Machine Architecture

Assembler Programming

Fredag 12. Januar 2024

Computersystemer 2023/2024

Christian Franck

Outline

- Registers
- RV32I Base Integer Instructions
- RV32M Integer Multiplication and Division
- Pseudo Instructions
- Calling Functions
- Function Arguments
- While-loop, For-loop, If-Statement
- Reverse engineering RISC-V to C
- Example: exam-2022-23
- Example: reexam-2022-23

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

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

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	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`

RV32M Integer Multiplication and Division

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

1.3 Assembler programming (about 14 %)

Consider the following program written in RICS-V assembler.

```
1  myfunc:
2          lbu     a5,0(a0)
3          beq     a5,zero,.L2
4  .L3:
5          sb      a5,0(a1)
6          lbu     a5,1(a0)
7          addi    a0,a0,1
8          addi    a1,a1,1
9          bne     a5,zero,.L3
10 .L2:
11         sb      zero,0(a1)
12         ret
```

Example: exam-2022-23

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: Descripe shortly the functionality of the program.

Question 1.3.6: What is a pseudo-instruction and give an example of a RISC-V pseudo-instruction?

Source: exam-2022-23-form.pdf

```
1   myfunc:
2       lbu a5,0(a0)            //load byte unsigned from address represented by a0
3       beq a5,zero,.L2         // if a5 == 0 goto L2 (ie if the value is 0, return)
4   .L3:
5       lb a5,0(a1)             //load byte
6       lbu a5,1(a0)            //load byte unsigned
7       addi a0,a0,1            //a0 ++
8       addi a1,a1,1            //a1 ++
9       bne a5,zero,.L3         //if a5 != 0 goto L3 -> repeat
10  .L2:
11      sb zero,0(a1)           //store 0 byte at the address represented by a1
12      ret                     // return to addr x1/ra
```

Example: exam-2022-23

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

Label .L3 and .L2 are considered part of myfunc.

Apart from *ret*, here are no *jalr* or *jal* instructions. Remember to look for the pseudo instructions *j, jr, call* (look at slide 7).

So no functions are called in myfunc.

Source: exam-2022-23-form.pdf

```
1   myfunc:
2       lbu a5,0(a0)            //load byte unsigned from address represented by a0
3       //beq a5,zero,.L2         // if a5 == 0 goto L2 (ie if the value is 0, return)
4   .L3:
5       lb a5,0(a1)             //load byte
6       lbu a5,1(a0)            //load byte unsigned
7       //addi a0,a0,1            //a0 ++
8       //addi a1,a1,1            //a1 ++
9       //bne a5,zero,.L3         //if a5 != 0 goto L3 -> repeat
10  .L2:
11      //sb zero,0(a1)           //store 0 byte at the address represented by a1
12      //ret                     // return to addr x1/ra
```

Example: exam-2022-23

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, a0 and a1 are both used on line 2, 5, and 6 respectively before written to on line 7 and 8.

Solution text: "The arguments are in a0 and a1, since these registers are used in the function, but never written"

a0 and a1 are written, but after they are read.

Source: exam-2022-23-form.pdf

```
1   myfunc:
2       //lbu a5,0(a0)            //load byte unsigned from address represented by a0
3       //beq a5,zero,.L2         // if a5 == 0 goto L2 (ie if the value is 0, return)
4   .L3:
5       lb a5,0(a1)             //load byte
6       lbu a5,1(a0)            //load byte unsigned
7       addi a0,a0,1            //a0 ++
8       addi a1,a1,1            //a1 ++
9       bne a5,zero,.L3         //if a5 != 0 goto L3 -> repeat
10  .L2:
11      //sb zero,0(a1)           //store 0 byte at the address represented by a1
12      //ret                     // return to addr x1/ra
```

Example: exam-2022-23

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).

```
17  myfunc:
18      //lbu a5,0(a0)            //load byte unsigned from address represented by a0
19  .L3:
20      beq a5,zero,.L2         // if a5 == 0 goto L2 (ie if the value is 0, return)
21      lb a5,0(a1)             //load byte
22      lbu a5,1(a0)            //load byte unsigned
23      addi a0,a0,1            //a0 ++
24      addi a1,a1,1            //a1 ++
25      j .L3                   //if a5 != 0 goto L3 -> repeat
26  .L2:
27      //sb zero,0(a1)           //store 0 byte at the address represented by a1
28      //ret                     // return to addr x1/ra
```

