CS61C Spring 2025

Boolean Algebra, CALL

Discussion 7

1 Boolean Logic

1.1 Simplify the following Boolean expressions:

(a) $(A+B)\left(A+\overline{B}\right)C$ = AC

A breakdown of the boolean algebra laws used to simplify the expression is shown below:

$$\begin{aligned}
(A+B)\left(A+\overline{B}\right)C &= \left(AA+A\overline{B}+A\overline{B}+B\overline{B}\right)C && \text{Distributive Property}\\
&= \left(A+A\overline{B}+0\right)C && \text{Idempotent \& Inverse Properties}\\
&= \left(A+A\overline{B}\right)C && \text{Identity Property}\\
&= A\left(1+\overline{B}\right)C && \text{Distributive Property}\\
&= AC && \text{Null Property}
\end{aligned}$$

(b) $\overline{A}\,\overline{B}\,\overline{C}+\overline{A}\,B\,\overline{C}+A\,B\,\overline{C}+A\,\overline{B}\,\overline{C}+A\,B\,C+A\,\overline{B}\,C$ = $\overline{C}+A$

$$\begin{aligned}
\overline{A}\,\overline{C}\left(\overline{B}+B\right)+A\,\overline{C}\left(B+\overline{B}\right)+AC\,\left(B+\overline{B}\right) &= \overline{A}\,\overline{C}+A\overline{C}+AC && \text{Distributive}\\
&= \overline{A}\,\overline{C}+A\overline{C}+A\overline{C}+AC && \text{Idempotent}\\
&= \left(\overline{A}+A\right)\overline{C}+A\left(\overline{C}+C\right) && \text{Distributive}\\
&= \overline{C}+A && \text{Inverse}
\end{aligned}$$

Alternatively, to simplify $\overline{A}\,\overline{C}+A\overline{C}+AC$ with the distributive property $x+yz=(x+y)\cdot(x+z)$:

$$\begin{aligned}
\overline{A}\,\overline{C}+A\overline{C}+AC &= \overline{C}\left(\overline{A}+A\right)+AC\\
&= \overline{C}+AC\\
&= \left(\overline{C}+A\right)\left(\overline{C}+C\right) && \text{Distributive (AND form)}\\
&= \overline{C}+A
\end{aligned}$$

(c) $\overline{A\left(\overline{B}\,\overline{C}+BC\right)}$ = $\overline{A}+B\overline{C}+\overline{B}C$

$$\begin{aligned}
\overline{A\left(\overline{B}\,\overline{C}+BC\right)} &= \overline{A}+\overline{\overline{B}\,\overline{C}+BC} && \text{De Morgan's}\\
&= \overline{A}+\overline{\left(\overline{B}\,\overline{C}\right)}\;\overline{BC} && \text{De Morgan's}\\
&= \overline{A}+(B+C)\left(\overline{B}+\overline{C}\right) && \text{De Morgan's}\\
&= \overline{A}+B\overline{C}+\overline{B}C && \text{Distributive}
\end{aligned}$$

(d) $\overline{A}(A+B)+(B+AA)\left(A+\overline{B}\right)$ = $A+B$

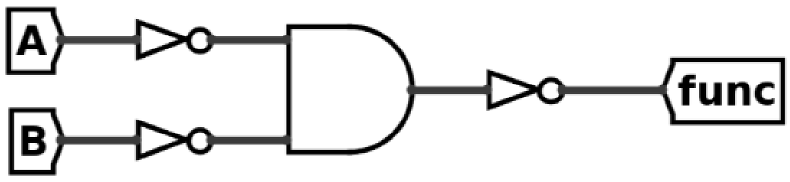
$$\begin{aligned}
\overline{A}(A+B)+(B+AA)\left(A+\overline{B}\right) &= \overline{A}(A+B)+(A+B)\left(A+\overline{B}\right) && \text{Idempotent}\\
&= (A+B)\left(\overline{A}+A+\overline{B}\right) && \text{Distributive}\\
&= (A+B)\left(1+\overline{B}\right) && \text{Inverse}\\
&= (A+B) && \text{Null}
\end{aligned}$$

2 Digital Logic Simplification

For the following digital logic circuits:

1. Write a boolean algebra expression that corresponds the physical circuit.
2. Simplify the expression and draw the simplified circuit.

