Chiphack: for teens

Silicon chip design for teenagers

Dan Gorringe
Embecosm

Application Note 12. Issue 1
Publication date October 2014

Table of Contents

1. Introduction 1
  - 1.1. What is an FPGA? 1
  - 1.2. Target Audience 1
  - 1.3. Difference to software design 1
  - 1.4. What you will need 1
2. Getting Quartus Going 2
  - 2.1. What is Quartus? 2
  - 2.2. Quartus for Windows 2
  - 2.3. Quartus for Linux 2
3. Getting Something Running 3
  - 3.1. LEDs 3
  - 3.2. Binary 3
4. Computer Logic 5
  - 4.1. Addition 5
    - 4.1.1. Logic Gates 5
5. Counters Projects 8
  - 5.1. Manual Counter 8
  - 5.2. Automatic Counter 9
    - 5.2.1. Clocks 9
    - 5.2.2. Implementing the Counter 10
  - 5.3. Fibonacci Counter 11
6. Our Lock 13
  - 6.1. What is a state machine? 13
  - 6.2. Our first State machine 13
7. *UART* 20
  - 7.1. What is a *UART*? 20
  - 7.2. Hello? world? 20
  - 7.3. Getting it to run on the screen 23
8. OpenRISC and SoC 25
  - 8.1. What is OpenRISC? 25
  - 8.2. What is a SoC? 25
  - 8.3. Installation 25
    - 8.3.1. General System Tools 26
    - 8.3.2. OpenRISC GNU tool chain precompiled for 32-bit linux 26
    - 8.3.3. Quartus Tools 26
    - 8.3.4. Icarus Verilog and *GTKWave* 26
    - 8.3.5. OpenOCD 26
    - 8.3.6. FuseSoC 27
    - 8.3.7. orpsoc-cores 27
  - 8.4. Waves 28
    - 8.4.1. Hello? Again? 28
    - 8.4.2. Waveform 28
  - 8.5. Running on the *DE0 Nano* 29
    - 8.5.1. USB to *UART* patch 29
    - 8.5.2. Programming the board 29
    - 8.5.3. Connecting the debug proxy 30
    - 8.5.4. Rebuilding our program 30
    - 8.5.5. Openning a terminal 30
    - 8.5.6. Connecting the debugger 30
    - 8.5.7. Running Linux on our FPGA 31

Glossary 32

References 33

List of Figures

3.1. Binary representations of numbers 4
4.1. Symbols for different logic gates 6
7.1. USB to *UART* connector. 24
8.1. Example of a wave trace 29

Chapter 1. Introduction

Chiphack [1] is a workshop which teaches the basics of silicon chip design. In this application note it has been redesigned to allow teenagers to learn silicon chip design.

1.1. What is an FPGA?

A Field Programmable Gate Array (FPGA) is basically a code-your-own circuit board in which you can design anything for hardware. To describe our design, we use a Hardware description language (HDL), and in this case we will be using Verilog. The goal of this is for teens to come away with enough know-how to be interested and be able to learn more about both FPGAs and computer science.

1.2. Target Audience

This guide is for teenagers who have an interest in computing. No previous knowledge of FPGAs or even programming/computing is needed, so therefore should be suitable for most people with an attention span and a minimal sense of humour.

1.3. Difference to software design

So what is the difference between this, and learning other computer languages such as Java or C? With Verilog you are learning to work with hardware, where unlike software devlopment, everything happens at once. It therefore needs to be approached differently and strange things may happen if you forget: *Verilog is Parallel*.

1.4. What you will need

1. An FPGA development board. I will be using a Terasic *DE0 Nano* (see [6]). If you wish to use a different board you will may need different tools to begin, though the concepts are common to any FPGA you may use.
2. A laptop or PC. This is used to program the device via USB.
3. An Internet connection
4. Willingness to learn

Chapter 2. Getting Quartus Going

2.1. What is Quartus?

Quartus is the Verilog compiler that we will be using to build our projects.

2.2. Quartus for Windows

For this you will need: a computer running Windows, a CD with Altera's Quartus software[4] , a drink/snack and an Internet connection. (If you don't have the CD to hand, you can grab the software online, making sure to get the web edition.)

1. Put in the CD, and run the **`setup.exe`** to install.
2. Wait for installation to complete. Get snack/drink (this will take ages).
3. Download Altera's Windows USB blaster driver[12] (to save time later).
4. Download the CP210x USB to UART Bridge Driver[9] and PuTTY terminal application [7]. These will be used in the examples to set up a serial connection to the *DE0 Nano*.
5. Download the examples (**`.zip`** format) from chiphack.org[1].

2.3. Quartus for Linux

For this you will need: a computer running Linux, CD with Altera's Quartus software[4], a drink/snack and an Internet connection. (If you don't have the CD to hand, you can grab the software online, making sure to get the web edition.)

1. Put in the CD, and run **`setup.sh`**.
2. Wait for installation to complete. Get snack/drink (this will take ages).
3. Download Altera's Linux USB blaster driver. [10]
4. Download the examples (**`.zip`** format) from chiphack.org[1].

Note

By default, to the board you will need root permissions (very important). Don't be mistaken by being able to follow the rest of the tutorial without doing so. When you first open the tools after installation, the tools will be run as root.

Chapter 3. Getting Something Running

3.1. LEDs

Open Quartus[4], go to **`File > Open Project`** then find the sample project **`DE0_NANO`**[6] from the **`.zip`** you downloaded as part of Getting Quartus Going.

Note

If you are running Linux you will need to rename **`.sdc`** to **`.SDC`**(note capitals).

Next open the **`.v`** file and add it to the project.

If you run this project as is, nothing will happen apart from the LEDs ceasing to glow nicely. Therefore under `REG/WIRE` declarations write:

```
wire [07:00] leds;
assign LED[07:00] = leds;
assign leds = 1;
```

Next run the assembler (very important), followed by the programmer. Ensure hardware is set to USB-Blaster, add the **`.sof`** file and press "start" (for best/any results have the *DE0 Nano* plugged in).

Note

You may have to change the dropdown in the `Tasks` window from the **`Full design`** flow to **`Compilation`**.

3.2. Binary

“There are 10 types of people in this world. People who understand binary, and those who don't.”

On the strip of LEDs you can see that 1 is represented by the first led lighting up. If we change the **`assign leds`** to equal 2 and run it, we see that the second LED lights up, and if we change it to 3 then both the first and the second LEDs light up. This is because these are represented in binary.

Binary is made up of 0s and 1s, and each column represents a different number, the first being 1, second being 2, third being 4, then 8 and 16 and so on. With these numbers you can make any other number for example 19 can be represented as 10011, as 16 + 2 + 1 = 19 (see Figure 3.1).

