PC = rs1 + imm
lui	Load Upper Imm	rd = imm << 12
auipc	Add Upper Imm to PC	rd = PC + (imm << 12)
ecall	Environment Call	Transfer control to OS
ebreak	Environment Break	Transfer control to debugger

Instructions

RV32I Base Interger & RV32M Interger Multiplication and division

- rd is destination register
- rs1 and rs2 are source registers
- imm is immediate value (constant)

RV32M Integer Multiplication and division

Inst	Name	Description (C)
mul	MUL	rd = (rs1 * rs2)[31:0]
mulh	MUL High	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	rd = (rs1 * rs2)[63:32]
div	DIV	rd = rs1 / rs2
divu	DIV (U)	rd = rs1 / rs2
rem	Remainder	rd = rs1 % rs2
remu	Remainder (U)	rd = rs1 % rs2

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 $\neq$ 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 $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ 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 $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, 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 x1, offset[31:12]` `jalr x1, x1, 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

Pseudo Instructions

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

How To Identify Control Flow Structures

Function Calls

- 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 functions

- Functions can be called with the pseudo instructions:
  - J offset : jump
  - JAL offset : jump and link
  - JR rs : jump register
  - JALR rs : jump and link register
  - CALL offset : jump and link to far away address

- RET (JALR x0, x1, 0) returns from a function.

Function Calling

- Registers a0-7 are used to store arguments.
- Argument registers are recognized by using the register in function without first having written to it in that function.

- Caller
  - Passes arguments to Callee, by loading registers a0-a7
  - Jumps to Callee (JAL, JALR, etc.)

- Callee
  - Performs function (read arguments from a0-7)
  - Returns results to Caller (if any) in a0-1
  - Returns to point of call (RET)

Function Calls

C Code

```
add(a, b) {
    return a + b;
}

main() {
    x = add(3, 4);
}
```

RISC-V Code

```
main:
ADDI a0, zero, 3     # first argument
ADDI a1, zero, 4     # second argument
JAL  ra, add         # call add(3, 4)
...
add:
ADD  a0, a0, a1      # a0 = a0 + a1
RET                  # return to caller
```

Conditional Statements

- If-statements, While-loop, For-loop
- Are usually denoted with any branch instruction:

beq	Branch ==	if(rs1 == rs2) PC += imm
bne	Branch !=	if(rs1 != rs2) PC += imm
blt	Branch <	if(rs1 < rs2) PC += imm
bge	Branch ≥	if(rs1 >= rs2) PC += imm
bltu	Branch < (U)	if(rs1 < rs2) PC += imm
bgeu	Branch ≥ (U)	if(rs1 >= rs2) PC += 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

- Identify loops by a jump or branch which goes backwards to an earlier label.

If Statement

C Code

```
If (a == b) {
    a = a + c;
}
a =  a – c;
```

RISC-V Code

```
#t0 = a, t1 = b, t2 = c

BNE t0, t1, L1        # if a != b, go to L1
ADD t0, t0, t2        # a = a + c
L1:
SUB t0, t0, t2        # a = a - c
```

Assembly tests opposite case of (a != b) of high-level code (a == b)

If/Else Statement

C Code

```
If (a == b) {
    a = a + c;
}
else {
    a =  a – c;
}
```

RISC-V Code

```
#t0 = a, t1 = b, t2 = c

BNE t0, t1, ELSE    # if a != b, go to ELSE
ADD t0, t0, t2      # a = a + c
J DONE              # skip ELSE
ELSE:
SUB t0, t0, t2      # a = a – c
DONE:
```

Assembly tests opposite case of (a != b) of high-level code (a == b)

While Loop

C Code

```
pow = 1;
x = 0;

While (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

RISC-V Code

```
#t0 = pow, t1 = x

ADDI t0, zero, 1        # pow = 1
ADDI t1, zero, 0        # x = 0
ADDI t2, zero, 128      # constant 128
WHILE:
BEQ  t0, t2, DONE       # while pow != 128
SLLI t0, t0, 1          # pow = pow * 2
ADDI t1, t1, 1          # x = x + 1
J WHILE
DONE:
```

Do-While Loop

C Code

```
pow = 1;
x = 0;

do {
    pow = pow * 2;
    x = x + 1;
} while (pow != 128);
```

RISC-V Code

```
#t0 = pow, t1 = x

ADDI t0, zero, 1      # pow = 1
ADDI t1, zero, 0      # x = 0
ADDI t2, zero, 128    # constant 128

DO:
SLLI t0, t0, 1        # pow = pow * 2
ADDI t1, t1, 1        # x = x + 1
BNE  t0, t2, DO       # while pow != 128
```

For Loop

C Code

```
int sum = 0;
int i = 0;

For (i = 0; i != 10; i++) {
    sum = sum + i
}
```

RISC-V Code

```
#t0 = sum, t1 = i

ADDI t0, zero, 0        # sum = 0
ADDI t1, zero, 0        # i = 0
ADDI t2, zero, 10       # constant 10
FOR:
BEQ  t0, t2, DONE       # while (pow != 128)
ADD t0, t0, t1          # sum = sum + i
ADDI t1, t1, 1          # i++
J FOR
DONE:
```

Exam Example

Assembler programming

Consider the following program written in RISC-V assembler.

```
1  myfunc:
2          lbu     a5,0(a0)
3          mv      a4,a0
4          li      a0,0
5          beq     a5,zero,.L4
6  .L3:
7          addi    a0,a0,1
8          add     a5,a4,a0
9          lbu     a5,0(a5)
10         bne     a5,zero,.L3
11         ret
12 .L4:
13         ret
```

Example: exam-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 is The purpose of the “lui” instruction and how does it work?

Source: exam-2023-24-form.pdf

```
1   myfunc:
2       lbu a5,0(a0)          // load byte unsigned from address represented by a0
3       mv  a4,a0             // move the value of a0 to a4
4       li  a0,0              // initialize a0 to 0
5       beq a5,zero,.L4       // if a5 == 0 goto L4 (exit function)
6
7   .L3:
8       addi a0,a0,1          // increment a0 (a0++)
9       add  a5,a4,a0         // calculate new address by adding a4 and a0
10      lbu  a5,0(a5)         // load byte unsigned from the new address
11      bne  a5,zero,.L3      // if a5 != 0 goto L3 (repeat loop)
12
13  .L4:
14      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
    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).

RECAP DONE!

Find this recap and useful links on github.