# *CSE 351 –Winter 2011*
# *Midterm Key*

***NOTE: These answers are much longer winded than was required for full credit.***

1. *[4 points]*
   Complete the memory layout diagram below, then **for each section of memory that you have added** state briefly what it is used for. (I have started the diagram by indicating that the program's machine instructions are loaded starting at location 0.)

| Instructions | *Literals* | *Static Data* | *Heap* | | *Stack* |
|---|---|---|---|---|---|

   *Literals – used to hold immutable (unmodifiable) data, such as literal strings.*
   *Static Data – holds (mutable) static variables, e.g., globals.*
   *Heap – holds dynamically allocated (new'ed) variables.*
   *Stack – holds procedure call frames / activation records: saved registers, arguments, automatic (local) variables, and other temporary space (whose lifetime matches that of the procedure invocation).*

2. *[4 points]*
   The following C code prints `20 20`. Why?

```
int* foo() {
   int fooResult = 10;
   return &fooResult;   // return a pointer to fooResult
}
int* bar() {
   int barResult = 20;
   return &barResult;   // return a pointer to barResult
}
int main(int argc, char* argv[]) {
   int* fResult = foo();
   int* bResult = bar();
   printf("%d  %d\n", *fResult, *bResult); // %d prints an int
   return 0;
}
```

   *When foo executes, fooResult is allocated on the stack, and so a pointer to a stack location is returned. When foo returns, its frame is popped off the stack. When bar executes, it allocates a frame, using the same stack memory foo just used. barResult is allocated at the same address fooResult had been allocated at, so fResult and bResult (in main) contain the same address. The last value written at that address before the printf call was 20, so 20 is printed twice.*

   *[Note: it is ALWAYS a serious error to return a pointer to stack allocated space. gcc*

*prints a warning if you try to compile this code.]*

3. *[3 points]*
Fill in the condition below so that the statement prints `yes` if the integer variable `sum` is divisible by 4 and prints nothing otherwise. **You are not allowed to use the division (/) or remainder (%) operators.**

```
if (!(sum & 0x3)) printf("yes");
```

*sum is a multiple of four if the lower order two bits are zero (just like a decimal number is a multiple of 100 if the lower order two digits are 0).*

4. *[3+3 = 6 points]*
Suppose the following C code is compiled into y86 instructions:

```
int foo() {
      int value[100];
      … <code that operates on all elements of value> …
}
```

(A) Briefly explain why the most current value of `value[20]` will sometimes be in a register and sometimes in memory (as this program executes).

*To operate on the values of variables, they must be in registers. There are too many variables to all fit in registers, though, so the code from time to time has to move a variable's value from memory into a register, work on it, and write it back to memory.*

(B) Give (as assembler) a y86 instruction that copies `value[20]` from memory into register 3, assuming that `foo()` saves all callee saved registers.

*value is a local variable, so it's addressed using the frame pointer register (%ebp). Moving towards lower addresses from where the fp register points, there are 3 saved registers, followed by the value array, laid out so that value[0] is at the lowest address. The displacement **in words** is therefore -3 (the saved registers) + -80 (value[20], value[21], …, value[99]). The displacement in bytes is therefore -83*4 = -332*

*mrmovl -332(%ebp), r3*

5. *[3 points]*
In C, type `char` is a signed, 8-bit integer. With that in mind, what does this output?

```
char a = 100;
char b= 30;
char c= a + b;
printf( "%d\n", c);  // The %d means "print as a decimal integer"
```

*The maximum signed 8-bit value is $01111111_2 = 1 + 2 + 4 + 8 + 16 + 32 + 64 = 127$, so this sum overflows. On overflow, we wrap to the most negative value, -128 and the count up*

*from there,, so 127+3 results in -126.*

6. *[2 + 2 = 4 points]*
(A) IEEE 754 is a standard for floating point that specifies both the bit-level representation of floats and the results of operations on them. Why is it useful to have a standard for this? For instance, what would likely "go wrong" if there were no standard?

*Without a standard, different processor manufactures could (and did) use different floating point representations, and dealt with roundoff differently. That meant that a program compiled and run on one processor might easily get different results if compiled and run on a different processor. (So, the key idea is portability of code across processors.)*

*Note: Let's consider processors not adhering to a standard. Two programs compiled for the same processor are extremely likely to use the "native floating point format" (the one that processor's hardware uses), and so can interoperate. Code running on one processor communicating over a network with code running on a different processor type must be very concerned about representations of all data – for instance, endianness will affect integers, even if both processors use 2's-complement (which they will). Translation between distinct data representations will necessarily be addressed (although it might not be possible to represent all values of one processor on the other, due to distinct choices for the number of mantissa bits, for instance).*

(B) Why is it most likely a mistake to compare two floating point variables for equality?

