Building A Custom System-On-A-Chip

Only a few years ago, we could only dream about building our very own custom microprocessor system on a chip. The manufacturing cost for producing a custom chip is just too prohibitive for a hobbyist or even for a small company. However with programmable logic devices (PLDs) getting larger in capacity and cheaper in cost, it is now very easy to design and produce even a fairly complex microprocessor with all the whistles and bells that you would expect to have in a current day microprocessor in a single PLD. In addition, you can even include all the necessary peripheral components to create an entire system on a single chip. To top it all off, advances in the development software for building microprocessors and integrating system components together is getting to a point where you don’t even need to know about the details of how to put a microprocessor together, or how the different components are connected together. That’s right, you don’t need to know the difference between the datapath and the control unit, or even between an AND gate and an OR gate. All you need to know is what kind of features you want to have in your custom microprocessor and the entire system. You only need to tell the “wizard” that you want this and that, and almost like magic you will have not just the custom microprocessor, but together with all the other system components interfaced correctly, not on a silver platter, but on a PLD.

Using PLDs is only half of the story and many articles have been written about them and how to put your circuits on them. What is even more intriguing is the other half of the story, and that is about the development software for custom microprocessors and custom systems-on-a-chip. Many companies now offer configurable processor and peripheral software cores as part of their development system. With this capability, you only need to tell it what you want to have in your system, and the development tool will produce a fully synthesizable HDL code of your complete system with all the features that you have specified, and all the components interfaced correctly. With this complete HDL code generated automatically in minutes, you can then synthesize it and download it to a PLD. The final product is your fully operational custom microprocessor in a complete system that is not only in your dreams. The icing on the cake is that the total cost is only a few hundred dollars and it takes less than an hour, and not months to produce it.

I will now demonstrate this entire process of building your own complete custom microprocessor system with peripherals on a PLD chip using the Nios configurable processor (1) and the Quartus development software (2) from Altera.

**Basic Features of the Nios Processor**

The Nios processor is a soft core CPU. In other words, it initially exists only as a HDL software description. Not until you synthesize and download it onto a PLD will you have a physical Nios processor. The Nios CPU is a pipelined RISC processor configurable to using either a 16-bit or 32-bit architecture. Since the CPU initially exists only as a HDL description, its many features, as shown in Figure 1, therefore, can be easily changed or configured before it is actually implemented on an actual chip. In fact, since it is only implemented on a PLD, it can be re-configured even after it has been implemented by simply downloading the new synthesized description to the PLD again.
• 5-stage pipelined general purpose RISC processor
• 32-bit or 16-bit architecture
• up to 512 internal general-purpose registers
• optional internal instruction and data cache
• separate instruction and data memory
• simple, complete and customizable instruction set
• powerful addressing modes
• optional hardware multiply unit
• up to 64 vectored interrupts

Figure 1. Nios processor features.

The advantage of using a configurable processor of course is that you can customize it to include only the features and power that you need. However, since you’re not designing a full custom processor from the ground up, but simply selecting or de-selecting the available features of a fully tested and fully feature-loaded processor, the design and testing time is considerably reduced.

System on a Chip

Typical current PLD technologies can implement a circuit with over 79,000 logic elements (LE), and up to 7.5Mbits of RAM. The Nios development board that we will use (see Figure 8) contains the 1S10 Stratix PLD, the smallest member of the Stratix family, with 10,570 logic elements and 920 Kbits of on-chip memory. The fully loaded Nios processor that we will demonstrate uses less than 23% of the logic resources and 60% of the available memory resources of the 1S10. The question is what are we going to do with the rest of the resources? I think you can guess what we will do. We’ll just implement the other system components onto the same PLD too. If we can put all the necessary peripheral logic, I/O drivers, memory, you name it, onto the same PLD along with the CPU, then what you end up with is the complete system on a single chip.

