Creating the Arithmetic Sub-Block

MIPS Integer ALU Arithmetic

Our ALU is required to support four integer arithmetic operations: Signed Addition (ADD), Unsigned Addition (ADDU), Signed Subtraction (SUB), and Unsigned Subtraction (SUBU). For signed operations, the inputs and output will be treated as 32-bit signed two's complement integers.

Operation	ALUOp(1)	ALUOp(0)
ADD	0	0
ADDU	0	1
SUB	1	0
SUBU	1	1

We can make use of several properties of signed two's complement numbers to significantly reduce the hardware requirements of our *Arithmetic* sub-block without significantly decreasing its speed. In fact, we can implement all four of the necessary operations using a single 32-bit adder!

Let's start by examining the signed vs. unsigned issue. The hardware for adding or subtracting two's complement signed integers and magnitude representation unsigned integers is actually exactly the same. The only difference comes in determining when *Overflow* occurs. *Overflow* is the condition where the result of the operations is greater in magnitude than that which a given storage location can store or represent. For signed numbers, overflow can only occur when adding two numbers of the same sign or subtracting two numbers of opposite signs. It is indicated for addition by the result having the opposite sign of the inputs and in subtraction by the result having the opposite sign of the first input. In unsigned numbers, and overflow would be indicated by a carry-out from the most significant bit.

A trickier issue is how we can perform both addition and subtraction with just a single 32-bit adder. The answer comes by restating the operation: $A - B = A + (-B)$. We can negate a two's complement number by inverting the bits and adding one: $(-B) = NOT(B) + 1$. So, $A - B = A + (-B) = A + NOT(B) + 1$. By adding and inverter and a multiplexor (to select B or NOT(B)) in front of the B input we can implement $A + B$ or $A + NOT(B)$ easily. To add one, we take advantage of the fact that our adder has a carry-in. By carrying in a '1', we are effectively adding one to the result. This trick also works out for unsigned subtraction.

