# Preface

***Don't think you are. Know you are.***

## Purpose

***--"What is this book?"***

--"This is a textbook, of course? A textbook of this course, you know."

***--"What makes you think this is a textbook? You haven't even read a bit of it yet."***

--"Well, if you put it that way. What is it then?"

***--"We do not know the answer to that question, and that's exactly why the question was asked."***

--"Wait a minute. Were you just saying that you don't even know what you've written? How is that possible?"

***--"A book read by no one is no book at all. It's only a purpose."***

--"What purpose?"

***--"The same purpose for which you took this course."***

***--"Look. This might sound a little weird, but we are sure there is a purpose much more powerful than just getting credits and knowledge that drives you here. In the past you might have been a person who accepted what you saw, being cheated and controlled by the world because you were expecting to wake up. Now your expectation leads you here, to this book, while the book is also looking for you. And now it's time to find out if it is true."***

--"Well, even if I believe you, how do you know this book is gonna wake me up?"

***--"We don't. But you will. And by the time you do, we will know too."***

## Course

This book is designed for and based on the course ***Computational thinking and system design***. One can gain critical knowledge not only for computer structure from hardware to software, but for the thinking and self-optimizing methods as well. Readers are required to have basic reading and calculating ability, and know at least one programming language(not really necessary).

## Structure

This book is divided into three separated chapters: **hardware**, **translators**, and **high level language and OS**. The recommended sequence is not necessary while you are learning.

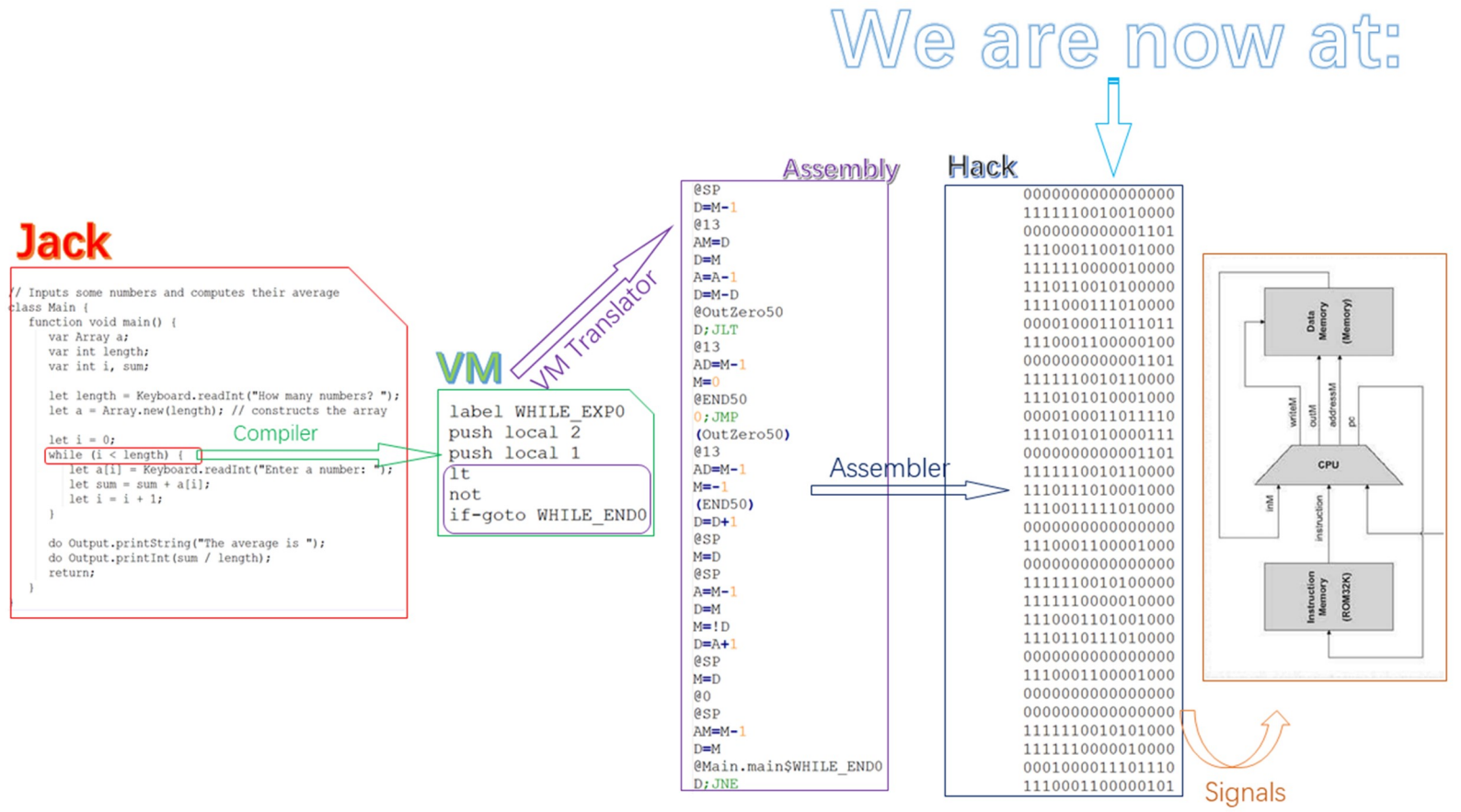
For every chapter and every section of a chapter we provided a logic model, to summarize the learning process and help you figure out the complete learn-and-know procedure. Remember, our logic model is only a guidance. Your own logic model could be quite different, and we are expecting that.

Building a modern computer from the very beginning is a huge and complicated project. One should be aware of which stage he is at in the procedure, and why is that necessary. To help the readers concentrate on each stage and summarize what they've learnt in the correct way, a logic model is needed.

Every section contains a logic model, the introduction to the section containing critical background knowledge and thoughts, specifications for the knowledge of that section, project instructions and additional thoughts. All projects include not only what we should build, but the measurable effects that can be used to test our output as well. In this way you will be able to **know that you know**.

Be aware that this book is designed to provide a more reasonable and effective way to gain all the knowledge from the original textbook ***Elements of Computer System***. In the following chapters of this book, similar specifications will **not** be included and will go into our references. **Both** should be read thoroughly if you want to learn all the details and know all the tricks.

# Hardware


We are now at:
Jack
// Inputs some numbers and computes their average
class Main {
function void main() {
var Array a;
var int length;
var int i, sum;
let length = Keyboard.readInt("How many numbers? ");
let a = Array.new(length); // constructs the array
let i = 0;
while (i < length) {
let a[i] = Keyboard.readInt("Enter a number: ");
let sum = sum + a[i];
let i = i + 1;
}
do Output.printString("The average is ");
do Output.printInt(sum / length);
return;
}
Compiler
VM
label WHILE_EXP0
push local 2
push local 1
lt
not
if-goto WHILE_END0
VM Translator
Assembly
@SP
D=M-1
@13
AM=D
D=M
A=A-1
D=M-D
@OutZero50
D;JLT
@13
AD=M-1
M=0
@END50
0;JMP
(OutZero50)
@13
AD=M-1
M=-1
(END50)
D=D+1
@SP
M=D
@SP
A=M-1
D=M
M=!D
D=A+1
@SP
M=D
@0
@SP
AM=M-1
D=M
@Main.main$WHILE_END0
D;JNE
Assembler
Hack
0000000000000000
1111110010010000
0000000000001101
1110001100101000
1111110000010000
1110110010100000
1111000111010000
0000100011011011
1110001100000100
0000000000001101
1111110010110000
1110101010001000
0000100011011110
1110101010000111
0000000000001101
1111110010110000
1110111010001000
1110011111010000
0000000000000000
1110001100001000
0000000000000000
1111110010100000
1111110000010000
1110001101001000
1110110111010000
0000000000000000
1110001100001000
0000000000000000
0000000000000000
1111110010101000
1111110000010000
0001000011101110
1110001100000101
Signals
Data Memory (Memory)
writeM
outM
addressM
pc
CPU
inM
instruction
Instruction Memory (ROM32K)

# Boolean logic

*By Chen Yihao*

---

| Context:<br>1. Boolean algebra introduced by George Boole laid the theoretical foundation.<br>2. Claude Shannon's circuit design theory demonstrates that Boolean logic is realizable in physics. | | | |
|---|---|---|---|
| **Goal: Understand Nand's completeness. Learn about the basic logic operations in Boolean algebra. Introduce bootstrapping concept in system design.** | | | |
| Effect<br>1. Learn that any basic gate's can be built by only Nand, and these gates can perform any operations in Boolean algebra.<br>2. Understand the logic connection between the bottom design and the overall system. | Output<br>1. Logic gates built in HDL.<br>2. Pass the test scripts. | Process<br>1. Learn about the Boolean algebra.<br>2. Based on the completeness of Nand operation and circuit theory, use the given Nand gate to implement other basic logic gates in HDL.<br>3. Test the gates. | Input<br>1. Nand gate.<br>2. HDL and hardware simulator.<br>3. Nand2Tetris course materials. |
| **Outside factors: None.** | | | |

---

## Introduction

When we talk about computers, what are we really talking about? I guess you must have had this question in your mind. Are they just calculators? Are they media players? Or are they communication tools? Or maybe you have your own ideas about that.

In a way, computers can be anything you can imagine. They have been part of the modern civilization, no matter how people defined them.

So here comes another question – quite a natural question – what features do these definitions have in common?

To put it another way, **what determines a computer's essence?**

If you once tried to explore the world of computers, you may have known that the information in today's computers is represented as patterns of **bits.** A bit (binary digit) is either one of two digits – 0 and 1 – which are symbols with no numeric meaning.

And amazingly, **this is exactly the essence, the origin and the foundation of our computers – Boolean logic.**

Boolean logic is introduced by Boole George to represent the truth values and the operations on them. Claude Shannon's circuit design theory later demonstrated that Boolean logic was realizable in physics. These act as the foundation of a computer system.

And it can be proved that **the Nand gate is complete** in Boolean logic. So in some way, we can say that the computer is constructed on the base of Nand gates.

In this first chapter, we are going to learn how the Boolean logic serves as a foundation of the whole computer system, which is also the basic knowledge for all the later chapters.

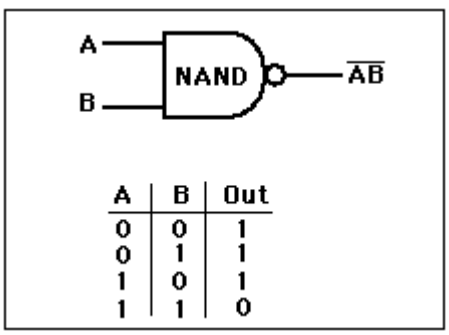
# Specifications

The book *The Elements of Computing Systems* gives very detailed information about the Boolean Algebra. For the specific principles of Boolean logic, you may refer to the pages from 24 to 30 in the book. For the implementation of Boolean logic in HDL, you may refer to the pages from 33 to 42 in the book.

Here we are going to focus particularly on the most basic Nand gate and explain its functional completeness, because we think these parts actually lay the foundation of the overall computer system, but were not given enough attention in the original textbook.

## *Nand gate*

> What is Nand? Literally, Nand is "not and". It is a logical operation on two logical values. It produces a value of true, only if at least one of the propositions is false.
>
> In Claude Shannon's circuit design theory, we can put the Boolean operation into practice by using electrical circuity. Such implementations are called gates. Nand gate is one of them.
>
> Of course there are other operations in Boolean logic, such as And, Or, Not and Xor. But we choose to focus on the Nand only, why?
>
> As is mentioned above, the **Nand only is functionally complete** enough in Boolean logic. That is to say, we can achieve all the operations in Boolean algebra with only Nand. By the way, Nor is another operation that is complete.
>
> Maybe you are curious why the only single operation of Nand can perform all the Boolean algebra. Here we are going to give the demonstration.




| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## *Completeness of Nand*

> It is evident that a binary operation in Boolean logic can have 16 possible outputs in total. As long as we demonstrate that all the 16 situations can be realized by only Nand, then we can say the Nand is functionally complete.
>
> To put the problem easier, we can firstly demonstrate that all the elements of the set {And, Or, Not} are functionally complete, and then demonstrate they can be constructed by Nand only. This way is also the application of **bootstrapping method.**
>
> By using the truth table, one can easily find that And, Or and Not together are complete. And we can also implement them three by only Nand like this:
>
> *Q = A And B = ( A Nand B ) Nand ( A Nand B ) Q = A Or B = ( A Nand A ) Nand ( B Nand B ) Q = Not( A ) = A Nand A*
>
> In summary, we demonstrate the completeness of Nand.

# Project

For this part, *The Elements of Computing System* gives very detailed information. You may just refer to the pages from 45 to 46 in the book.

# Additional Points

Now that we have a powerful “construction material” Nand gate which is complete, we can just use it to construct more complex structures, including And, Not, Or, Xor and more. It has been proven just now that by this way, we can perform any operations in Boolean algebra.

Then one may ask: “So…what?”

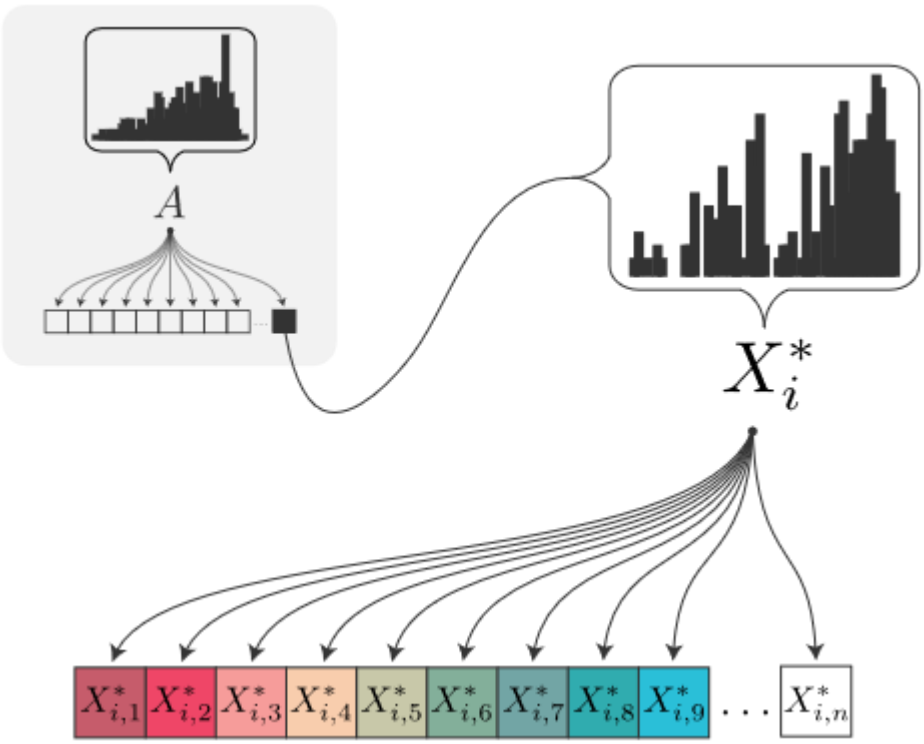
What is the significance of that?

