

An Example Application for the TME Package

Dave Galloway

*Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto*

January 2014

Introduction

This document describes a simple application circuit for the TME package on the DE-4 or DE-5 demonstration board, and explains how to create, compile and run it.

The circuit implements a 32-bit counter. A program running on a workstation will read the output of the counter over the network.

Getting the Source for the Circuit

The DE-4 source for this example circuit is on the EECG machines in the directory `~tm4/tme/examples/de4/counter`. The DE-5 source can be found in the directory `~tm4/tme/examples/de5/counter`.

Sign on to one of the EECG Debian Linux workstations (ex: `laggan.eecg`). Make a new directory, and copy some of the files from the example directory:

```
mkdir counter
cd counter
mydir=`pwd`
pushd ~tm4/tme/examples/de4/counter
cp makefile top.qpf top.sdc top.qsf top_pll.v $mydir
cp counter.v top.v top.ports suncounter.c $mydir
popd
```

Add the following directory to your path, so that the Transmogriker commands are available to you:

```
PATH="/jayar/i/tm4/tme/bin:$PATH"
```

You should probably add that line to your `.profile` file so that the change will be permanent.

The files that you copied include:

<code>counter.v</code>	the circuit that will run on the board, described in Verilog
<code>top.v</code>	a top level wrapper that connects <code>counter.v</code> to the <code>pcie_portmux</code> circuit
<code>top_pll.v</code>	a Quartus megafunction output file describing the circuit's PLL clock generator

<code>top.sdc</code>	a file containing timing constraints for Altera's Quartus software
<code>top.qsf</code>	a file containing compiler settings for Altera's Quartus software
<code>top.qpf</code>	another file that Altera's Quartus software wants to have
<code>top.ports</code>	a description of the inputs and outputs of the circuit
<code>makefile</code>	used to compile the circuit and the program
<code>suncounter.c</code>	a C program that runs on a UNIX workstation and talks to the circuit

Compiling the Example

To compile the circuit into a form that can be used to program the board, type:

```
make bits
```

The circuit source will be compiled by the `quartus_sh` and `qcmd` commands that are supplied with Altera's Quartus software. The results, along with a number of intermediate files, will be placed into your current directory. The whole process will take about 12 minutes.

To compile the program that runs on a Linux workstation and talks to the circuit, type:

```
make linux_counter
```

Downloading the Example to the Board

First, you must decide which demonstration board you want to use. By default, the commands will use the board(s) attached to the skynet.eecg workstation. If you want to use a different board, you must set the `TM_SERVER` environment variable to the name of the workstation that is attached to the board you want, something like this:

```
export TM_SERVER=some_other_machine.eecg
```

At this point, you may want to run the status monitor display on your workstation. Make sure your `DISPLAY` environment variable is set correctly and run:

```
tmstatus &
```

This will produce a new window on your screen which displays the name of the person currently using the board, if any.

If the `tmstatus` command complains about "can't connect to tmemon", it means that someone needs to restart the `tmemon` daemon on the machine that contains the DE-4 or DE-5 board. If this happens to you, sign on to that machine over the network, and run:

```
tmemon &
```

and then try the `tmstatus` command again.

Now use the `tmget` command to reserve the board for 10 minutes:

```
tmget
```

If someone else is using the machine, you will get a message telling you how long you will have to wait:

```
sorry, drg has it (priority 100) for 597 more seconds
```

To download your circuit into the board, run:

```
tmrun
```

Once the FPGA has been programmed, you must make the Linux kernel negotiate PCIe settings with it. Sign on to the machine that has the board. If you are using the DE-4, type these commands (as root):

```
echo 1 > /sys/bus/pci/devices/0000:03:00.0/remove
echo 1 > /sys/bus/pci/rescan
```

If you are using the DE-5, type these commands instead (again, as root):

```
echo 1 > /sys/bus/pci/devices/0000:04:00.0/remove
echo 1 > /sys/bus/pci/rescan
```

Interacting with the Circuit

To communicate with the circuit, run the `linux_counter` program that you compiled earlier:

```
% ./linux_counter
0 1 2 3 4 5 6 7 8 9
```

Every time you run it, you will get another 10 numbers from the counter.

