Nand2Tetris *

Jens Lechtenbörger

IT Systems, Summer Term 2024

In the first part of our course, you will build a computer in a software simulator, which is part of the project Nand to Tetris. As the name suggests, you start from a simple logic gate that implements the `Nand` function. From that starting point, you build successively more complex gates and circuits until you arrive at a general-purpose, programmable computer that can run games such as Tetris.

1 Introduction

Video by creators of Nand to Tetris: `https://yewtu.be/watch?v=wTl5wRDTOCU`

In this video, the creators of Nand to Tetris, Professor Nisan and Professor Schocken, explain the project.

1.1 Today's Core Question

- How to start with Nand to Tetris?

This presentation serves as starting point for the projects of Nand to Tetris, which form the basis for the first part of IT Systems.

1.2 Learning Objectives

- Build simple chips with HDL
- Simulate chips in Hardware Simulator

Please recall the importance of learning objectives.

In general, tasks in class and quizzes in Learnweb will support your learning towards these goals. Please do not hesitate to ask if questions arise.

1.3 Retrieval Practice

- What do you remember about Part 1 of IT Systems?
- Other parts?

Questions on retrieval practice slides are meant to be answered by you before you continue. They offer learning opportunities and stimulate meta-cognitive processes.

What do you remember about Part 1 of IT Systems?

What else will be covered in IT Systems?

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

1.4 Recall: Goal of Part 1

- Recall: Build **general-purpose**, **programmable** computer; here, the **Hack** platform

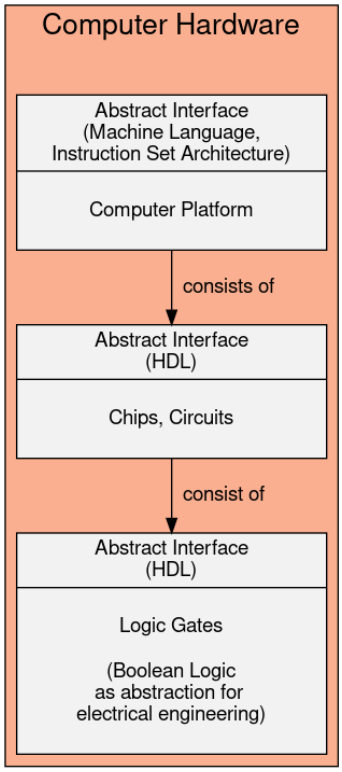

Figure 1: Computer hardware with layers of abstraction

  - Programmable in machine language
  - Built from chips and gates specified in hardware description language (HDL)
    - (Gates are "simple" chips)
  - Nand2Tetris with sequence of projects

You already saw that we aim to build a programmable computer, starting from logic gates. We do so with projects of Nand to Tetris.

Agenda

The agenda of this presentation is as follows. After this introduction, we next turn to a preview of the Hack computer architecture to get a feeling for the goal of Nand to Tetris. Then, we look at typical ingredients of Nand to Tetris projects, before we build a first simple chip, namely the `And` gate. Subsequently, we point out important project resources. As usual, conclusions end the presentation.

2 Preview

Let us start with a short preview of our target computer architecture.

2.1 Hack Machine Language

- Hack **CPU** executes instructions in machine language

  The brains of computers reside in processing units, or processors for short. In case of Hack there is a single central processing unit, or CPU. This CPU continuously executes instructions. Each instruction is just a certain bit pattern, which is interpreted by the CPU to determine what operation to perform on what operands.

  - Each Hack **instruction** consists of 16 bits
    - Consider increment of variable: `i = i + 1`
    - Translated into two Hack instructions
    - `0000000000010000` (A-instruction, starts with 0, binary number for 16)
      - Human-readable assembly language instruction: `@16`
      - Used here to specify memory (RAM) location for value of variable
    - `1111 110111 001 000` (C-instruction, starts with 1, encodes operation)
      - Human-readable assembly language instruction: `M=M+1`
      - 6 red bits specify operation (compute `M+1`), 3 blue bits specify where to store the result (in `M`, the memory location for our variable)

    Here, you see an example how the increment of some variable might be represented by two instructions in the Hack machine language. Notably, different parts of the instruction's bit pattern serve different purposes.

