Combinational Circuits I [1][2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

After having built simple gates and chips, we now look at **combinational** circuits to perform arithmetic and logical operations on integer numbers. In such circuits, the output is purely dependent on the input combinations.

Next to such circuits, **sequential** circuits exists as well, where the output depends not only on the current inputs but also on some previous state or memory.

The topic of combinational circuits is covered in two presentations, of which this is the first one.

1 Introduction

Let us begin with a brief look at the core of our topic and its learning objectives, followed by a recap.

1.1 Today's Core Question

- How can we perform arithmetic and logic operations on integer numbers, starting from Boolean logic?
  - Based on Chapter 2 of (Nisan and Schocken 2005)

Our goal is to perform arithmetic and logic operations on integer numbers, starting from Boolean logic.

1.2 Learning Objectives

- Compute 2's complement of a binary number and use it for addition and subtraction
- Build, test, and analyze combinational circuits leading to the Hack ALU (Project 2)
  - Half and full adder, Ripple-Carry Adder (`Add16`)
  - Incrementer (`Inc16`)
  - `ALU`
- Determine ALU operation based on control bits

You will learn how to add and subtract binary numbers. Notably, you learn to compute the 2's complement of a positive number to represent the negative of that number. Afterwards, you can compute addition and subtraction using traditional addition, where we add digit by digit.

You implement combinational circuits for such operations, in particular the ALU of our Hack computer. For the ALU you will then be able to determine its computation based on given control bits.

1.3 Retrieval Practice

- Prior knowledge
  - How do you add two numbers with pen and paper, say 4242 + 6789?

Please take a brief break and add two decimal numbers here. Quite likely, in elementary school you learned how to do so with pen and paper, writing the numbers underneath each other and adding them digit by digit.

  - What is 101010 in decimal?

Convert the shown binary number to decimal.

- What is a multi-bit chip?

Recall what a multi-bit chip is.

[1] This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.

[2] Material created by Jens Lechtenbörger; see end of document for license information.

Agenda

- Part 1
  - Introduction
  - Addition
  - 2's Complement
  - Building an Adder Chip
- Break for self-study
- Part 2
  - Hack ALU
  - Project 2
  - Conclusions

The material for combinational circuits is as follows.
After this introduction, we discuss the addition of binary numbers: We present 2's complement to represent signed binary numbers of a fixed length. Based on this background, we delve into the implementation of adder chips.
Then, you might want to take a break and apply covered techniques.
Afterwards, in part 2, we look at the Hack ALU, which is the major part of Project 2 of Nand2Tetris.

2 Addition

Let us analyze addition.

2.1 Pen-and-paper method

We add digits from **right to left**, possibly with **"carry"**

- In decimal

```
          4242
      +   6789
Carry:   11110
         -----
         11031
```

- In binary

```
          10110111
      +   11000101
Carry:   100000110
         ---------
         101111100
```

This slide shows how to add numbers digit by digit from right to left. Please verify the calculations and convince yourself that the method works for numbers of any base. Subsequently, we are interested in base 2 for binary numbers.

- Note
  - Rightmost (lowest) position with carry 0
    - Only need to add **2 bits**
  - Other positions need to add **3 bits**

For the rightmost position, we just need to add two digits, while other positions may have a nonzero carry. Such a carry needs to be added as third digit.

- Both sample results are 1 digit **larger** than the inputs
  - **Overflow** in case of fixed digits/bits; incorrect result if discarded

The result of an addition might need more digits than its inputs. If we use a fixed number of digits, the result might be too large to be represented. Then, an **overflow** occurs, and the result is incorrect. We expand on that thought next.