Releasing the Board

You should now release the board so that someone else can use it:

```
tmrelease
```

How the Sample Design Works

The circuit is written in Verilog. The complete `counter.v` file contains this:

```
module counter (
    input  clk,
    input  reset,
    output reg [31:0] result,
    output reg count_ready,
```

```
input want_count);

reg [31:0] count;

always @(posedge clk or posedge reset) begin
    if(reset) begin
        count <= 0;
        count_ready <= 0;
        result <= 0;
    end
    else begin
        if(count_ready && want_count) begin
            count_ready <= 0;
            count <= count + 1;
        end
        else begin
            result <= count;
            count_ready <= 1;
        end
    end
end

endmodule
```

The circuit has one 32-bit output port called `result`, and two 1-bit handshaking variables called `want_count` and `count_ready`. The circuit sits in an infinite loop, waiting for someone to ask for a count. It performs a handshake with the outside world using the `want_count` and `count_ready` variables, increments the counter, and waits for the next request.

The circuit uses the TME ports package to communicate with the outside world. The connections to the outside world are handled by the component named `pcie_portmux`. This component is automatically supplied by the TME ports package. Read *The TME Ports Package* for a description of how this works. The `top.ports` file contains:

#	Name	direction	bits	Handshake_from_circuit	Handshake_from_workstation
result	o	32	count_ready	want_count	

It describes the single 32-bit output called `result` along with the two handshaking variables.

The `top.v` that connects the counter module to the automatically generated `pcie_portmux` component contains:

Input/Output ports omitted to save space ...

```
//
// User stuff begins here
//
```

```
// Tie some unused signals down to suppress warnings.
assign HSMC_SCL = 1'b1;
assign HSMC_SDA = 1'bz;
assign PCIE_WAKE_n = 1'b0;
assign PCIE_SMBDAT = 1'bz;
assign GCLKOUT_FPGA = 1'b0;

// Generate some clocks
wire clk50;
wire clk125;

top_pll top_pll_inst
(
    .inclk0(OSC_50_BANK3),
    .c0(clk50),
    .c1(clk125)
);

// Connect the user's counter.v circuit to the pcie_portmux wrapper that was
// generated by the tme command.

wire [31:0] result;
wire count_ready;
wire want_count;

wire rstn;

assign rstn = PCIE_PREST_n;

pcie_portmux pcie_portmux_inst(
    .port0_circuit_data(result),
    .port0_circuit_ready(count_ready),
    .port0_circuit_wanted(want_count),

    .PCIE_REFCLK_p(PCIE_REFCLK_p),
    .PCIE_PREST_n(PCIE_PREST_n),
    .PCIE_RX_p(PCIE_RX_p),
    .PCIE_TX_p(PCIE_TX_p),

    .clk125(clk125),
    .clk50(clk50)
);

counter counter_inst(
    .reset(~rstn),
    .clk(clk50),
```

```
        .result(result),  
        .count_ready(count_ready),  
        .want_count(want_count)  
    );  
  
endmodule
```

It connects to a DE-4 clock signal called OSC_50_BANK3, and to a number of DE-4 signals with names that start with PCIE_ that connect to the PCIe bus on the DE-4 board. It connects those signals to the pcie_portmux component, and also connects the user's signals between the counter and pcie_portmux.

The DE-5 top.v file is similar, but uses different pin names and PLL settings.

The suncounter.c program that runs on the workstation looks like this:

```
/* A program to test a simple counter circuit, using the ports package */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
main(argc, argv)  
    int argc;  
    char *argv[];  
    {  
        int portresult;  
        int *result;  
        int i, count, j;  
  
        tm_init("");  
  
        if((portresult = tm_open("result", "r")) < 0) {  
            printf("Can't open port result\n");  
            exit(1);  
        }  
  
        count = 10;  
        if(argc>1)  
            count = atoi(argv[1]);  
  
        result = (int *) malloc(count * sizeof(int));  
        if(result == NULL) {  
            printf("Can't allocate memory\n");  
            exit(1);  
        }  
  
        if(tm_read(portresult, result, count * sizeof(int))  
            != (count * sizeof(int))) {
```

```
        fprintf(stderr, "suncounter: error in reading\n");
        exit(1);
    }

    for(i=0; i<count; i++) {
        printf("%d ", result[i]);
        if((i % 10) == 9) {
            printf("\n");
        }
    }
    printf("\n\n");

    exit(0);
}
```

It initializes the ports package by calling `tm_init()`, opens the `result` port with `tm_open()`, reads 10 values of the counter with a single call to `tm_read()` and then prints them out.