Example of Full System Implementation

Stage 1: The RISC-V Toolchain

Submit by 16/03/2024 at 23:59

1 Objective

In this final multi-part assignment, you will get an overall perspective of everything we learned in the course and a bit more, by running a full system implementation. This implementation will take a simple piece of **C** code, compile it for a **RISC-V** processor implementation, run a cycle-accurate simulation of the code upon the RTL, synthesize, place and route a system-on-chip (SoC) with the processor, and run a gate-level simulation to ensure that the final design is true to the original intent and even measure the power consumption of your program.

Specifically, this project will include the following stages:

1. Create a simple SoC including a **RISC-V** processor, **SRAM** instances, and **I/O** pads.
2. Compile an example **C** program ("Bubble Sort") for a **RISC-V** architecture implementation.
3. Run a testbench in **Xcelium** (`xrun`) that loads the binaries of the compiled **C** program into the memory of the **SoC** and executes the program.
4. Synthesize the RTL of the **SoC** into a gate-level netlist.
5. Create a floorplan for the **SoC**, including deciding on the positions of the **I/O pads** and the **SRAM** macros.
6. Continue the place and route flow to get a **DRC** clean layout that meets **timing**.
7. *(Optional)* Run a gate-level simulation on the post-layout netlist to verify design correctness.
8. *(Optional)* Estimate the power consumption due to gate toggling and leakage when running your **C** program.

2 Project Overview

The project will be divided into three stages (plus one optional), as follows:

- **Stage 1**: The RISC-V Toolchain: During this stage, you will take a program written in a **high-level language**, compile it, load the binaries into the processor memory, and run a logic simulation on the processor **RTL**.
- **Stage 2**: Synthesis: During this stage, you will run synthesis on the **SoC**. To do this, you will need to first compile **memory** macros and integrate them into the **RTL**.
- **Stage 3**: Place & Route: During this stage, you will floorplan your chip and run place and route to provide a **post-layout block**. To do this, you will need to define your **I/O ring**.
- **Optional Stage 4**: Gatelevel Simulation: During this stage, you will verify that your **C** program runs on the **post-layout netlist**. In addition, you will save **toggling information** to extract power consumption.

Today, you will be starting with Stage 1. But first, let's give you an overview of the **SoC** environment.

3 Getting to know your SoC

In this project, you will take a soft IP – a **RISC-V** processor – and combine it with some **memory** and **I/Os** to create a simple system-on-chip that can run a program. We have provided you with the entire framework, but you will need to make a few changes. To start, let us overview the three components of the **SoC**:

3.1 RI5CY Processor

You may have heard about the revolution going on in the world of computer architecture, but even if you haven't, we will now introduce you to **RISC-V**. **RISC-V** is a completely open-source RISC architecture, backed by a foundation with members spanning academia and industry (https://riscv.org). Many implementations of **RISC-V** cores are available, but we chose to run our example on the **RI5CY** (pronounced "*risky*") core, which is a low-power 4-stage embedded core that is provided by our colleagues at ETH-Zurich, as part of the **PULPino** platform (https://www.pulp-platform.org/ and https://github.com/pulp-platform/pulpino).

We have downloaded and configured a version of the **RI5CY** and provided it to you under `sourcecode/rtl/riscv-master`. The top-level module of the **RI5CY** processor is called `riscv_core` and the top-level **System Verilog** file is provided at `sourcecode/rtl/riscv-master/riscv_core.sv`.

3.2 Memory Blocks

A processor by itself cannot do much. Any serious block of code needs some memory, and a processor, as a pretty serious block of code, needs memory to store its instructions and manage its data. However, since the actual amount of memory is configurable ("*the more the memorier…*"), it is provided externally through a pair of busses that are interconnected within a wrapper (…a "**SoC**").

In the case of our small **SoC**, we have defined some **instruction memory** and some data memory that are divided into *banks* with 4 byte words.

Note that the separation of instructions and data into separate memory banks is known as a "**Harvard Architecture**". This is in contrast to a "**Princeton Architecture**", which stores instructions and data inside the same memory space with a single bus to access both of them. The Princeton Architecture is also known as a **Von Neumann Architecture**, named for John von Neumann, who is credited for the stored-program architecture concept, despite it actually being conceptualized by Alan Turing.

