Design Project Report

Title: System C-based Design of RISC-V CPU

Authors: Vipul Sharma(B19CSE099) and Darsh Patel(B19CSE115)

Mentor's Names: Dr. Binod Kumar

Abstract: We are creating a high level design for CPU. The instruction set which we are using here is RISC V, a new and open source ISA. We are using SystemC on top of C++ as the language for design.

Introduction:

System C: It is a standard C/C++ based modeling platform. It is a fast system modeling containing multiple source components and it models for multiple abstraction levels, multiple users and multiple purposes.

Instruction Set Architecture(ISA):

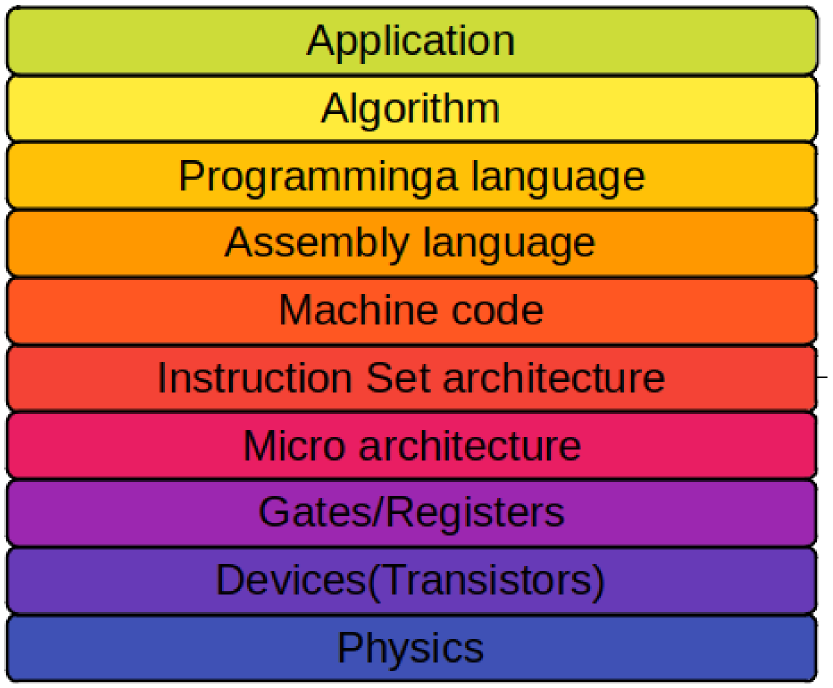


Image source: scimos.com/ wp-content/uploads/2020/07/ABSTRACTION-LAYERS

ISA is an abstract model of a computer.
Components of an instruction:

- Operation Code(Opcode)
- Source operand
- Destination operand

Reduced Instruction Set Computer-Five (RISC-V) is an ISA standard.

Problems with other ISAs like MIPS, ARM, X86: They are commercially protected with the help of patents. Preventing attempts to replicate computer systems in the real world.

Advantages of RISC-V: It is open for everyone, permitting anyone to construct compatible computers. It uses associated software. It is a real ISA which is highly suitable for direct native hardware implementation and not only simulation or binary translation.

RISC-V ISA also includes:
– A small base integer ISA, can be used by itself as a base for customized accelerators or for educational purposes.

Design Problem Formulation:

To create a high level CPU design based on RISC V ISA.

In order to come up with a basic CPU design following Instructions were required to implement:

1. Addition
2. Subtraction
3. Bitwise AND
4. Bitwise OR
5. Bitwise XOR
6. Load from memory
7. Store to Memory
8. Unconditional Jump
9. Conditional Jump