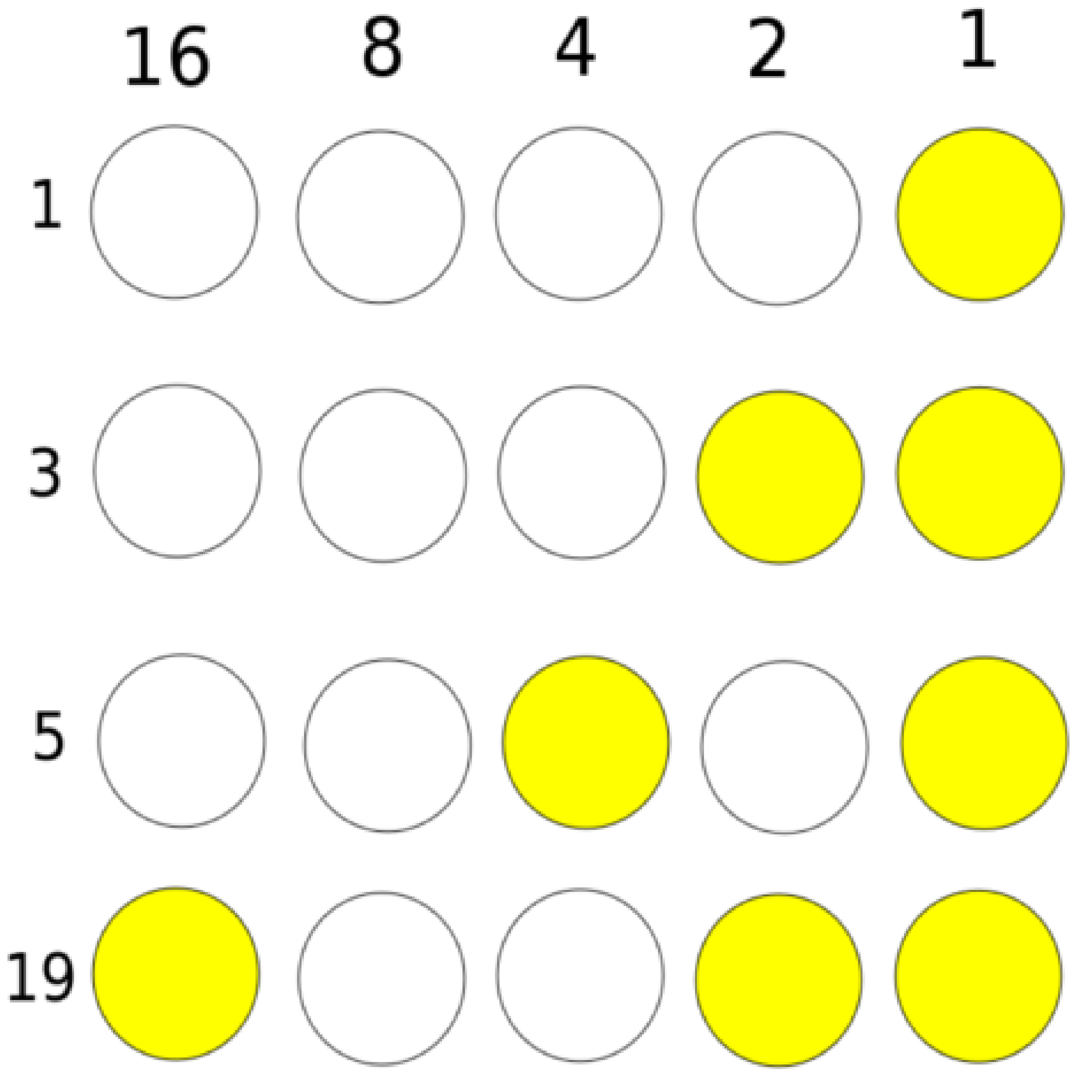


Figure 3.1. Binary representations of numbers

Chapter 4. Computer Logic

When programming in Verilog, you are designing hardware, therefore it is important you can understand how it works, so what is a computer's 'logic'?

To begin with computers are effectively complex circuit boards, they use wires to transfer inputs, however a wire can only be in 2 states: on (1) or off (0).

4.1. Addition

Adding numbers together on a calculator is easy, but how is it done? As we have already found out, computers use binary. Therefore we must first see what the sums would look like in binary, or atleast the inputs and outcomes.

```
01 one
01 plus one
__
10 two
```

```
01 one
00 plus zero
__
01 one
```

```
010 two
010 plus two
__
100 four
```

With this we can spot a pattern, which we can use to make a basic calculator. If there is a single 'one' then it will display that else if there are two 'ones' then you shift over the 'one'. For example:

```
010 two
001 plus one
__
011 three
```

4.1.1. Logic Gates

To then create this we would have to use logic gates. A logic gate is a building block for manipulating inputs into outputs, for example a **NOT** gate will take in a '0' and output a '1'. Other gates, use 2 inputs, though only produce 1 output. For example an **AND** gate will only set its output on if both of its inputs are on. You can probably guess what an **OR** gate does: if at least one input is 1, then the output is 1. You could use an **XOR** gate, which will only output '1' if only one of its inputs are '1' not both.

Note

You can use gates such as **NAND** which is both **NOT** and **AND** so if will output 1 if both input wires are not 1. The same applies for **NOR** and **XNOR**.

Logic gates are normally explained through diagrams, such as those found in Figure 4.1.

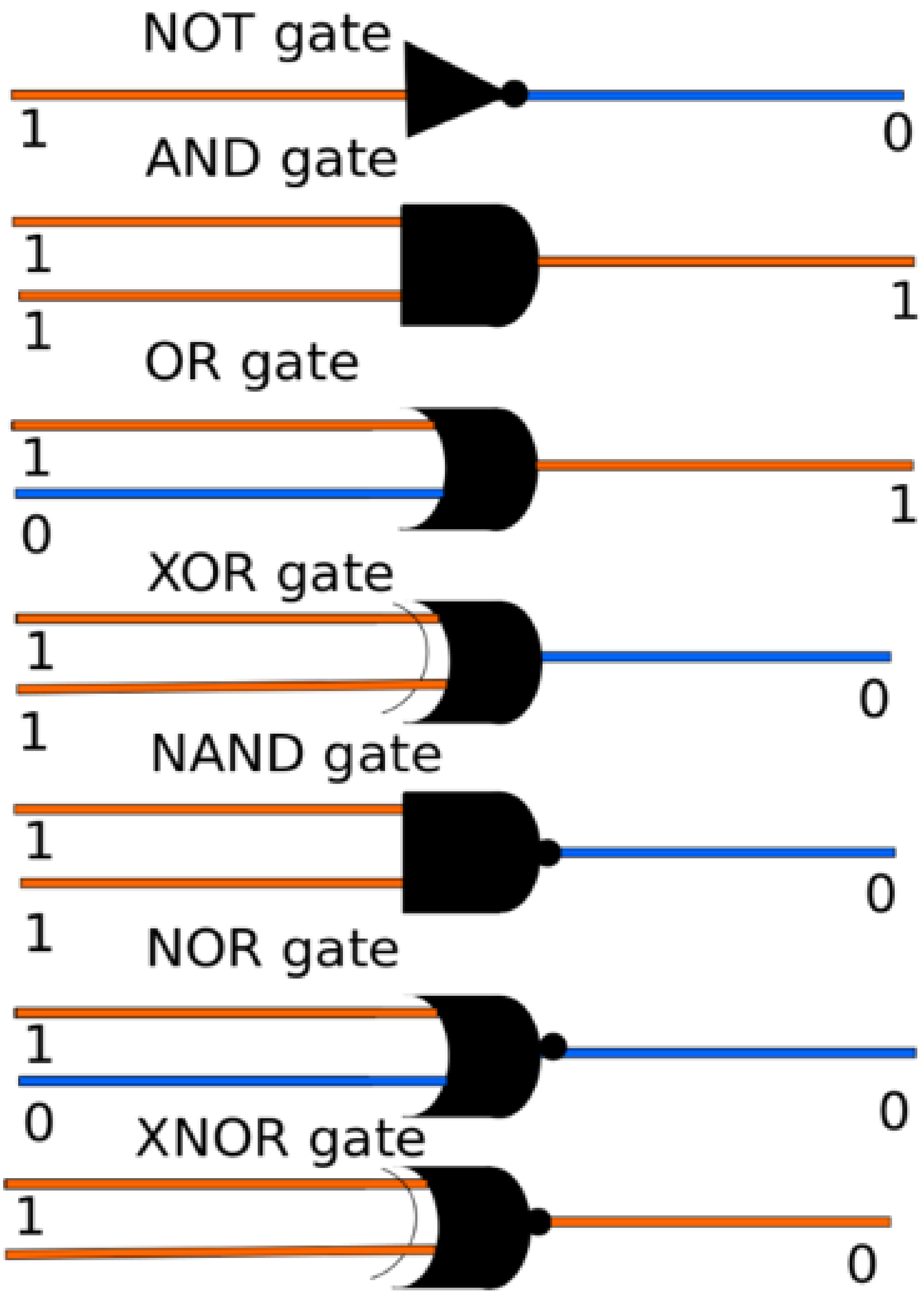
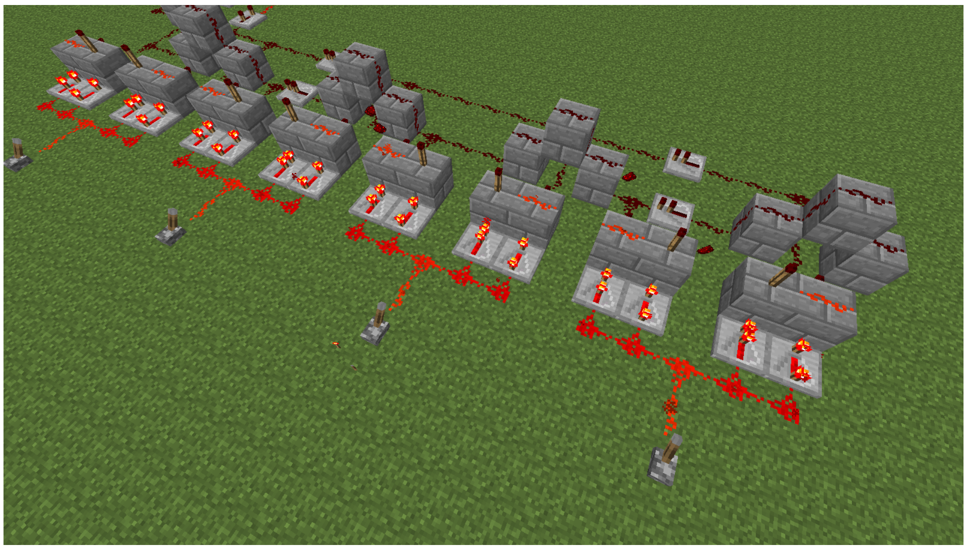


