

Instruction Driven ALU

September 14, 2021

1 Instructions

This document outlines your Verilog take-home test instructions. There are three Sections in this document; **Instructions**, **Design of a small ALU**, and **Your Task**. Make sure to read them all!

1.1 Files

When you unpack your zip-file you should find the files listed in Table 1 in a folder called `alu/`

File name	Description
<code>alu.sv</code>	This is the file you need to modify
<code>alu_tb.sv</code>	Top level for the simulation
<code>Makefile</code>	Used to build and simulate with Icarus Verilog
<code>test.asm</code>	Assembly for the test program
<code>test.hex</code>	Assembled version of the test program
<code>test_reference.output</code>	Expected output from the test program
<code>secret.hex</code>	A randomly generated program
<code>spec.pdf</code>	This file

Table 1: Files

1.2 Running

We recommend using Icarus Verilog and GTKwave to compile, simulate, and debug your design. Most distributions based on Debian will have these tools ready be installed from their package managers. For example on a fresh install of Ubuntu 21.04¹ you can get the necessary tools by running:

```
sudo apt install iverilog gtkwave make
```

When these tools are installed, you can navigate to the just extracted folder and run the simulation of the test program by executing **make**. To run the secret program just execute **make secret**. Note that the simulation will run forever without any output before you implement your solution.

When running with the test program, i.e. **make**, your output will be compared against `test_reference.output`. If there are mismatches the return code will be non zero and you will see something like what is shown in Listing 1.

Running the simulation will capture the waves in either **test.fst** or **secret.fst**. With GTKwave installed you can view the waves with **gtkwave test.fst** or **gtkwave secret.fst**

Note, **iverilog** has support for most of Verilog and limited support for SystemVerilog. See Table 2 for known limitations and suggested work-arounds.

You can choose to complete the assignment using any other tools you are familiar with! However, we will only assist you with troubleshooting Icarus Verilog and/or GTKwave issues.

¹If you are currently not on Linux or your distribution does not have these tools, consider installing Ubuntu 21.04 on a local VM.

```

...
diff -u test.output test_reference.output
--- test.output 2021-09-13 16:56:41.391309086 -0400
+++ test_reference.output      2021-09-13 16:56:01.440457148 -0400
@@ -1,7 +1,7 @@
    0x048 (H)
    0x052 (R)
    0x054 (T)
-0x000 ()
+0xfff ()
    0xfff ()
    0x000 ()
    0xfc0 ()
make: *** [Makefile:11: test] Error 1
...

```

SV construct	Work-around
<code>always_comb</code>	<code>always@*</code> or <code>assign</code>
<code>always_ff@(posedge clk)</code>	<code>always@(posedge clk)</code>
unpacked array in <code>struct</code>	break up into <code>item_0</code> , <code>item_1</code> , etc.
<code>assign a = some_type_t'(b)</code>	casts are not needed, just do <code>assign a = b</code>
<code>s[N:0] = o[N-1:0] + p[N-1:0]</code>	<code>s[N:0] = {1'b0, o[N-1:0]} + {1'b0, p[N-1:0]}</code>

Table 2: Known **iverilog** SystemVerilog limitations

If you cannot get Icarus Verilog and/or GTKwave to work and/or you have no other tools at your disposal, please email hrthardwareintern@hudson-trading.com.

1.3 Submitting

Please submit your work by sending an email to hrthardwareintern@hudson-trading.com. Put your name in the **subject line** and attach the files listed in Table 3 as **plain text files**.

File name	Description
<code>alu.sv</code>	The file you made your changes to
<code>secret.output</code>	Generated when you ran make secret

Table 3: Files to submit

2 Design of a small ALU

Your task is to design a simple instruction driven ALU following the specifications in this document. The ALU is a single stage pipeline design, meaning that register values are read and written in the same cycle, thus there are no read before write issues. A sketch of the ALU system is shown in Figure 1.

The ALU should have four general purpose registers that are 12 bits wide. All operations are performed on 12 bits unsigned integers unless otherwise stated. If the result of an operations exceeds 12 bits, only the 12 least significant bits are stored back to the destination register.

