CS 61C — RISC-V Assembly, Functions

Spring 2023 — Discussion 4

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

1.2 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` (offset 0 from the value in register `a0`) and execute instructions from there.

1.3 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

1.4 After calling a function and having that function return, the `t` registers may have been changed during the execution of the function, while `a` registers cannot.

1.5 In order to use the saved registers (`s0-s11`) in a function, we must store their values before using them and restore their values before returning.

1.6 The stack should only be manipulated at the beginning and end of functions, where the callee saved registers are temporarily saved.

2 Arrays in RISC-V

Comment what each code block does. Each block runs in isolation. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst`, whose first element is located at address `0xABCD0000`. Let `s0` contain `arr`'s address `0xBFFFFF00`, and let `s1` contain `lst`'s address `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed. Assume that `lst`'s last node's `next` is a NULL pointer to memory address `0x00000000`.

```
struct ll {
    int val;
    struct ll* next;
}
```

2.1

```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

2.2

```
loop: beq  s1, x0, end
      lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s1)
      lw   s1, 4(s1)
      jal  x0, loop
 end:
```

2.3

```
       add  t0, x0, x0
loop:  slti t1, t0, 6
       beq  t1, x0, end
       slli t2, t0, 2
       add  t3, s0, t2
       lw   t4, 0(t3)
       sub  t4, x0, t4
       sw   t4, 0(t3)
       addi t0, t0, 1
       jal  x0, loop
 end:
```

3 Memory Access

Using the given instructions and the sample memory arrays provided, what will happen when the RISC-V code is executed? For load instructions (`lw, lb, lh`), write out what each register will store. For store instructions (`sw, sh, sb`), update the memory array accordingly. Recall that RISC-V is little-endian and byte addressable.

3.1

```
li x5 0x00FF0000
lw x6 0(x5)
addi x5 x5 4
lh x7 2(x5)
lw x8 0(x6)
lb x9 3(x7)
```

0xFFFFFFFF	
	...
0x00FF0004	0x000C561C
0x00FF0000	36
	...
0x00000036	0xFDFDFDFD
0x00000024	0xDEADB33F
	...
0x0000000C	0xC5161C00
	...
0x00000000	

What value does each register hold after the code is executed?

3.2

```
li x5 0xABADCAFE
li x6 0xF9120504
li x7 0xBEEFCACE
sw x5 0(x6)
addi x6 x6 4
addi x5 x5 4
sh x6 2(x5)
sb x7 1(x7)
sb x7 3(x6)
sb x7 3(x5)
```

0xFFFFFFFF	
0xF9120504	
0xABADCAFE	
0x00000004	
0x00000000	0x00000000

Update the memory array with its new values after the code is executed. Some memory addresses may not have been labeled for you yet.

4 Calling Convention Practice

4.1 In a function called `myfunc`, we want to call two functions called `generate_random` and `reverse`.

`myfunc` takes in 3 arguments: `a0, a1, a2`

`generate_random` takes in no arguments and returns a random integer to `a0`.

`reverse` takes in 4 arguments: `a0, a1, a2, a3` and doesn't return anything.

```
myfunc:
    # Prologue (omitted)

