I have always wanted to create my own CPU design using VHDL or Verilog and synthesize it inside an FPGA. In the next collection of articles, I will showcase my progress of designing a very simple cpu in Verilog and progressively making it more complex with features like pipelining, memory and IO, coprocessors and it's own assembler using Flex and Bison.

Field Programmable Gate Arrays (FPGAs) are a classification of logic chips that include basic building blocks of complex integrated circuits (see Very Large Scale Integrated Circuits (VLSI) and FPGAs). A hardware schematic or behavioral system is coded using an High Definition Language (HDL) such as VHDL or Verilog and the FPGA is programmed to take on the behavior of this HDL description. *In short, with an FPGA you can make it behave logically however you wish...including the behavior of a custom designed microprocessor!*

Here is the chapter layout, completed chapters are already hyperlinked:

Chapter 1 - Basic CPU

In the first chapter we implement a very basic cpu to introduce ourselves to the verilog behavioral language, basic cpu structure and instantiating the cpu module in a basic testbench.

Chapter 2 - Branching

Branching instructions are added to the cpu. A single subroutine instruction that stores the return program counter (PC register) is also implemented.

Chapter 3 - Your First Assembler (yfasm)

What good is a cpu without an assembler? In this chapter we use flex and bison (i.e. lex and yacc) to make an assembler for our cpu. The assembler generates a binary file that can be loaded by the verilog $readmemh statement, or in a synthesized FPGA we would use the binary file to initialize the instruction memory.

Flex and bison are used to create many compilers for many languages and was also used to create the GCC compiler collection. For flex we create a lexer file that defines how the input assember program as text is broken down into tokens, identifiers and constants. For bison we

create a language grammer file that determines the syntax of the tokens, identifiers and constants coming from the lexer and specifies actions to emit the assembler binary output. The flex and bison input files are processed at compile time and these programs generate C code that implements our lexer and parser and a small main function ties it all together for a complete assembler.

Chapter 4 - Pipelining

To increase the throughput of our cpu we add pipelining to each cycle of the cpu. This means that at each clock cycle another instruction is fetched, decoded, executed and wrote back just like the pipelined operation of a manufacturing assembly line. This gives us a theoretical four times increase in throughput. However, hazards and pipeline stalls must be considered which may decrease the performance slightly.

Chapter 5 - Optimizing the design

In chapter 4, we will pipeline the cpu yet still not consider any further optimizations of the overall design. Until this point, our goal with each iteration of the cpu design is to maintain easy to understand verilog code as a tutorial. In this chapter, we consider all possible optimizations including changing code style or language constructs used so the verilog compiler will generate better logic. Timing profiling will also be used to determine where bottlenecks of the cpu exist. Simulation will still be used so optimizing such things as FPGA floorplans and routing will likely be left for a later chapter, possibly called FPGA Synthesis.

Chapter 6 - Memory and IO

In this chapter we create an external cpu bus to connect to large memory and external IO.

Chapter 7 - Coprocessors (maybe)

In this chapter we create a coprocessor to perform application specific processing and offload the main cpu. This chapter is intended to be a tutorial on creating efficient coprocessor busses and coprocessor communication. Probably, the coprocessor design will be a fixed point math processor with 1 or more multipliers and cross-multipliers. (Multiply and Divide will probably not be an instruction in the cpu since these instructions have horrible long timing requirements.)

Chapter 8 - Retargeting GCC

The GNU GCC compiler collection contains a c/c++ compiler that can be targeted to many different hardware platforms. In this chapter, we will create a new GCC platform so this compiler

can generate c/c++ code for our cpu. We will most likely use the current Atmel target platform code as a reference, and edit to fit our cpu. GCC working on our cpu is the first step to compiling and running uclinux on our cpu!

Chapter 9 - FPGA Synthesis

Using our sample reference design given in Appendix A, we synthesize our cpu, coprocessors and IO processors inside an FPGA. The sample reference design includes some input and output peripherals to see our cpu work interactively.

Chapter 10 - Building Linux! (maybe)

If benchmarks show the cpu is robust enough, in this chapter we will compile and run uclinux on the reference hardware design. This will involve downloading and compiling the uclinux kernel using a custom config and the retargeted GCC compiler we made from chapter 8.

Appendix A - Sample of a yfcpu hardware reference design

Schematics and pcb layout is given for an example FPGA hardware design that is capable of running our cpu uclinux.

Appendix B - Creating our microcomputer's BIOS

To boot the uclinux kernel and provide some basic services our microcomputer will require a BIOS. We can likely port a small BIOS from another uclinux platform that will easily compile with our retargeted GCC.

Appendix C - Installing ssh and apache

A small chapter on installing and configuring ssh and apache (php?) on our new uclinux yfcpu! Includes a sample dynamic web page that shows some details of our cpu activity.