Assignment 6

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 `1`, since we invert the number for subtraction. Since we are counting on the number to overflow 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 if, given this, 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 viewer will note that if we add `-7` and `-8`, we will get `1`. 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! This turns out to be our rule for overflow.

Signed Subtraction is clearly the trickiest of all. Actually, it's not: it's really just the same as addition when you invert the second operand. In fact, you can clearly use the exact same gates for both operations. A more detailed explanation follows anyway.

Ostensibly, we track $\texttt{a}_{31}$, $\texttt{b}_{31}$, `signed`, $\texttt{B}_{inverse}$, and `sum`, but NOT the `carry`. 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 actually pretty straightforward: 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 `-15`, but really is `1`. Conversely if we subtract `8` by `-7`, we should end up with 15, 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. But here's where it gets a bit dicey: the times where it overflows are when we subtract two numbers of different signs, but B gets inverted, so here they will end up having the SAME sign as far as the ALU is concerned (as indicated by $\texttt{a}_{31}$ and $\texttt{b}_{31}$, anyway. So, whatever happens, $\texttt{a}_{31}$ and $\texttt{b}_{31}$ will both be the same in cases where we will overflow. What seals the deal is that we are essentially, at this point, adding two numbers of the same sign together. So if the `sum` bit is not the same, then clearly the operation has pushed too far.

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.

Note also that signed addition and subtraction are actually identical, even though subtraction will always have the $\texttt{B}_{inverse}$ bit flipped. I have thus included this bit as a paranthetical in the table, and omitted it from both the circuit and the Sum-Of-Products.

	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	0	1	(1)	-

SUM-OF-PRODUCTS EQUATION:

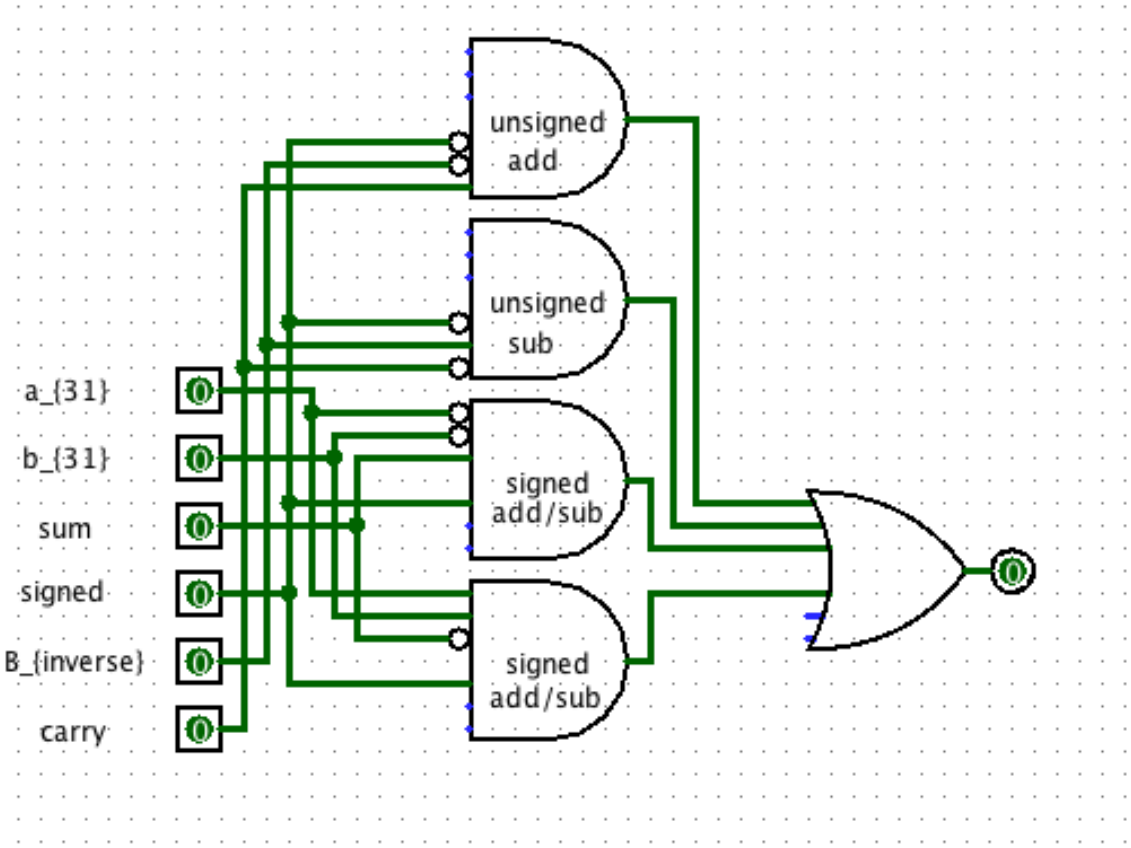
Key: a_{31} = `a`; b_{31} = `b`; *sum* = `su`; *signed* = `si`; $B_{inverse}$ = `bi`; *carry* = `c`

(si' bi' c) ∨ (si' bi c') ∨ (a' b' su si) ∨ (a b su' si)

If this isn't clear, this one is basically the same thing, but with more obvious names:

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

ALU CIRCUIT:


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