Transmogrifier C

Dave Galloway

Department of Electrical and Computer Engineering
University of Toronto

September 1997

Introduction

Transmogrifier C (or tmcc) is a compiler for a simple hardware description language [1]. It takes a program written in a subset of the C programming language, and produces a circuit that will implement the program. The circuit is intended for a Xilinx XC4000 series FPGA, but other FPGAs, CPLDs or even ASICs could be used. This manual describes version 3.1 of tmcc.

Transmogrifier C borrows its name from the Transmogrifier series of field-programmable systems built at the University of Toronto [2] [3]. The Transmogrifier-1 (or TM-1) contains four Xilinx XC4010 FPGAs, two Aptix AX1024 field-programmable interconnect chips and four 32Kx9 SRAMs. The Transmogrifier-2 is a modular machine containing as many as 32 Altera 10K50 FPGAs, 64 I-cube IQ320 interconnect chips and 8 Mbytes of SRAM.

Example

Here is a simple tmcc program that makes the 8 LEDs on the Xilinx 4000 demo board count up:

```c
#pragma intbits 8

main() {
    int lights, count;

    outputport(lights, 60, 59, 58, 57, 66, 65, 62, 61);

    count = 0;
    while(1) {
        count = count + 1;
        lights = ~count;
    }
}
```

The program starts with a #pragma statement telling the compiler that integer variables are eight bits long. It declares two of these variables. The outputport routine tells the compiler that the lights variable is really a set of 8 output pins on the chip, and specifies the pin numbers. The program then goes into an infinite loop, incrementing the count. The LEDs on the demo board are wired so that a 0 output turns them on, and a 1 output turns them off. To reverse this, we use C’s ~ complement operator.
Running tmcc

To compile the sample program, call it counter.c, and run:

```
tmcc counter.c
```

The compiler will produce a counter.xnf file, then pass it through the Xilinx ppr and makebits programs to produce a counter.lca and a counter.bit file. All other intermediate files will be removed. If you want to save them, use the -v flag on tmcc.

The compiler currently tells ppr to place and route as quickly as possible, without doing any optimization. If you want the standard ppr optimization to occur, use the -O option on tmcc.

If you just want to see the xnf output, and not run ppr, use the -S flag. The optional -p partname flag can be used to tell tmcc what Xilinx XC4000 part you are using. The compiler will put that information into the xnf file that it generates, for the use of other tools. The compiler also supports these normal C compiler flags: -U, and -D.

Differences from Standard C

These C operators and keywords are implemented by the tmcc compiler:

```
!  ! =  &  &&  &=  ()  +  
++ += = -= == < <<
<= <= == > >= >>
>>= ^ ^= break else if int
return while {} | |= || ~
```

These C operators and keywords are not implemented by the tmcc compiler:

```
"strings" % % = (casts) *
*= , -> . /
/= : ? [ ] auto
case char continue default do
double extern float for goto
long register short sizeof static
struct switch typedef unary & unary *
union unsigned
```

In other words, tmcc has integer variables, constants, expressions and assignment statements. It has if statements, while loops and function calls. You can also use any of the cpp # directives and macros. It does not have multiply or divide, arrays, pointers or structures. You can not use recursion.

Some of the simple omitted stuff may get added to tmcc in the future.
Extensions to C

```c
#pragma intbits nnn
```

The `intbits` pragma sets the number of bits stored in each integer. Any integer variables declared after the `intbits` directive will have `nnn` bits in them. You can have any number of `intbits` directives in your program. They only affect the integer declarations up to the next `intbits` directive. The default number of bits in an integer is 1.

Input, Output and Bus Ports

A variable can be associated with a set of output pins with the call:

```c
outputport( variablename [ , nnn { , nnn } ] );
```

Any assignments to that variable from that point on will cause the named pins to take on the new value. They will continue to assert that value until they are explicitly changed. Trying to use the value of the variable in an expression is undefined.