So where do we get the memory banks from? We will get to that at Stage 2 of the project. For now, all you need is a behavioral (RTL) model of the SRAM banks that can be used for logic simulation. These models can be found at `sourcecode/rtl/soc/sram_sp_hde_m*_simple.sv`.

3.3 Top-level and I/O Pads

The **SoC** itself is the entire system that combines the **processor**, the **memories**, the **buses**, and other **peripherals**. We have a very simple SoC that basically only consists of the processor and the memories. The *toplevel module* of the **SoC** is defined at `sourcecode/rtl/soc/lp_riscv.v` On top of the SoC, we have another hierarchy that just wraps together the SoC with the interface to the outside world – the I/O cells and bonding pads. So the overall top-level of the entire project is defined in the file `sourcecode/rtl/soc/lp_riscv_top.v`. The top-level has:

- 8 single bit input ports (such as `PAD_CLK` and `PAD_RST_N`)
- 3 busses
- Single output bit

We will discuss the I/Os in depth during Stage 3 of the project.

An overall schematic of the SoC can be seen in Figure 1.

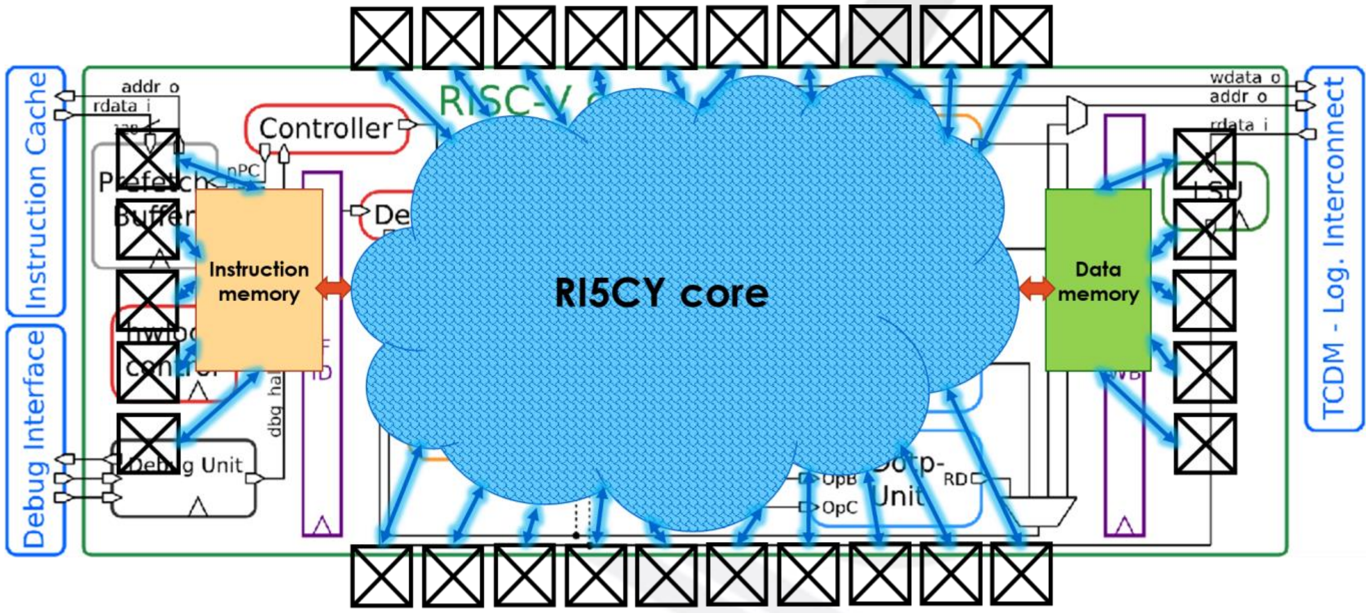

Figure 1: The Full System Implementation SoC

4 Stage 1: The RISC-V Toolchain

The first step in our project is to create a "binary" to load into our processor and run a program. A binary is the assembled and linked code of a program, generally written in a high-level language, such as **C**. The process of taking such a program, compiling, linking to provide the binaries is called the "toolchain". We have provided you with a **RISC-V** toolchain that uses the popular `gcc` compiler to compile **C** code into **RISC-V** instructions. We will compile a simple "bubble sort" program for running on a RISC-V compliant processor, such as the one implemented on our SoC.

