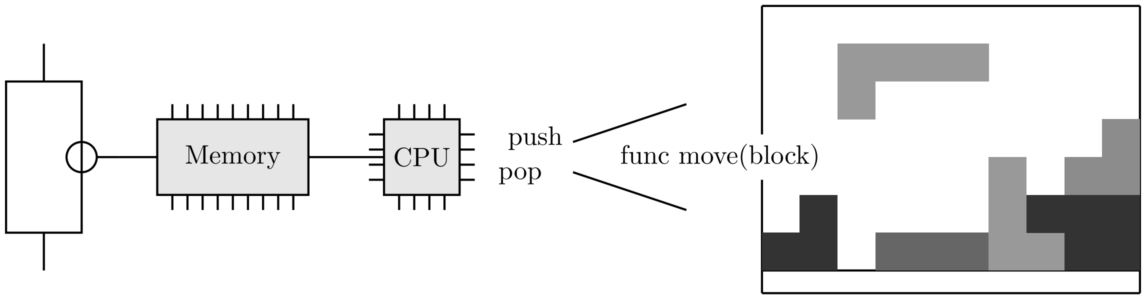


NAND 2 TETRIS

From simple logic gates to the popular game TETRIS

Author:
Eldar Omerovic
https://www.github.com/omeldar

Project Repository:
https://www.github.com/omeldar/nand2tetris

Credits:
I completed this project during my NAND2TETRIS class
at the Lucerne University of Applied Sciences and Arts.

February 2024

Contents

1 Introduction 6
1.1 Prerequisites and Tools . . . 6

2 Chapter 1 - Boolean Logic 7
2.1 Boolean algebra . . . 7
2.2 Why is NAND so special . . . 7
2.3 Logic Gates . . . 7
2.4 Gates, Multiplexer, Demultiplexer . . . 8

3 Chapter 2 - Boolean Arithmetic 10
3.1 Binary Numbers . . . 10
3.2 Binary Addition . . . 10
3.3 Signed Binary Numbers . . . 10
3.4 Half-Adder and Full-Adder . . . 10
3.4.1 Half-Adder . . . 10
3.4.2 Full-Adder . . . 11
3.5 Multi-Bit Adders . . . 11
3.6 Incremeter . . . 11
3.7 Arithmetic Logic Unit (ALU) . . . 11
3.7.1 Examples . . . 12

4 Chapter 3 - Memory 13
4.1 Overview . . . 13
4.2 Clock Signal . . . 13
4.3 Data Flip-Flop (DFF) . . . 13
4.4 Register . . . 13
4.5 Random Access Memory (RAM) . . . 13
4.6 Counter . . . 14
4.7 Key Concepts . . . 14

5 Chapter 4 - Machine Language 15
5.1 Overview . . . 15
5.2 Hardware Structure . . . 15
5.3 Hack Machine Language Specification . . . 15
5.4 Control Instructions . . . 15
5.5 Example Program . . . 15
5.6 Screen . . . 16
5.7 Keyboard . . . 16

6 Chapter 5 - Computer Architecture 17
6.1 Overview of the Von Neumann Architecture . . . 17
6.2 Key Components of the Hack Platform . . . 17
6.3 Stored Program Concept . . . 17
6.4 Fetch-Execute Cycle . . . 17
6.5 Hack Platform Execution and Fetch Behavior . . . 17
6.6 Hack CPU Architecture . . . 17

6.6.1 Instruction Decoding and Execution . . . 18
6.6.2 Example Instruction Analysis . . . 18
6.6.3 Additional Explanation on Data Flow . . . 19
6.6.4 Example Program Execution . . . 19

7 Chapter 6 - Assembler 20
7.1 Machine language . . . 20
7.2 Symbols . . . 20
7.3 Hack Assembly Specification . . . 20
7.3.1 Type A-Instructions . . . 20
7.3.2 Type C-Instruction . . . 21
7.4 Labels . . . 21
7.5 Translating assembly to binary . . . 22
7.5.1 Handling instructions . . . 22
7.5.2 Handling symbols . . . 22
7.6 Assembler Implementation . . . 22

8 Chapter 7 - VM I : Stack Arithmetic 24
8.1 Stack and Stack Arithmetic . . . 24
8.2 RAM Usage . . . 24
8.3 Understand Stack Operations . . . 25
8.4 VM Implementation on the Hack Platform . . . 25
8.4.1 VM Abstraction . . . 25
8.4.2 Memory Segments . . . 25
8.4.3 Translating VM code . . . 26

9 Chapter 8 - VM II : Program Control 27
9.1 Implementation of Branching Commands . . . 27
9.2 Implementation of Function Commands . . . 28
9.3 Responsibilities of the Parser and CodeWriter . . . 28

10 Chapter 9 - High-Level Language (Jack) 29
10.1 Introduction to Jack . . . 29
10.2 Grammar for Formal Languages . . . 29
10.3 CYK Algorithm for Context-Free Languages . . . 29
10.3.1 Example of CYK algorithm . . . 30

11 Chapter 10 - Compiler I : Syntax Analysis 32
11.1 Key Concepts . . . 32
11.2 Implementation Details . . . 32
11.2.1 JackAnalyzer Module . . . 32
11.2.2 JackTokenizer Module . . . 32
11.2.3 CompilationEngine Module . . . 32
11.2.4 Example XML Output . . . 33
11.2.5 Important Tables and Concepts . . . 33
11.3 Translating Jack to VM code . . . 33

12 Chapter 11 - Compiler II : Code Generation 35
12.1 Key Concepts 35
12.1.1 Code Generation 35
12.1.2 Symbol Table 35
12.1.3 Data Translation 35
12.1.4 Handling Variables, Arrays and Objects 35
12.1.5 Commands Translation 35
12.2 Implementation 36
12.3 Translation Steps 36
12.3.1 Tokenization 36
12.3.2 Parsing 36
12.3.3 Symbol Table Management 37
12.3.4 Code Generation 37
12.3.5 VM Code Output 37
12.4 Summary of all forms 37

13 Chapter 12 - Operating System 38
13.1 The 8 main classes 38
13.1.1 Memory Management 38
13.1.2 Array Handling 38
13.1.3 Mathematical Operations 38
13.1.4 String Manipulation 38
13.1.5 Output Handling 38
13.1.6 Screen Management 38
13.1.7 Keyboard Input 38
13.1.8 System Operations 39
13.2 Key Algorithms 39
13.2.1 Multiplication Algorithm 39
13.2.2 Division Algorithm 39
13.2.3 Line Drawing Algorithm 39
13.2.4 Heap Management 40
13.2.5 Text Output 40

14 Chapter 13 - System Models (Theory) 41

15 Chapter 14 - Predictability (Theory) 42

16 Conclusions 43

17 Appendix 44
17.1 VM Translation to Hack table 44
17.2 VM Implementation of Function Commands 51
17.2.1 call 51
17.2.2 return 51
17.3 Example Process of VM Parser and CodeWriter 52
17.4 Operating System Helpers 53
17.4.1 Multiply Helper Implementation 53
17.4.2 Divide Helper Implementation 53

17.4.3 Line Drawing Helper 53
17.4.4 Alloc Helper 54
17.4.5 Print String Helper 54

1 Introduction

The NAND2TETRIS project is a valuable resource for IT professionals and anyone interested in computer science. But what makes it so special? It's a project where you get to build a digital system that can run the game TETRIS. Unlike many other educational resources, NAND2TETRIS guides you through building everything from the basic logic gates to the final software. The goal is to help you understand how digital systems work right from the ground up. Usually, we only learn about certain parts of this process, but with NAND2TETRIS, you get to experience it all firsthand.

1.1 Prerequisites and Tools

You do not need a lot of experience or knowledge in computer science to start with NAND2TETRIS. It's easier the more experience you have, but you can complete this project without any experience or knowledge at all. There is a lot of documentation on their official website and youtube videos.

There are certain tools (software) which you might want to install to test your implementations here https://www.nand2tetris.org/software.

2 Chapter 1 - Boolean Logic

2.1 Boolean algebra

Boolean algebra deals with binary values, such as true/false, 1/0, or yes/no. In this context, we will use 1 and 0. Since computers operate exclusively with 0s and 1s, it is essential to formulate and analyze Boolean functions when designing computer architecture. The most straightforward method to specify a Boolean function is by listing all possible input variables along with their corresponding outputs, a representation known as a "truth table."

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

The first three columns display the various values of the input variables. For each of the 2^n possible combinations $(v_1, v_2, \ldots, v_n)$, the last column shows the corresponding value of $f(v_1, v_2, \ldots, v_n)$.

Another way to define a Boolean function is by using Boolean operations applied to the input variables. The operations x AND $y = 1$, x OR $y = 1$, and NOT $x = 1$ are common Boolean operations. These can be combined to create expressions such as (x AND y) OR z, which means that either both x and y need to be 1, or z needs to be 1 for the output to be 1.

There is an alternative representation for these Boolean operations that we will be using throughout this documentation. For instance, $x \cdot y$ represents x AND y, $x + y$ represents x OR y, and $\overline{x}$ represents NOT x.

Every boolean function can be defined by using only three boolean operations: AND, OR and NOT.

2.2 Why is NAND so special

Using NOT and AND, we can construct the NAND operation. Additionally, with NAND, we can construct every other logic gate. For example:

$$x \text{ OR } y = (x \text{ NAND } x) \text{ NAND } (y \text{ NAND } y)$$

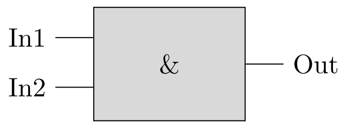
This is crucial because it implies that if we can construct hardware capable of performing NAND operations, we can combine these hardware devices to execute all Boolean operations by assembling them in specific configurations.

2.3 Logic Gates

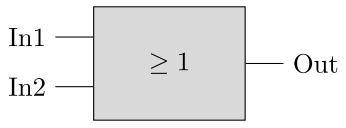
Logic gates are physical devices that implement Boolean functions. When a Boolean function f takes n variables and produces m results, the corresponding logic gate has n inputs and m outputs. The simplest logic gates consist of very small switches, known as transistors, which, when configured in a specific topology, create a functioning logic gate.