    # assign registers to hold arguments to myfunc
    addi t0 a0 0
    addi s0 a1 0
    addi a7 a2 0

    jal generate_random

    # store and process return value
    addi t1 a0 0
    slli t5 t1 2

    # setup arguments for reverse
    add a0 t0 x0
    add a1 s0 x0
    add a2 t5 x0
    addi a3 t1 0

    jal reverse

    # additional computations
    add t0 s0 x0
    add t1 t1 a7
    add s9 s8 s7
    add s3 x0 t5

    # Epilogue (omitted)
    ret
```

4.1 Which registers, if any, need to be saved on the stack in the prologue?

4.2 Which registers do we need to save on the stack before calling `generate_random`?

4.3 Which registers do we need to save on the stack before calling `reverse`?

4.4 Which registers need to be recovered in the epilogue before returning?

CS 61C Reference Card

Version 1.6.0

	Instruction	Name	Description	Type	Opcode	Funct3	Funct7
Arithmetic	add rd rs1 rs2	ADD	rd = rs1 + rs2	R	011 0011	000	000 0000
	sub rd rs1 rs2	SUBtract	rd = rs1 - rs2	R	011 0011	000	010 0000
	and rd rs1 rs2	bitwise AND	rd = rs1 & rs2	R	011 0011	111	000 0000
	or rd rs1 rs2	bitwise OR	rd = rs1 | rs2	R	011 0011	110	000 0000
	xor rd rs1 rs2	bitwise XOR	rd = rs1 ^ rs2	R	011 0011	100	000 0000
	sll rd rs1 rs2	Shift Left Logical	rd = rs1 << rs2	R	011 0011	001	000 0000
	srl rd rs1 rs2	Shift Right Logical	rd = rs1 >> rs2 (Zero-extend)	R	011 0011	101	000 0000
	sra rd rs1 rs2	Shift Right Arithmetic	rd = rs1 >> rs2 (Sign-extend)	R	011 0011	101	010 0000
	slt rd rs1 rs2	Set Less Than (signed)	rd = (rs1 < rs2) ? 1 : 0	R	011 0011	010	000 0000
	sltu rd rs1 rs2	Set Less Than (Unsigned)		R	011 0011	011	000 0000
	addi rd rs1 imm	ADD Immediate	rd = rs1 + imm	I	001 0011	000	
	andi rd rs1 imm	bitwise AND Immediate	rd = rs1 & imm	I	001 0011	111	
	ori rd rs1 imm	bitwise OR Immediate	rd = rs1 | imm	I	001 0011	110	
	xori rd rs1 imm	bitwise XOR Immediate	rd = rs1 ^ imm	I	001 0011	100	
	slli rd rs1 imm	Shift Left Logical Immediate	rd = rs1 << imm	I*	001 0011	001	000 0000
	srli rd rs1 imm	Shift Right Logical Immediate	rd = rs1 >> imm (Zero-extend)	I*	001 0011	101	000 0000
	srai rd rs1 imm	Shift Right Arithmetic Immediate	rd = rs1 >> imm (Sign-extend)	I*	001 0011	101	010 0000
	slti rd rs1 imm	Set Less Than Immediate (signed)	rd = (rs1 < imm) ? 1 : 0	I	001 0011	010	
	sltiu rd rs1 imm	Set Less Than Immediate (Unsigned)		I	001 0011	011	
Memory	lb rd imm(rs1)	Load Byte	rd = 1 byte of memory at address rs1 + imm, sign-extended	I	000 0011	000	
	lbu rd imm(rs1)	Load Byte (Unsigned)	rd = 1 byte of memory at address rs1 + imm, zero-extended	I	000 0011	100	
	lh rd imm(rs1)	Load Half-word	rd = 2 bytes of memory starting at address rs1 + imm, sign-extended	I	000 0011	001	
	lhu rd imm(rs1)	Load Half-word (Unsigned)	rd = 2 bytes of memory starting at address rs1 + imm, zero-extended	I	000 0011	101	
	lw rd imm(rs1)	Load Word	rd = 4 bytes of memory starting at address rs1 + imm	I	000 0011	010	
	sb rs2 imm(rs1)	Store Byte	Stores least-significant byte of rs2 at the address rs1 + imm in memory	S	010 0011	000	
	sh rs2 imm(rs1)	Store Half-word	Stores the 2 least-significant bytes of rs2 starting at the address rs1 + imm in memory	S	010 0011	001	
