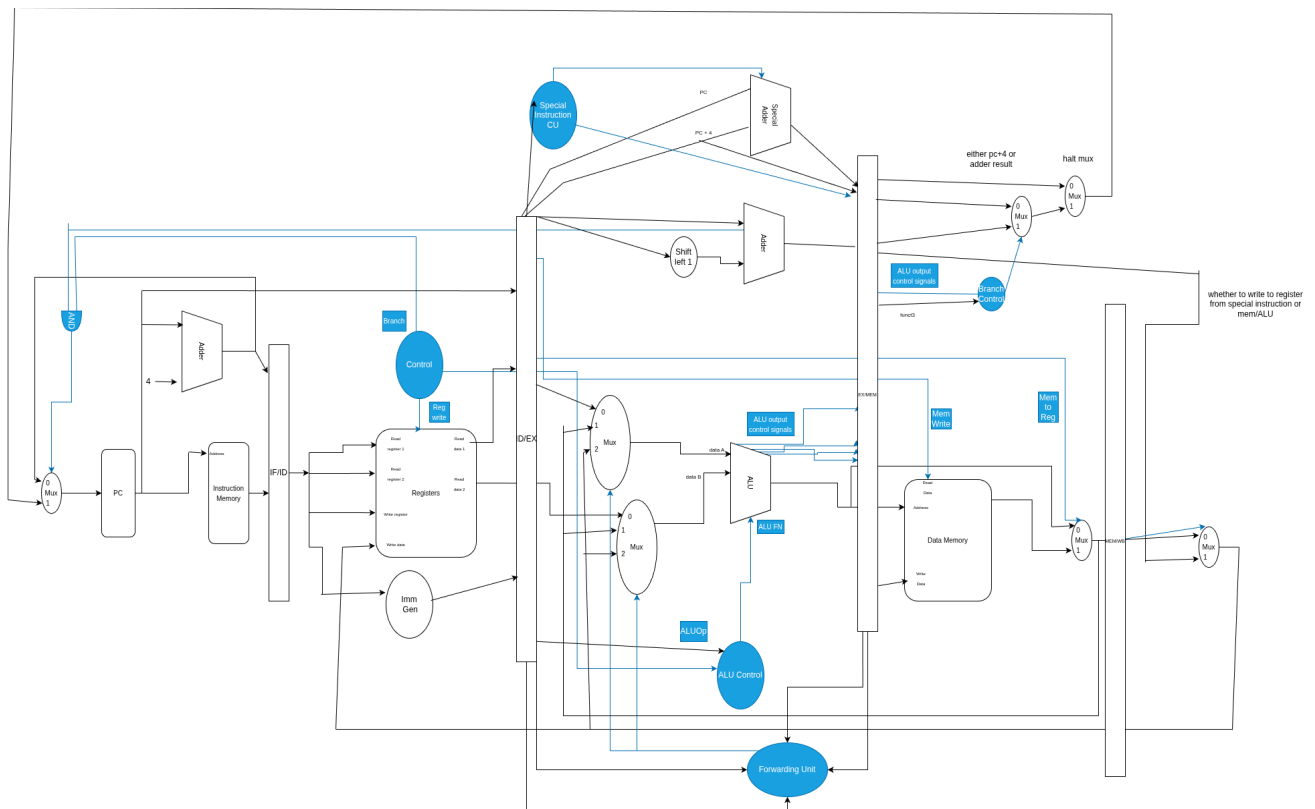


A RISC-V CPU Implementation in Verilog

Contains pipelining, forwarding, hazard detection and branch flushing

Mohammed Amer, Aly Youssef



Our Design

We designed our CPU in the usual five stage implementation with registers in between each stage. We prevented hazards by adding stalling in the control unit and added a forwarding unit. We supported instructions that depend on the Program Counter (PC) using a module that we called 'Special Instruction' as these instructions took the PC as an operand instead of two registers or a register and an immediate. Based on the opcode we figure out whether we want to write to registers the output of the Special Instruction part or the standard ALU/MEMORY part. According to our testing 42/42 instructions work correctly.