Figure 4.1. Symbols for different logic gates

For our basic calculator, we will be able to add by saying there are either zero, one or two for each binary digit. We will have two inputs per binary digit, so for each digit you can check to see if: its output should be one (with **XOR**) or if both are on (with **AND**), if both are down, add one to the the next digit, if it is on its own, make the output for said wire '1'.

Note
Using the popular game Minecraft, and its redstone mechanic you can create computer logic. It's not too hard to create a working calculator such as this.



Chapter 5. Counters Projects

5.1. Manual Counter

First we will make a counter which goes up on our command alone. For this we need an empty **.v** file, so load up **DE0_NANO.v**[6] and empty it. For this project, we will be learning how to make a **.v** file from the beginning, and to start we need to declare a module, which is done like this:

```
module DE0_NANO(LED,KEY);
```

More generally the following format is used:

```
   module <module name> (<list on inputs and outputs>);
```

First we declare the name of the module to be **DE0_NANO** , as the top-level module needs to be the name of the project. We next declare the inputs and outputs used by the module, fully stating which are inputs and which are outputs.

```
input  [01:00] KEY; // we then declare that there are two buttons
output [07:00] LED; // and 8 LEDs
```

We then need to create our registers and wires, but first it is important to understand what the difference between these are. A wire can not store data, it is either on or off, whereas a register can store data.

```
 wire [07:00] leds_out;
 reg  [07:00] counter;
```

We then need to assign our registers to their outputs:

```
assign leds_out = counter;            // note: a comment is simply two
assign LED[07:00] = leds_out[07:00];    // slashes, and creates a comment
assign add = KEY[01];                 // for the rest of the line
```

Note

We can assign wires to registers straight away. All we need to do is add an equals sign followed by what its going to be assigned to, followed by a semicolon. For example we could have said:

```
wire [07:00] leds_out = counter;
```

Then we need to create an initial statement so that our register starts with a known value, as otherwise it would be random.

```
initial counter = 1

/* Note there are also block comments that comment out the entire
   area between them and is simply a slash and a star.
 */
```

Once we have declared all our wires, registers and infrastructure we have to write the code that does things, which in this case is a counter that works manually, on the press of a key. We start this using an **always** block: within a **begin** and **end**, you state what will happen every **posedge** (positive edge) clock, or whatever you chose (you can also use **negedge**).

```
always @(posedge add) begin
        counter <= counter + 1;
end
```

Note

Here you can see that we have used '<=', which represents "will become". This means that on the next "add", **counter** will be incremented by one. This called a *non-blocking assignment* as it doesn't block execution while it occurs. It does not happen straight away, as this may interfere with other commands in your code, which would then create a race condition in the hardware. For example if you have an **if (count == 1)** but if **counter** = **count + 1** then this would happen straight away and **count** would not equal 1 when you want it to.

Make sure to always use non-blocking assignments.

```
end module
```

The **end module** is not for the **always** block but instead finishes the module we started at the beginning, it is very important to have enough **ends** and **end module**s, as well as keeping them properly indented so your code is easily read. Now when we run this we should get a lovely manual counter. What should happen is that every time that we press down on the add button the variable of count will go up by one and then the leds will display the count.

5.2. Automatic Counter

Since we have created a manual counter and we are getting of bored of trying to find a pen to press down on KEY[01], it is time we made an autonomous counter!

5.2.1. Clocks

For this we will be using a clock, but first what is a clock? (note: Not the one you find on the wall). In hardware, a clock is less of a clock more of a crazy metronome for fans of sabre dance[11], but instead of sound it outputs ones and zeros. A clock changes from 0 to 1 and back inside a cycle. This is what changes when you hear about the 'speed' of a computer, for example in the fastest i7 processor, a cycle will happen 3,800,000,000 (a very big number) times a second! However clocks are not perfect, they don't precisely go up on every cycle instead they slope, therefore making a positive and a negative edge, positive being from 0 to 1 and negative the opposite.

5.2.2. Implementing the Counter

If we take the code we have just written we can change the line which tells us on KEY[01] do this to on **CLOCK_50**. Though first we will need to create **CLOCK_50** as an input of the module. It is named this as of the clock in the chip runs at 50 MHz. So the module will become:

```
module DE0_NANO (LED,KEY,CLOCK_50) // note we do not have to use
                                   // all of the inputs
   input [01:00] KEY;
   input CLOCK_50;
   output [07:00] LED;
```

and now we can change the trigger in the **always** block to **CLOCK_50**.

```
always @(posedge CLOCK_50)
   count <= count +1
```

However if we now run this we will see 255 displayed on the leds, this is because the clock is going so fast that we cannot even see it count, therefore to fix it we will use wider registers: we will change our count register to be 32 bits wide.

```
   reg [31:00] count;
```

If we were to run this we would get the same result as previously. However if we then stated that we wanted the LEDs to only show the output of the highest 8 digits, it will appear as if the clock is going slower. We create this by only selecting a small amount of the register, the highest 8 digits. Imagine that there are also another 24 LEDs to the left/right of your display.

```
   assign leds_out = count[31:24]
```

If we now run this we will see that the leds will now count slowly. Note you can change the registers to create a slower or faster clock if you wish.