It is true that the Nand only is complete enough for our Boolean algebra, and is the foundation of all the computer system. All of our computer science can be seen as an extension of this tiny chip. This is one aspect – **the high degree of consistency and form-simplicity in computer science.**

But if you want to build a computer totally from scratch, using mountains of piles of Nand gates, then you will find it is not so easy.

That is another aspect. Still we are talking about the **bootstrapping concept**. In the Boolean logic, we particularly focus on Nand and see it as a point of departure, so we use Nand gate to build other gates. In this way, we are actually performing the process of bootstrapping. After we build an more advanced layer of our system, we can just forget its inner implementation but to freely use its interface. That can largely decreases the difficulty and complexity of our building tasks.

And bootstrapping is exactly where the computational thinking hides.




# About

This first chapter talks about how the Boolean logic serves as a foundation of the whole computer system, which is also the basic knowledge for all the later chapters. Next chapter will introduce the more complex composition of Boolean logic, which is called *combinational logic*.

# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] Wikipedia about nand logic from: https://en.wikipedia.org/wiki/NAND_logic.

[3] Brookshear, J. Glenn. (2011). Computer Science: An Overview.

# ALU

*by Lekang Yuan*

---

| Context:<br>1. Have built the basic gates using Nand gate, which we will use in this chapter<br>2. Get started with HDL | | | |
|---|---|---|---|
| Goal:<br>1.Extend from Boolean logic into Combinational logic<br>2.The computing center: ALU | | | |
| Effect<br>1. Understanding the powerful concept of rooting many operation into few.<br>2. Deepen the understanding of Boolean logic and translation.<br>3. Know the structure of ALU<br>4. Practice computational thinking when building ALU | Output<br>HDL programs of:<br>1. Adder<br>2. ALU | Process<br>1. Follow our book, and refer to the original book only when recommended, to understand the most by yourself<br>2. Do the project with our hints. | Input<br>1. Backgrounds of binary numbers。<br>2. The functions and the implementation guidelines of half-adder, full-adder, 16bit adder, 16-bit Inc and the ALU. |
| Outside factors: ALU is the first complex project. | | | |

---

## Introduction

`Where we are?`

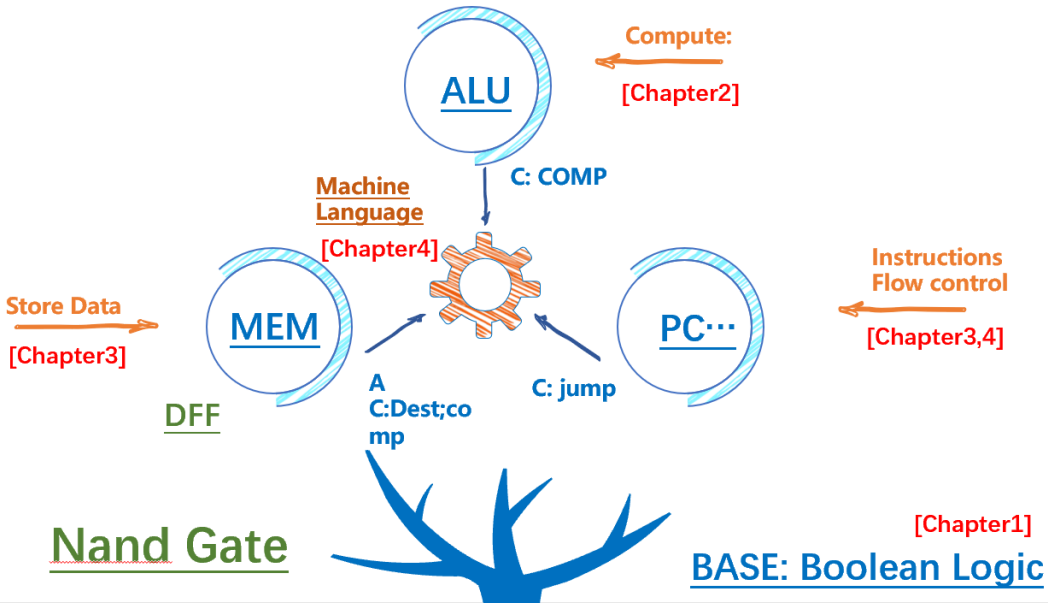
The chapter 1 gets us through the door by introducing Boolean Logic, and in this chapter we will take a step further to **combinational logic**, which enables our computer to do **arithmetic operations**. And instead of only talking about logic, we will also focus on **building the first component of a real computer**: the ALU.

`What to do to build a computer?`

Computer is all about **operations on the state of data**. The OS and all the software we see can be divided into 2 parts, frontend and backend. Backend by nature is about data structure and logic. Frontend deals with what users see and how we interact with computer, and these are also data: RGB, (x,y), a character.

So there is only 2 things we are going to consider about, **how to maintain data** (memory in chapter3), and **how to operate on data**. The latter one can be further divided into **what's the basic operations**(ALU in this chapter) and **how to read instructions(machine language,chapter5) and do these operations**(PC in chapter3, and other parts of CPU in chapter 4).We will build all these parts based on Boolean Logic, using only one gate Nand in most cases. And that's how our book develops, and you can see it in this picture:

---




---

```
"Only Add"
```

How many kinds of operations do we need in a computer? 1000, or 10000? Actually, we only need one (roughly): add.

Consider about how we do **math**. Subtraction can be viewed as adding a nagetive number, and we get "×"and "÷" by adding or subtracting in a common procedure. Even those operations in the Advanced Mathematics, like matrix operation and integration, is no more than adding and substraction, though there will be there will be many steps. And do **draw something on the screen**, computer are only need to calculate the states of the pixels, which is again a series of adding. (You can see that in the part of OS).

Well, maybe add can't literally do everything (shifting for example), but we can see that great numbers of operations can be **rooted to several meta operations**. So here, we can focus only on **building fundamental gates**, instead of having trouble dealing with the countless needs one by one.

```
Consider about Logic and Translation
```

So how to building these new gates? Are we solving circuit problems? No. As we can see in the first chapter, **computer is all about logic**. What we are going to do is only to **translate**, first clarify how we do things, then consider about the corresponding expression in **Boolean Logic**. For basic 2 bits logic, we can consider about **truth table**, and for more complex ones, we can abstract the gates built before by their functions. (For example, we can use Mux and DMux to "choose" or build "branch".)

# Specification

`Binary numbers`

You can see what's binary number, how to convert binary with decimal number in 2.1. Binary Numbers.

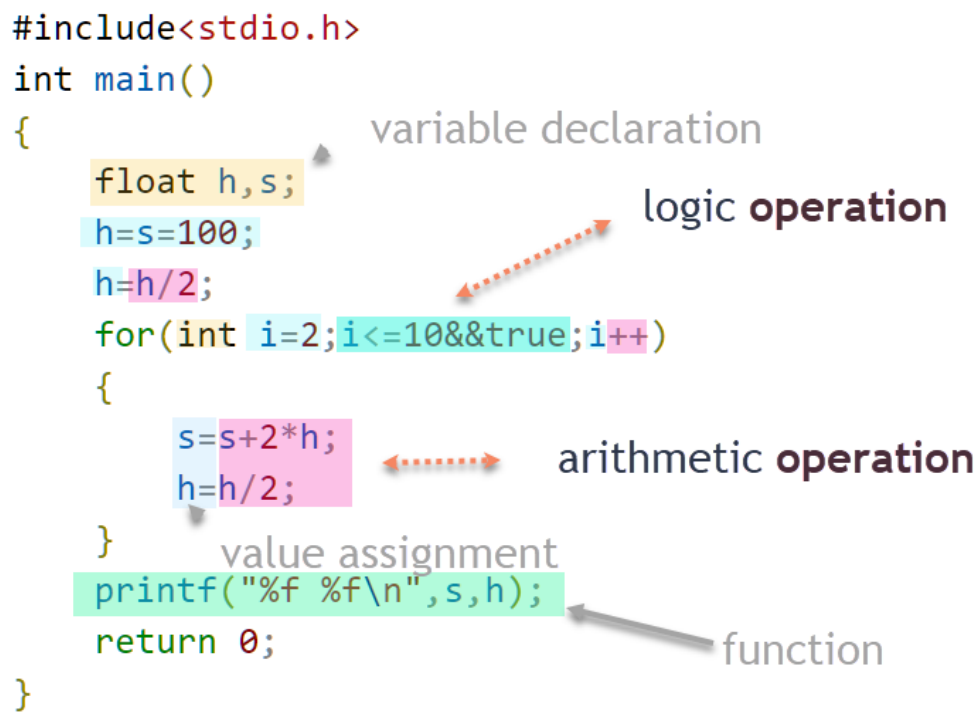
`Data storage in computer`

The figure is listed in 2.1_figure2.1. Notice that the **top is limited**. It's like a clock, whose top is 12. And adding in a clock is something interesting, in that 5 hours past 8 is not 13, but 1. And how you can get 1 from 8 by subtraction?

`ALU`

Think about this program as an example. Any program can be divided in the manner shown in the picture. Ingore the function call, as it's just a series of formal statements. We will talk about variable declaration and value assignment in the latter chapters (Machine language, High-Level Language.). In the program we can see there is only 2 kinds of operations, **arithmetic operations** like "+", "-", and **logic operations**. Logic operations can be something like "And" in Boolean Logic, or comparing operations like " <" which compares two numbers and returns true or false. An operation **returns an value**, which you can **assign to a variable**, or **judge** what to do next (In a loop or a Branch).

In ALU we want to achieve some **necessary basic operations on hardware level**, so we can achieve the left ones by **designing algorithms on software level**. So what do we need? We need some basic **arthmetic oprations** like add or subtraction, and some basic **logic operations** like and,not,or. So how can we do that? Instead of creating the operations one by one, ALU does like this:




1.Do **one** binary **logic** operation (And) and **arithmetic** operation (+)

2.**Change** on inputs and calculated result. (By combining 1 and 2 we can get 28 common binary logic operation and arithmetic operation.)

3.**Compare** the result with 0.

You can also see a more detailed description in CPT2. 2.2.

When doing the projects in the latter chapter, you can see that all the projects we are going to build can be based on these operations.

# Project

You've already have all the knowledge to build the project of this chapter. Instead of giving you all the details(As the original book does), we strongly suggest you do the project by you own. If you have any problem, you can read the following hints.

## *Adder*

Build a 16bits adder.

**How to get started?**

Consider about adding on one bit first. What's the logic of adding two numbers? Consider about the truth table.

**All right. But we have more than one bit. How to deal with it?**

When we are doing adding , there will be a carrier. When adding two numbers on one bit, what's the carrier? What's adding with a carrier? (Just add twice, right?)

## *ALU*

Build an ALU. The function list is in 2.2.2. As ALU is the first complex project we are going to build in this book, though much easier than the latter ones, we will use it to demonstrate important thinking method used in building big project.

**How can you divide the project?**

We can **divide it into 4 parts**: process the input, basic arithmetic or logic operation, process the result, analyze the final output. The **input of one part is the output of the previous part**, and for each part we only have to consider its own function. By doing so, we change one complex problem into 4 smaller problems.

**What's the key in building each part?**

Actually, there is only two kinds of things we are going to do: **make choice and 3 operations: not,and,add.** We already have gates to do the operations, so how to make choices? We can use branch gates built in chapeter1. For example, when we want to get !x when nx==1 and x hwn x ==0, we can calculate !x first, and use a Mux to choose between the result and x.

**So what kind of method we can learn from this?**

In building the complex system, we will focus on building realizable **basic blocks**, and then to **abstract** them to build more complex things. Abstraction means that we will only focus on the port, the surface, the overall function, without considering about the inside structure and realization. We are using the strategy when we are using previous gates like adder, not, and, DMux.

Having had these blocks, we are still not going to build the whole system at once. Instead, we are going to **divide the system into several parts**, each for a specific goal. In doing this, we will try to reach the goal of "**High cohesion, low coupling**", so we don't have to bother about other parts when building one part.

# Additional points

**Analog computer**: Differ from the digital computer we are building, it uses the continuously changeable aspects of physical phenomena such as electrical, mechanical, or hydraulic quantities to model the problem being solved. By comparing it with our computer, you can further understand that the principle of digital computer is more about logic rather than circuit.(Not for really building.)

**"One to all strategy":**

In chapter one we rooted all the gates to nand, and in this chapter we rooted all the operations to add (and a few more).When building a system, we can think of its **basic words**, the **only root**, focus on the **unity** instead of spending time thinking about trivial details. This strategy can be seen in many areas, like the particle system in After Effect: we only consider about altering the motion and shape of particles, and countless things can be achieved——even those not so particle-alike.

# Next chapter

Having finished this chapter, there is still something remained to be solved:

From the point of hardware: ALU is a function which has 2 inputs,1 output,and 64 possible operations . So where to get the input and store the output? Which operation to do?

From the point of logic: we've introduced combinational logic in this chapter. Can it solve any problems?

We will see in the next chapter.

# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press, p47-p60.

[2] Information about ALU fromhttps://en.wikipedia.org/wiki/ALU

[3] Information about Analog computer fromhttps://en.wikipedia.org/wiki/Analog_computer

[4] Information about Particle system from https://en.wikipedia.org/wiki/Particle_system

# Sequential logic

*By Lekang Yuan*

---

| Context:<br>Have known about combinational logic in first chapter | | | |
|---|---|---|---|
| Goal:<br>1.Sequential logic: expansion across time.<br>2.Storage and Control units of the computer. | | | |
| Effect<br>1. Understand the two important concepts of Memory: space and time.<br>2. Deepen your understanding of Boolean logic<br>3. Know the structure of memory. | Output<br>HDL programs of<br>1.Register<br>2.Memory<br>3.Counter | Process<br>1. Read the introduction part and think about the concept of time<br>2. Build the project under the guidance,<br>3. Read the left parts.<br>4. Think again about time and space in memory. | Input<br>1. Sequential logic<br>2. DFF |
| Outside factors: Not familiar with the physical implement of DFF and the clock. | | | |

---

## *Introduction*

`where we are?`

In this chapter we are going to introduce another kind of logic besides combinational logic: sequential logic. In the same time we will build memory chips, and we will focus on understanding the key concept of memory: **space and time**.

`why sequential?`

**Need of previous state**

Combinational logic means that the output of a combinational chip **depends solely on the current input**, regardless of what **the sequence of previous input** the chip has. So here is a question: Can we build a computer with only combinational logic?

Not actually. See this simple example. Let's suppose there is a counter, whenever you press the bottom, the number will increase by one. To do this, only **the current input** (whether or not you are pressing the bottom)is not enough, the **state** of the counter(the total number), which is **the result of the previous input** should also be an input. And to record the state, we must introduce the concept of time.