4.1 Included Folders and Files

For the purpose of creating our binaries, we have provided a new folder in our workspace called "`apps`". Inside the apps folder, you will find four directories:

- `bubblesort` : An application example folder running bubble sort program
- `sw_utils` : Software utilities and scripts for compilation, linking, translation etc.
- `libs` : Some pre-compiled libraries to be linked with the code for proper booting.
- `ref` : Some more pre-made objects and link maps needed for the compilation utility.

4.2 The bubble-sort program example

Please look at `apps/bubblesort/bsort.c`

This simple program sorts a vector of **integers** located in a static pre-defined memory address, the program is executed upon hardware reset and upon sorting completion it writes the done_flag static variable also located at a fixed address.

Inside our **RISC-V** module (lp_riscv.v) we have included a design for capturing the done_flag access event and asserting the done_flag signal that is connected to PAD_DONE_FLAG output I/O in lp_riscv_top. This pin will be sensed by the testbench in order to terminate the simulation.

4.3 Compiling the program

The **RISC-V** architecture is fully supported by the most popular compiler in the world – the open-source **GCC** compiler. We have provided a script for using **GCC** to compile a **C** file to **RISC-V** binaries at apps/sw_utils/comp_app_local.sh. You can take a look at the script, and then run it from the apps/bubblesort/ directory:

```
%> tsmc65
%> cd <project>/apps/bubblesort
%> ../sw_utils/comp_app_local.sh bsort
```

This compiles the program links it with some boot required code and generates a memory image in a readable text format named l2_stim.slm which is also automatically copied to your testbench path (sourcecode/tb/slm_files/) in order to be loaded by the testbench.

5 Running a Cycle-Accurate Test Bench

At this stage of the process, we have the RTL of a full **SoC**, including a processor and memory, and we have binaries of a simple program, compiled for the architecture of our processor. We can now run a **Verilog** test-bench to evaluate functionality of our RTL.

Note that I wrote "evaluate functionality" and not "verify functionality", since a single running program is far from full verification of a hardware block!

5.1 The Test Bench

A testbench for executing our compiled code is provided at sourcecode/tb/tb_lp_riscv.v

The testbench performs the following:

1. Drives some constant values to some configuration or unused inputs.
2. Instantiates the **DUT** (Device Under Test) which is the implemented **RISC-V** top **SoC**.
3. Loads the compiled program code image into the instruction memory using mem_preload() task defined at sourcecode/tb/tb_lp_riscv_mem_pkg.
4. Generates, displays and loads random integer values to the data memory at the program vector defined address (see note below).
5. Toggles the clock and performs the reset sequence.
6. Upon indicating the done_flag pin assertion, displays the sorted vector and finishes the simulation.

Note on memory load:

In favor of project simplicity, the testbench artificially directly *pre-loads* the memories at the simulation environment. This, obviously, is not possible in real-life and does not represent a true memory external access sequence, which would require some dedicated hardware interface and a protocol to load the memory from an external device, such as a UART or SPI interfaces.

Also notice that the testbench sources also have **gate-level** definitions, if the `GATE_LEVEL` macro is defined (`xrun -define GATE_LEVEL`). This applies minor modifications due to the hierarchical position and naming differences of the memories since they are affected by synthesis.

5.2 Pre-synthesis RTL simulation:

To execute the testbench, we have provided settings for **Xcelium** to run either *RTL, gate-level,* or *timing backannotated* simulation under the `scripts` folder. These settings are called `xrun_options` with the extensions `rtl`/`glv`/`backannotation`. You should call Xcelium with this file as the `-f` argument:

```
%> cd <project>/workspace

%> xrun -f ../scripts/xrun_options.rtl
```

If all is ok, it should display the following:

```
pre-sorted Vector

mem[h00000800] = 1234567890
mem[h00000801] = 1537947638
mem[h00000802] =  169828691
mem[h00000803] = 3457865947
mem[h00000804] = 4219885877
mem[h00000805] = 1347386847
mem[h00000806] = 2423626847
mem[h00000807] = 4233660215
mem[h00000808] = 3536671972
mem[h00000809] = 1250551251

post-sorted Vector

mem[h00000800] =  169828691
mem[h00000801] = 1234567890
mem[h00000802] = 1250551251
mem[h00000803] = 1347386847
mem[h00000804] = 1537947638
mem[h00000805] = 2423626847
mem[h00000806] = 3457865947
mem[h00000807] = 3536671972
mem[h00000808] = 4219885877
mem[h00000809] = 4233660215
```

Please verify that the post-sorted vector is nicely sorted.

