PIPELINEC

WOSET 2022

Easy open-source hardware description between RTL and HLS

github.com/JulianKemmerer/PipelineC

Summary

- What is PipelineC? HDL/RTL/Generator/HLS? Origin Story?
- Fundamental building block examples
- Basic use of the tool
- Advanced features
- Quick graphics example design demo

“We do these things not because they are easy but because we thought they would be easy”

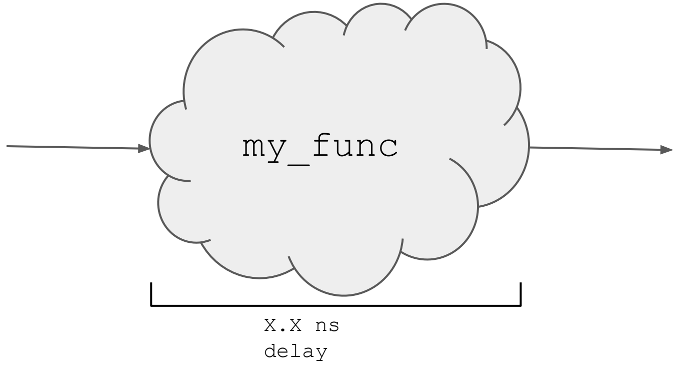
-Internet programming meme

Also apparently my thinking in 2014~2015

Pure Functions = Comb. Logic

HDLs know this: ‘function’ from VHDL/Verilog...
PipelineC combines functions+modules from HDL

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

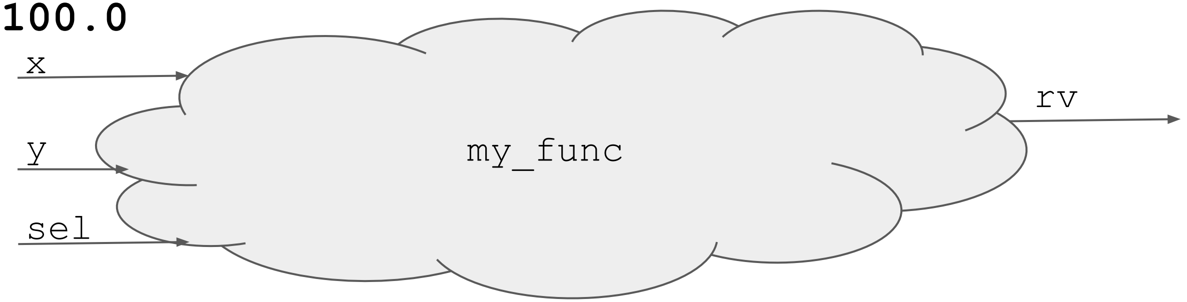


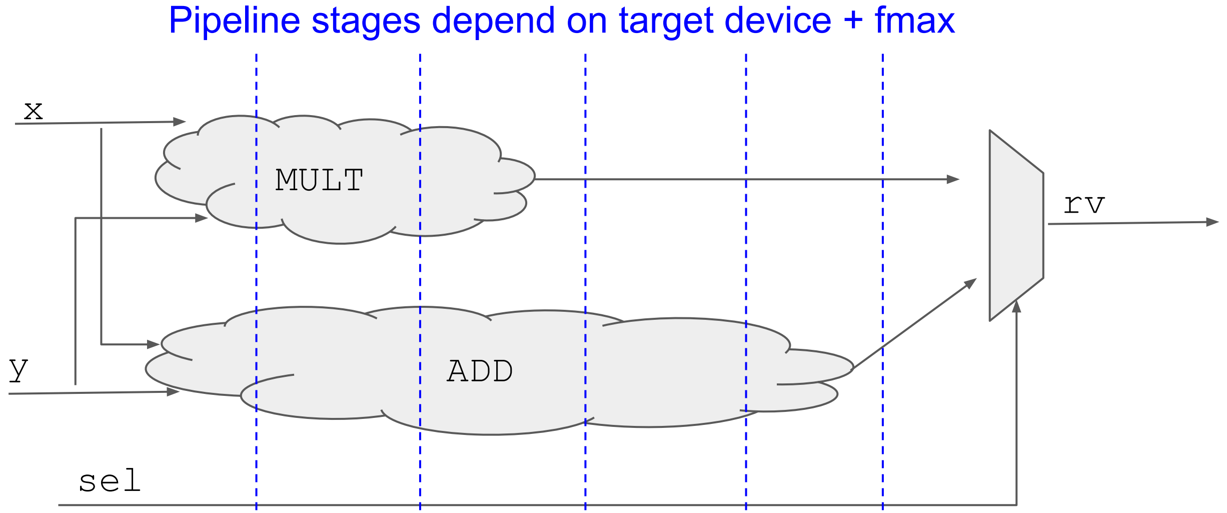
```
-- Generated PipelineC VHDL
entity my_func is
port(
  input_name :
     in unsigned(7 downto 0);
  return_output :
     out unsigned(7 downto 0)
);
end my_func;
```

Comb. logic can be autopipelined

HLS tools know this too (do it ~well and go further, initialization interval !=1 w/ state machines, RAMs, AXI etc)...

```
#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;
}
```





Basic use of the tool...