Folgend sehen wir die verschiedenen Standardsymbole für AND, OR and NOT.

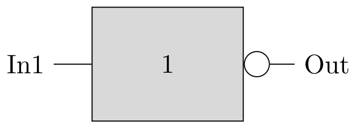
AND gate



OR gate



NOT gate



2.4 Gates, Multiplexer, Demultiplexer

NOT-Gate

- **Function:** Inverts the input bit.
- **Implementation with NAND:**

$$\text{NOT}(A) = A \text{ NAND } A$$

A	NOT(A)
0	1
1	0

AND-Gate

- **Function:** Outputs TRUE only if both inputs are TRUE.
- **Implementation with NAND:**

$$\text{AND}(A, B) = \text{NOT}(A \text{ NAND } B)$$

$$= (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$$

A	B	AND(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

OR-Gate

- **Function:** Outputs TRUE if at least one input is TRUE.
- **Implementation with NAND:**

$$\text{OR}(A, B) = \text{NOT}(A) \text{ NAND } \text{NOT}(B)$$

$$= (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$$

A	B	OR(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

NAND-Gate

- **Function:** Outputs FALSE only if both inputs are TRUE.
- **Implementation with NAND:** A NAND gate is already a basic gate and does not require further implementation.

A	B	NAND(A,B)
0	0	1
0	1	1
1	0	1
1	1	0

NOR-Gate

- **Function:** Outputs TRUE only if both inputs are FALSE.
- **Implementation with NAND:**

$$\text{NOR}(A, B) = \text{NOT}(A \text{ OR } B)$$

$$= (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$$

A	B	NOR(A,B)
0	0	1
0	1	0
1	0	0
1	1	0

XOR-Gate (Exclusive OR)

- **Function:** Outputs TRUE if one input is TRUE and the other is FALSE.
- **Implementation with NAND:**

$$\text{XOR}(A, B) = (A \text{ NAND } (A \text{ NAND } B))$$

$$\text{NAND } (B \text{ NAND } (A \text{ NAND } B))$$

A	B	XOR(A,B)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR-Gate (Exclusive NOR)

- **Function:** Outputs TRUE if both inputs are the same.
- **Implementation with NAND:**

$$\text{XNOR}(A, B) = \text{NOT}(\text{XOR}(A, B))$$

A	B	$\text{XNOR}(A, B)$
0	0	1
0	1	0
1	0	0
1	1	1

Multiplexer (MUX)

- **Function:** Selects one of several input signals and forwards the selected input to a single output line.
- **Implementation with NAND:**

$$\text{MUX}(A, B, S) =$$

$$(A \text{ NAND NOT}(S)) \text{ NAND } (B \text{ NAND } S)$$

Where S is the select signal. The MUX outputs A when S is 0, and outputs B when S is 1.

A	B	S	$\text{MUX}(A, B, S)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

Demultiplexer (DEMUX)

- **Function:** Takes a single input signal and selects one of many data-output-lines, which is to be sent to the single input.
- **Implementation with NAND:**

$$\text{DEMUX}(A, S)$$

$$= (A \text{ NAND NOT}(S)), (A \text{ NAND } S)$$

Where S is the select signal. The DEMUX routes the input A to the first output when S is 0, and to the second output when S is 1.

A	S	$\text{DEMUX}_0(A, S)$	$\text{DEMUX}_1(A, S)$
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Multiplexer (MUX)

A multiplexer, or MUX, is an electronic component with multiple input signals and one output signal. It selects one input and forwards it to the output based on control signals.

Purpose of a MUX

- **Data Line Selection:** Combines data from multiple sources into a single line.
- **Resource Efficiency:** Allows efficient use of lines.
- **Time Division Multiplexing:** Sends different data streams alternately over time.

Demultiplexer (DEMUX)

A demultiplexer, or DEMUX, is the opposite of a MUX. It has one input signal and multiple outputs. It forwards the input to one of the outputs based on control signals.

Purpose of a DEMUX

- **Signal Line Distribution:** Distributes a signal to multiple lines.
- **Data Distribution:** Forwards data packets to different destinations.
- **Parallel Processing:** Splits a serial input into multiple parallel outputs.

3 Chapter 2 - Boolean Arithmetic

3.1 Binary Numbers

In contrast to the decimal system, which is based on 10, the binary system is based on 2. When given a binary pattern, such as 10011, and told it represents a whole number, the decimal value is calculated as follows:

$$(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

In general, let $x = x_n x_{n-1} \ldots x_0$ be a sequence of digits. The value of x in base B, denoted as $(x)_B$, is defined as:

$$(x_n x_{n-1} \ldots x_0)_B$$

$$= x_n \cdot B^n + x_{n-1} \cdot B^{n-1} + \ldots + x_1 \cdot B + x_0$$

$$= \sum_{i=0}^{n} x_i \cdot B^i$$

For binary base $B = 2$, we write $(x)_b$. For decimal base $B = 10$, we write $(x)_d$, and for hexadecimal base $B = 16$, we write $(x)_h$.

3.2 Binary Addition

Binary addition can be performed digit by digit from right to left, similar to decimal addition. The rightmost digits, called the least significant bits (LSB), are added first. The resulting carry bit is then added to the sum of the next significant bit pairs. This process continues until the most significant bits (MSB) are added.

For example:

$$\begin{gathered} x : 111000000111 \\ y : 000111000111 \\ \text{Carry} : 000000000001 \\ x + y : 111111111000 \\ \text{Overflow} : \text{none} \end{gathered}$$

3.3 Signed Binary Numbers

A binary system with n digits can produce 2^n different bit patterns. When representing signed numbers, we split this space into two equal subsets: one for positive and one for negative numbers. The 2's complement method is widely used for representing signed numbers.

For a binary system with n digits, the 2's complement of a number x is defined as:

$$\text{2's complement of } x = \begin{cases} 2^n - x & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

For example, in a 5-bit binary system, the 2's complement representation of -2 is:

$$2^5 - (00010)_b = (11110)_b$$

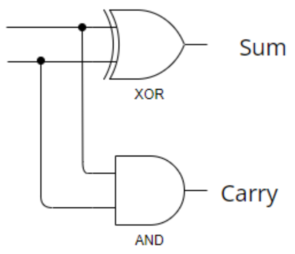
3.4 Half-Adder and Full-Adder

3.4.1 Half-Adder

A half-adder adds two bits and produces a sum and a carry bit.

Chip Name: HalfAdder
Inputs: a, b
Outputs: sum, carry
Function:

$$\text{sum} = \text{LSB of } a + b$$
$$\text{carry} = \text{MSB of } a + b$$

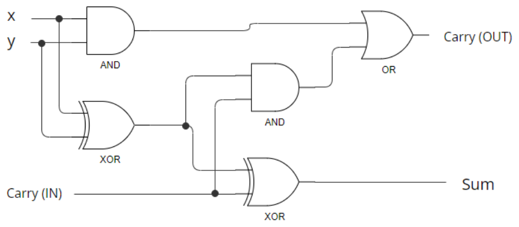


3.4.2 Full-Adder

A full-adder adds three bits (including carry) and produces a sum and a carry bit.

Chip Name: FullAdder
Inputs: a, b, c
Outputs: sum, carry
Function:

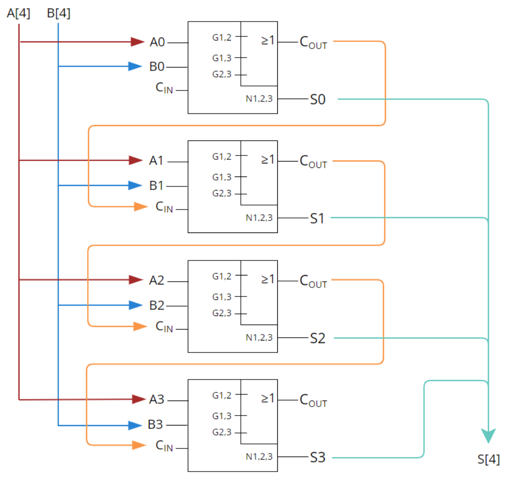
$$\text{sum} = \text{LSB of } a + b + c$$
$$\text{carry} = \text{MSB of } a + b + c$$



3.5 Multi-Bit Adders

A multi-bit adder adds two n-bit numbers. The basic building blocks are half-adders and full-adders.

Chip Name: Add16
Inputs: a[16], b[16]
Outputs: out[16]
Function: out = a + b

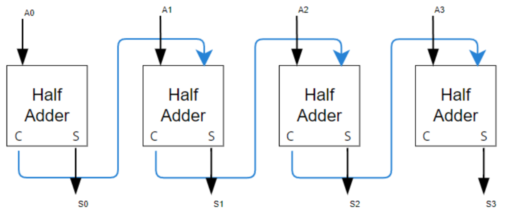


This is a 4 bit adder. For a 16 bit adder you would continue this pattern.

3.6 Incremeter

An incrementer adds 1 to a given number.

Chip Name: Inc16
Inputs: in[16]
Outputs: out[16]
Function: out = in + 1



3.7 Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logical operations. It has multiple control bits that determine the operation.

Chip Name: ALU
Inputs: x[16], y[16], zx, nx, zy, ny, f, no
Outputs: out[16], zr, ng
Function:

$$\text{if } zx = 1 \text{ then } x = 0$$
$$\text{if } nx = 1 \text{ then } x = \neg x$$
$$\text{if } zy = 1 \text{ then } y = 0$$
$$\text{if } ny = 1 \text{ then } y = \neg y$$
$$\text{if } f = 1 \text{ then out} = x + y$$
$$\text{else out} = x \& y$$
$$\text{if } no = 1 \text{ then } out = \neg\text{out}$$
$$\text{if } out = 0 \text{ then } zr = 1$$
$$\text{else } zr = 0$$
$$\text{if } out < 0 \text{ then } ng = 1$$
$$\text{else } ng = 0$$

Control Bits and Functions:

zx	nx	zy	ny	f	no	Function
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	$\neg x$
1	1	0	0	0	1	$\neg y$
0	0	1	1	1	1	$-x$
1	1	0	0	1	1	$-y$
0	1	1	1	1	1	$x+1$
1	1	0	1	1	1	$y+1$
0	0	1	1	1	0	$x-1$
1	1	0	0	1	0	$y-1$
0	0	0	0	1	0	$x+y$
0	1	0	0	1	1	$x-y$
0	0	0	1	1	1	$y-x$
0	0	0	0	0	0	$x\&y$
0	1	0	1	0	1	$x\|y$

3.7.1 Examples

Zero Output

If the bits are $zx = 1, nx = 0, zy = 1, ny = 0, f = 1$ and $no = 0$, then the ALU will output 0.

