Problem Set 4: Register Machines. x86 and Arm.

C S 395T

15 September 2021

This is the problem set for week 4 of C S 395T SIMPL. It is intended to help you learn the material by working out more examples and exercises than is possible to cover in the videos. Feel free to work individually or in groups. Ask questions on Piazza. You are not required to submit anything, and the problem set doesn't count directly towards your course grade. The solution key for the problem set will be made available one week after its release.

Problem 1. *Address generation.*
The following C code fragment transposes array `A` into array `B`.

```
#define M 40
#define N 50

void transpose(short A[M][N], B[N][M]) {
  for (int i = 0; i < M; i++)
    for (int j = 0; i < N; j++)
      B[j][i] = A[i][j];
}
```

(a) Show how you would generate code for the assignment statement `B[j][i] = A[i][j];` in a simple manner. The parameter `A` is passed to you in register `%rdi` and `B` in register `%rsi`. Keep `i` in register `%r8d` and `j` in register `%r9d`. Use `LEAQ` instructions wherever possible. Use additional registers if needed.

(b) Simplify the generated code by exploiting regularities in the sequence of accesses to arrays `A` and `B`. Use additional registers if needed.

Problem 2. *Code generation.*
The following C code counts the number of 1-bits in the variable `x`. Hand-translate it into x86-64 assembly language code. The input parameter `x` is provided to you in register `%rdi`, and the result needs to be returned in register `%rax`.

```
int pop(unsigned x) {
  x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x + (x >> 4)) & 0x0F0F0F0F;
  x += x >> 8;
  x += x >> 16;
  return x & 0x0000003F;
```

Also, it might be instructive to examine the assembly code that `gcc` generates for these code fragments, and see whether it looks anything like the assembly code you generated by hand.

Problem 3. *Reverse-engineering stack frames.*
The assembly code generated by `gcc` for a C function with the prototype `long bar(long);` is as follows.

```
bar:
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp
movq %rdi, -8(%rbp)
cmpq $1, -8(%rbp)
jg LBB0_2
movq $1, -16(%rbp)
jmp LBB0_3
LBB0_2:
movq -8(%rbp), %rax
movq -8(%rbp), %rcx
subq $1, %rcx
movq %rcx, %rdi
movq %rax, -24(%rbp)
callq bar
movq -24(%rbp), %rcx
imulq %rax, %rcx
movq %rcx, -16(%rbp)
LBB0_3:
movq -16(%rbp), %rax
addq $32, %rsp
popq %rbp
retq
```

(a) Show the layout of the stack frame for this function. Indicate each of the four areas of the stack frame, how much each area takes, and the total size of the stack frame.

(b) The function is invoked with an argument value of 2. Show the state of the stack just before program execution reaches the recursive call to `bar`.

(c) Show the state of the stack when the function is in the recursive call to `bar`.

(d) What value does the function return when invoked with an argument value of 2?

Problem 4. *Caller-saved and callee-saved registers.*
For each of the C procedures below, identify the minimal sets of caller-saved and callee-saved registers that will be saved/restored in the assembly code generated for the procedure. The normal x86-64 procedure call/return linkage conventions are followed, and each procedure is compiled separately.

(a)
```
unsigned long fn1(long x, long y) {
  return x*x+y*y;
}
```

(b)
```
unsigned long fn2(long x, long y) {
  return fn1(x+y, y-2);
}
```

(c)
```
unsigned long fn3(long x, long y) {
  return fn1(x, y) - x*y;
}
```

(d)
```
unsigned long fn4(long x, long y) {
  y = fn1(y, x);
  return fn2(x, y);
}
```

(e)
```
unsigned long fn5(long x, long y) {
  return fn1(x, y) + fn2(y, x);
}
```