    You can ignore the details for now.

  - Modern processors behave similarly
    - Later: Fetch-Decode-Execute Cycle
      - Fetch next instruction from memory, decode bits to determine operation and operands, execute
      - **Von Neumann architecture** (von Neumann 1945)

    Similarly to the case of Hack, also modern processors execute architecture-specific machine instructions, which are fetched from RAM, decoded, and then executed.

    This computer architecture is known as **von Neumann architecture**, following a seminal description by John von Neumann in 1945.

2.2 Preview: Hack Computer Architecture

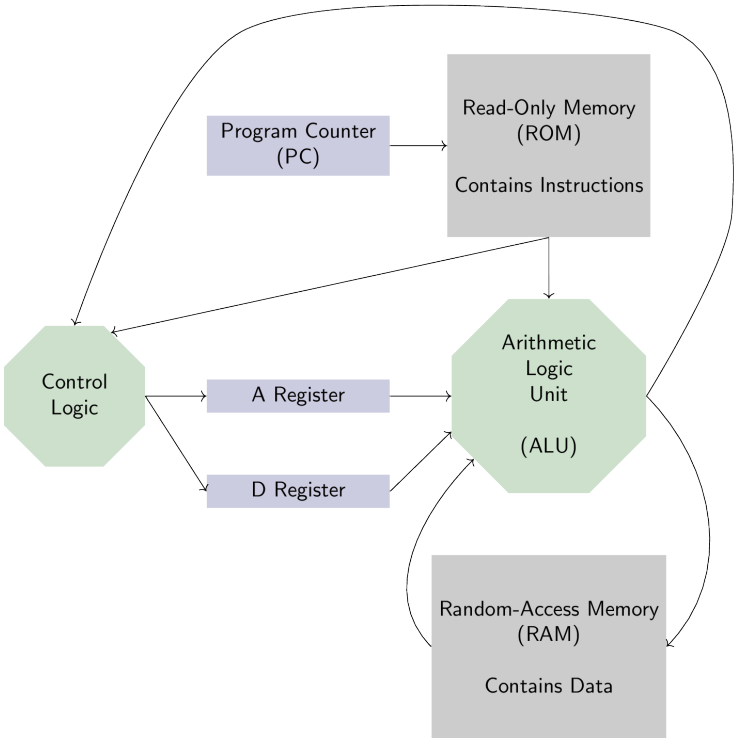

Figure 2: "Hack Computer Architecture" under CC0 1.0; from GitLab

This figure shows major elements of the Hack computer architecture, which will be explored in great detail in upcoming weeks.

In Hack, two types of memory are distinguished: **Random-Access Memory**, or RAM for short, and **Read-Only Memory**, ROM for short. Random-access means that data can be read and changed in any order, at the same speed irrespective of the location, while read-only means that such memory receives its contents upon fabrication, which can only be read, again in any order, but not be changed. Data items in memory are equal-sized bit strings, e.g., of length 8 bits to form **bytes**, or **words** of 16 bits in Hack. **Addresses** specify which data item to read or write.

Octagons visualize active components, in particular the **Arithmetic Logic Unit**, ALU for short, which performs computations and is the major part of the computer's CPU. The ALU executes operations that are specified by instructions which in turn are retrieved from ROM. Depending on the specific instruction, a control logic determines on what operands the ALU operates, including values stored in registers or in RAM, and if and where to store a result.

In addition, **registers** are small storage devices whose size and number depends on the architecture of the computer. In Hack, the following three registers exist, each of which contains a word of 16 bits:

First, the **program counter** stores the address of the next instruction to execute. Thus, this register is wired to the ROM's address input.

Second and third, the registers **A** and **D** can store addresses as well as inputs for and results of ALU operations.

Similarly to Hack, modern computers contain processing units, registers, and memory. Differently from Hack, RAM stores not only data but also instructions. Thus, in our computers, we can simply load new programs into RAM to execute them, while in Hack ROM

chips would have to be replaced. In fact, out computers follow the model of the von Neumann architecture mentioned previously.

3 Projects of Nand2Tetris

Let us see in more detail what to expect in the Nand to Tetris projects.