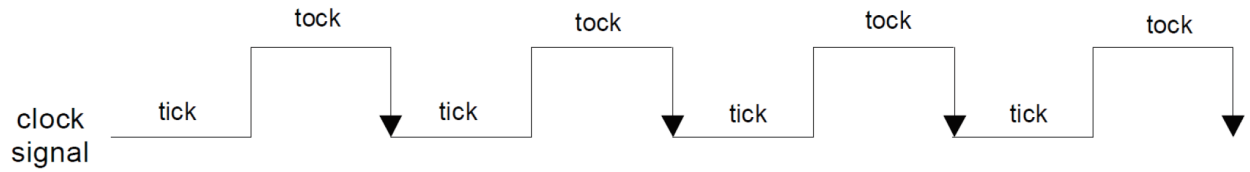
**Reduce space complexity**

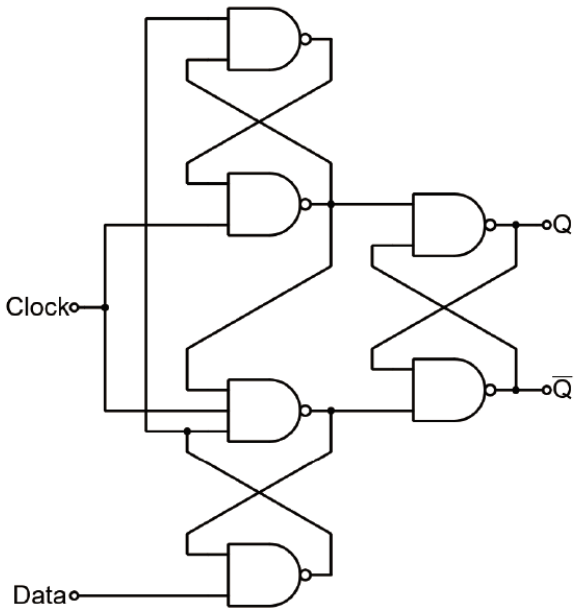
Another reason for why we need time is that by expand through time we can greatly reduce **space complexity**. If there is only combinational logic, we can not form a loop(If so, there will be logic conflict like O=I+1;O=I), so each component can only be used once. If we want to solve a complex problem, we can link many ALUs together, each to carry out one step--It works, but it's so stupid as it cost too much. So we choose to deal with the problem by **expanding through time**, by **doing one step in one clock cycle**(introduced later), so we can use only one ALU to solve the same problem.

`Introducing the time`

**Master clock**

Since maintaining state or memorizing values is **time-dependent**, we must talk about the **Clock**. In most computers, there is a master clock that continuously delivers periodic change signals. **Each sequential chip receives this time signal** so that different chips can perform functions in a **uniform time frame**.




About its implement and the meaning of being uniform, please read CPT3.21,CPT3.1_Figure3.4.

**DFF**

Next we should find a **component** which is **time-dependent.** The most basic time-dependent expression can be: **out(t) = in(t-1)**, and we can achieve this with DFF (Data flip-flop/Delayed flip-flop). In this way we maintain(or delay) states for one time cycle, and by forming a loop, inputing the output we can **maintain states** for many time cycles. Also, we can **change** state at by define in(t-1).

Note that we will abstract away the complexity of building DFF, and simply use it as a **build-in** building block like Nand gate.For more details, please read CPT3.2.1. (Actually The a way to build DFF is to use nand, so we can further see the completeness of nand here. )

# Specification

`Memory series`

Having introduced the concept of time, let's see what should be considered to make a memory.

First are the **operations**. It shuold have the ability to maintain state, and we can read(get the stored state) or write(change the state). Think about the time expression, and think how we can do that. Deal with that on one bit, and we will get a 1-Bit register.

Next is **expansion through space**. First in one word, expand 1bit register tio 16bits register. Then bundle register together to form RAM of different sizes.To access one register, we shuold make choice by its **address**.

`Control chip: counter`

We can store data in memory, and we can also **store instruction** in memory. The figure is that a set of instructions is stored in memory, one by one, and **in each time cycle** we fetch and execute one instruction by its **address**.

So we need a chip to help with that. In common cases, we will just goto the next instruction, so the address should increase by 1. Sometimes we want to jump to another location, which can be done by inputing a address and use it as the next address. When we want to run the program again,the address should be reset to 0. These are all we need from a counter.

# Project

We will build the register, RAM of different sizes and a counter.

The project is not hard. Just go to proj_03 to do it. Focus on how to make choose between operations and locations. ( Recall the gates built in chapter1, and think about how we do the same thing in chapter2. )

# Additional Points

**Comparison: Different kinds of Memory**

Forming a circuit cycle is not the only way to build memory. We can also build non-volatile memory like disk, using whether or not the material is magneitized to represent state. It can maintain the state when powered off, but it's slower.

**Address**

In the previous parts we've spilled a lot of ink talking about time, why we need time, and the basic operation based on time: maintain, read, write. Here let's see another important concept of memory: space.

When we are considering about space we need to focus on address. Address is an **abstraction of the real place** where the state stores, abstracting the place of a memory chip to a number. And it is an important topic in programming. When we are introducing a new variable, we must **allocate** memory for it and **record** the address. When we want to **access** previous declared variables, we get them by address. When some variables are not needed, we **recycle** the memory in that address. Actually, every **variable name** or function name in the following chapter, no matter in assembly, VM, or high level project, **is just a represent of a certain address**.

So for every problem involved with memory, you can consider about it in the following manner:

- Are we doing read or write?
- How the memory is allocated?

**Expansion through space: File and Web**

In the above discussion, we only focused on what happens in one program. So what if we want to deal with **different programs**? Use **files**. The operating system provides us with a file system , so we can carry out file operations: **open,close,read,write**. Open and close enables us to switch between programs or other files, so we can run them(read) or change them(write). And to get the file, we should also get an address, or a **URL**.

After we have files, we can consider about the web. In essence, web is no other than a larger file system, which includes all the computers in the world. When we are browsing webpage, we just get a HTML file in another computer, open ot and run it with the browser. As there are so many computers, we have to take more attention on the arrangement of address, but as long as we have it, nothing can stop us from accessing any bits in the world.

# Next chapter

As we already have memory and ALU, the thrilling time is just in front of us. In the next chapter we will combine everything together to form a completed computer! Σ(⊙▿⊙"a.

# References


[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press, p60-p73.

[2] Information about Combinational logic from: https://en.wikipedia.org/wiki/Combinational_logic

[3] Information about Sequential logic from: https://en.wikipedia.org/wiki/Sequential_logic

[4] Information about Clock signal from: https://en.wikipedia.org/wiki/Clock_signal

[5] Information about URL from: https://en.wikipedia.org/wiki/URL

[6] Information about Computer data storage from: https://en.wikipedia.org/wiki/Computer_data_storage

# Computer Architecture

*By Chen Yihao*

**NOTE**: the order has been rearranged, and this chapter is corresponded with original chapter 5.

---

| Context: The chips built in the previous projects. The machine language syntax. The Von Neumann architecture and relevant instructions. | | | |
|---|---|---|---|
| Goal: Learn about the Von Neumann architecture. Use the chips built previously to make a general Hack computer. Understand the working of hardware. Prepare for the assembler's implementation. | | | |
| Effect<br>1. Understand the generality of Von Neumann architecture.<br>2. The test programs' working supports the practicability of Von Neumann architecture and the completeness of Turing machine. | Output<br>1. Memory.hdl, CPU.hdl and Computer.hdl<br>2. Pass the test programs. | Process<br>1. Use HDL to modify the hardware and by analyzing the instructions of Hack machine language, combine the right chips to achieve the function of each part.<br>2. Coordinate memory and CPU. | Input<br>1. PC, RAM4K, RAM16K, ALU, Screen, Keyboard and other chips.<br>2. Nand2Tetris course.<br>3. HDL and hardware simulator. |
| Outside factors: relationship between Hack machine language and hardware is hard to understand | | | |

---

## Intoduction

In the previous chapters, we have learned about combinational logic and sequential logic. So, how can we use these discrete parts of knowledge to build a powerful computer?

Secrets hide in the way we "compute". Whenever we want to solve a problem, we always need a pattern to follow. If not, we have to find one. To solve similar problems at any time, we memorize the exact steps. In computers it is quite the same. Those "steps" become a category of instructions and are stored in the computer's "Instruction Memory". After that, when we want to calculate something, we always need a place to do it and note down the intermediate values. That place is called the "Data Memory". Also we need our brain, whose counterpart in computer is the CPU, and a reminder of what to do next, which is the Program Counter.

All these units form our ideal computer structure, and each one of them can be implemented by our logic gates. As you can see in our "Additional thoughts", our hack computer inherits the features of this structure, both advantages and disadvantages. **This structure is "one of" the computer architectures.**

The first documented computer architecture was in the correspondence between Charles Babbage and Ada Lovelace, describing the analytical engine. And in mordern computer science, the most well-known architecture is **Von Neumann architecture**, which is exactly what we are going to build in this chapter.

After that, the hardware of our Hack computer is done and we are going to software part. So in some way, the computer architecture is also a bridge between hardware and software. The upper level software will finally use the machine language instructions to coordinate the machine, and this process is just determined by the computer architecture.

Besides Von Neumann architecture, of course there are other computer architectures. **What are the connections among them?** And **what are the differences?** These questions are also what we are going to discuss.

# Specifications

In this chapter, we are going to build a Hack computer based on the Von Neumann architecture. So we should firstly know what Von Neumann architecture is. The book of *The Elements of Computing Systems* has given very detailed information about the basic components in Von Neumann architecture as well as their functions. So you may just refer to the pages from 100 to 122 in the book. But the book doesn't tell us "why" -- **why we choose Von Neumann architecture**. To answer this question, we are going to focus on the characteristics of Von Neumann architecture rather than the common specifications.

## *Von Neumann architecture*

It is quite clear that we must build our computer in Von Neumann architecture. But before that, there is a question -- why Von Neumann?

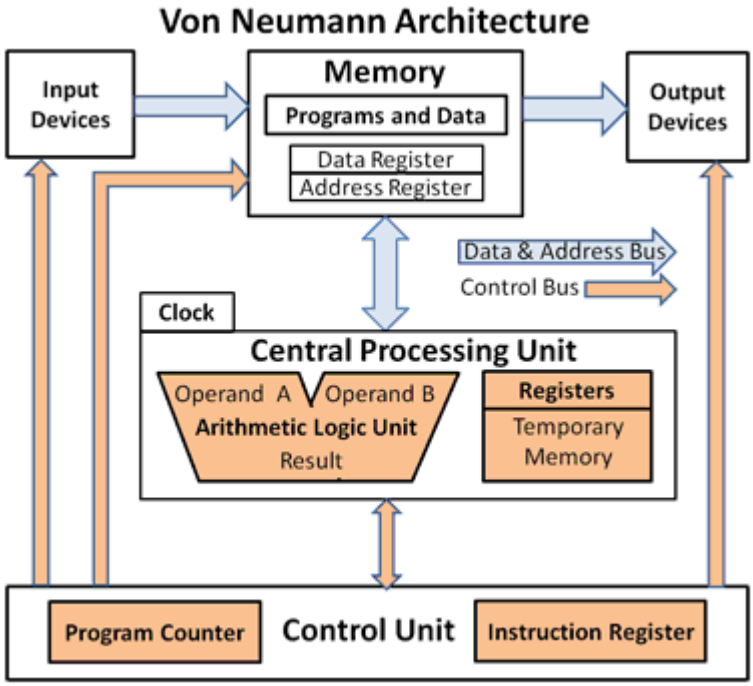
In other words, **what are the characteristics of Von Neumann architecture?**

> The von Neumann architecture, which is also known as the von Neumann model and Princeton architecture, is a computer architecture based on the 1945 description by the mathematician and physicist John von Neumann and others in the First Draft of a Report on the EDVAC. This describes a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers; a control unit containing an instruction register and program counter; a memory to store both data and instructions; external mass storage; and input and output mechanisms. The meaning has evolved to be **any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus**. This is referred to as the von Neumann bottleneck and often limits the performance of the system. (*Wikipedia*)

From the descriptions in *Wikipedia* we can get such points:

- It has some basic parts like CPU, CU, memory, mass storage and I/O devices.
- An instruction fetch and a data operation cannot occur at the same time.
- Its performance is limited, but it is simpler compared with other architectures.

So that is what we should know about Von Neumann architecture -- **a standard and ralatively simple stored program computer architecture which has its bottleneck in performance**. It is the answer to the question and we should always understand that what we build is based on the Von Neumann architecture and it has its advantages as well as disadvantages.




## *Von Neumann bottleneck*

As is mentioned above, there exists what is called Von Neumann bottleneck in Von Neumann architecture. Then you come to wonder how it could happen. So here is the detailed explanation to Von Neumann bottleneck.

In Von Neumanm architecture, there is **only one shared bus between the program memory and data memory**. Such design leads to the von Neumann bottleneck -- the limited throughput (data transfer rate) between the central processing unit (CPU) and memory compared to the amount of memory. Because the single bus can only access one of the two classes of memory at a time, throughput is lower than the rate at which the CPU can work.

This seriously limits the effective processing speed when the CPU is required to perform minimal processing on large amounts of data. The CPU is continually forced to wait for needed data to be transferred to or from memory. Since CPU speed and memory size have increased much faster than the throughput between them, the bottleneck has become more of a problem, a problem whose severity increases with every newer generation of CPU.

That is the answer.

So here you come to wonder -- **what can be done to avoid such bottleneck?**

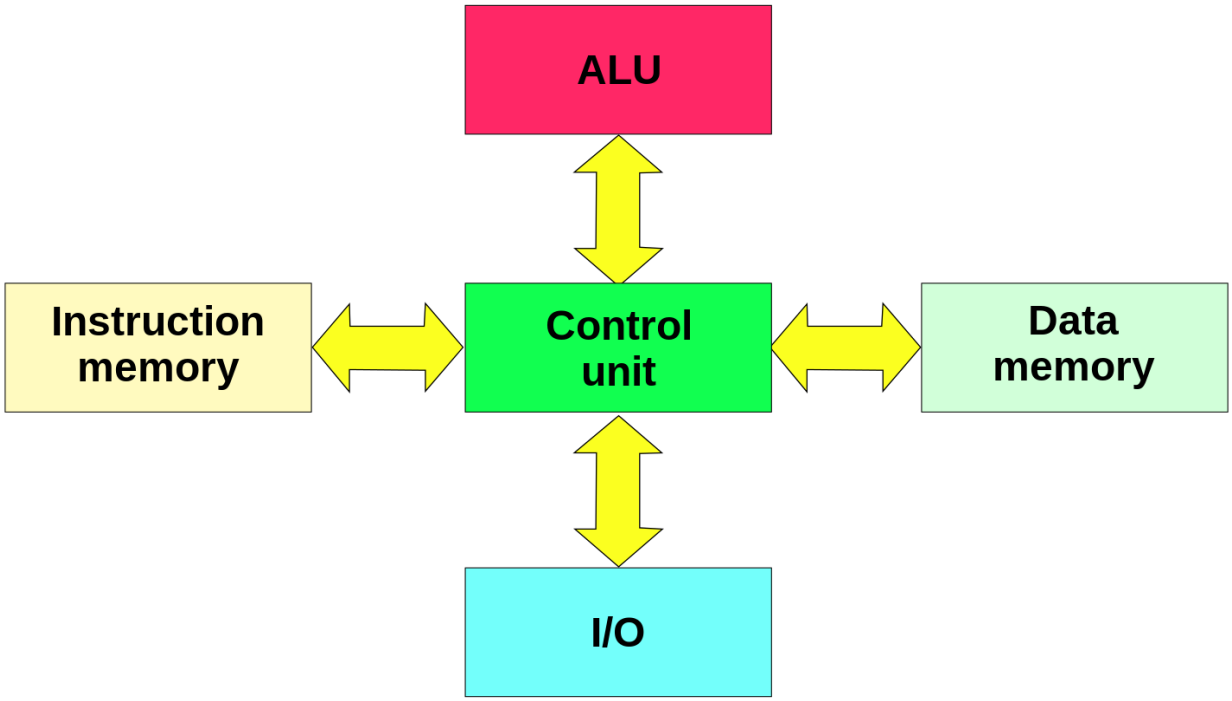
And that is the story of the other computer architectures. A relatively more advanced architecture called Harvard just solve this problem. We'll talk about it later.

