Joseph Blethen
Alex Karapetkov
02 - 26 - 2024
Team 4

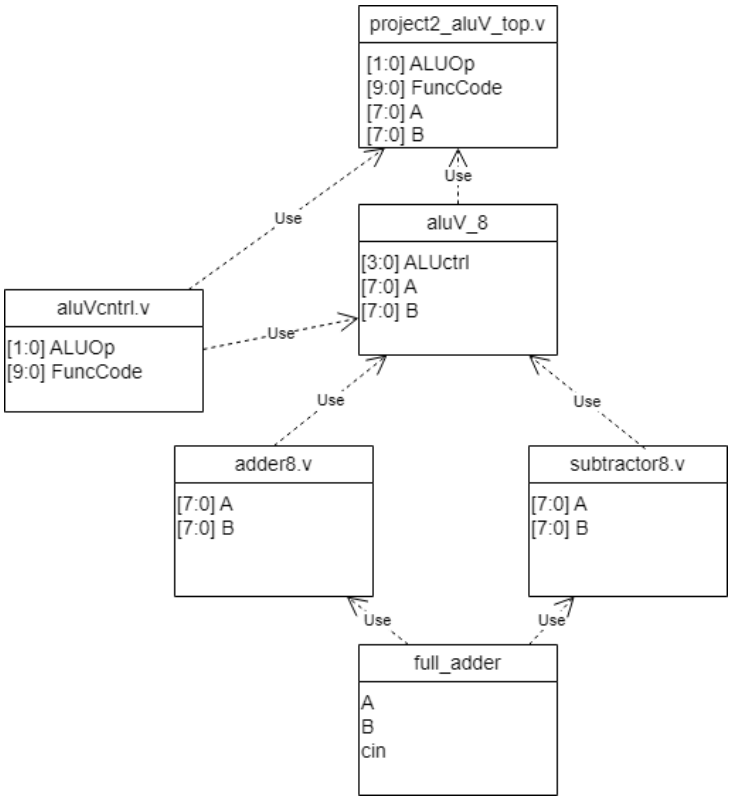
Project 2 - RISC V ALU

This work complies with the JMU Honor Code

The following design specification is an implementation of the arithmetic-logic unit found within the RISC-V architecture. Overall, the design was reduced to 8-bit operations for ease of programming and features only the AND, OR, add, subtract, set-less-than, and NOR operations. To select which one to use, the ALU uses an ALU controller that interprets two parts of an instruction. These instruction parts include the ALUOp which indicates the format of the instruction i.e., R-type, I-type, or U-type, etc., and the function-code which is the concatenated func7 and func3 instruction identifiers within the filtered instruction type. Our ALU design took this into account and safely ensured the instruction type matched the exact function code before committing to an operation. Although it was not part of the required specification, our design also accounts for the set less than instruction in both the 8-bit ALU and ALU control unit modules.

Hierarchy of Verilog code:

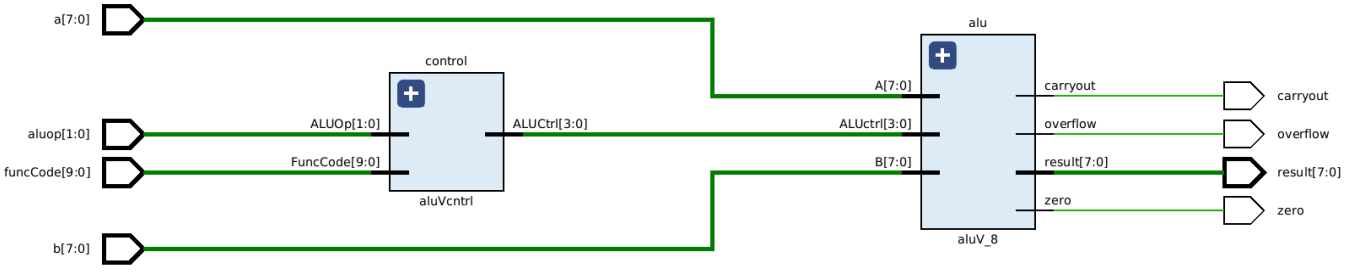
Figure 1. Block Diagram of Verilog Code Hierarchy