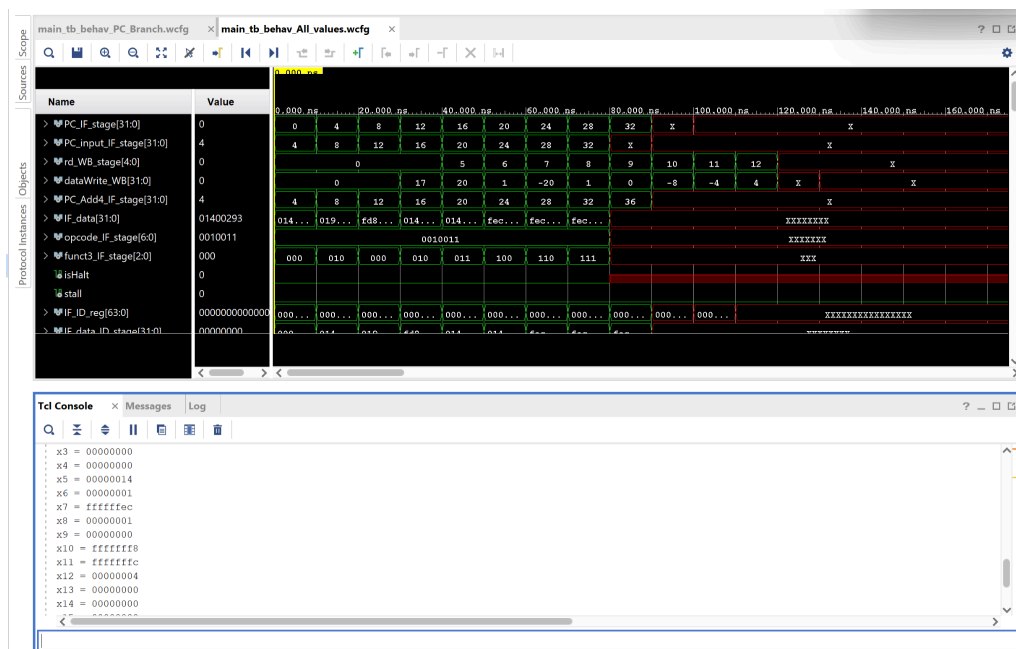
Issues we encountered

We found it somewhat difficult to figure out whether our CPU was stalling correctly and whether it was indeed loading/storing into memory. To solve this problem we wrote a task in the register file module and integrated it into the main testbench that outputs the values of all the registers at the end of program execution which we compared with the true results to make sure that our CPU was working properly.

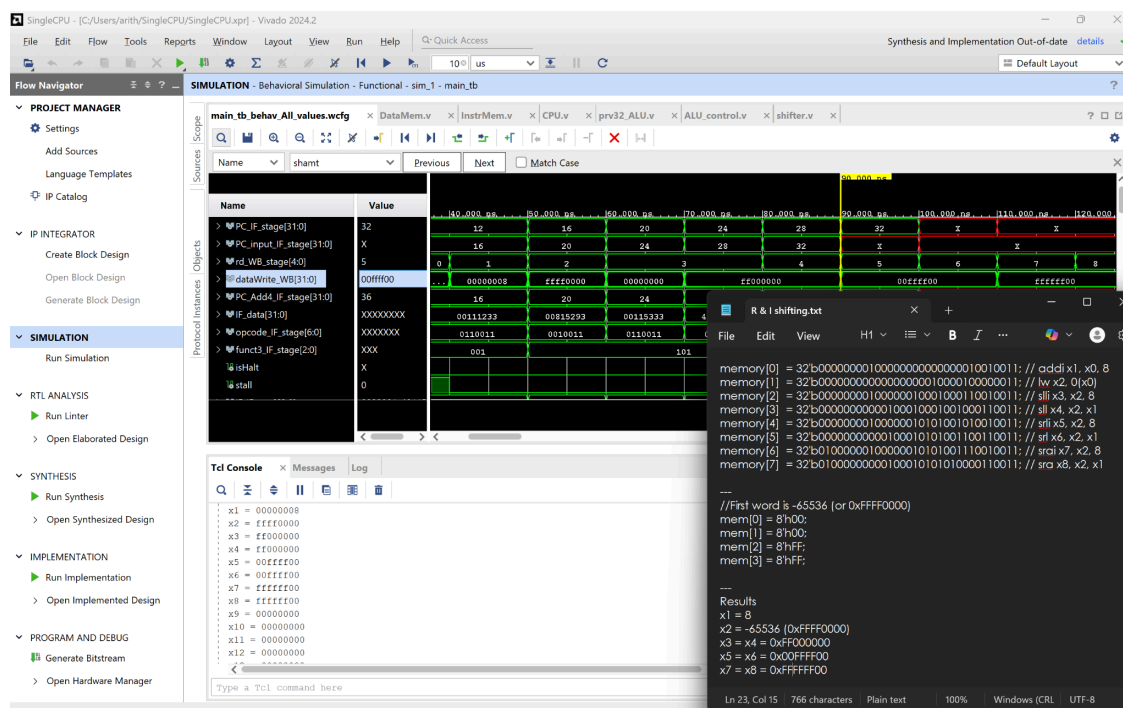
We also found it very difficult to combine the data and instruction memory as it would need extensive design changes and many stalls to prevent reading and writing at the same time which would corrupt the data.

