CS 61C Fall 2020

RISC-V Intro & Control Flow

Discussion 4: September 21, 2020

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 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.

t rigester may change, a registers cound change as well
t registers don't change

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

no, the elements in x don't have to be 4 bytes and x could only have 1 element

1.3 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.

It could be ✓

1.4 Assuming integers are 4 bytes, adding the ASCII character `'d'` to the address of an integer array would get you the element at index 25 of that array (assuming the array is large enough).

right ✓

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

jalr will store the ra in assigned register
jal jumps to a label (translated into an immediate by the assembler)

1.6 Calling `j label` does the exact same thing as calling `jal label`.

jalr jumps to an address stored in a registe

no, j label doesn't return

j is short for jal X0, label, it will return to the memory address specified in the second argument

2 Basic Instructions

For your reference, here are some of the basic instructions for arithmetic operations and dealing with memory (Note: ARG1 is argument register 1, ARG2 is argument register 2, and DR is destination register):

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts ARG1 by ARG2 and stores in DR
srl	Logical right shifts ARG1 by ARG2 and stores in DR
sra	Arithmetic right shifts ARG1 by ARG2 and stores in DR
slt/u	If ARG1 < ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison

[inst]	[register] [offset]([register containing base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register

[inst]	[argument register 1] [argument register 2] [label]
beq	If ARG1 == ARG2, moves to label
bne	If ARG1 != ARG2, moves to label

[inst]	[destination register] [label]
jal	Stores the next instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that ARG2 becomes an "immediate" or an integer instead of using a register. There are also immediates in some other instructions such as **sw** and **lw**. Note that the size (maximum number of bits) of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

2.1 Assume we have an array in memory that contains `int *arr = {1,2,3,4,5,6,0}`. Let register `s0` hold the address of the element at index 0 in arr. You may assume integers are four bytes and our values are word-aligned. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

0 4 8 12

a) `lw t0, 12(s0)` -->

load value 4 to t0

b) `sw t0, 16(s0)` -->

store 4 to arr[4]

c)
```
slli t1, t0, 2
add  t2, s0, t1
lw   t3, 0(t2)
addi t3, t3, 1
sw   t3, 0(t2)
```
-->

$t_1 = 16$ $t_2 \rightarrow arr+4$

$t_3 = 4 \rightarrow t_3 = 5$

$arr[4] = 5$

d)
```
lw   t0, 0(s0)
xori t0, t0, 0xFFF
addi t0, t0, 1
```
-->

$t_0 = 1 \rightarrow t_0 = 0 \rightarrow t_0 = 1$

3 C to RISC-V

3.1 Translate between the C and RISC-V verbatim.

C	RISC-V

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	li s0, 4
	li s1, 5
	li s2, 6
	add a, a, b
	add a, a, c
	addi a, a, 10
	mv z, a

`// s0 -> int * p = intArr;`
`// s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`
	li t0, 0
	sw t0, 0(s0)
	add s1, x0, 2
	sw s1, 0(s0)
	add s0, s0, s1
	sw s1, 0(s0)

`// s0 -> a, s1 -> b`
`int a = 5, b = 10;`
`if(a + a == b) {`
`  a = 0;`
`} else {`
`  b = a - 1;`
`}`
	li s0, 5
	li s1, 10
	add t0, s0, s0
	beq t0, s1, zero
	addi s1, s0 -1
	zero: li s0, 0

`int a = 0;`
`int b = 1;`
`for ( a; a<30; a++)`
`{`
`  b = b+b;`
`}`
	`addi s0, x0, 0`
	`addi s1, x0, 1`
	`addi t0, x0, 30`
	`loop:`
	`beq s0, t0, exit`
	`add s1, s1, s1`
	`addi s0, s0, 1`
	`jal x0, loop`
	`exit:`

`// s0 -> n, s1 -> sum`
`// assume n > 0 to start`
`for(int sum = 0; n > 0; n--) {`
`  sum += n;`
`}`
	li s1, 0
	li s0, n
	beq n, x0, exit
	add s1, s1, s0
	addi s0, s0, -1
	exit:

4 RISC-V with Arrays and Lists

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;
}
```

4.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

$t_0 = 3$
$t_1 = 4$
$t_2 = 7$
$arr[1] = 7$

4.2
```
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:
```

while $s_1 != end$;
$t_0 = ll \rightarrow val$
$t_0 = t_0 + 1$
$ll \rightarrow val = t_0$
$s_1 = ll \rightarrow next$

4.3
```
       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:
```

$t_0 = 0$
$t_1 = 1$
if $t_1 = 0$ end
$t_2 = t_0 < 2 \ ? \ 1 : 0$
$t_3 = arr + t_2$
$t_4 = arr[t_2]$
$t_4 = -t_4$
$arr[t_2] = -t_4$
$t_0 += 1$

5 RISC-V Calling Conventions

5.1 How do we pass arguments into functions?

Through $a_0 - a_7$

5.2 How are values returned by functions?

jal ra use a0 and a1 as the return value registers

5.3 What is `sp` and how should it be used in the context of RISC-V functions?

sp is short for stack pointer { addi sp, sp, -x / addi sp, sp, x

5.4 Which values need to saved by the caller, before jumping to a function using `jal`?

ra and values will be changed by callee

a0 - a7, t0 - t6 and ra

5.5 Which values need to be restored by the callee, before returning from a function?

ra sp, gp (global pointer), tp (thread pointer) and s0 - s11

5.6 In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers aren't guaranteed to be the same?

s registers a and t registers

(a0 - a7, t0 - t6, ra not same)

(sp, gp, tp, s0 - s11 are same)

6 Writing RISC-V Functions

6.1 Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

First, let's implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter `n`? What registers should hold `square`'s parameter and return value? In what register should we place the return value of `sumSquare`?

parameter : n → a0

return : sum → a0

```
li    s2, 0
mv    s0, a0
beq   a0, x0, exit
mul   s1, s0, s0
add   s2, s2, s1
addi  a0, a0, -1
jar   sumsquare

exit:
    jr  ra
```

6.2 Since `sumSquare` is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.

```
prologue:  addi  sp, sp, -16
           sw    ra, 0(sp)
           sw    s0, 4(sp)
           sw    s1, 8(sp)
           sw    s2, 12(sp)

epilogue:  addi  sp, sp, 16
           lw    ra, 0(sp)
           lw    s0, 4(sp)
           lw    s1, 8(sp)
           lw    s2, 12(sp)
```