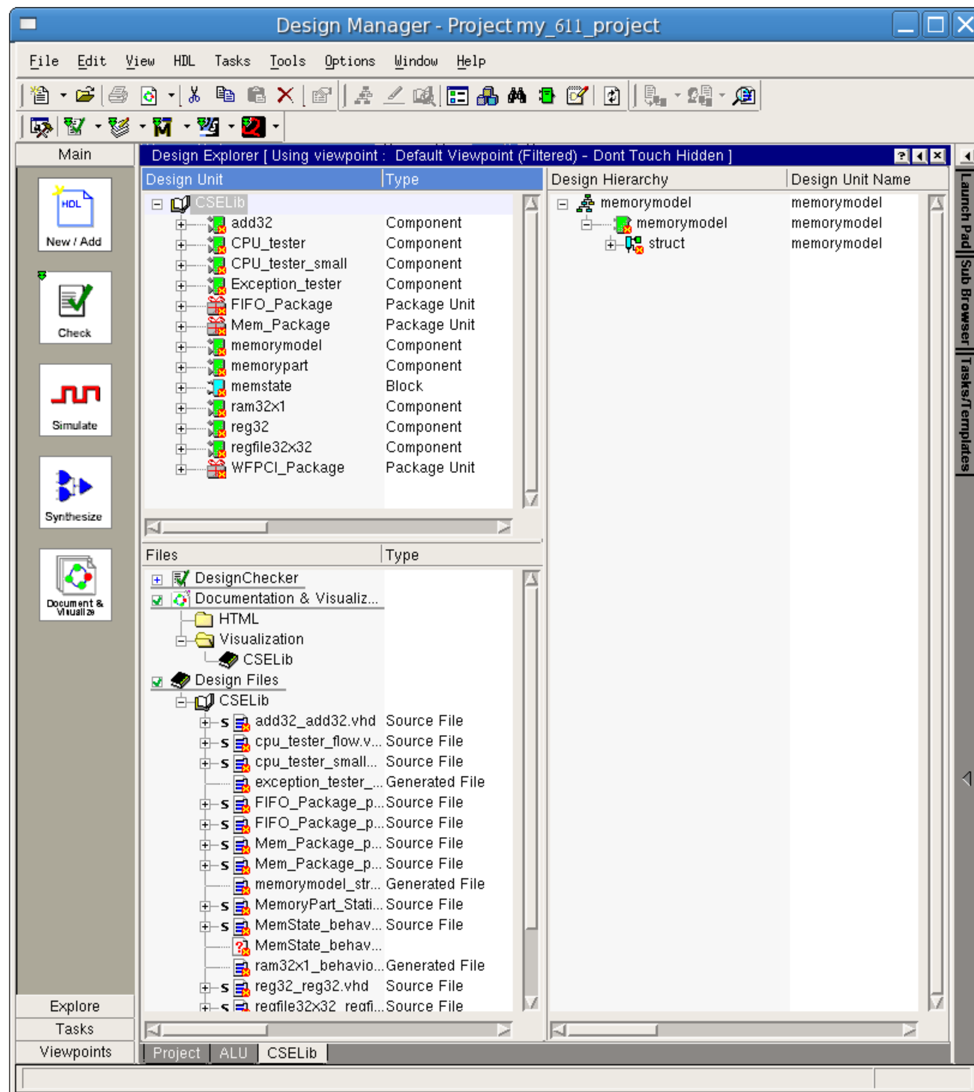
Building the Arithmetic Sub-Block

To create the *Arithmetic* sub-block we will need to have a 32-bit adder with carry-in, carry-out, and overflow computation. Instead of going through the process of building this from scratch, we will be using an adder which has already been designed and instantiating it as a *Component* in our design.

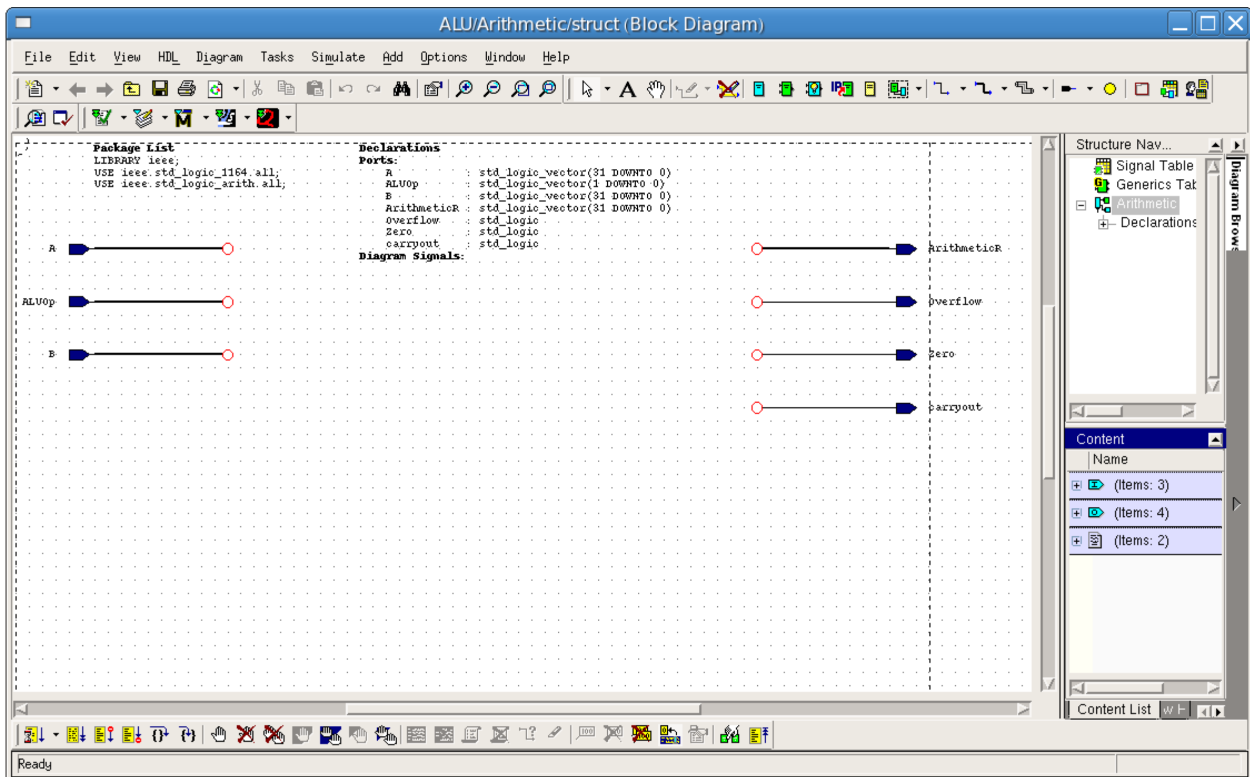
You have already seen that the blue colored sub-blocks which you have placed in the *ALU* block diagram generate entity declarations with corresponding architectures created from their views which are then instantiated in the *ALU* architecture as components. In **FPGA Advantage**, however, these sub-blocks are not referred to as components because you cannot instantiate them in another design as they are; they are tied to one instance in one design. A sub-block can, however, be converted into a **FPGA Advantage** component, represented by a green colored box. They are essentially the same except for the fact that a component can be accessed from other design units to be instantiated.