# Project

This chapter's project is going to build a general Hack computer based on Von Neumann architecture. The book of *The Elements of Computing Systems* gives very detailed information about this part, and you may just refer to the pages from 125 to 126 in the book.

# Additional Points

## *Harvard architecture*




As is explained above, the Von Neumann bottleneck is the consequence of the shared bus, so the most direct way to solve it is just adding another bus. And by this means, we have Harvard architecture.

> In a system with a pure von Neumann architecture, instructions and data are stored in the same memory, so instructions are fetched over the same data path used to fetch data. This means that a CPU cannot simultaneously read an instruction and read or write data from or to the memory. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache.
>
> A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway. (Wikipedia)

Also, a Harvard architecture machine has **distinct code and data address spaces**: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight-bit byte that is not part of that twenty-four bit value.

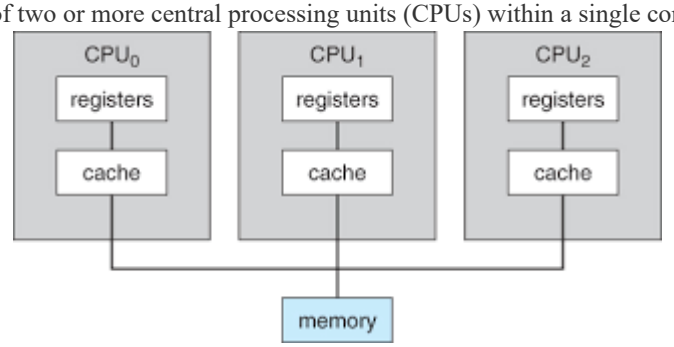
## *Some other advanced architectures*

As we all know, electric pulses travel through a wire no faster than the speed of light. Since light travels approximately 1 foot in a nanosecond (one billionth of a second), it requires at least 2 nanoseconds for the control unit in the CPU to fetch an instructions from a memory cell that is 1 foot away. Consequently, to fetch and execute an instruction in such a machine requires several nanoseconds -- which means that increasing speed of a machine ultimately becomes a miniaturization problem. Although fantastic advances have been made in this area, there appears to be a **limit**.

In an effort to solve this dilemma, computer science concentrates on **throughput** rather than merely execution speed. And throughput here refers to the total amount of work the machine can accomplish in a given amount of time rather than how it takes to do one task.

And some other advanced architectures based on this concept have been invented.

> ### Pipelining
>
> In computing, a pipeline, also known as a data pipeline, is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion. Some amount of buffer storage is often inserted between elements. (Wikipedia)

> ### Multiprocessing
>
> Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system.
>
> 
> 

In conclusion, all these architectures are different steps towards **parallel processing**, which is the performance of several activities at the same time.

---

# About

In this chapter, we learned how to build a general-purpose Hack computer, which is in Von Neumann architecture. Actually, in the process we can found that any computers' functions, specifications and performance were directly revelent to the design of computer architectures. In a word, **computer architecture is the skeleton of the hardware part of computers**. Only with the well constructed computer architecture can the soul of the software part fully perform its functions.

After this chapter, we will go into the software part. And the next chapter is about machine language and assembly language, which are highly revelent to the structure of our computer architecture -- Von Neumann architecture. If you have problems in understanding them, you may as well go back here to have a look again.

# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] Wikipedia about Von Neumann architecture from: https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[3] Wikipedia about Harvard architecture from: https://en.wikipedia.org/wiki/Harvard_architecture.

[4] Wikipedia about pipelining from: https://en.wikipedia.org/wiki/Pipelining.

[5] Wikipedia about multiprocessing from: https://en.wikipedia.org/wiki/Multiprocessing

[6] Brookshear, J. Glenn. (2011). Computer Science: An Overview.

# Assembly Language

*Han Zhilei*

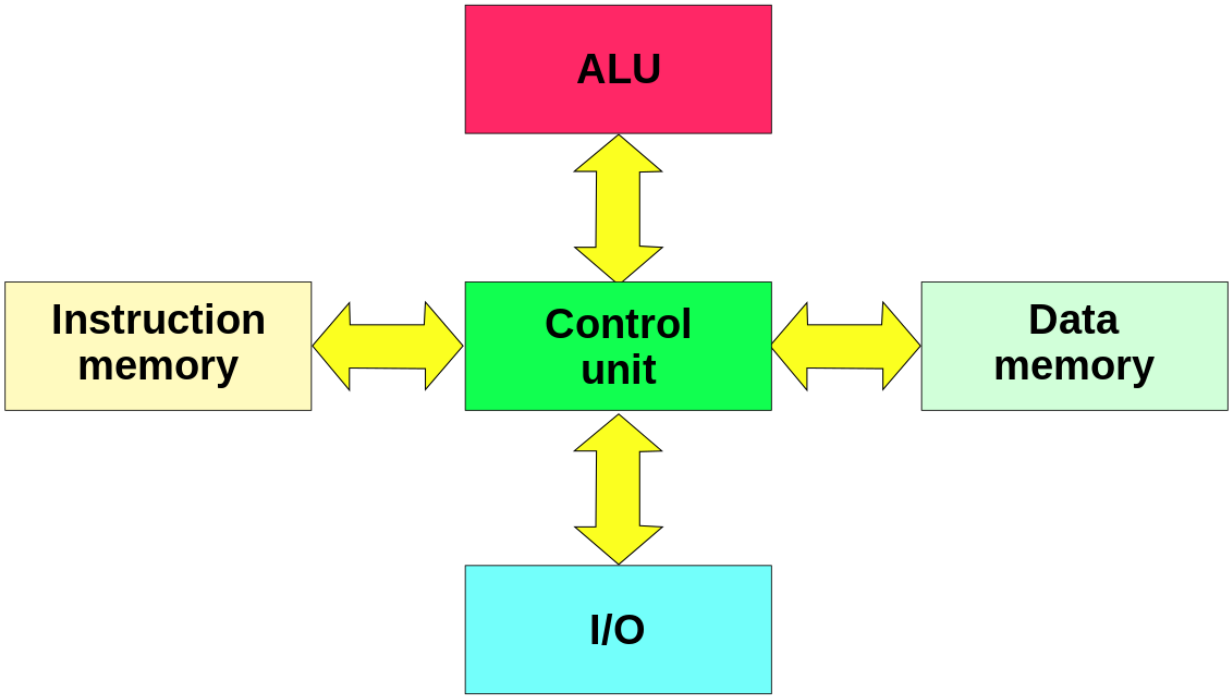
**NOTE**: the order has been rearranged, and this chapter is corresponded with original chapter 4.

After building the entire functional Hack computer, this chapter advances towards the hardware-binded control method of the computer, which is called **assembly language**. It's our first stop in the journey through the *software layer*.

| Context:<br>1. Entire Hack computer built in previous chapters<br>2. The already defined instructions set of the CPU | | | |
|---|---|---|---|
| Goal:Specify the machine language,design an according assembly language | | | |
| Effect<br>1. Hack computer is able to be controlled by the assembly language built this chapter.<br>2. Provide the fundamental of more high-level software abstraction layers | Output<br>1. Runnable programs written in Hack Assembly<br>2. Pass the testcases provided | Process<br>1. Design a proper assembly for Hack platform<br>2. translate manually instructions for the computer to assembly codes<br>3. Implement algorithms and processes | Input<br>1. Nand2tetris reference and files<br>2. The input set of CPU and entire architecture of Hack computer |
| Outside factors:Hack assembly also provides user-friendly utilities,which are not necessary | | | |

## Introduction

Congratulations on your implementation of the Hack computer. It's fully functional, and also powerful. We originate from **Nand** gate only to build the whole computer and adopt Harvard architecture, which facilitate matters when we look into the underlying structure.Here's a graph to make your memory of **computer architecture** come back.[1]




We're now going to build our software layers and this chapter will discuss the language that directly generated by hardware implementation and design an assembly language. In the end, we shall be able to get the machine running as instructed.

###What is program?

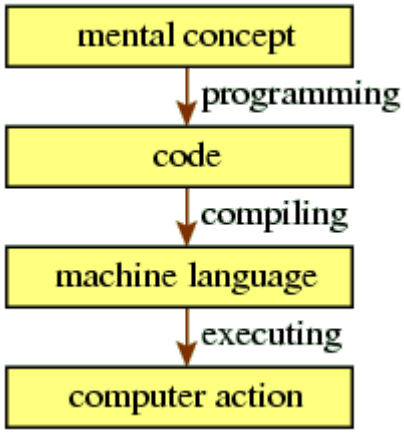
Instruction memory, or *ROM*, is where the instructions are stored. Recall the design of our Hack computer, and you may remember the **Reset** button on the surface of the computer, which will redirect PC to be 0 - equally restart the program. Computer reads instructions in ROM according to the content of PC, and all the instructions form the *program*.

Since you have already built the computer, it's quite easy to find the instructions can be classified into 2 kinds: One kind changes the content of A-register, while another carries out operations. You can divide the latter into 2 aspects as well :either jump among the instructions of the program or conduct arithmetical&logical operation.

What we use to encode our program is machine-binded string of 0s and 1s, computer explicitly obey the codes.We call them **Machine language**,because of its direct connection with the underlying hardware platform.

## *What is programming?*

In essence, what we are going to do in the following chapters is **all about solving problems by programming**. With our hardware architecture working perfectly, we have **potentially** a very powerful tool. To unlock this potential, we have to translate our human thoughts and concepts into a notation that is in conformation with the specifications of this architecture. **This** is what programming is ALL about. This process is better summarized in the graph bellow. [2]

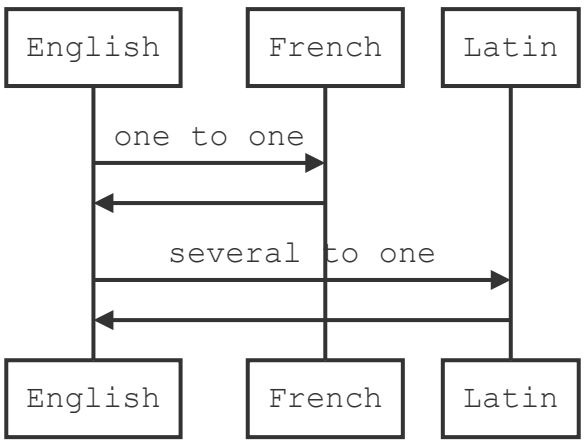



As you can see, machine language serves as the fundamental of the software layer as well as the interface for hardware. We can prove that anything a computer can do is able to be expressed by this language.When we are **programming**,the code produced all originate from machine language. What we will do is just facilitate the process of programming.

## *Assembly:Why do we need it?*

The difference between a machine and a human brain decides there should be higher level of language for programmers.**Assembly Language** is the most fundamental abstraction, which can be understood as mnemonic notation for the machine language. For instance , `1110001110000000` can be represented by `D-1`.

The design of assembly language is not fixed, but it should be tightly-binded to machine language. Generally speaking, a single assembly instruction is in correspondance with a single machine instruction, so there is no true difference between these two, because the **vocabulary** of the languages is the same - as stated before, jumping and operations - Just like :




But we will see later that although unnecessary, the assembly language can provide utilities for *programmers*,for the sake of simplifying the **programming process**.

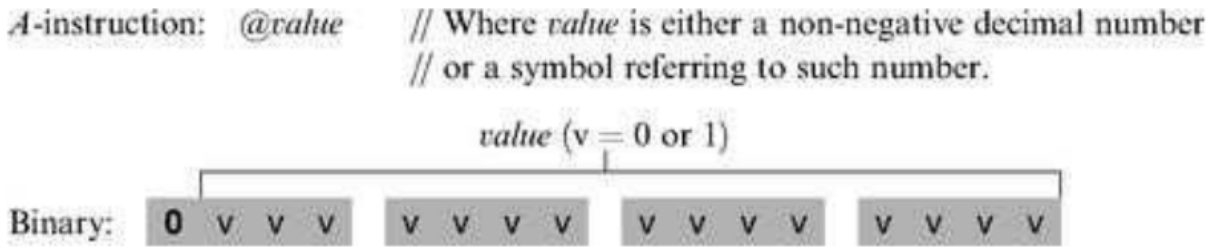
# Specifications

In this part we will see how the Hack Assembly language is designed, and learn how to program using this language. A vast part of details in this section can be found in **The Element of Computing system section 4.2**. Although we rearrange the order of the chapters, it still throws light on the topic.

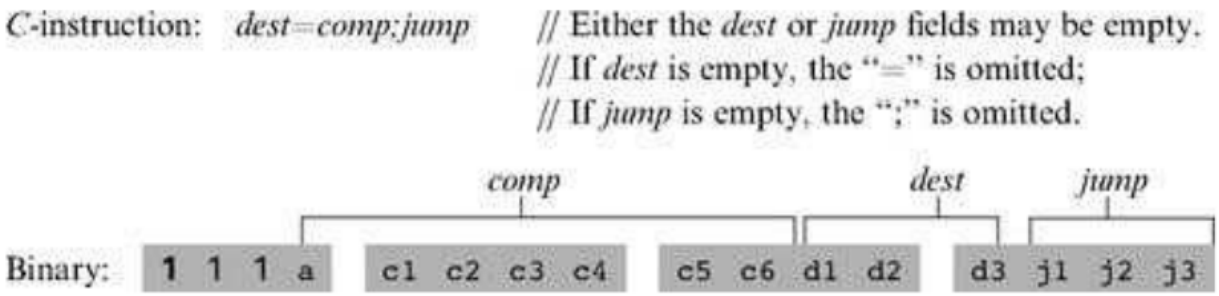
## *A- and C- Instructions*

As mentioned before, from the input of CPU, we classify the instructions to two kinds.In assembly language we define them to A- and C- instructions, respectively.

A-instruction is used to change the content of A register, its syntax is:




while C-instruction is used to carry out operation,jump through program and change the content of D and M register:




These two kinds of instructions are all user-friendly notation for corresponding machine code,as shown above.For more information,just refer to the Nand2Tetris textbook where more details are discussed.

## *Expanded Utilities of Hack Assembly Language*

It's unnecessary to provide more feature for assembly language, as is in the sentence *Keep it simple and stupid*.But with care,adding a few of them boost the process of programming.

- The First Improvement: You may notice that **A,D,M** are all pre-defined symbol respectively referring to A-,D-,M-register.As long as we have symbols for certain RAM location,we can bring in more predefined ones to simplify the coding.You can refer to section 4.2.4 for more information.

- The Second Improvement: We should provide the feature that is the most useful.When programming using assembly,counting the number of instructions is the greatest nuisance. A good way to get rid of it is to define a "symbol" as well.But the symbol is not for register but for instruction,and we call it **label**. A label represent the next following instruction and can be referred to by jump operation. It greatly reduces the time for programming.