The code could be written with *j* instruction. <------

Source: exam-2022-23-form.pdf

```
25  // myfunc:
26  //     lbu a5,0(a0)            // unsigned char* a5 = *a0 (a0 is the address of a first byte/char))
27  //     beq a5,zero,.L2         // if a5 != 0 -> loop
28  // .L3:
29  //     sb a5,0(a1)             // *a1 = a5 (a1 is the address of ANOTHER first byte/char)
30  //     lbu a5,1(a0)            // unsigned char* a5 = *(a0+1) (a0 is the address of a first byte/char))
31  //     addi a0,a0,1            // a0 ++
32  //     addi a1,a1,1            // a1 ++
33  //     bne a5,zero,.L3         // repeat check beq. if a5 != 0 -> loop
34  // .L2:
35  //     sb zero,0(a1)           // *a1 = 0;
36  //     ret                      // return to addr x1/ra
37
38  // a0 = from, a1 = to
39  char* myfunc_correct(char* a0, char* a1) {
40      unsigned char a5 = *a0;
41      while (a5 != 0) {
42          *a1 = a5;
43          a5 = *(a0+1);
44          a0 ++;
45          a1 ++;
46      }
47      *a1 = 0;
48      return a0;
49  }
```

Example: exam-2022-23

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 return type in a0 or a1 is a char* hence the type that is returned is most likely this.

void can also be returned as a default.

Source: exam-2022-23-form.pdf

Example: exam-2022-23

Question 1.3.5: Descripe shortly the functionality of the program.

a program that copies a string from a pointer to another pointer. This can both be in memory or on the stack depending on the arguments given.

Source: exam-2022-23-form.pdf

Example: exam-2022-23

Question 1.3.6: What is a pseudo-instruction and give an example of a RISC-V pseudo-instruction?

A pseudo instruction is an instruction handled by the assembler by translating it into one or more real (non-pseudo) instructions. A simple example is beqz Xn,Label (branch equal to zero), which is implemented as beq X0,Xn,Label

(Slide 6)

Source: exam-2022-23-form.pdf

1.2 Assembler programming (about 14 %)

Consider the following program written in RICS-V assembler.

```
1  func:
2          addi    sp,sp,-16
3          sw      ra,12(sp)
4          sw      s0,8(sp)
5          sw      s1,4(sp)
6          mv      s0,a0
7          li      a5,1
8          bgt     a0,a5,.L4
9  .L2:
10         mv      a0,s0
11         lw      ra,12(sp)
12         lw      s0,8(sp)
13         lw      s1,4(sp)
14         addi    sp,sp,16
15         jr      ra
16 .L4:
17         addi    a0,a0,-1
18         call    func
19         mv      s1,a0
20         addi    a0,s0,-2
21         call    func
22         add     s0,s1,a0
23         j       .L2
```

Example: reexam-2022-23

Question 1.2.1: The code snippet is a function. Does this function contain a loop? Why/Why not? It does not contain a loop.

There is only one backward jump, placed at the end of the block .L4. This jumps to block .L2, but block .L2 ends with a return, so there is no path back into .L4

Question 1.2.2: Does this function call a function? Why/Why not?
It does call a function. This can be seen by the 2 call instruction in the last block

Question 1.2.3: This function contains an if-statement. Which of the instructions belong to the if-statement? Explain why?
The instructions in the last block (starting with .L4) is the body of an if statement. The conditional jump at the end of the first block determines whether the if statement is true or not

Question 1.2.4: Explain in a single sentence the role of the "ra" register in the code. The "ra" register holds the return address

Question 1.2.5: Explain in a single sentence the role of the "a0" register in the code. The "a0" register holds the first argument into the function as well as the return value from function.