- **`#pragma MAIN_MHZ my_func 100.0`**
  - Single instance top level `MAIN`
    - Inputs and outputs are top level ports
  - Function runs at 100MHz
  - Most functions don't specify frequency
    - Instead inferred from the call location instance inside a `MAIN`
    - Top level functions also inferred frequency from clock crossings between `MAIN` functions

- **`#pragma PART "xc7a100tcsg324-1"`**
  - Pipelining will be specific to the Xilinx 7 series (Artix 7 100T)
  - Different FPGA models can have vastly different timing characteristics and built-in primitives(ex. DSPs/multipliers)
  - ‘One size fits all solutions’ can be over/under pipelined

Example outputs...

```
pipelinec my_func.c --sim --modelsim
```

- Produces VHDL of II=1 pipeline
  - Some fixed latency of N cycles
  - As required to meet FMAX on specific device
    - ex. 100MHz, 5 cycles
- Generates helper scripts for importing into other tools
  - Ex. Vivado TCL scripts, Quartus .qip IP files
- Or Verilog via GHDL+yosys
  - Yay another layer of code generation

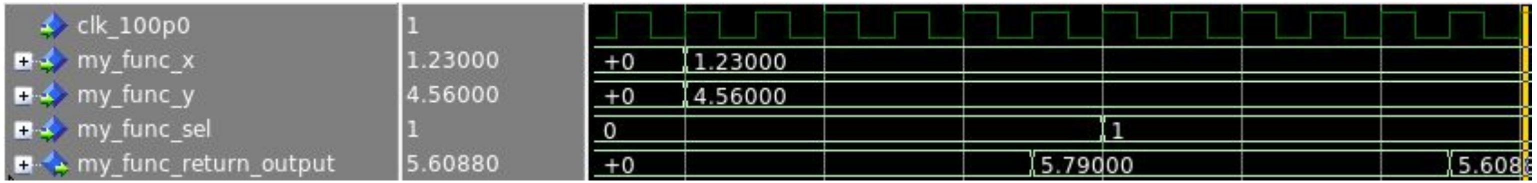
```
-- Generated PipelineC VHDL
entity top is
port(
  clk_100p0 : in std_logic;
  -- IO for each main func
  my_func_x
   : in std_logic_vector(31 downto 0),
  my_func_y
   : in std_logic_vector(31 downto 0),
  my_func_sel
   : in unsigned(0 downto 0),
  my_func_return_output
   : out std_logic_vector(31 downto 0)
);
end top;
```

```
my_func Clock Goal: 100.00 (MHz) Current: 116.16 (MHz)(8.61 ns) 5 clks
Met timing...
================== Writing Results of Throughput Sweep ================
Output VHDL files: pipelinec_output/read_vhdl.tcl
Done.
====================== Doing Modelsim Simulation =====================

          ...
```

Simulation support...

- ~Simulation: compile parts of PipelineC as C - simulate by writing a C/C++ program
- Generated VHDL is completely synthesizable
  - Simulates reliably, limited/no simulator specific constructs/dependencies
- Compatible with all HDL simulators, can generate templates for some:

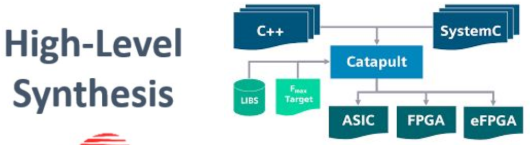
```
--sim

  --modelsim
  --verilator --main_cpp
  --cocotb --makefile
  --ghdl
  --edaplay
```



```
clk_100p0               1
my_func_x               1.23000   +0  1.23000
my_func_y               4.56000   +0  4.56000
my_func_sel             1         0              1
my_func_return_output   5.60880   +0        5.79000     5.608
```

Pause: What is PipelineC not?

- High level synthesis tool
  - Cannot use arbitrary C code
  - No global shared memory model, pointers, threads, etc
- Compiled C based hardware simulator
  - Entire whole designs cannot simply be compiled and run
  - Only parts of PipelineC code can be compiled by C compilers and run (is encouraged).
- Meta-programming hardware generator
  - Uses C type system and preprocessor
- Stitching tool automating the build flow from code to bitstream:
  - Does not create PLLs, etc for clock generation
  - Does not know top level pin locations, etc
  - Does partially automate synthesis runs but automation to final bitstream left to user



YosysHQ/yosys

#1562 **write_cxxrtl: new backend**

amaranth-lang/
amaranth

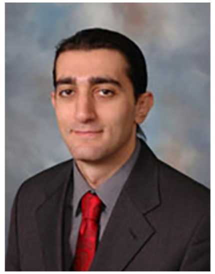
Build your hardware, easily!

