# 8051 Verilog RTL IP core

## FPGA coding: from logic to embeded

How do we begin FPGA coding? Of course, sometimes, FPGA coding is restricted to RTL sub-set of Verilog HDL. First, we have a big block of different input/output/inout ports. Then, we have our top design module, like this:

module xxx\_top(

input clk,

input rst,

…

output [7:0] xxxdata

);

…

endmodule

This top module will have several sub-modules, and each has his own sub-modules or leaf cells.

…

…

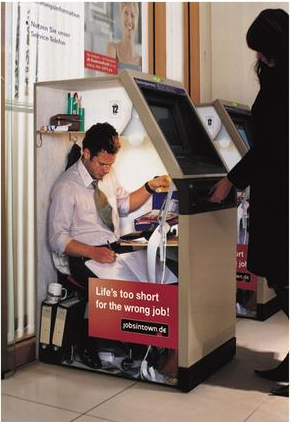
TOP

The above diagram shows us what a design looks like. It has different ports defined according to functions and is full of various leaf logic cells. These ports and leaf cells are connected with wires and this connection will make this design work to meet our need.

We do not need to place thousands of leaf cells and connect them manually. A synthesis tool will transform verilog files written according to RTL styles to netlist which describes cells and connections.

This is a typical logic design flow. It need us to manage RTL design files, make sure their grammar right and meet various resource and time requirement. It is a hard flow, just like a ballet dancer, always make your foot straigt and play some motion. Everyone could be a dancer, but not a ballet one, because it takes great effort to have this special technique. Every one could write C code without great effort, but not verilog RTL code.

There is a method to change, turn verilog coding to C coding. Firstly, let us observe an ATM machine. If the cover of an ATM machine, what we could see? It just likes the left of the below picture. In an ATM machine, it should be full of miscellaneous mechanical and electrical devices, just like our logic design. But how about the right picture, a worker stays inside an ATM machine? If a man inside, he will serve you smarter and smoother. As to our logic design, we could manage a man inside, too. That is embeded programing.



In embeded programming, a smart CPU is used to execute binary code compiled by a C /assembler compiler. In below diagram, miscellaneous logic cells are removed and a CPU IP core is embeded withits regular fellows: ROM and RAM. ROM is a logical symbol, which means binary code is stored byte by byte. RAM is intermediate store space, which is used by CPU core when binary code is interpreted and executed. This CPU IP core will act like a man, which monitors input ports and change output ports accordingly.

…

…

TOP

CPU IP Core

ROM

RAM

The smarter ATM machine cost more, because you should pay for a worker and he will not work all day and night. Similarly, the central part: CPU IP core is too expensive to afford. Commercial CPU IP cores are developed by some big company. It is a long procedure to apply for trial ones. If you turn to free open source CPU IP cores, another problem arises: how could I trust it, if it works badly? Commericial cores are reliable but you have to seek help from application engineers, who will teach you how to fit them into your design and charge a lot. No one will tell you how to manage serveral files and if problems are available, how to trace them between these files?

I am being devoted to provide reliable CPU IP cores, which is compitable with commerical ones. My major concern is whether you could utilize main-stream commerial embeded tool kits directly. You are not a dedicated embeded developper, and could not afford one assistant. You will have to take advantage of easygoing IDE tools, such as RealView MDK. IAR Embedded Workbench IDE. If you have to seek help, it is easy because there is a lot of documentation on web and milions of engineers who work on it.

How could I trust you? You do not, but you trust yourself. You will follow me to master how these tiny and powerful CPU IP cores are developed. It is a kind of state-of-the-art. RTL coding is a hard developpment and if you devote into it in enough time, you will have you own understanding and products, too. Besides that, commerical IP cores have all kinds of functions to accommodate different customers and they contain redundant parts for most engineers.

I have developped one ARM9-compatible CPU IP core, which only has more than 1700 lines for all functions. I managed to start Linux OS in a verilog simulator. It is illustrated by my book named: “ARM9-compatible soft processor design--FGPA based”. Many readers have fitted it into their own FPGA designs and benefited from its simple interface and elegant style. It is easy to trace internal signals because it contains less signals to treat and easy to be instanced because it is only one file of 1700 lines.

Its sister book: “practical design on 8051 soft processor” has presented a more powerful soft processor: all miscellaneous 111 instructions supported, only 700-line description. A 32-bit CPU core is too powerful for FPGA designer, and they perfer an appropriate one to meet their need. An 8051 soft processor is great for FPGA implementation: its powerful 111 CISC instructions will make binary code shorter.

The Moore Law is making the same die size have more transistors, which means large capacity FPGA/ASIC. Since that, you do not need an ATM machine made by electrical devices, and you could afford a worker inside your design. My powerful CPU cores will be a essential fragment, which acts like a man inside.

If you like my design, just follow my book, and enjoy it.

## Elaborating 8051 ISA

Intel introduced 51 CPU in the beginning of 80’ in the 20th century. This good old fashioned 8-bit CPU is so popular that millions of engineers and students are familiar with its architecture. 8051 is a name of one kind of popular MCU, which is a mixture of fixed peripherals. We could tell that one typical 8051 MCU has 32 descrete I/O pins, two 16-bit timer/counters and one full duplex UART etc. These peripherals are combined with 8051 so closely that most of 8051 CPU IP cores have to include implentation of them.

Why should we only utility 8051 ISA and abandon their fixed peripherals? Keeping these fixed peripherals will make developped 8051 code work well without any modification. But embeded engineers are so smart that they make very less code deal with physical interface, and most code is stable and no need to change. It is not annoyance because FPGA programmers are newbies for embeded design and they just treat embeded design as auxiliary aid.

Let us simplify 8051 CPU system. We will abandon any peripherals or registers attached and make sure that any compiler will not refer to these abandoned peripherals implicitly. For example, a SP pointer is called implicitly by some instructions and it should be contained in CPU core internally. A timer is not and it is called explicitly. Abandonning the timer is necessory and harmless, but a SP pointer should be included in CPU core.

The below diagram shows a simple illustration of CPU mechanism. The left bar is labeled as “binary code”, which is generated by compilers. It contains massive instructions, which are put in order according to C code. One of them is fetched to decode and execuate. “Processor” implys processing data. Most of data are from data pool, which contains massive data. Usually, one of them serves as one of its operands. The “Processor”is a machine to process data one by one.

binary Code

active instruction

……

……

active data

……

……

data pool

……

……

instruction A

data B

Processor Model

8051 is a kind of CPU. Its “code pool” or “binary code” has 16-bit address line, so it has 64K bytes to store programs. Normally, an 8051 processor will fetch its first byte from address 0000. Anyway, its “data pool” is complex. It has two different type: 16-bit address width “XDATA” and 8-bit address width others.



We know, other processors have another part: registers. Registers are inside the processor and data are loaded to them temporarily. These processors of 16/32 registers are called the “RISC” type. Every data in data pool has an address, which is used in fetching and storing. An 8051 system doesn’t have registers like RISC type, which are named as R0~15 etc. The registers of 8051 are labeled to some address.