Abstraction of Verilog Descriptions

Submission Date: 02-05-2022

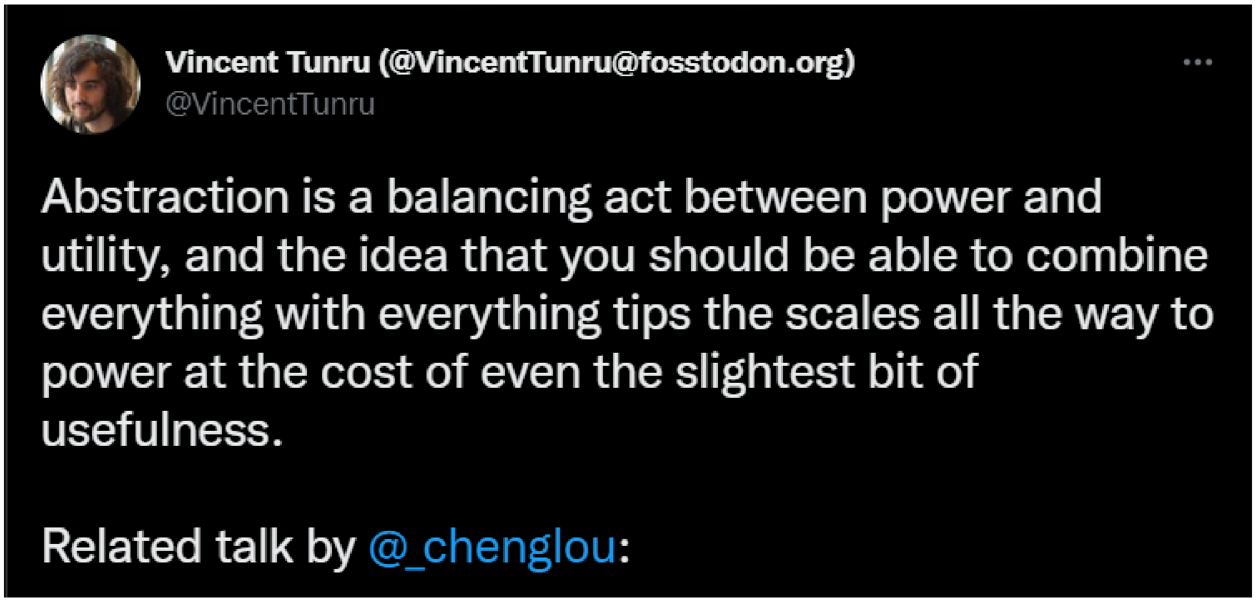
Link to Github repository

Submitted By: -
Nitin Singh Kanyal(B19EE058)
Nimit Khanna (B19EE057)

In complex systems, abstractions are critical. Modeling hardware systems involves abstracting away wide datapaths **but keeping low-level details of the underlying control logic in place.** Consequently, the state space is significantly reduced and intricate control interactions can be formalized. The abstraction process in these languages, however, must be done manually, an error-prone and tedious process. In this project, we have put forward an automating the Abstraction process for Verilog designs.

What is Abstraction?

The concept of abstraction refers to an arrangement of computer systems' complexity. An individual interacts with the system according to a certain level of complexity, while the more complex details are suppressed below that level.

