COMSM1302 Lab Sheet 2 (Solutions)

Half adder

The truth table for C_out is the exact same as for $A \wedge B$ - for implementation see subcircuit *half adder*.

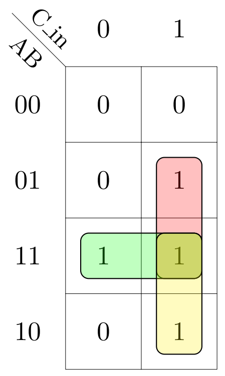
A	B	C_out
0	0	0
0	1	0
1	0	0
1	1	1

The truth table for S is the exact same as for $A \oplus B$ - for implementation see subcircuit *half adder*.

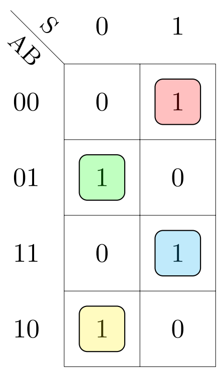
A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

An incrementer adds 1 to its input. Since the half adder subcircuit only accepts 1-bit signals, we first split the 4-bit input into four separate bits. Each bit is processed by a half adder, so we need four copies of the subcircuit. There is no B input, so this pin on each adder is connected to the C_{out} of the previous one, except for the LSB adder, where B is set to a constant 1. Finally, we recombine the four S outputs into a single 4-bit signal with a splitter. For implementation see subcircuit *4-bit incrementer*.

Full adder



$C\_out \equiv (A \wedge B) \vee (C\_in \wedge (A \oplus B))$



$S \equiv A \oplus B \oplus C$

Our logic for the C_out and S outputs on the full adder can be found through the above Karnaugh maps, which is simplified to the formula in red. We can replace $A \vee B$ with $A \oplus B$ in the C_out formula because they differ only when $A = B = 1$. In that case, $(A \wedge B) = 1$, so the whole expression is already 1 regardless of the second term. For further details, see lecture 2-2: binary addition and for implementation see subcircuit *full adder*.

A full adder can also be built from two half adders: HA1 computes $A + B$, and HA2 computes $(A + B) + C\_in$. This produces two carry signals, which are combined with OR to give a single C_out. A truth table confirms this:

HA1 C_out	HA2 C_out	FA C_out	
0	0	0	e.g. when A = 0, B = 0, C_in = 0
0	1	1	e.g. when A = 0, B = 1, C_in = 1
1	0	1	e.g. when A = 1, B = 1, C_in = 0
1	1	X	we never gets these at the same time

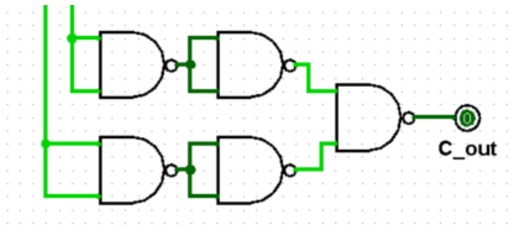
The 4-bit adder can then be constructed using 4 copies of the full adder subcircuit, as explained at the end of the binary addition lecture i.e. connecting each C_out to the next C_in. Splitters divide A and B into 1-bit inputs and recombine the sum bits into the 4-bit output. For implementation see subcircuit *4-bit adder*.

To extend this design to subtraction, we input either B (for addition) or $-B$ i.e. $\neg B + 1$ (for subtraction). A multiplexer selects between B and $\neg B$. The unused C_in of the LSB adder is then connected to the multiplexer's control signal, ensuring that in subtraction mode it adds the extra 1. Thus, when control $= 1$, the circuit performs $A - B$; when control $= 0$, it performs $A + B$. For implementation see subcircuit *4-bit adder/subtractor*.

NAND adders

From the previous lab, we know an XOR can be built with 4 NAND gates (though being able to prove this is non-examinable, and using the 5 NAND gate configuration would be sufficient), which we can use to generate S. For C_out, which requires AND, we need 2 NAND gates. Since the first NAND in the XOR already computes $\neg(A \land B)$, we can reuse that signal and pass it through one more NAND to obtain C_out. For implementation see subcircuit *nand half adder*.

To extend this to a full adder, we duplicate the half adder design and connect the copies like in the standard version. The only addition needed is the NAND equivalent of an OR gate. Adding this produces a 5-NAND configuration, which simplifies to a single NAND by removing double negations:



For implementation see subcircuit *nand full adder*.

