RiSC-16 Sequential Implementation in ASIC and FPGA Domains

Yavuz Selim Tozlu
tozlu16@itu.edu.tr

July 18, 2020

Contents

Preface

RiSC-16 is a 16-bit **Ri**diculously **S**imple **C**omputer developed for pedagogical purposes. It is an excellent starting point for students willing to learn more about computer architecture and implementation.

In this project, I present both an ASIC and an FPGA implementation of this computer. The implementations are done in Verilog HDL. In ASIC domain, Cadence® Genus™ synthesis tool, along with Xcelium™ simulator are used. ASIC synthesis is done using TSMC's 90nm cell library. In FPGA domain, Xilinx Vivado® is used.

In the project directory, fully synthesizable Verilog sources are presented for both platforms.

Due to how primitive the computer is, the program to be ran on it must be hardcoded in its memory. In this project, a multiplication algorithm is loaded into memory as a testbench. The assembly code for the algorithm can be found in the repository.

The end result is synthesizable Verilog code for ASIC, and implementable code for FPGA.

For more information about the RiSC-16 architecture, check out the link below,

`https://user.eng.umd.edu/~blj/RiSC/`

Chapter 1

ASIC Implementation

1.1 Components

The top module(cpu core) consists of 6 submodules: ALU, Program Counter (PC), Data Memory, Instruction Memory, Register File and Control Unit.

1.1.1 ALU

The ALU has 4 functions; `add`, `nand`, `load upper immediate` and `pass`. It has three inputs, two of which are the operands and the third one determines the function. It has two outputs, one for the result of the arithmetic, and one for the equivalence check for branch instruction.

It is a fully combinational circuit, no latches or flip-flops are used.

1.1.2 Program Counter

Program counter generates the instruction address for the instruction memory. It increments the address value at the end of each instruction execution. However, as the computer is not pipelined, the program counter has to wait until the current instruction is done executing before incrementing the address value. Program counter knows how many cycles it needs to wait by looking at the `wait_cycle` signal from the control unit.

It also has a reset signal input which resets the address value to zero. It is a clocked, sequential circuit.

1.1.3 Data Memory

Data memory is the memory on which load and store operations are performed. All data is stored here. The size of the data memory is quite arbitrary. In this project, it is 64 halfwords (64×16).

Reads and writes are done on the rising edge of clock. Write enable signal determines which operation will be performed. Also, reset signal initializes the data memory values.

In ASIC, memories are synthesized using D-Flip-Flops.

1.1.4 Instruction Memory

Instruction memory is where the program is loaded. It is basically a ROM, and it is programmed within its source code. This is a poor way of loading a program, but given the simplicity of the computer, it makes no sense to add external hardware for programming. Also, in FPGA domain, there exists a proper way of loading memories which is discussed in chapter two.

Reset signal is utilized to load the instructions into the memory. Reads are done on the rising edge of clock. The instruction memory is also of size 64×16.

1.1.5 Register File

Register File is a two port read, one port write memory. It houses a total of eight 16-bit registers. The operation is determined based on the write enable signal.

In advanced processors, the register file often writes on a rising edge and reads on a falling edge, or vice versa. In this processor, reads and writes are done on the same clock edge which is a rising edge.

1.1.6 Control Unit

Control unit is the module which controls the muxes, ALU, write enables and the wait cycle of program counter. To do this, it looks at the operation field of the instruction, then decodes is accordingly. It also makes the branch decision.

The control unit is implemented as a 3-to-8 decoder, thus it is a fully combinational circuit.

1.1.7 Top Module

Top module is the place where all the submodules come together to form the processor. This module also contains the muxes and other trivial blocks that were not worth creating a separate module.

It has two inputs: clock and reset. Reset signal sets the program counter to the beginning of the instruction memory, and initializes the memories.

All the internal wiring is done in the top module.

1.2 Memory Organization

There are two memories which are physically separate in RiSC-16: Instruction Memory and Data Memory.

- Instruction Memory: Contains only the instructions and no data.
- Data Memory: Contains only data.