3.1 What to Expect in a Project

- Projects focus on successively more complex chips

  Our goal is to build a digital computer in a sequence of projects, which aim to build successively more complex chips.

  - From simple logic gates to entire computer

    We start from a simple logic gate for the Boolean function `Nand`, and we end at a programmable computer.

  - Digital/binary devices with **bit** as basic unit of information
    * Bit = **Logical state** with two values: true/false or 1/0
    * Data and instructions to be represented as sequences of bits

    Boolean logic is a suitable starting point for building computers as we consider electrical devices starting from two basic states, namely electrical current being present or not. These two states can be interpreted a a single **bit** of information, carrying either a Boolean value true or false, or alternatively an Integer value 1 or 0.

    While we first manipulate individual bits based on logical operations, we later see how to encode numbers and machine instructions with sequences of bits, for which you saw a sketch on an earlier slide.

  - Chips come with **specifications/interfaces** (and tests)
    * **What** to do. **Abstraction**!

    All the chips will come with **specifications and interfaces** specifying **what** each chip is supposed to do. Your task is to come up with an **implementation** specifying **how** to perform those operations. Clearly, those specifications provide a level of **abstraction** over the implementation.

  - Results/implementations for each project are building blocks for subsequent ones
    * **Abstraction and modularity**!

    Then, you will use your implementations in subsequent projects as **building blocks** for even more complex pieces of the computer. Again, this provides a level of **abstraction** and also **modularity**.

    Thus, we break down the complex task of building a computer into more manageable modular units.

  - You work on projects individually
    * Without immediate impact on grading, but lasting impact on brains (and, thus, on exam results)
    * (Solutions can be found on the web)

    As a side note, you work on those projects individually, and we do not look at your results. Thus, the project work does not have an immediate impact on grading, but it will have a lasting impact on your brain, and therefore also on the grading in the final exam. In case you get stuck, note that you will be able to find solutions out there on the web.

3.2 What to Expect in Project 1

- Project 1

1. Given: `Nand(x, y)`, `false`

x	y	Nand(x, y)
0	0	1
0	1	1
1	0	1
1	1	0

   - `Nand` is a binary **Boolean function**
     - **Specified** by **truth table** to right
     - `0` represents `false`, `1` represents `true`

   In Project 1, we start from `Nand` and `false`. Thus, we suppose that a `Nand` gate as well as the constant `false` are given to us. In reality, gates are usually fabricated from different types of transistors, which is beyond class topics. Also, we do not consider quantum computing.

   Here you see the notation `Nand` of x and y, which indicates that `Nand` can be understood as binary function, i.e., as function that takes two arguments. As those arguments are interpreted as Boolean values, `Nand` is a binary Boolean function.

   Boolean functions can be specified by truth tables, such as the one shown here. Truth tables specify what output value is computed by the function for each of the possible input value combinations.

   The truth table for `Nand` contains four rows because `Nand` is a function with two arguments, each of which can have either of two values, for a total of four combinations. We see that `Nand` of x and y results in 1 (or True) unless both arguments have the value 1 (or True).

2. Learn about Boolean logic

   Later on, you will learn about Boolean logic and build successively more complex logic gates and chips, starting from `Nand`. Some possible steps are outlined next.

3. Implement sequence of chips (from specifications)

   (a) `Not(x)`
   - `Not` flips its argument: `true = Not(false)`, `false = Not(true)`
   - Implementation: `Not(x) = Nand(x, x)`

   First, using just `Nand`, we can build a chip for the function `Not`, which takes a single Boolean value as input argument and flips it, i.e., `Not` turns 1 into 0 and vice versa.

   Please take a moment to convince yourself that `Not` of x equals `Nand` of x with itself.

   Besides, `Not` can compute `true` from `false`, which is why we can say that we start our journey towards a computer only from `Nand` and `false`.

   (b) `And`
   - `True` if and only if all arguments are true; revisited subsequently

   Having a chip for `Not` in addition to `Nand`, it makes sense to build a chip for the `And` function, which only produces 1 as output if all input arguments are equal to 1.

   (c) `Xor`, `Or`, `Mux`, and several more...

As next steps, more chips can be built, including additional Boolean functions and so-called multiplexors.

Project 1 asks you to implement several chips, and subsequent slides provide an outlook of the general approach using examples of the `And` gate.