2.2 Adding with fixed bits

- Binary addition of previous slide
  - $10110111 + 11000101 = 101111100$ $(= 380)$
  - Suppose restriction to 8 bits
    - Largest number is $2^8 - 1 = 255$
    - 8-bit-result (drop leading 1 and 0): 1111100 $(= 124)$
      - Incorrect, **overflow**, **bug**
      - Examples at Wikipedia

Here you see how the result of our calculation would be incorrect if numbers were restricted to 8 bits. In general, it is the programmer's responsibility to be aware of potential size limitations and to check for overflows; otherwise, the program would be buggy.

See Wikipedia for examples if you are interested. Note that the icon of the hyperlink here indicates that this link goes beyond class topics.

- Insight (recall Fail task)
  - Restricting to n bits corresponds to **Modulo operation**
    - mod 2^n
  - $n = 8$: 10110111 + 11000101 = 1111100 mod 256

Please convince yourself that restricting a number to n bits is the same as computing the modulo operation with 2^n.

Note that in this case, the icon of the hyperlink indicates essential information. Thus, please follow such links if you cannot explain their topics.

3 2's Complement

2's complement is one popular way to define negative numbers.

3.1 Definitions

- Consider n-bit number k
  - $k_c = 2^n - k$ is **2's complement** of k (and vice versa)
    Given an n-bit number k, we say that 2^n - k is the **2's complement** of k.
    - Note: $k + k_c = 2^n = 0 \mod 2^n$, i.e., complementary numbers add up to zero
    - E.g., $n = 4$, $2^4 = 16$: 1 and 15 are complements of each other; 8 is complement of itself

    In other words, 2's complement defines n-bit numbers to be complements of each other if their sum is 2^n.

    Again in other words, their sum is 0 modulo 2^n.

    E.g., with 4 bits, 1 and 15 are complements of each other as they add up to 16.
  - If most significant bit of k is 0, interpret as (usual) **positive** number
  - If most significant bit is 1, interpret bit pattern as **negative** number: $k = -k_c$
    - E.g., $15 = (1111)_2$ and $8 = (1000)_2$ are interpreted as negative numbers under 2's complement: $(1111)_2 = -1$ and $(1000)_2 = -8$

    Recall that the leading bit of a number is the most significant one. If the most significant bit of a number is 0, we interpret the bit pattern as ordinary, positive binary number.

    If the most significant bit of a number is 1, we interpret the bit pattern as negative number under 2's complement. Namely, we interpret such a bit pattern as negative of its complement. This interpretation makes sense as both numbers (in the usual interpretation) add up to 0 as remarked above.

    You see examples here.

3.2 Example and Conversion

- 2^n signed numbers between -2^{n-1} and $2^{n-1} - 1$
  Under this interpretation, with n bits we represent 2^n signed numbers in the range shown here.
  - E.g., $n = 4$

0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

  E.g., with 4 bits under 2's complement, these numbers arise.

- Note
  - Positive numbers start with 0
    - Usual binary number
  - Negative numbers start with 1
  - To convert a number
    - Leave all trailing 0's and first 1 intact, flip all remaining bits or
    - Flip all bits and add 1

Thus, positive numbers start with 0 and are our usual binary numbers. In contrast, negative numbers start with 1. To convert a number, two methods are shown here.

Maybe the second method is easier to remember: Flip all bits and add 1. Please try this out on your own.

3.3 Addition with 2's Complement

- We just add bit patterns, mod 2^n
  - No need to treat negative numbers specially
  - **Subtraction** as addition with 2's complement
    - Example

```
2-5 = 2 + (-5):  0010
                + 1011
                  ----
                  1101 = -3
```

  - In Hack, overflows will be ignored
    - Programming **bug**

Thanks to the design of 2's complement, when adding numbers, we do not need to watch out for the sign. We can just add numbers bit by bit, and subtraction arises automatically when adding a negative number.

An example is shown here.

Importantly, overflows can occur. If that happens, our Hack hardware will silently ignore this. As mentioned earlier, it is the programmer's task to avoid such bugs. In fact, other hardware may have methods to signal overflows, and as an exercise we could also add an overflow output to the Hack ALU.

