Combinational Circuits II *

Jens Lechtenbörger

IT Systems, Summer Term 2024

This is the second presentation on combinational circuits.

1 Introduction

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

In part 1, basics of addition were introduced.
In part 2, we discuss the Hack ALU, which is the major part of Project 2.

2 Hack ALU

The Hack ALU will be the brain for our Hack Computer. Let us see what is involved.

2.1 ALU Design

- **Arithmetic Logic Unit** (ALU)
  - Computes fixed set of functions
    - Arithmetic functions, e.g., x + y, x - y, x + 1
    - Logical functions, e.g., or(x, y), and(x, y)
  - Specific function determined by **control bits**
  - **Programming** the ALU means setting the control bits

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

An ALU is able to compute a fixed set of functions, in particular arithmetic functions, such as addition and subtraction, and logical functions.

At any point in time, a number of control bits determine what the ALU is supposed to do. If we want to program our ALU, we have to make sure to set the correct control bits for our purposes.

2.2 Hack ALU Overview

- Pins

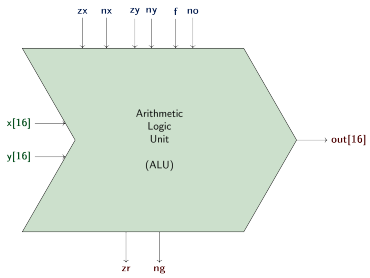

Figure 1: Figure under CC0 1.0

  - Two 16-bit input numbers
  - One 16-bit output number
    - And two output bits
      - Is output zero (zr)?
      - Is output negative (ng)?
  - Six control bits
    - For 2^6 possible operations; 18 actually specified in (Nisan and Schocken 2005)
    - **Important symbols** in ALU context
      - Arithmetic operations: x+y, x-y, -x
      - Logical operations: x&y (and), x|y (or), !x (not)
      - Thus, "+" denotes addition, not the logical Or

The Hack ALU operates on two 16-bit inputs and produces a 16-bit output as well as two output bits. The output bits simply indicate whether the 16-bit output is zero or negative.

The ALU's operation is determined by 6 control bits, for a total of 32 possible operations. Out of those 32 operations only a subset of 18 operations are documented in our textbook.

Importantly, in the ALU context, we use the plus and minus symbols for addition, subtraction, and negative numbers. In contrast, ampersand, vertical bar, and exclamation mark denote logical operations. See also the next slides.

2.3 Hack ALU Operations

- Bit-wise logical operations on 16-bit numbers, e.g.:
  - `And16`: 1010101010101010 & 0000000011111111 = 0000000010101010
  - `Not16`: !1010101010101010 = 0101010101010101
    - Negate/invert/flip all bits

- `Add16`: Addition of 16-bit numbers x and y (in 2's complement)

Here you see typical operations of the Hack ALU, which point at chips that might be used as parts. As we will see shortly, other operations can be reduced to the ones shown here.

2.4 Negation in ALU Context

- -x has no special meaning: -x = (-1)*x
  - Using 2's complement

Subsequently, the minus sign is used for subtraction and negative numbers as usual.

- !x is **bit-wise negation** of x (see previous slide)
  - In the following, "negate" means "flip all bits"
  - Fact: !x = -x - 1 (in 2's complement)
    - Proof:
      - !x + x = 1...1 (all bits 1, as each bit that is 1 in x is 0 in !x and vice versa) = -1
      - Subtract x on both sides

As already explained, the exclamation mark indicates the negation of a number by flipping all its bits. You see here that flipping all bits turns x into -x - 1. Please take note of this fact.

- Fact: x - y = !(!x + y)
  - Proof: !(!x + y) = -(-x - 1 + y) - 1 = x - y

As shown here, by using negation twice we can express subtraction.

2.5 Hack ALU Chip

```
zx = zero   x
nx = negate x
zy = zero   y
ny = negate y
f  = apply function, either add or a
no = negate output

zr = out zero
ng = out negative
```

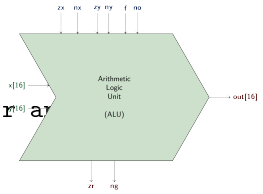

Figure 2: ("Hack ALU" under CC0 1.0; from GitLab)

The control and output bits of the ALU have specific meanings, which are spelled out here and defined on the next slide.