*Float point operations can (and commonly do) involve roundoff errors – imprecision due to the finite number of bits available to represent values. Therefore, two computations that mathematically should result in the same value may result in slightly different values when done using a finite precision arithmetic (i.e., floating point).*

7. *[4 points]*
Suppose we want to include the notion of "default arguments" in C, as illustrated by the following code. This line declares a function that takes four arguments, two of which have defaults:

```
int mySub( int arg1, int arg2, int arg3=0, int arg4=7);
```

Here are some invocations of that function:

```
rval = mySub( 7, 12, 4, 8);   // legal, with the usual meaning
rval = mySub( 7, 12, 4);      // legal – arg4 = 7
rval = mySub( 7, 12 );        // legal – arg3 = 0, arg4 = 7
```

Does the existing gcc/x86 procedure call convention have to be modified if default arguments are allowed? **If so, briefly explain why. If not, briefly explain why not.**

*This is purely a compile time issue, and so no change to the procedure call convention is required. As part of normal type checking, the compiler must have access to the type signature of mySub (which includes the default argument information) while compiling any call to it. It therefore can recognize that the argument list given in a call is shorter than the 4 arguments required. In those cases, it simply re-writes the call, by supplying default values for the missing arguments, and then compiles code for the call as always.*

*For instance, the compiler would first replace* mySub(7,12) *with* mySub(7,12,0,7) *and then compile the latter.*

8. *[4 + 4 = 8 points]*
Imagine a processor architecture "just like" y86 except that there are 32 general purpose registers, rather than just 8.

(A) Give a specific, convincing argument for why the processor with 32 registers might execute C programs **faster** than the processor with 8 registers.

*If code uses more variables than there are available registers, values for those variables must be moved between memory and registers as the code runs. Those memory operations are expensive. Having more registers means that more variable values can be kept in them, (most likely) reducing the number of memory reads/writes required by the compiled code.*

(B) Give a specific, convincing argument for why the processor with 32 registers might execute C programs **slower** than the processor with 8 registers.

*There are three answers. Only the first two come from this class, but the third was given often enough that I assume it has been discussed in some non-prerequisite class (370?):*

*(1) Instructions require 5 bits to name a register,meaning existing y86 machine instructions have to be made longer than they are when there are only 8 registers. Longer instructions means more memory has to be moved just to fetch the instruction stream, leading to slower execution.*

*(2) The procedure call convention has caller-saved and callee-saved registers. Inevitably, some callers/callees will save registers that didn't need saving, because the compiler doesn't have enough information to minimize register saving for every call. The more registers there are, the more that may be needlessly saved, and so the slower the execution in some cases that perform a lot of procedure calls.*

*(3) As the register file (the bank of registers) gets larger, access time to retrieve the value of a single register gets longer. Eventually, the overhead of longer retrieval time on every register fetch outweighs the benefits the compiler can obtain by having more registers in which to stuff currently used variable values.*

9. *[4 points]*
The declaration of `myGlobal` in the following code **is not executable**, while the declaration of `myLocal` **is executable** – that is, the compiler does not emit any instructions to be executed at run time for the former, while it does for the latter. **Briefly explain why (for both cases), and how** `myGlobal` **is initialized if there are no run time instructions for it.**

```
int myGlobal = 10 + 7;

int mySub() {
        int myLocal = 0;
        …
 }
```

*myGlobal is a static variable, meaning a decision about what memory location it will occupy is made before execution. The compiler can (and does) evaluate the expression* 10+7 *at compile time. When the .exe file for the code is created, it contains an indication that the memory location for myGlobal should be initialized to 17 on load. (This is exactly like the data (i.e., the array + pointer) you created in your "code file" in HW1.) So, no instruction needs to be emitted to initialize myGlobal, as it is initialized by the process of loading the program.*

*In contrast, myLocal is allocated on the stack each time mySub is invoked. It's impossible to know at compile time what those locations will be, and so impossible to find a way to initialize them using the loader. Instead, they must be initialized once their location is know, i.e., at run time. That requires that an instruction be emitted to perform that initialization.*

10. *[*3 *points* + 2 optional extra-credit points*]*
Consider the following (legal) C code:

```
typedef struct {
      int partNumber;
      int numInStock;
} PartStruct;

PartStruct findPart(int partNum);    // method returning a struct

int main( int argc, char* argv[]) {
      PartStruct myPart;
      …
      myPart = findPart( 302845 );
      …
}
```

(A) What is there about the statement calling `findPart()` that presents a problem for the compiler, assuming that (a) it uses the gcc/x86 subroutine call convention, and (b) all returns are return-by-value?

*The procedure call convention (as we've talked about it) passes back the return value using a register. findPart needs to return a struct, and the struct is larger than a register can hold. (So, we can't use the convention that the caller finds the return value in a register when the call returns.)*

(B) **OPTIONAL EXTRA CREDIT (2 points)**
Suggest how the compiler can manage to compile code like this, keeping in mind that `main()` and `findPart()` may be located in different files.

*The compiler is aware, both when compiling the caller and the callee, that the callee is returning a struct. It therefore compiles the caller to allocate stack space to hold the return value of the call, just before making the call. When compiling the callee, it knows that the caller has allocated space just before the call, and so the stack memory just after the saved return address is that space. Therefore, the compiler generates instructions in the callee that put the return value in that stack space (rather than a register) and generates instructions in the caller that look for the return value there (rather than in a register).*

*Note that the caller must allocate this space; the callee cannot. The reason is that stack space used to hold the result, if allocated by the callee, would be part of the callee's frame, which is popped when the callee returns.*

*(There were many answers that suggested return-by-reference, but didn't address the issue of how/who allocated the space that was being referenced (which is critical). A couple people suggested return-by-reference to heap allocated space; that was graded as correct, since it could be made to work, even though it is much, much slower than the method outlined above.)*