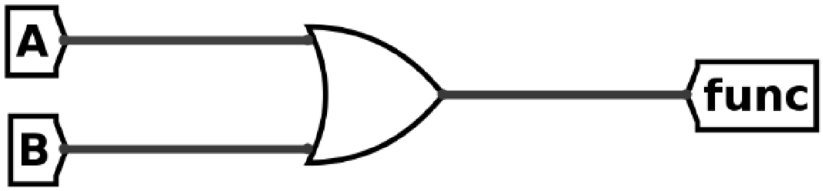
$\boxed{2.1}$



We can start by labeling the inputs / outputs of each of our logic gates. The first two gates after the A and B inputs are NOT gates which output $\overline{A}$ and $\overline{B}$ respectively. These are fed as inputs to the AND gate which will output $\overline{A} \cdot \overline{B}$. Lastly, this is fed into NOT gate which is our output expression $F(A, B) = \overline{\overline{A} \cdot \overline{B}}$. We can simplify this expression with De Morgan's law to get a simplified expression:

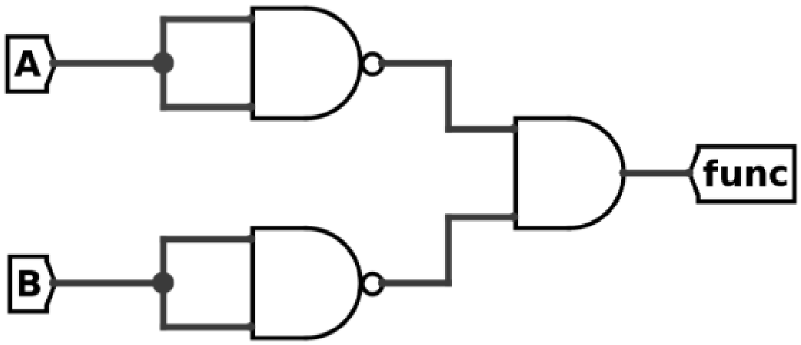
$$\begin{aligned} F(A, B) &= \overline{\overline{A} \cdot \overline{B}} \\ &= \overline{\overline{A}} + \overline{\overline{B}} \\ &= A + B \end{aligned}$$

Redrawing our simplified circuit, we get:



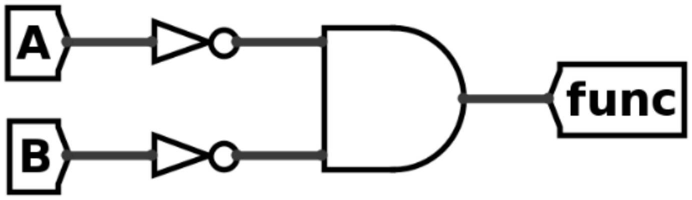
Which is just an OR gate. As extra practice, you can verify the simplification by writing truth tables for each expressions and verifying that they match.

2.2

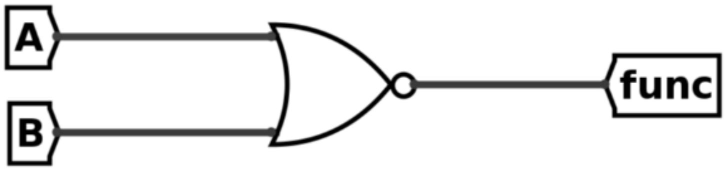


Following similar logic as above, can write expressions for the inputs and outputs of each of the logic gates. The first NAND gate has inputs A and A for an output of $\overline{A \cdot A}$ ("not A and A") and similarly for the second NAND gate. Lastly, the outputs are fed into an AND gate and can be simplified:

$$\begin{aligned} F(A, B) &= \left(\overline{A \cdot A}\right)\left(\overline{B \cdot B}\right) \\ &= \overline{A} \cdot \overline{B} \end{aligned}$$

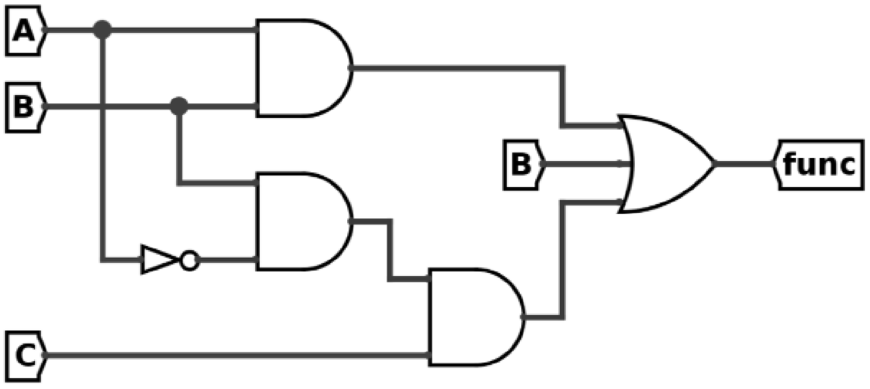


Additionally, $\overline{A} \cdot \overline{B}$ is equivalent to a NOR gate and can be redrawn as:



Note that this demonstrates how NOT gates can be formed by feeding the same input to both inputs of a NAND gate. In fact, *all* boolean algebra circuits can be formed only using NAND gates (think about why this may be the case?).

$\boxed{2.3}$

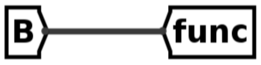


We can label the outputs of each gate in the circuit. In the first layer, we two AND gates for AB and $\overline{A}B$. The final AND gate takes $\overline{A}B$ as its first input and C as its second input for a combined output of $\overline{A}BC$. Lastly, an OR gate combines the previous gates with the input B for the final function of $F(A, B, C) = AB + \overline{A}BC + B$

Following the procedures in question 1, we can use boolean algebra to simplify the equation:

$$\begin{aligned} AB + \overline{A}BC + B &= \left(A + \overline{A}C + 1\right)B \\ &= B \end{aligned}$$

Thus, we can redraw the circuit as simply:



2.4 Why might it be useful to simplify logic circuits?

Complex digital circuits can be simplified to minimize different objectives such as area or cost. In practice, computers use sophisticated algorithms to optimize circuits for many factors including area, cost, and timing requirements (which will be explored in future week's discussions).

3 Combinational Logic from Truth Tables

For this question, we have a single 3-bit input and a single 4-bit output. We want to design a combinational logic circuit to achieve the desired output given the appropriate combinations of input bits (`Input=001` $\implies$ `Output=0011`, and so on...). Here is the truth table we wish to implement:

Input	**Out**
000	0001
001	0011
010	1111
011-111	xxxx

The `x`'s for the final entry of the table indicate that any output is valid for the case that `Input` is `011`, `100`, `101`, `110`, and `111`

3.1 Write out and simplify boolean expressions for each of the output bits `Out[3]`, `Out[2]`, `Out[1]`, and `Out[0]` in terms of the input bits `In[2]`, `In[1]`, `In[0]`.

When deriving expressions for multi-bit values, we find split up the values and find expressions for the individual bits.

Working from right-to-left starting with `Out[0]`, we see that its value is one in all cases which are defined. We can set it to the expression `Out[0] = 1`.

For `Out[1]`, we see that it is `1` whenever `Input=001`, `Input=010`, or for one of the undefined input cases. We can write translate this to an expression as $\text{Out}[1] = \overline{\text{In}[2]}\,\overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]}$. We can also introduce input terms from the undefined cases to help with simplification, namely `Input=101` and `Input=110` to get:

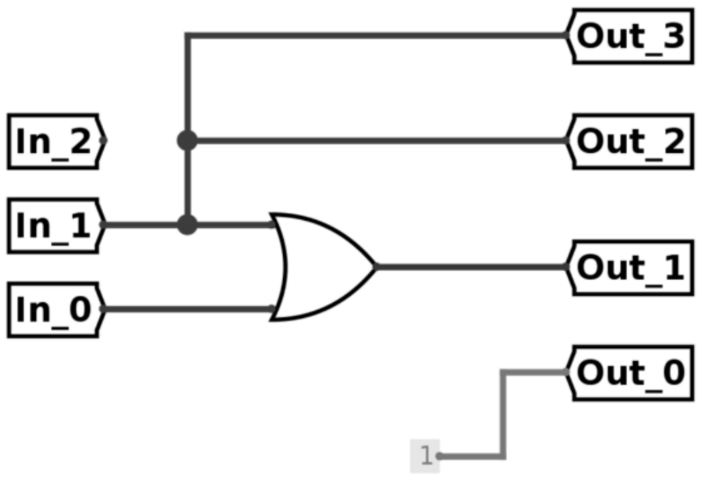
$$\begin{aligned}\text{Out}[1] &= \overline{\text{In}[2]}\,\overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]} + \text{In}[2]\,\overline{\text{In}[1]}\,\text{In}[0] + \text{In}[2]\,\text{In}[1]\,\overline{\text{In}[0]}\\ &= \overline{\text{In}[2]}\,\overline{\text{In}[1]}\,\text{In}[0] + \text{In}[2]\,\text{In}[1]\,\overline{\text{In}[0]} + \text{In}[2]\,\overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]}\\ &= \overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[1]}\,\text{In}[0]\end{aligned}$$