Question 1.2.6: Explain the role of the "s0" and "s1" registers in the code. These registers are callee-saves and used to hold temporaries with a function body

Question 1.2.7: What is the purpose of the 3 sw instructions in the beginning of the function and the corresponding 3 lw instructions in the middle? These instructions adds/removes a stack frame and saves/loads register values from the frame

Question 1.2.8: 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. int func(int a) if (a < 2) return a; else return func(a-1) + func(a-2);

Source: reexam-2022-23-form.pdf

```
1   func:                   // 1. loop call?
2       //addi sp,sp,-16    // add instruction to allocate stack space  16 bytes
3       //sw ra,12(sp)      // save return address
4       //sw s0,8(sp)       // save s0
5       //sw s1,4(sp)       // save s1
6       //mv s0,a0          // s0 = a0
7       //li a5,1           // a5 = 1
8       bgt a0,a5,.L4     // if a0 > a5 (1), jump to .L4
9   .L2:
10      //mv a0,s0          // a0 = s0
11      //lw ra,12(sp)      // restore return address
12      //lw s0,8(sp)       // restore s0
13      //lw s1,4(sp)       // restore s1
14      //addi sp,sp,16     // deallocate stack space 16 bytes
15      jr ra             // return
16  .L4:
17      //addi a0,a0,-1     // a0 = a0 - 1
18      call func         // call func (auipc and jalr)
19      //mv s1,a0          // s1 = a0
20      //addi a0,s0,-2     // a0 = s0 - 2
21      call func         // call func (auipc and jalr)
22      //add s0,s1,a0      // s0 = s1 + a0
23      j .L2             // jump to .L2
```

Example: reexam-2022-23

Question 1.2.1: The code snippet is a function. Does this function contain a loop? Why/Why not?
It does not contain a loop.

1 forward jump at line 8.

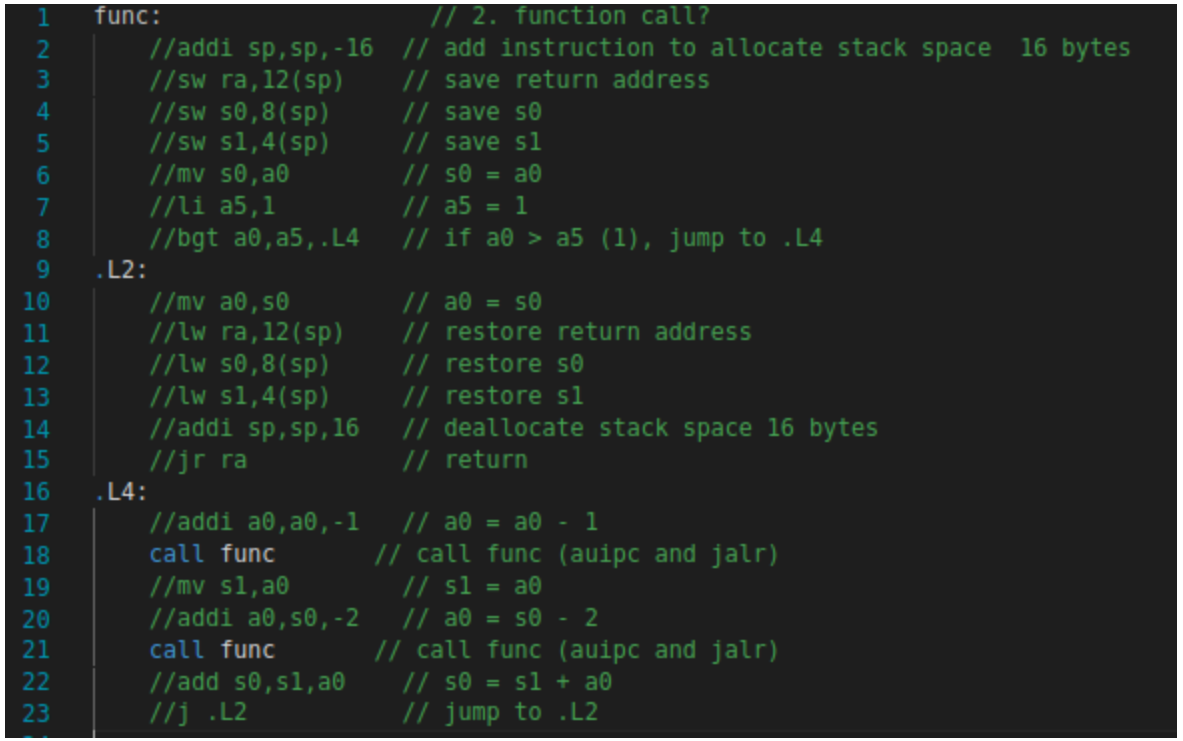
1 backward jump at line 23 to .L2

.L2 ends with a return.

call to function on 18 and 21, these also return in the end.

therefore no loop, but it does look like recursion.