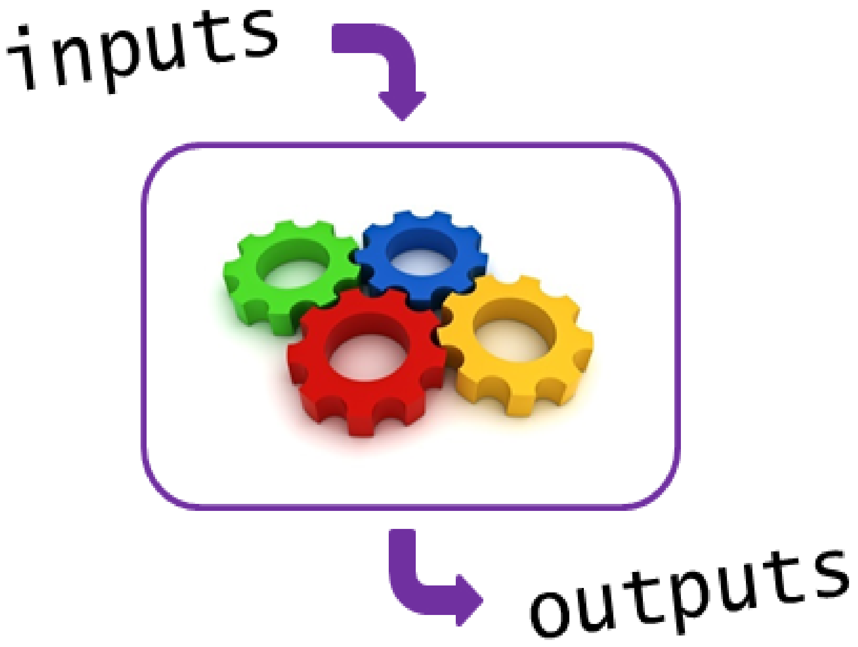

Vincent Tunru (@VincentTunru@fosstodon.org)
@VincentTunru

Abstraction is a balancing act between power and utility, and the idea that you should be able to combine everything with everything tips the scales all the way to power at the cost of even the slightest bit of usefulness.

Related talk by @_chenglou:

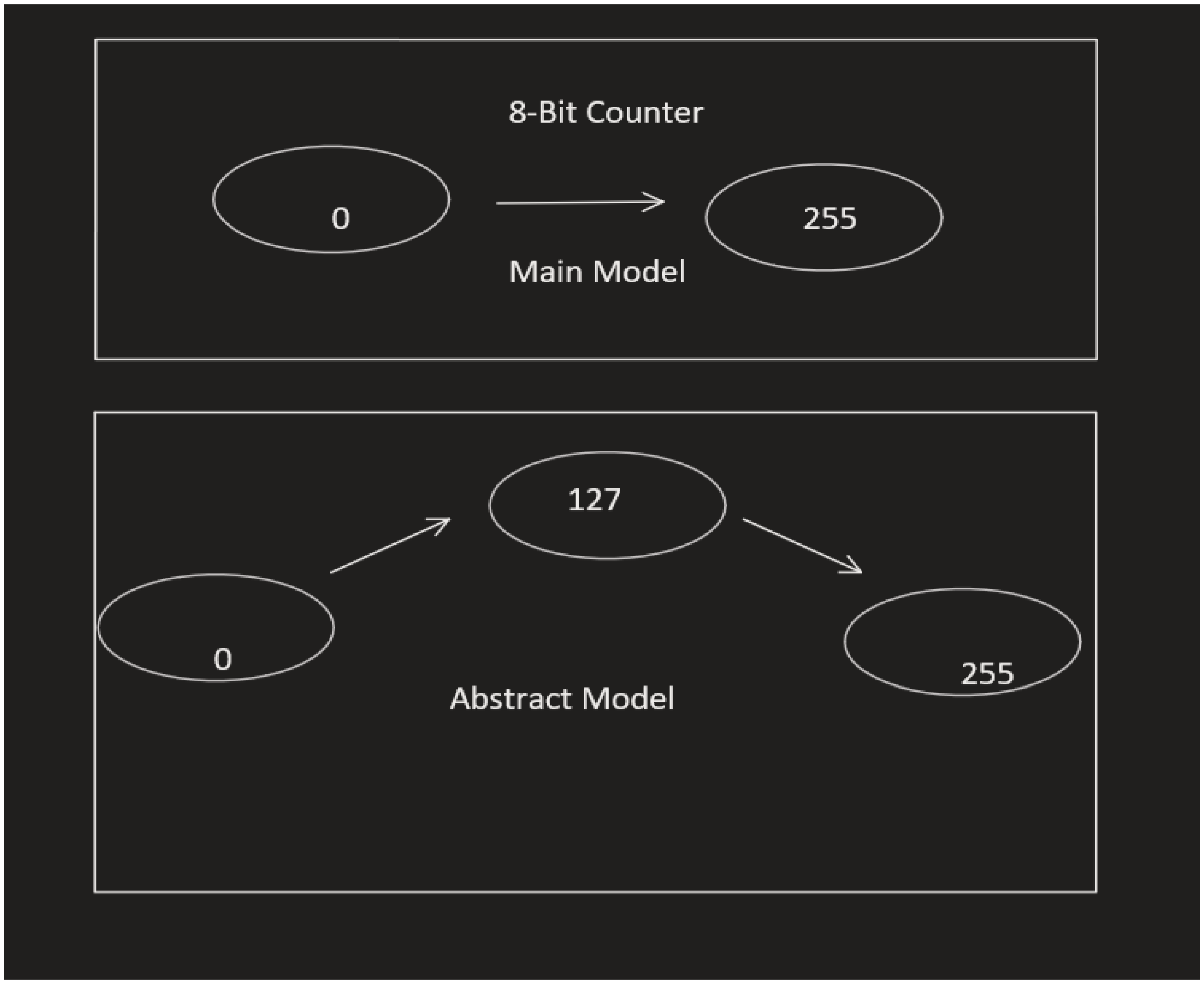
Why is Abstraction necessary?