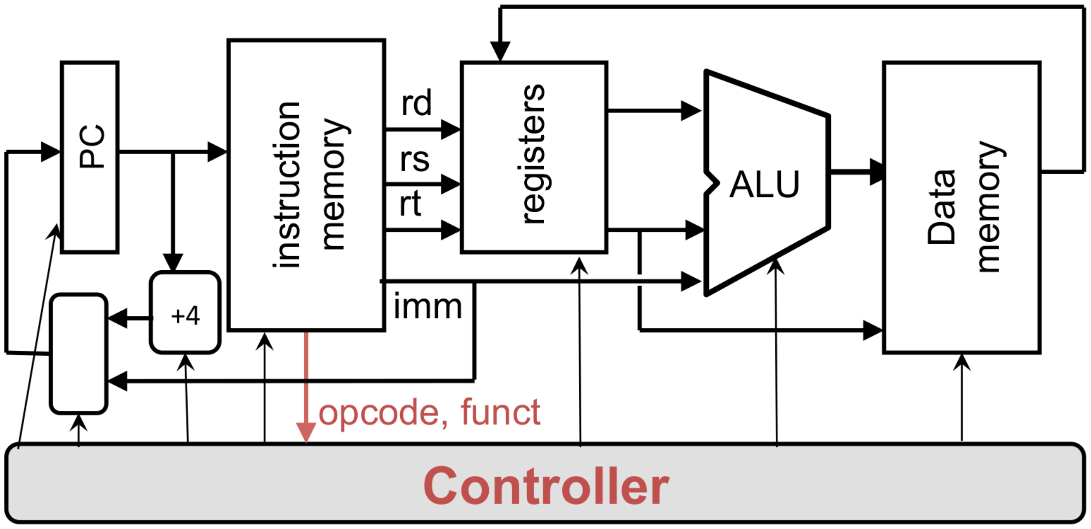
Possible Methods to solve the problem:

We can use other ISAs like MIPS.

Or we can implement it in Verilog. But we are using System C because it provides a faster way of making high level designs with help of C++.

Methodology adopted for the project:

A CPU as shown below has following components:



Img source: Lecture 08: RISC-V Single-Cycle, Implementation,CSE 564 Computer Architecture Summer 2017,Department of Computer Science and Engineering,Yonghong Yan, yan@oakland.edu

So during execution, our module will run on a testbench. The testbench will provide input to our module. The testbench will also provide a clock as an input to our module. In our SystemC constructor we have made our module sensitive to the positive edge of the clock. At each positive edge of the clock, our module will read the instruction pointed by the Program Counter and execute it (Single-Cycle implementation).

Program Counter (PC): It gives us the location of the next instruction in the memory. We have kept it as an SystemC integer variable. After reading the instructions pointed by it, we increment it.

Instruction Memory: It has the instructions which will be executed by the CPU. It has been implemented via a SystemC bit vector, from which we read in the sets of 32 bits, forming a single instruction.

After reading the instruction pointed by the Program Counter, we decode it with the help of conditional statements of C++, which simulates the Control Unit. We have also stored the Opcodes and other details, which help decode an instruction, hence simulating the Control memory.

Registers : Registers are fast memory devices belonging to a CPU. They are represented by an array of size 32, where each element of the array is a 32 bit integer.

ALU: It performs arithmetic and logic operations. We are using the operations provided by C++ to simulate the same, since this is a High level design project.

Data Memory: It has raw data .It has been implemented via a SystemC bit vector, from which we read and write in the sets of 32 bits.

Justification of the design choices:

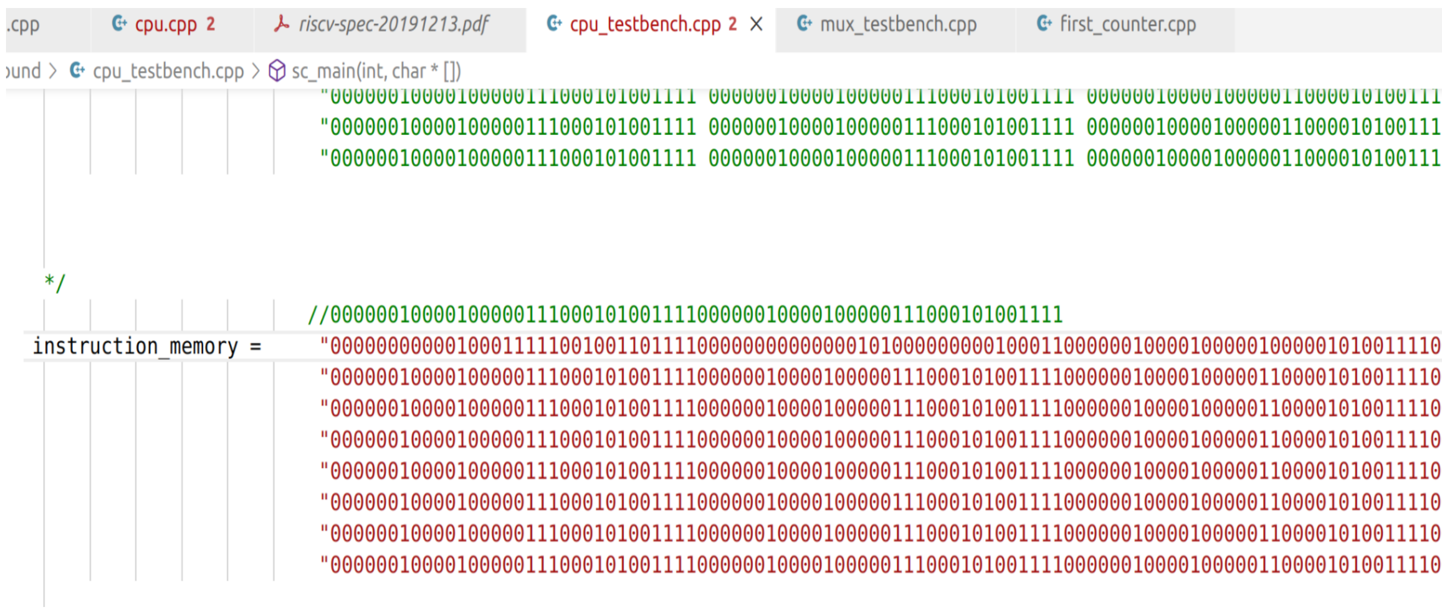
In our design we have kept the Program Counter as an inout port to our CPU module, allowing to increment Program Counter from the CPU module.

Instruction memory and data memory are also kept as input and inout port respectively, so that when we implement store instruction, we can write to data memory. They are initialized via the testbench. So we can also use a third file to take input at the beginning of simulation to get instruction and data memory and at the end of simulation to write the updates back permanently.

We have used operators(add, subtract, etc.) and conditional(if-else) provided by C++ since this is a high level design. But we can see that corresponding mapping to hardware devices is not difficult for low level design. (Add, subtract using an adder circuit, if-else using MUX, etc.)

Results and Analysis:

So here we have shown how we are using the instruction memory from the testbench.