  Likewise,we can allow programmers to define own symbols for certain registers.The user-defined identifier (label or symbol) will make the process more automatic.See section 4.2.4 for more information

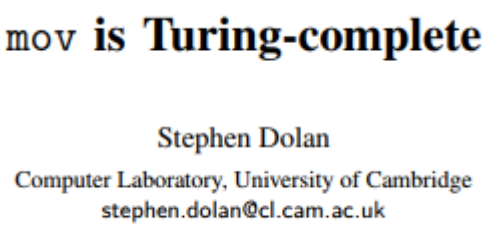
These **improvement** adds up to the power of the assembly language,by which improves the efficiency.

# Project

After fully learned the principles of assembly language,try to write programs to carry out practical opearations using hack assembly language,with specification of the project described in section 4.4

# Additional Points

## *completenesss of mov,and the instruction set*



`mov` **is Turing-complete**

Stephen Dolan

Computer Laboratory, University of Cambridge

stephen.dolan@cl.cam.ac.uk

**In previous chapter**,we have stated that **Nand is complete** and **Add is complete**,.etc. What about the language part? Is a single instruction able to represent all the others?

*mov* is an instruction in x86 architecture,Stephen Dolan has proved that by using *mov* and using it only we can represent any process that a computer can do.[3]

However,just like we do not use Nand everytime we want to implement a chip,and do not use add always,we do not use a single instruction to carry out all the operations.Modern CPU takes several to thousands instructions as their input,and is divided to RISC and CISC accroding to the number of the instructions.Learn more from the link in [4].

# About

By learning how to write programs and what is programming using Hack Assembly language,you are supposed to have basic concept of what we will do in following chapters discussing more software abstraction layers. Before we advance to next layer,we have to implement all the features we learned here in next chapter,by a program called **assembler**.

# References

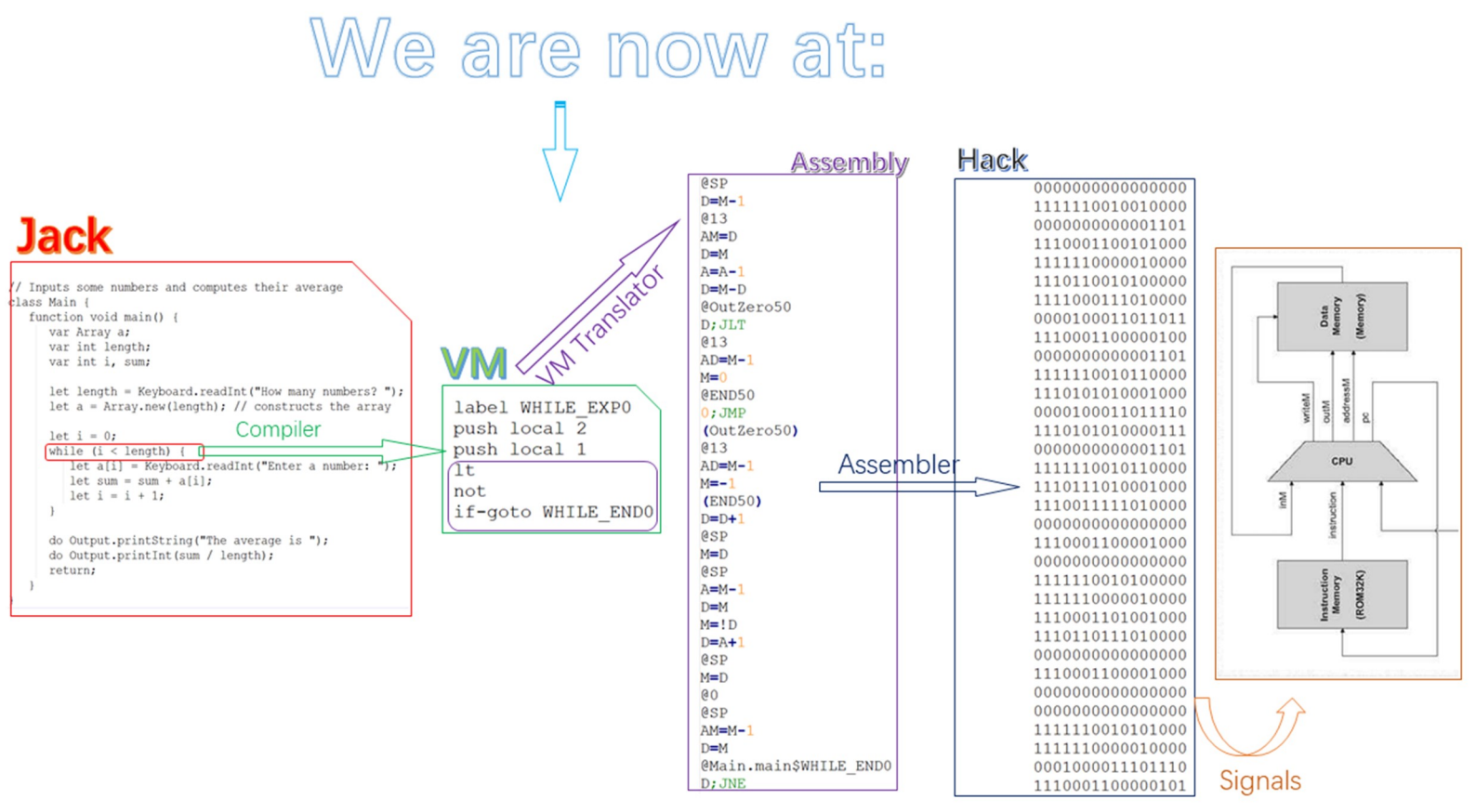
[1] Picture from http://www.toves.org/books/java/ch01-overview/index.html

[2] Picture from https://en.wikipedia.org/wiki/Harvard_architecture

[3] Stephen Dolan(2013),mov is Turing-complete,Computer Laboratory, University of Cambridge.Accessible from here:https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf

[4] Wikipedia,Reduced_Instruction_set_computer,see here:https://en.wikipedia.org/wiki/Reduced_instruction_set_computer

# Translators



# Assembler

*By ZhengZihan*

---

| **Context:**<br>**Writing programs in machine language is achievable but not efficient enough, but machine cannot read any other form of language.** | | | |
|---|---|---|---|
| **Goal:**<br>**Build a translator between assembly language and machine language.** | | | |
| **Effect**<br>1. Have a good understanding of interpreting one form of expression to another.<br>2. Gain the ability to distinguish critical parts of a certain form of language.<br>3. Know how to examine the completeness of the assembler. | **Output**<br>1. Assembler.<br>2. Programs.hack<br>3. Comparing files and results. | **Process**<br>1. Gain full knowledge and specification of machine language and assembly language.<br>2. Figure out the right sequence to interpret that can avoid bugs.<br>3. Write your program, test it and optimize it. | **Input**<br>1. Specification on assembly and machine language provided.<br>2. One programming language.<br>3. String analysing method given. |
| **Outside factors: Your precious time~** | | | |

---

## Introduction

Congratulations! We really appreciate your comprehension which enables you to implement PC and learn that complicated hack machine language. However, one must be aware of the fact that we create machine language because our PC is, at this moment, foolish and cannot comprehend our thoughts.

So what can we do?

Let us not **forget**, my friends, that we were once the same foolish as our machine is, when we were kids. But now we are not anymore, not just because we *learnt things*, but because we were *guided to learn* as well.

That's why in this chapter we will ***focus*** on ***how a machine can learn***, and ***by what means we can guide it***.

### *How can a machine learn?*

To answer this question we must first try to figure out the answer to the other one. That is: **"How can we learn?"**

Unfortunately we are not able to give you a perfect answer because of the complicated mechanism of learning inside our brain. But it's OK for us to suppose that when you learn something, there must be a **physical evidence** in your organism. For us, it might be a curve in our brain; for machine, it can be a new chip added or a new function implemented.

~~***(One critical advantage we have over machine now is that this physical improvement can be made by ourselves. A machine cannot optimize its hardware by itself, at least for now)***~~

## *By what means can we guide it?*

Human beings learn things, especially a second language, mostly by using their mother tongue. They memorize new vocabulary, learn grammar, build connections in between, and then they can understand.

Fortunately, machine can be taught in the same way. And their second language, used to communicate with us, is the ***assembly***.

# Specifications

## *Assembly*

You might have noticed by now that when we were trying to teach you machine language, we've given a great deal of names to those "1"s and "0"s such as"A-instructions", "C-instructions", "comp field", "jump field", etc.

**-"WHY? "**

-"Because it helps us understand? "

**-"Correct! But HOW? "**

Now we are talking.

Names, or to be more accurate, symbols, provide abstraction. Abstraction stands for connection. A symbol that refers to a group of certain elements creates sequence and order. Instead of memorizing ***all*** the details, we just need to understand ***each*** one of them, and build a symbol table to refer to when necessary. Moreover, if the symbol table can be read and carried out by our machine, we will have to manufacture those hard works no more.

**This *table* is exactly what we are going to achieve in this chapter: The assembly.**

*Now, before we move on, we will have to point out that from this project on, you'll need at least one programming language. If you know no programming language at all, you might as well start learning one to finish the subsequent project.*

## *Specification for Assembly*

Assembly, is somehow like a bijection. (Almost)Every line hack machine language you've learnt before can find its counter part in our assembly. So, the main purpose of creating assembly is to make machine language more easy to write.

Eg: instead of writing 0000000000000010 (if you learnt to write a program in machine language you will know how hard it is), we first understand that we are writing an A-instruction, or a "noun", literally. And for noun, we give it a symbol **@**. And since we know exactly how to deal with decimalism and the binary system, we simply use the decimal number. So in assembly, this line can be written in "***@2***".

## *Specificaion for A-Instruction*

There are two types of A-instruction: "@142" and "@qwe"***(Really?)***. The former gives us a specific number to refer to and is of more practical use(that is, when you are calculating). The latter is more like a symbol of address, telling people that "I'll store a number in a memory that is called ***qwe.***", and the assembler automatically assign the "qwe" a number(starting from 16). And then the line "@qwe" will always be translated into "@13".

**Please**, stop for one second and see what we've just done. Our thoughts, though only a small part of it, will be able to be understood by our machine through this new program. ***How did it happen? Why can't our other thoughts?*** We will leave that question to you for now.

Because we decided and started to make a difference. The programming process will seem to be quite simple, and that is because **we did the difficult part for you**. (Which part?) You will find it not so easy when you are facing the problems that follow with bare hands. We are happy to remind you that they seem to be hard because you are making efforts, which means ***you care***, and ***are trying*** to find a solution.

Just some more details in this section:

*1. When meeting a line starting with "@", it goes into "A-Instruction Class".*

*2. When you see any non-digital symbol in the following, you assign a memory address to that symbol, otherwise you translate the decimal number into binary number, making sure it is sixteen bit.*

Now, why not get your hands dirty and **begin** achieving some of our functions?

(Go to our subproject 1 and find your task~)

## *Specification for C-Instruction*

**Unlike** A-Instructions, C-Instructions(verbs) are more complicated. **Like** A-Instructions, C-Instructions can also be classified. Remember, there in nothing but symbols, and combination of symbols. The ability you must acquire to build all translators is figuring out the proper connection between one group of elements and another. In this particular section, it is how the symbol table is generated and how it works.

Take a look at the following example:

| **Machine Codes** | **Assembly** |
| --- | --- |

| Machine Codes | Assembly |
|---|---|
| 0000 0000 0001 0000 | @counter |
| 1110 1111 1100 1000 | M=1 |
| 0000 0000 0001 0001 | @sum |
| 1110 1010 1000 1000 | M=0 |
| | (LOOP) |
| 0000 0000 0001 0000 | @counter |
| 1111 1100 0001 0000 | D=M |
| 0000 0000 0110 0100 | @100 |
| 1110 0100 1101 0000 | D=D-A |
| 0000 0000 0001 0010 | @END |
| 1110 0011 0000 0001 | D;JGT |
| 0000 0000 0001 0000 | @counter |
| 1111 1100 0001 0000 | D=M |
| 0000 0000 0001 0001 | @sum |
| 1111 0000 1000 1000 | M=D+M |
| 0000 0000 0001 0000 | @counter |
| 1111 1101 1100 1000 | M=M+1 |
| 0000 0000 0000 0100 | @LOOP |
| 1110 1010 1000 0111 | 0;JMP |
| | (END) |
| 0000 0000 0001 0010 | @END |
| 1110 1010 1000 0111 | 0;JMP |

***This*** is a powerful example. Here are two A-class codes with labels. Look how their address are assigned. And also there are loops, too, making it easy to do the following clarifications.

- Codes like (LOOP) will not be translated. Nor will it be considered one line.(***So what are such lines for?***)
- When an A-Instruction has the same symbol existing inside a parenthesis, say "@LOOP" the rules changed. That line should be translated into the address which the label (LOOP) are **pointing at**(that is, the address of the line that follows the label). In our example you can see it clearly.

- Go to the previous page with a "(really?)". Now you know that there in fact are three types of AInstructions. That thought should go directly into your translator.
- Where is our "Table"? Try to figure that out.

## *Some more details*

- **File names** By convention, programs in binary machine code and in assembly code are stored in text files with ‘‘hack’’ and ‘‘asm’’ extensions, respectively. Thus, a Prog.asm file is translated by the assembler into a Prog.hack file.
- **Constants and Symbols** Constants must be non-negative and are written in decimal notation. A user-defined symbol can be any sequence of letters, digits, underscore (_), dot (.), dollar sign ($), and colon (:) that does not begin with a digit.
- **Comments** Text beginning with two slashes (//) and ending at the end of the line is considered a comment and is ignored.
- **White Space** Space characters are ignored. Empty lines are ignored.
- **Case Conventions** All the assembly mnemonics must be written in uppercase. The rest (user-defined labels and variable names) is case sensitive. The convention is to use uppercase for labels and lowercase for variable names.
- **Predefined Symbols** Any Hack assembly program is allowed to use the following predefined symbols.

| Assembly | RAM address | Machine code |
|---|---|---|
| @SP | 0 | 0000 0000 0000 0000 |
| @LCL | 1 | 0000 0000 0000 0001 |
| @ARG | 2 | 0000 0000 0000 0010 |
| @THIS | 3 | 0000 0000 0000 0011 |
| @THAT | 4 | 0000 0000 0000 0100 |
| @R0~@R15 | 0~15 | |
| @SCREEN | 16384 | 0100 0000 0000 0000 |
| @KBD | 24576 | 0110 0000 0000 0000 |

***(What are these? If you are so eager to know, these are pointers. Please go to VM section for more information on stack and its functions.)***

# Project

In this chapter, you will build an assembler in whatever programming language you like to translate assembly into Hack machine language. Make sure you know all the details of assembly, and start working on it. If you are not so sure, you can go through the following instructions first.

# *Instructions for project Assembler*

The following tests will help you focus again on the language features and understand the proper way to write your assembler.

You are recommended to accomplish the project following a two-tier strategy. First, implement a translator which does not deal with symbols. Then add new functions to your program to deal with symbols.

