Tutorial 4 – Week of Oct. 4th 2021

In most cases, C int is 32 bits. (4 bytes.)

Question 1) 2.27

Translate the following loop into C. Assume that the C-level integer `i` is held in register `x6`, `x5` holds the C-level integer called `result`, and `x10` holds the base address of the integer

```
.data
MemArray:                   // array of 100 words
          .word     0
          .word     1
// …
          .word     99

.text
     addi x6, x0, 0         // i = 0
     addi x5, x0, 0         // result = 0
     addi x29, x0, 100      // x29 = 100
LOOP:lw   x7, 0(x10)        // x7 = Mem[&MemArray[0] + 0] = MemArray[0]
     add  x5, x5, x7        // x5 = result + MemArray[0]
     addi x10, x10, 4       // x10 = x10 + 4 = &MemArrya[4]
     addi x6, x6, 1         // i = i + 1
     blt  x6, x29, LOOP     // if i > 100 goto LOOP
out of loop:
```

x6 = i
x5 = result
x10 = &MemArray[0]

Solution:

```
int i;
int result = 0;
for (i = 0; i < 100; i++) {
     result += *MemArray;
     MemArray++;
}
return result;
```

// Many people would write the code this way

```
int i;
int result = 0;
for (i = 0; i < 100; i++) {
     result += MemArray[i];
}
return result;
```

This is a non-leaf procedure example. (procedure that calls another procedure in its body).

Question 2) 2.31

In most cases, C int is 32 bits. (4 bytes.)

x2 ($sp) is stack pointer.

Translate function f into RISC-V assembly language. Assume the function declaration for `g` is

```
int g(int a, int b).
```

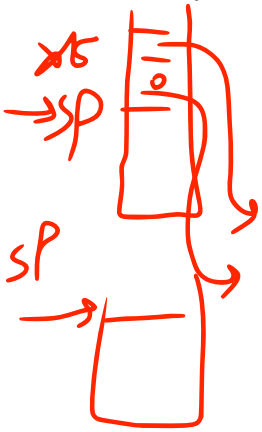
The code for function f is as follows:

```
int f(int a, int b, int c, int d){
    return g(g(a,b), c+d);
}
```

Assume (by convention):
for g:
x10 = 1st_arg, x11 = 2nd_arg, x10 also store return value after execution.
for f:
x10 = 1st_arg, x11 = 2nd_arg, x12 = 3rd_arg, x13 = 4th_arg
Initially, x10 also store return value after execution.
x10 = a, x11 = b, x12 = c, x13 = d

we know that x10, x11, x12, x13 should be saved by caller by convention.

Solution:



```
f:
    addi x2, x2, -8      // Allocate stack space for 2 words
    sw   x1, 0(x2)       // Save return address
    add  x5, x12, x13    // x5 = c+d
    sw   x5, 4(x2)       // Save c+d on the stack
    jal  x1, g           // Call x10 = g(a,b)
    lw   x11, 4(x2)      // Reload x11= c+d from the stack
    jal  x1, g           // Call x10 = g(g(a,b), c+d)
    lw   x1, 0(x2)       // Restore return address
    addi x2, x2, 8       // Restore stack pointer
    jalr x0, 0(x1)
```

return to f's caller, use x0 because we don't need to store our current PC address any more.

Risk: "shared" means other procedure could modify it during the execution of our procedure. Solution: we need a "lock" (lr.d and sc.d) to check if the shared variable is NOT interfered by other procedure. If it does being interfered by other procedure, we need to start from the beginning and execute our procedure again.

Question 3) 2.37

Write the RISC-V assembly code to implement the following C code as an atomic "set max" operation using the `lr.d` and `sc.d` instructions. Here, the argument `shvar` contains the address of a shared variable which should be replaced by `y` if `y` is greater than the value it points to. Assume `x10` is address of integer pointed by `shvar` and value of `y` is in `x11`.

Assumption:
x10 = shvar
x11 = y

```
void setmax(int* shvar, int y) {
     // Begin critical section
     if (y > *shvar)
          *shvar = y;
     // End critical section}
}
```

Solution:

1. **ld x5, 0(x10)**
2. **register reservation to x10**
3. **sd x5, 0(x10)**
4. **if success, write zero to x7**

```
setmax:
     try:
     lr.d x5, (x10)         # Load-reserve *shvar
     bge  x5, x11, release # Skip update if *shvar > y // if *shvar >= y goto release
     addi x5, x11, 0

release:
     sc.d x7, x5, (x10)
     bne  x7, x0, try # If store-conditional failed, try again
     jalr x0, 0(x1)
```