4 And Gate

Let us now look at an implementation of the And gate, starting from its specification.

4.1 Specification of And Gate

- Three files for binary function `And(a, b)`

  Chips in Nand To Tetris typically come with three files as follows.

  1. HDL file with interface and specification, `And.hdl`

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/01/And.hdl

/**
 * And gate:
 * out = 1 if (a == 1 and b == 1)
 *       0 otherwise
 */

CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
}
```

  Most importantly, an HDL file describes the interface of a chip in terms of input and output pins, here starting at line 12. In addition, a comment in lines 6 to 10 specifies the expected behavior. Here, the output should be 1 exactly if both inputs are 1.

2. File `And.cmp`

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

   – Test cases (here, truth table)

Second, test cases specify what the chip under construction should compute for given inputs.

3. File `And.tst`

   – Test file for hardware simulator

   – To check implementation against test cases

Finally, a test script can compare expected results against actually computed values.

4.2 Implementation of `And` Gate (1/3)

- **What** an `And` gate should do
  - Specification: `out == 1` if and only if `a == b == 1`

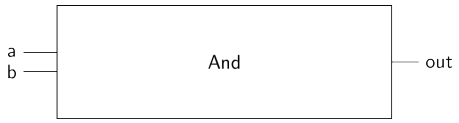

Figure 3: Figure under CC0 1.0

  - Interface in HDL skeleton: `And.hdl`

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
// Put your code here:

}
```

Our goal is to create a chip with two inputs and one output that satisfies the specification shown here. Please verify on your own that this specification is equivalent to the truth table of the previous slide.

With Nand To Tetris, specifications are given as comments in the chip's HDL skeleton, which also specifies the interface in terms of input and output pins.

4.3 Implementation of `And` Gate (2/3)

- **Idea** to implement `And` gate
  - **Gate logic**: `And(a, b) = Not(Nand(a, b))`

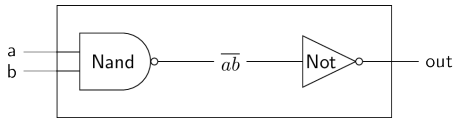

Figure 4: Figure under CC0 1.0

  - Interface in HDL skeleton: `And.hdl`

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
// Put your code here:

}
```

We build gates and more complex chips starting from previously built ones. As stated on an earlier slide, the `Not` gate should be our starting point. Building on `Nand` and `Not`, here we see a logical formula to express `And`.

Please convince yourself that this formula is correct.

Besides, we see how we can wire a `Nand` gate with a `Not` gate to fill the previous black box for the `And` gate.

4.4 Implementation of `And` Gate (3/3)

- **Implementation** of `And` gate
  - Gate logic: `And(a, b) = Not(Nand(a, b))`

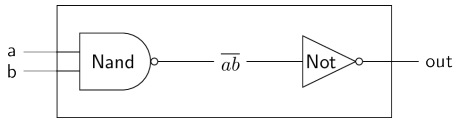

Figure 5: Figure under CC0 1.0

  - **Completed** HDL file: `And.hdl`

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
Nand(a = a, b = b, out = aNandB);
Not(in = aNandB, out = out);
}
```

Finally, we write down the wiring of `Nand` and `Not` in terms of HDL.

Note that we can give arbitrary names to internal pins, for example, we can assign a name of our choice to the output of `Nand`. Then, we use that name as input for `Not`. That way, `Nand` and `Not` are connected.

In contrast, the output of `Not` is given the name `out`, which is also the output of the overall `And` gate. That way, we say that the output of `Not` is really the output of the overall gate.

5 Nand To Tetris Resources

- Download Java software: `https://www.nand2tetris.org/software`
  - ZIP archive, contains tools and project files
    - I had to do this in the `tools` subdirectory: `chmod u+x *.sh`
  - We start with the Hardware Simulator
  - (Later on, we also use the CPU Emulator)

Download the software now. Note the installation instructions, which also explain that you may need to set proper permissions on executable files.