Increment Y

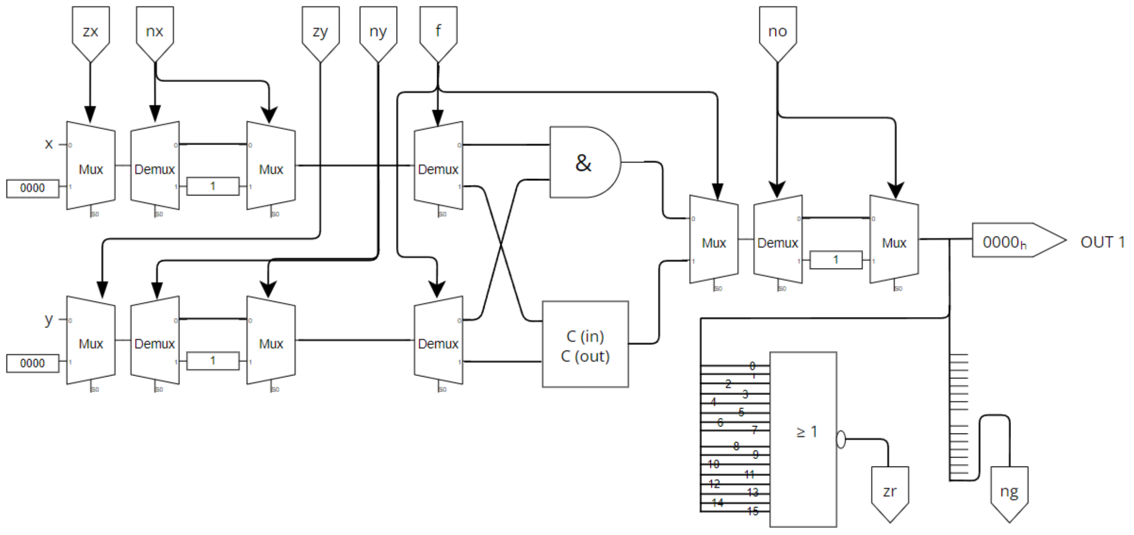
If the bits are $zx = 0, nx = 0, zy = 0, ny = 0, f = 1$ and $no = 1$, then the ALU will output $y + 1$.

Decrement X

If the bits are $zx = 0, nx = 0, zy = 1, ny = 1, f = 1$ and $no = 0$, then the ALU will output $x - 1$.

X plus Y

If the bits are $zx = 0, nx = 0, zy = 0, ny = 0, f = 1$ and $no = 0$, then the ALU will output $x + y$.



Logic Diagram of a Hack ALU.

4 Chapter 3 - Memory

4.1 Overview

Sequential logic differs from combinational logic in that it can retain state information, allowing for the storage and retrieval of data over time. This is essential for building memory elements in computers, which rely on feedback loops, clock signals, and flip-flops.

4.2 Clock Signal

The passage of time in digital circuits is represented by a clock signal, which oscillates between high and low states. Each complete cycle from low to high and back again defines a discrete time unit known as a clock cycle. This signal synchronizes all sequential elements in a system.

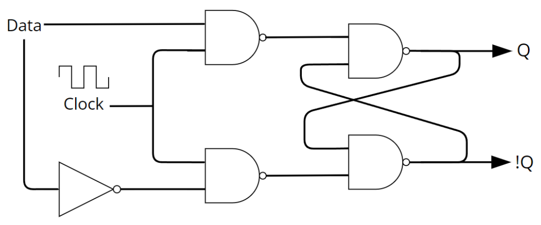
4.3 Data Flip-Flop (DFF)

The Data Flip-Flop (DFF) is the fundamental building block of sequential logic. It has a single-bit input and output, with its output at any time t being the input from the previous time $t-1$. This introduces a one-cycle delay, enabling the storage of state information.

Chip Name: DFF
Inputs: in
Outputs: out
Function:

$$\text{out}(t) = \text{in}(t-1)$$

This basic behavior is the cornerstone for all memory elements in a computer, from single-bit storage to complex memory banks.



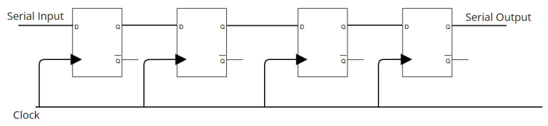
4.4 Register

A register is a multi-bit storage element that can hold a binary value over time. It consists of an array of DFFs and includes a load signal that determines whether the register should update its value or retain the current one.

Chip Name: Register
Inputs: in[16], load
Outputs: out[16]
Function:

$$\text{if load}(t-1) = 1 \text{ then out}(t) = \text{in}(t-1)$$
$$\text{else out}(t) = \text{out}(t-1)$$

Registers can be expanded to handle multi-bit values by combining multiple single-bit registers. This forms the basis for larger storage units in a computer system.



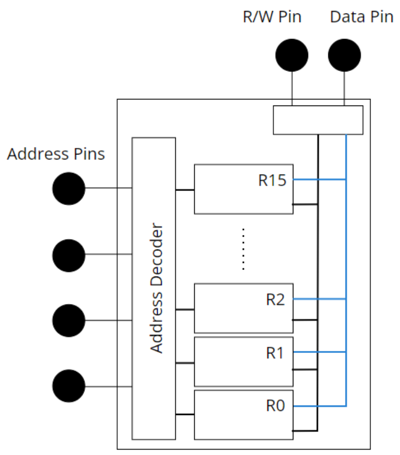
4.5 Random Access Memory (RAM)

RAM is constructed from multiple registers and allows for random access to any memory location. Each location is uniquely identified by an address, enabling direct access regardless of the physical location within the memory.

Chip Name: RAMn
Inputs: in[16], address[k], load
Outputs: out[16]
Function:

$$\text{if load}(t-1) = 1 \text{ then RAM}[address](t)$$
$$= \text{in}(t-1)$$
$$\text{out}(t) = \text{RAM}[address](t)$$

RAM modules are hierarchical, with smaller units combining to form larger memory banks, allowing for scalable and efficient storage solutions.



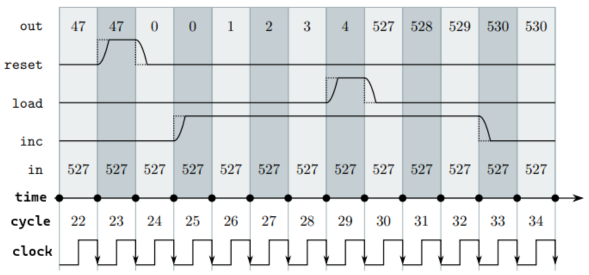
4.6 Counter

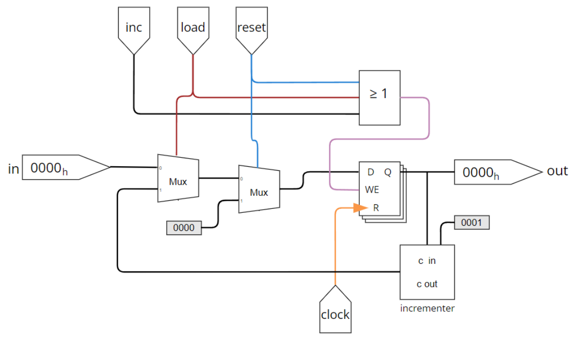
A counter is a sequential element that increments its stored value with each clock cycle. It is used in various applications, such as keeping track of instruction addresses in a program or generating timing sequences.

Chip Name: PC (Program Counter)
Inputs: in[16], inc, load, reset
Outputs: out[16]
Function:

$$\begin{gathered}\text{if reset}(t-1)=1 \text{ then out}(t)=0\\ \text{else if load}(t-1)=1 \text{ then out}(t)=\text{in}(t-1)\\ \text{else if inc}(t-1)=1 \text{ then out}(t)=\text{out}(t-1)+1\\ \text{else out}(t)=\text{out}(t-1)\end{gathered}$$

Counters are essential for controlling the sequence of operations in a computer, particularly in program execution.





Combinational vs. Sequential chips

Combinational chips are digital circuits where the output solely depends on the current inputs, without any memory or feedback elements. They are stateless, meaning their outputs are determined by the combination of present input values. Common examples of combinational chips include:

Sequential chips, on the other hand, incorporate memory elements that allow them to store and recall past inputs or states. Their outputs depend not only on the current inputs but also on the history of inputs. This makes sequential chips stateful devices.

4.7 Key Concepts

1. **Clock Signal**: Synchronizes state changes in sequential elements.

2. **DFF**: Fundamental storage element introducing a one-cycle delay.

3. **Register**: Multi-bit storage element controlled by a load signal.

4. **RAM**: Hierarchical memory structure allowing random access.

5. **Counter**: Sequential element for tracking and controlling sequences.

5 Chapter 4 - Machine Language

5.1 Overview

Machine language is a low-level programming language understood directly by the computer's hardware. It bridges the gap between hardware and software, allowing the programmer to instruct the processor to perform arithmetic and logical operations, fetch and store values in memory, and control the program flow.

5.2 Hardware Structure

The Hack platform's hardware consists of several key components:

- **Registers**: The Hack CPU has two main registers - the A-register and the D-register. The A-register can hold both data and addresses, while the D-register is used exclusively for data.
- **RAM (Random Access Memory)**: Used for data storage.
- **ROM (Read-Only Memory)**: Stores the program instructions.