Source: reexam-2022-23-form.pdf



```
1  func:                    // 2. function call?
2      //addi sp,sp,-16   // add instruction to allocate stack space  16 bytes
3      //sw ra,12(sp)     // save return address
4      //sw s0,8(sp)      // save s0
5      //sw s1,4(sp)      // save s1
6      //mv s0,a0         // s0 = a0
7      //li a5,1          // a5 = 1
8      //bgt a0,a5,.L4    // if a0 > a5 (1), jump to .L4
9  .L2:
10     //mv a0,s0         // a0 = s0
11     //lw ra,12(sp)     // restore return address
12     //lw s0,8(sp)      // restore s0
13     //lw s1,4(sp)      // restore s1
14     //addi sp,sp,16    // deallocate stack space 16 bytes
15     //jr ra            // return
16 .L4:
17     //addi a0,a0,-1    // a0 = a0 - 1
18     call func        // call func (auipc and jalr)
19     //mv s1,a0         // s1 = a0
20     //addi a0,s0,-2    // a0 = s0 - 2
21     call func        // call func (auipc and jalr)
22     //add s0,s1,a0     // s0 = s1 + a0
23     //j .L2            // jump to .L2
```

Example: reexam-2022-23

Question 1.2.2: Does this function call a function? Why/Why not?

calls function func on line 18 and 21.

Source: reexam-2022-23-form.pdf

```
1  func:                    // 3. if statement and if body?
2      //addi sp,sp,-16   // add instruction to allocate stack space  16 bytes
3      //sw ra,12(sp)     // save return address
4      //sw s0,8(sp)      // save s0
5      //sw s1,4(sp)      // save s1
6      //mv s0,a0         // s0 = a0
7      //li a5,1          // a5 = 1
8      bgt a0,a5,.L4    // if a0 > a5 (1), jump to .L4
9  .L2:
10     //mv a0,s0         // a0 = s0
11     //lw ra,12(sp)     // restore return address
12     //lw s0,8(sp)      // restore s0
13     //lw s1,4(sp)      // restore s1
14     //addi sp,sp,16    // deallocate stack space 16 bytes
15     //jr ra            // return
16 .L4:
17     addi a0,a0,-1    // a0 = a0 - 1
18     call func        // call func (auipc and jalr)
19     mv s1,a0         // s1 = a0
20     addi a0,s0,-2    // a0 = s0 - 2
21     call func        // call func (auipc and jalr)
22     add s0,s1,a0     // s0 = s1 + a0
23     j .L2            // jump to .L2
```

Example: reexam-2022-23

Question 1.2.3: This function contains an if-statement. Which of the instructions belong to the if-statement? Explain why?

line 8 is the if statement.
.L4 contains the body/block of the if-statement (line 17-23).

Source: reexam-2022-23-form.pdf

```
1   func:                   // 4. return addr ?
2       //addi sp,sp,-16  // add instruction to allocate stack space  16 bytes
3       sw ra,12(sp)    // save return address
4       //sw s0,8(sp)     // save s0
5       //sw s1,4(sp)     // save s1
6       //mv s0,a0        // s0 = a0
7       //li a5,1         // a5 = 1
8       //bgt a0,a5,.L4   // if a0 > a5 (1), jump to .L4
9   .L2:
10      //mv a0,s0        // a0 = s0
11      lw ra,12(sp)    // restore return address
12      //lw s0,8(sp)     // restore s0
13      //lw s1,4(sp)     // restore s1
14      //addi sp,sp,16   // deallocate stack space 16 bytes
15      jr ra           // return
16  .L4:
17      //addi a0,a0,-1   // a0 = a0 - 1
18      call func       // call func (auipc and jalr) saves pc + 4 in ra
19      //mv s1,a0        // s1 = a0
20      //addi a0,s0,-2   // a0 = s0 - 2
21      call func       // call func (auipc and jalr) saves pc + 4 in ra
22      //add s0,s1,a0    // s0 = s1 + a0
23      //j .L2           // jump to .L2
```

Example: reexam-2022-23

Question 1.2.4: Explain in a single sentence the role of the "ra" register in the code.