So...what is PipelineC?

- Hardware description language
- Started as a comb. logic autopipelining tool
  - Not originally a full HDL...
- Not actually regular C.
  - Can be partly compiled by gcc/llvm for basic 'simulation'.
- Can reasonably replace Verilog/VHDL.
  - Compiler produces synthesizable and human readable+debuggable VHDL.
- Autopipelining is now one of many features “**between RTL and HLS**”

- Thanks Dr. Taskin!

DREXEL UNIVERSITY College of Engineering



Baris Taskin

Professor
Electrical and Computer Engineering

PipelineC is more about...

Functionality ‘inside the module’:

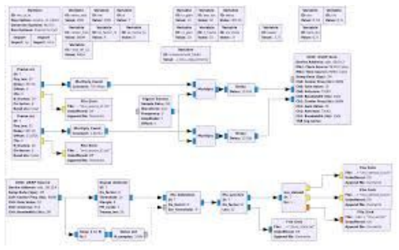
- Describing hardware not design intent:
  - “How does the design do what it does?”
- State machines
- Pipelines
- Inferred BRAMs, DSPs/multipliers, etc
- Not in love with C
  - Want easy, common, low level, compilable
- Just trying to make RTL design easier

PipelineC is less about...

- Purely ‘stitching’ IP/modules/ together
- ‘Build a SoC’ / software+hardware system generators
- Device bus/address space management
- 1-click bitstreams

Related Futures

- High level software, Python, Cloud-Scale
  - LiteX, FuseSoC, SiliconCompiler
  - DFiant Remote, Sabana
- Open source block diagram
  - ex. GNURadio Companion a start?
  - Reason why FPGA tools have block diagrams...



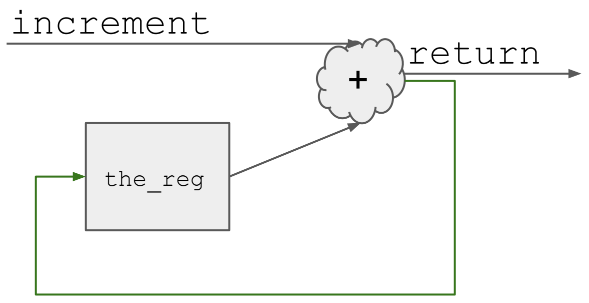
Build your hardware, easily!

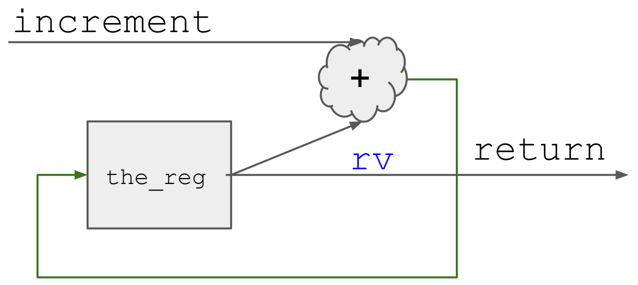
Comb. logic can be attached to registers

RTL knows this: clocked processes, state machines from VHDL/Verilog...
“Explicit” **static** state registers, ~repeating functions in single clock domain
Not C: Each function call location is a new instance!

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

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



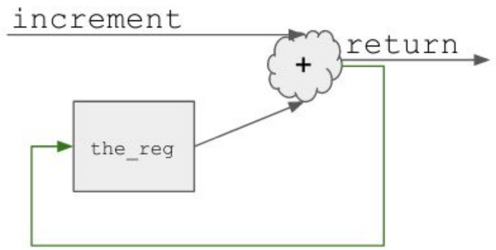


Does not autopipeline!

Getting device specific timing feedback...

- Works with many tools
  - Xilinx Vivado
  - Intel Quartus
  - Lattice Diamond
  - GHDL+Yosys+nextpnr
  - Efinix Efinity
  - PyRTL ASIC Timing Models
- Increment Example:

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



This device specific timing information allows autopipelining!

```
Function: BIN_OP_PLUS_uint8_t_uint8_t, path delay: 2.236 ns
my_counter Clock Goal: 500.00 (MHz) Current: 459.98 (MHz)(2.17 ns)
Cannot pipeline path to meet timing:
START:  my_counter/the_reg =>
 ~ 2.174 ns of logic+routing ~
END: => my_counter_return_output
```

What if accumulating a 32b **`float`**?

```
Function: BIN_OP_PLUS_float_float, path delay: 19.702 ns
my_counter Clock Goal: 500.00 (MHz) Current: 51.46 (MHz)(19.43 ns)
Cannot pipeline path to meet timing:
START:  my_counter/the_reg =>
 ~ 19.433 ns of logic+routing ~
END: => my_counter/the_reg
```

Advanced PipelineC Features...