We can also introduce the undefined terms `Input=011` and `Input=111` then we end up with the simplification:

$$\begin{aligned}\text{Out}[1] &= \overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[1]}\,\overline{\text{In}[0]} + \overline{\text{In}[2]}\,\text{In}[1]\,\text{In}[0] + \text{In}[2]\,\text{In}[1]\,\text{In}[0]\\ &= \overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[1]}\,\overline{\text{In}[0]} + \text{In}[1]\,\text{In}[0]\\ &= \text{In}[1] + \text{In}[0]\end{aligned}$$

Following a similar process as above, the final two bits simplify to `Out[3] = Out[2] = In[1]`.

$\boxed{3.2}$ Draw out the boolean circuit based on your simplified expressions above. You may use constants 0 and 1, and the logic gates AND, OR, NOT.



4 Two-Pass Assembly

Consider the following assembly code. Assume that `printf` exists in the C standard library and that `msg` exists at an unknown address in the `.data` section.

```
Address | Assembly
--------|---------------------------
.data   | msg: .string "Hello World"
        |
.text   |
0x0C    |         add  t0, x0, x0
0x10    |         addi t1, x0, 4
0x14    | loop:   beq  t0, t1, end
0x18    |         addi a0, a0, 1
0x1C    |         la   a0, msg      // load address of `msg`
0X20    |         jal  ra, printf
0X24    | n:      addi t0, t0, 1
0X28    |         j    loop
0X2C    | end:    ret
```

4.1 This code is output from the Compiler and may contain pseudoinstructions.

The compiler is responsible for translating high-level language code (e.g. C) to assembly. The compiler's output may contain pseudoinstructions which gets translated by the assembler.

4.2 Assume we are using a two-pass assembler. Fill out the symbol table after the first pass (top-to-bottom) of the assembler. Not all lines may be used.

Symbol Table	
Label	**Address**
`msg`	?
`loop`	`0x14`
`n`	`0x24`
`end`	`0x2C`

The assembler performs two passes over the program to compute all the offsets. Branches and PC-relative jumps that target labels with *positive* offsets are unknown in the first pass (ex: during the assembler's first pass, it encounters the label **`end`** in **`beq t0, t1, end`** before it has seen the address of **`end`**).

The solution is to take two passes over the program. Pass 1 remembers the positions of labels which are stored in the symbol table, and pass 2 uses label positions to generate the machine code. References to static data and external functions cannot be determined at this stage, however, because full 32-bit addresses are unknown until the linker creates the executable.

For the first pass of the assembler:

(a) **`msg`** is defined in the data section and will be static data with an absolute address. Thus, it's address will be unknown until the linker stage. **`msg`** will be defined in the symbol table and passed to the relocation table.
(b) **`loop`** is defined at address **`0x14`** and is defined in the symbol table.
(c) **`n`** is defined at address **`0x24`** and is defined in the symbol table.
(d) Lastly, the assembler encounters **`end`** defined at **`0x2C`** which is then defined in the symbol table.

4.3 After the first pass of the assembler, which of the instructions do not have their addresses fully resolved?

Answer:
(a) **`beq t0, t1, end`**
(b) **`la a0, msg`**
(c) **`jal ra, printf`**

At the end of the first-pass, all instructions with forward references (positive PC-relative offsets) are still unknown. This is the case for **`beq t0, t1, end`**. Additionally, any absolute addresses or links to external library functions are unknown and cannot be translated until the linker stage (**`jal ra, printf`** and **`la a0, msg`**).

$\boxed{4.4}$ After the second pass of the assembler but before the linker, which of the instructions do not have their addresses fully resolved?

Answer: **`jal ra, printf`** and **`la a0, msg`**

The address of the label **`end`** is known in the symbol table at the end of the assembler's first pass. On the second pass, the address is resolved and the instruction can be translated to machine code.

`printf` is an external library function whose address is unknown and is determined by the linker. Additionally, **`msg`** will be stored as absolute address (not PC-relative) at an address determined by the linker, and thus **`la a0, msg`** cannot be translated at this stage.