	sw rs2 imm(rs1)	Store Word	Stores rs2 starting at the address rs1 + imm in memory	S	010 0011	010	

	Instruction	Name	Description	Type	Opcode	Funct3

Control	**`beq   rs1 rs2 label`**	Branch if EQual	**`if (rs1 == rs2)`**
			`PC = PC + offset`
				B	**`110 0011`**	**`000`**

	`bge   rs1 rs2 label`	Branch if Greater or Equal (signed)	**`if (rs1 >= rs2)`**
			`PC = PC + offset`
				B	**`110 0011`**	**`101`**

	`bgeu  rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		B	**`110 0011`**	**`111`**

	`blt   rs1 rs2 label`	Branch if Less Than (signed)	**`if (rs1 < rs2)`**
			`PC = PC + offset`
				B	**`110 0011`**	**`100`**

	`bltu  rs1 rs2 label`	Branch if Less Than (Unsigned)		B	**`110 0011`**	**`110`**

	`bne   rs1 rs2 label`	Branch if Not Equal	**`if (rs1 != rs2)`**
			`PC = PC + offset`
				B	**`110 0011`**	**`001`**

	`jal   rd label`	Jump And Link	**`rd = PC + 4`**
			`PC = PC + offset`
				J	**`110 1111`**	

	`jalr  rd rs1 imm`	Jump And Link Register	**`rd = PC + 4`**
			`PC = rs1 + imm`
				I	**`110 0111`**	**`000`**

Other	**`auipc rd imm`**	Add Upper Immediate to PC	**`rd = PC + (imm << 12)`**	U	**`001 0111`**	
	`lui   rd imm`	Load Upper Immediate	**`rd = imm << 12`**	U	**`011 0111`**	
	`ebreak`	Environment BREAK	Asks the debugger to do something (**`imm = 0`**)	I	**`111 0011`**	**`000`**
	`ecall`	Environment CALL	Asks the OS to do something (**`imm = 1`**)	I	**`111 0011`**	**`000`**
Ext	**`mul   rd rs1 rs2`**	MULtiply (part of mul ISA extension)	**`rd = rs1 * rs2`**	(omitted)		

#	Name	Description	#	Name	Desc
x0	**zero**	Constant 0	**x16**	**a6**	*Args*
x1	**ra**	*Return Address*	**x17**	**a7**	
x2	**sp**	Stack Pointer	**x18**	**s2**	Saved Registers
x3	**gp**	Global Pointer	**x19**	**s3**	
x4	**tp**	Thread Pointer	**x20**	**s4**	
x5	**t0**	*Temporary Registers*	**x21**	**s5**	
x6	**t1**		**x22**	**s6**	
x7	**t2**		**x23**	**s7**	
x8	**s0**	Saved Registers	**x24**	**s8**	
x9	**s1**		**x25**	**s9**	
x10	**a0**	*Function Arguments or Return Values*	**x26**	**s10**	
x11	**a1**		**x27**	**s11**	
x12	**a2**	*Function Arguments*	**x28**	**t3**	*Temporaries*
x13	**a3**		**x29**	**t4**	
x14	**a4**		**x30**	**t5**	
x15	**a5**		**x31**	**t6**	
Caller saved registers					
Callee saved registers (except **x0**, **gp**, **tp**)					

Pseudoinstruction	Name	Description	Translation

`beqz rs1 label`	Branch if EQuals Zero	**`if (rs1 == 0)`**
		`PC = PC + offset`
			`beq rs1 x0 label`

`bnez rs1 label`	Branch if Not Equals Zero	**`if (rs1 != 0)`**
		`PC = PC + offset`
			`bne rs1 x0 label`

`j label`	Jump	**`PC = PC + offset`**	**`jal x0 label`**
`jr rs1`	Jump Register	**`PC = rs1`**	**`jalr x0 rs1 0`**
`la rd label`	Load absolute Address	**`rd = &label`**	**`auipc`**, **`addi`**
`li rd imm`	Load Immediate	**`rd = imm`**	**`lui`** (if needed), **`addi`**
`mv rd rs1`	MoVe	**`rd = rs1`**	**`addi rd rs1 0`**
`neg rd rs1`	NEGate	**`rd = -rs1`**	**`sub rd x0 rs1`**
`nop`	No OPeration	do nothing	**`addi x0 x0 0`**
`not rd rs1`	bitwise NOT	**`rd = ~rs1`**	**`xori rd rs1 -1`**
`ret`	RETurn	**`PC = ra`**	**`jalr x0 x1 0`**

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

Immediates are sign-extended to 32 bits, except in I* type instructions and **`sltiu`**.