This figure conveys the structure and hierarchical relationships of the different Verilog modules created to implement the RISC-V ALU. At the bottom of the hierarchy is a one-bit full adder which adds two one-bit inputs along with a one-bit carry-in and outputs a one-bit sum and carry-out. The 8-bit adder on the next level of the hierarchy is a ripple carry adder consisting of eight full adders; it takes two 8-bit inputs and adds them in parallel where the carry-out of each full adder is connected to the carry-in of the next full adder. The 8-bit subtractor works in a similar fashion where it consists of eight full adders and the borrow-out of each full adder feeds into the borrow-in of the next full adder. The ALU control module takes in a 2-bit ALU opcode along with a 10-bit function code to interpret what operation the ALU should perform. At the next level of the hierarchy, the 8-bit ALU takes in the output of the ALU control unit, along with two 8-bit inputs to perform an operation on. If the ALU is tasked with performing addition or subtraction, it instantiates and utilizes the 8-bit adder or 8-bit subtractor to perform the operation and properly account for overflow. The final Verilog module at the top of the hierarchy, the ALU top file, is used to put everything together by instantiating both the ALU control module and 8-bit ALU; it feeds the respective inputs into both of those modules.

RISC-V ALU

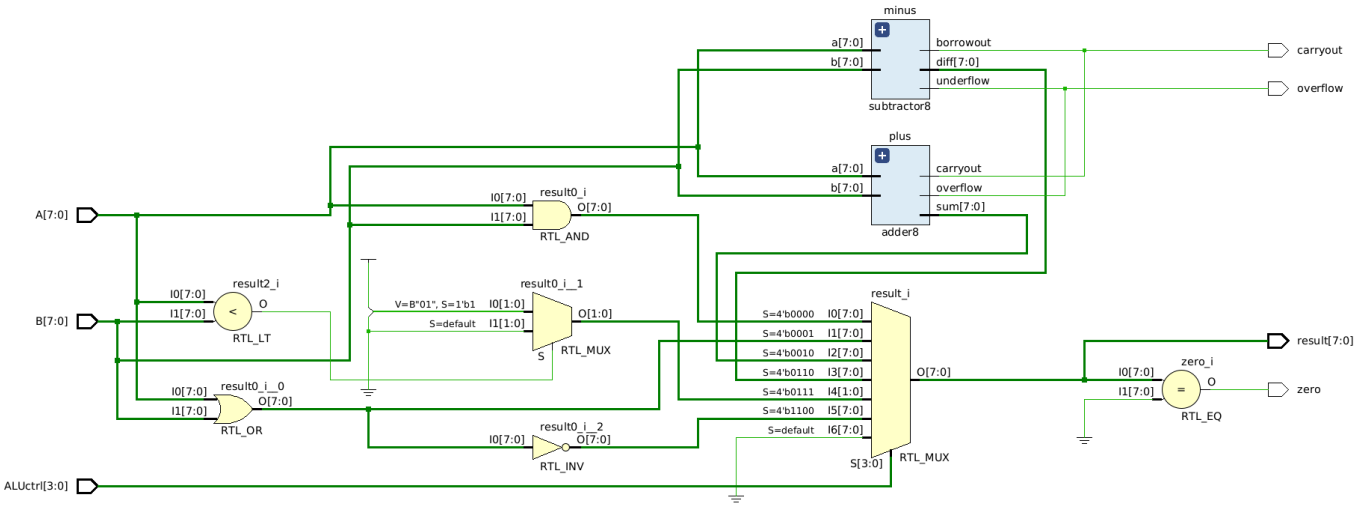
Figure 2. Schematic of RISC-V ALU



This module is a completed RISC-V ALU which consists of an ALU control unit and ALU. The ALU control unit outputs a 4-bit control signal based on its ALUOp and function code inputs; the control signal corresponds to what operation the ALU should perform and is taken along with 2 8-bit operands as input by the ALU. If the specified operation is addition or subtraction, the ALU outputs the result along with a carryout and indication of overflow if necessary. The zero output is used to signal whether two operands are equal or not.