A RiSC-16 assembly code merely consists of data and instructions. Thus, the organization above is valid.

Loading the memories properly is vital to achieve expected outcome from a program. In RiSC-16 assembly, there are two pseudo-instructions that initialize data:

- `.fill`
- `.space`

Data generated using these commands must be loaded into Data Memory. Everything else -which are instructions- should be loaded into the Instruction Memory. The address of the data or instruction is determined by its location in the code. Figure below depicts a simple program and how it should be loaded in the memory.

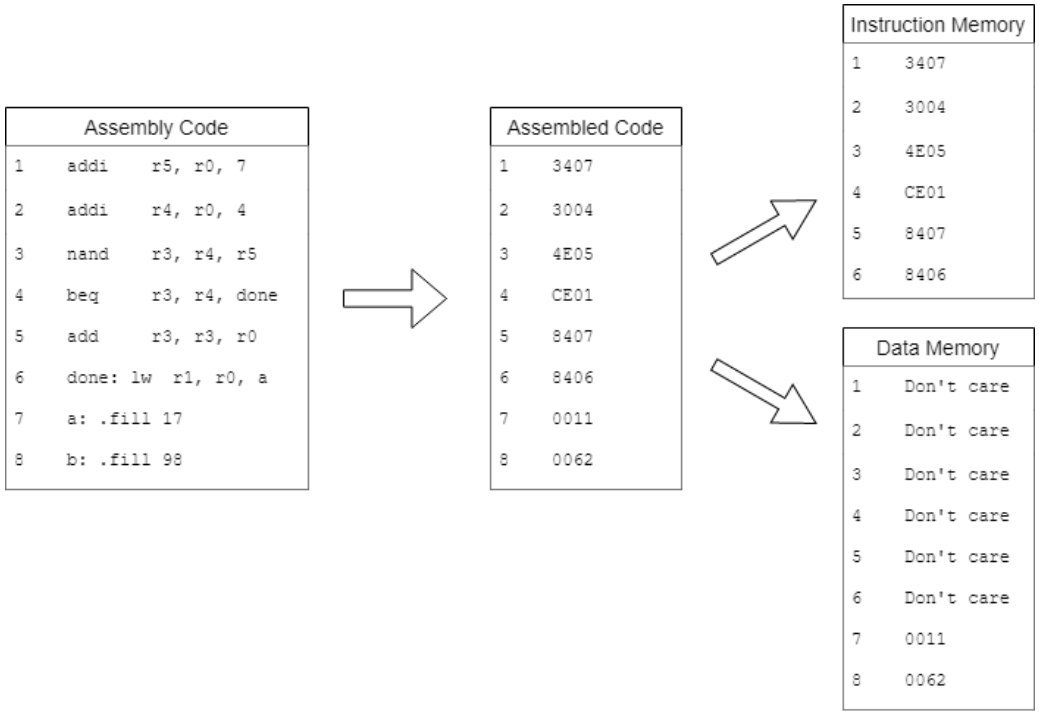

Figure 1.1: Memory mapping of a simple RiSC-16 program.

As mentioned earlier, the memory contents are defined within the source files. It is the developer's responsibility to load a given program properly. Failure to comply will result in ambiguous, incorrect results.

1.3 Testbench

Memories are initialized when the reset signal is asserted. Therefore, at the beginning of the processor's operation, it needs to be reset. Also, a clock with a frequency of 10MHz is applied. These signals are defined in the testbench file.

1.4 Synthesis

1.4.1 Synthesis Process

The synthesis process was a bit challenging as the synthesis tool has a tough learning curve, and has hundreds of commands from which the designer has to find the appropriate ones for his task. It is an iterative process of reading logs and re-synthesizing until the expected behavior from the output is achieved.

1.4.2 Scripting

A proper logic synthesis process outputs a design which meets a given timing constraint and a load, plus correct functionality. In this project, the synthesis tool is provided with a script which contains the constraints and executable commands.