5.3 Hack Machine Language Specification

The Hack machine language has two types of instructions: A-instructions and C-instructions.

A-Instruction: The A-instruction is used to set the A-register to a specific value. It is represented by the symbol `@` followed by a value or a symbol.

```
@value
```

- `@value`: Sets the A-register to the specified value.

C-Instruction: The C-instruction performs computations and controls the flow of the program. It consists of three fields: `dest`, `comp`, and `jump`.

```
dest = comp; jump
```

- `dest`: Specifies where to store the result of the computation.
- `comp`: Specifies the computation to be performed.
- `jump`: Specifies the conditional jump.

5.4 Control Instructions

The Hack machine language provides instructions to control the program flow, such as conditional and unconditional jumps.

- `JMP`: Unconditional jump.
- `JEQ`: Jump if equal to zero.
- `JGT`: Jump if greater than zero.
- `JLT`: Jump if less than zero.

5.5 Example Program

Here is an example program that adds the numbers from 1 to 100 and stores the result in the D-register.

```
@i
M=1 // Initialize i to 1
@sum
M=0 // Initialize sum to 0
(LOOP)
@i
D=M // D=i
@100
D=D-A // D=i-100
@END
D;JGT // If D>0, goto END
@i
D=M // D=i
@sum
M=D+M // sum=sum+i
@i
M=M+1 // i=i+1
@LOOP
0;JMP // Goto LOOP
(END)
@END
0;JMP // Infinite loop to end program
```

The Hack assembly language specification is detailed further in Chapter 6 (Assembler).

5.6 Screen

The screen is a memory-mapped device in the Hack platform. This means that a portion of the RAM is designated to represent the pixels of the screen. The screen memory map starts at RAM address 16384 (0x4000) and ends at 24575 (0x5FFF), covering a total of 8192 addresses. Each address corresponds to a 16-bit word, with each bit representing one pixel on the screen. A bit value of 1 indicates a white pixel, while a bit value of 0 indicates a black pixel.

Screen Memory Layout The screen is organized as a grid of 256 rows by 512 columns. Each 16-bit RAM word controls 16 horizontal pixels. For example, to turn on the first pixel of the screen, you can write the following Hack assembly code:

Listing 1: Turning on the first pixel of the screen

```
@16384
M=1
```

To turn on the second pixel, you would write:

Listing 2: Turning on the second pixel of the screen

```
@16384
M=2
```

Example: Drawing a Horizontal Line The following example code draws a horizontal line on the first row of the screen:

Listing 3: Drawing a horizontal line

```
@16384
M=-1 // Sets all 16 pixels of the first word
     to white
@16385
M=-1 // Sets the next 16 pixels to white
// Repeat until the end of the first row
```

The screen resolution is 256 rows by 512 columns, and each word in memory controls 16 horizontal pixels. Therefore, each row consists of $\frac{512}{16} = 32$ words. So this would need to be completed 32 times to draw a horizontal line across the whole screen.

5.7 Keyboard

The keyboard is another memory-mapped device in the Hack platform, mapped to RAM address 24576 (0x6000). The value at this address represents the ASCII code of the key currently pressed. If no key is pressed, the value is 0.

Example: Handling Keyboard Input

The following example code waits for a key press and stores the ASCII code of the key pressed in the D-register:

```
(LOOP)
@24576
D=M
@LOOP
D;JEQ // If no key is pressed, keep looping
// D now contains the ASCII code of the key
     pressed
```

- `(LOOP)`: This is a label that marks the beginning of a loop.
- `@24576`: This is an A-Instruction that sets the A-register to the address 24576. This RAM address is mapped to the keyboard input. The value at this address is the ASCII code of the key currently pressed. If no key is pressed, the value is 0.
- `D=M`: This is a C-Instruction. It sets the D-register to the value stored in the memory location currently addressed by the A-register (which here is 24576). Essentially, this instruction reads the keyobard input into the D-register.
- `@LOOP`: This is another A-instruction that sets the A-register to the address of the `(LOOP)`-label. This allows the program to jump hack to the beginning of the loop.
- `D;JEQ`: This C-instruction with a jump condition performs a jump to the address in the A-register (here the `LOOP`-label) if the value in the D-register is equal to zero. If no key is pressed, the value in the D-register will be 0, and the programm will jump back to `(LOOP)`, effectively creating a loop until a key is pressed.

6 Chapter 5 - Computer Architecture

6.1 Overview of the Von Neumann Architecture

The Von Neumann architecture, also known as the stored-program computer, is the foundational model for nearly all modern computers. It consists of a central processing unit (CPU), memory, input/output devices, and a single communication channel for instructions and data.

6.2 Key Components of the Hack Platform

The Hack computer is a simple yet powerful example of a Von Neumann machine. It includes the following key components:

- **CPU (Central Processing Unit)**: Executes instructions and processes data.
- **Memory (RAM and ROM)**: Stores data and program instructions.
- **Input/Output Devices**: Includes a keyboard for input and a screen for output.

6.3 Stored Program Concept

The stored program concept is central to the Von Neumann architecture. It means that instructions and data are stored together in memory. This allows the computer to be reprogrammed by simply changing the instructions in memory without altering the hardware.

6.4 Fetch-Execute Cycle

The CPU operates on a continuous loop known as the fetch-execute cycle:

1. **Fetch**: The CPU retrieves the next instruction from memory.
2. **Decode**: The instruction is decoded to determine the required action.
3. **Execute**: The CPU performs the operation specified by the instruction.
4. **Store**: The result of the operation is stored back in memory or a register.

This cycle is repeated continuously, enabling the execution of complex programs.

6.5 Hack Platform Execution and Fetch Behavior

The Hack platform implements the fetch-execute cycle as follows:

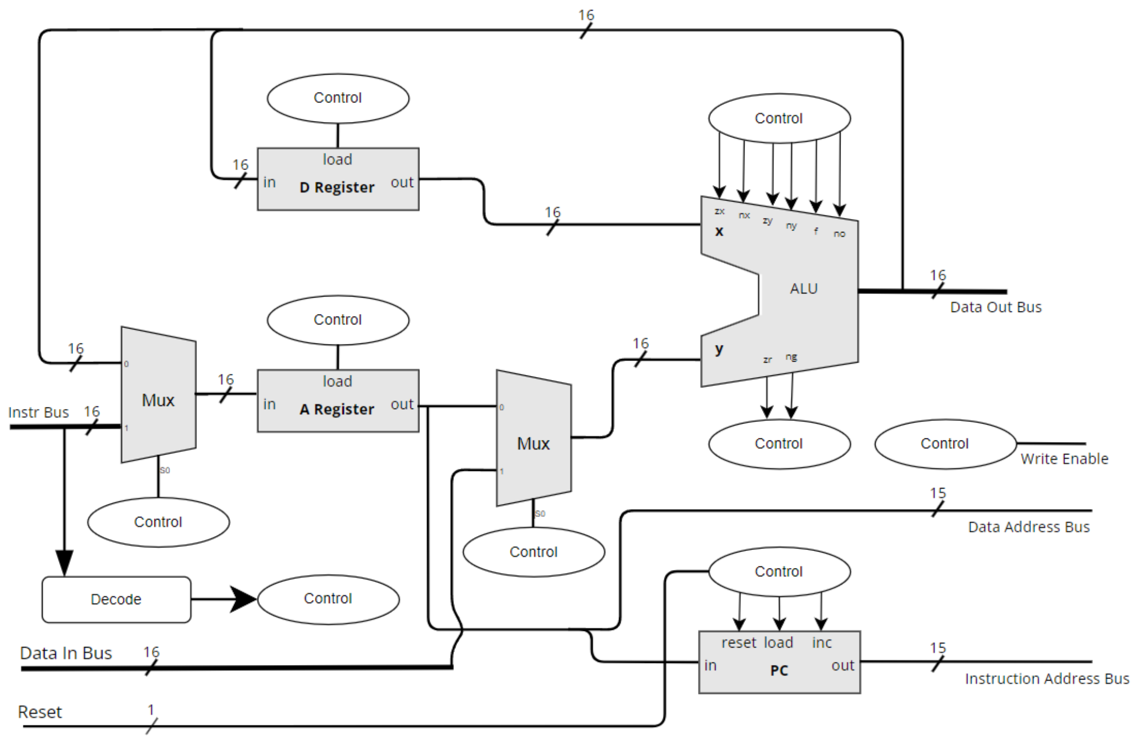
- **Execute**: Depending on the instruction type (A or C), the CPU updates its registers and memory.
- **Fetch**: The next instruction is fetched from the ROM based on the program counter (PC). If a jump instruction is executed, the PC is updated accordingly; otherwise, it increments sequentially.

6.6 Hack CPU Architecture

The Hack CPU consists of the ALU, registers, and control logic:

- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logical operations.
- **Registers**: Include the A-register (address/data) and D-register (data).
- **Control Unit**: Decodes instructions and generates control signals for execution.

The CPU interacts with memory and I/O devices through memory-mapped I/O, where specific memory addresses correspond to hardware components.



6.6.1 Instruction Decoding and Execution

Understanding how instructions are decoded helps in grasping how the CPU operates at a fundamental level.

Instruction Types There are two main types of instructions in Hack machine language:

- **A-Instruction**: Used to set the A-register to a specific value or address. Example: `@value`.
- **C-Instruction**: Specifies a computation and where to store the result. It has the format `dest=comp;jump`.

Decoding Process

- **A-Instruction Decoding**: The control unit interprets the value following the `@` symbol and sets the A-register accordingly.
- **C-Instruction Decoding**: The control unit breaks down the instruction into its components: destination (dest), computation (comp), and jump. It then generates the appropriate control signals to execute the operation.

6.6.2 Example Instruction Analysis

For a C-instruction such as `D=D+M;JGT`:

- **Decode**: Identify the destination (D), computation (D+M), and jump condition (JGT).
- **ALU Configuration**: Set the ALU to add the value in the D-register to the value in the memory location pointed to by the A-register.
- **Result Storage**: Store the result back in the D-register.
- **Conditional Jump**: If the result is greater than zero, update the PC to the new address; otherwise, continue with the next instruction.