Due to the complexity of life, abstraction is necessary. Complex systems must be simplified for people to understand and use them.



To understand this in a simple form. Let us take an example of **an 8-Bit Counter.**

8-Bit Counter will count from **0 to 255**. Let us assume that this design is a bit complicated. To make this design simpler we have to do abstraction. This can be done by introducing new states in between the counter or reducing the bit width. It is important to note that the new design must be simplified and the working of the model must be preserved. The new design/Abstract model will count from **0 to 127, and then 127 to 255**. This can be easily visualized in the following diagram:


8-Bit Counter
0
255
Main Model
127
0
255
Abstract Model

Some examples of manual abstraction

8 - Bit Counter -

Main Model -

```
module up_counter (

   out     , // Output of the counter
   clk     , // clock input
   data    , // Data to load
   reset     // reset input
);

output [7:0] out;
input [7:0] data;
input clk, reset;
reg [7:0] out;

always @(posedge clk)

if (reset)
   begin // active high reset
      out <= 8'b0 ;
   end
else
   begin
      out <= out + 1;
   end
endmodule
```

Manually abstracted model -

```
module up_counter (

   out     , // Output of the counter
   clk     , // clock input
   data    , // Data to load
   reset      // reset input
);

output [6:0] out;
input [6:0] data;
input clk, reset;
reg [6:0] out;

always @(posedge clk)

if (reset)
   begin // active high reset
      out <= 7'b0 ;
   end
else
   begin
      out <= out + 1;
   end
endmodule
```

32 - Bit Adder abstraction is available at the github link => [Github Link]

How to automate the abstraction of verilog models?

Reducing the Bit width

To automate the abstraction process we have to reduce the bit width on the basis of level of abstraction. For example in the model of 8 bit counter stated above, we have reduced the 8 bit registers to 7 bit registers. This means that the 0->255 counter will now be abstracted in the 0->127 counter.

Final code and input outputs

The final code along with the inputs and outputs on various verilog models are present in the following github repository -

Link to Github repository

What else could be done?

There are more possibilities of achieving the abstraction. Here are some of the ways by which we can achieve abstraction:

Abstraction of Verilog Global Variables and Constants

If there are any global variables/constants then they must be mapped to a hash table. So that when we have to reduce bit width, the values of variable could be mapped in hash table.

For example: Below is the code snippet for some module in which there are two global variables. The hash table would look like this :

