CS 61C
Fall 2022

RISC-V Assembly, Control

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`.

False. This only holds for data types that are four bytes wide, like `int` or `float`. For data-types like `char` that are only one byte wide, `4(a0)` is too large of an offset to return the element at index 1, and will instead return a `char` further down the array (or some other data beyond the array, depending on the array length).

1.2 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).

True. There is no fundamental difference between integers, strings, and memory addresses in RISC-V (they're all bags of bits), so it's possible to manipulate data in this way. (We don't recommend it, though).

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.

True. If your compiler/OS allows it (some do not, for security reasons), it is possible for your code to jump to and execute instructions passed into the program via an array. Conversely, it's also possible for your code to treat itself as normal data (search up self-modifying code if you want to see more details).

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

False. `j label` is a pseudo-instruction for `jal x0, label`. `jalr` is used to return to the memory address specified in the second argument. Keep in mind that `jal` jumps to a label (which is translated into an immediate by the assembler), whereas `jalr` jumps to an address stored in a register, which is set at runtime.

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

False. As from the previous problem, `j label` is short for `jal x0, label` — since it's writing the return address to `x0`, it's effectively discarding it since we have no need to jump back to the original PC. `jal label` is short for `jal ra, label`.

2 Instructions

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 snippet of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
int x = 5;
y[2];
y[0] = x;
y[1] = x * x;
```

```
// x -> s0, &y -> s1
addi s0, x0, 5
sw   s0, 0(s1)
mul  t0, s0, s0
sw   t0, 4(s1)
```

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.

```
a) lw   t0, 12(s0)        -->       Sets t0 equal to arr[3]

b) sw   t0, 16(s0)        -->       Stores t0 into arr[4]

c) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)         -->       Increments arr[t0] by 1
   addi t3, t3, 1
   sw   t3, 0(t2)

d) lw   t0, 0(s0)
   xori t0, t0, 0xFFF     -->       Sets t0 to -1 * arr[0]
   addi t0, t0, 1
```

2.2 Assume that `s0` and `s1` contain signed integers. Without any pseudoinstructions, how can we branch on the following conditions to jump to some LABEL?

s0 $<$ s1	s0 $\neq$ s1	s0 $\leq$ s1	s0 $>$ s1
`blt s0, s1, LABEL`	`bne s0, s1, LABEL`	`bge s1, s0, LABEL`	`blt s1, s0, LABEL`

Note that RISC-V does not provide a bgt instruction because you can manipulate the blt instruction to get an equivalent result. Also note that the above solutions assume that `s0` and `s1` contained signed integers. If they are unsigned, then we would use the unsigned variants of the above commands (namely, `bltu, bgeu`).

3 Lost in Translation

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;`
	`addi s0, x0, 4`
	`addi s1, x0, 5`
	`addi s2, x0, 6`
	`add  s3, s0, s1`
	`add  s3, s3, s2`
	`addi s3, s3, 10`

`// 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, 4(s0)`
	`slli t0, s1, 2`
	`add  t0, t0, s0`
	`sw   s1, 0(t0)`

```
// 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, s0
    bne  t0, s1, else
    xor  s0, x0, x0
    jal  x0,  exit
else:
    addi s1, s0, -1
exit:
```

```
// computes s1 = 2^30
// assume int s1, s0; was declared above
s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
    s1 *= 2;
}
```

```
    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
loop:
    beq  s0, x0, exit
    add  s1, s1, s0
    add  s0, s0, -1
    jal  x0, loop
exit:
```

4 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;
}
```

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

Sets `arr[1]` to `arr[0]` + `arr[2]`.

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:
```

Increments all values in the linked list by 1.

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:
```

Negates all elements in `arr`.

5 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.

5.1

```
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?

`x5` will hold `0x00FF0004`, after adding 4 to the initial address. `x6` will hold 36, loading the word from the address `0x00FF0000`. `x7` will hold `0xC`, loading the upper half of the address `0x00FF0004`. `x8` will hold the word at $36 = 0x24$, so `0xDEADB33F`. Finally, `x9` will hold `0xFFFFFFC5`, taking the most significant byte and sign-extending it.

5.2

```
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.

0xFFFFFFFF	
0xF9120508	0xCE000000
0xF9120504	0xABADCAFE
0xBEEFCAD2	
0xBEEFCACE	0x0000CE00
0xABADCB02	0xCE080000
0xABADCAFE	
0x00000004	
0x00000000	0x00000000