6.6.3 Additional Explanation on Data Flow

Data Transfer between Components When an instruction requires data from RAM:

1. **Data Fetch**: The address in the A-register points to the specific memory location. The data at this address is fetched and placed on the Data In Bus.

2. **Register Load**: The fetched data is loaded into the D-register or directly used in the ALU for computations.

ALU Operations and Results The ALU performs the required arithmetic or logical operation based on the inputs from the registers:

- **Inputs**: Values from the A-register and D-register.

- **Operation**: Defined by the control signals, the ALU processes the inputs.

- **Output**: The result is placed on the Data Out Bus and can be stored back in RAM or a register.

Program Counter (PC) Update After each instruction execution:

- **Increment**: The PC increments to the next instruction address.

- **Jump Instructions**: For conditional or unconditional jumps, the PC is updated with a new address from the A-register.

This detailed explanation ensures a comprehensive understanding of how the Hack computer's components work together to execute instructions, providing a solid foundation for answering questions related to the Von Neumann architecture and the Hack platform's execution model.

6.6.4 Example Program Execution

To illustrate how the Hack computer operates, let's consider a simple program that loads a value from memory, increments it, and stores it back. The program in Hack assembly language is:

Listing 4: Increment a value in memory

```
@100 // Load address 100 into A-register
D=M // Load value from RAM[100] into D-
    register
D=D+1 // Increment the value in D-register
@100 // Load address 100 into A-register
    again
M=D // Store the incremented value back into
    RAM[100]
```

Step-by-Step Hardware Interaction

1. **Instruction Fetch**: The program counter (PC) fetches the instruction '@100' from ROM. The address 100 is loaded into the A-register. - *Diagram Reference*: The PC sends the address to the Instruction Address Bus, and the fetched instruction is placed on the Instr Bus.

2. **Load Value from RAM**: The next instruction 'D=M' is fetched, decoded, and executed. The data at RAM[100] is loaded into the D-register. - *Diagram Reference*: The address in the A-register points to RAM[100]. The control unit sends a signal to read from RAM, and the value is placed on the Data In Bus and loaded into the D-register.

3. **Increment Value**: The instruction 'D=D+1' is fetched and decoded. The ALU is configured to add 1 to the value in the D-register. - *Diagram Reference*: The control signals configure the ALU to perform the addition operation using the value in the D-register. The result is placed on the Data Out Bus.

4. **Store Incremented Value**: The instruction '@100' is fetched again, setting the A-register to 100. The instruction 'M=D' stores the incremented value back into RAM[100]. - *Diagram Reference*: The address 100 is set in the A-register, and the control unit sends a signal to write to RAM. The value on the Data Out Bus is written to RAM[100].

7 Chapter 6 - Assembler

The assembler converts assembly into binary. In this chapter, we will delve into the details of how an assembler works and what considerations are necessary when implementing our own. By the end of the chapter, you should be able to implement your own assembler that converts Hack assembly into binary code, executable on Hack hardware platforms.

7.1 Machine language

Machine language can typically be specified in two ways: binary and symbolic. A binary instruction might look like this:

```
11000010000000110000000000000111
```

This instruction consists of microcodes that can be decoded by specific hardware platforms. The first 8 bits, `11000010`, could represent the operation `load`. The next 8 bits, `00000011`, could denote register `R3`. The remaining 16 bits could specify the value `7`. When developing a machine language and hardware architecture, we can define that such a 32-bit instruction would mean: load the constant `7` into register `R3`.

It's clear that binary representation is not very human-readable, which is why we can use symbolic machine language. For example, we can define the same instruction symbolically as follows:

```
load R3, 7
```

The symbolic syntax codes are often called mnemonics. When writing low-level software, it makes sense to use symbolic notation, which can then be translated into binary code by a tool like our assembler. An assembler converts assembly language (symbolic machine language) into the corresponding binary codes.

7.2 Symbols

Symbols are used to assign names to specific addresses. For example, consider the symbolic instruction `goto 312`. This instruction tells the computer to execute the instruction located at address 312. Suppose this address marks the start of a loop. It would be much more readable to use `goto LOOP` instead of `goto 312`. The assembler can then translate `LOOP` to `312`, simply remembering that `LOOP` points to address 312.

This enhancement of the assembly language significantly improves readability and portability across different systems. Generally, we use symbols in assembly for three main reasons:

- **Names**: Symbols can point to different addresses in the code, often used as labels. For example, `LOOP` and `END`.
- **Variables**: Symbols can represent variables. For example, `i` and `sum`.
- **Predefined Symbols**: These symbols point to predefined addresses in the computer's memory. For example, `SCREEN` and `KBD` (KBD points to the memory-mapped I/O address for the keyboard).

Our assembler needs to accurately track and manage symbols when translating code into binary. Keeping track of symbols is one of the assembler's most critical tasks.

7.3 Hack Assembly Specification

7.3.1 Type A-Instructions

- Symbolic: `@value`
- Binary: `0vvvvvvvvvvvvvvv`
  (where v is 0 or 1)

Every assembly program can use predefined symbols for `value`:

- `R0`, `R1`, ..., `R15` stand for the Hack RAM addresses 0, 1, ..., 15.
- `SP`, `LCL`, `ARG`, `THIS`, `THAT` stand for the Hack RAM addresses 0, 1, 2, 3, 4.
- `SCREEN` and `KBD` stand for the Hack RAM addresses 16384 and 24576.

Any symbol `xxx` used for `value` in an assembly program that is not predefined and not defined elsewhere by a label declaration (`xxx`) is treated as a variable. The first variable encountered in a program is mapped to `RAM[16]`, the second to `RAM[17]`, and so on. They are always mapped to the next free memory address.

Translating Type A-Instructions to binary involves converting the decimal address to its binary equivalent. For example, the instruction `@2` (a constant) will be converted to `0000000000000010`. Similarly, the instruction `@R0` will be converted to `0000000000000000`. Predefined symbols, such as `THIS`, will be converted to `0000000000000011` (decimal 3).

7.3.2 Type C-Instruction

- Symbolic: `dest = comp; jump`
- Binary: `111 a cccccc ddd jjj` (where a, c, d, j are 0 or 1)

Table 1: Computation mnemonics

c1	c2	c3	c4	c5	c6	Mnemonic	
						for $a = 0$	for $a = 1$
1	0	1	0	1	0	0	
1	1	1	1	1	1	1	
1	1	1	0	1	0	-1	
0	0	1	1	0	0	D	
1	1	0	0	0	0	A	M
0	0	1	1	0	1	!D	
1	1	0	0	0	1	!A	!M
0	0	1	1	1	1	-D	
1	1	0	0	1	1	-A	-M
0	1	1	1	1	1	D+1	
1	1	0	1	1	1	A+1	M+1
0	0	1	1	1	0	D-1	
1	1	0	0	1	0	A-1	M-1
0	0	0	0	1	0	D+A	D+M
0	1	0	0	1	1	D-A	D-M
0	0	0	1	1	1	A-D	M-D
0	0	0	0	0	0	D&A	D&M
0	1	0	1	0	1	D | A	D | M

Table 2: Destination Field

d1	d2	d3	Mnemonic	Destination
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (Memory Register)
0	1	0	D	D-Register
0	1	1	MD	Memory[A] and D-Register
1	0	0	A	A-Register
1	0	1	AM	A-Register and Memory[A]
1	1	0	AD	A-Register and D-Register
1	1	1	AMD	A-Register, Memory[A] and D-Register

Table 3: Jump Field

j1	j2	j3	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	Jump if out `>` 0
0	1	0	JEQ	Jump if out = 0
0	1	1	JGE	Jump if out `>=` 0
1	0	0	JLT	Jump if out `<` 0
1	0	1	JNE	Jump if out != 0
1	1	0	JLE	Jump if out `<=` 0
1	1	1	JMP	Jump

When translating C Instructions the single parts (dest, comp and jump) get converted and assembled to a 32-bit value. With an instruction like `D=D+A` we only get a destination (`D`) and a mnemonic for computation (`D+A`), meaning jmp is `null`. This then would translate to `000010` for comp, `010` for dest and `000` for jmp. All together we get `1110000010010000`.

7.4 Labels

Labels are similar to symbols. Instead of pointing to memory addresses, they point to ROM addresses in the Hack machine. In other words, a label refers to a specific instruction in the subsequent assembly program. Labels in the assembly code are enclosed in parentheses, like this: `(LABEL)`. They always point to the instruction immediately following their appearance in the program.

```
 1  @2
 2  D=A
 3  @0
 4  M=D
 5  (LOOP) // label definition
 6  @LOOP
 7  D=D-1
 8  @END
 9  D;JEQ
10  @0
11  M=M+1
12  @LOOP
13  0;JMP
14  (END)
15  @END
16  0;JMP
```

In this example, the label `(LOOP)` on line 5 points to the instruction `D=D-1` on line 6 (the line in which the label is defined is not being counted). This creates a loop that decrements `D` and increments the memory at address 0 until `D` equals 0. Once `D` is zero, the program jumps to the `END` label and stops.
We see that the labels are being used like pointers in our program.

7.5 Translating assembly to binary

The assembler processes a given set of assembly instructions and converts them into an equivalent set of binary instructions. These binary instructions can be directly loaded and executed by the computer. To perform this translation, the assembler must be capable of converting both instructions and symbols.

7.5.1 Handling instructions

To handle instructions correctly, the assembler needs to disassemble them into their smaller components: `comp`, `dest` and `jmp`. For each of these symbolic values, the assembler must look up the corresponding binary equivalent and assemble them into a single binary instruction. This process ensures that the high-level assembly instructions are accurately translated into the low-level machine code that the computer can execute.

7.5.2 Handling symbols

Assembly programs are allowed to use symbols that have not yet been defined. This rule makes developing with assembly easier but complicates the development of the assembler. A common solution to this problem is to assemble the code in two iterations.
In the first iteration, the assembler scans through all labels and maps them to their corresponding Hack ROM addresses.
In the second iteration, the assembler uses this map to look up the values of any newly encountered symbols. A symbol's value can be found either in the symbol table (where all symbols are mapped to their corresponding addresses) or in the map of labels from the first iteration.