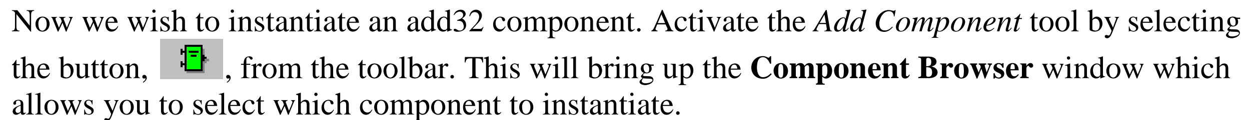
Components allow for the creation of design libraries of commonly used objects which can be instantiated in new designs. A 32-bit adder component has been created and placed it in the **CSELib** library, which holds public design components that we will be using.

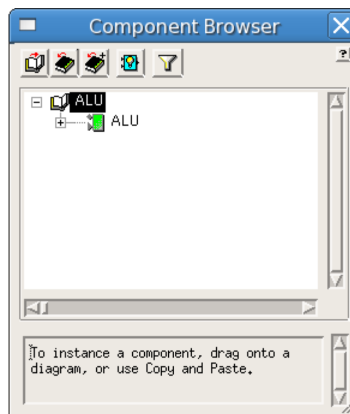
It is not necessary for a library to be open in the **Design Manager** in order to instantiate a component from it. In order to see what is in the library, though, click on the **Project** tab on the bottom of the **Design Manager**. Within that window, double-click on the **CSELib** under Protected Libraries. **Design Explorer** will now open up the *CSELib* so that you may browse it. Expand the design tree in the source window to see a single design unit, in green, called *add32* with a symbol, a *VHDL Entity* and a *VHDL Architecture* view called *add32*. The **Design Manager** window will look like the figure below:



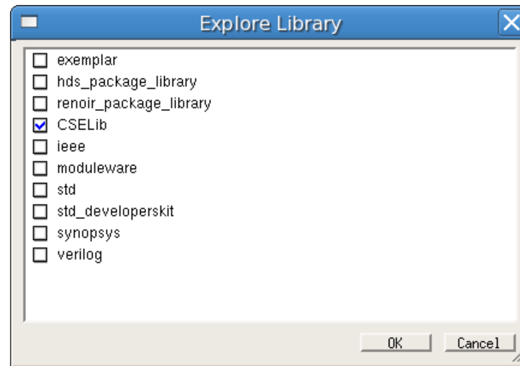
Now that we know what's in the **CSELib** library, we can get started designing the *Arithmetic* unit. Go back to your ALU design by clicking on the **ALU** tab below and opening the design. Once you have done this, create a new block diagram view for the *Arithmetic* sub-block by double-clicking on the sub-block and selecting **Block Diagram** from the window that opens. Click the **Next** button and then the **Finish** button. This will bring up the block diagram with ports seen in the figure below:



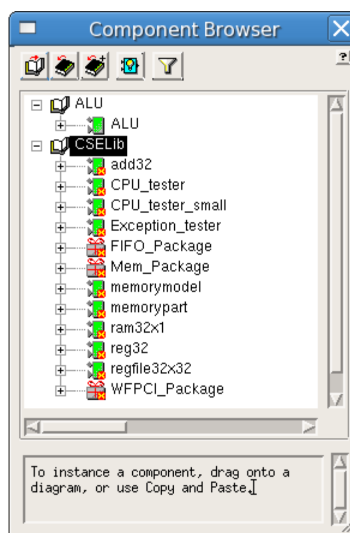

Now we wish to instantiate an add32 component. Activate the *Add Component* tool by selecting the button, , from the toolbar. This will bring up the **Component Browser** window which allows you to select which component to instantiate.