Plexers

AB \ Sel	0	1
00	0	0
01	0	1
11	1	1
10	1	0

$Out \equiv (A \land \neg Sel) \lor (B \land Sel)$

For implementation of 2-to-1 multiplexer, based on this Karnaugh map, see subcircuit *2-1 mux*.

The truth table for $Out0$ is the same as for $In \land \neg Sel$, while the truth table for $Out1$ is the same as for $In \land Sel$ - for implementation see subcircuit *1-2 demux*.

In	Sel	Out0	In	Sel	Out1
0	0	0	0	0	0
0	1	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

A 4-to-1 multiplexer selects one of four inputs using two select bits. Each select bit essentially decides *between two options*. Since a 2-to-1 multiplexer already performs a single binary choice, we can combine them hierarchically:

- The first select bit chooses *within each pair* of inputs (e.g. $In0$ vs $In1$, $In2$ vs $In3$), which needs two 2-to-1 multiplexers.
- The second select bit then chooses *between those two results*, which requires a third 2-to-1 multiplexer.

So the structure directly reflects how the two select bits encode the choice out of four: one bit narrows it down to a pair, the other decides within the pair. For implementation see subcircuit *4-1 mux*.

A 1-to-4 demultiplexer works in the reverse way. One input needs to be directed to one of four outputs, according to two select bits. A 1-to-2 demux handles a single binary decision, so we cascade them:

- The first select bit decides which *pair of outputs* will receive the signal.
- The second select bit then decides *which output within that pair*.

Again, the hierarchy mirrors the binary structure of the select lines: two bits $\rightarrow$ two stages of binary decision. For implementation see subcircuit *1-4 demux*.

Arithmetic Logic Unit

The Hack ALU can be built using 2-1 multiplexers. Order matters - zeroing must happen before possible negation, the function f must be applied to the post-processed operands, and finally the *no* negates the resulting signal. If you forget the order, reconstruct it from the Hack truth table!

The zr output is true if (and only if) all the outputs bits are 0 i.e. $\neg bit0 \land \neg bit1 \land \neg bit2 \land \neg bit3 \equiv \neg(bit0 \lor bit1 \lor bit2 \lor bit3)$, while the ng flag is true if (and only if) the output's MSB is 1 i.e. $bit3$. For implementation see subcircuit *alu*.

Number representations

Binary	**Octal**	**Decimal (2's complement)**	**Hexadecimal**	**Decimal (signed magnitude)**	**Decimal (1's complement)**	**Floating point**
10101101_2	255_8	-83_{10}	AD_{16}	-45_{10}	-82_{10}	-1.40625_{10}
01110101_2	165_8	117_{10}	75_{16}	117_{10}	117_{10}	6.625_{10}
11011100_2	334_8	-36_{10}	DC_{16}	-92_{10}	-35_{10}	-3.75_{10}
10000101_2	205_8	-123_{10}	85_{16}	-5_{10}	-122_{10}	-0.578125_{10}
11001010_2	312_8	-54_{10}	CA_{16}	-74_{10}	-53_{10}	-2.625_{10}
00101001_2	051_8	41_{10}	29_{16}	41_{10}	41_{10}	1.28125_{10}
01111011_2	173_8	123_{10}	$7B_{16}$	123_{10}	123_{10}	7.375_{10}
11111101_2	375_8	-3_{10}	FD_{16}	-125_{10}	-2_{10}	-7.625_{10}
10101100_2	254_8	-84_{10}	AC_{16}	-44_{10}	-83_{10}	-1.375_{10}
10010010_2	222_8	-110_{10}	92_{16}	-18_{10}	-109_{10}	-0.78125_{10}
01111111_2	177_8	127_{10}	$7F_{16}$	127_{10}	127_{10}	7.875_{10}
11011010_2	332_8	-38_{10}	DA_{16}	-90_{10}	-37_{10}	-3.625_{10}

For floating point, recall the format here is S(1), E(2), M(5). The decoded value depends on these bit widths - changing the number of exponent or mantissa bits would change the result.

For signed integers, note the pattern: 2's complement, 1's complement, and signed magnitude all give the same decimal values for positive numbers - they only differ in how negatives are represented.