7.6 Assembler Implementation

Let's go through the single steps of making an own assembler.

1. Initialization
In this step, we need to load the assembly code into an appropriate data structure for further processing. We must also handle cases where users provide the wrong number of arguments or when the specified file does not exist. Essentially, we are preparing everything for the first iteration of label processing.

2. Preprocessing
In assembly language, comments are used to help developers understand specific parts of the code or to clarify the reasoning behind certain implementations. Comments and empty lines should not be processed by the assembler. To handle this, the assembly code can be preprocessed to remove these elements, ensuring that only relevant instructions are considered in the subsequent stages.

3. First iteration: Label Resolution
During the first iteration, the assembler scans through the entire program to identify and record labels. Each label encountered is mapped to its

corresponding address in the Hack ROM. This mapping is stored in the symbol table, which will be used in the second iteration to resolve symbols. Remember that after extracting the labels, they must also be removed from the code before further processing. All label definitions (e.g., `(LOOP)`) need to be eliminated to avoid any interference during the translation phase.

4. Second Iteration: Instruction Translation
In the second iteration, the assembler rescans the program to translate each instruction into its binary form. When encountering symbols and variables, the assembler looks them up using the map created in the first iteration. Symbols' values can either be found in the predefined symbol table or in the map of labels from the first iteration. If a symbol is not predefined, it is treated as a variable and assigned a memory address if it does not already have one.

5. Instruction Translation Details
To translate instructions correctly, the assembler needs to disassemble them into their smaller components: comp, dest, and jmp. For each of these symbolic values, the assembler must look up the corresponding binary equivalent and assemble them into a single binary instruction. This ensures that high-level assembly instructions are accurately translated into low-level machine code that the computer can execute.

- A-Instructions: An A-instruction is represented symbolically as `@value` and translated into a binary format as `0vvvvvvvvvvvvvvv`, where v is the binary representation of value. For example, the instruction `@2` is translated to `0000000000000010`.

- C-Instructions: A C-instruction is represented symbolically as `dest = comp; jump` and translated into a binary format as `111 a cccccc ddd jjj`, where a, c, d, and j are binary codes for the comp, dest, and jmp fields. For example, the instruction `D=D+A` translates to `1110000010010000`.

6. Output the Binary Code
In the final step, the assembler generates the output file containing the binary instructions. This file is ready to be loaded and executed by the computer. It is essential to ensure that the output is correctly formatted and that all instructions are properly translated into their binary equivalents.

8 Chapter 7 - VM I : Stack Arithmetic

In this section, we will cover the essential concepts and practical implementations related to virtual machines (VMs), focusing on stack arithmetic operations.

8.1 Stack and Stack Arithmetic

What is a Stack?. A Stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It supports two primary operations:

- **Push:** Adds an element to the top of the stack
- **Pull:** Removes the element from the top of the stack

Stack Operations
In a Stack machine, every operation involves pushing and popping values to or from the stack. Here are the basic operations:

- Push, `push segment index`: This command pushes the value from the specified segment at the given index onto the stack.
- Pop, `pop segment index`: This command pops the top value from the stack and stores it in the specified segment at the given index.

Arithmetic Operations
Arithmetic operations in a stack machien are straightforward. They take operands from the stack, perform the operation, and push the result back onto the stack.

- Addition, `add`: Pops the top two values from the stack, adds them, and pushes the result to the stack.
- Substraction, `sub`: Pops the top two values, subtracts the second from the first, and pushes the result.
- Negation, `neg`: Pops the top value, negates it, and pushes the result.

Arithmetic and Logical Operations

The following table summarizes the arithmetic and logical operations supported by the Hack virtual machine. These operations manipulate the values on the stack and produce results according to the specified operations.

CMD	**Calc**	**Comment**
add	$x + y$	Integer addition in two's complement
sub	$x - y$	Integer subtraction in two's complement
neg	$-y$	Arithmetic negation in two's complement
eq	$x == y$	Equality relation
gt	$x > y$	Greater than relation
lt	$x < y$	Less than relation
and	x and y	Bitwise AND
or	x or y	Bitwise OR
not	not y	Bitwise NOT

8.2 RAM Usage

The Hack RAM consists of 32K 16-bit words. VM implementations should use the upper part of this address space as follows:

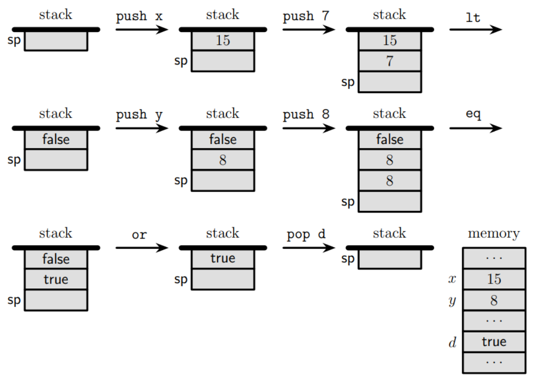
RAM Address	**Usage**
0 to 15	16 virtual registers, usage described below
16 to 255	Static variables
256 to 2047	Stack

Remember that according to the Hack machine language specification, RAM addresses 0 to 4 can be referred to by the symbols SP, LCL, ARG, THIS, and THAT. This convention was introduced to help developers of VM implementations write readable code. The expected use of these addresses in VM implementation is as follows:

Name	Memory Location	Usage
SP	RAM[0]	The address of the top of the stack
LCL	RAM[1]	Base address of the local segment
ARG	RAM[2]	Base address of the argument segment
THIS	RAM[3]	Base address of the this segment
THAT	RAM[4]	Base address of the that segment
TEMP	RAM[5-12]	Used as temp segment
R13-R15	RAM[13-15]	Free registers for assembly code generation

8.3 Understand Stack Operations

The diagram below shows a sequence of stack operations in a vritual machine. Initially the stack is empty.



First, the value `15` (referred to as `x`) is pushed onto the stack, followed by the value `7`. On each of the operations the stack pointer gets increased as well. The `lt` (less than) operation compares the top two values (`7` and `15`). Since `15` is not less than `7`, the result `false` is pushed onto the stack.

Next, the value `8` (referred to as `y`) is pushed onto the stack, followed by another `8`. Since the `eq` (equal) operation compares the top two values (`8` and `8`), and since they are equal, `true` is pushed onto the stack.

Finally, the `or` operation compares the top two values (`true` and `false`) using a bitwise OR, resulting in `true` being pushed onto the stack. The `pop d` operation then removes the top value (`true`) from the stack and stores it in the memory location `d`.

8.4 VM Implementation on the Hack Platform

8.4.1 VM Abstraction

The Hack VM mdoel abstracts a stack-based architecture that supports arithmetic operations, memory access (push/pop), branching, and function calls. Here's how you can translate VM commands into Hack assembly language.

8.4.2 Memory Segments

The Hack VM uses various memory segments, each serving a specific purpose:

- **Argument:** Holds function arguments.
- **Local:** Holds local variables.
- **Static:** Holds static variables.
- **Constant:** Represents constant values.
- **This/That:** Pointers to object fields.
- **Pointer:** Holds the base addresses of `this` and `that`.
- **Temp:** Temporary storage.

8.4.3 Translating VM code

Push Operation

```
push constant 5
```

This would be translated to:

```
// Push the constant value 5 onto the stack
@5 // Load constant value 5
D=A // Store the value 5 in register D
@SP // Access the stack pointer
A=M // Get the address of the top of the
     stack
M=D // Store the value 5 at the top of the
     stack
@SP // Access the stack pointer again
M=M+1 // Increment the stack pointer
```

Pop Operation

```
pop local 0
```

This would be translated to:

```
// Pop the top value from the stack and store
      it in local segment 0
@SP // Access the stack pointer
AM=M-1 // Decrement the stack pointer and get
      the top of the stack address
D=M // Store the top value in register D
@LCL // Access the base address of the local
     segment
A=M // Get the base address of the local
     segment
M=D // Store the popped value into local
     segment 0
```

Add Operation

```
add
```

This would be translated to:

```
@SP
AM=M-1
D=M
A=A-1
M=M+D
```

Here's a breakdown of the assembly:

- `@SP`: Set `A` to the address of the stack pointer.
- `AM=M-1`: Decrement the value in `@SP` and th en set both `A` and `M` to the new top of the stack address. This makes `A` point to the new top of the stack.
- `D=M`: Store the value at the new top of the stack (previously the second item from the top) in `D`.
- `A=A-1`: Decrement `A` to the value at the new top of the stack.
- `M=M+D`: Add the value in `D` to the value at the new top of the stack (previously the first item from the top) and store the result back at this location.

Subtract Operation

```
sub
```

This would be translated to:

```
@SP
AM=M-1
D=M
A=A-1
M=M-D
```

Negation

```
neg
```

This would be translated to:

```
@SP
A=M-1
M=-M
```

More examples of translation from VM instructions to Hack assembly can be found in the appendix.

9 Chapter 8 - VM II : Program Control

In VM part 2, we build upon the foundation from VM part 1 and extend the VM translator to handle more operations, including branching and function calls. The primary goal is to expand the basic VM translator into a full-fledged translator that can handle multi-file programs written in the VM language.

Branching Commands

Branching commands allow the VM to implement control flow. These commands include `label`, `goto`, and `if-goto`.

1. `label labelName`: Marks a position in the code within a function. This label can beused as jump destination.
2. `goto labelName`: Unconditionally jumps to the specified label within the same function.
3. `if-goto labelName`: Pops the top value from the stack; if the value is not zero, the execution jumps to the specified label. Otherwise, it continues to the next instruction.

Function Commands

Function commands enable function definition, calling, and returning. These include `function`, `call`, and `return`.

1. `function functionName nVars`: Declares a function and specifies the number of local variables it will use.
2. `call functionName nArgs`: Calls a function, passing a specified number of arguments.
3. `return`: Returns from a function, restoring the state of the caller function.