Click the **Add Library** button, , then select the **CSELib** library and click the **OK** button.



You will then see a list of the components in this library. Drag and drop the *add32* component into your design.



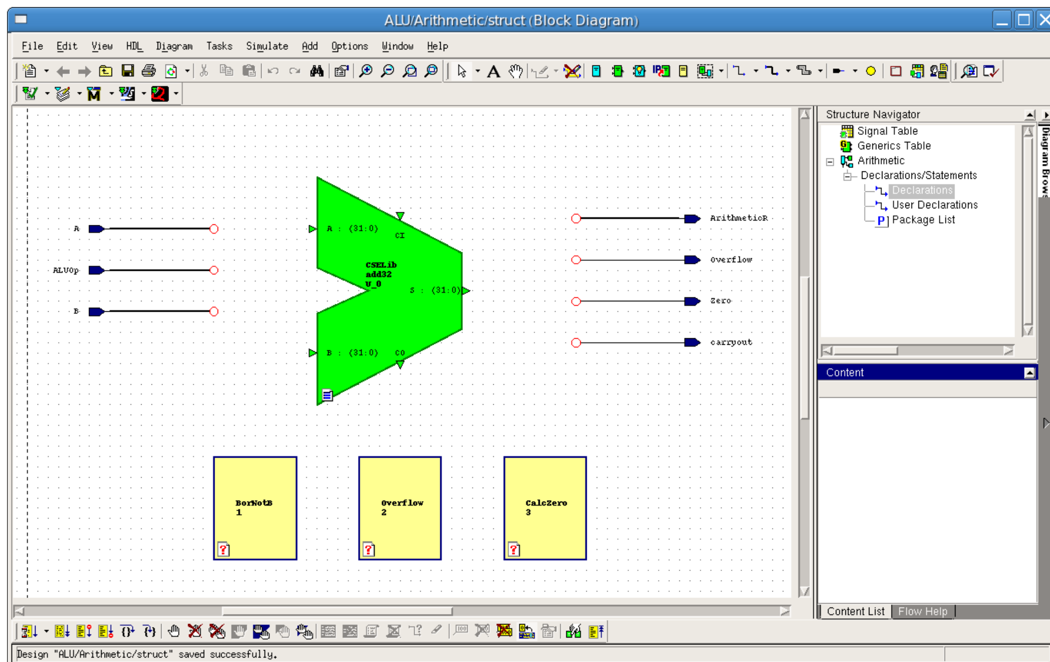
You will now see a message box pop up asking whether you wish to add packages to your design. This is a VHDL package which is used in the adder. Click on the **Yes** button to move on.

In addition to the **add32** component, there are several other tasks which need to be accomplished: selecting *B* or *NOT(B)* and deciding whether to carry-in a ‘1’ to configure for addition or subtraction; deciding whether or not to compute overflow; and calculating *Zero* which is high whenever the result is equal to zero and low otherwise.

These tasks will be implemented using *Embedded Blocks* just like in the *Logical* sub-block. We need to place

- One embedded block called *BorNotB* and put it to the left of the **add32** component. This will perform two functions, one is to create an inverted version of *B* and the other is to assign *B* to the internal signal *Btemp* for addition (ALUOp(1) = ‘0’) and assign the inverted version of *B* for subtraction (ALUOp(1) = ‘1’).
- The second embedded block will determine whether *ADD and SUB* operations generate an overflow. Note ADDU and SUBU operations ignore overflow. The block will require the ability to read the sign bits of inputs A, B and ArithmeticR, the ALUOp as well as the CO signal. If the conditions are met, the block will output a ‘1’ to the Overflow output, otherwise a ‘0’ will be passed to the port. Place this block to the right of and below the **add32** component and call it *Overflow*.
- The last embedded block, *CalcZero,* will be placed to the right of the **add32** component. Its primary function is to set the *Zero* output high when the adder result is equal to zero and set it low otherwise. This requires the block to "read" the result coming out of the **add32** component. Since it is not allowed to “read” values from output only ports, we cannot directly connect the result *ArithmeticR* to the output of the adder. We must instead create an internal signal which can be “read” to determine zero. Thus, as a second function, this embedded block will assign the value of the internal signal *S* to the output *ArithmeticR*.