However with this we have to command over what happens, with no control to stop the clock. We can reintroduce **KEY[01]** to stop and then restart the counting. This is done by adding a new register called **go** and **if** statements to declare when and when not to run. Firstly we need to add a register.

```
   reg go;
```

As you can see we don't need to specify the size of our go register as is only being used as 0 and 1. The default size of 1 bit is sufficient.

Then we introduce the KEY[01] and by pressing once it will turn go on, and then again it will turn the count of.

```
always @(posedge KEY[01]) begin
   if (go == 1)
      go <= 0;
   else
      go <= go + 1;
end
```

We then add **if** statements to the posedge of the clock to stop the count once **go** is 0.

```
always @(posedge CLOCK_50) begin
   if (go == 1)
      count <= count + 1;
   if (go == 0)
       count <= 0;
end
```

Now once compiled and run you should have an automatic counter at your control, *insert evil laugh here*.

5.3. Fibonacci Counter

The Fibonacci sequence is a sequence in which the subsequent number is the sum of the previous for example, the start would be 0 + 1 = 1 and then 1 + 1 = 2 and then 1 + 2 = 3 and so on. This is achieved similarly to the counters project, however as we need to remember the previous count, we therefore need to start with 2 registers.

```
reg [07:00] pcount, count; // pcount represents previous count
```

Note

You can create more than one register at a time by placing in commas, we will then use the wires and assigns used for the normal count .

```
wire [07:00] leds;
assign LED[07:00] = leds;
assign leds = count;
assign reset = ~KEY[0];
assign next = ~KEY[1];
```

Now we have to write the code to create a Fibonacci sequence.

```
    always @(posedge next or posedge reset ) begin
       if (reset == 1'b1)begin
          count <= 0;
       end
       else begin                // else would be equal to if on next
          count <= count + pcount;
          pcount <= count;
    if (count == 0) begin  // need to change otherwise the
                             // answer will always be 0
              count <= 1;     // this applies to both the reset and starting
          end
       end
    end
```

In this example, an **always** block which contains 2 possibilities is used, one for each KEYs begin pressed down individually this is so that the variables can be controlled with both keys. First I created a reset and then I made **count** 1, as if it were zero it could not then create the sequence. (Technically it works, but is a little boring). So you need to make sure you can set the initial values, and then you create the sequence: **count** becomes previous **count** plus **count**, and previous **count** will become count. (Am I the only one who has an urge to watch a vampire movie?)

Chapter 6. Our Lock

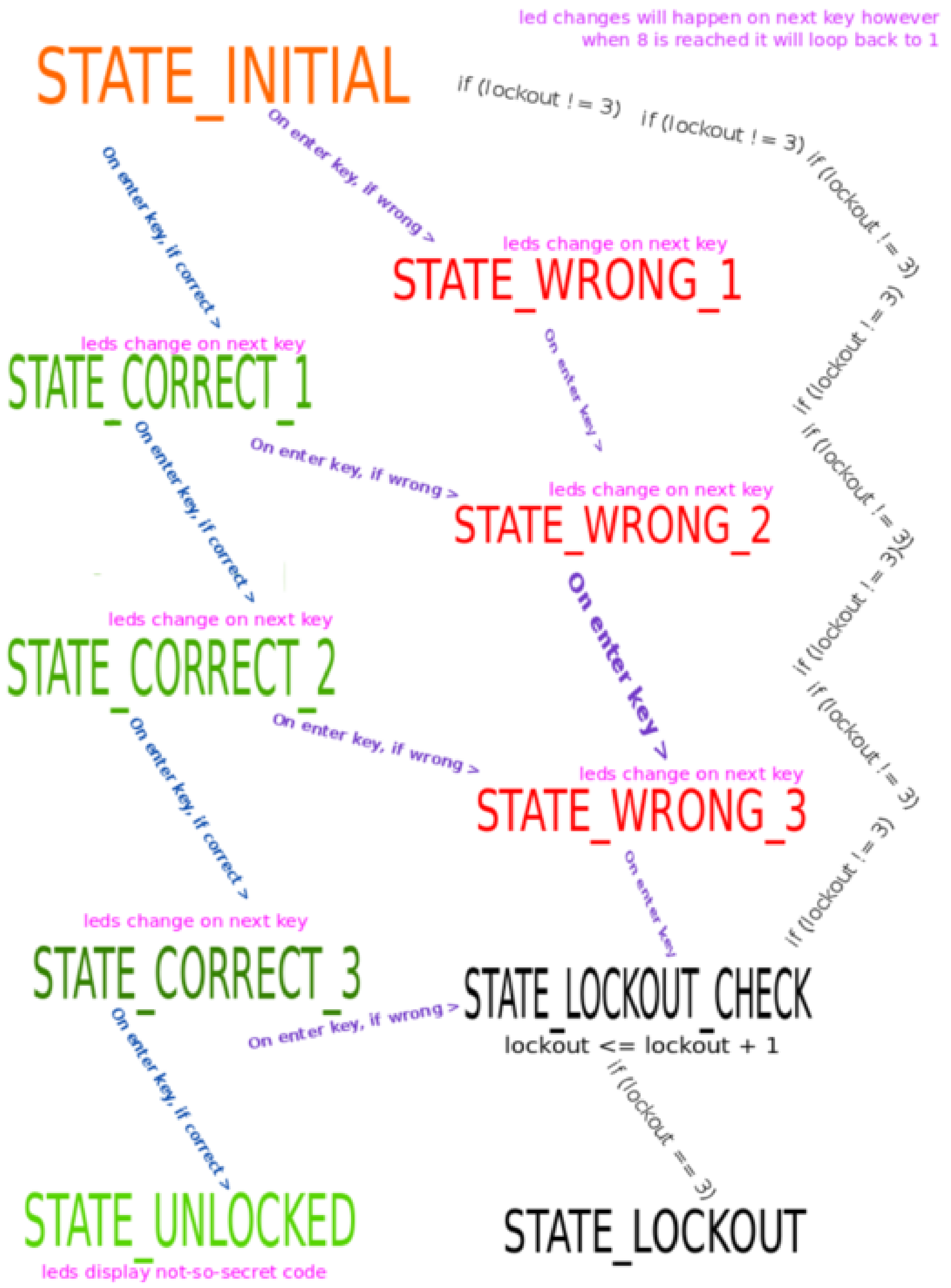
Out next project is a lock to keep our secret number safe behind a 4 digit code. However we will aproach this by a different method; we will work out how to implement this using a state machine.

6.1. What is a state machine?

A state machine is a machine which represents a number of different states. These help us create solutions to complex questions by representing it as a set of states where each does one or two simple functions. The next state is chosen based on the inputs to the state machines and the outputs are set depending on the current state.

6.2. Our first State machine

Our first step will be to draw out our state machine so we are completely clear how it will work, making sure to name our states in a meaningful way.