4 Building an Adder Chip

- Sequence of chips
  - Increasing complexity
    1. `HalfAdder`: Adds two bits, produces two output bits
    2. `FullAdder`: Adds three bits, produces two output bits
    3. Ripple-Carry-Adder, `Add16`: Adds two 16-bit numbers, produces 16-bit output

We now turn to the construction of chips for addition. We do so in a modular fashion from half adders that can add two bits, over full adders that add three bits, to adder chips for numbers of arbitrary fixed length.

4.1 Half Adder

a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The half adder adds two bits and produces two bits, namely the sum and a potential carry. Note that carry and sum can be understood as 2-bit number that represents the result. E.g., in the final row, we add two 1s, which is 2, represented as binary number 1 0.

Figure 1: Figure under CC0 1.0

- Several implementations possible
  - DNF with Not, And, Or
  - Alternative based on two gates
    - `sum = Xor(a, b)`
    - `carry = And(a, b)`

You can implement the half adder in several ways. You might start from the DNF or you use the alternative based on two gates shown here. Please convince yourself that these two gates reflect the truth table.

4.2 Full Adder Specification

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The full adder adds three bits and produces two bits, again the sum and a potential carry. Note again that carry and sum can be understood as 2-bit number that represents the result. E.g., in the final row, we add three 1s, which is 3, represented as binary number with two 1s.

Figure 2: Figure under CC0 1.0

- Several implementations possible
  - DNF with Not, And, Or
  - Based on two Half Adders, see next slide

While your implementation could start from the DNF, a usual full adder implementation is based on two half adders as shown next.

4.3 Full Adder Implementation

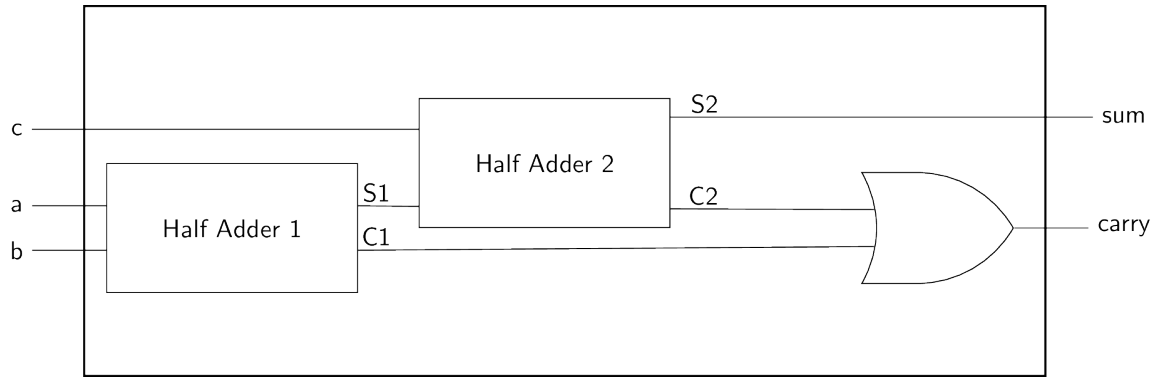

Figure 3: Full Adder based on Half Adders (Figure under CC0 1.0)

- Internal names
  - C1 = And(a, b)
  - S1 = Xor(a, b)
  - C2 = And(S1, c) = And(Xor(a, b), c)
  - S2 = Xor(S1, c) = Xor(Xor(a, b), c)
- Outputs: carry = Or(C1, C2); sum = S2

Note how a full adder can be built from half adders.

If you recall the truth table, then the `sum` is one if an odd number of 1s are added. This is achieved by an exclusive or of the three inputs.

The `carry` is supposed to be 1 if at least two inputs are 1. Here, this is the case if inputs a and b are 1 or if c is 1 in addition to either of them.