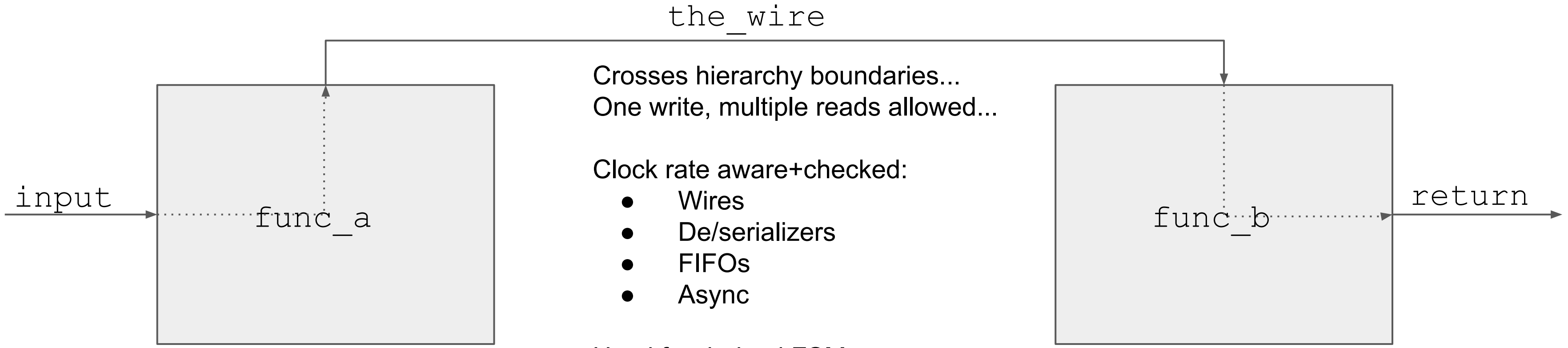
Global Variables / Clock Crossings

Multiple **MAIN** functions...

```
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;
}
```



Crosses hierarchy boundaries...
One write, multiple reads allowed...

Clock rate aware+checked:

- Wires
- De/serializers
- FIFOs
- Async

Used for derived FSM arbitration/shared global resources

Great for board IO, resets, other ~global 'interfaces'. High level ~stitching too.

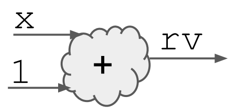
Functions with implicit state can derive state machines (w/ valid+ready handshaking!)

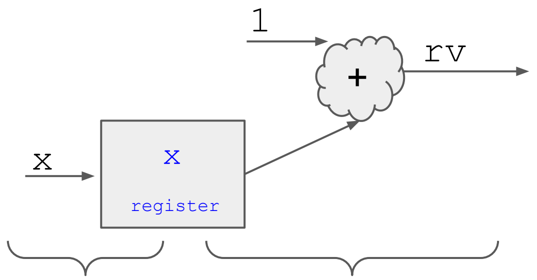
Experimental/New for PipelineC but awesome HDLs like **Silice** knew ~this too!

```
// No clock cycles,
// combinatorial logic
uint32_t test0(uint32_t x)
{
  uint32_t rv = x + 1;
  return rv;
}
```

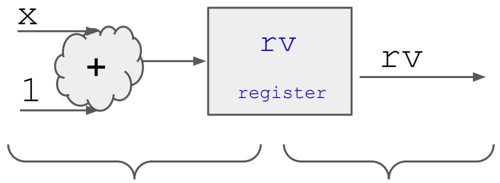
```
// One clock cycle
// before add, a FSM
uint32_t test1(uint32_t x)
{
  __clk();
  uint32_t rv = x + 1;
  return rv;
}
```

```
// One clock cycle
// after add, a FSM
uint32_t test2(uint32_t x)
{
  uint32_t rv = x + 1;
  __clk();
  return rv;
}
```