9.1 Implementation of Branching Commands

label

A `label` within the vm is defined as:

```
label LOOP_START
```

Translated to hack it looks like this:

```
(LOOP_START)
```

goto

A `goto` instruction within the vm is defined as:

```
goto LOOP_START
```

Translated to hack it looks like this:

```
@LOOP_START // Addresses the label
0;JMP // Unconditional jump
```

if-goto

An `if-goto` instruction within the vm is defined as:

```
if-goto LOOP_START
```

Translated to hack it looks like this:

```
@SP // Address stack pointer
AM=M-1 // Decrement SP and address top
D=M // Store top of stack in D
@LOOP_START // Address the label
D;JNE // Jump to LOOP_START if D != 0
```

We see here, that `if-goto` only jumps to `LOOP_START` whenever the top of the stack is not equal to zero. So if we were to do something like `if` $(7 < 8)$ `then goto LOOP_START`, we would write it as:

```
push constant 7
push constant 8
lt
if-goto LOOP_START
```

This is because this would result in `true` being on top of the stack, which is `-1` (and therefore something other than 0), resulting in a jump.

9.2 Implementation of Function Commands

function

Declaring functions within the vm environment will look like this:

```
function SimpleFunction 2
```

The 2 at the end stands for the number of local variables the function will use. Translated to hack this will look like:

```
(SimpleFunction) // Declares the function
@SP
D=M
@LCL
M=D // Initializes LCL
@SP
A=M
M=0 // Initialize first local variable to 0
@SP
M=M+1
@SP
A=M
M=0 // Initialize second local variable to 0
@SP
M=M+1
```

call

Calling functions within the vm environment will look like this:

```
call SimpleFunction 2
```

Translation can be found in the appendix. The translation is too long to be included here. The reason for this being that you need to adjust all local variables used by the function such as `THIS` and `THAT`.

return

Returning function results within the vm environment will look like this:

```
return
```

Translation can be found in the appendix. The translation is too long to be included here. This is pretty long because we need to reset all the local variables when returning from a function.

The table for arithmetic operations can be found in the appendix.

9.3 Responsibilities of the Parser and CodeWriter

Responsibilities of the Parser

The Parser is responsible for reading and analyzing the VM commands. Its main tasks include:

1. Reading the VM File: The Parser reads the input file containing the VM commands.
2. Tokenizing Commands: The Parser breaks down each VM command into its fundamental components (tokens), such as the command type, arguments and memory segments.
3. Identifying Command Types: The Parser identifies the type of each VM command (e.g. arithmetic, push, pop, label, goto, if-goto, function, call, return).
4. Providing Access to Commands: The Parser provides methods to access the current command and its components, and to advance to the next command in the file.

Responsibilities of the CodeWriter

The CodeWriter is responsible for translating the parsed VM commands into Hack assembly code.

1. Initializing the Out File: The CodeWriter initialized the out file.
2. Writing Assembly Code: The CodeWriter translates each VM command provided by the Parser into the corresponding Hack assembly instructions.
3. Handling different command types: The CodeWriter contains methods for handling different types of VM commands (arithmetic, memory access commands, program flow commands, function commands).
4. Generating Unique Labels: The CodeWriter generates unique labels for internal use in the assembly code to ensure correct branching and function handling.

There is an example of this process in the appendix.

10 Chapter 9 - High-Level Language (Jack)

10.1 Introduction to Jack

Jack is a simple object-based high-level language inspired by Java and C++, but without inheritance. It is designed to help programmers write high-level programs and serves as a basis for building a compiler and operating system.

Jack programs consist of classes, each saved in a separate file. Each class can have static and field variables, constructors, methods, and functions.

A Hello World program in Jack would look like this:

```
class Main {
    function void main() {
        do Output.printString("Hello World");
        do Output.println(); // New line
        return; // return is necessary
    }
}
```

The `Main` class contains a single function `main` which prints `Hello World` to the output.

The following program reads integers from the keyboard, stores them in an array and computes their average:

```
/** Inputs integers and computes their
     average. */
class Main {
    function void main() {
        var Array a;
        var int length, i, sum;
        let i = 0;
        let sum = 0;
        let length = Keyboard.readInt("How
             many numbers?");
        let a = Array.new(length); //
             Constructs the array
        while (i < length) {
            let a[i] = Keyboard.readInt("Enter
                 a number: ");
            let sum = sum + a[i];
            let i = i + 1;
        }
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}
```

10.2 Grammar for Formal Languages

Formal grammars can be specified using production rules. Jack programs can be analyzed using context-free grammar as part of the Chomsky hierarchy:

- **Regular Grammars**: Can be represented by regular expressions.
- **Context-Free Grammars**: Suitable for describing the syntax of programming languages.
- **Context-Sensitive Grammars**: More powerful, can describe certain constraints in natural languages.
- **Recursively Enumerable Grammars**: The most powerful, can describe all computable languages.

10.3 CYK Algorithm for Context-Free Languages

The CYK (Cocke-Younger-Kasami) algorithm is used to determine if a given string belongs to a context-free language. It uses dynamic programming to parse strings efficiently.

- **Input**: A string and a context-free grammar in Chomsky Normal Form.
- **Output**: Whether the string can be generated by the grammar.

10.3.1 Example of CYK algorithm

In order to look at the CYK algorithm, we will work with an example. Lets say we want to check if a word `abacba` is part of a grammar. The grammar of the language is defined as:

- $S \to SA|a$
- $A \to BS$
- $B \to BB|BS|b|c$

Step 0: Write the word into the table

Begin by writing the input string into the first row of the table, where each cell corresponds to a single character of the string.

0.	a	b	a	c	b	a
1.						
2.						▨
3.					▨	▨
4.				▨	▨	▨
5.			▨	▨	▨	▨
6.		▨	▨	▨	▨	▨

Step 1: Fill in the Table for Length 1 Substrings

From now on in all steps we will work with a certain length of substrings. Meaning we will always look at a greater part of the table and therefore characters. For now, we only need to do a 1:1 translation. So we translate each of the characters to their non-terminal counterpart.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.						▨
3.					▨	▨
4.				▨	▨	▨
5.			▨	▨	▨	▨
6.		▨	▨	▨	▨	▨

Step 2: Fill in the Table for Length 2 Substrings

Now we start looking at the bigger picture. We start with the first column and always look at one pair of characters. Then we check if we can translate that pair to a non-terminal value. If we can, we write that non-terminal value into the table below the "leftest" of the characters checked. If we can not translate it we write an empty set ∅ into the table.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	▨
3.					▨	▨
4.				▨	▨	▨
5.			▨	▨	▨	▨
6.		▨	▨	▨	▨	▨

For the first column, we take the first two values from row 1. and we look for `''SB''` in our language definition. Since we can not find `''SB''` we put an empty set ∅ in this cell. We do this all the way to the right until our last pair: `''BS''`. We can find that in two different locations: For `''A''` and for `''B''`, so we write both into the cell.

Step 3: Fill in the Table for Length 3 Substrings

Now we eventually do the same thing, only for length 3 substrings.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	▨
3.	S	B	∅	A, B	▨	▨
4.				▨	▨	▨
5.			▨	▨	▨	▨
6.		▨	▨	▨	▨	▨

When looking for the correct values in row 3, we need to compare values across each other. So for the first row we first compare `''S''` with `''A, B''` (red). When comparing with multiple values in a cell we always build substrings with all the options we have. Here its `''SA, SB''`. We can translate ``SA'' to ``S'', but we can not find anything for ``SB''. So we just write ``S'' into the cell.

But not only that, we as well need to compare a little more. We need to compare the empty set from row 2, with the 2nd ``S'' from row 1 (blue). The result of both comparisons will be written into the result cell (purple).

This process is being repeated towards the right side of the table until you have completed row 3.

Step 4: Fill in the Table for Length 4 Substrings

Now we repeat this for length 4 substrings.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			
5.						
6.						

In this example we see what happens with empty sets when matching the values to non-terminal ones. We always need two parts `X` and `Y`. So if we ever encounter an empty set, we can already discard finding a match for that. But be careful, you might find an other matching pair for that field since we here (in row 4) always have 3 pairs to compare. Here in this example our three pairs are: ``SB'', ``∅∅'', ``SB''. No non-terminal values exist for these pairs so we just write ∅ into the cell.

Step 5: Fill in the Table for Length 5 Substrings

Here we will follow the pattern from before. So there won't be much explaining.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			
5.	∅	A, B				
6.						

Step 6: Fill in the Table for Length 6 Substrings

Exact same patter. The only thing to know now is that we here need a ``S'' as result to confirm the word matching the grammar.

0.	a	b	a	c	b	a
1.	S	B	S	B	B	S
2.	∅	A, B	∅	B	A, B	
3.	S	B	∅	A, B		
4.	∅	B	S			
5.	∅	A, B				
6.	S					

We here see that we get `S` as our final value, confirming that the word `abacba` matches our grammar.

Why do we exactly look for S?

In the final cell (6,1) which represents the whole string "abacba", the presence of `S` indicates that the entire string can be derived from the start symbol `S`, confirming that "abacba" is part of the language defined by the grammar.

If the bottom-left cell contains `S` among other non-terminals, it confirms that there is at least one derivation sequence that starts from `S`.

However, the presence of other non-terminals alone does not confirm the string is derivable from `S` and therefore part of the language.

Quick Note

I have always used the first column to show you the pattern (unless in first example I showed you the first and last). You will need to continue the patterns in the table to the complete right to fill in all these rows. So you can always take the colored fields and move the colors one to the right, do the same thing again, and repeat until the row is completed.

11 Chapter 10 - Compiler I : Syntax Analysis

In this part, we build a syntax analyzer for the Jack language. The syntax analyzer will output the syntactic structure of the Jack program in an XML format. This step is essential for understanding the structure of the program before generating executable code.

11.1 Key Concepts

Lexical Analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. It involves reading the input characters and grouping them into meaningful sequences known as tokens while ignoring white spaces and comments. For example, in the code:

```
while (count <= 100) {
    count++;
}
```

The lexical analyzer would generate tokens such as `while`, `(`, `count`, `<=`, `100`, `)`, {, `count`, `++`, `;` and }. These tokens are the basic building blocks for further syntactic analysis.