The pin numbers are listed with the least significant bit first. If you do not specify the pin numbers, `tmcc` will leave them undefined in the output, and the Xilinx placement and routing software will pick them for you. Non-numeric pin numbers can be specified using double quotes, for example:

```c
outputport( variablename, 23, "J7", "K8", 16 );
```

A variable can be associated with a set of input pins with the call:

```c
inputport( variablename [ , nnn { , nnn } ] );
```

Any reference to the variable after the `inputport` statement will return the value on the given input pins. Assignments to the variable are not allowed. As in the `outputport` case, the pin numbers may be left undefined.

A variable can be associated with a set of bi-directional input/output pins with the call:

```c
bus_port( variablename [ , nnn { , nnn } ] );
```

Any reference to the variable after the `bus_port` statement will return the value on the given pins. Any assignment to the variable after the `bus_port` statement will drive the pins to the new value. As in the `outputport` case, the pin numbers may be left undefined.

A `bus_port` will start out in input (ie: tristate) mode, and will not be driven. An assignment to a `bus_port` variable will cause the pins to go into output mode. They will stay in output mode until the program puts them back into input mode with the call:

```c
bus_idle( variablename );
```
Timing and Clock Ticks

Tmcc generates a simple synchronous design. It has one clock, and all flipflops change on the same edge of the clock.

Tmcc will attempt to stuff as much of your program as it can into the current clock period. Multiple assignment statements and even if statements will all get packed into the current clock period. It will only stop doing this and wait for the next clock tick at any of these points in the program:

- the top of a `while()` loop
- a function call

For example, suppose we want to raise an output for 2 clock ticks and then lower it. The following program:

```c
main() {
    int out;
    outputport(out);
    out = 1;
    out = 1;
    out = 0;
}
```

does not work. All 3 assignments will be packed into the same clock tick, and only the last one will have any affect. The port will be set to 0.

To do it correctly, we need to do this:

```c
main() {
    int out;
    #pragma intbits 2
    int count;
    outputport(out);
    count = 2;
    while(count) {
        out = 1;
        count = count - 1;
    }
    out = 0;
}
```

The while loop will be executed twice, taking one clock tick in each case. The output port will be set to 1 for 2 clock ticks, and 0 thereafter.
Although tmcc attempts to pack as many statements as possible into the same clock tick, it still works like C. Statements are executed in the order that you write them in. Variable assignments take effect immediately, although they may not show up on an output pin until the following clock cycle. This code:

```c
    temp = a;
    a = b;
    b = temp;
```

will exchange the values of a and b, just as it does in C. The exchange will take place in one clock cycle, and no flipflops or other circuitry will be generated for temp.

**Clock Source**

By default, tmcc gets the clock for the circuit from the XC4000 series internal oscillator, running at 15 Hz [sic]. This is suitable for testing the compiler, but useless for anything else.

If you want to get the clock from a different source, define a signal called CLK in your own xnf file, and compile the program this way:

```
    tmcc program.c myclock.xnf
```

For instance, this xnf file will use a clock signal from an external pin named FPSCLK:

```
    SYM, CLK-AA, BUFGP
    PIN, I, I, FPSCLK
    PIN, O, O, CLK
    END
    EXT, FPSCLK, I
    SYM, STARTUP, STARTUP
    PIN, CLK, I, CLK
    END
```

If you want to use a name other than CLK for the clock source, use the `-cYOURCLOCKNAME` option. A simple `-c` option will leave the clock name as CLK, but will not generate the default 15 Hz clock circuitry.

**Multiple Threads of Control**

A simple tmcc program has one thread of control, which starts at the beginning of the main() routine and continues from there. Many circuits have to do several tasks simultaneously, for instance: handling the bus protocol for an input and output bus, talking to several RAM chips, and doing some real-time computation. Rather than try to cram several different parallel processes into the same program by hand, tmcc allows you to have multiple threads of control, one for each independent task.

To do this, write a separate tmcc program for each thread, compile them separately and then merge the resulting xnf files. A circuit with 3 parallel threads can be produced like this:
Each of the tmcc programs has its own main() routine, functions and variables. To ensure that temporary variable names do not conflict between the separate compilations, use the -Tnnn flag on tmcc, which will modify the names in the output net list. The -c flag prevents the compiler from generating multiple copies of the default clock circuit, one for each thread.