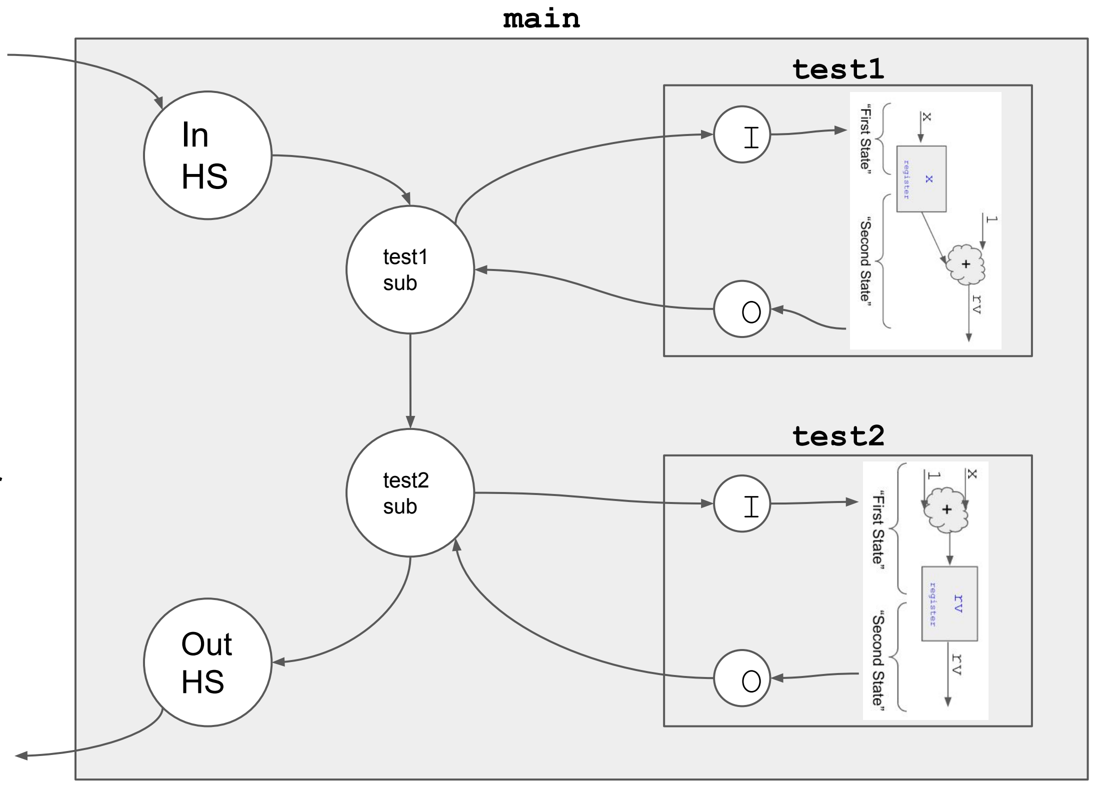
“First State” “Second State”



“First State” “Second State”

Derived Finite State Machines (continued)

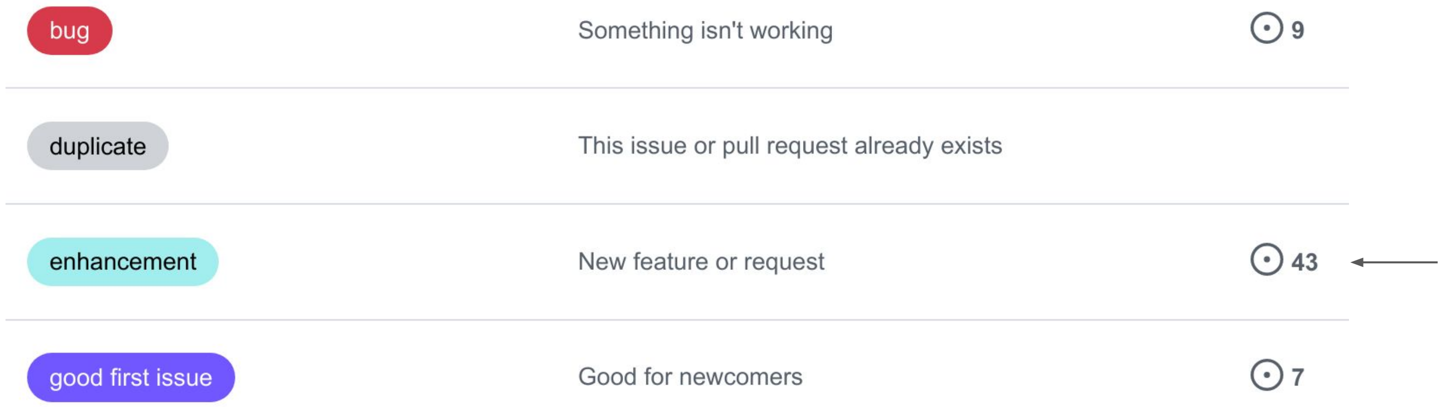
```
// Two subroutine FSMs
uint32_t main(uint32_t a)
{
  uint32_t b = test1(a);
  uint32_t c = test2(b);
  return c;
}
```



- Functions are still modules
  - Just with built in handshaking for function entry+return
  - Func body code derives states
- Logical state diagram
  - Actual states from `__clk()`
- Each function call is a new instance
  - Most of the time*

More Advanced Features:

- Inferred clock enables, since modules~=functions: if(cond) my_module(...)
- Raw VHDL, black boxes, IP, use
- User generated/derived-clocks, async/false paths
- Feedback wires (flow control / ~comb. loops)
- Shared/arbitrated derived FSMs*, ~threads, ~atomics, ~remote system calls
- `volatile` keyword for ... weird stuff
- Generated helper software C code



bug	Something isn't working	9
duplicate	This issue or pull request already exists	
enhancement	New feature or request	43
good first issue	Good for newcomers	7

Example project...