There are two types of instructions **Arithmetic** and **Immediate** the instruction formats are specified in Table 4, where *Rd* is where the result is written, *Rx/Ry* are where the operand values are taken from, and *immediate* is a constant value carried with the instruction. The fill bits in the **Arithmetic** instructions should be ignored by hardware and will be set to 0 in the instruction word. For example $R3 = R1 + R2$ would result in the instruction word `0x7D8` and $R3[5:0] = 0x17$

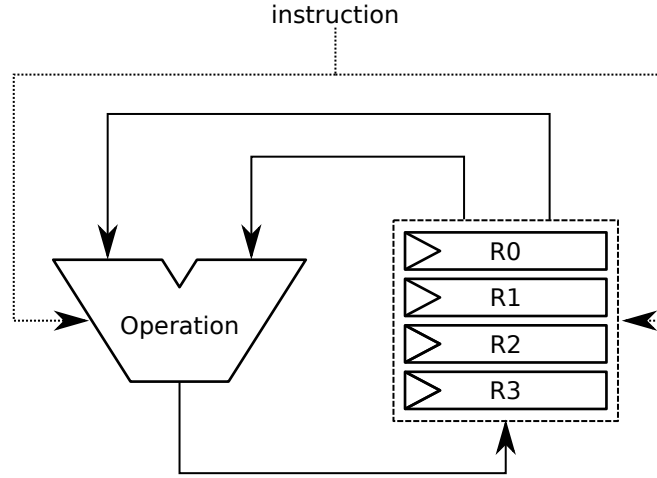


Figure 1: ALU sketch

Type	bits				
	11:8	7:6	5:4	3:2	1:0
Arithmetic	op-code	Rd	Rx	Ry	fill
Immediate	op-code	Rd	$immediate$ (bits 5:0)		

Table 4: Instruction format definition

in the instruction word 0xBD7. Table 5 list the operations that the ALU should be able to perform. Note that only ADD and ADDC should update the carry, it shall remain unchanged for any other instruction.

Operation	op-code	operation	description
OR	0x0	$Rd = Rx \vee Ry$	bitwise or
XOR	0x1	$Rd = Rx \oplus Ry$	bitwise xor
AND	0x2	$Rd = Rx \wedge Ry$	bitwise and
NOT	0x3	$Rd = \neg Rx$	bitwise not
LSHIFT	0x4	$Rd = Rx \ll 1$	logical left shift of Rx by one bit
RSHIFT	0x5	$Rd = Rx \gg 1$	logical right shift of Rx by one bit
ARSHIFT	0x6	$Rd = Rx \ggg 1$	arithmetic right shift of Rx by one bit
ADD	0x7	$\{c, Rd\} = Rx + Ry$	add and update carry
ADDC	0x8	$\{c, Rd\} = Rx + Ry + c$	add with and update carry
SUB	0x9	$Rd = Rx - Ry$	subtraction
LOADLO	0xA	$Rd[5 : 0] = immediate$	load lower 6 bits of Rd
LOADHI	0xB	$Rd[11 : 6] = immediate$	load upper 6 bits of Rd
OUT	0xC	$out_data = Rx$	Output value of Rx and also assert out_data_valid
HALT	0xD	$out_data = Rx$	Same actions as for OUT while also asserting $halt$

Table 5: ALU operations

3 Your Task

Implement your code in the skeleton file `alu.sv` where you see `// Your code here...`. You should not need to modify any other file.

Running **make** will run a small test program, please see **test.asm** for more details on what this program does.

When you think your ALU behaves correctly, run **make secret**. This will run the program found in `secret.hex`. This will also generate `secret.output` that you should submit together with your ALU implementation. We will compare your output with our reference implementation.

Also please answer the following questions in a comment in `alu.sv` with your submission.

1. There are no NOP (don't change any register values) or CLEAR (set a register to 0) instruction specified, how can we perform these operations with the instructions we have available to us?
2. What would have to change if we added a pipeline stage right after the ALU operations in hardware as shown in Figure 2, i.e. before a value is written back to *Rd*. Assuming, given the same program, the same output should be generated.

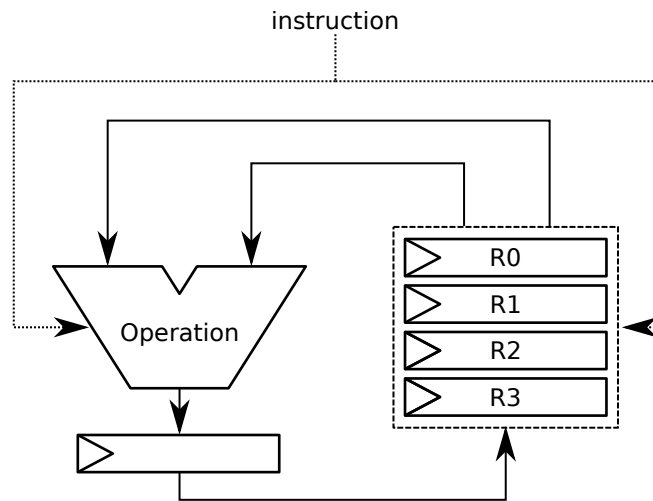


Figure 2: Pipelined ALU sketch