Next we define these names and associate these with a number. We can use the names in place of the number when writing the code that drives out state machine. However also for this project we will need to create another slow clock, so we need to define some other stuff. We will be making our 4 digit lock by having the LEDs light up a single light to represent such number, one key will be used to move this one place to right and the other will be used to enter the digit, if all 4 digits are correct we will then have it display our top secret message, or if incorrect have it show nothing at all.

```
`define STATE_INITIAL 10'd0
`define STATE_CORRECT_1 10'd1
`define STATE_CORRECT_2 10'd2
`define STATE_CORRECT_3 10'd3
`define STATE_WRONG_1 10'd4
`define STATE_WRONG_2 10'd5
`define STATE_WRONG_3 10'd6
`define STATE_UNLOCKED 10'd7
`define STATE_LOCKOUT_CHECK 10'd8
`define STATE_LOCKOUT 10'd9
`ifdef SIMULATE
`define COUNTER_SIZE 8 // 32 bits // for later, when we use GTKWave
`define SLOW_CLK_BIT 2 // 16th bit // as will not need to be visible
`else
`define COUNTER_SIZE 32 // 32 bits // but we kind of what that now
`define SLOW_CLK_BIT 20 // 16th bit
`endif
```

We have defined these with names as is easier for us to remember, as otherwise we would have to remember that **4'd7** is equal to unlocked and other general nastiness of the sort. Next we need to define our inputs and outputs.

```
module DE0_NANO (CLOCK_50,KEY,LED);
   input CLOCK_50;
   input [01:00] KEY;
   output [07:00] LED;
```

Now we can work on the slow clock, which is needed so that we can see what is running, we will start by using somebody else's open source code.

As you can see during the code we use a **dummy_reset**. This is because the code we started with used a reset based on a key. As we are using both keys already and do not need a reset key, we therefore need something to replace the reset in the code that doesn't affect anything.

```
`ifdef SIMULATE
 `define COUNTER_SIZE 8
 `define SLOW_CLK_BIT 2
`else
 `define COUNTER_SIZE 32
 `define SLOW_CLK_BIT 20
`endif

   reg  [`COUNTER_SIZE-1:00] clkcount;
   assign slow_clock = clkcount[`SLOW_CLK_BIT-1]; // you can see with '`'
   reg dummy_reset;                               // we refer to defines
   initial dummy_reset = 0;

   edge_detect ed_0 (.CLK( slow_clock),
                     .RST (dummy_reset),
                     .IN (next),
                     .OUT (next_ed));
   edge_detect ed_1 (.CLK (slow_clock),
                     .RST (dummy_reset),
                     .IN (enter),
                     .OUT (enter_ed));

   always @(posedge CLOCK_50) begin
      if (dummy_reset == 1'b1) begin
         clkcount <= 0;
      end
      else begin
         clkcount <= clkcount + 1;
      end
   end
endmodule
```

```
module edge_detect(
  input  CLK,
  input  RST,
  input  IN,
  output OUT );

  reg a, b;
  // the edge detect signal is (b AND (NOT a))
  assign OUT = a & !b;

  always @(posedge CLK) begin
    // it's always good to have a reset condition, otherwise
    // the state of the register will show up as undertemined
    // in simulation ('x')
    if (RST == 1'b1) begin
      a <= 0;
      b <= 0;
         end
         else begin
      a <= IN;
      b <= a;
         end
  end
endmodule
```

Now we can create all the registers and wires needed for our states. Additionally, we will need our normal LED ones, a count, as well as **lockout**, **nextlockout**, **state** and **nextstate**. We will be using the edge detect on our keys so that we get a proper result, as keys/buttons don't work how you would expect them to; instead of nicely going up once, they often spike, and then reach the normal level, therefore sometimes creating 2 pulses.

```
    reg [07:00] ledscount;
    reg [03:00] lockout,nextlockout;
    reg [02:00] count;
    reg [15:00] state,nextstate;

    wire next, enter;
    wire next_ed, enter_ed;   // 'ed' stands for edge_detect

    assign LED[07:00] = ledscount[07:00];  // which LEDs will be lit up
    assign next = ~KEY[00] ;               // to scroll through the leds
    assign enter = ~KEY[01] ;              // to select which value to enter

    initial lockout = 0;
    initial state = `STATE_INITIAL;
    initial nextstate = `STATE_INITIAL;
    initial count = 0;
```

Now we move onto the code that manages states, what the states do but also which they move to under what conditions. First we start with our initial state, it will need to have the positive edge of the next key move the leds along 1 slot and to know which key is correct, so it can make the next state **STATE_CORRECT_1** or **STATE_WRONG_1** depending on if the value is correct.

```
    always @(posedge slow_clock) begin
       if (state = `STATE_INITIAL) begin
          ledscount <= 3'd1 << count;
          if (next_ed) begin
             count <= count + 3'd1;
          end
          else if (enter_ed) begin
             if (count == 3'd3)
                nextstate <= `STATE_CORRECT_1
             else if (count != 3'd3)
                nextstate <= `STATE_WRONG_1
          end
       end
```

As you can see in this example, I have used **3'd3**. This represents (in reverse order) a number 3, in decimal, 3 bits long. In general this is **<size>'<type><number>**. Therefore you can see the first digit of this code is the number 3. This is then repeated for the next two states, replacing the nextstates. The following represents **STATE_CORRECT_3**.

```
    if (state == `STATE_CORRECT_3) begin
       ledscount <= 3'd1 << count;
       if (next_ed) begin
          count <= count + 3'd1 ;
       end
       else if (enter_ed) begin
          if (count == 3'd7)
             nextstate <= `STATE_UNLOCKED;
          else if (count != 3'd7)
             nextstate <= `STATE_LOCKOUT_CHECK;
       end
    end
```

You can see in this one that if the value is correct, instead of moving to **STATE_CORRECT_4**, it has gone to **STATE_UNLOCKED**. However if you get it wrong it doesn't go straight back to the initial state, but instead goes to a lockout check. This checks if lockout is 2 and if not adds 1 to lockout. Additionally, if lockout is 2 then we move to **STATE_LOCKOUT**, otherwise it is sent to **STATE_INITIAL** so that another attempt to enter the pass code can be made (my pass code is not 1337).

We now need to make our **STATE_WRONG** states. These are purely in place to prevent people being able to guess the code, so needs to have have no notable difference to the others, so needs the LEDs to be able to move from left to right and to move forward one, and once enter is pressed move to the next state until 4 digits have been entered. In order to make it harder to guess the 4 digit code, it will only ever link to the next **STATE_WRONG** state or to **STATE_UNLOCKED_CHECK**. You will need to make your own versions or **STATE_WRONG_2** and 3 linking to the appropriate places, such as the following.

```
    if (state == `STATE_WRONG_1) begin
       ledscount <= 3'd1 << count;
       if (next_ed) begin
          count <= count + 3'd1 ;
       end
       if (enter_ed) begin         // go to `STATE_WRONG_2
          nextstate <= `STATE_WRONG_2;
       end
    end
```

We now need to write the lockout and unlocked states, first starting with **`STATE_LOCKOUT_CHECK`** as this will be the most difficult. In this state, you need to check that lockout is not 2 (this gives the unlocker 3 opportunities as on his/her first go round he/she will have 0 lockouts, on his/her second 1, and on his/her third he/she will have 2, and if he/she gets to the end he/she will then will be locked out). If lockouts is set, the next state is set to lockout, if it is not, we make nextlockout equal to lockout plus 1, and then transition to the initial state.

```
    // How many times did we get it wrong?
    if (state == `STATE_LOCKOUT_CHECK) begin
       count <= 0;                        // will reset number
       if (lockout == 4'd2) begin
          nextstate <= `STATE_LOCKOUT;     // has already had 3
                                             // incorrect attempts
       end
       else begin
          nextstate <= `STATE_INITIAL;     // has had 2 or less
          nextlockout <= lockout + 4'd1;   // incorrect attempts

       end
    end
    // wrong more than 3 times
    if (state == `STATE_LOCKOUT) begin
       ledscount <= 255;
    end
    // Unlocked because 4 digits were entered correctly
    if (state == `STATE_UNLOCKED) begin
       ledscount <= <mystery code>;
    end
```

Remember we need to make sure we state that on every clock cycle **`nextstate`** becomes **`state`** and **`nextlockout`** becomes **`lockout`**. We then finish with ending the module.

```
    state <= nextstate;
    lockout <= nextlockout;

 endmodule
```

It should now work, making sure you have correctly set your code, you can now keep a number up to 255 vaguely secure from another bunch of teenagers, hurrah!