- A-1 Translate the following assembly codes manually and learn again the grammar in it. (This exercise is for non-symbol translator.)

| **Target Assembly** | **Your Hack machine language** |
|---|---|
| // This line is to check whether it ignores notes | |
| @/*Although not required, this note should also be ignored*/5 | |
| D=A-1/** Some notes are written this way */ | |
| A=D // Another way to write A-instructions | |
| AM=A-1 // You will find such codes powerful in Chapter VM | |
| 0;JMP // Which line is it pointing at? | |

Now the first part of the project will be quite easy for you.

- A-2 Translate the following assembly codes manually and focus on your strategy and sequence when translating. You will need them when you start writing the second part. (This is for assembler dealing with symbols.)

| **Target Assembly** | **Your Hack machine language** |
|---|---|
| @Start // Where is the address? | |
| M=1 | |
| @SUM | |
| M=0 | |
| (LOOP1) //What must you do to this line? And **When**? | |
| @SUM | |
| MD=M+1 | |
| @LOOP1 | |
| D;JGT | |
| @LOOP2 //Careful! (LOOP) can exist before **AND** after @LOOP | |
| //How can you deal with that? | |
| 0;JMP | |

| Target Assembly | Your Hack machine language |
| --- | --- |
| @Whatever  //This line will be skipped | |
| (LOOP2) | |
| @TheEnd | |

- Remark:
- 1. One recommended way of finish part II is first going through the codes without translating any of them. Instead, form a symbol table which includes all the critical information***(What information?).*** And then with the help of that table, translate the codes in the right way.
  2. Exercise above has three "@Xxx" A-instructions, but only two of them need to be assigned addresses.

Now we are sure that you can go to the real project and finish it without confusing yourself. Good luck! (See Chapter 6.5 in the original textbook for your project specifications.)

# Additional Points

There are quite a few tricks in our design of our assembler. Go to the **original textbook Chapter 6.3 & 6.4** for more details.

# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

# Virtual Machine Part I: Stack Architecture

*By Habib Derbyshire*

---

| **Context:**<br>**1. An intermediary language known as VM will make life easier for us** | | | |
|---|---|---|---|
| **Goal: 1. Create a virtual machine abstraction and a language;**<br>**2. Have a way of translating the VM language into assembly;** | | | |
| **Effect**<br>**1. Readers should understand the motivation behind the VM abstraction**<br>**2. Readers should be capable of implementing a translator for our VM language** | **Output**<br>**1. Working VM translations** | **Process**<br>**1. Understand the stack architecture**<br>**2. Implement basic VM arithmetic commands**<br>**3. Understand VM memory access commands**<br>**4. Implement memory access commands** | **Input**<br>**1. .vm file**<br>**2. Test case** |
| **Outside factors: It's pretty hard to build such a language** | | | |

---

## Introduction

### *What is translation?*

If you are reading this chapter, you probably have finished creating an assembler. Congratulations on creating your first translator! Over the course of reading this book, you will realize that computing, from the bare-bones hardware, all the way to the most complicated algorithms, is just a series of translations. By knowing how these translations work, you will hold the secret to understanding ***all*** computing systems. *Some* of these translations are absolutely necessary for the computer to be able to work. In the case of the hack computer, we have to create a translation from machine code to basic Boolean logic, so that when given an instruction, our ALU will be able to conduct a calculation and give us an output. Sometimes, however, a translation is not an integral part of the system's function, but rather an aid to us mere humans. They are introduced at some point, to make life easier for whoever's job it is to work with these translations.

## *Two tier compilation: bridging the gap*

As you should know by now, assembly is very hard for (most) humans to understand, but as far as the computer is concerned, it doesn't get simpler than this. At the other end of the spectrum, there's our high level language called Jack (which we will study in detail in the chapters to come), which is quite easy for us to understand, but in order for it to be meaningful, we have to translate it into assembly code. We've already done translation once, and that wasn't too easy a task (at least not for us!), and that was from assembly to machine code, two very similar languages. Imagine the challenge of translating two languages entirely different in complexity, dimension and syntax! Indeed, such a task is very difficult.

Translating from Jack to assembly also presents another challenge: the resulting assembly will work for the computer we built, but what about another computer, with a different kind of assembly? We would have to go through the process of writing the translator all over again! Surely this is a tedious task.

Since the root of all these difficulties is the disparities between Jack and assembly, why don't we find a way to bridge that gap? If we can create a language that is both "low" enough to translate into machine code easily, and yet "high" enough so that Jack can easily translate into it, it would make our lives (as compiler designers) a lot easier. And the added benefit we get is that if we should wish to compile Jack for any other machine, we can leave the part that deals with Jack alone, and only rewrite the part that deals with the assembly. We call this "in-between language" VM (virtual machine) language, and soon enough, this funny name will make sense to you.

## *Virtual machine*

Differing from the last chapter, in this chapter we are doing a **lot** more than just creating another language. We are creating, out of thin air, a machine capable of many functions, known simply as the virtual machine. Assembly and machine language, as previously mentioned, are a bijection of each other, meaning that they are two sides to the same coin; every sentence in machine language has it's exact equivalent in assembly, and vise versa. Thus, they are capable of doing the same thing, and they do the same thing in the exact same way. If we wanted to create something different, we need more than just a set of **vocabulary** that has corresponding words in the vocabulary of assembly (or machine code), but rather we create an abstraction machine capable of doing different things, and then find a way to do those things in assembly. This abstraction called virtual machine then comes with a language which we can use to make the machine do these different things. The next section will show you just what this virtual machine can do.

# Specifications

## *Virtual machine specification*

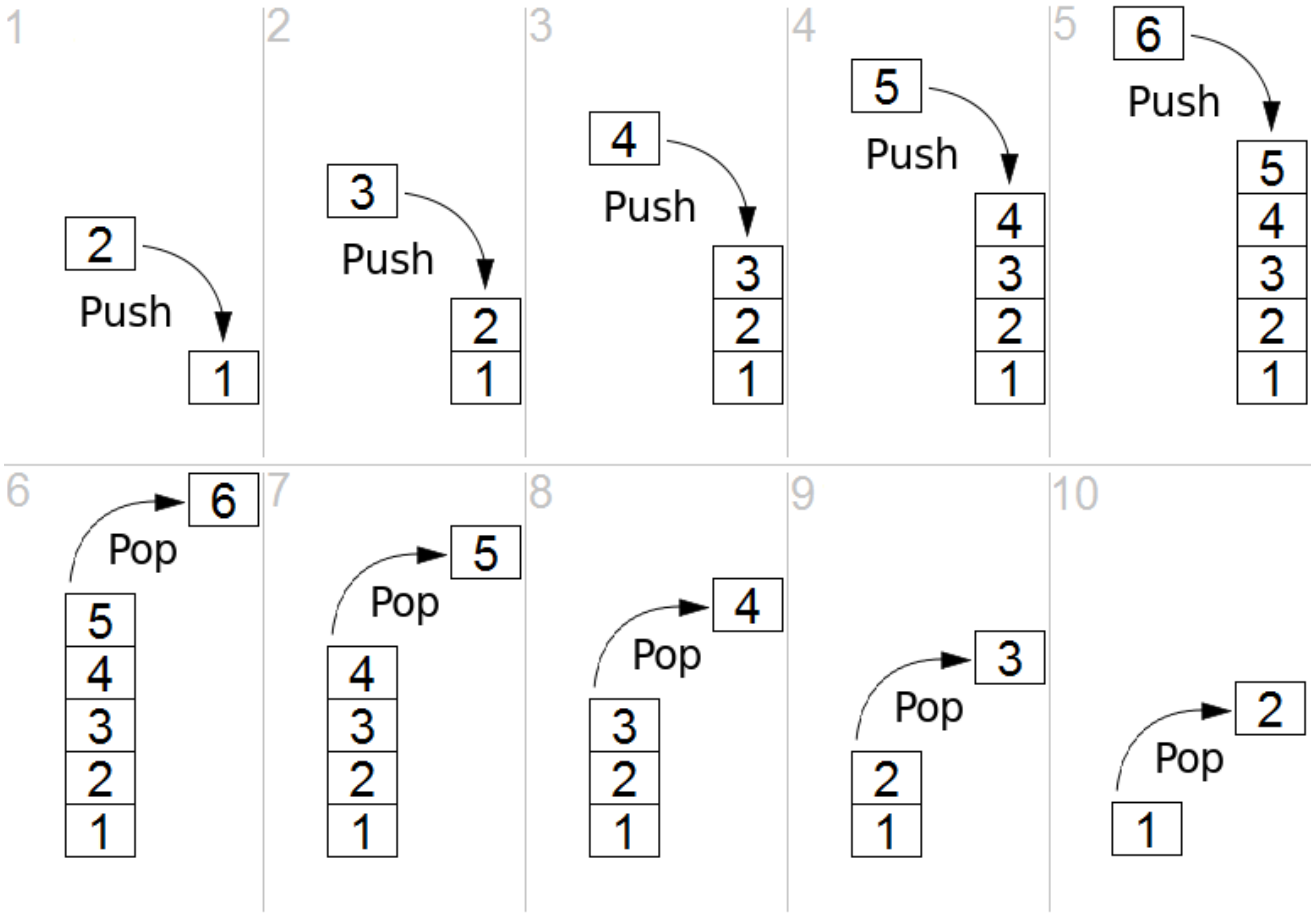
Ok, so this is what the virtual machine needs to be able do:

- Arithemtic operations, adding, subtracting, negation, comparing
  Read from and store into the memory
    Program flow commands, such as executing certain lines of code
    Function calling

At first glance, this seems quite similar to what assembly is capable of doing. However, the way it goes about doing it is quite different. When reading from and writing to the memory, assembly does it in the most basic way possible: by directly reading from given addresses in the memory, sometimes called a register machine. However, in the virtual machine, we have a *stack machine* to deal with memory read/write.

## *Introducing the Stack*

Imagine a very tall stack of books, only the book on the very top is exposed to our view. If we want to see the face of the second book, we will have to take the top book out of the stack, making the second book the topmost book, exposing it's cover for us to see. If we wanted to add a book to the stack, we would have to put it on the top of the stack. This is exactly how the stack machine works. We create a "stack" of registers, each time we can only read or write to the topmost register of the stack. This method of reading memory is sometimes called (quite fittingly) *last in first out*, or LIFO. In the stack machine, the act of reading the value of the topmost register is called *pop*, and the act of adding a value to the stack is called *push*.




All the other stack operations are based around push and pop: when adding two numbers, we first pop the two topmost numbers from the stack, calculated the sum and push it back into the stack

## *Stack arithmetic*

Our virtual machine language comes with a set of arithmetic operators: *add, sub *//addition and subtraction* *neg *//negation* *eq, gt, lt *//compare the size of two numbers* *and, or, not *//boolean operators*

When these operators are executed, they pop one or two numbers from the stack, perform the calculation, then push the result back into the stack.

One example of using 'push': push 3 //pushes the number 3 into the stack

However, all we did was push a static number onto the stack, meaning that the value for the number was explicitly stated in the program. How would we push other numbers, such as variables? Fear not. The virtual machine lives on the hack platform we created, thus it is able to access the memory. We can still read from registers in the memory like we did in assembly. However, in the virtual machine, we have broken the memory into a few segments, which serve different purposes. The memory segments are named "static", "this", "local", "argument", "that", "constant", "pointer" and "temp". In the next chapter you will learn exactly what each of these segments is for. For now, you just have to know in the VM they behave exactly the same, being able of pushing numbers to the stack or receiving popped numbers. The syntax is as follows: pop "someSegment" "index" push "someSegment" "index" where someSegment is one of the 8 segments, and index is an integer.

# Project

This project requires a little bit hack programming tricks. Please review the assembly language before you start writing your program. We recommend you to follow the instructions from the original book **Elements of computing system** Chapter 7.5 to build and test your program step by step.

Note: one things we want to make clear is the distinction between a virtual machine and virtual machine language. The virtual machine is an abstract machine we have created, and the vm language is what we use to program it. Without the language, the machine is unusable. However, we can change the details of the language, such as its syntax, and modify our translator so it can still run on the virtual machine.

# Additional Points

**ATTENTION!**

Our virtual machine is just a sample designed to demonstrate its feature and spirit. A real virtual machine can be much more complicated and quite different. If you want to learn more, we suggest you go find other virtual machines like JVM, Docker, etc, and their implementation.

# References

[1] Nisan, N., & Schocken ,S. (2005). The elements of computing systems: Building a modern computer from first principles. Cambridge, Mass.: MITPress, p148-p179.

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

[3] Wikipedia,Stack (abstract data type), https://en.wikipedia.org/wiki/Stack_(abstract_data_type).

[4] Stack OverFlow: Registers vs Stacks: StackOverFlow, https://stackoverflow.com/questions/164143/registers-vs-stacks.

[5] Squawks of the Parrot: Registers vs stacks for interpreter design, http://www.sidhe.org/~dan/blog/archives/000189.html.

# Virtual Machine Part II : Program Control

*By Zheng Zihan*

---

| **Context:**<br>**You've learnt how the virtual machine works and implemented the basic stack commands, but to make it even more powerful, you'll need extra functions to control your programs and subroutines.** | | | |
|---|---|---|---|
| **Goal:**<br>Implement program control commands and function calling protocol to simplify our programming. | | | |
| **Effect**<br>1. Gain knowledge of how the stack works when calling a subroutine.<br>2. Learn powerful strategy of information processing when building translator | **Output**<br>1. VM translator<br>2. (Program).asm and its testing results | **Process**<br>1. Learn specifications for VM prat II provided. Extra material is provided in hyperlink.<br>2. Follow the project instructions to write your program step by step. | **Input**<br>1. Sufficient details and ideas of how virtual machine is designed.<br>2. Helpful features in Assembly language learnt before. |
| **Outside factors:** | | | |

---

## Introduction

### *Why are we here?*

We are here because we want more.

Remember the loop statement in our assembly? You just made a simple label "(LOOP)"and then you can just use a "@LOOP" to tell your computer to go to run certain lines of codes.

***Is loop statement, or to be more accurate, "(LOOP)" , necessary in a hack program?***

Absolutely not. If the address of the following line is 143, you can just write "@143" instead of using two lines in your assembly.

***Then why are there such statements?***

Because we are lazy, and we don't want to count the lines everytime we want to start a loop. We want the assembler to count it for us. You certainly have built this part in your assember. You know all the tricks.

What you might not fully understand is the answer to the subsequent question.

***Is it the assembler that makes your programming work easier?***

The answer is no, it's not the assembler. It's the mind effort you made when you tried hard to simplify things makes it easier.

***Now you might have already known what we are gonna ask you to do.***

Make it even simpler. Last time you use "@LOOP" to point at one certain line.

This time, we will eventually possess a more powerful language, which enables you to use "call xxx" to instruct your computer to run a subroutine.

***Always remember, the more mind effort you make when designing and manufacturing your tools, the easier the following job will be.***

We would like you to build this amazing function in two steps. First, finish the program flow command, that is,the basic loop statement. And then we go on dealing with the function calling protocol.

# Specifications

## *Program Flow Commands*

We are no strangers to this part. Last time we implemented the if-goto statement by using two lines like:

- @xxx
- D;JGT

This time we want to simplify them so that we can write codes easier.