The separate threads of the computation can communicate using input and output port variables. Such port variables should have one writer, and one or more readers. The writer thread declares the variable to be an output port. The reader threads declare the same variable name to be an input port. A call to the portflags() routine can be used to produce port variables that do not have external pins, and are suitable for communicating between threads.

Modifying Input and Output Port Semantics — portflags()

An input or output port has a set of attributes, which can be set with the call:

```c
#define PORT_WIRE 0x0
#define PORT_PIN 0x1
#define PORT_REGISTERED 0x2
#define PORT_PULLUP 0x4
#define PORT_PULLDOWN 0x8

portflags( variablename, constant_expression );
```

PORT_PIN means that the port needs a pin on the outside of the chip. PORT_REGISTERED on an input port means that the input signal will be captured in a flipflop and the output of the flipflop will be used as the variable by the rest of the circuit. PORT_REGISTERED on an output port means that the output signal will be saved in a flipflop, and the output of the flipflop will drive the external circuitry. PORT_PULLUP will enable a pull up resistor on an external bi-directional pin. PORT_PULLDOWN will enable a pull down resistor on an external bi-directional pin. PORT_WIRE means none of the above, and acts like a wire.

The default attributes of an input port are: PORT_PIN. Output ports are: (PORT_REGISTERED|PORT_PIN).

Output Formats

The -target flag specifies the output netlist format. The default -target xc4000gates will generate a simple XNF (Xilinx Netlist Format) file that uses AND, OR and INV gates. This format can be read by several FPGA synthesis and optimization CAD tools.

The compiler performs a technology mapping step as it is compiling a program, and converts the circuit to a network of 4 input lookup tables and flipflops. The -target xc4000roms flag will generate a more compact output format with each lookup table expressed as a 16x1 bit ROM. The -target xc4000eqns flag will generate a more readable compact format with each lookup table expressed as a
Boolean equation. The -target flex8000 flag will generate an XNF file using AND, OR and INV gates that is suitable for the Altera Flex 8000 parts, and can be read by the Altera MaxPlus software.

The output circuit is a single clocked synchronous circuit, with a "one hot" state encoding scheme. A 1 bit port variable with the name xxx will be called xxx in the circuit. An n-bit port variable called xxx will be called xxx-0, xxx-1, and so on. Temporary variables generated by the compiler will be called TTT_NNN_BBB_LMMMstring, where TTT is the thread number (usually 0), NNN is the line number in the program where this variable was produced, BBB is the bit number, MMM is a unique id, and string is some indication of what the variable is for.

**Circuit Size and Speed**

The compiler will generate carry select adders and subtractors by default. These adders are usually almost twice the speed of a simple ripple carry adder, but are roughly 50% larger in area. If you are more concerned about size than speed, use the -fno-carry-select flag to force the compiler to use ripple carry adders instead.

The -dverbose flag will print an estimate of the circuit’s size and speed on the standard error output. It will include the number of lookup tables and flipflops needed, and the number of lookup tables encountered in the longest combinational path. The estimate may be incorrect, as ppr may find a different way of implementing the circuit.

**Generating Good Circuits**

Using < or >= comparisons will produce smaller, faster circuits than using <= or >, since the circuit just has to check the sign bit of a subtractor in the first two cases.

Using the same variable for different things in your circuit may produce a larger and slower circuit. Each assignment to a variable adds another input to a multiplexor, and large multiplexors may become the critical path in your circuit. Use different variables for different things, and don’t try to save space by re-using a variable.

**Common Design Errors**

There are several things that can cause a tmcc-generated circuit to fail mysteriously. First, be sure to use xdelay to check that the circuit will run at the desired clock frequency.

