Session 3 Functions and the stack

3.1 Exercises

Exercise 1 Convert the following code to Risc-V assembly. Assume that `main()` does not return like common functions. Why is this assumption necessary right now? Use Risc-V calling conventions.[1].

```
1  int x = 10;
2  int y = 20;
3  int z;
4
5  int doubleIt(int a)
6  {
7    return a + a;
8  }
9
10 int sum(int a, int b)
11 {
12   return a + doubleIt(b);
13 }
14
15 int main()
16 {
17   z = sum(x, y);
18 }
```

[1] `https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf`

Exercise 2 We have compiled the function `func` from the code example below to Risc-V 32-bit (RV32I) using `gcc`. The compiled function can be found in `3.2-gcc.asm`. Translate the main function manually to Risc-V. Follow the calling conventions to pass all arguments correctly.

```
1
2  long long result;
3  long long i = 6;
4
5  long long func(int a, long b, long long c, long long d,
6                 long long e, long long *f)
7  {
8    return a + b + c + d + e + *f;
9  }
10
11 int main()
12 {
13     result = func(1, 2, 3, 4, 5, &i);
14 }
```

Exercise 3 Fix the function `sum_fixme` in file `3.3-callersave.asm`. Only add code at the designated `TODO`-points, don't modify the existing code. Use the stack to make sure *caller-save* registers are saved by the *caller* and *callee-save* registers are saved by the *callee*. Note that `sum_fixme` acts both as a caller and a callee. Your solution is correct if the execution terminates with

1. no errors;
2. the value 3 in `a0`;
3. the value `0xdeadbeef` in `s0`.

Exercise 4 Consider the following recursive function which calculates n!.

```
1  unsigned int fact(unsigned int n) {
2          if (n < 2) return 1;
3          return n*fact(n-1);
4  }
```

1. Convert this function to Risc-V.
2. Consider the call `fact(3);`. What is the state of stack when it reaches its maximum size (at the deepest level of recursion)?
3. In exercise 3 of the previous session you implemented an iterative factorial function. Compare both factorial implementations in terms of memory usage. Which implementation do you prefer?

Exercise 5 A *tail call* occurs whenever the last instruction of a subroutine (before the return) calls a different subroutine. Compilers can take advantage of tail calls to reduce memory usage.

1. The call `fact(n-1)` in the previous exercise is *not* a tail call. Why not?
2. We have converted the factorial program to use tail recursion. Translate this program to Risc-V. Try to avoid using the call stack during the `fact_tail` implementation. Why is this possible?

```
1  unsigned int fact_tail(unsigned int n, unsigned int result) {
2          if (n <= 1) return result;
3          return fact_tail(n - 1, n * result);
4  }
5
6  unsigned int fact(unsigned int n){
7    return fact_tail(n, 1);
8  }
9
10 int main(){
11   int n = 5;
12   int r;
13   r = fact(n);
14 }
```