2.6 Hack ALU Specification

```
 * [Excerpt of ALU.hdl from Nand2Tetris]
 * Computes out = one of the following functions:
 *     0, 1, -1, x, y, !x, !y, -x, -y,
 *     x + 1, y + 1, x - 1, y - 1, x + y, x - y, y - x,
 *     x & y, x | y
 * [...]
```

```
// Implementation: Manipulates the x and y inputs
// and operates on the resulting values, as follows:
// if (zx == 1) sets x = 0        // 16-bit constant
// if (nx == 1) sets x = !x       // bitwise not
// if (zy == 1) sets y = 0        // 16-bit constant
// if (ny == 1) sets y = !y       // bitwise not
// if (f == 1)  sets out = x + y  // integer 2's complement addition
// if (f == 0)  sets out = x & y  // bitwise and
// if (no == 1) sets out = !out   // bitwise not
```

This excerpt of the HDL file for the ALU defines its operations and control bits.

Importantly, in the specification with if statements, the order matters: Those statements are processed from top to bottom. First, if the zero bit of an input is 1, replace that input with the constant 0. Afterwards, if the negate bit for that input is 1, negate the bits of the value obtained so far. In particular, if both bits are set, first set the input to 0, then negate its bits, leading to the value of 16 1s, which is -1 in 2's complement. This happens for both inputs.

Subsequently, depending on the function bit, either add both values, or combine their bits with logical And.

The final output value is determined by the negate output bit: If that bit is 0, the value computed so far is the final output. Otherwise, negate the bits obtained so far to produce the final output.

Besides, as mentioned already, two additional output bits indicate whether the computed output is zero or whether it is negative.

2.7 ALU Logic Examples

- Table in Figure 2.6 of (Nisan and Schocken 2005) documents ALU control bits
  - Let us look at some rows of that table

Out textbook documents which control bits implement what ALU functions. Let us look at some examples.

2.7.1 Row 14, x+y

zx	nx	zy	ny	f	no	out
0	0	0	0	1	0	x+y

- zx=nx=zy=ny=0: Neither zero nor negate inputs
- f=1: Perform addition
- no=0: Do not negate output

Let us start with a simple example, where only the function bit is 1. In that case, the inputs are used as they are, without any zeroing or negation. According to the Hack ALU Specification, a function bit of 1 leads to an addition. As the negate output bit is not set, the resulting sum of x and y is the final output.

Note how a row with all 0s would similarly compute the logical And.

2.7.2 Row 15, x-y

zx	nx	zy	ny	f	no	out
0	1	0	0	1	1	x-y

- As explained previously: x - y = !(!x + y)
- Thus
  - zx=0, nx=1: Negate x
  - zy=0, ny=0: Keep y unchanged
  - f=1: Add
  - no=1: Negate result
- Self-study: How to implement logical Or? (De Morgan!)

We observed previously how to subtract y from x by suitable use of negation and addition. The row here implements that insight.

Please try to come up with bits that implement the logical Or operation. Towards that goal, recall the De Morgan rules, which involve negation to transform between logical Or and And.

If you are not sure, verify your result in our textbook.

2.7.3 Row 5, y

zx	nx	zy	ny	f	no	out
1	1	0	0	0	0	y

- zx=1, nx=1: Zero, then negate x: x $\rightarrow 00\ldots0 \rightarrow 11\ldots1$
- zy=0, ny=0: Keep y unchanged
- f=0: Perform bit-wise And
  - All 1s of zero-negated x preserve y
- no=0: Keep result

In this case, we just want to keep the input y. We can do so as follows: Zero and negate x to replace x with all 1s. Then, use y in unchanged form. Finally, apply bitwise And, which preserves all bits of y.