An interesting point was deciding on how to structure the verilog file that calls all the different modules. We found that the simplest way was to make a wire for every signal that's needed at the specific stage so we had wires like 'rd_EX_stage', 'rd_MEM_stage' and 'rd_WB_stage' for the register destination at every stage. And every part of the verilog file is dedicated to a specific stage with the registers between each stage in the code which makes it very intuitive and similar to how the block diagram is structured.

I-type Instructions



Shifting Instructions



The screenshot displays the Vivado IDE interface during the synthesis and implementation of a Verilog design. The top window shows the Verilog code, which includes memory initialization, final results, and a testbench. The bottom window shows the 'Synthesis' and 'Implementation' tabs, with the 'Synthesis' tab selected, displaying the synthesized netlist and the 'Tcl Console'.

Verilog Code (Top Window):

```

memory[0] = 32'b00000000111100000000000010010011; // addi x1, x0, 15
memory[1] = 32'b00000000101000000000000010001001; // addi x2, x0, 20
memory[2] = 32'b00000000000100000100000010100001; // addi x3, x1, x2
memory[3] = 32'b010000000100000100000100010011; // sub x4, x1, x2
memory[4] = 32'b010000000001000100000101011001; // sub x5, x2, x1
memory[5] = 32'b000000000100000101000110011001; // slt x6, x1, x2
memory[6] = 32'b0000000000010001000110011011001; // slt x7, x4, x2
memory[7] = 32'b00000000000100010001101000011001; // sltu x8, x4, x2
memory[8] = 32'b0000000001000001100010101001001; // xor x9, x1, x2
memory[9] = 32'b0000000000010000011110101001001; // or x10, x1, x2
memory[10] = 32'b0000000000010000011100101101001; // or x11, x1, x2

Final results
x1 = 15
x2 = 20
x3 = 35
x4 = -5
x5 = 5
x6 = 1
x7 = 1
x8 = 0
x9 = 27 [1B]
x10 = 4
x11 = 31 [1F]

```

Synthesis and Implementation (Bottom Window):

- Synthesis:** The 'Synthesis' tab is selected, showing the synthesized netlist. The 'Tcl Console' displays the following commands and results:


```

x0 = 00000000
x1 = 0000000F
x2 = 00000014
x3 = 00000023
x4 = ffffffff
x5 = 00000005
x6 = 00000001
x7 = 00000001
x8 = 00000000
x9 = 0000001b
x10 = 00000004
x11 = 00000012

```
- Implementation:** The 'Implementation' tab is visible, showing the implementation details. The 'Tcl Console' displays the following commands and results:


```

x0 = 00000000
x1 = 0000000F
x2 = 00000014
x3 = 00000023
x4 = ffffffff
x5 = 00000005
x6 = 00000001
x7 = 00000001
x8 = 00000000
x9 = 0000001b
x10 = 00000004
x11 = 00000012

```

The screenshot displays the Vivado IDE interface during a logic simulation. The main window shows a logic analyzer with a table of signals and their values over time. The signals are organized into a hierarchy on the left, with the top-level signal being `PC_IF_stage[31:0]`. Below it, the `rs2_MEM_stage[31:0]` signal is shown with a value of `00000000`. The `memRead_MEM` and `memWrite_MEM` signals are also shown with values of `0`. The bottom section of the logic analyzer shows a list of 8-bit registers, each with a value of `00`. The right side of the window displays a waveform view, showing the timing of the signals. A Tcl Console window is open at the bottom, showing the commands used to set up the memory array.