After adding these blocks your block diagram should look like the figure below:



Now that the embedded blocks and the **add32** component have been placed on the block diagram, connect the input and output ports and create the necessary internal signals so that the block diagram resembles the figure below:

The final step, after all signals are connected properly, is to define the behavior of the embedded blocks. Two of the three blocks will have a **Text** view whereas the Overflow block has a **Truth Table** view. Double-click on the embedded blocks and set their associated view. The embedded block views are as follows:

- Block Name *BorNotB*
- Type of View *Text*
- VHDL Codes *Btemp <= B when ALUOp(1) = '0' else not(B);*
- Description
  This embedded block implements a multiplexor with *B* as the input when ALUOp(1) equals zero and *NOT(B)* when it equals one.

- Block Name *CalcZero*
- Type of View *Text*
- VHDL Codes:
  ArithmeticR <= S;
  Zero <= NOT(S(31) OR S(30) OR S(29) OR S(28) OR S(27) OR S(26) OR S(25) OR S(24) OR S(23) OR S(22) OR S(21) OR S(20) OR S(19) OR S(18) OR S(17) OR S(16) OR S(15) OR S(14) OR S(13) OR S(12) OR S(11) OR S(10) OR S(9) OR S(8) OR S(7) OR S(6) OR S(5) OR S(4) OR S(3) OR S(2) OR S(1)OR S(0));
- Description
  This block contains two VHDL statements, necessary because we cannot simultaneously assign the output of the adder directly to *ArithmeticR* and at the same time use that value to determine the *Zero* output. The first statement takes the internal value of the adder output, *S*, and assigns it

to the result, *ArithmeticR*. The second statement determines whether the adder output is equal to zero by ORing the individual bits together. If the result of the OR is '0', meaning the result is zero, we want to output a '1' and if the result of the OR is a '1', meaning the result is not zero, we want to output a '0'. Thus we invert the result of the OR to obtain the *Zero* output.

- Block Name *Overflow*
- Type of View *Truth Table*
- Truth Table:

ALU/Arithmetic/struct:Overflow (Truth Table)

	A	B	C	D	E	F
1	ALUOp	A(31)	B(31)	S(31)	CarryOut	Overflow
2	"00"	'0'	'0'	'1'		'1'
3	"00"	'1'	'1'	'0'		'1'
4	"10"	'0'	'1'	'1'		'1'
5	"10"	'1'	'0'	'0'		'1'
6						'0'

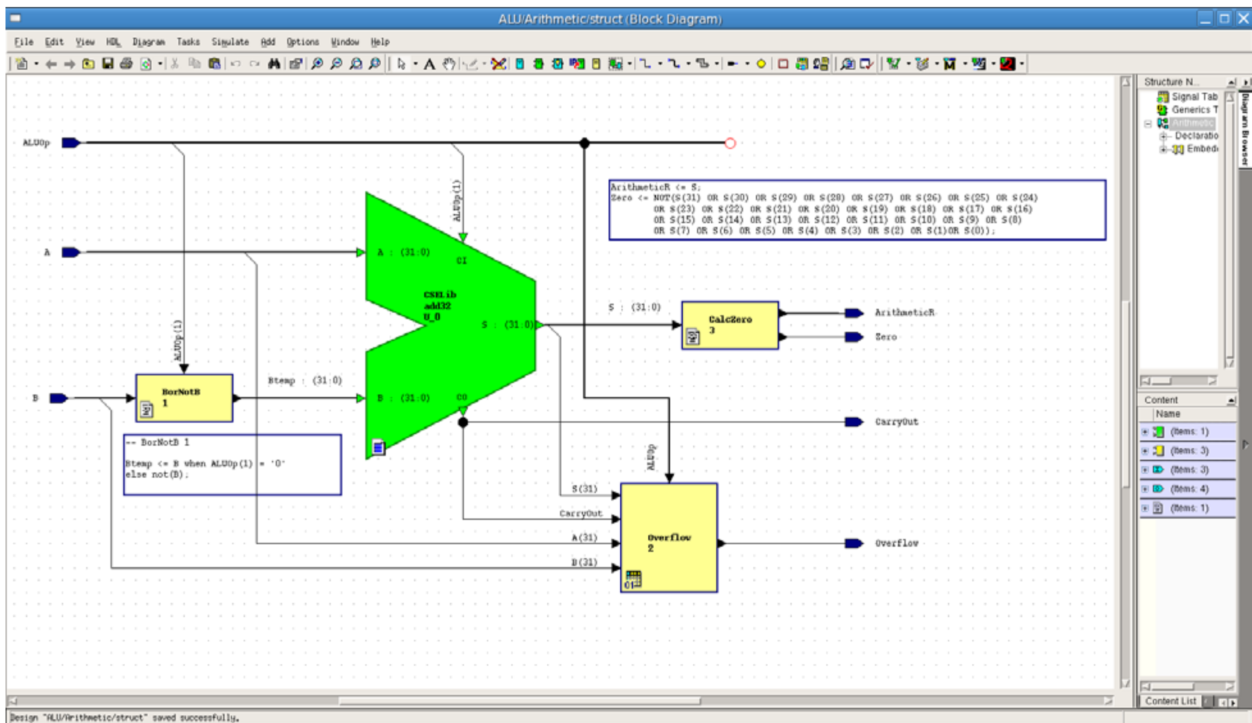
Design "ALU/Arithmetic/struct" saved successfully.