2.7.4 Row 2, 1

zx	nx	zy	ny	f	no	out
1	1	1	1	1	1	1

- zx=1, nx=1: Zero, then negate x: x $\rightarrow 00\ldots0 \rightarrow 11\ldots1$
  - All bits one is -1 in 2's complement
- zy=1, ny=1: Zero, then negate y: y $\rightarrow 00\ldots0 \rightarrow 11\ldots1$
  - All bits one is -1 in 2's complement
- f=1: Add
  - (-1) + (-1) = -2 = $1\ldots10$
- no=1: Negate output, $0\ldots01 = 1$

The ALU can also produce some constants. Here, you see how to produce 1.

2.7.5 Row 10, x+1

zx	nx	zy	ny	f	no	out
0	1	1	1	1	1	x+1

- zx=0, nx=1: Negate x, !x
- zy=1, ny=1: Zero, then negate y, -1
- f=1: Add
  - !x + (-1) = !x - 1
  - Since !x = -x - 1, we have !x - 1 = -x - 1 - 1 = -x - 2
- no=1: Negate output, !(-x - 2) = -(-x - 2) - 1 = x + 1

The implementation of an increment operation is shown here. It is surprisingly complex.

2.7.6 Self-Study

zx	nx	zy	ny	f	no	out
0	1	0	1	1	1	?

What operation is performed by the Hack ALU for the above control bits?
Based on the previous explanations, determine what the Hack ALU does in the case shown here. Note that this operation is not among the officially documented ones.

As a hint, you can invoke ALU functions in the hardware simulator: Just load the ALU from the builtin directory, set the control bits and try out some input values.

3 Project 2

- Build
  - Adders, in particular `Add16`
  - `ALU`
  - `Inc16`, an incrementer

As part of Project 2, you build adders, the ALU, and an incrementer, which just adds 1 to an input number.

3.1 Notes on ALU

- What parts to use? How?
  - Clearly, use `Add16`, `And16` to implement main functions
  - Process x and y in parallel for zeroing and negation
    - At least 2 chips `Not16`
  - Use at least one `Mux16` for each if statement, e.g.:
    - Initially, use zx: `Mux16(a=x, b=false, sel=zx, out=zeroX);`
      - (Note next slide for `b=false`)
    - Finally, use f: `Mux16 (a=xAndY, b=xPlusY, sel=f, out=xOpY);`
      - (In between, how would you compute `xAndY` and `xPlusY`?)

The bullet points here are meant to get you started with the ALU.

3.2 HDL Tips

- If you need constant numbers, use `true` and `false`, e.g.:
  - `a = false` and `a[0..15] = false` assign 0 to `a`
  - `a[0] = true, a[1..15] = false` assigns 1 to `a`
- You can give names to parts of a word, e.g.:
  - `out[0..7] = outlow`
  - Then use `outlow` elsewhere as input
- The chip's output `out` cannot be used as input in the same chip
  - Assign new names if that should be necessary, e.g.:
    - `out = out, out = othername`
    - Then use `othername` elsewhere as input

This slide lists some tips regarding HDL. Note that our textbook contains an appendix on that language.

4 Conclusions

Let us conclude.

4.1 Summary

- Binary adders from half and full adders
  - Use 2's complement for negative numbers and subtraction
  - Ripple-carry adder for any number of bits
- Hack ALU uses 6 control bits to specify operation
  - Simple operations, e.g., no multiplication
  - Build it as part of Project 2!

We now know how to represent signed 16-bit numbers using 2's complement, and how to add and subtract such numbers. In particular, ripple-carry adders work by chaining a fixed number of single bit adders to simulate a pen and paper method for addition of numbers.

The Hack ALU provides simple arithmetical and logical operations, which are controlled by 6 bits. Your task is to build it as part of Project 2.

4.2 Q&A

- Please ask questions and provide feedback on a regular basis
  - Something confusing?
    - What did you understand? Where did you get lost?
  - Maybe suggest improvements on GitLab
    - Did you create exercises, experiments, explanations?
  - Use Learnweb: Shared, anonymous pad and MoodleOverflow

This slide is a copy of an earlier one. It serves as reminder that I am happy to obtain and provide feedback.

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