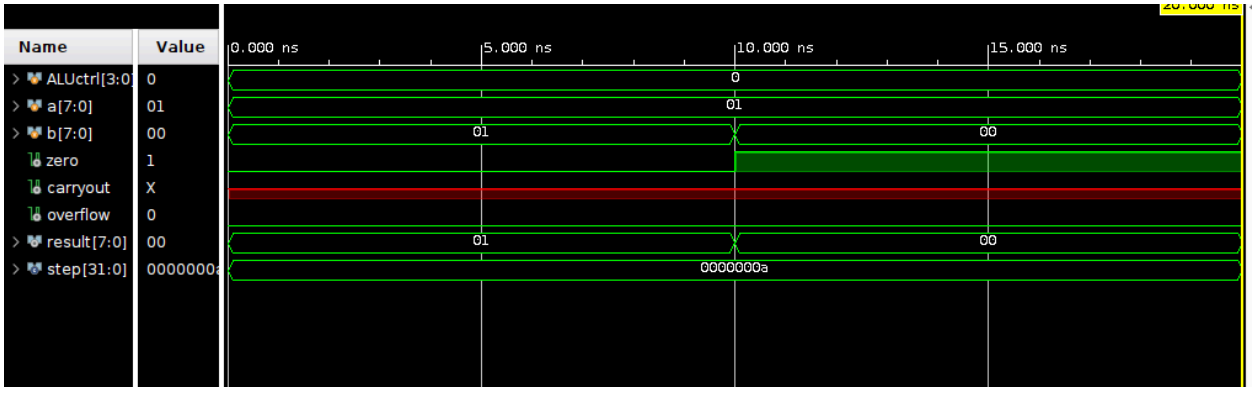
Component: 8-bit ALU

Figure 3. Schematic of 8-bit ALU



The schematic seen above is an arithmetic logic unit with AND, OR, add, subtract, set-if-less, and NOR operations. All operations aside from the add and subtract were implemented in behavioral Verilog. Its inputs are two 8-bit numbers and the 4-bit ALU controller bus which tells the ALU's multiplexer which operation to perform as a selection. Once it completes any single operation, the 8-bit resulting number will be delivered from the mux. If the resulting number is equal to zero, then the zero bit is activated. Naturally, only the add and subtract can have carryout and overflow bits. Lastly, of all the operations implemented, the select-if-less is the strangest as it connects the mux to another mux. This other multiplexor compares the given numbers and sends either a 1 or 0 to the result depending on if one is less than the other.

Figure 4. Timing of AND operation resulting in Zero

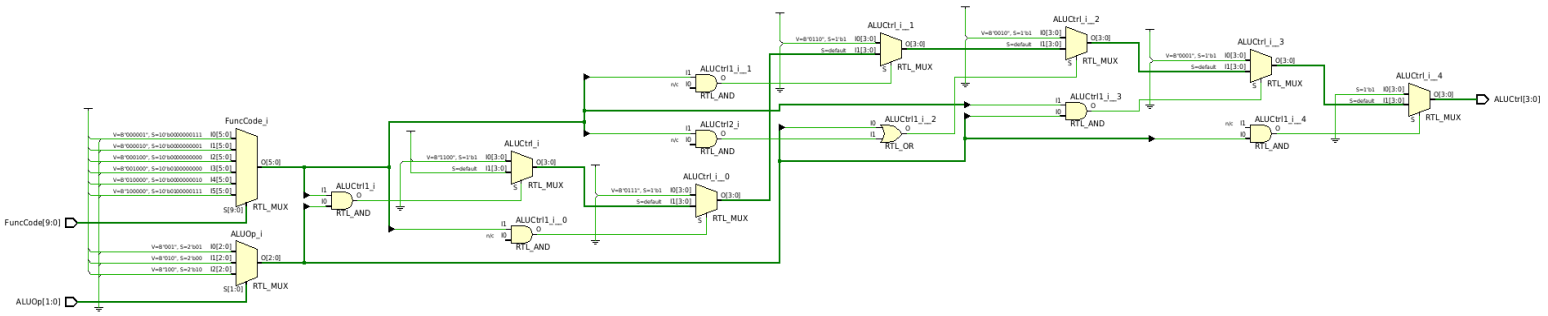


As a feature of the ALU's design, the AND operation is built directly into it through the behavioral bitwise operation. We thought it was unnecessary to separate the AND operation just to set the zero bit since the zero bit can be assigned by checking the result of all operations. In the diagram

above, we show this by keeping "A" at 1 and toggling between 1 and 0 for "B", which shows the proper "result" response. It behaves exactly as an AND operation should. Other larger numbers were tested in the timing diagram of the Top module.