- Description

  Truth Table will test the defined situations that will produce an overflow. If conditions are met, the Overflow output will be set accordingly. Once again, by utilizing don't cares, we can reduce the amount of conditions to get the minimum logics required.

Another step necessary to implementing subtraction, carrying in a one, is taken care of by connecting ALUOp(1) to the carry-in of the adder. For addition, the carry-in is zero; for subtracting, the carry-in is one.

Your Arithmetic sub-block should now resemble the figure below:



Simulating the Arithmetic Sub-Block

Now that we have designed the *Arithmetic* sub-block, we need to verify that it is functioning correctly. Since we are once again dealing with a fairly small design unit, we will test it by directly stimulating the signals in the simulator and observing the results in the Wave window.

You will need to create a simulation macro file (do file) which will run at least one set of vectors to test each possible outcome of each operation:

- **ADD :** Signed Addition or Addition with Overflow
  - *A* is positive, *B* is positive, and the result is less than or equal to $+2^{31}-1$ (no overflow).
  - *A* is positive, *B* is positive, and the result is greater than $+2^{31}-1$ (overflow).
  - *A* is positive, *B* is negative, the result is **not equal** to zero, there should never be overflow.
  - *A* is positive, *B* is negative, the result is **equal** to zero, there should never be overflow.
  - *A* is negative, *B* is positive, the result is **not equal** to zero, there should never be overflow.
  - *A* is negative, *B* is positive, the result is **equal** to zero, there should never be overflow.
  - *A* is negative, *B* is negative, and the result is greater than or equal to -2^{31} (no overflow).
  - *A* is negative, *B* is negative, and the result is less than -2^{31} (overflow).
- **ADDU :** Unsigned Addition without Overflow
- **SUB :** Signed Subtraction or Subtraction with Overflow
  - *A* is positive, *B* is positive, the result is **not equal** to zero, there should never be overflow.
  - *A* is positive, *B* is positive, the result is **equal** to zero, there should never be overflow.
  - *A* is positive, *B* is negative, and the result is less than or equal to $+2^{31}$ (no overflow).
  - *A* is positive, *B* is negative, and the result is greater than $+2^{31}$ (overflow).

- *A* is negative, *B* is positive, and the result is greater than or equal to -2^{31} (no overflow).
- *A* is negative, *B* is positive, and the result is less than -2^{31} (overflow).
- *A* is negative, *B* is negative, the result is **not equal** to zero, there should never be overflow.
- *A* is negative, *B* is negative, the result is **equal** to zero, there should never be overflow.

•**SUBU :** Unsigned Subtraction without Overflow

You may find it helpful when trying to create your test vectors to use a calculator. Most calculator programs allow you to convert between decimal and signed two's complement binary representations of numbers.

Create and run your test macro to confirm the function of your *Arithmetic* unit. Comment your macro file to indicate the decimal equivalents of the test vectors as well as the expected values of *ArithmeticR*, *Zero*, and *Overflow*.

In the next tutorial, we will create the Comparison sub-block.