4.4 Ripple-Carry 4-bit Adder

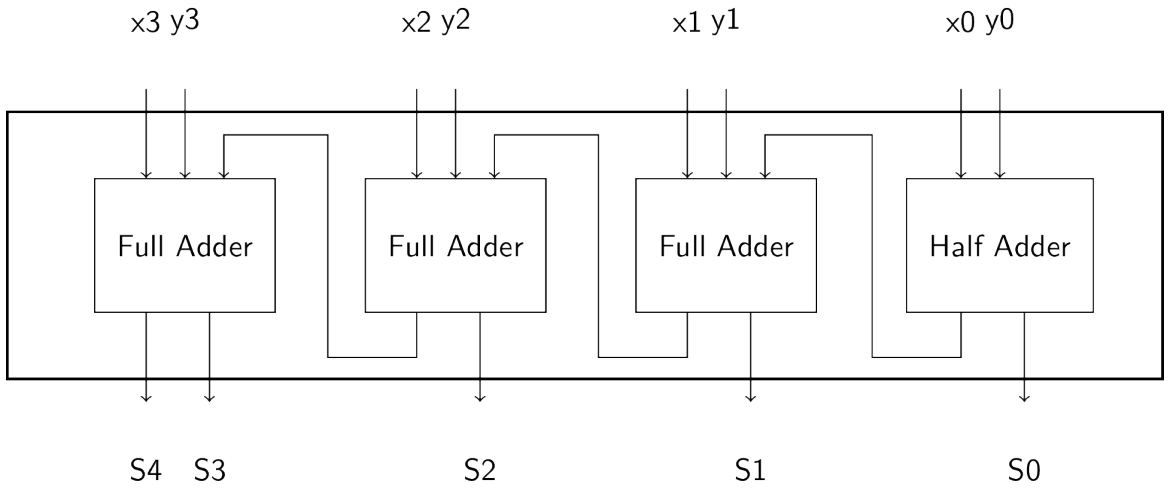

Figure 4: Figure under CC0 1.0

- Add 4-bit numbers $x = x_3x_2x_1x_0$ and $y = y_3y_2y_1y_0$
  - Produce 5-bit number $S = x + y = S_4S_3S_2S_1S_0$
  - (Or ignore/omit S_4 to produce 4-bit number)

By chaining 1-bit adders one after the other, we can build adders for any number of bits. Here you see an example for a 4-bit adder. As observed earlier, the rightmost adder can be a half adder as only two digits need to be added without carry.

The construction here is called **ripple-carry** adder as the carry bit may propagate, or ripple, from the right all the way to the left. This causes some **propagation delay** before the computation is completed, and several adder optimizations exist to speed up the process.

However, such optimizations are beyond our topics.

Regarding the number and naming of outputs, there is some freedom. The adder shown here produces 5 output bits, of which the final one can also be understood as carry bit. Alternatively, the fifth bit may be omitted entirely, to produce again a 4-bit number. As observed previously, with less than 5 output bits, overflows may occur, which lead to incorrect results.

Note that the adder works independently of any assumption regarding the interpretation of resulting bits. The adder can be understood to add positive numbers (without overflows), or we may interpret inputs and output as numbers under 2's complement (which requires to ignore the fifth bit).

In class, we may build a ripple-carry adder for 16-bit addition. That adder has two 16-bit inputs and produces one 16-bit output, all of which are interpreted as numbers under 2's complement (potentially leading to overflows).

5 Self-Study Tasks

Maybe take a break.

5.1 Negative Numbers

- What is -42 as 7-bit number in 2's complement?

To support your learning, pause here to work on the given task.

5.2 Recall Adders

- What parts make up a half adder? A full adder? A ripple-carry adder?

Draw the various adders.

Bibliography

Nisan, Noam, and Shimon Schocken. 2005. *The Elements of Computing Systems: Building a Modern Computer from First Principles.* The MIT Press. https://www.nand2tetris.org/.

The bibliography contains references used in this presentation.

License Information