Component: ALU Control Unit

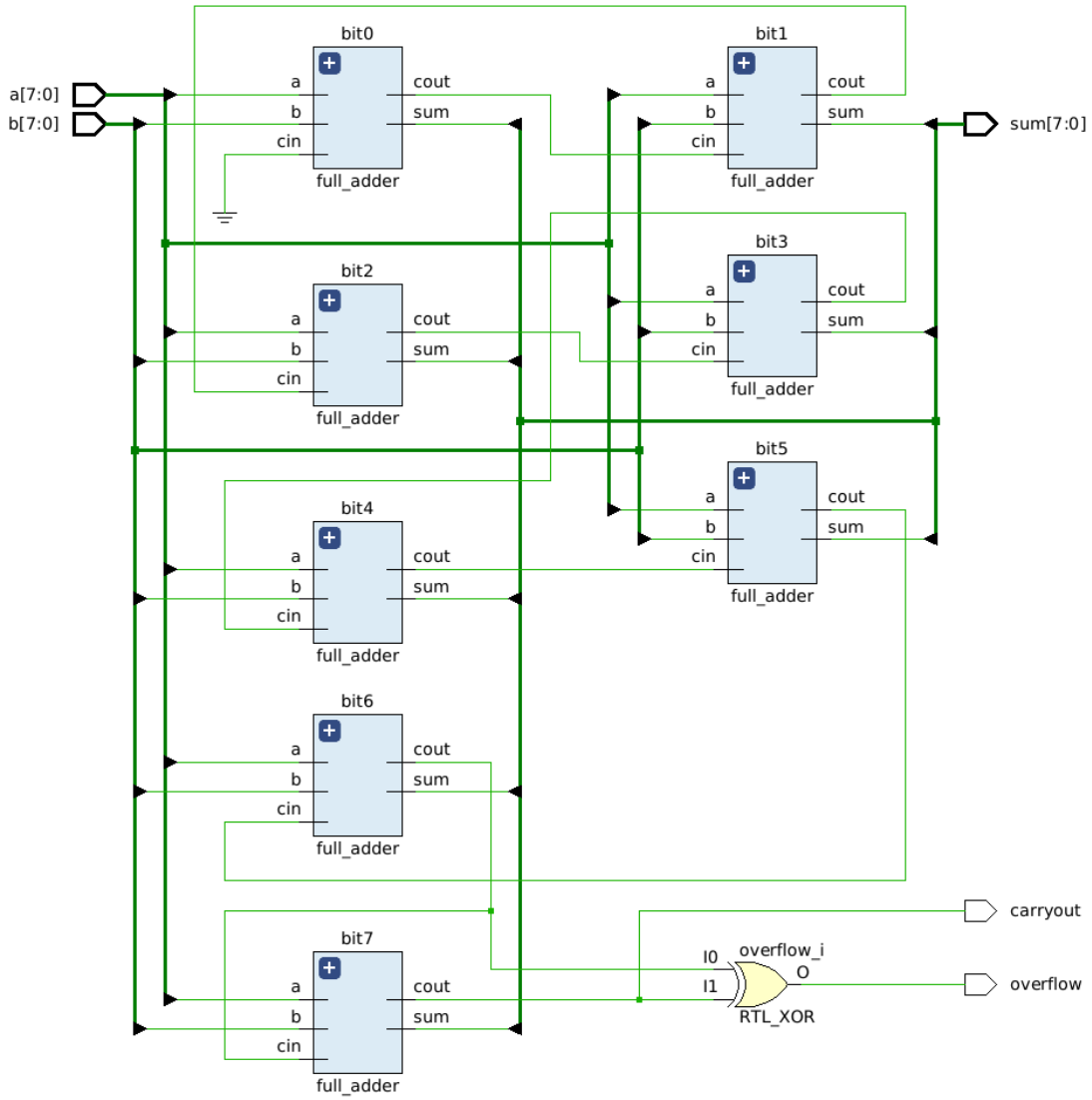
Figure 5. Schematic of ALU Control Unit



Perhaps one of the most complex circuits we designed, the ALU control unit was purely designed in behavioral Verilog. Using a line of if-else statements, we check the 2-bit ALU-Op for the instruction type whereby 0 is the U-type instruction for load and store, 1 is the I-type instruction for logical operations like AND and OR, and 2 is the R-type instruction for register operations like add and subtract. Once the instruction type is filtered, then the 10-bit function code selects which exact operation is being used. This method of if/else-if/else control flow resulted in a very long diagram of instructions. This could have been simplified by simply ignoring the ALU-Op and instead just focusing on the function code, but we decided not to do this for better future implementation. A decision like this implies that a "branch equals" does not result in subtraction, instead it appears as an error to the ALU control unit because both ALU-Op and function-code must be accurate. Ultimately, this circuit is too large to fit neatly on a single page.