There are many off-the-shelf microcontrollers that you can get now with a lot of stuff built-in. This system-on-a-chip (SOC) idea is not to duplicate that, even though you can. Instead, it is to bring to you, the designer, the capability and ease with which you can customize a complete system, and not just a microcontroller, on a single chip.

Specifying and Creating the System

Building a complex system on a chip is just as easy as building, say a 2-input multiplexer if you have all the necessary building blocks for the two. In both cases, you will use the Schematic Editor in the development system. For the mux, the building blocks are two AND gates, an OR gate, and an inverter. In the editor, you would insert these blocks in, and connect them up with “wires”. For the system on a chip, you would do exactly the same thing, that is, in the editor you insert the blocks in and connect them up. The only difference, of course, is that the building blocks for the system are not simple gates but rather the system components (microprocessor, memory, UARTs, etc.) Nevertheless, in the editor, no matter how complex these components are, they will simply show up as black boxes (rectangles) with connections; just like the simple gates.

So now, if we have these system components as building blocks, then we are all set. The
good news is that, yes, many of these standard system components are already defined and available in the development system library. And many other components that are not in the library can be obtained through third party vendors. Furthermore, since all these components, like the Nios processor, exist initially as HDL descriptions, they can all be easily configured to your liking.

We begin by starting the Quartus system development program with a new project file for our custom microprocessor. Next, we bring up the SOPC Building tool as shown in Figure 2. This tool allows us to define not only the Nios processor, but also the rest of the modules needed for our system. The available modules in the library are listed in the left panel. You simply select the module that you want to have in your custom system and click on the Add button. Each module that you add will have its own customization window. In our sample system, in addition to the 32-bit Nios CPU, we have added some standard components for building a complete computer system. The names and descriptions of these components are listed in the right panel. The modules listed are: the Nios CPU, a boot monitor ROM, 64K onchip RAM, an UART for connecting to the system console, an interval timer, an input port for push buttons, a bi-directional PIO port for interfacing with a LCD, an output port for connecting to LEDs, and another output port for connecting to the seven-segment LEDs. After adding these components for the system, all you need to do is to click on the Generate button to have the complete system automatically generated.

After the successful generation of the Nios system, we are now ready to connect the various components together using the Schematic Editor. During system generation, the SOPC Builder created a symbol for our Nios system, which includes all the internal components that we have added. We can now insert this Nios system symbol into the Editor, along with all the necessary components.
I/O port pins for connecting with the actual external components. Figure 3 shows the window for selecting and inserting the components into the Schematic Editor. The left panel provides a list of all the available components for you to use. Notice that the SOPC Builder has added the nios32 module at the top of the list. This module, of course, is what we have just created in the previous step. By selecting the nios32 component (or any other components in the list), the panel on the right will display the symbol with all the connections for that component. By clicking on the OK button, the selected component will be inserted into the Schematic Editor as shown in Figure 4.

**Figure 3.** Symbol selection to be inserted into the Schematic Editor. The Nios component, which was created by the SOPC Builder, is currently selected. The logic symbol is displayed on the right with all the I/O ports listed.

**Figure 4.** Schematic Editor with the complete Nios system as a single module and all the I/O ports connected.
In addition to the nios32 module, which includes all the system components that we have added, we need to add the I/O connections for interfacing the nios32 module with the actual external components such as the LCD, buttons, and LEDs. Figure 4 shows the main nios32 module connected to all the necessary I/O ports in the Schematic Editor. We will also need to specify exactly which pin on the PLD that these I/O ports must go to, so that the external components are connected correctly. This is done either using the FloorPlan Editor, or running an automated Tcl script.

Now we are ready to compile this complete system. This is accomplished either by selecting the Compile command from the menu or clicking on the Compile button. The compilation process translates this complete system, which at this point exists as HDL source code that was automatically generated by the SOPC Builder and the Schematic Editor, into the actual netlist that realizes the system. During compilation, a progress window shows you the entire compilation progress. After compilation, a message window pops up telling you that the compilation was successful. You can also look at a summary of the compilation and the resulting system as shown in Figure 5. Notice that our complete system uses only 2518 logic elements, which is only 23% of the total for this particular PLD that we are using.

