ABSTRACT

This paper presents PipelineC, an open source project which sits between traditional register transfer language (RTL) and high-level synthesis (HLS). PipelineC aims to make hardware design accessible to domain specialists with basic programming skills by using C-like syntax. Major features include: timing feedback from synthesis and place and route (PnR) tools that drives automatic pipelining of combinatorial logic, as well as derived finite state machines and global point to point wires/clock crossings for composing complex designs outside of feedforward dataflow function call syntax.

1 EASY C SYNTAX FOR RTL

The core idea of RTL: registers holding state and combinatorial transfer functions, naturally maps to variables and functions from software languages like C. In PipelineC function call locations instantiate process-like modules 'executing individual clock cycles', with static local variables as registers for storage. PipelineC aims to make anyone with basic programming skills (reasoning about variables, operations, loops, functions, etc) into a junior level RTL designer or better.

1.1 Function Semantics

PipelineC supports roughly two ways of synthesizing functions as hardware: 1) combinatorial logic/pipelines and 2) derived finite state machines.

1.2 Functions As Modules

Pure functions exist in SystemVerilog and VHDL as a way to describe combinational logic. However they compose differently than modules. PipelineC combines functions and modules allowing complex dataflow to be composed all using function call syntax. Designers can easily map changes in C code to the expected HDL output as in Fig. 3.

1.4 Combinatorial Logic and Registers

(i.e. RTL)

In PipelineC static local variables are used to declare registers, and as in Fig. 4, the body of the C function describes the dataflow of the combinatorial logic transfer function. It becomes very easy to make sense of timing paths from registers based on how static local variables are used with function inputs to drive output return values.

Fig. 5 shows how conditional use of stateful function calls infers clock enables in addition to dataflow muxing.
1.5 Multiple Instantiations

```c
// Two parallel foo() instances
// one bar() instance
uint8_t my_func(uint_t cond, uint8_t x)
{
    uint8_t rv = 0;
    if (cond)
    {
        rv = foo(x);
    }
    return rv;
}
```

Figure 6: Function call dataflow semantics

Each function call location is a new module instance in the current clock cycle dataflow. Recursion is not allowed, but functions can otherwise be composed as in normal C, creating more complex functions out of smaller ones. In Fig. 6 multiple foo() function call locations correspond to multiple module instantiations.

1.6 Loops

```c
// Three iteration loop
uint8_t my_func(uint8_t x)
{
    for(i=0;i<3;i++)
    {
        x = foo(x);
    }
    return x;
}
```

Figure 7: Dataflow loop semantics

As in traditional HDLs, loops will be compile time unrolled when elaborating the design. Unbounded while, or for loops that cannot be unrolled are not allowed. The PipelineC tool will unroll all single clock for-loops to produce comb. logic (as opposed to multi-clock loops that are not unrolled but instead can derive finite state machines). Fig. 7 shows how three loop iterations are unrolled to produce three module instances.

1.7 Software C Compilers

Parts of PipelineC code can indeed be compiled and run with standard C compilers like LLVM[1] and GCC[2]. However, this is not generally true for entire top level designs. Instead ‘compiling-as-C’ can be used on smaller units of code and makes for an easy option to construct ultra-fast module level ‘simulations’ for quick development.

2 HELP MEETING TIMING

RTL design involves orchestrating what combinatorial logic can fit in timing paths between registers. As such, timing analysis is one of the most difficult challenges that digital designers will face on a regular basis. However, software developers will not be familiar with the timing characteristics of various operations implemented differently on physical devices, etc. To aid in this, using the #pragma PART to specify FPGA hardware, PipelineC offers the feature of running synthesis and PnR tools from various manufacturers. Like software profilers identifying long running pieces of code, this feedback helps identify long timing paths in a way that doesn’t require intimate knowledge of device specific synthesis and PnR flows.

2.1 Function Delay

```c
uint8_t two_adds_int(uint8_t value, uint8_t inc)
{
    value += inc;
    value += inc;
    return value;
}
```

Function: BIN_OP_PLUS_uint8_t_uint8_t path delay: 2.236 ns (447.2 MHz)
Function: two_adds_int path delay: 5.648 ns (278.707 MHz)

```c
float two_adds_float(float value, float inc)
{
    value += inc;
    value += inc;
    return value;
}
```

Function: BIN_OP_PLUS_float_float path delay: 19.702 ns (50.75 MHz)
Function: two_adds_float path delay: 37.886 ns (26.395 MHz)

Figure 8: Example path delays for two combinatorial addition operators in series

Each function in the design is synthesized and the user receives a report of each function’s combinatorial delay / unpipelined maximum operating frequency. Figs. 8 and 9 show examples delays for BINary OPerator PLUS (addition) between uint8_t’s v.s. float. When operators are combined in larger functions the larger combinatorial delays can be observed, helping users build an intuition for how much logic certain operations require.

