CS 61C Summer 2020

RISC-V Intro & Control Flow

Discussion 4: July 1, 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.

F, caller saved registers can be changed during the execution

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

F, depend on what type of array, array of int, or char

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

T

1.4 Adding the 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).

T

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

F

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

F

2 RISC-V: A Rundown

RISC-V is an assembly language, which is comprised of simple instructions that each do a single task such as addition or storing a chunk of data to memory.

For example, on the left is a line of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
                                      // x -> s0, &y -> s1
int x = 5, y[2];                      addi s0, x0, 5     x=5
y[0] = x;                             sw   s0, 0(s1)     y[0]=x
y[1] = x * x;                         mul  t0, s0, s0    t=x * x
                                      sw   t0, 4(s1)     y[1]=t
```

2.1 Can you figure out what each line in the RISC-V code is doing?

3 Registers

In RISC-V, we have two methods of storing data: main memory and registers. Registers are much faster than using main memory, but are very limited in space (32 bits each). Note that you should ALWAYS use the named registers (e.g. `s0` rather than `x8`).

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

3.1 Can you convert each instruction's registers to the other form?

```
add s0, zero, a1     -->     add x8, x0, x11
or  x18, x1, x30     -->     add s2, ra, t5
```

4 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: The size of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

4.1 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.

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

t0 = arr[3] = 4

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

arr[4] = 4

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

t1 = t0 << 2 = 16
t2 now points to arr[4]
t3 = arr[4] = 4
t3 = 5
arr[4] = 5

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

t0 = arr[0] = 1
t0 = t0 ^ 0xFFF = 0xFFE
t0 = t0 + 1 = 0xFFF

5 C to RISC-V

5.1 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;
	addi s0 x0 4
	addi s1 x0 5
	addi s2 x0 6
	add t0 s0 s1
	addi t1 s3 10
	add s3 t0 t1

// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;
	sw x0 0(s0)
	addi s1 x0 2
	sw s1 8(s0)
	lw t0 8(s0)
	sw t0 4(s0)

// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
	addi s0 x0 5
	addi s1 x0 10
	add t0 s0 s1
	bne t0 s1 branch
	addi s0 x0 0
	j end
	branch: sub s1 s0 1
	end:

// s0 -> i, s1 -> x;
int i = 0, x = 1;
while (i != 30) {
  x += x;
  i++;
}
	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;
}
	addi s1 x0 0
	addi t0 x0 0
	loop: bge t0 s0 exit
	add s1 s1 s0
	sub s0 s0 1
	jal x0 loop
	exit:

6 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;
}
```

6.1

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

add a[0] and a[2] as a[1]

6.2

```
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:
```

increment every list element by 1

6.3

```
       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:
```

Set every array element to its reverse

7 RISC-V Calling Conventions

7.1 How do we pass arguments into functions?

7.2 How are values returned by functions?

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

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

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

7.6 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?

8 Writing RISC-V Functions

8.1 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`?

a0 for parameter n
a0 for square's parameter and return value
a0 for return value from sumSquare

```
          addi t0 x0 0
          add t1 x0 a0
loop:
          bge x0 t1 end
          mv a0 t1
          jal ra square
          add t0 t0 a0
          addi t1 t1 -1
          j loop
end:
          mv a0 t0
          ret
```

8.2 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.

```
        addi sp sp -12
        addi t0 x0 0
        add t1 x0 a0
loop:
        bge x0 t1 end
        mv a0 t1
        sw ra 8(sp)
        sw t1 4(sp)
        sw t0 0(sp)
        jal ra square
        lw t0 0(sp)
        lw t1 4(sp)
        lw ra 8(sp)
        add t0 t0 a0
        addi t1 t1 -1
        j loop
end:
        mv a0 t0
        addi sp sp 12
        ret
```

9 More Translating between C and RISC-V

9.1 Translate between the RISC-V code to C. What is this RISC-V function computing? Assume no stack or memory-related issues, and assume no negative inputs.

C	RISC-V

`// a0 -> x, a1 -> y,`
`// t0 -> result`
 	
`int func(int x, int y) {`
`  int t = 1;`
`  while (y > 0) {`
`    t *= x`
`    y - -;`
`  }`
`  return t;`
`}`
 	
`// function return x to the y-th power`
	`Func: addi t0 x0 1`
	`Loop: beq a1 x0 Done`
	`      mul t0 t0 a0`
	`      addi a1 a1 -1`
	`      jal x0 Loop`
	`Done: add a0 t0 x0`
	`      jr ra`