Jimmy Wu and Ian Daniher
CompArch Fall 2012
Lab 0b10

Introduction

We built an ALU that implements add, subtract, xor, set if less than, shift left, and shift right for 32 bit numbers. The ALU accepts two one-word (thirty-two bit) inputs and a five bit control line. All modules were implemented predominantly structural Verilog, with behavioral verilog utilized to simplify and clarify. The ALU itself is simply a control mux implemented with behavioral Verilog's case switching to connect the appropriate "out" from the function-specific port to the "out" register of the ALU.

We ran our simulation using Icarus Verilog. Wave output was visualized using gtkwave. Version control was performed using Git and Github. The repository can be viewed here: https://github.com/JimmyFW/ca-Lab2. We used Python to assist in the creation of some of the modules.

Module construction

A brief tour of the inside of each of the modules we constructed can be found below.

adder.v

adder.v implements two modules, Adder1Bit and Adder32Bit. The one-bit-wide full adder was created in a typical manner using two 'xor' gates, two 'and' gates, and one 'or' gate, each with two inputs and each having a 20u delay. The 32b adder accepts two 32b inputs and gives one 32b out and one 1b 'carryout.' It was created from 32 of the one-bit full adders with chained carry bits, with 'carryout' being the carry output of the last adder.

sub.v

sub.v works as follows: take the two's complement of one of the inputs, and add it to the other input. As a result, it is dependent on adder.v, which adds the inputs together, and sign.v, which is responsible for taking the two's complement.

xor

xor was created in alu.v using implicit structural verilog. The line "#20 out = a ^ b;" instantiates 32 parallel xors, each acting on one bit of a and one bit of b. This approach offers a very low delay and was extremely efficient to write.

slt.v

"set if less than" was implemented in slt.v using the previously created subtractor and the a right-shift. The output of the 'slt' module is the sign bit of the result of the expression 'a-b,' such

that if a was less than b, the result would be negative, and the sign bit would be one. A bit shift was used to select the last bit by shifting out the first 31b.

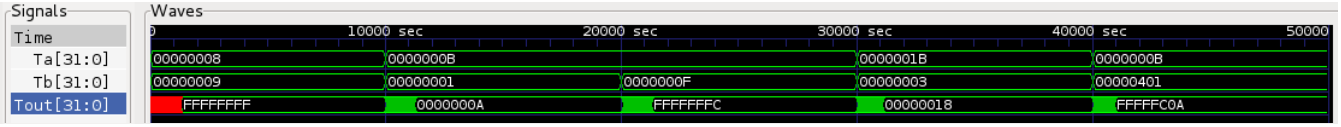
shiftLeft.v
shiftRight.v

shiftLeft and shiftRight are two modules with very similar construction. Originally, shifts were built using shift registers and were locally clocked with a 10u period. This implementation had an extremely concise behavioral implementation, but the synthesis left semantic ambiguity as to the implementation. It was recreated using a five-stage set of muxes, one “level” for each useful bit of inB. Each of the five muxes accepted one bit of input, either a zero or a one, corresponding to the nth bit of inB. If that bit is 1, the mux’s output is equivalent to wires connected from the zeroth bit of the input to the 1<<n’th bit of the output, such that each stage offsets by an amount directly proportional to the the power of two by which it’s controlled.

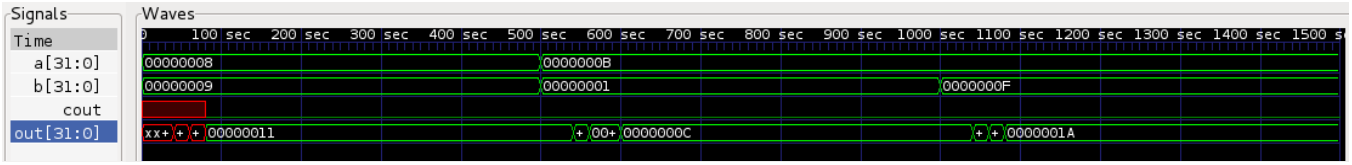
Timing Analysis

Note: u refers to a timing unit in Verilog, currently unspecified.

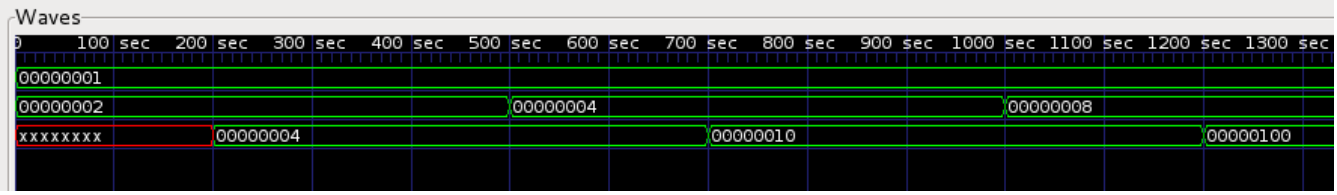
Subtract:
It takes a maximum of 1290u for an add instruction to stabilize. Our test file gives each instruction a wait time of 1500u to execute. The delay is dominated by the propagation of our one-bit adders.



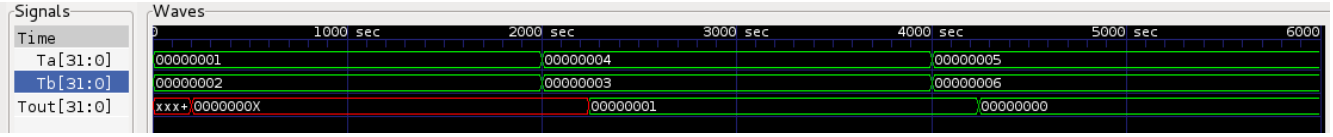
Add:
It takes a maximum of 1240u for an instruction to stabilize. Each of the one-bit adders depends upon the carryout of the adder before it, requiring a pseudo-sequential dataflow operation.



Shift:
(same for both left and right shifts)
It takes a maximum of 200u for our shifters to stabilize - each of the five muxes takes 40u to stabilize and is fed with the output of the previous mux.



Set if less than:
It takes a maximum of approximately 1500u for our SLT operation to stabilize, as it depends upon both a 31b left shift and the subtraction of two 32b numbers.



xor:
Our xor implementation is constructed from 32 parallel xor gates. It features a quasi-arbitrary propagation delay of 20u.

Design Justification

Throughout the course of our design process, we made a handful of very intentional decisions on the size/speed dichotomy. Our final arithmetic / logic unit would synthesize to a very small amount of silicon due to the extensive reuse, but display subpar performance, most notably due to the long maximum propagation delay of our adder, used in almost all of our functions. The implementation of our shifters was based around muxes to ensure clear structural functionality, at the cost of a large amount of silicon space. Should local clocking be feasible, a more refined solution might involve 32 chained shift registers, clocked by a high speed counter.

Conclusion

Our ALU is tiny. It’s not super-fast, but it’s pretty darn small. We reused a ton of stuff. If we wanted to make it better, we’d finish our multiply implementation (even slower) or, more likely, redo our adder to remove the huge performance hit caused by the carry-ripple with a carry-lookahead adder.