Easy open-source hardware description between RTL and HLS

github.com/JulianKemmerer/PipelineC

C Is Easy For Everyone...

Traditional hardware description languages (HDLs) like SystemVerilog and VHDL are notorious for their steep learning curves. High level synthesis (HLS) tools are also notorious for lack of precise hardware control and unexpected results. PipelineC aims to make hardware description easier for everyone from RTL designers to software minded HLS users.

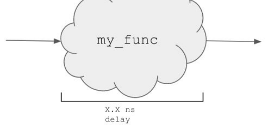
The first step towards easier hardware description is a familiar, basic, C-like syntax. This makes 'simulating in your head' much easier than when reading any old HDL source file: where blocking and non-blocking assignments make combinatorial logic wiring/dataflow difficult to pull from the code (let alone any simulation-only constructs that can creep in). Instead, in PipelineC, simple C functions describe synthesizable feedforward dataflows modules.

Being based on a software language it is then natural to compile parts of PipelineC code with standard C compilers. This makes it easy for users to construct custom ultra-fast compiled-C based "simulations" during development.

Pure Functions = Combinatorial Logic

```
uint8_t my_func(uint8_t input_name)
{
  return ...
}
```

```
-- Generated PipelineC VHDL
entity my_func is
port(
  input_name :
    in unsigned(7 downto 0);
  return_output :
    out unsigned(7 downto 0)
);
end my_func;
```

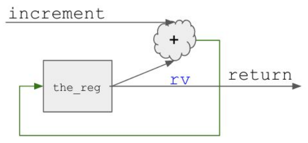


Pure functions exist in SystemVerilog and VHDL. However they compose differently than modules. PipelineC combines functions and modules allowing complex dataflow to be composed all using simple function call syntax.

Comb. Logic + Registers = RTL

Hardware description languages (HDLs) use processes to describe combinatorial logic and registers. A common name for designing this way is register transfer language (RTL). That is, register are inferred and the combinatorial logic transfer function determines the next state value. In PipelineC static local variables are used to declare registers, and the body of the C function describes the dataflow of the combinatorial logic transfer function.

```
uint8_t my_counter(
  uint8_t increment
){
  static uint8_t the_reg
  uint8_t rv = the_reg;
  the_reg += increment;
  return rv;
}
```



Helpful Timing Analysis

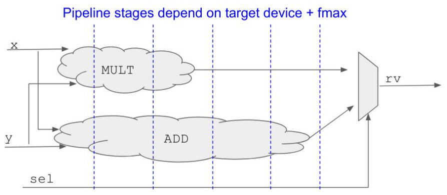
Timing analysis and meeting operating frequency goals are some of the most difficult challenges that digital designers will face on a regular basis. So PipelineC offers the feature of running synthesis and place and route tools from various manufacturers in order to help users become familiar with device specific timing characteristics. Operator delays and paths that cannot be automatically pipelined are reported to the user.

VIVADO
LATTICE DIAMOND DESIGN SOFTWARE
QUARTUS PRIME DESIGN SOFTWARE
OSS CAD Suite
efinity
PYRTL

```
For example:
Function: BIN_OP_PLUS_uint8_t_uint8_t, path delay: 2.236 ns
Function: BIN_OP_PLUS_float_float, path delay: 19.702 ns
```

Automatic Pipelining of Comb. Logic

```
#pragma PART "xc7a100tcsg324-1"
#pragma MAIN_MHZ my_func 100.0

float my_func(
  float x,
  float y,
  uint1_t sel
){
  float rv;
  if(sel){
   rv = x*y;
  }
  else{
   rv = x+y;
  }
  return rv;
}
```



Using the timing feedback provided to the user, the tool is then internally able to automatically pipeline regions of feedforward combinatorial logic. Auto-pipelining works for arbitrarily deep/wide chains of combinatorial logic: ex. hundreds of logic levels have been tested.

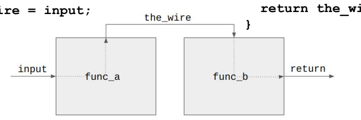
Global Variables

Global variables exists as the mechanism for moving data between / outside / around the data flow of top level `MAIN` functions. Conceptually they are point to point links from a single instance of a function writing the variable, to potentially many instances where the variable is read. Because `MAIN` functions can exist in different clock domains, these variables are also use in declaring clock crossings.

```
uint8_t the_wire; // Globally defined+visible
```

```
#pragma MAIN_MHZ func_a 100.0
void func_a(uint8_t input)
{
  ...
  the_wire = input;
}
```

```
#pragma MAIN_MHZ func_b 100.0
uint8_t func_b()
{
  ...
  return the_wire;
}
```



Clock Crossings

A global variable is used as part of declaring a clock crossing. If this variable is written to and read from in the same clock domain - it is not a clock crossing: simple synchronous wires and registers are inferred. For more complex clock crossings a header file is also generated that presents clock crossings as function calls specific to the type of clock crossing occurring. Otherwise use of the global variable directly is pointed out as an invalid clock crossing.

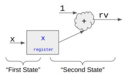
```
uint8_t the_async_fifo[DEPTH]; // Globally defined+visible
#include "clock_crossing/the_async_fifo.h" // Generated

// Try writing 1 element into fifo
the_async_fifo_write_t write =
   the_async_fifo_WRITE_1(wr_data, wr_en);
if(write.ready) // write success

// Try reading 1 element from fifo
the_async_fifo_read_t read =
   the_async_fifo_READ_1(rd_en);
if(read.valid) // read.data valid, success
```

Clock rate aware+checked:

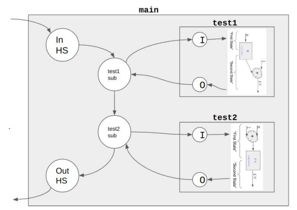
- Wires
- De/serializers
- FIFOs
- Async/false paths

State Machines Derived from Functions

```
// One clock cycle
// before add, a FSM
uint32_t test1(uint32_t x)
{
  __clk();
  uint32_t rv = x + 1;
  return rv;
}
```



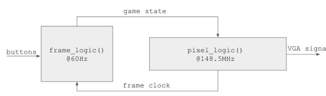
```
// Two subroutine FSMs
uint32_t main(uint32_t a)
{
  uint32_t b = test1(a);
  uint32_t c = test2(b);
  return c;
}
```



Sequential finite state machines can be derived from C code that manually marks clock cycle locations with the `__clk()` function. Each subroutine call location is a new submodule instance. Valid+ready handshaking is used for function entry and return signalling.

Ray Tracing Demo

"Sphery vs. Shapes" is a realtime raytraced bouncing ball game created specifically for PipelineC by Victor Suarez Rovere. There is no software, or CPU of any kind. Animation is done in a simple one frame per clock 60Hz state machine. Global wires are used to link the frame animation state to graphics rendering logic. There is no frame buffer, the 1080P 60FPS 148.5MHz "chasing the beam" pixel clock rendering takes place in a large ~480 clock auto-pipelined C function.



game state
buttons
frame_logic() @60Hz
pixel_logic() @148.5MHz
VGA signals
frame clock