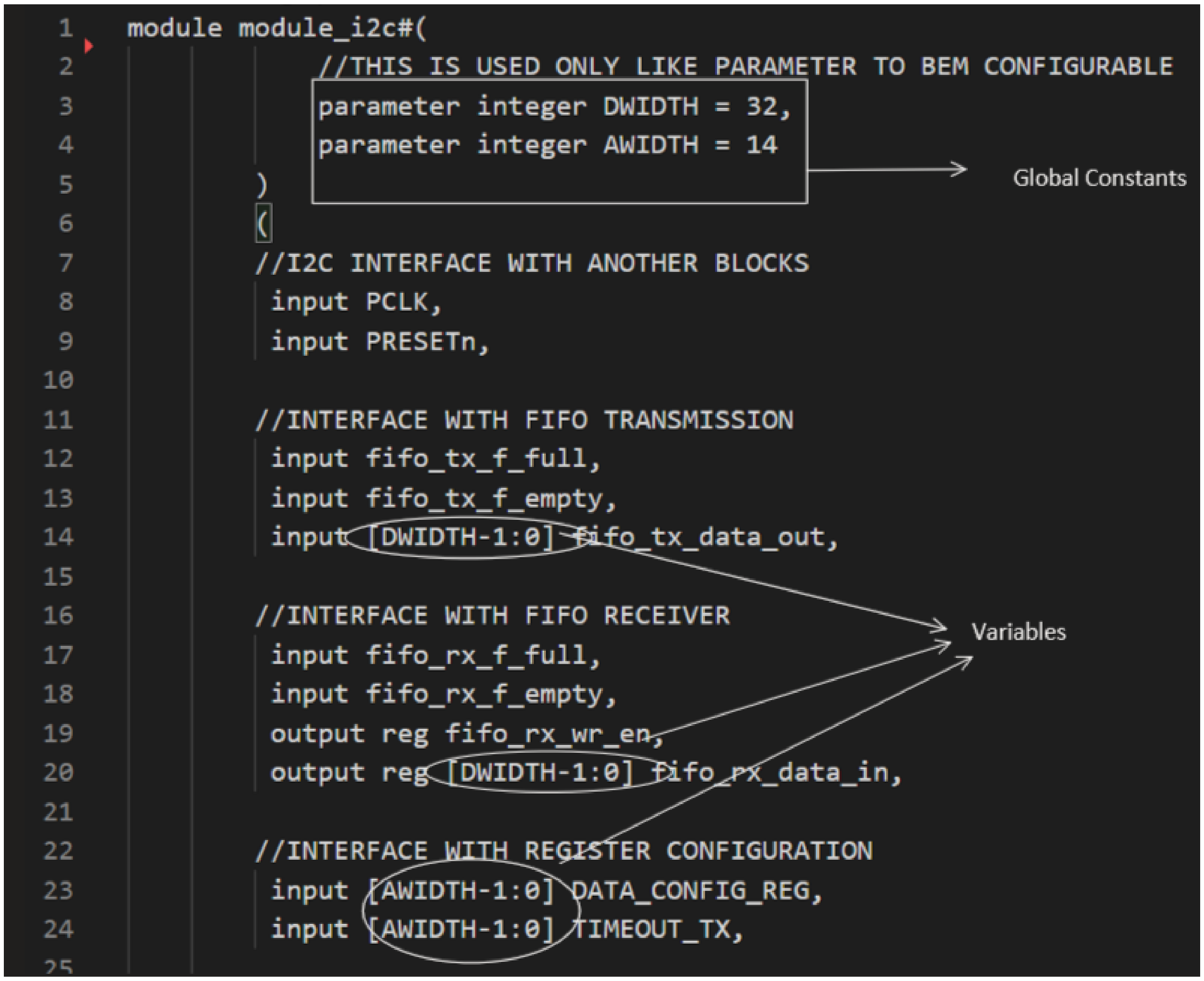
Hash Table

DWIDTH	32
AWIDTH	14

Now the Hash Table must be updated according to the level of Abstraction

DWIDTH	8
AWIDTH	4

Code snippet



```
module module_i2c#(
        //THIS IS USED ONLY LIKE PARAMETER TO BEM CONFIGURABLE
        parameter integer DWIDTH = 32,
        parameter integer AWIDTH = 14
    )
    (
    //I2C INTERFACE WITH ANOTHER BLOCKS
     input PCLK,
     input PRESETn,

    //INTERFACE WITH FIFO TRANSMISSION
     input fifo_tx_f_full,
     input fifo_tx_f_empty,
     input [DWIDTH-1:0] fifo_tx_data_out,

    //INTERFACE WITH FIFO RECEIVER
     input fifo_rx_f_full,
     input fifo_rx_f_empty,
     output reg fifo_rx_wr_en,
     output reg [DWIDTH-1:0] fifo_rx_data_in,

    //INTERFACE WITH REGISTER CONFIGURATION
     input [AWIDTH-1:0] DATA_CONFIG_REG,
     input [AWIDTH-1:0] TIMEOUT_TX,
```

When the execution is at the point where it is needed to find the values of variable, then, we have to look up in the hash table and replace that variable with the value from hash table.

Abstraction of verilog variables

Verilog variables are classified into three main types.

• Single-bit variables which are 2-valued. It generally consists of wire.

• Multi-bit words which are viewed as unsigned integers. For example, **reg [31:0] PC**

• Word arrays which typically denote memories or register files. For example, **reg [63:0] RF [31:0];**

In order to abstract these variables, we can use UCLID representation of these -

- Two-valued single-bit variables which naturally model as UCLID TRUTH variables.
- Multi-bit words that are interpreted as unsigned integers and translated into corresponding UCLID TERM variables.
- A word array represents a memory or register file and is conveniently represented by an UCLID UF variable.

Abstraction of verilog constants

Constants in Verilog are treated as unsigned integers. There are two types of constants. The first one is small constants. It does'nt require to be abstracted. Second one is large constants, these constants has to be abstracted. For abstraction of large constants we can disregard the numerical value of these constants and treat them as a collection of independent unordered integers.

Depending on the level of abstraction, we can also abstract the small constants in a similar way.

Removing the same registers

We can remove the registers that are redundant to reduce the complexity of the model. Utilization of every register must be close to 1. If the register has less utilization that means it is redundant.

Resources-

Predicate Abstraction and Refinement Techniques for Verifying Verilog

Verilog HDL A Brief Introduction

Project discussion by Dr. Binod Kumar Sir

Github repository of Sample projects

https://opencores.org/

https://link.springer.com/chapter/10.1007/0-306-47658-4_3?noAccess=true

Thank You!!!