Chapter 7. UART

7.1. What is a UART?

A Universal Asynchronous Receiver/transmitter (*UART*) is a simple bit based method of transferring data between machines. A method of communicating via serial communication to a computer, or to a peripheral.

7.2. Hello? world?

Our first *UART* project shall be to transmit "Hello, world!" to the monitor, so first we need to create the *UART* transmitter on the FPGA. We create a register to hold the state of our transmit state, one for the data we want to transmit and one for the word state.

```
reg [3:0] transmit_state;  // will be used like a state machine
reg [13:00] word_state;    // will be used to determine where
                           // in the sentence we are
reg [07:00] transmit_data;
```

We will also be transmitting at 115200 baud so we will need to use a clock divider register to create a clock that runs at the right speed.

```
reg [09:00] clock_divider_counter
reg uart_clock;
```

For the clock divider, we divide the clock by 217, as this roughly goes into 50,000,000 230400(2 x 115200) times (this is because we will need a posedge for our UART).

```
always @(posedge CLOCK_50) begin
   if (reset == 1'b1)                     // reset if reset button hit
      clock_divider_counter <= 0;
   else if (clock_divider_counter == 217)  // reset if too high
      clock_divider_counter <= 0;
   else
      clock_divider_counter <= clock_divider_counter + 1;
end


always @(posedge CLOCK_50) begin
   if (reset == 1'b1)
      uart_clock <= 0;
   else if(clock_divider_counter == 217)
      uart_clock <= ~uart_clock;
end
```

We then create the code that creates our message. This will be implemented as a state machine, but this time a **`case`** statement will be used to implement the state machine.

```
   always @(posedge uart_clock or posedge reset) begin
      if (reset) begin        // Reset to the "IDLE" state
         transmit_state <= 0;
         word_state <= 1;
         UART_TX <= 1;     // The UART line is set to '1'
                              // when idle, or reset
      end
```

Firstly we create the reset condition. If we are in reset it will then set **transmit_state**, **word_state** and **UART_TX** and not do anything else in the always block. Once we are out of reset we can process the main logic as follows:

```
      else begin
         // What follows is the skeleton of the state machine to control
        // the bits going onto the UART transmit line.
        // You will want to, from the idle state:
        // 1. detect the pushbutton press and go to the the start bit state
        // 2. then the 8 data bits (LSB first)
        // 3. finally the stop bit
        // 4. return to this state ready for the next transmit
         case (transmit_state)
         0:
            begin
            if (key1_edge_detect == 1)
               transmit_state <= 1;
            end
         1:
            begin
               UART_TX <= 0;
               transmit_state <=2;
               // Start bit state, and progress onto the next state
        end
         2,3,4,5,6,7,8,9:
            begin
               UART_TX <= transmit_data[transmit_state - 2];
               transmit_state <= transmit_state + 1;
               // Data bits
               // when transmit_state is 2 we want transmit_data[0]
               // when transmit_state is 3 we want transmit_data[1]
               // ...
        // when transmit_state is 9 we want transmit_data[7]
        /* Fill me - assign appropriate data bit to UART_TX here
        - don't forget to continue incriminating the
        state
        */
            end
```

We then make sure that our word state stays valid by checking if it is 14, and if not making sure to add 1 to the **word_state**.

```
        10:
           begin
              UART_TX <= 1;
              transmit_state <= 0;
              if (word_state == 14) begin
                 word_state <= 1;
           end
           else
              word_state <= word_state + 1;
```

To send our message, we look at **word_state**. If it equals for example 1, **transmit_data** will become 'H' or the next character in our message. Characters are sent using a hexadecimal representation, therefore you will need an ASCII table [5] to look up which codes you want. I wonder how many pop culture references you can display solely using this FPGA.

Note

This is still inside the always block.

```
      begin
        if (word_state == 1)
           transmit_data <= 8'h48; //H
        if (word_state == 2)
           transmit_data <= 8'h65; //e
        if (word_state == 3)
           transmit_data <= 8'h6c; //l
        if (word_state == 4)
           transmit_data <= 8'h6c; //l
        if (word_state == 5)
           transmit_data <= 8'h6f; //o
        if (word_state == 6)
           transmit_data <= 8'h2c; //,
        if (word_state == 7)
           transmit_data <= 8'h20; //
        if (word_state == 8)
           transmit_data <= 8'h57; //W
        if (word_state == 9)
           transmit_data <= 8'h6f; //o
        if (word_state == 10)
           transmit_data <= 8'h72; //r
        if (word_state == 11)
           transmit_data <= 8'h6c; //l
        if (word_state == 12)
           transmit_data <= 8'h64; //d
        if (word_state == 13)
           transmit_data <= 8'h21; //!
        if (word_state == 14)
           transmit_data <= 8'h20; //
     end
```

Next we clean up, and finish the code, by introducing a default condition it shouldn't reach, but if it does it will go back to the idle state, also setting the LEDs and the edge_detect state.

```
      end
         default:

            transmit_state <= 0;
         endcase
      end
   end

   always @(posedge uart_clock)
      key1_reg <= KEY[1];

   assign key1_edge_detect = ~KEY[1] & key1_reg;
                                 // Detect the change in level
   assign LED = transmit_data;   // or change to what you
```

```
   endmodule
```

Always remember to finish modules with an **`endmodule`**.

7.3. Getting it to run on the screen

You will need to use one of Embecosm's USB *UART*s and have it plugged in correctly to the board but also into the computer you are want to display the message on. See Figure 7.1.