Most Complex Hardware Demo Yet

This work would not have been possible without collaboration from @suarezvictor.

He wrote an interactive **raytraced game specifically using PipelineC**. The game also compiles as C++ and runs in realtime on a PC. He also wrote a C++ parser/generator as part of his CFlexHDL project, that converts operations on fixed point, floating point and vector types to PipelineC-compatible C syntax, using C++ operator overloading.

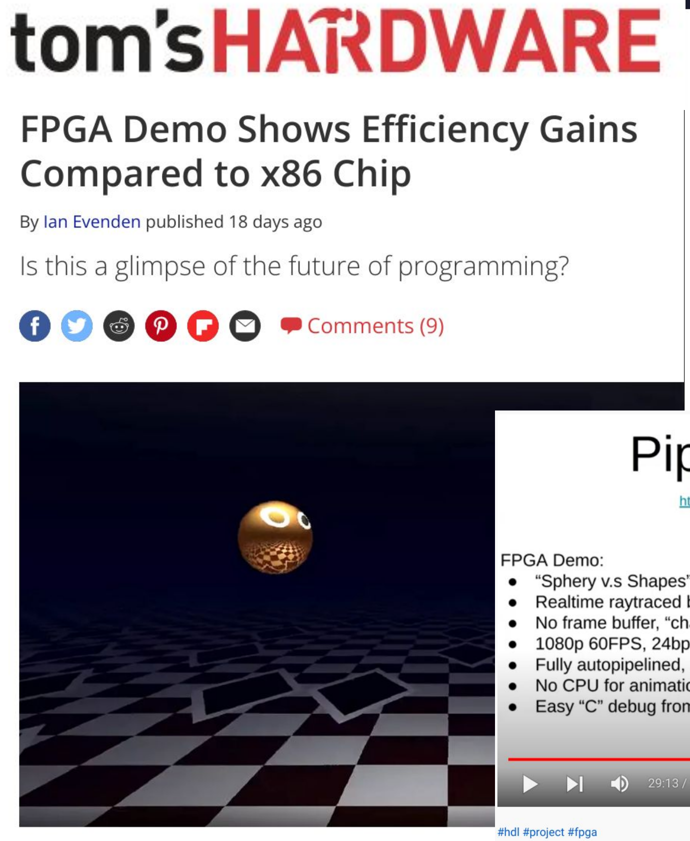
We found many bugs and optimizations opportunities together, over close to a year of work together :) Thanks Victor!

The specifics of the raytracer and parser/generator really should be a talk of its own...

Victor Suarez Rovere
@suarezvictor

https://youtu.be/F8jIJapQbFY

More on Sphery vs. Shapes...



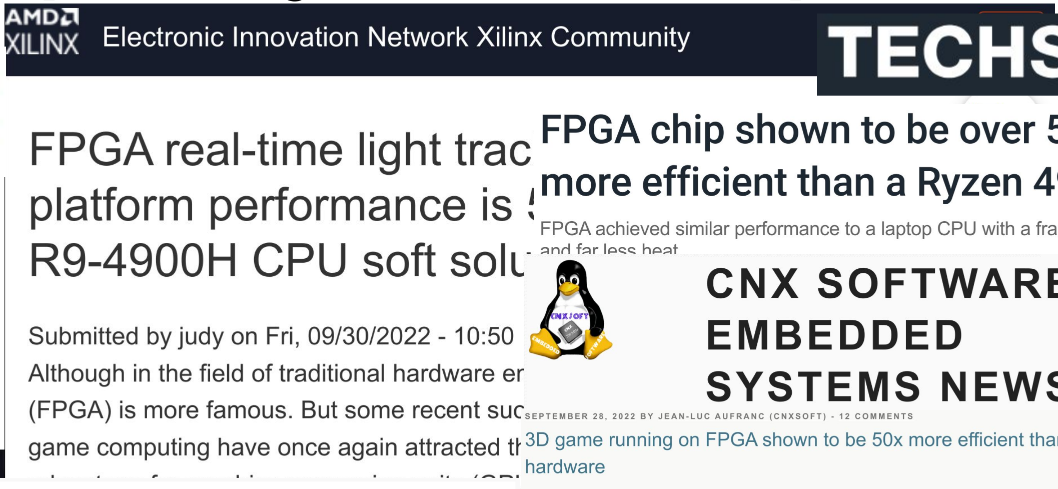
tom's HARDWARE

FPGA Demo Shows Efficiency Gains Compared to x86 Chip

By Ian Evenden published 18 days ago

Is this a glimpse of the future of programming?

