Home
The Go Board
FPGA 101
VHDL
Verilog
YouTube Channel
GitHub
Patreon *NEW*

What every software programmer needs to understand about hardware design

The most important article for a new digital designer

Every software developer who knows a language such as C or Java has the same problems when trying to start programming in VHDL or Verilog. They make assumptions about how code works. These assumptions are usually universal for all software languages. Unfortunately, those assumptions do not hold for hardware description languages. If you are new to hardware development but know a software language or two, read this first! This article gives examples of code and explains how the code works in a software world and in a hardware world to show you the difference.

Assumption #1: Serial vs. Parallel Logic

This is probably the most fundamental difference between hardware and software programming languages. Software designers have only ever seen serial code, yet they probably do not realize this fact. What is meant by serial code is that the lines of code are executed one at a time. For example, Line #2 can only execute after Line #1 is complete. VHDL and Verilog do not behave this way! They are known as parallel logic languages and all of the lines of code can and will execute all at the same time. This is known as concurrency. Here is an example demonstrating the difference between serial and parallel logic. Let's say that a designer wants to light up an LED once every ten clocks.

Example Software Code:

```
1  while (1)
2  {
3    if (count == 9)
4    {
5      LED_on = 1;
6      count = 0;
7    }
8    else
9    {
10     LED_on = 0;
11     count = count + 1;
12   }
13 }
```

Equivalent VHDL Code:

```
1  P_INCREMENT : process (clock)
2  begin
3    if rising_edge(clock) then
4      if (count < 9) then
5        count <= count + 1;
6      else
7        count <= 0;
8      end if;
9    end if;
10 end process P_INCREMENT;
11
12 LED_on <= '1' when count = 9 else '0';
```

The important thing to realize here is that in the software code, each line executes, then the next line is allowed to execute. This is NOT true in VHDL and Verilog, and this is demonstrated by the last line that assigns the LED_on signal. That line is running concurrently with the VHDL process. It is always assigning either a '1' or a '0' to LED_on. If this were software, this line would only be reached once the preceding lines of code had been executed. A good digital designer needs to always remember that VHDL and Verilog are parallel languages.

Assumption #2: For Loops

This is a huge problem that new hardware developers have. They have seen for loops hundreds of times in C, so they think that they are the same in Verilog and VHDL. Let me be clear here: For loops do NOT behave the same way in hardware as in software. Until you understand how for loops work, you should not use them.

Example Software Code:

```
1  For (int i=0; i<10; i++)
```

```
  data[i] = data[i] + 1;
```

This code will take every value in the array “data” and increment it by 1. Here is equivalent code in VHDL:

```
P_INCREMENT : process (clock)
begin
  if rising_edge(clock) then
    if (index < 10) then
      data[index] <= data[index] + 1;
      index <= index + 1;
    end if;
  end if;
end process P_INCREMENT;
```

index acts as counter signal

How else would you do a for loop in Verilog?

Starting to see a pattern here? Code written in C is almost always going to be less lines to do similar functionality as code in VHDL or Verilog. I’m going to be bold and say this: if you haven’t done at least 3 FPGA designs you should *never* be using for loops. So think about how the code you wrote in software could be rewritten to never use a for loop. Usually all you need is to add a counter signal (like index in the example above) to do the same thing that the for loop will do.

Assumption #3: Execution of Code is Immediate

This ties in to the serial vs. parallel logic discussion above but it is a common mistake that software developers often make. Software developers need to remember that a line of code is not immediately executed, with the value of the signal updated for use. An example below details this assumption with a state machine. The example below shows an initialization routine for some peripheral component, say an Analog to Digital converter. The FPGA needs to write in order to the ADC: 5, 6, 1, 0 on the data line. The state machine as described in software will allow the value of data to change as the states change. Unfortunately in the broken example of hardware code below this is not the case:

Example Software Code:

```
state = INITIALIZE;
data = 5;
state = LOAD_1;
data = 6;
state = LOAD_2;
data = 1;
state = DONE;
data = 0;
```

NOT Equivalent Hardware Code: (Broken Code)

```
P_STATE_MACHINE : process (clock)
begin
  if rising_edge(clock) then
    state <= INITIALIZE;
    data <= 5;
    state <= LOAD_1;
    data <= 6;
    state <= LOAD_2;
    data <= 1;
    state <= DONE;
    data <= 0;
  end if;
end process P_STATE_MACHINE;
```

Every line is concurrent.
Every line is like a goroutine

The hardware representation of this code is very broken! Since every line in a process executes concurrently, data will be always stuck at 0 and state will always be stuck at DONE. This code will never execute the state machine the way that it will in the software code. The code below is the correct way to write this state machine in VHDL:

Equivalent Hardware Code:

```
P_STATE_MACHINE : process (clock)
begin
  if rising_edge(clock) then
    if (state == INITIALIZE) then
      data <= 5;
      state <= LOAD_1;
    elsif (state == LOAD_1) then
      data <= 6;
      state <= LOAD_2;
    elsif (state == LOAD_2) then
      data <= 1;
      state <= DONE;
    elsif (state == DONE) then
      data = 0;
    else
```

value of `state` always is unknown

```
16          state <= INITIALIZE;
17        end if;
18      end if;
19    end process P_STATE_MACHINE;
```

The three examples above are three places that new software developers often have a hard time with when starting a new design in VHDL or Verilog. The three assumptions above should always be in the mind of a new digital designer. Questions about concurrency, for loops, and execution of code should always be considered. Once the examples above are thoroughly understood, the rest of the articles and examples on this web page can help turn you into a successful digital designer!

Help Me Make Great Content! **Support me on Patreon!** **Buy a Go Board!**

Reply • Share ›

Sivakrubakaran.s • 2 years ago

I need more explanation #3 program? I am beginner for this i don't know why
1) if "(state == INITIALiZE) " then 2) data <= 5;

Reply • Share ›

Sandip Vishwakarma ➚ Sivakrubakaran.s • 2 years ago

say for the first positive clock edge, all the code is executed at once and state is not equal to initialize (it is supposed that all the variables is containing some data like bit or integer)
say initialize=8, state=6, load_1=3, load_2=5, done=1.
then the last else case will saisfy and it will make state equal to initialize i.e,. state=8
now after 2nd positive clock edge, 1st if case satisfies, making data=5 and state=3 (load_1)
now after 3rd , 2nd case satisfies and data=6 and state=5...and goes on...
We require to appraoch in this way for hard typed language since in C, all instructions are processed one by one and old data is refreshed to new data easily but not in HDLs.

Reply • Share ›

P H • 2 years ago • edited

The message of this article is correct in that code is serial and hardware is in parallel however the example under Assumption #3 is not clear.
A statemachine is not needed for the software version but a output function is.
As it stands there is no transmission to the output to the ADC happening in the software version.
In the hardware the transmission is implict (assumed by the nature of an FPGA).

In the software version an extra line would be need:
for example Output-to-ADC(data) after each change of the data variable.

The state variable is meaningless in the software version (in this case) just because of the serial process characteristic of software code.

This would be a more realistic example highlighting the serial/ parallel differnce between the coding technics.

data = 5;
Output-to-ADC(data);

data = 6;

see more

Reply • Share ›

Sanity ➚ P H • 2 years ago

I guess you've never seen a register-mapped variable before? A variable is just a location in memory. If that location maps to a hardware port, writing to the variable writes to the hardware port. The AVR SDK, for instance, uses these for all the ports on the device. We can assume "data" maps to the ADC hardware port.

I think the state variable is there to show the transition to VHDL more clearly, otherwise it's not clear where the state labels come from. It's not needed in the C version at all, though, as you say.

1 Reply • Share ›