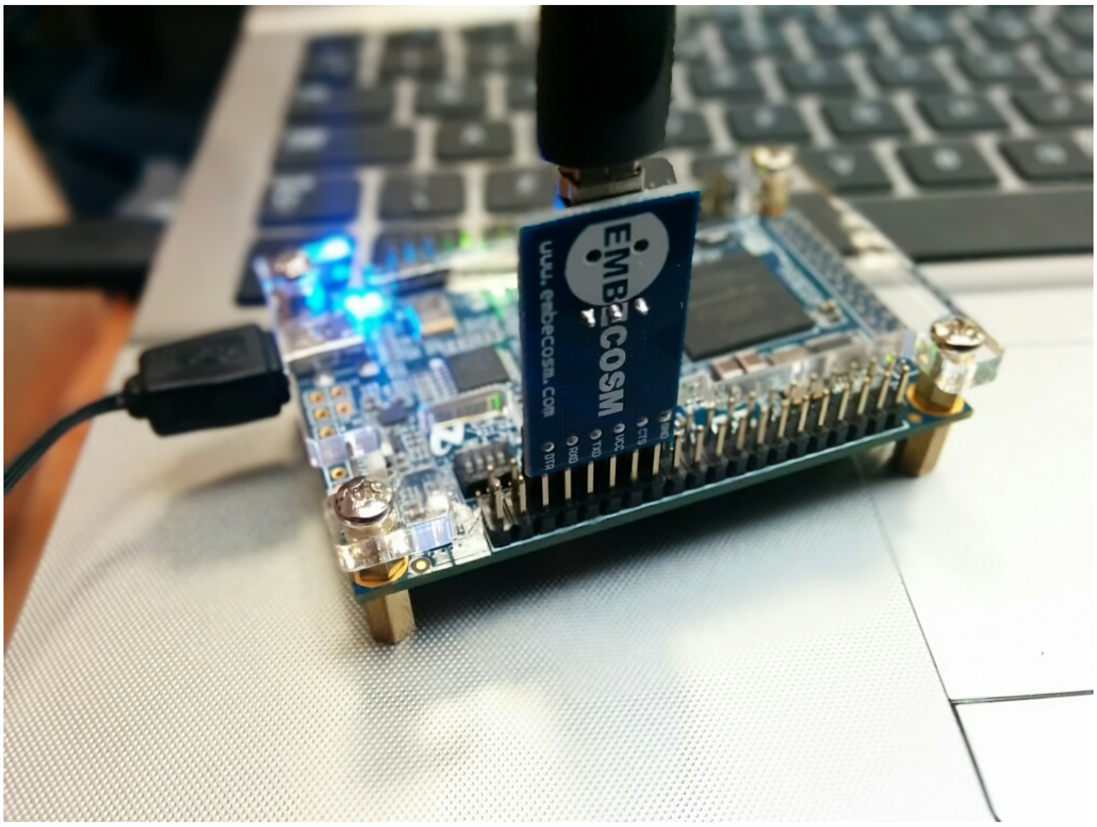


Figure 7.1. USB to *UART* connector.

On Windows, to view the *UART* you will need to get a terminal — we suggest PuTTY. Once loaded you will need to change the connection type to serial, change the serial line to the address of the device (for example COM3, this may differ per machine, you may need to look it up under devices), and the speed to 115200.

For Linux you will need to be in the **`dialout`** group and then you can use the command below in the terminal to display your message.

```
$ screen /dev/ttyUSB0 115200
```

Chapter 8. OpenRISC and SoC

Caution
This is more advanced work, and you are advised to use Linux. There will be no Windows support, so consider a using a virtual machine.

8.1. What is OpenRISC?

No it is not the open source remake of the Parker Brothers board game, instead OpenRISC stands for open reduced instruction set computer and is a project which has created a computer architecture and implementation and tools for its development. It is a design specification for an open source processor.

The OpenRISC 1000 architecture has a 32-bit instruction word and either 32-bit or 64-bit data. As it is reduced instruction set, its instructions are relatively simple like:

```
add register 3 with register 6 and store in register 8
```

or

```
load the data at the memory address held in register 4 into register 5
```

Whereas a more complex instruction set computer (CISC) may be capable of doing much more in a single instruction:

```
     load the data at the memory address in register 2, increment it, compare
     with zero, and store back at the address held in register 4 while
     incriminating both registers 2 and 3
```

8.2. What is a SoC?

A System on (a) Chip. When we bring this together with our own synthesisable models of peripheral controllers, communications I/O and system infrastructure we have a system capable of many things. Typically the brains of the system is the programmable CPU.

8.3. Installation

Warning
This will take a long time.

To install you will need:

1. The OpenRISC GNU tool chain (bare metal, newlib-basic, or1k-elf-)
2. The FuseSoC devlopment environment
3. Icarus Verilog and *GTKWave*

4. The Altera Quartus tools (for synthesis, board programming)[4]

5. The OpenOCD debug proxy

8.3.1. General System Tools

These will be necessary for various parts of the flow. On Debian or Ubuntu systems you can install them with:

```
sudo apt-get -y install build-essential make gcc g++ flex bison    \
patch texinfo libncurses5-dev libmpfr-dev libgmp3-dev libmpc-dev   \
libzip-dev python-dev libexpat1-dev libftdi-dev libtool autoconf   \
libftdi-dev subversion libelf-dev elfutils
```

8.3.2. OpenRISC GNU tool chain precompiled for 32-bit linux

You will need to find the or1k-elf toolchain online and extract it in the **/opt** directory, creating the directory **or1k-toolchain**. These tools will need to be in your PATH in order to use, them, so the following needs to be run to enable this by default.

```
echo "# OpenRISC tool chain path" >> ~/.bashrc
echo "export PATH=\$PATH:/opt/or1k-toolchain/bin" >> ~/.bashrc
```

Note
If this does not work, consult the Chiphack Wiki.

8.3.3. Quartus Tools

You should already have these installed, however if you have not visit their website and download and install *Altera Quartus II Web Edition*.

These can also be added to your **PATH** using the following, noting to change the version number to the one you have installed:

```
echo "# Altera Quartus tools path" >> ~/.bashrc
echo "export ALTERA_PATH=/opt/altera/13.1" >> ~/.bashrc
echo "export PATH=\$PATH:\$ALTERA_PATH/quartus/bin" >> ~/.bashrc
```

8.3.4. Icarus Verilog and GTKWave

These are both open source projects, and can be easy installed from any modern Linux distribution. Otherwise you can follow an install guide from the Icarus Verilog wiki.

```
sudo apt-get install iVerilog gtkwave
```

8.3.5. OpenOCD

OpenOCD is the debug proxy we'll use to talk to the board over *JTAG*.

Download the source to **$HOME/or1k**

```
      git clone https://github.com/openrisc/openOCD.git
```

Go into the **OpenOCD** directory and, bootstrap it:

```
./bootstrap
```

You may need to install *libtool* and *autoconf* via your package manager to run the bootstrap process.

Once that is finished, configure and compile:

```
./configure --enable-usb_blaster_libftdI --enable-adv_debug_sys \\
 --enable-altera_vjtag --enable-maintainer-mode
make
make install
```

Note
I suggest downloading and playing a game of *greed* in the wait (it takes along time).

```
sudo apt-get install greed
greed
```

8.3.6. FuseSoC

The SoC development tool is now needed.

Clone this from GitHub into **$HOME/or1k**

```
git clone https://github.com/olofk/fusesoc.git
```

Now go into **fusesoc** and run the following:

```
autoreconf -i
./configure && make
make install
```

8.3.7. orpsoc-cores

The OpenRISC set of configurations for *FuseSoC* to work with, needs to be downloaded next.

Clone this from GitHub into **$HOME/or1k**

```
git clone https://github.com/openrisc/orpsoc-cores.git
```

Hurrah, we did it! No more installs!

8.4. Waves

8.4.1. Hello? Again?

Now we will be writing something to run on the OpenRISC and for us to be able to debug in *GTKWave*. Unlike software programming it is very hard to debug hardware, and viewing the **.vcd** is as close to a debugger as you can get. This allows us to view the values of every register and wire so therefore lets us understand what has occured. We start by writing a simple program in a file called **hello.c**:

```
int main(void)
{
    printf("Hello world, from an OpenRISC system!\n");
    return 0;
}
```

We then compile it using the OpenRISC toolchain.

```
or1k-elf-gcc hello.c -o hello.elf
```

Then we can run it on a simulator using **fusesoc**.

```
fusesoc sim mor1kx-generic --elf-load hello.elf
```

Note
If it doesn't run correctly make sure you have all the tools installed.