There are three types of program flow commands:

- **label declaration** Label declaration is quite the same as "(LOOP)". We use the pattern "label ***label***" to point at the address of the following line. ***label*** can be any string consisting of numbers, letters, underlines(_), dots(.) and colons(:), and start with non-numbers.
- **goto statement** Goto statement follows the pattern "goto ***label***". It is a unconditional jump to where the ***label*** is pointing at.
- **if-goto statement** This is a conditional jump. First, pop the boolean value from the top of the stack. If it is true, jump to the line which "***label***" is pointing at. Otherwise continue the following lines.

## *Some Suggestions*

1. Whenever you write a program flow command, make sure that the "***label***" is pointing at a line **inside this program**.
2. Since the subroutines we write may contain same labels, we decided that every ***label*** created in the VM language should gain the counterpart ***functionname$label*** in the assembly.

Quite easy, isn't it? Now you can go to our **project instructions** to start part I. Enjoy!

# *Function Calling Protocol*

## *Backgrounds*

Let us suppose that you are solving a math problem. At one certain step you suddenly realize that a theorem in one of your textbooks can be applied to this part. You go to your study and find that book, find the page that introduce the theorem, use it, memorize the answer, and then come back to your problem to continue.

We are no strangers to such measures we take when solving problems. And this is exactly how your virtual machine is gonna work. So, let us go into details and see how this **"referring to"** method really works.

First, you need to learn that theorem, or you'll never be able to use it. In our program, it means you'll have to write certain codes to implement that function.***~~(Of course you'll do it!)~~***

Next, remember the name of the function, and in which book this theorem is recorded. In our program, this is called "***Function Declaration***".

Then the time comes when you need to use that theorem. So you note down the numbers you need and go to that certain page of that certain book. This, is the so-called "***Function Call***" .

Finally with the help of that theorem you managed to get what you want, so you go back to where you were, maybe your desk, find where you stopped and comtinue. We name this procedure "***Return***".

The above three together is what we called the "***Function Calling Protocol***".

Now you've known all the details of this method. We are now going to make it clearer to you.

## *The Function Calling Protocol*

Remember that all the calculations done by the VM language is conducted in the **STACK**, including the calculations of the main function and the subroutines'. However, the stack itself is totally invisible to our program. That is, the programs cannot see what the stack is like. They can only get access to the stack by using pointers: ARG, LCL, SP, THIS, and THAT. This means that even if ARG is not pointing at the **real** argument segment, the subroutine will not know. It will take what ARG is pointing at as the arguments.

With this feature, we can ***fool*** our subroutines and let all the functions we call happen only in our stack.

The trick is, whenever we want to call a function, say "Multiply" (whose name was already made a symbol), we take the following steps:

## *Function Call*

(VM code: ***call f n*** )(Note: "***f***" is the function's name and n is the number of the arguments it needs.)

1. Push all the arguments that function needs into the stack.
2. Mark where we stop so that when the subroutine is over, we will know where to go to. Use the VM code

   > push return-address-multiply

   The label ***"(return-address-multiply)"*** points at the end of this set of codes.(Given below)
3. Store our previous pointers in stack:

   > push LCL
   >
   > push ARG
   >
   > push THIS
   >
   > push THAT
4. Generate our "*fake*" pointers for our subroutine:

   > ARG = SP-n-5
   >
   > LCL = SP
5. Go to our subroutine:

   > goto multiply
6. Label our return address

   > (return-address-multiply)

And then your subroutine, in this case "multiply", will run successfully in our stack. After that, it is time to consider how this will end and go back to continue the main program. That is, the "***Return***".

## *Return*

(VM code: ***return***)

1. Store the pointer LCL in a temporary variable **FRAME**:

   > FRAME = LCL
2. Store your return address in a temporary variable **RET**:

   > RET = *(FRAME-5)

3. A VM subroutine ***always*** return a value, now you need to pop it out, so:

   > *ARG = pop()

4. Reset all the pointers:

   > SP = ARG+1
   >
   > THAT = *(FRAME-1)
   >
   > THIS = *(FRAME-2)
   >
   > ARG = *(FRAME-3)
   >
   > LCL = *(FRAME-4)

5. Go to return address:

   > goto RET

Now all we lack is to declare a function.

## *Function Declare*

(VM code: ***function f k*** )(Note: "***k***" is the number of ***f***'s local variables)

1. Generate a label for your function.

   > (f)

2. Repeat the following code **k** times to initialize the "local" section:

   > push constant 0

**Warnings:**

1. All the VM (pseudo) codes above should be inplemented in assembly.
2. *ARG represents the value stored in ARG.
3. The caller must push necessary arguments, and wait for the callee to return, but this won't be your concern. We hope this will help you understand why there must be a ARG = pop() in the implementation of ***return***.
4. Be careful with your RET's address.

# Project

***~~(Feel free to ignore this part if you are confident in yourself when building your translator.)~~***

***~~(ALSO feel free to come back when you find it difficult.)~~***

In order to make sure you've understand the new VM codes, we suggest you write the following program on your own.

Exercise VM-1

Write the following programs in VM language. Use the given VM translator to get your assembly and run them in the CPUEmulator to test your codes.

- 1) Mult.vm, the multiply subroutine.

  This subroutine should be declared as ***function mult 2***

  This function can multiply the two arguments on the top of the stack and leave the result there.

- 2) Factorial9.vm, the factorial main function.

  This program should be designed to get the value of 9! and store it into static.5

If your program goes right, then you can go for the real project. Turn to the **original textbook Chapter 8.3~8.5** for more detailed information.

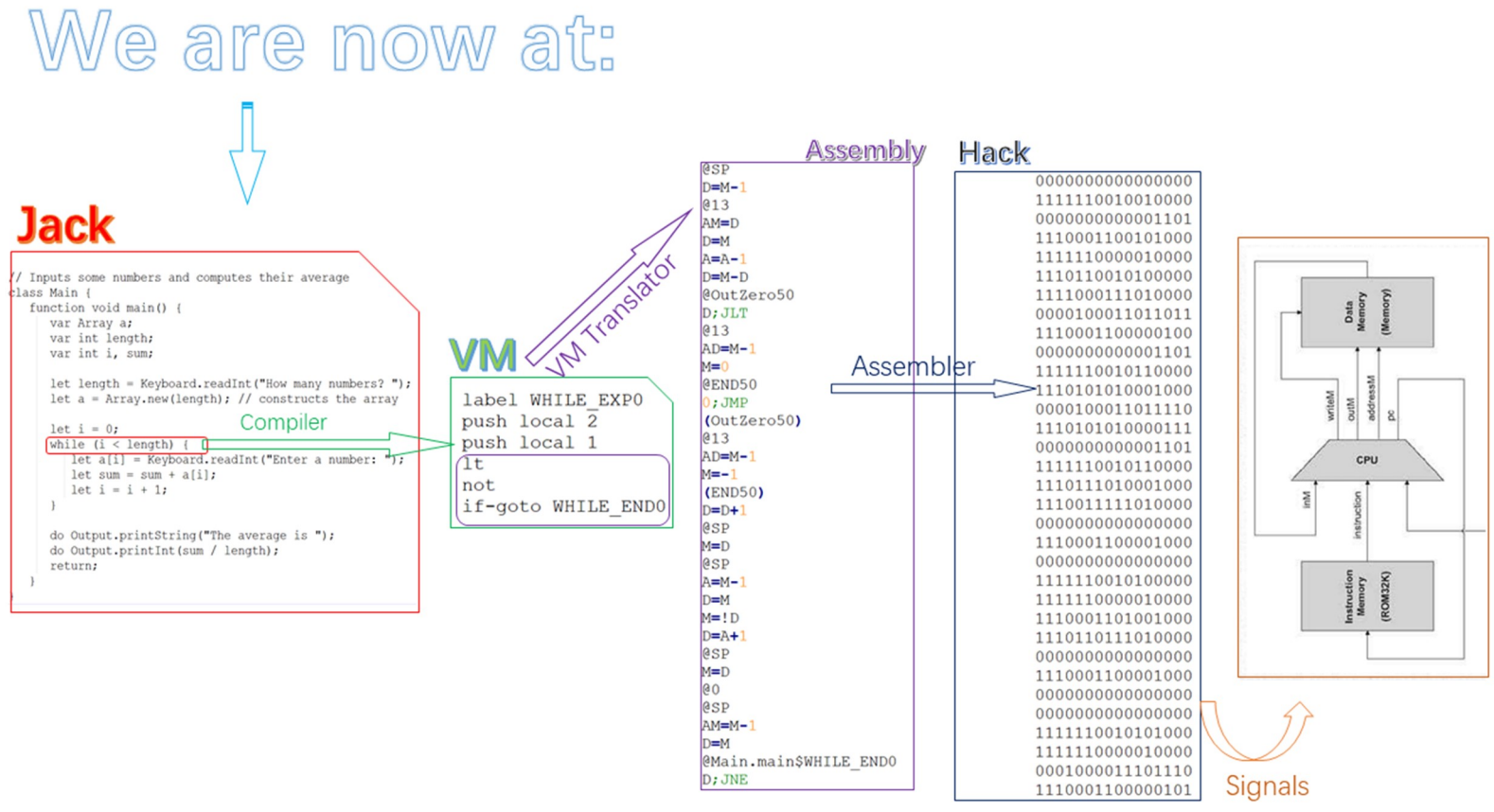
# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

# High-level Language

# And

# operating system



# High-Level Language

*By NaXin*

This chapter is related to **chapter 9 in The Element of Computing system**.

## Logic Model For High-Level Language

| | | | |
|---|---|---|---|
| **Context:**<br>1. We have a working VM language, but it relies too much on hardware platform<br>2. Nowadays we have languages like python and Java that are easy to write | | | |
| **Goal:** | | | |
| **Effect**<br>1. We do not need to use low-level language such as assembly<br>2. You can use a process-oriented, object-oriented approach to build a larger project<br>3. Our own high-level language can run on any hardware that supports it. | **Output**<br>1. A high-level language with such capabilities as the Java language | **Process**<br>1. Build grammars and keywords<br>2. Determine variable types and symbol rules<br>3. Identify grammar rules and errors<br>4. Provide system interfaces<br>5. Build the compiler to connect with VM | **Input**<br>1. Existing VM language rules<br>2. Instructions that can be used |
| **Outside factors: Building a high level language is pretty hard.** | | | |

# Introduction

## *Connection*

So far, what we know is all about the hardware and the lower layer of the system, and usually users will not directly deal with them. Therefore, we introduce here a simple, object-oriented, high-level language called **Jack**.

So why do we need high-level languages?

We use staging strategy to rise from the bottom up step by step, and they are all dealing directly with the logic gates. Imagine if our underlying architecture changed, a specific program we wrote would not be able to run on the new architecture. To solve this problem, we can design an "idealized" high-level language, as long as the "compiler" provided by the underlying architecture can successfully compile the high-level language into the language of this architecture, and the language we design can be used in all architectures, means to run on them.

## *Hign-Level Language*

A simple way to define it is that high-level languages are a system off the hardware that are written in a way that is easier for people to understand.

Departing from the hardware system means that its instructions are **independent** and are just **convenient representations** that are **easier to express**. At the same time, the high-level language code is more **readable**. An example is declaring an array (as a data structure). The high-level language only cares about how to use it. It doesn't have to worry about how to save the array in memory.

The readability is: The encoding process can be simply understood as telling the machine what to do next.

## *Category*

We briefly introduce the classification of four high-level languages:

1. **Imperative language**: The basis is a computable Turing machine model that simulates “data storage/operations”, which is a way to meet the current mainstream architecture.
2. **Functional languages**: languages based on mathematical function concepts and mappings, such as Haskell
3. **Logical language**: language built based on a set of known formal logic systems
4. **Object-Oriented**: Most high-level languages now have object-oriented as a basic model

## *Jack Language*

One feature of Jack language is that it is very similar to Java, but its grammar structure (or Syntax) is very simple. Using **context-free grammar** to describe the language clear enough requires only a little space(**The Element of Computing system Picture 10.5**).

Of course, even so, Jack still has the same ability as other languages. This means that anything that can be done in other languages can be implemented using Jack. (Of course it may not be so fast)

# Specifications

This section refers directly to **The Element of Computing system chapter 9.2** . We recommende readers also read the contents of the **9.1** section here. So that you will have a good understanding of the syntax of the high-level language.

# Project

After learning about the Jack language, you can build your own simple Jack program. You can see this in **The Element of Computing system chapter 9.5**.

The project in Chapter 9 does not require much coding. but you need to have enough knowledge of Jack to start building Jack's compiler in the coming chapter.

# Additional Points

We believe that after reading **chapter 9.2** of the book, you already have a fairly good understanding of Jack's language. Let's take a look at some of the parts that we think are important and interesting.

## *Subroutine call*

In the VM language, we already have the concept of a subroutine call. For example, when I need the product of two numbers, I don't need to implement multiplication **on the spot**. Instead, I call a function, which reduces the workload. And moreover, the code is also simpler. This involves the **modularization** issue. We divide the problem into some typical modules. Each module runs step by step to get the results we want. The advantage is that if we need to use a certain part in the future, we can directly call this module without re-implementation, code reuse.

## *Expression*

In Jack's expression section, we find that when two expressions are connected by one symbol we get another expression. Here is the idea of the so-called **"recursion"** in which expressions can continuously generate expressions.

## *Standard Library*

The standard library is the function **provided by the system** itself and can be called directly. We will explain this in detail in Chapter 12.

# References

[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] Kindler, E.; Krivy, I. (2011). "Object-Oriented Simulation of systems with sophisticated control". International Journal of General Systems: 313–343.

[3] Hopcroft, John E.; Ullman, Jeffrey D. (1979), Introduction to Automata Theory, Languages, and Computation, Addison-Wesley. Chapter 4: Context-Free Grammars, pp. 77–106; Chapter 6: Properties of Context-Free Languages, pp. 125–137.

# Syntax Analysis

*by NaXin*

## Logic Model For Syntax Analysis

| | | | |
|---|---|---|---|
| Context:<br>1. After chapter 9, we now have a "high level language" in our mind<br>2. To implement this, we need to tokenize and parse Jack code to generate a syntax tree for Chapter 11 | | | |
| Goal: Generate the syntax tree to finish the first part of the compiler | | | |
| Effect<br>1. Use advanced tools to make your job easier<br>2. Understand why the book recommend us to take two steps to do this.<br>3. Understand the data structure——Tree | Output<br>1. Finish the first part of the compiler of high level language. | Process<br>1. Figure out what tokenizing and parsing are<br>2. Learn to use python to build the XML tree<br>3. Learn how to write regular expressions | Input<br>1. Nand2tetris files<br>2. Some document about syntax and regular expressions to help |
| Outside factors: High-Level language is much more complex than the VM compiler | | | |

# Introduction

## *Connection*

In the previous chapter we learned about the Jack language. In this chapter we will begin to **build the Jack language compiler**, a translator from jack to the VM language whicih we are familiar with. The process is divided into two parts, parsing and code generation. The content of parsing (or **syntax analysis**) is introduced in this chapter.

Why do we need to analyze grammar, and how do we analyze grammar? For previous assembly language and VM language, their grammatical forms are closer to "instructional", giving only **a few specific characters at a time**. In the high-level

language stage, due to its complexity of form, the structure must be given by a context-free grammar. In such cases, it is very important to analyze the grammar. The way we analyze grammar is:

1. tokenize

   breaks grammar into one character or string, each independent.