5.1 Hardware Simulator

- Previous `And` implementation in Hardware Simulator

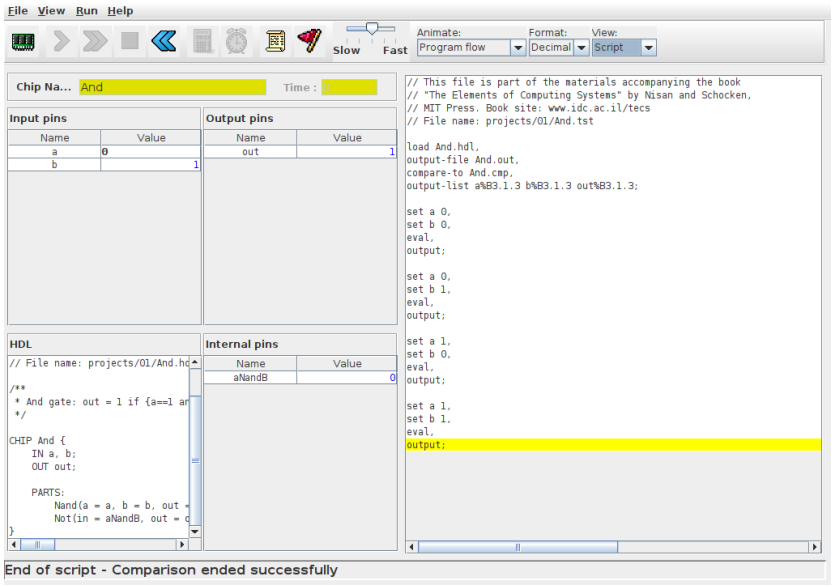

Figure 6: "Screenshot of Hardware Simulator for Nand to Tetris" under GPLv2; from GitLab

  - Major window parts
    - HDL file in lower left, test script on right
    - Values of "Input pins" can be entered, resulting "Output pins" and "Internal pins" can be computed

        * Note message area at bottom. If not shown, **resize** window!

Here you see the Hardware Simulator after the test script for the `And` implementation was executed successfully. Try it out yourself!

Note that the message area at the bottom of the screenshot currently shows that the test script ended successfully. It might also show error messages, for example, if your files contain syntax errors. Maybe you need to resize the window to make that area visible.

5.2 Textbook for Nand to Tetris

- (Nisan and Schocken 2005) The Elements of Computing Systems
  - Book chapters are available online: `https://www.nand2tetris.org/course`
    * Hyperlinked from "reading person icons"
    * We use Chapters/Projects 1, 2, 4, 5
- Download chapters now, start to read and work on Project 1
  - Note "Tips" and "Steps" at end of Chapter 1

Nand to Tetris is explained in the book mentioned here. Have a quick look now, read it later.

6 Conclusions

Let us conclude.

6.1 A Related Story, 2021

- From Nand to Zero-Day Exploit
  - Google Project Zero on attack software targeting iPhones
    * Discussion with hyperlink to `https://www.nand2tetris.org/`!
    * Summary
      · NSO Group used logic operations (And, Or, Xor, Xnor; functionally complete) of an image format (JBIG2, mostly outdated) to hijack iPhones
      · 70,000 segment commands to define computer architecture with registers and 64-bit adder
    * Quote
      · "The bootstrapping operations for the sandbox escape exploit are written to run on this logic circuit and the whole thing runs in this weird, emulated environment created out of a single decompression pass through a JBIG2 stream. It's pretty incredible, and at the same time, pretty terrifying."
  - Quotes regarding working at Project Zero
    * "learn about coding and how computers work"
    * "learning the fundamentals of programming, operating systems, and machine architecture is a great starting point"

Beyond learning objectives you may be interested in the story mentioned here. Maybe read the blog post as well.

6.2 Summary

- Nand to Tetris guides you to build a computer
  - Based on abstraction and modularity
  - Weekly projects, with guidance from us

Nand to Tetris is really about building, and thereby understanding, computers. This complex task will be broken down into manageable units thanks to abstraction, modularity, and help.

6.3 Q&A

Figure 7: "Uncovering questions" under CC0 1.0; background changed from Pixabay

Please do not hesitate to ask!

Bibliography

Neumann, John von. 1945. "First Draft of a Report on the EDVAC." University of Pennsylvania. https://web.mit.edu/STS.035/www/PDFs/edvac.pdf.

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