8.4.2. Waveform

We can now inspect the program we have just run in *GTKWave*, which we installed earlier. First we run it to produce a **.vcd**, which is a file that *GTKWave* can open.

```
fusesoc sim mor1kx-generic --elf-load hello.elf --vcd
```

This can then be opened with *GTKWave*

```
gtkwave build/mor1kx-generic/sim-icarus/testlog.vcd
```

The program should now load up. In the hierarchy browser (top left corner) expand **orpsoc_tb** then **dut**. Then highlight mor1kx (the processor), this will now list the signals in the window below. Select all signals beginning with **iwbm** and insert, then do the same for **dwbm**. To be able to see properly, zoom in so you can see 100s of nanoseconds, similar to Figure 8.1:

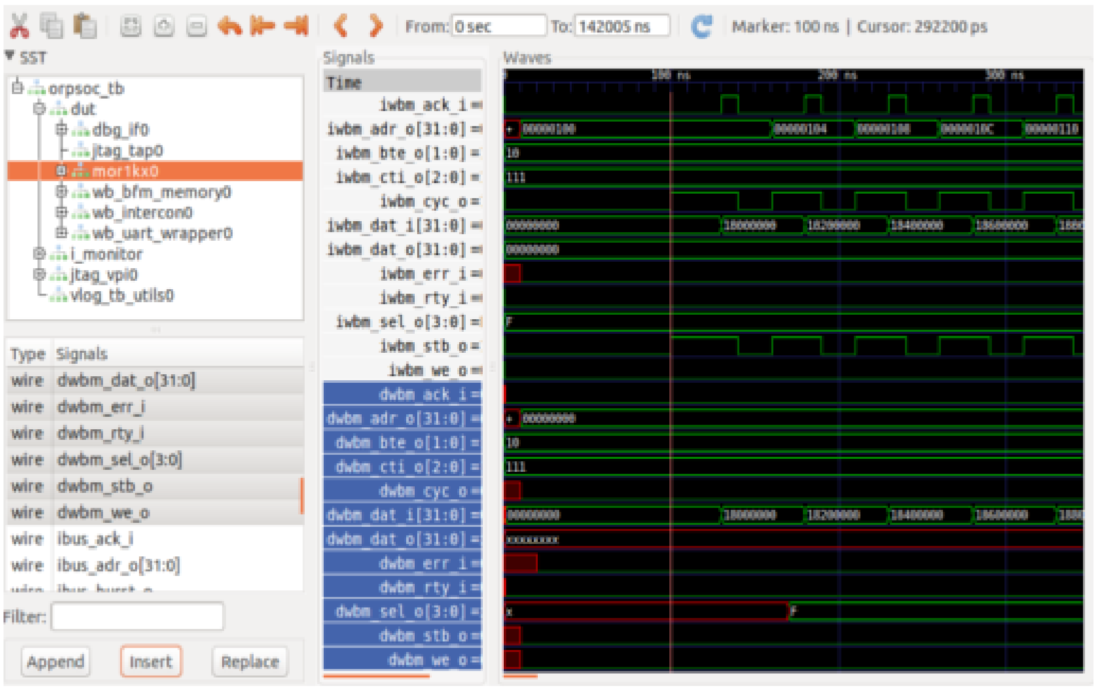


Figure 8.1. Example of a wave trace

8.5. Running on the DE0 Nano

Now we will be building and then running the system on our FPGA.

8.5.1. USB to UART patch

We will modify the source for the *DE0 Nano* system in **`orpsoc-cores`** to support the use of the Embecosm USB to *UART* board. Firstly download and apply the following patch:

```
wget http://goo.gl/xw74Aa
git am xw74Aa
```

The system's source is now suitable to work with the Embecosm USB to *UART*.

8.5.2. Programming the board

We will now build the image to be programmed onto the *DE0 Nano*. Note this will take a while. From the **`or1k`** directory.

```
fusesoc build de0_nano
```

In the synthesis directory there is also a makefile recipe for programming the board:

```
fusesoc pgm de0_nano
```

Note

There are several things could go wrong here. The first is that the system JTAG daemon running needs to be killed and the Altera version run instead, to do this run:

```
killall jtagd
sudo /opt/altera/13.1/quartus/bin/jtagd
```

Another problem might be that the OpenOCD debugger is still using the JTAG/USB port. Exiting OpenOCD will fix this.

Another could be basic permissions on the USB device and this may fix things:

```
sudo make pgm
```

8.5.3. Connecting the debug proxy

From the OpenOCD directory run the following:

```
sudo ./build/src/openocd -f ./tcl/interface/altera-usb-blaster.cfg \\
    -f altera-dev.tcl
```

8.5.4. Rebuilding our program

We are going to take the c code we wrote earlier and now recompile it to run on the *DE0 Nano*

```
or1k-elf-gcc hello.c -o hello_de0_nano.elf -mboard=de0_nano
```

8.5.5. Openning a terminal

Next open a terminal, like we did for the *UART* using the following:

```
screen /dev/ttyUSB0 115200
```

8.5.6. Connecting the debugger

We will be using a debugger, solely for running a program on the machine however these can be used to stop a program executing at any time and evaluate the state of the machine. For example we can look at the value of any variable in our code.

In a new terminal run the OpenRISC GNU Debugger (GDB) and specify the executable we want to run:

```
or1k-elf-gdb hello_de0_nano.elf
```

Within GDB we tell it to connect to the port that OpenOCD is running on:

```
(gdb) target remote :50001
```

We can now access the system memory and registers, for example to look at the memory at address zero, we use:

```
x 0x0
```

Now, to run the program, first we begin by:

```
(gdb) load
```

and then:

```
(gdb) continue
```

Hurrah, it works!

8.5.7. Running Linux on our FPGA

Congratulations on getting this far, but you have not finished yet… Now, instead of running "hello world", we will run Linux **dramatic music**. but first we need to download it:

```
wget https://www.dropbox.com/s/bi5vx8kmqnjdldx/vmlinux-de0_nano
```

Now load it in the GDB as before, though this time with a couple of changes:

```
(gdb) file vmlinux_de0_nano
(gdb) load
(gdb) spr npc 0x100
(gdb) c
```

Look! it's moving. It's alive. It's alive…

However, unfortunately it is a embedded version of Linux, little is doable on it therefore it is solely a bragging right, for now…

Glossary

Binary	A way of displaying numbers only using '1's and '0's.
DE0 Nano	The model of FPGA i have been using.
FPGA	Field Programmable Gate Array, are able to be changed using a HDL such as Verilog.
HDL	HDL stands for hardware description language, for example Verilog is a language which is used to describe the FPGA
OpenRISC	A set of opensource design specifications for a processor, we shall be using an implementation of it.
UART	A *UART* (Universal Asynchronous Receiver/Transmitter), used to communicate between chips (not potatoes).

References

[1] Chiphack Repository Available at http://chiphack.org.

[2] Basic Logic Available at http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/index.html#orgate.

[3] Chiphack Wiki Available at https://github.com/embecosm/chiphack/wiki.

[4] Quartus Available at http://www.altera.co.uk/products/software/quartus-ii/web-edition/qts-we-index.html.

[5] ASCII Table Available at http://www.asciitable.com/.

[6] DE0_NANO Available at https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593/.

[7] PuTTY Available at http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[8] Icarus Verilog alternate install method Available at http://iVerilog.wikia.com/wiki/Installation_Guide.

[9] CP210x USB to *UART* Bridge Available at http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx.

[10] Altera USB Blaster driver for linux Available at http://www.altera.co.uk/download/drivers/dri-usb_b-lnx.html.

[11] Sabre Dance Available at http://www.youtube.com/watch?v=gqg3l3r_DRI.

[12] Altera USB Blaster driver for Windows Available at http://www.altera.co.uk/download/drivers/usb-blaster/dri-usb-blaster-vista.html.