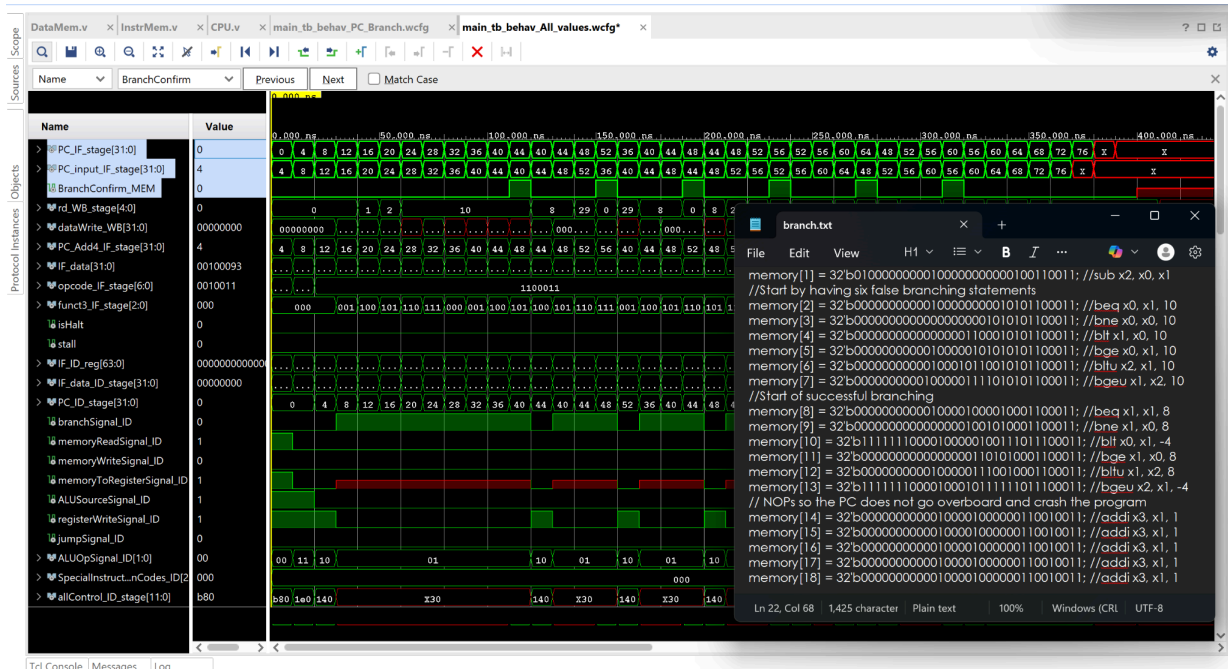
Signal	Value
<code>PC_IF_stage[31:0]</code>	<code>0</code>
<code>rs2_MEM_stage[31:0]</code>	<code>00000000</code>
<code>memRead_MEM</code>	<code>0</code>
<code>memWrite_MEM</code>	<code>0</code>
<code>[0][7:0]</code>	<code>ee</code>
<code>[1][7:0]</code>	<code>dd</code>
<code>[2][7:0]</code>	<code>cc</code>
<code>[3][7:0]</code>	<code>ff</code>
<code>[4][7:0]</code>	<code>00</code>
<code>[5][7:0]</code>	<code>00</code>
<code>[6][7:0]</code>	<code>00</code>
<code>[7][7:0]</code>	<code>00</code>
<code>[8][7:0]</code>	<code>00</code>
<code>[9][7:0]</code>	<code>00</code>
<code>[10][7:0]</code>	<code>00</code>

```

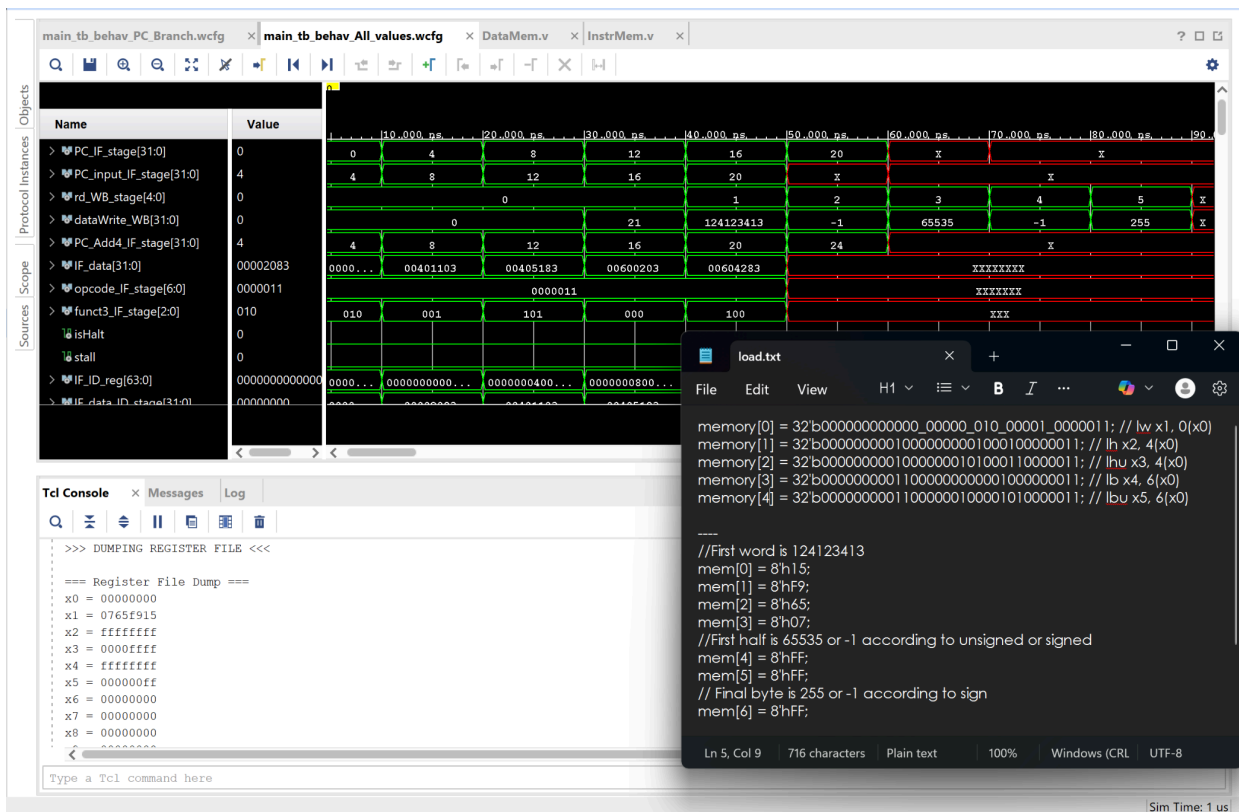
Tcl Console
x28 = 00000000
x29 = 00000000
x30 = 00000000

//First word is 0xFFCCDDEE
mem[0] = 8'hEE;
mem[1] = 8'hDD;
mem[2] = 8'hCC;
mem[3] = 8'hFF;
  
```

Branch Instructions



Load Instructions



Further Improvements

This CPU implementation doesn't unfortunately take advantage of the different types of Caches which might increase speed. It also doesn't have access to any DRAM or Secondary storage, it only relies on the predefined tiny Data memories and Instruction memories. It also doesn't offer a way to view or use the results of its computation in a useful way so it is unlikely to be used in real-world applications.

Another potential improvement is to implement this CPU on a PCB to allow it to run natively instead of emulating it on an FPGA, doing this would allow us to add many custom features like video output or input from devices like mice or keyboards.

CONCLUSION

This project was incredibly fun, interesting and an incredible learning experience for us. We learned the basics and somewhat intermediate parts of how CPU's work and how to design one. Our implementation is simple, intuitive and relatively easy to add further improvements to as we intended to go further with this project than we were able to. Thank you for leading us every step of the way until we wrote a functional central processing unit ourselves and we hope to use this knowledge to benefit others.