Problem Set 4: Register Machines. x86 and Arm. Solution Key

C S 395T

22 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.

Solution:

(a) We know that `A[i][j] == *(A+50*i+j)` and `B[j][i] == *(B+40*j+i)` at the C level. These offsets have to be scaled by `sizeof(short)` in the assembly code, so we really need to compute `A+2*(50i+j)` and `B+2*(40j+i)`.

```
LEAQ (%r8, %r8, 2), %r10 # 3i in %r10
LEAQ (%r8,%r10, 8), %r10 # 25i in %r10
LEAQ (%r9, %r10, 2), %r10 # 50i+j in %r10
MOVW (%rdi, %r10, 2), %r11 # A[i][j] in %r11
LEAQ (%r9, %r9, 4), %r10 # 5j in %r10
LEAQ (%r8, %r9, 8), %r10 # 40j+i in %r10
MOVW %r11, (%rsi, %r10, 2) # %r11 moved to B[j][i]
```

(b) Array `A` is accessed in row-major order, while array `B` is accessed in column-major order. This means that successive accesses to `A` are spaced two bytes apart. Successive accesses to `B` within the same column are spaced $2 \times 40 = 80$ bytes apart (there are 50 such accesses) before going to the next column. The idea is to keep additional pointer variables that can be incremented appropriately on each iteration, rather than calculating addresses from scratch each time.

Fleshing out the rest of the solution is left as an exercise for the student.

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.

Solution: Here is the code generated by `gcc` on my Mac for the original version of the code, lightly edited for clarity.

```
movl %edi, %eax
andl $1431655765, %eax       ## imm = 0x55555555
movl -4(%rbp), %ecx
shrl $1, %ecx
andl $1431655765, %ecx       ## imm = 0x55555555
addl %ecx, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
andl $858993459, %eax        ## imm = 0x33333333
movl -4(%rbp), %ecx
shrl $2, %ecx
andl $858993459, %ecx        ## imm = 0x33333333
addl %ecx, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
movl -4(%rbp), %ecx
shrl $4, %ecx
addl %ecx, %eax
andl $252645135, %eax        ## imm = 0xF0F0F0F
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
shrl $8, %eax
addl -4(%rbp), %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
shrl $16, %eax
addl -4(%rbp), %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
andl $63, %eax
```

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?

Solution:

(a) The matching `pushq` and `popq` instructions show the save/restore of callee-saved registers. Only `%rbp` is saved there. The instruction `subq $32, %rsp` shows the extension of the stack frame by another 32 bytes. There is a recursive call to `bar`, which means that eight more bytes will be pushed on the stack for the return address. Since the prototype for `bar` has one parameter, there is no argument build area. Of the 32 bytes, eight are used for the caller-saved register `%rdi`, eight are used for an unnamed temporary that is used to ensure that the result value is found in the same memory location when the two execution paths converge at the epilog block, and the remaining sixteen bytes are unused.

All of the stack addressing is done using negative offsets from `%rbp` rather than positive offsets from `%rsp`, which may be a little confusing.

(b) If you've reverse-engineered the function correctly, you'll know that it's just computing factorial. Knowing the stack layout from above, you can show the details of the single stack frame.

(c) The difference here is that there will be two stack frames, one in which the argument is 2 and the other in which the argument is 1 (base case of the recursion). Make sure you draw both stack frames.

(d) When `bar` is called with an argument value of 2, the return value is 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);
}
```

Solution:

(a) `fn1` is a leaf procedure, so no caller-saved or callee-saved registers will be saved/restored.

(b) Even though `fn2` is not a leaf procedure, no caller-saved or callee-saved registers will be saved/restored. This is the case because the argument registers can be overwritten in-place and the result is returned immediately after the call to `fn1`.

(c) In this case, the parameter values are needed after the call to `fn1`, so the registers `%rdi` and `%rsi` are saved/restored around the call as caller-saved registers.

(d) In this case, note that the call to `fn1` reverses the order of the arguments, but that `x` is needed in the following call to `fn2`. So the incoming `%rdi` has to be saved as a caller-saved register. Since the value of `y` used in the call to `fn2` is the returned value from the first call, the incoming `%rsi` value doesn't need to be saved.

(e) In this case, `%rdi`, `%rsi`, and `%rax` need to be saved across the call of `fn1` as caller-saved registers.