The "ra" register holds the return address

call func, stores pc + 4 in ra.

jr ra, jumps to ra

Source: reexam-2022-23-form.pdf

Example: reexam-2022-23

Question 1.2.5: Explain in a single sentence the role of the "a0" register in the code.

The "a0" register holds the first argument into the function as well as the return value from function. (Slide 3)

Source: reexam-2022-23-form.pdf

Example: reexam-2022-23

Question 1.2.6: Explain the role of the "s0" and "s1" registers in the code.

These registers are callee-saves and used to hold temporaries with a function body (Slide 3)

Source: reexam-2022-23-form.pdf

```
1  func:                    // 7. sw and lw ?
2      //addi sp,sp,-16  // add instruction to allocate stack space  16 bytes
3      sw ra,12(sp)     // save return address
4      sw s0,8(sp)      // save s0
5      sw s1,4(sp)      // save s1
6      //mv s0,a0         // s0 = a0
7      //li a5,1          // a5 = 1
8      //bgt a0,a5,.L4    // if a0 > a5 (1), jump to .L4
9  .L2:
10     //mv a0,s0         // a0 = s0
11     lw ra,12(sp)     // restore return address
12     lw s0,8(sp)      // restore s0
13     lw s1,4(sp)      // restore s1
14     //addi sp,sp,16    // deallocate stack space 16 bytes
15     //jr ra            // return
16 .L4:
17     //addi a0,a0,-1    // a0 = a0 - 1
18     //call func        // call func (auipc and jalr) saves pc + 4 in ra
19     //mv s1,a0         // s1 = a0
20     //addi a0,s0,-2    // a0 = s0 - 2
21     //call func        // call func (auipc and jalr) saves pc + 4 in ra
22     //add s0,s1,a0     // s0 = s1 + a0
23     //j .L2            // jump to .L2
```

Example: reexam-2022-23

Question 1.2.7: What is the purpose of the 3 sw instructions in the beginning of the function and the corresponding 3 lw instructions in the middle?

lw loads values from the stack into registers

sw stores values from registers unto the stack

The stack is expanded on line 2.
The stack is retracted on line 14.

sp is callee saved and (line 2 and 14) necessary otherwise values on the stack may be overwritten.

Source: reexam-2022-23-form.pdf

```
1  func:
2      addi sp,sp,-16  // add instruction to allocate stack space  16 bytes
3      sw ra,12(sp)    // save return address
4      sw s0,8(sp)     // save s0
5      sw s1,4(sp)     // save s1
6      mv s0,a0        // s0 = a0
7      li a5,1         // a5 = 1
8      bgt a0,a5,.L4   // if a0 > a5 (1), jump to .L4
9  .L2:
10     mv a0,s0        // a0 = s0
11     lw ra,12(sp)    // restore return address
12     lw s0,8(sp)     // restore s0
13     lw s1,4(sp)     // restore s1
14     addi sp,sp,16   // deallocate stack space 16 bytes
15     jr ra           // return
16 .L4:
17     addi a0,a0,-1   // a0 = a0 - 1
18     call func       // call func (auipc and jalr)
19     mv s1,a0        // s1 = a0
20     addi a0,s0,-2   // a0 = s0 - 2
21     call func       // call func (auipc and jalr)
22     add s0,s1,a0    // s0 = s1 + a0
23     j .L2           // jump to .L2
```

```
26 int func(int a0){
27     int s0 = a0;
28
29     int a5 = 1;
30
31     // .L4
32     if (a0 > a5){
33         int tmp = a0 - 1;
34         // call func , mv s1, a0
35         int s1 = func(tmp);
36
37         a0 = a0 - 2;
38         a0 = func(a0);
39
40         // add s0, s1, a0
41         s0 = s1 + a0;
42     }
43     // implicit j. L2
44     // .L2
45     a0 = s0;
46     return a0;
47 }
```

Example: reexam-2022-23

Question 1.2.8: 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.

int func(int a) if (a < 2) return a; else return func(a-1) + func(a-2);

Source: reexam-2022-23-form.pdf

Recap^2

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