2.2 Critical Paths

If the tool encounters a critical path that will not be automatically pipelined (due to feedback signals / static local
PipelineC: Easier Hardware Description Between RTL and HLS

Figure 9: Single cycle feedback paths, integer vs floating point example

3 NONSTANDARD FEATURES

The ease of 'writing hardware in C' as described so far has ultimately been just a 'new skin' over concepts present in existing HDLs and synthesis+PnR flows. But just as simple stateful functions map to RTL concepts there are yet other ways of interpreting what’s been expressed in code. The next sections describe several other HLS-like advanced features that can be used to create complex designs.

3.1 Automatic Pipelining

Figure 10: An automatically pipelined function

Using synthesis+PnR tool driven timing feedback, the PipelineC tool is able to automatically pipeline regions of feedforward combinational logic as in Fig. 10. This initialization interval of one (II=1) auto-pipelining works for arbitrarily deep/wide chains of combinational logic and composes as expected with function call syntax. This functionality is equivalent to what proprietary HLS tools can do when functions are marked with #pragma HLS pipeline II=1 and similar extra specifiers.

It is also possible to compose dataflow that mixes feedforward autopipelined pure combinational logic functions and not-autopipelined stateful modules with local feedback. In Fig. 11 the MAIN function my_func contains instances of both pure bar() and stateful functions foo(). Only the pure functions can be automatically pipelined. This increases the latency of the design but the entire dataflow of signals is correctly retimed/delayed to behave as a single II=1 pipeline.

3.2 Derived Finite State Machines

State machines can be derived from code that executes over several clock cycles (as opposed to one-clock combinatorial logic). As in Fig. 12, cycle accurate control is maintained as state transitions are derived from manual __clk() clock cycle locations. This is similar to Silice’s[3] clock step operator.

Note that because the user is describing clock-by-clock behavior, these functions, like plain RTL style code, will not be autopipelined. Everything between __clk() markers is RTL-like, but multi-cycle state machine subroutine calls and loops have different semantics.

3.2.1 FSM Loop Semantics. Like regular C language, unbounded runtime-variable for/while loops are allowed. In PipelineC they are used to derive finite state machines. The body of the loop must take one or more clock cycles to execute. That is, multiple iterations of the loop cannot fit into a single clock cycle. The loop body must execute a __clk() or invoke a function that does, etc (otherwise an ‘infinite iterations per clock’ might be implied). This has to do with the ‘sequence of states/instructions’ nature of the hardware: in order to jump back and re-execute a sequence of states again (as in a loop), this requires at least one clock cycle to branch the program counter and cannot be done many / unlimited times in every clock.

3.2.2 FSM Function Call Semantics. Complex nested / hierarchical state machines can be described using familiar function call syntax, however recursion is still not allowed. This is because each instance of state machine execution has one copy of subroutine FSM resources (ex. func args, local variables) available for use at a time, i.e. the ‘call stack’ can only include a function once as there are no other copies of the design but the entire dataflow of signals is correctly retimed/delayed to behave as a single II=1 pipeline.
of resources to be used for recursive calls. Note that unlike RTL style code there is not an exact mapping of function call locations equating to rendered module instantiations. This stems from the fact that there can be multiple ‘threads’ of these derived FSM instances running at once and sharing resources.

Figure 13: Multiple FSM instances use handshaking to share a resource

By default multiple state machine ‘threads’ (i.e. FSM module instances) do not share subroutine function resources, each thread is running its copy of the subroutine. However, by specifying the FSM as SINGLE_INST ‘single instance’, as in Fig. 13, only one copy of the subroutine will exist in hardware. To share the FSM among multiple simultaneous threads the tool generates arbitration logic for function entry and return valid-ready handshake signalling. This allows for shared/atomic/one-at-a-time use of the shared function resource.