Secondly, tmcc assumes that all of the inputs to the circuit will remain stable during a clock period, and will change only when the clock changes. If that is not true for one or more of your inputs, you must specify an input register for those signals using a call to portflags(). For example:

```c
int changing_input;
inputport(changing_input);
portflags(changing_input, PORT_REGISTERED|PORT_PIN);
```

If you do not do this, and an input changes late in a clock period, part of your circuit’s state machine may see the change, and part of it may miss the change, resulting in undefined behaviour. The state machine may halt, or start executing code from two different parts of your program at the same time.
Make sure that each thread has a different thread number, using the -T flag. Never compile two
different threads with the same -Tnnn flag, or their internal nets may be connected together at random.

All tmcc integer variables are signed integers, and they will be sign-extended when used in
expressions. For example:

```c
#pragma intbits 8
int a, b;
#pragma intbits 16
int result;

result = (a<<8) | b;
```

does not work when b is 0xF0, because it will be sign-extended to 0xFFF0 before being or-ed with a.
Instead, use:

```c
#pragma intbits 8
int a, b;
#pragma intbits 16
int result;

result = (a<<8) | (b&0xFF);
```

which works because literal constants are sign-extended only when they are clearly negative.

However, the last fix fails when the variables are more than 31 bits wide, because the compiler can
only handle constants up to 32 bits wide. In particular, 0xFFFFFFFF is identical to -1, and will be sign
extended. To build a larger mask, use a constant expression like ((1<<NBITS) - 1).

**Known Bugs**

Tmcc has not been used extensively, and probably has many bugs. If it is doing something funny,
please contact me and I’ll be glad to look at it. The bugs listed here are the ones I know about right now.
Most of them can be fixed.

Although there is no limit on the size of a variable, integer constants are limited to 32 bits.

If the program exits from main(), the circuit will hang.

If a function changes a global variable, the rest of the circuit won’t see the change until the next clock
tick. There isn’t an automatic clock tick when a function returns. If the function changes a global
variable and immediately returns, you can’t use the value of that global variable until after the next clock
tick.
For More Information

There are more details on the internals of the compiler in [1]. The most recent version of the compiler can be retrieved by anonymous ftp from:


The tmcc World Wide Web page can be found at URL:

http://www.eecg.toronto.edu/EECG/RESEARCH/tmcc/tmcc

References


A Larger Example

This program drives the 7 segment displays on the XC4000 demo board, and makes them count from 0 to 99 repeatedly.

#pragma intbits 8

seven_seg(x)
#pragma intbits 4
    int x;
{
    #pragma intbits 8
    int result;

    x = x & 0xf; result = 0;
    if(x == 0x0) result = 0xfc;
    if(x == 0x1) result = 0x60;
    if(x == 0x2) result = 0xda;
    if(x == 0x3) result = 0xf2;
    if(x == 0x4) result = 0x66;
    if(x == 0x5) result = 0xb6;
    if(x == 0x6) result = 0xbe;
    if(x == 0x7) result = 0xe0;
if(x == 0x8) result = 0xfe;
if(x == 0x9) result = 0xf6;
if(x == 0xa) result = 0xee;
if(x == 0xb) result = 0x3e;
if(x == 0xc) result = 0x9c;
if(x == 0xd) result = 0x7a;
if(x == 0xe) result = 0x9e;
if(x == 0xf) result = 0x8e;
return(result);
}

delay(n)
  int n;
  {
    while(n != 0)
      n = n - 1;
  }

twodigit(y)
  int y;
  {
    int tens;
    int leftdigit, rightdigit;

    outputport(leftdigit, 37, 44, 40, 29, 35, 36, 38, 39);
    outputport(rightdigit, 41, 51, 50, 45, 46, 47, 48, 49);

    tens = 0;
    while(y >= 10) {
      tens++;
      y -= 10;
    }
    leftdigit = seven_seg(tens);
    rightdigit = seven_seg(y);
  }

main()
{
  int count;
  int switches;

  inputport(switches, 28, 27, 26, 25, 24, 23, 20, 19);
count = 0;
while(1) {
    twodigit(count);
    count = count + 1;
    if(count >= 100)
        count = 0;
    delay(switches);
}
}