The scripts are not very complicated. In fact, all the different modules have almost identical synthesis scripts. Following is a rough template of the scripts,

- Define cell libraries
- Read and elaborate Verilog files
- Define clocks
- Define delays, loads and driving cells
- Set synthesis effort
- Perform synthesis
- Write outputs (SDF, area, timing, power)

The reader can view the full script that is used to synthesize the top module in the repository.

1.5 Simulation Results

As mentioned in the Preface, a simple multiplication algorithm is used to verify functionality of the implementation. The opcodes are hard-wired into appropriate memory regions during synthesis. For this document, three different multiplication cases are simulated and results are shown in the figures below. Note that addresses 24 and 25 of data memory hold multiplicand and multiplier respectively. Address 0 holds the result. Only relevant signals are shown in the waveforms.

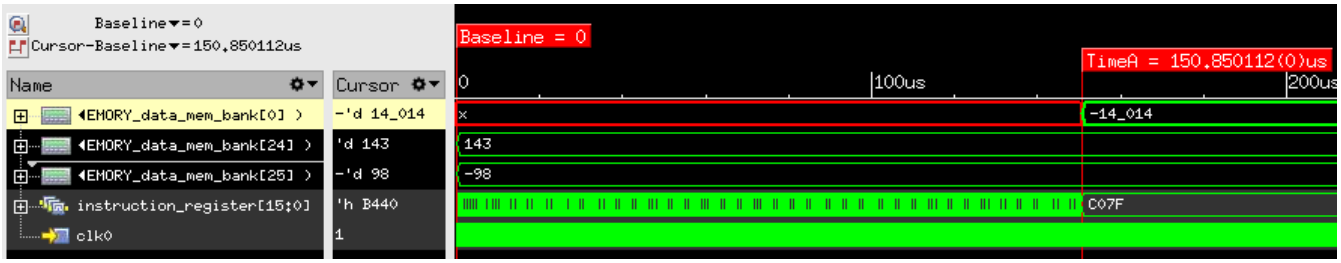

Figure 1.2: Multiplication of $r1 = 143$ by $r2 = -98$. Result is $-14,014$ which is stored in data memory.

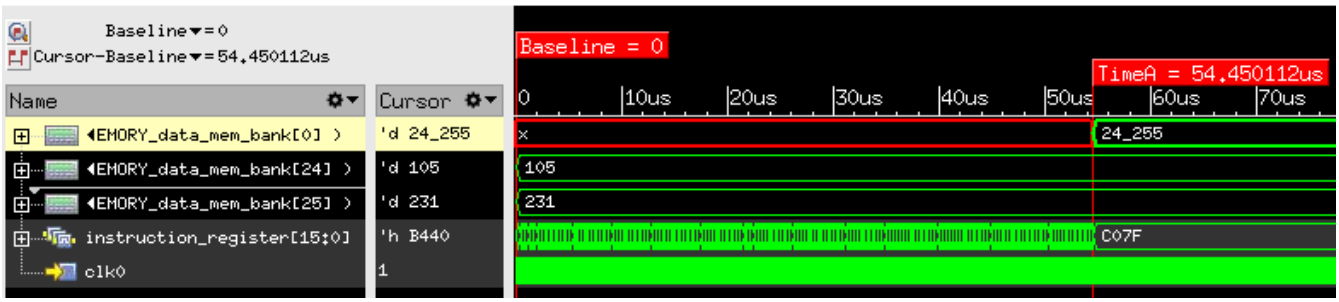

Figure 1.3: Multiplication of 105 by 231. Result is $24,255$ which is stored in data memory.