3.3 Global Variables

Global variables exist as a mechanism for moving data between / outside / around the feedforward dataflow of top level #pragma MAIN functions. Their simplest and default behavior are point to point wires from a single instance of a function writing/driving the variable, to potentially many instances where the variable is read. Point to point wires allow for composing designs in intuitive ways with easier syntax than routing wires through nested instantiated modules as required by traditional HDL. Because MAIN functions can exist in different clock domains, these variables are also used in declaring clock crossings.

3.3.1 Example: Many Read Instances. One basic example of using global wires is for exposing top level IO ports as wires that can be used anywhere in the design. The go-to example for this is reset. Fig. 14 shows connecting a top level input port to a global wire called sync_reset_wire. This wire can be read from in any function / module without the need for explicitly wiring the reset down into every instance from the top level port.

3.3.2 Example: Many Write Instances. Wires typically have one driver connected to one-or-more things driven by that wire (as in the reset example above). That is, a global variable is usually written to in just a single function instance, and read from in many other simultaneous function instances. However, if a global variable is written to in more than one function instance then this is a ‘multiple driver’ problem common to all HDLs and by default the tool will produce an error message.

Alternatively, PipelineC allows global variables to be marked with #pragma INST_ARRAY which, as in Fig. 15, allows multiple driver instances on a single wire to map to an array of single-driver wires. All of these simultaneously driven values are available all at once in a single clock cycle. The user decides how to handle the situation, often implementing custom one-at-a-time arbitration logic.

3.4 Clock Domain Crossings

If a global 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. Otherwise, a header file is generated that presents the data transfer as WRITE and READ function calls specific to the type of clock domain crossing (CDC) that is occurring: In Fig. 16 for example: an asynchronous FIFO write function has input arguments of write enable and data signals, with a return value containing flags like ready / full. Note that the the_async_fifo variable is not directly used in code: use of the global variable directly
PipelineC: Easier Hardware Description Between RTL and HLS

3.5 Floating Point

PipelineC has built in support for the standard C float floating point type. This 32 bit type has 1 sign bit, 8 exponent bits and 23 mantissa bits. An alias for this type in PipelineC is float_8_23_t, which comes from the built in float_e_m_t.h header supporting custom floating point types. That is, any number of exponent or mantissa bits are supported, ex. bfloat16 with 8 exponent bits and 7 mantissa bits is available as float_8_7_t. All floating point operators are pure functions and can be arbitrarily autopipelined.

4 OUTPUT PRODUCTS

The tool renders human readable VHDL and supports all HDL synthesizers and simulators (VHDL to Verilog conversion can be included via GHDL+yosys[4], ex. for Verilator[5]). Users can also insert arbitrary chunks of hand written VHDL, closing any gaps that exist when using PipelineC for full designs (ex. to wrap/instantiate existing IP cores).

5 WORKING HARDWARE DEMOS

Demos working in hardware include: Raytraced retro style bouncing ball game, realtime MNIST digit recognition neural network, cryptographic hashes, RISC-V CPU, packet processing, I2S audio, AXI integration, and more on github.

6 FUTURE WORK

PipelineC started as a 'simple pipelining tool'. Progress from there has been incremental in spare time, a sort of multi-years long weekends hackathon project. Immediate future goals include addressing open issues the community has brought forward. Then improving the internal code generation for derived finite state machines, the most recent major new feature. The hardware description for the those state machines is generated PipelineC code, which acts as a sort of intermediate representation - which brings up questions of possible HLS->PipelineC->RTL flows. In terms of brand new features, many syntax related things like C++ template type support and function overloading are goals. Improvements to the iterations required for autopipelining and better (perhaps graphical) ways of presenting rendered HDL names and details of timing/resource usage feedback are also being considered. Resource reuse is an interesting area to explore if pipeline initiation intervals are allowed to be greater than one II>1 or faster integer rate ratio clocks can be used. Adding built in support for fixed point types like floating point types seems to make sense. Finally, being able to 'compile as C' the entire design for fast simulations would be great.

The project is looking for contributors, especially those knowledgeable of emerging compiler technologies like MLIR[6] + CIRCT[7] that might help achieve many of these goals. As an FPGA engineer, I have no formal compiler's education, so some hand holding is likely needed.

7 CONCLUSION

There is a huge world out there of software developers, many with embedded C backgrounds that can’t wait to get into digital design. PipelineC should empower non hardware experts with basic programming experience to just go hacking at some C code and see what comes out the other side of synthesis+PnR. Always happy to take questions, comments, suggestions, for making the tool better. -Julian

REFERENCES
[1] LLVM Project: llvm.org