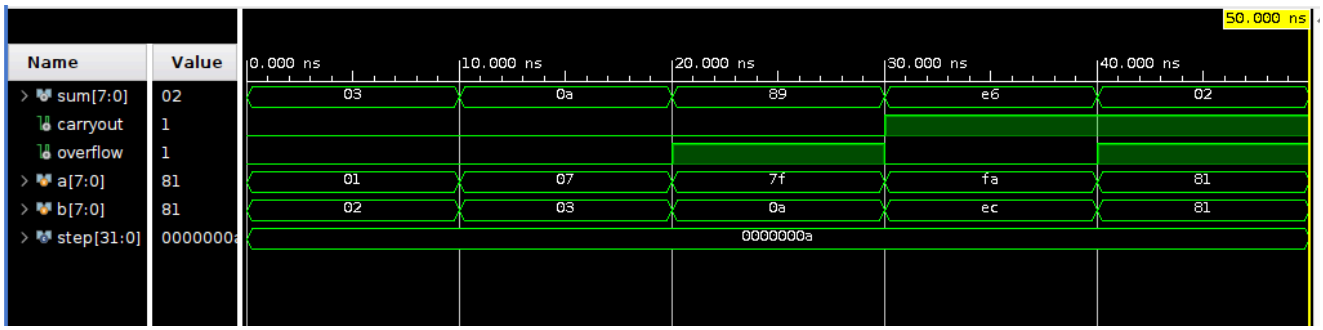
Component: 8-bit Adder

Figure 6. Schematic of 8-bit Adder



A schematic of the adder module which takes in two 8-bit inputs and adds them together. It uses a smaller 1-bit full adder for each bit computed and moves each carry-out into the next carry-in, a technique known as ripple adding. Overflow is detected by a single XOR gate which compares the last carry-in and the last carry-out. If they are different, then overflow has occurred since overflow means that the sign flips. The last bit is the sign bit.

Figure 7. Timing Diagram of ADD operation resulting in overflow



For this particular timing diagram, random numbers were tested which fall into specific categories based on overflow. It goes in the following respective order:

$(positive + positive = positive)$ expecting no overflow,

$(positive + positive = positive)$ expecting no overflow,

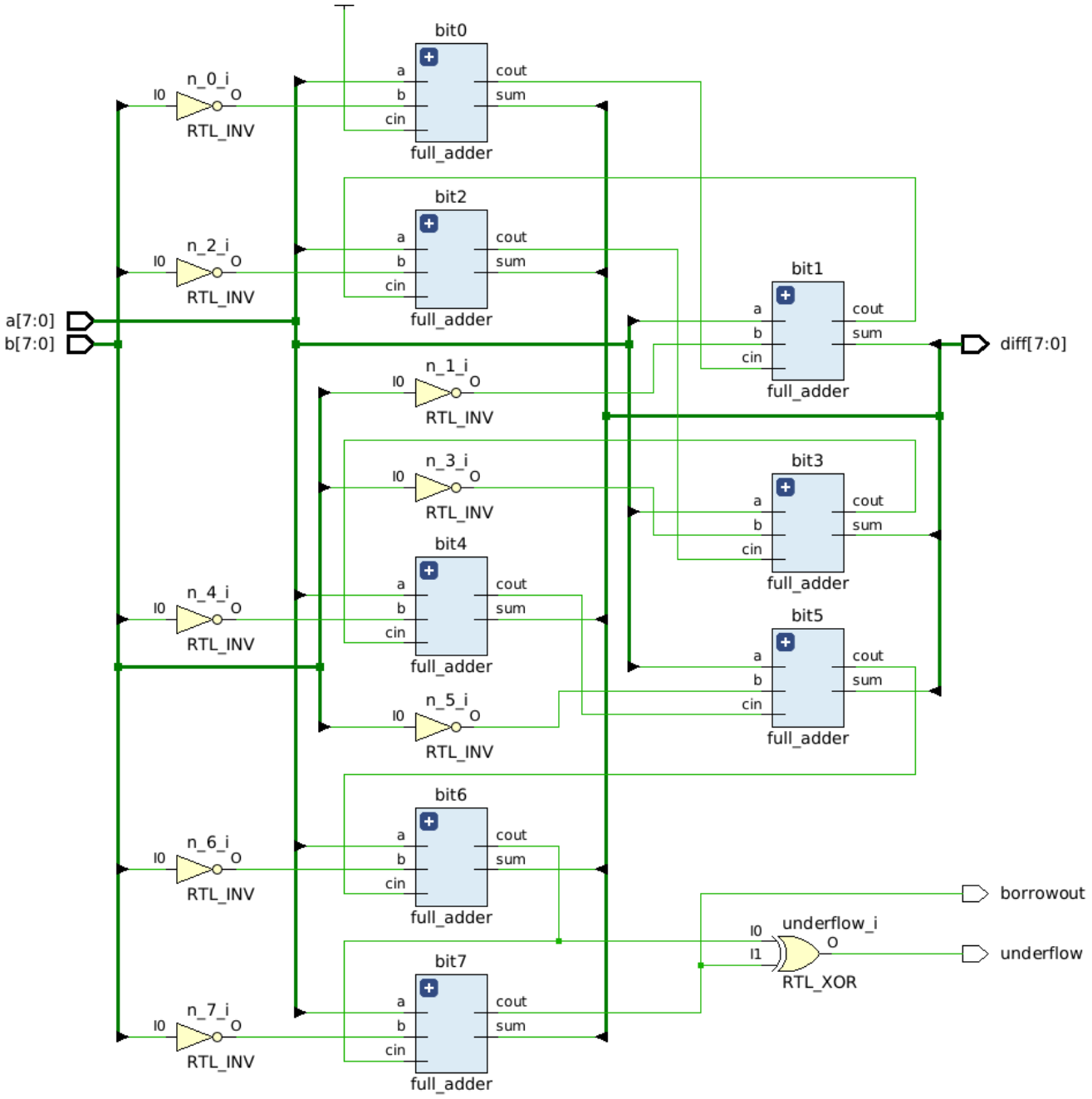
$(positive + positive = negative)$ expecting overflow,

$(negative + negative = negative)$ expecting no overflow,

$(negative + negative = positive)$ expecting overflow.

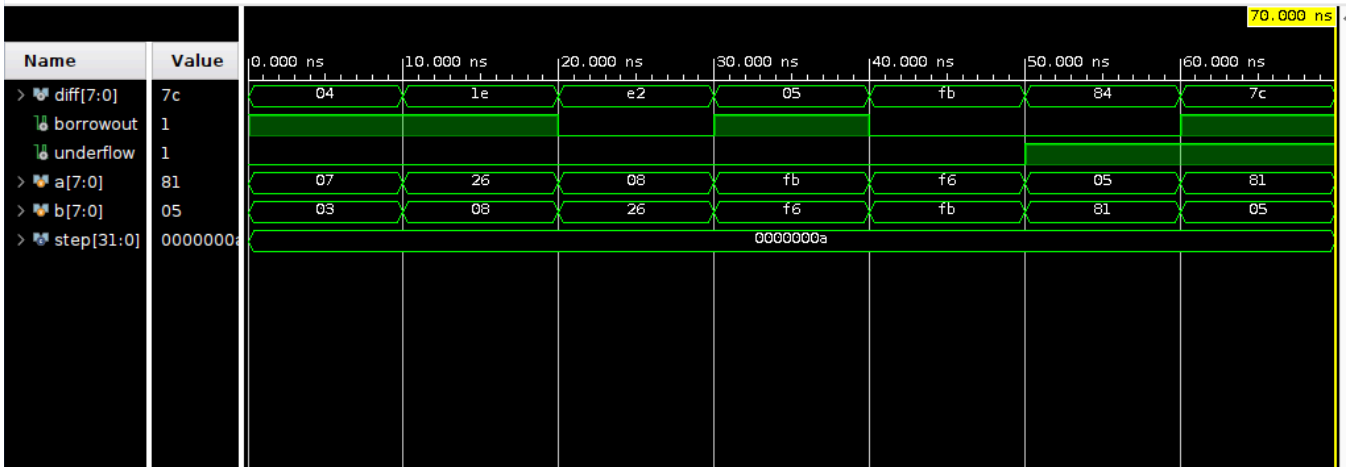
Component: 8-bit Subtractor

Figure 8. Schematic of 8-bit Subtractor



Everything about this 8-bit subtractor schematic is the exact same as the adder, but the difference here is that each bit of the second argument B, also known as the subtrahend, is being flipped by a not-gate and the first borrow-in is set to a constant of 1. The reasoning behind why this is done is to turn the subtrahend into a negative through two's complement negation before adding.

