CS 61C
Fall 2018

RISC-V Intro

Discussion 3: September 10, 2018

1 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.

```
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)
```

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

`addi s0, x0, 5` evaluates to `x = 5`. `sw s0, 0(s1)` evaluates to `y[0] = x`. `mul t0, s0, s0` calculates `x * x`. `sw t0, 4(s1)` evaluates to `y[1] = x * x`.

2 Registers

In RISC-V, we have two methods of storing data, one of them is main memory, the other is through registers. Registers are much faster than using main memory, but are very limited in space (32-bits)

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

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

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

3 Basic Instructions

For your reference, here are a couple of the basic instructions for arithmetic operations and dealing with memory:
Basic Operations:

[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 AR1 by AR2 and stores in DR
srl	Logical right shifts AR1 by AR2 and stores in DR
sra	Arithmetic right shifts AR1 by AR2 and stores in DR
slt/u	If AR1 < AR2, stores 1 in DR, otherwise stores 0, u does unsigned comparison

[inst]	[register] [offset]([register with 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 AR1 == AR2, moves to label
bne	If AR1 != AR2, moves to label

[inst]	[destination register] [label]
jal	Stores the current 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 AR2 becomes an "immediate" or an integer instead of using a register.

3.1 Assume we have an array in memory that contains `int* arr = {1,2,3,4,5,6,0}`. Let the values of arr be a multiple of 4 and stored in register `s0`. 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) 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)

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

3.2 While only using the instructions (and their "i" forms) given above, how can we branch on the following conditions:

s0 < s1	s0 ≥ s1	s0 > 1
`slt t0, s0, s1`	`slt t0, s0, s1`	`sltiu t0, s0, 2`
`bne t0, zero, label`	`beq t0, zero, label`	`beq   t0, zero, label`

4 C to RISC-V

4.1 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`
`s1 = 1;`
`for(s0=0;s0<30;s++) {`
`    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, s1, 0`
	`loop:`
	`    beq  s0, x0, exit`
	`    add  s1, s1, s0`
	`    add  s0, s0, -1`
	`    jal  x0, loop`
	`exit:`