Assignment 5

Alex Clemmer
Student number: u0458675

DISCUSSION:

Unsigned Addition is stupid-simple. The `signed` bit *must* be `0`, as must be the $\texttt{B}_{inverse}$ bit. The only way to overflow an unsigned number is to trip the `carry` bit, so that's what we look for.

Unsigned Subtraction is nearly as simple. Its $\texttt{B}_{inverse}$ bit *must* be tripped, since we invert the number for subtraction. Since we are counting on the number to overflow until and go to the other side of the number wheel, we look for the `carry` bit to be UNtripped in the case of overflow.

Signed Addition is trickier. Its `signed` bit is clearly `1` (unlike the previous two), and its $\texttt{B}_{inverse}$ bit is `0`, because we are not subtracting. We ignore the `carry` bit and instead concentrate on the last bits of the operands and the sum. The key here is that $\texttt{a}_{31}$ and $\texttt{b}_{31}$ must be the SAME as each other (either both positive or both negative), while the `sum` must be DIFFERENT (i.e., the opposite of them).

We do this because it is impossible to add two numbers of different signs from within the range of possible values and end up with something outside the range. For example, if our range is `-8` to `7`, and we add the minimum positive integer, `1` and the minimum negative integer, `-8`, we end up with only `-7`.

On the other hand, the careful reviewer will note that if we add `-8` and `-8`, we will get `0`. But note there that we have added two numbers of the SAME sign, and then the sum had a DIFFERENT sign – which is exactly what we're checking for here!

Signed Subtraction is clearly the trickiest of all. We track $\texttt{a}_{31}$, $\texttt{b}_{31}$, `signed`, $\texttt{B}_{inverse}$, and `carry`, but NOT the `sum`.

Here's what all that means: the number is signed, so `signed` is always `1`. The operation is subtraction, so $\texttt{B}_{inverse}$ is also always `1`. That's the easy stuff.

The more challenging part is, where can subtraction give us overflow? The answer is: when we are subtracting two numbers with different signs. For example: given the range (again) of `-8` and `7`, if we subtract `-8` by `7`, we end up with what should be `-14`, but really is `1`. Conversely if we subtract `8` by `-7`, we should end up with 14, but really end up with `-1`.

Note also that if you subtract two numbers of different signs, you will always end up in the range. e.g., `-8 - 7 = -1`. Since this was covered in the last paragraph of signed addition, I won't retread here.

So on the high level, that's how it works, but when you get down to it, what matters to us is how the bits are represented in the actual ALU. We know that the second operand, B, will get inverted so that we can add it to A. Here's where it gets a bit dicey: since the times where it overflows are when we subtract two numbers of different signs, and B gets inverted, here they will end up having the SAME sign. So, whatever happens, $\texttt{a}_{31}$ and $\texttt{b}_{31}$ will both be the same. We also know that this operation WILL trip the `carry` bit, because we're overflowing.

ALL these conditions must be present for overflow in signed subtraction. They individually do not constitute overflow themselves.

TRUTH TABLE:

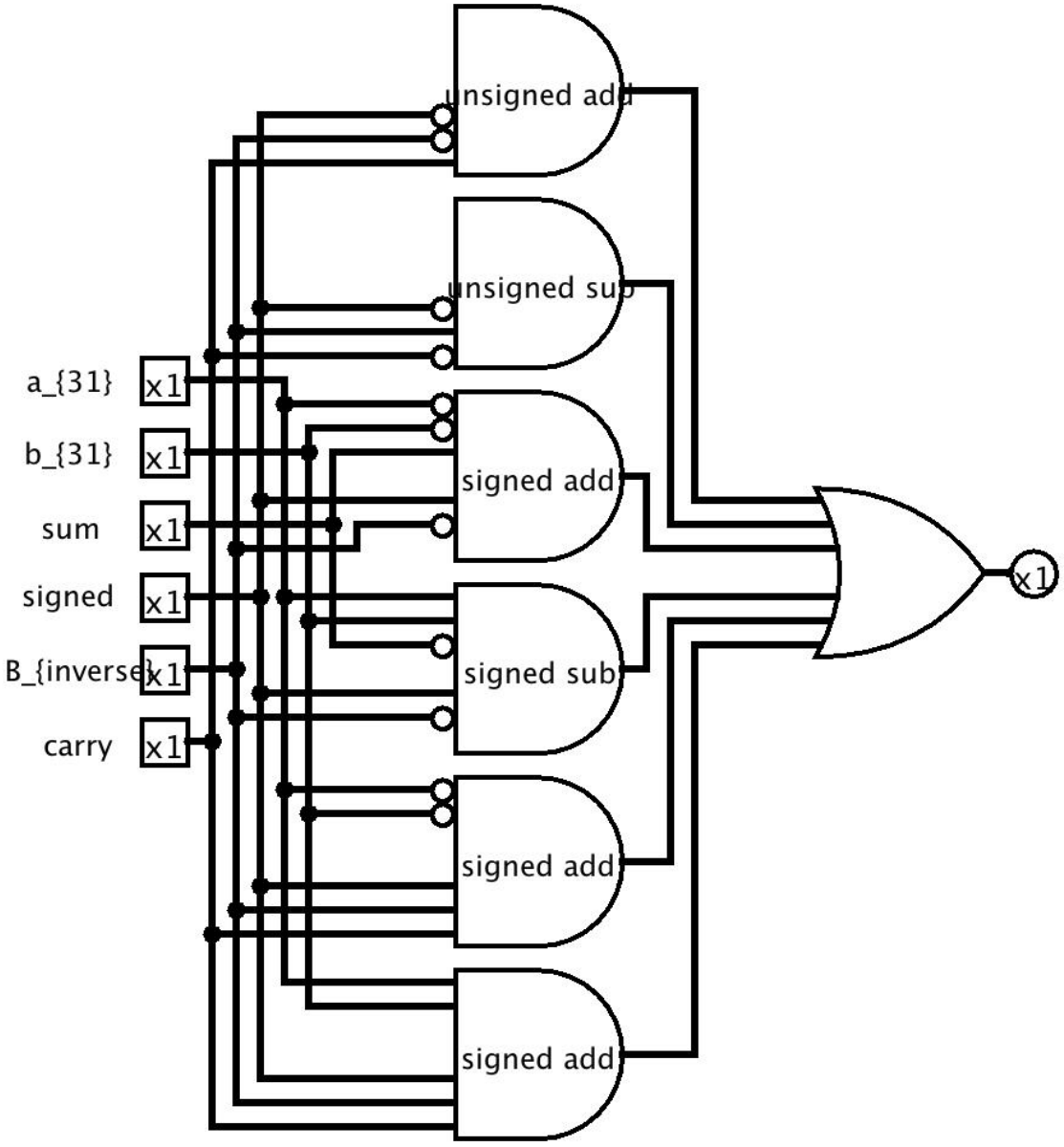
Note that the character '-' indicates that the given bit does not factor into whether the given operation overflows.

	a_{31}	b_{31}	*sum*	*signed*	$B_{inverse}$	*carry*
unsigned addition	-	-	-	0	0	1
unsigned subtraction	-	-	-	0	1	0
signed addition	0	0	1	1	0	-
	1	1	0	1	0	-
signed subtraction	0	0	-	1	1	1
	1	1	-	1	1	1

SUM-OF-PRODUCTS EQUATION:

$$(SIGN'B'_{inv}CARRY) \vee (SIGN'B_{inv}B'_{inv}) \vee (a'_{31}b'_{31}SUMSIGNB'_{inv}) \vee (a_{31}b_{31}SUM'SIGNB'_{inv}) \vee (a'_{31}b'_{31}SIGNB_{inv}CARRY) \vee (a_{31}b_{31}SIGNB_{inv}CARRY)$$

ALU CIRCUIT:


unsigned add
unsigned sub
signed add
signed sub
signed add
signed add
a_{31}
b_{31}
sum
signed
B_{inverse}
carry
x1
x1
x1
x1
x1
x1
x1