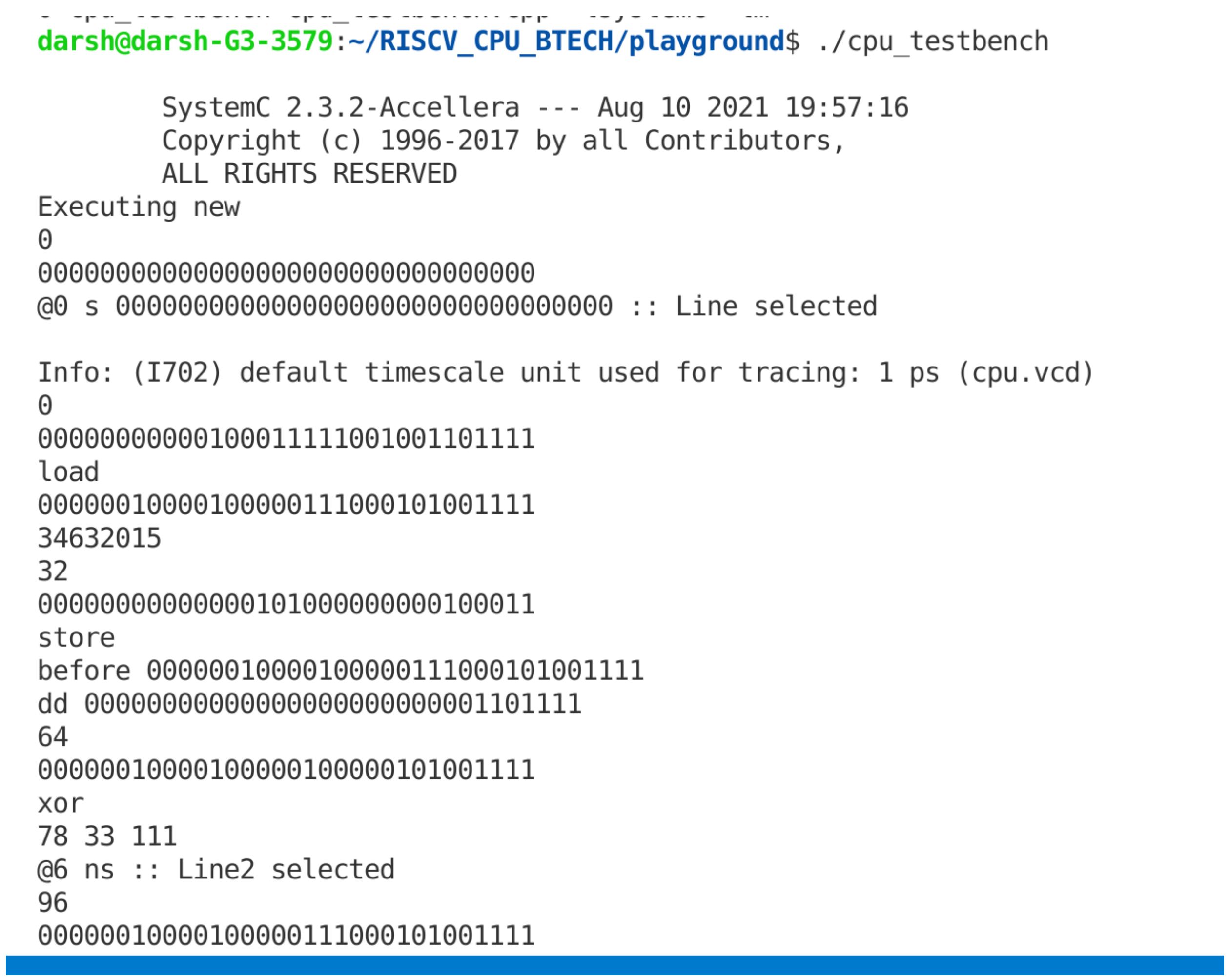


```
                    "00000010000100000111000101001111 00000010000100000111000101001111 0000001000010000011000010100111
                    "00000010000100000111000101001111 00000010000100000111000101001111 0000001000010000011000010100111
                    "00000010000100000111000101001111 00000010000100000111000101001111 0000001000010000011000010100111



 */
                   //0000001000010000011100010100111100000010000100000111000101001111
instruction_memory =   "000000000010001111100100110111100000000000000101000000000100011000000100001000001000001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
                    "0000001000010000011100010100111100000010000100000111000101001111000000100001000001100001010011110
```

Now corresponding to the instructions in the instruction memory, we can see that when we run our module, the instructions are getting decoded according to the defined standards and they are executed.



```
darsh@darsh-G3-3579:~/RISCV_CPU_BTECH/playground$ ./cpu_testbench

        SystemC 2.3.2-Accellera --- Aug 10 2021 19:57:16
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
Executing new
0
00000000000000000000000000000000
@0 s 00000000000000000000000000000000 :: Line selected

Info: (I702) default timescale unit used for tracing: 1 ps (cpu.vcd)
0
00000000000100011111001001101111
load
00000010000100000111000101001111
34632015
32
00000000000001010000000000100011
store
before 00000010000100000111000101001111
dd 00000000000000000000000001101111
64
00000010000100000100000101001111
xor
78 33 111
@6 ns :: Line2 selected
96
00000010000100000111000101001111
```

Conclusions:

So here we have created a high level design of a CPU following RISC-V ISA, which can run basic instructions. Future work can be to extend this to more complex instructions involving floating point numbers and pipelined instructions.

References:

1. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213, Editors: Andrew Waterman, Krste Asanovi´c SiFive Inc., CS Division, EECS Department, University of California, Berkeley

2. http://csl.snu.ac.kr/courses/4190.307/2020-1/riscv-user-isa.pdf

3. https://devopedia.org/risc-v-instruction-sets
4. https://forums.accellera.org/topic/5436-newbie-problem-writing-to-a-range-of-bits-of-sc_out-ports-declared-as-sc_lv/
5. https://scimos.com/wp-content/uploads/2020/07/ABSTRACTION-LAYERS-e1594137453801.png
6. https://howto.tech.blog/2016/11/27/installing-systemc-2-3-1/