

# Languages for Embedded Systems

## Lab assignments

Prof. Stephen A. Edwards  
Columbia University, USA  
sedwards@cs.columbia.edu  
<http://www.cs.columbia.edu/~sedwards>

I have installed lcc, Java, Icarus Verilog, and SystemC 2.0.1 in my account on the class machines. A number of environment variables (e.g., PATH, CFILES, etc.) need to be set to access them. The easiest way to do this is to source the “setup” script that sets the appropriate variables, i.e.,

```
-bash-$ . /home/edwards/setup
-bash-$ which lcc
/home/edwards/bin/lcc
```

1. There are often many different ways to implement the same functionality in assembly language. We will illustrate this using two C compilers installed on the class machines: lcc by C. W. Fraser and D. R. Hanson, and gcc from the GNU project. While both produce code for the x86, they can produce different, although equally correct, results.

Create a file containing the following C program. (You can type it manually or copy it from `~edwards/examples/euclid.c`)

```
int euclid(int m, int n)
{
    int r;
    while ( (r = m % n) != 0 ) {
        m = n;
        n = r;
    }
    return n;
}
```

Ask the C compiler to produce assembly code with and without optimization:

```
lcc -S euclid.c
mv euclid.s euclid.lcc.s
gcc -S euclid.c
mv euclid.s euclid.gcc.s
gcc -O -S euclid.c
mv euclid.s euclid.gcc-O.s
```

Compare the four versions of the program. How does the output differ? What instructions have the two compilers chosen? Have the two compilers ordered instructions differently? What effect does the -O flag have on the output? Does it seem that one compiler does a better job optimizing than the other?

Add the following main function in a file called euclid-main.c

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int count;
    int i, j;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s max\n", argv[0]);
        return 1;
    }

    count = atoi(argv[1]);

    for ( i = 2 ; i < count ; i++ )
        for ( j = 2 ; j < count ; j++ )
            euclid(i,j);
    return 0;
}

```

Compile the two together and time the result

```

lcc -o euclid.lcc euclid.c euclid-main.c
time ./euclid.lcc 500

```

Adjust the number of iterations (500 in this example) so it takes between 1 and 2 seconds. The goal here is to run the program long enough to be easily measured, but no longer.

Compare the time it takes for that same number of iterations under all three combinations of compilers and optimizations. Run each a few times and average the result to get more accurate numbers. Report the times you measure.

Which compiler/optimization flag produced the fastest code? Can you see from the assembly source why this is?

Annotate (write comments on) the assembly language listings to really understand what is happening.

## 2. Concurrent Java programs

The Java compiler “javac” and the Java interpreter “java” both reside in /home/edwards/j2sdk1.4.2/bin, which should be part of your \$PATH environment variable. The “setup” script mentioned at the top should take care of this.

The most basic Java program is “Hello World,” a copy of which is in /home/edwards/examples/hello.java.

```
class hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Compile this with “javac hello.java,” which should produce a file “hello.class.” Make sure “.” is part of your \$CLASSPATH environment variable (again, “setup” should have done this) and then execute the program with “java hello,” which should print “Hello World!”

- (a) Write a threaded FIFO buffer for integers in Java.
- (b) Use it in a simple program with two threads. One thread should put successive integers (i.e., 1, 2, 3, etc.) in a buffer. The other should repeatedly remove an integer from the buffer and print it.
- (c) Modify your program to create two copies of the sequence generator, each feeding into the same buffer. Does your program print 1 1 2 2 3 3 or something else? Modify it so it does.

The point of this exercise is not to write a program that happens to work, but one that will always work regardless of the Java implementation.

Do not use `sleep()`.

In all cases, show me see your source code as well as program output.

### 3. Verilog programming.

I've installed Icarus Verilog, which is a free, fairly complete (but not that fast) Verilog simulator available on the web.

As an example, running "iverilog -o testxor testxor.v" on the following program (in ~edwards/examples/testxor.v) produces an executable called "testxor."

```
module testxor;
  reg a, b;
  wire c;

  xor (c, a, b);


  initial begin
    $monitor($time,,, "a b c: %b%b%b", a, b, c);
    #10 a = 0;
    #10 b = 0;
    #10 a = 1;
    #10 b = 1;
    #10 a = 0;
  end
endmodule
```

Running "testxor" produces

```
0 a b c: xxx
10 a b c: 0xx
20 a b c: 000
30 a b c: 101
40 a b c: 110
50 a b c: 011
```

- (a) A seven-segment decoder converts the first ten four-bit binary numbers into seven on/off signals controlling the seven segments of a numeric display as shown below.

0 1 2 3 4 5 6 7 8 9



Write a structural Verilog module for a circuit built from NAND gates that generates the output for the "a" (topmost) segment of a seven-segment display. Include a testbench like the one shown above and show me both the Verilog source file and the output from the testbench.

- (b) Write the smallest behavioral Verilog module you can that decodes every segment. Again, show me the source and the output from the testbench showing every number is decoded properly.
- (c) Hook two copies of your decoder module to a behavioral module that counts synchronously (i.e., every "posedge clk") from 00 to 99 in BCD and show me the source and output from your testbench for the numbers 00 to 12.

#### 4. SystemC programming.

Repeat part 3. but use SystemC.

To get you started, examine the SystemC version of testxor.v in `/home/edwards/examples/testxor.cpp`. Normally, a SystemC model would consist of many header files (e.g., `.h` files) and C++ source files (e.g., `.cpp` files), but this example comprises only a single file.

Compile and run this example as follows:

```
$ c++ -o testxor testxor.cpp -lsystemc -lm
$ ./testxor
time: 30 a b y: 1 0 0
time: 30 a b y: 1 0 1
time: 40 a b y: 1 1 1
time: 40 a b y: 1 1 0
time: 50 a b y: 0 1 0
time: 50 a b y: 0 1 1
```

(I have omitted some warnings and credits: you will see more junk.)

Notice that the “display” module prints things every time something has changed, not just at the end of a simulation cycle, so we see three spurious outputs, i.e.,

```
time: 30 a b y: 1 0 0

time: 40 a b y: 1 1 1

time: 50 a b y: 0 1 0
```

that don’t correspond to the proper behavior of the XOR gate. In fact, they are there because the inputs to the XOR just changed but the XOR gate has not had a chance to observe them yet.

There are more SystemC examples in `/home/edwards/systemc/examples/systemc`.