Comments (9)



Electronic Innovation Network Xilinx Community

FPGA real-time light trac
platform performance is
R9-4900H CPU soft solu

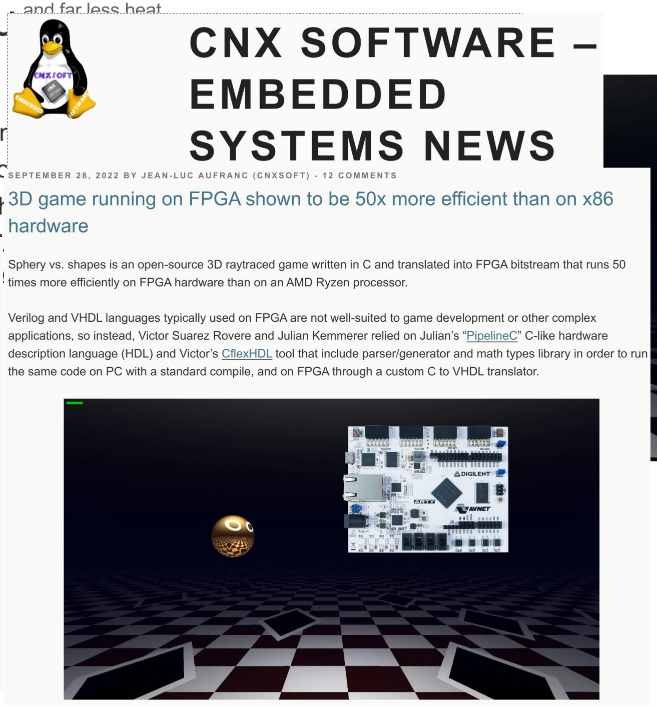
Submitted by judy on Fri, 09/30/2022 - 10:50

Although in the field of traditional hardware er
(FPGA) is more famous. But some recent suc
game computing have once again attracted th

TECHSPOT

FPGA chip shown to be over 50 times more efficient than a Ryzen 4900H

FPGA achieved similar performance to a laptop CPU with a fraction of the energy and far less heat



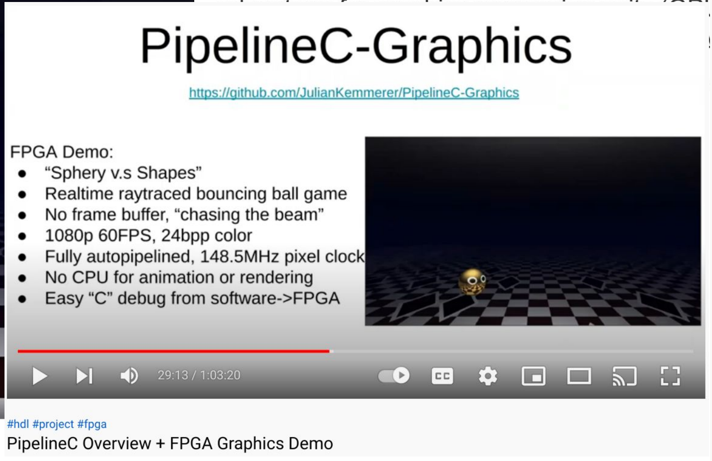
CNX SOFTWARE – EMBEDDED SYSTEMS NEWS

SEPTEMBER 28, 2022 BY JEAN-LUC AUFRANC (CNXSOFT) - 12 COMMENTS

3D game running on FPGA shown to be 50x more efficient than on x86 hardware

Sphery vs. shapes is an open-source 3D raytraced game written in C and translated into FPGA bitstream that runs 50 times more efficiently on FPGA hardware than on an AMD Ryzen processor.

Verilog and VHDL languages typically used on FPGA are not well-suited to game development or other complex applications, so instead, Victor Suarez Rovere and Julian Kemmerer relied on Julian's "PipelineC" C-like hardware description language (HDL) and Victor's CflexHDL tool that include parser/generator and math types library in order to run the same code on PC with a standard compile, and on FPGA through a custom C to VHDL translator.