2. grammar analysis.

   Form a grammar structure tree through analysis of rules in CFG.

### *Tokenize*

Tokenize is to cut the code into the smallest unit that composes it. Please refer to **The Element of Computing system chapter 10.1.1** directly. It is easy to understand.

### *Gramma analysis*

First of all, it is necessary to make clear the grammatical structure of Jack's language. This part has already been embodied in Chapter 9, and the book has described the specific content very clearly. Here we talk about context-free grammar. All the statements in Jack are generated by it. Some of the statements can generate other statements. And through this, they continue to generate Jack's language.

After understanding how the Jack language is produced, let's look at how to parse it. First we need to understand the **grammar parsing tree**: just like a context-free language, if we consider the source of a rule **as a root**. Let's treat **output results** as leaves, then this forms our grammar analysis tree. Since this process can be arbitrarily repeated, a **multi-level syntax analysis tree** is obtained.

The parse tree in this case is a tree constructed from Jack characters. Each non-final statement can generate a **subtree**. The writing process if the process of constructing subtrees.

## Specifications

This section refers directly to **The Element of Computing system chapter 10.2** . We recommend readers also read the contents of the **10.1** section here. Because we believe the book has been clear enough about how to do grammar analysis.

## Project

After the above introduction, you should be more familiar with how to analyze the syntax in two steps. If this part is indeed difficult, please refer to the help file given in **References**.

## Additional Points

In this chapter, we add the following content for the reader to see.

### *Context-free languages*

Context-free languages are similar to regular languages and are used to describe a string generation rule. The resulting string collection is what we call "language." And it's basic idea is to start from a variable and use a certain set of rules to generate a target string.

In automata theory, context-free language is a very important part of a class of automata. It can be equivalent to what is called push-down automaton. For more on this part please see the references.

## *Data Structure - Tree*

The **tree (k-tree)** is a classic data structure. Many of the problems we encountered in programming can be reduced to a tree. For example, the problem of expression evaluation, any expression can be converted into a tree. Each tree has a ** root**, and they have many** children**. At the same time, a child node can also be a tree root with his own** subtree**. The syntax parse tree here is a special case of the Jack language.

# References


[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] NaXin;Help file for Project 10(https://hackmd.io/s/SkmfwFKpz)

[3] Andrei Popov and Ekaterina Enikeeva. 2017. Template Search Algorithm for Multiple Syntactic Parses. In Proceedings of the International Conference IMS-2017 (IMS2017). ACM, New York, NY, USA, 164-170. DOI: https://doi.org/10.1145/3143699.3143732

# Code Generation

*by NaXin*

## Logic Model For Code Generation

| | | | |
|---|---|---|---|
| Context:<br>1. In Chapter 10, we have completed the grammar analysis of high-level language and built a parse tree.<br>2. On the other hand, the understanding of the Jack language has also been greatly improved, preparing for the final project. | | | |
| Goal: Compilation of the Jack language compiler, write the code writer section | | | |
| Effect<br>1. After building the xml tree, learn how to traverse and access the xml tree, which is very simple to do in python<br>2. Complete the language compiler build so that the entire nand2tetris only has operating system related items | Output<br>1. Complete project 11, a Code Generator. | Process<br>1. Figure out how recursive algorithm works<br>2. Analyze the process of code generation<br>3. Review the VM language<br>4. Writing the code with python | Input<br>1. Nand2tetris files<br>2. Sample code in the book to do some tests |
| Outside factors: Generally, project 11 is very difficult. The expression part and the amount of code is also very large. | | | |

# Introduction

## *Connection*

In the previous chapter we analyzed Jack's grammar and got a grarmmar tree. In this chapter we use the tree to generate the VM code. This translation process has two main parts:

1. Data storage
   Data storage means that we need to store all variables in the program, including their name, type, and so on. The storage of the array and the object is a bit more complex.
2. Command translation
   The command translation phase is divided into two parts. One is the evaluation of the expression and another is the control of the program flow. In the previous chapter we have revealed that expression evaluation is a tree. And process flow is some jedgement and loop.

### *Data storage*

In the data storage, the data is saved using several "symbol tables", among which there are **class variable table** and **function variable table**, which is used to store the name type, etc. Please read **The Element Of Computing system chapter 11.1.1** directly in this part. In array processing, the array of Jack in the form of **str[i]**. It means to access the number i element in array **str**. In the implementation process, we will see the array as a **Pointer**, which points to a memory address (stores this address directly). We do this with directly access to the address **array+i** in the memory. In the processing of classes, since VM has provided us with access methods separated by **point(.)**, we can use this method to generate VM code directly.

### *Command Translation*

1. Expression evaluation
   First, after the previous chapter, we have obtained the expression tree. Through this tree of expressions we iteratively generate the VM code. Note that all operations can be completed by calling the function.
2. Process control
   Process control can be implemented using the VM's **goto** and **if-goto**. Note that the label names should be different from each other.

## Specifications

This section refers directly to **The Element of Computing system chapter 11.2** . We highly recommend readers also read the contents of the **11.1** section here. Beacause we believe the book has been enough clear about how to do grammar analysis.

## Project

The overall structure of the code generation section in this chapter is very similar to the previous chapter. The previous chapter built a parse tree. In this chapter, we traverse the tree. The translation method given above and in the book should be enough hints for you. If this part is difficult, please refer to References.

## Additional Points

In this chapter, we add the following content for the reader.

### *Pointer*

Pointers are a very important part of the C programming language. Even if pointers are no longer available to programmers in languages such as Java (because they are **too complex**), they still play an important role at the bottom of the system. A pointer holds an **integer variable**, except that this integer is a definite address in the **RAM**. It means it **points to** the value in the address. With pointers we can implement the **array** data structure by saving the array's **initial address** as a pointer, and by adding an abstract position to this value we can get the exact position of every elements in the array.

## *Object Storage*

Through Chapter 9, you should have enough knowledge of object orientation. And the object saved in Jack is based on pointers. Variables in the object are only generated after built using **Constructor**.

# References


[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] NaXin;Help file for Project 11(https://hackmd.io/s/r1sl3TW0f)

[3] Liskov, Barbara; Zilles, Stephen (1974). "Programming with abstract data types". Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. SIGPLAN Notices. 9. pp. 50–59.

[4] Automaton - Definition and More from the Free Merriam-Webster Dictionary http://www.merriam-webster.com/dictionary/automaton

# Operating System

*by NaXin*

## Logic Model For JackOS

| | | | |
|---|---|---|---|
| Context:<br>1. Based on the compiler we have built, we will provide some system interfaces for users<br>2. The Jack OS can be simply understood as some system calls for the user<br>3. The existence of an operating system is so important that it is the soul of a high-level language. | | | |
| Goal: Implement a very simple operating system with some interfaces | | | |
| Effect<br>1. Understand the importance of the operating system and become interested in this subject | Output<br>1. Complete project 12, a simple implementation of a operating system | Process<br>1. Learn about how the JackOS is designed.<br>2. Read the book to learn to implement each class<br>3. Write the code with Jack language | Input<br>1. Nand2tetris files<br>2. Essays about Bresenham's Algorithm to draw a line on the screen. |
| Outside factors: Operating system is a very important interface between us and the computer, as today the OS we are using provides everything we need | | | |

# Introduction

## *Connection*

Congratulations! Through the previous two chapters you have successfully built your own Jack language compiler. The next chapter will be easier. Let's form a simple "operating system" by building some **system calls**. So first, let's see what is an operating system. In simple terms, an operating system is defined as interfaces between users and hardware devices. Users do not deal directly with hardware like we do. So we need to provide some interfaces(**API**) for users to use. In JackOS, it is simplified to some system calls. System calls are very important in the high-level language stage, because our Jack language has not connected to specific hardware devices. Without connection with hardware, our language cannot be truly seen. So system calls can be understood as the soul of high-level language.

## *System Calls*

There are many types of system calls that we need to implement, including math operations, screens, and keyboards. For this section please read **The Element of Computing system chapter 12.1** directly. Moreover, the book has given very detailed implementation methods.

# Specifications

All of the functions that need to be done has been listed in **The Element of Computing system chapter 11.2**. We believe the book has been enough clear about how to do grammar analysis.

# Project

The project in this chapter is to achieve all the system call functions in the book. Because the book has given a more detailed implementation instructions, this part should not be difficult to achieve. If you have questions about certain places, ask questions in the github link in references.

# Additional Points

In this chapter, we add the following content for the readers.

## *Operating System Related*

The functions we implement are actually in a very special and simple "operating system". In a modern operating system, its content is far beyond these. Usually an operating system's system calls provide these major categories of interfaces:

1. Process control
   At present, typical common operating systems support **multiple processes**, and they can support multiple applications running at the same time. Our JackOS obviously has only one process running.
2. File reading and writing
   We should all be familiar with this function, but for the Hack computer we built, it does not have a **ROM storage**, and naturally there is no file structure.
3. System control
   Since there are many **resources (hardware devices)** for a computer, we need to control how applications use these resources.
4. Memory Management
   This point, our Jack language has also been implemented to cover the operation of allocating and reclaiming memory to applications.
5. Network
   This part has functions like getting IP address, establish connection and other operations.

Through this, we can see that JackOS is a very simple system. Perhaps you have a considerable interest in the operating system. We provide you with an operating system designed for education called **xv6**. This is the operating system that MIT uses for teaching. Students add functionality to it to improve the experience. For more information see References.

# References


[1] Nisan, N., & Schocken, S. (2005). The elements of computing systems : Building a modern computer from first principles. Cambridge, Mass.: MIT Press.

[2] Stallings (2005). Operating Systems, Internals and Design Principles. Pearson: Prentice Hall. p. 6.

[3] Dhotre, I.A. (2009). Operating Systems. Technical Publications. p. 1.

[4] Wikipedia for xv6(https://en.wikipedia.org/wiki/Xv6)

# Appendix

## About Us

### *Chen Yihao*

I learned a lot from this course Computational Thinking and System Design. I got to konw the overall hierarchy of the computer system and also understand the significance of computational thinking. In additon, the speeches from experts in various areas also broadened my horrizons.

### *Yuan Lekang*

In constructing the hole architecture of a jack computer, we've learnt important ideas of computational thinking, such as using a single word to build everything, doing staging and modularity to build big system, dividing the book into several stages and think about their connections. These thinking methods can be applied into all the subjects and create a much better understanding. Besides, this class taught me we can understand what we learn better by writing, and it's important to cooperate with others.

### *Han Zhilei*

During this semester's study,I went through the main part of a computer system and build it from the very bottom. Instead of focusing on the details, simplifications were applied and it emphasis on the connections between the parts of the system.Besides,the process of writing this publication also teach me a good lesson:It shows how powerful teamwork is,and how learning process should be by producing this very book.

### *Habib Derbyshire*

The insights I gained through the study of this course far exceded any theoretical computing class I have taken. Not only did I learn HOW computers work, but also WHY they are built this way. To manage the mental task of implementing such complicated, one is forced to adopt a computational method of thinking, constantly keeping in mind context, goals, measurable effect, output and process. Additionally, I learned a valuable lesson on teamwork, and what it means in this day and age.

## *Zheng Zihan*

In the procedure of implementing a fully functional computer from basic circuits, not only did we learn the complete structure of a computer, but critical computational thinking method of dealing with complicated problems. The course also provided us with sufficient documents, powerful managing and cooperating tools and thought-provoking ideas that would influence our future studying and researching. We also learnt how those discrete parts of knowledge can be put together to generate greater things.

## *Na Xin*

After learning computational thinking lessons, I gained a lot of new knowledge and new skills. I benefited from finishing all the nand2tetris projects. I understand how a computer links the bottom and the upper layer. I understand the application of automata theory in computers by combining the knowledge of automata I have learned. I also learned to use many tools and learned to work together with others to complete a project (such as the publication). This experience has brought me a lot of new challenges, and being able to complete this project makes me very excited.

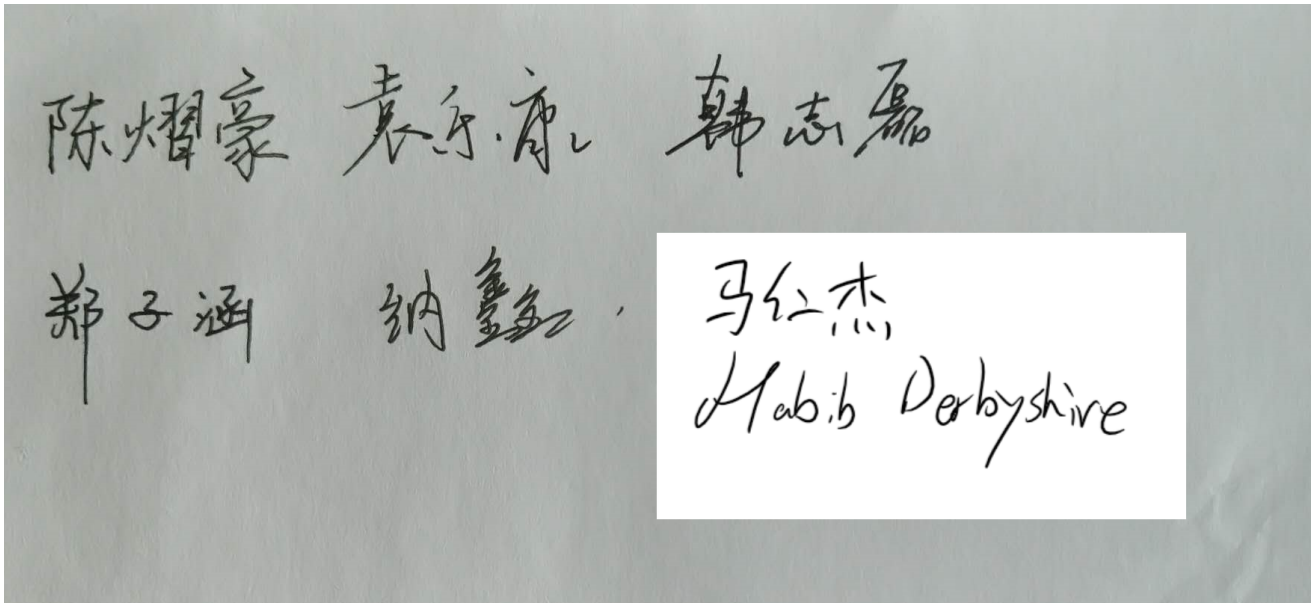
# Letter of Authorization

I hereby give my permission to Professor Ben Koo at Tsinghua University, Beijing, China, the right to use learning and academic data generated by me during my studies of the course “Fundamentals of Computational Thinking and System Design” (course code 01510223) during the Spring semester of 2018 at Tsinghua University for academic and research purposes. Learning and academic data relevant to the course include contents and information generated on the course site www.toyhouse.cc and connected data on Holors.org, Phabricator, Github, and other online sources. I hereby give my permission to Professor Ben Koo to use the data for publication, content analysis, user behaviour analysis and other academic activities.

Professor Ben Koo acknowledge the importance in protecting the signee’s personal information, and will take necessary actions and precautions.

This authorization shall be effective immediately.

Signatures:




Date: June 12th, 2018. Beijing, China.