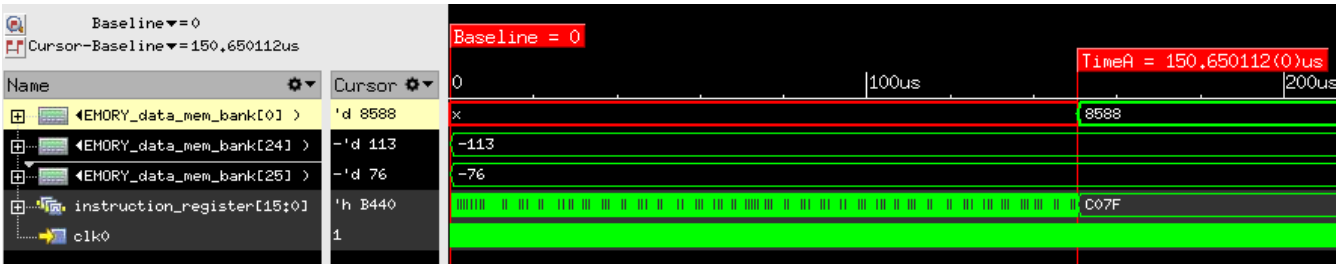

Figure 1.4: Multiplication of -113 by -76. Result is 8588 which is stored in data memory.

In the figures, "TimeA" Marker marks instance where the result is stored into memory. Prior to that, as the memory is not initialized, the value is 'X'.

Of course, these three simulations do not guarantee that the design is functional thoroughly. In fact, functional verification of a microprocessor design is no trivial task. However, given the simplicity of the microprocessor in our case, it is sufficient to perform only a few simulations as a proof of concept.

Chapter 2

FPGA Implementation

2.1 Migrating from ASIC to FPGA

FPGAs are excellent devices to test digital circuits. The microprocessor in this project can be implemented and tested on an FPGA with little effort. There are two main differences between ASIC and FPGA implementations as pointed out below,

- Memory definitions and initialization
- Monitoring the result

Ideally in FPGAs, the Instruction Memory and the Data Memory should be mapped to **Block RAMs**. Block RAMs are memory blocks provided intrinsically within FPGA chips. The catch here is that the memory source codes should comply with Block RAM definition guidelines in order for the synthesizer to map them.

In ASIC, the memory contents were still viewable post-synthesis. In FPGA, this is not the case. Therefore, different measures have to be taken. For this purpose, firstly the assembly code is slightly modified. It now constantly stores and loads the result into and from the data memory. Also, in hardware, the output of the data memory is defined as an output port of the top module. This way, it is possible to view the result of multiplication.

Everything else remains exactly the same as in ASIC implementation.

Both the assembly code and memory contents can be found in the repository. Data memory contents can be changed to multiply any two numbers.

The target FPGA chip is `xc7s50csga324-1`, featured on the Arty-S7 board.

2.1.1 Memory Initialization

In Xilinx FPGA chips, Block RAMs can be loaded with initial values very easily. This task is accomplished via the `"initial $readmemh"` Verilog command. This command is utilized in the memory source files. It requires two parameters: memory content file and its directory. Memory content files are plain text files with `".mem"` file extension. Memory content files for this project can be found in the repository. For instance, the Data Memory contents can be changed to multiply any two numbers.

2.2 Simulation Results

FPGA simulations are done in Xilinx Vivado®. All simulations are post-implementation timing simulations. Multiplications are identical to those in ASIC. Following figures show the results,

Figure 2.1: Multiplication of 143 by -98. Result is $-14,014$.

Figure 2.2: Multiplication of 105 by 231. Result is $24,255$.

Figure 2.3: Multiplication of -113 by -76. Result is 8588.

One important remark is that the result value is not constant, it keeps changing values. This is because once the algorithm ends, the processor goes into an infinite loop of loading from the data memory. Within that loop, output of the Data Memory changes with the clock signal. As the `result` signal is assigned to that output, it also changes constantly. Thus, this does not indicate that the processor is flawed, it is just a byproduct of the debug method.

Conclusion

In conclusion, it can be said that the implementation is successful. Simulation results add up and microprocessor performs as expected.

This project is a great learning opportunity especially for undergraduate students. Specifically, it gives an insight to processor design and implementation. One has to write and understand assembly, understand each basic block in a processor, learn simulation and synthesis tools, get familiar with Hardware Description Languages. Moreover, during the project, one sees concretely how an assembly code is executed within a processor.

Questions, suggestions can be sent to the email provided in the Preface section.