Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2021-11-12
Due: 2021-11-19

ICS 2021 Problem Sheet #10

Problem 10.1: *assembler programming* (1+2+3+1 = 7 points)

The following program has been written for the simple central processing unit introduced in class. The table below shows the initial content of the 16 memory cells. The first column denotes the memory address and the second column shows the memory content in hexadecimal notation.

Cell	Hex	Binary	Assembler	Description
0	2f			
1	6a			
2	4f			
3	21			
4	71			
5	41			
6	a9			
7	d0			
8	e0			
9	6f			
10	01			
11	02			
12	03			
13	04			
14	05			
15	06			

a) Convert the machine code from hexadecimal notation into binary notation.

b) Write down the assembler code for the machine code. Add meaningful descriptions.

c) The program leaves a result in memory cell 15 when it halts. What is the value? Explain how the program works.

d) What happens if the value stored in memory cell 9 is changed to 0x70 before execution starts? Explain.

Solution:

Cell	Hex	Binary	Assembler	Description
0	2f	001 0 1111	`LOAD 15`	acc = m[15]
1	6a	011 0 1010	`ADD 10`	acc += m[10]
2	4f	010 0 1111	`STORE 15`	m[15] = acc
3	21	001 0 0001	`LOAD 1`	acc = m[1]
4	71	011 1 0001	`ADD #1`	acc += 1
5	41	010 0 0001	`STORE 1`	m[1] = acc
6	a9	101 0 1001	`EQUAL 9`	if acc == m[9] skip
7	d0	110 1 0000	`JUMP #0`	pc = 0
8	e0	111 0 0000	`HALT`	halt
9	6f	011 0 1111	`ADD 15`	acc += m[15]
10	01	000 0 0001	`DATA 1`	
11	02	000 0 0010	`DATA 2`	
12	03	000 0 0011	`DATA 3`	
13	04	000 0 0100	`DATA 4`	
14	05	000 0 0101	`DATA 5`	
15	06	000 0 0110	`DATA 6`	

a) See table above.

b) See table above.

c) The value left in memory cell 15 is 21 (0x15). The program sums up the numbers stored in the memory cells 10:15 and leaves the sum in memory cell 15. The program is self-modifying, the ADD instruction in memory cell 1 is updated in each iteration to refer to different memory cells.

d) The change causes one more addition to be made. This has the effect that the sum is calculated and then added to itself. Hence, we obtain 21+21=42 (0x2a).

Marking:

a) - *subtract .1pt for each incorrect binary number (minimum score 0pt)*

b) - *subtract .1pt for each incorrect assembler instruction (minimum score 0pt)*
- *subtract .1pt for each incorrect/missing description*

c) - *1pt for the correct answer*
- *2pt for a proper explanation*

d) - *1pt for a proper explanation*

Problem 10.2: *integer multiplication in risc-v rv32i assembler* (2+1 = 3 points)

The 32-bit RISC-V base integer instruction set (rv32i) does not support multiplication and division operations. To deal with this, a compiler may call a function when a multiplication is needed. For example, `gcc` expects that a function `__mulsi3(unsigned int a, unsigned int b)` is provided to multiply two integers. A multiplication can be carried out by repeated additions and shifts:

```
unsigned int __mulsi3 (unsigned int a, unsigned int b)
{
    unsigned int r = 0;

    while (a) {
        if (a & 1) {
            r += b;
        }
        a >>= 1;
        b <<= 1;
    }
    return r;
}
```

a) Translate the above C code into equivalent RISC-V rv32i assembler code. Comment the assembler code to explain how the calculation proceeds. Note that the arguments are passed via the registers a0 (x10) and a1 (x11) and that the result is returned in a0 (x10).

b) Does the function need function call prolog and epilog? Explain why or why not.

You are invited to use emulsiV to develop and test your assembler code.

Solution: This solution is derived from compiling the C code into rv32i assembler using optimization (`-O2`) and then commenting it.

a) The following assembler code for `__mulsi3()` was derived from compiling the C code into rv32i assembler using optimization (`-O2`) and then commenting it.

```
__mulsi3:
        mv      a5,a0       ; a5 = a0       ; copy argument a into a5
        li      a0,0        ; a0 = 0        ; initialize the r to 0
        beq     a5,zero,.L3 ; pc = .L3 if a5 == 0
```

```
.L1:
        andi    a4,a5,1      ; a4 = a5 & 1  ; extract the left most bit
        srli    a5,a5,1      ; a5 = a5 >> 1 ; shift a5 right by 1 position
        beq     a4,zero,.L2 ; pc = .L2 if a4 == 0
        add     a0,a0,a1     ; a0 = a0 + a1 ; add argument b to r
.L2:
        slli    a1,a1,1      ; a1 = a1 << 1 ; shift a1 left by 1 position
        bne     a5,zero,.L1 ; pc = .L1 if a5 != 0
.L3:
        ret                  ; return
```

It is also possible to create a handcrafted version that avoids some needless jumps.

```
__mulsi3:
        mv      a2, a0       ; a2 = a0      ; copy argument a into a2
        li      a0, 0        ; a0 = 0       ; initialize the r to 0
.L1:
        andi    a3,a1, 1     ; a3 = a1 & 1  ; extract the left most bit
        beqz    a3,.L2       ; pc = .L2 if a3 == 0
        add     a0,a0,a2     ; a0 = a0 + a2 ; add argument a to r
.L2:
        srli    a1,a1,1      ; a1 = a1 >> 1 ; shift a1 right by 1 position
        slli    a2,a2,1      ; a2 = a2 << 1 ; shift a2 left by 1 position
        bnez    a1,.L1       ; pc = .L1 if a1 != 0
        ret                  ; return
```

b) Since the function does not call any other functions, it it not required to allocate a stack frame. The return address stays in the return register ra (x1) and the local variable used to hold the product can be kept in a register.

Marking:

a) - *1pt for a correct translation*
- *1pt for a suitable description*

b) - *1pt for an explanation that a stack frame is not needed*