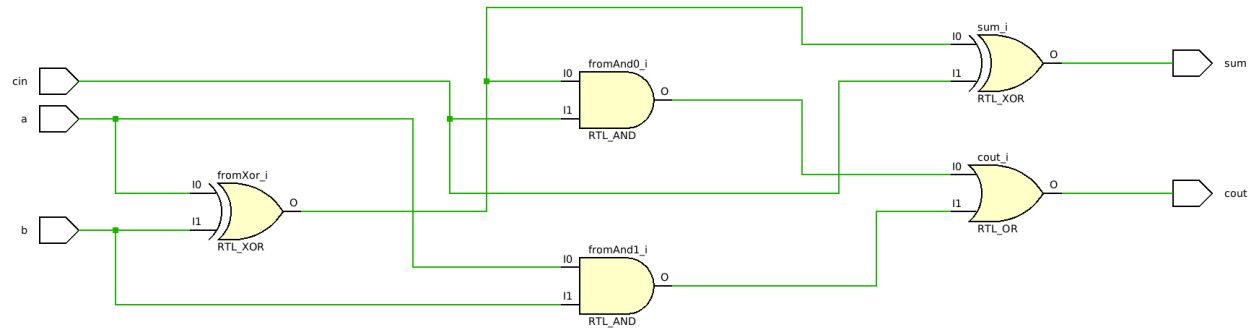
Figure 9. Timing Diagram of SUBTRACT operation



Similar to the 8-bit adder timing diagram, the 8-bit subtractor has random numbers tested which are categorized by expected overflow (or underflow). It goes in the following order:
$(positive - positive = positive)$ expecting no overflow,
$(positive - positive = positive)$ expecting no overflow,
$(positive - positive = negative)$ expecting no overflow,
$(negative - negative = positive)$ expecting no overflow,
$(negative - negative = negative)$ expecting no overflow,
$(positive - negative = negative)$ expecting overflow.
$(negative - positive = positive)$ expecting overflow.

Component: Full Adder

Figure 10. Schematic of Full Adder



This is the exact same tried and true 1-bit full-adder used in the first project. The schematic of this module has not changed at all. Throughout the design, this circuit is only used in the 8-bit adder and subtractor.

Design Tradeoffs:

There are a couple of tradeoffs in our design regarding the use of structural Verilog versus behavioral Verilog. Behavioral Verilog is easier to write and understand because it is more abstract and is similar to higher-level programming languages like C. For example, in our ALU module, we wrote a case statement to handle the different possible operations the ALU can perform instead of structurally implementing a multiplexer. There was no need to implement the Ainvert, Binvert, and overflow detection components that are shown in the book examples, so our implementation with behavioral Verilog was simpler.

However, there are some downsides to using behavioral Verilog as well; because it is at a higher level of abstraction, it is harder to visualize the actual circuit that the code is implementing and how everything in the circuit is connected. For example, Figures 3 and 5 demonstrate the complexity of the circuits when using behavioral Verilog. They are difficult to follow and may be overly complex; behavioral Verilog does not give you full control over the structure of the circuits being implemented. Figures 6, 8, and 10 convey how structural Verilog can be used to generate simpler circuits that are easier to understand. Structural Verilog gives you full control over the structure of each circuit.

Another limitation of behavioral Verilog is its inability to implement certain components that were necessary to complete the RISC-V ALU. We were unable to implement the carry-out output of our subtractor and adder using behavioral Verilog since it only supports a normal add operation; we utilized structural Verilog to create a ripple carry adder and subtractor (Figures 6 and 8). There were some instances where we used both structural and behavioral Verilog in the same module in order to take advantage of their strengths, such as in the aluV_8.v file, where structural Verilog is used to instantiate the adder and subtractor, and behavioral Verilog is used to choose which operation to perform based on the value of the ALU control.

We also decided to make use of both the ALUOp and FuncCode inputs in our ALU control unit instead of just the FuncCode input. We check to make sure the ALUOp and FuncCode both correspond to the correct operation. It made our circuit much more complicated (Figure 5), but it made our code safer and added two fail-safes to our control unit.