PipelineC-Graphics

https://github.com/JulianKemmerer/PipelineC-Graphics

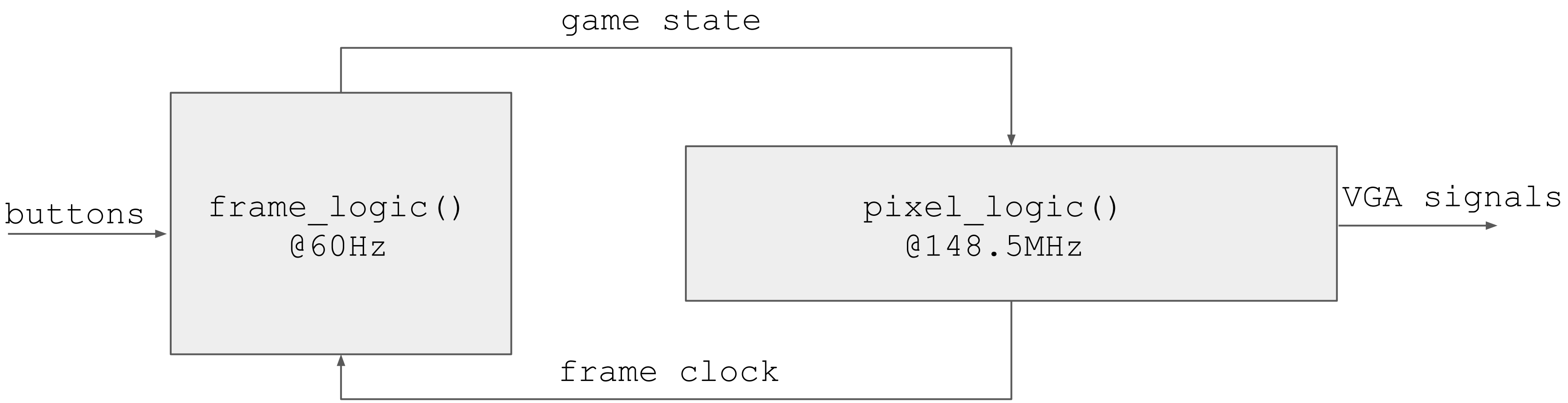
FPGA Demo:
- "Sphery v.s Shapes"
- Realtime raytraced bouncing ball game
- No frame buffer, "chasing the beam"
- 1080p 60FPS, 24bpp color
- Fully autopipelined, 148.5MHz pixel clock
- No CPU for animation or rendering
- Easy "C" debug from software->FPGA

#hdl #project #fpga

PipelineC Overview + FPGA Graphics Demo

Ray Tracing Simple Top Level Design

- Two clock domains, ~two **`#pragma MAIN`** functions, two important global wires



- Things occurring at 60Hz
- Per frame animation, object pos, etc
- **Very slow state machine**

- Things occurring at pixel clock, 148.5MHz
- Generating VGA timing: h/vsync, x,y positions to render
- Generating frame clock, 60Hz
- **Long pixel rendering pipeline from C function**
  `pixel_t render_pixel(uint16_t x, uint16_t y)`

Future Work

- Improvements to autopipelining
  - Report+optimize for area/resources
  - Dealing with feedback / stall signals
- Improved derived finite state machines
  - Shared resources / threads
  - Better code generation
- Template types+functions:
  - How to parameterize functions?
    - Compile time computation
  - Currently ugly preprocessor hacks
- Full support for ‘compile entire PipelineC designs with software compilers’
  - Built in ultra-fast simulations
- Modern hardware compiler frameworks / intermediates + tools like CIRCT+XLS

Thanks folks! Questions? Comments? More code?

https://github.com/JulianKemmerer/PipelineC

https://github.com/JulianKemmerer/PipelineC-Graphics <- w/ white paper and more videos!

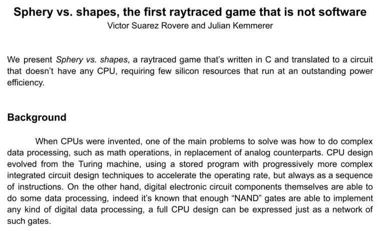
https://github.com/suarezvictor/CflexHDL

Demo Video: https://youtu.be/F8jlJapQbFY

Talk w/ more ray tracer details: https://youtu.be/41nqzbSqBbA

Twitter: @pipelinec_hdl @suarezvictor

Talk to us on PipelineC Discord: https://discord.gg/Aupm3DDrK2



Sphery vs. shapes, the first raytraced game that is not software

Victor Suarez Rovere and Julian Kemmerer

We present *Sphery vs. shapes*, a raytraced game that's written in C and translated to a circuit that doesn't have any CPU, requiring few silicon resources that run at an outstanding power efficiency.

Background

When CPUs were invented, one of the main problems to solve was how to do complex data processing, such as math operations, in replacement of analog counterparts. CPU design evolved from the Turing machine, using a stored program with progressively more complex integrated circuit design techniques to accelerate the operating rate, but always as a sequence of instructions. On the other hand, digital electronic circuit components themselves are able to do some data processing, indeed it's known that enough "NAND" gates are able to implement any kind of digital data processing, a full CPU design can be expressed just as a network of such gates.

END