The last thing we need to do is to download this resulting netlist of our complete system onto the PLD. Again, like the compilation step, we simply need to click on the Program button, and after several seconds, this PLD is now our complete custom system.

**Figure 5.** Summary report of a successful compilation of the Nios system.

**Testing the System**

Do we really have a complete computer system on the PLD? Well, we’ll just have to try it out by writing a program for it and running it on the system. The development system comes with the GNUPro Toolkit, which includes a C compiler that is customized for generating code for the Nios processor. In addition, the SOPC Builder, when it was generating the system, also generated the necessary include files (SDK) that are specific for our custom system.

In order to compile and run C programs on our system, we need to start up the console shell, which is a UNIX-like command shell running on the PC. This console shell communicates with our custom system via the serial port on the PC that is connected to the console UART port on
our system. From the command prompt, we use the **nios-build** and **nios-run** utilities to compile a C program, and then load and run it on our Nios system. Figure 6 shows the console shell after executing the **nios-run** command on our HelloWorld test program. The program simply prints the words “Hello world! Nios demo” on the LCD and displays a countdown from 19 to 0 on the seven-segment LEDs. The countdown is also displayed on the console screen. The listing for the helloworld.c test program is shown in Figure 7. Figure 8 shows a snapshot of the actual running of the HelloWorld test program on our custom system with the count at 8 being displayed on the seven-segment LEDs.

**Figure 6.** Trace of the run of the helloworld.c test program that is written in C. The **nios-build** command compiles the C program for our custom microprocessor system. The **nios-run** command downloads the executable code to our system. The code is then run on our custom system.

**Conclusion**

I have shown how a complete custom computer system, including a custom 32-bit RISC microprocessor, can be built very easily and quickly. After synthesis, the complete system is downloaded onto a PLD chip. The result is a complete custom system-on-a-chip. Programs written in C can then be compiled and run on this system.

**Reference**

(1) Nios development board, Altera Corp., (408) 544-7000, [www.altera.com](http://www.altera.com)
(2) Quartus II development software, Altera Corp., (408) 544-7000, [www.altera.com](http://www.altera.com)
// helloworld.c
// February 2003, Enoch Hwang

#include <stdio.h>
#include "pio_lcd16207.h"
#include "nios.h"

void show_decimal(int x){
    // display decimal on the seven-segment LEDs
    nr_pio_showhex((x / 10) * 16 + x % 10);
}

int main(void){
    long i;
    np_pio *led = na_led_pio; // for LEDs
    int ledBit = 1;

    // Start up with a greeting...
    printf("\n\nNow running the Hello World test program.\n");

    nr_pio_lcdinit(na_lcd_pio); // initialize the LCD

    // Now, countdown from 19
    for(i = 19; i >= 0; i--){
        printf("%d ",i); // display count on the console screen
        show_decimal(i); // display count on the seven-segment LEDs
        led->np_piodata = ledBit; // turn on an LED
        if(ledBit == 128)
            ledBit = 1; // wrap around LED
        else
            ledBit = ledBit << 1; // shift LED

        // print message on the LCD
        nr_pio_lcdwritescreen(" Hello World! Nios demo");
        nr_delay(600); // delay for 600ms

        // flash the message
        nr_pio_lcdwritescreen(" Hello World! ");
        nr_delay(600); // delay for 600ms
    }

    nr_pio_lcdwritescreen("*** The END! ***");
    printf("\n\nThe end.\n")
    return 0;
}

Figure 7. The helloworld.c test program listing.
Figure 8. Our custom Nios system implemented on the development board showing a sample run of the HelloWorld test program with the count at 8 and the LCD displaying the Hello World message.