The simulation script also generates **SimVision** waves in the file `workspace/waves.shm` in case they are needed for debug.

6 Stage 1 Homework Instructions

Okay, so now for the bottom line – ***what are you supposed to do***?

6.1 Exercise Parameters

Let us define the 9-digit ID number of each student as ABCDEFGHI. With this definition, find the value of *MY_N* in the table below:

H	***MY_N***
0	30
1	12
2	14
3	16
4	18
5	20
6	22
7	24
8	26
9	28

6.2 Clone the repo and create a branch

1. Clone the git repository at: https://gitlab.com/dvd24/dvd24_hw/hw6.git

```
git clone https://gitlab.com/dvd24/dvd24_hw/hw6.git
cd hw6
```

2. Checkout specific version and fork with your own branch:

```
git checkout v1.0 -b <ID>
./setup.sh
```

6.3 Set Your Parameters in the Source files

1. Open the files `apps/bubblesort/bsort.c` and `sourcecode/tb/tb_lp_riscv.v` and:
   a. Replace the definition of `MY_ID` with your ID number (keep the `*10`!).
   b. Replace the definition of `MY_N` with your MY_N parameter from the table above.
2. Run the bubble sort program, according to Section 4.3, above to create your binaries.
3. Run an RTL simulation, according to Section 5.2, above, and check the simulation output to ensure that the numbers are sorted.

6.4 Report where your ID ended up

Did you notice the strange message at the end of the simulation printout?

```
Look/, Ma! The numbers are sorted!
And my ID is at index number:  99
```

Was your ID at index 99? I don't think so! (considering the largest array should have had 30 entries…). Gotta fix this!

1. Open `bsort.c` and `tb_lp_riscv.v` again.
2. Find out why you get this strange number printed out.
3. Add a few lines to `bsort.c` to make your simulation print out the correct index (i.e., where your ID number is really sorted to).

Note that we have made this super easy for you. Use the `defines` that we set up for you!

4. Add a few lines to `tb_lp_riscv.v` to print out an output file called "`reports/MY_SORTED_LIST.txt`", that includes:
   a. First line: MY_ID
   b. Second line: MY_N
   c. After that, the sorted list

For example:

```
MY_ID: 123456789
MY_N: 10
mem[h00000800] =  169828691
mem[h00000801] = 1234567890
mem[h00000802] = 1250551251
mem[h00000803] = 1347386847
mem[h00000804] = 1537947638
mem[h00000805] = 2423626847
mem[h00000806] = 3457865947
mem[h00000807] = 3536671972
mem[h00000808] = 4219885877
mem[h00000809] = 4233660215
```

6.5 Submission

Please add, commit and push your files that should include:

1) Modified `bsort.c`
2) Modified `tb_lp_riscv.v`
3) `MY_SORTED_LIST.txt`

7 Further background

Beyond exercising a full flow of a simple design, this project also provides some taste of very basic hardware-software co-design. Much of today's chip designs integrate some embedded processor. Typically, less performance-critical functionality is assigned to software tasks executed by the embedded processor, while the more performance-critical functionality is implemented by some accelerating hardware. The processor and the hardware communicate in many methods, one simple way is by **memory mapped addressable objects** as applied in the bubble-sort simple '*done-flag*' example.

For a slightly more complex example, one could just think of integrating a simple **FSM** that does string matching into such an SoC as an accelerator, where the software code will only need to write arguments and read the result through some predefined memory mapped interface with the accelerator. This is likely to save many string match calculation cycles compared to processor execution, while also freeing the processor for other tasks.

Special thanks to Yehuda (Udi) Kra, who put together this exercise and is our own personal “RISC-V Guru”!