Computer Architecture Homework 2

110062219

March 30, 2023

Contents

1 Andes C GCD project

1.1 Find the memory addresses

The address of the global variable `result` is shifted 4968 from the global register `gp`, which equals `0x11538`.

The address of the procedure `gcd` is `0x104a8`.

1.2 `remw` instruction

The `remw` instruction performs the **modulus** operation on **words**, i.e., 32-bit integers. Specifically, we do signed division on the lower 32 bits of the source registers and store their remainder with sign extension to the destination register.

1.3 Stack

1.3.1 Stack frame

```
c.addi16sp sp,-48
c.sdsp ra,40(sp)
c.sdsp s0,32(sp)
c.addi4spn s0,sp,48
c.mv a5,a0
c.mv a4,a1
sw a5,-36(s0)
c.mv a5,a4
sw a5,-40(s0)
lw a4,-36(s0)
lw a5,-40(s0)
remw a5,a4,a5
sw a5,-20(s0)
```

When entering the `gcd()` function, we would allocate a stack frame of size 48 and push the previous *return address*, *frame pointer* to the top of it.

Moreover, we would save the local variables a, b and the register that stores $a \bmod b$ into the frame as well.

Table 1: Stack block

Addresses	Contents	Offsets	Notes
High		0	*Frame pointer*
	Saved *return address*	-8	
	Saved *frame pointer*	-16	
	Saved a mod b	-20	
		-32	
	Saved a	-36	
	Saved b	-40	
Low		-48	*Stack pointer*

1.3.2 Lowest address of stack pointer

The depth of the recursive calls is 6. Hence the lowest address *stack pointer* would reach should be `0x2FFFED0`.

1.4 Assembly code corresponding to the tail recursion

```
lw a4,-20(s0)
lw a5,-40(s0)
c.mv a1,a4
c.mv a0,a5
jal ra,0x104a8 <gcd>
add a5,zero,a0
```

First we load b, a mod b from memory we stored previously. Then we copy them to `a0`, `a1` and do the recursive call.

1.5 Assembly code corresponding to the tail recursion with `-Og`

```
c.mv a0,a5
jal ra,0x104a8 <gcd>
```

We copy b to `a0`. Since the remainder has been stored in `a1`, we could thus do the recursive call. With `-Og`, we make better use of registers and reduce the access to memory.

2 RISC-V codes

2.1 Hexadecimal representation

The offset is -6, `0b1111_1110_1000` in 12-bit 2's complement, so we have the table:

Table 2: Binary representation

imm_{12}	$imm_{10:5}$	rs_2	rs_1	$funct_3$	$imm_{4:1}$	imm_{11}	$opcode$
1	111111	00000	01010	000	0100	1	1100011

, which is equivalent to `0xfe0504e3`.

2.2 Jump-and-link instruction

We would jump to the second next instruction `00000000101000011011110000100011`, which indicates `sd a0,24(gp)`.

2.3 Long jump

The address of `beq` is `0x80000038`.

Table 3: Long jump instructions

`Addresses`	`Instructions`
`0x20000000`	`lui t0,0x80000`
`0x20000004`	`jalr zero,0x038(t0)`

After `jalr`, the `PC` would become `0x80000038`.

3 Endianness

Since an ASCII character is a single byte, it makes no difference whether it's little or big endian.

Table 4: Memory addresses & ASCII data

Addresses	Data
0x00000000	0x52
0x00000001	0x49
0x00000002	0x53
0x00000003	0x43
0x00000004	0x2D
0x00000005	0x56
0x00000006	0x32
0x00000007	0x76

4 C to RISC-V

```
slli t0,x5,4 # (i*2) * 8 = i * 16
add t0,x6,t0 # t0 = &A[i * 2]
ld t0,0(t0)  # t0 = A[i * 2]

slli t0,t0,3
add t0,x7,t0 # t0 = &B[A[i * 2]]
ld t0,0(t0)  # t0 = B[A[i * 2]]

addi x10,t0,-16
```

5 RISC-V to C

I name the pointer `x30`, variable `x31` to be `tmp0`, `tmp1` respectively.

```
tmp0 = D;                     // addi x30,x14,0
for (i = 0; i < m; i++)       // addi x11,x0,0; bge x11,x3,ENDI; addi x11,x11,1
{                             // LOOPI:
    tmp1 = *tmp0;             // lw x31,0(x30)
    for (j = 0; j < n; j++)   // addi x12,x0,0; bge x12,x4,ENDJ; addi x12,x12,1
    {                         // LOOPJ:
        tmp1 += (i + j) * 7;  // add x27,x11,x12; slli x28,x27,3; sub x28,x28,x27
                              // add x31,x31,x28
    }                         // jal x0,LOOPJ; ENDJ:
    *tmp0++ = tmp1;           // sw x31,0(x30); addi x30,x30,4
}                             // jal x0,LOOPI; ENDI:
```

We could further simplify:

```
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        D[i] += (i + j) * 7;
```

6 C to RISC-V again

```
func:
  addi sp,sp,-32
  sd ra,24(sp)
  sd s0,16(sp)
  sd s1,8(sp)
  sd s2,0(sp)
  addi s0,a0,0    # mv s0,a0
  addi a5,zero,2  # li a5,2
  bltu a5,a0,L2   # bgtu a0,a5,.L2
L1:
  ld ra,24(sp)
  ld s0,16(sp)
  ld s1,8(sp)
  ld s2,0(sp)
  addi sp,sp,32
  jalr zero,0(ra) # jr ra
L2:
  addiw a0,a0,-1
  jal ra,func     # call func
  mulw s2,s0,s0
  add s2,s2,a0
  addiw a0,s0,-2
  jal ra,func     # call func
  slli s1,a0,3
  add s1,s1,s2
  addiw a0,s0,-3
  jal ra,func     # call func
  add a0,s1,a0
  jal zero,L1     # j .L1
```

7 True or false

7.1 Big-endian

False. We would get `0xB3`.

7.2 Temporary registers

False. Caller should save `t0`, `t1` if necessary.

7.3 Load upper immediate

False. `x6` would become `0xFFFFAAAA`.

7.4 Memory access

True. 2 `ld` and a `sd` add up accessing the memory thrice.