Context-Free Grammars

Context-free grammars (CFG) define the syntax rules of a language. They specify how tokens can be combined to form constructs in the language. CFGs consist of terminals (tokens) and non-terminals (syntactic categories). For instance, the Jack language grammar uses specific notations for terminals and non-terminals to describe the structure of classes, methods, variable declarations, and expressions. An example rule might state an expression can be formed by combining two terms with an operator, such as `term op term`.

Parsing

Parsing is the process of analyzing a sequence of tokens to determine its grammatical structure according to a given number. The parser matches the token stream against the grammar rules, building a parse tree or derivation tree that represents the structure of the program. In recursive descent parsing, the parser uses a top-down approach to handle the nested structure of the language. For example, the parser would recognize a sequence like `if (x < y)` { `doSomething();` } by breaking it down into its components: `if` statement, condition, and block of statement.

11.2 Implementation Details

11.2.1 JackAnalyzer Module

The JackAnalyzer module serves as the top-level driver that sets up and invokes the tokenizer and parser modules.

1. Create a JackTokenizer from the input file.
2. Create an output XML file.
3. Use the CompilationEngine to parse the tokens and generate the XML output.

11.2.2 JackTokenizer Module

The JackTokenizer module is responsible for removing comments and whitespaces. It as well breaks the input stream into Jack-language tokens. The module includes methods like:

- `hasMoreTokens`: Checks if there are more tokens in the input
- `advance`: Gets the next token from the input.
- `tokenType, keyWord, symbol, identifier, stringVal`: Return the type and value of the current token.

11.2.3 CompilationEngine Module

The CompilationEngine module handles the actual compilation process. It reads tokens from the JackTokenizer and generates XML output. The module includes methods like:

- `compileClass`: Compiles a complete class.
- `compileClassVarDec`: Compiles static or field declarations.

- `compileSubroutine`: Compiles methods, functions or constructors.

- `compileParameterList`, `compileVarDec`, `compileStatements`, `compileDo`, `compileLet`, `compileWhile`, `compileReturn`, `compileIf`, `compileExpression`, `compileTerm`, `compileExpressionList`: Compile various language constructs as per the Jack grammar.

11.2.4 Example XML Output

The syntax analyzer generates an XML file reflecting the syntactic structure of the Jack program. For example, given a Jack class definition, the XML output would represent the hierarchical structure of the class, including its subroutines, variable declarations, and statements.

11.2.5 Important Tables and Concepts

Lexical Elements
The Jack language includes five types of terminal elements (tokens):

- Keywords: `class, constructor, function, method, etc.`

- Symbols: `‘‘{’’, ‘‘}’’, ‘‘(’’, ‘‘)’’, ‘‘[’’, ‘‘]’’, ‘‘.’’, ‘‘,’’, ‘‘;’’, ‘‘+’’, ‘‘-’’, ‘‘*’’, ‘‘/’’, ‘‘&’’, ‘‘|’’, ‘‘<’’, ‘‘>’’, ‘‘=’’, ‘‘ ’’`

- Integer Constants: A decimal number in the range 0 .. 32767.

- String Constants: A sequence of Unicode characters enclosed in double quotes.

- Identifiers: A sequence of letters, digits, and underscores not starting with a digit.

Lexical Elements

The Jack grammar defines the syntax for classes, subroutines, variable declarations, statements, expressions, etc. Here is a simplified example:

```
class: 'class' className '{' classVarDec*
     subroutineDec* '}'

classVarDec: ('static' | 'field') type
     varName (',' varName)* ';'

type: 'int' | 'char' | 'boolean' | className

subroutineDec: ('constructor' | 'function' |
     'method') ('void' | type) subroutineName
      '(' parameterList ')' subroutineBody

parameterList: ((type varName) (',' type
     varName)*)?

subroutineBody: '{' varDec* statements '}'

varDec: 'var' type varName (',' varName)* ';'
```

11.3 Translating Jack to VM code

To translate Jack code to VM code, we need both the syntax analysis (covered in Compiler Part 1) and the code generation capabilities (which will be covered in Compiler Part 2). However, we can still provide some basic examples of translating simple Jack code to VM code based on the concepts introduced so far.

Example 1 - Variable Declaration

```
class Main {
   function void main() {
      var int a;
      let a = 5;
   }
}
```

This simple main function with a variable declaration can be translated to:

```
function Main.main 1
push constant 5
pop local 0
```

Example 2 - Arithmetic Operation

```
class Main {
   function void main() {
      var int a;
      var int b;
      let a = 5;
      let b = a + 3;
   }
}
```

This simple main function with an arithmetic operation (adding 3 to a variable) can be translated like this:

```
function Main.main 2
push constant 5
pop local 0
push local 0
push constant 3
add
pop local 1
```

Example 3 - Simple If-Statement

```
class Main {
    function void main() {
        var int a;
        let a = 5;
        if (a > 3) {
            let a = a - 1;
        }
    }
}
```

This simple main function with an if-statement can be translated to:

```
function Main.main 1
push constant 5
pop local 0
push local 0
push constant 3
gt
if-goto IF_TRUE
goto IF_FALSE
label IF_TRUE
push local 0
push constant 1
sub
pop local 0
label IF_FALSE
```

Example 4 - Simple While-Loop

```
class Main {
    function void main() {
        var int a;
        let a = 0;
        while (a < 5) {
            let a = a + 1;
        }
    }
}
```

This simple main function with a while-loop can be translated to:

```
function Main.main 1
push constant 0
pop local 0
label WHILE_EXP
push local 0
push constant 5
lt
if-goto WHILE_TRUE
goto WHILE_FALSE
label WHILE_TRUE
push local 0
push constant 1
add
pop local 0
goto WHILE_EXP
label WHILE_FALSE
```

12 Chapter 11 - Compiler II : Code Generation

In this chapter, we extend the syntax analyzer developed in Part 1 into a full-scale compiler that converts each high-level construct into an equivalent series of VM operations. This apporach follows the modular analysis-synthesis paradigm underlying the construction of most compilers.

12.1 Key Concepts

12.1.1 Code Generation

Code generation is the process of converting the parsed representation of a program into target code that can be executed on a virtual machine. The translation process involves managing a symbol table, representing and generating code for variables, objects, and arrays, and translating control flow commands into low-level instructions.

12.1.2 Symbol Table

The symbol table keeps track of all identifiers introduced by the program. It records what each identifier stands for, ether it's a variable, class name, or function name, and maps them to constructs in the target language.

Here you can see an example of a symbol table.

Name	**Type**	**Kind**	**#**
nAccounts	int	static	0
id	int	field	0
name	String	field	1
balance	int	field	2
sum	int	argument	0
status	boolean	local	0

12.1.3 Data Translation

Programs manipulate various types of variables, including simple types like integers and booleans, and complex types like arrays and objects. The compiler must map these variables to an equivalent representation suitable for the target platform and manage their life cycle and scope.

12.1.4 Handling Variables, Arrays and Objects

Different kinds of variables have different life cycles. For Example:

- Static Variables: A single copy is kept alive during the program's runtime.
- Object Fields: Each object instance has different copies of its instance variables.
- Local and Argument Variables: New copies are created each time a subroutine is called.

Arrays are stored as sequences of consecutive memory locations. The array name is treated as a pointer to the base address of the memory block allocated to store the array.

Object instances are represented by a pointer containing the base address of their fields. The memory space for the object is allocated when the object is created via a class constructor.

12.1.5 Commands Translation

High-level commands are translated into the target language using the following strategies:

Expression Evaluation

Expressions are translated into stack-based VM code using recursive post-order traversal of the underlying parse tree.

```
x + g(2, y, -z) * 5
```

This expression turns into:

```
push x
push 2
push y
push z
neg
call g
push 5
call multiply
add
```

Flow Control

High-level control flow structures like `if`, `while`, `for` are translated into low-level VM commands using conditional and unconditional `goto` commands.

```
if (cond) {
    // code block
} else {
    // else block
}
```

This expression turns into:

```
# VM code for computing ~(cond)
if-goto L1
# VM code for the "if" block
goto L2
label L1
# VM code for the "else" block
label L2
```

12.2 Implementation

JackCompiler

The JackCompiler module operates on a given source, which can be either a file name or a directory name containing one or more .jack files. It creates a JackTokenizer and an output .vm file for each input .jack file, using the CompilationEngine, SymbolTable, and VMWriter modules to generate the output file.

JackTokenizer

The tokenizer module breaks the input stream into Jack-language tokens and provides methods to retrieve the type and value of the current token.

SymbolTable Module

The symbol table manages identifier scopes and properties. It uses two hash tables: one for the class scope and another for the subroutine scope.

VMWriter Module

The VMWriter module emits VM commands into a file, using the VM command syntax.

CompilationEngine Module

The CompilationEngine reads the input from a JackTokenizer and writes its output into a VMWriter. It is organized as a series of `compilexxx()` routines, each handling a specific syntactic element of the Jack language.

12.3 Translation Steps

Let's take a very simple Jack language statement and show the steps it goes through from the source code to vm code.

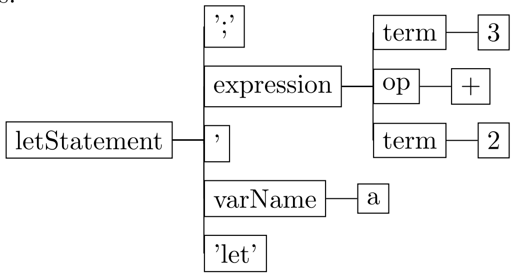
```
let a = 2 + 3;
```

12.3.1 Tokenization

The tokenizer breaks down the source code into individual tokens: `let, a, =, 2, +, 3, ;`.

12.3.2 Parsing

The parser organizes these tokens into a hierarchical structure (tree) based on the grammar rules of the Jack language. So the parse tree looks like this:



12.3.3 Symbol Table Management

The symbol table keeps track of variable names and their types, scope, and locations.

Name	Type	Kind	#
a	int	local	0

12.3.4 Code Generation

Translate the parse tree into intermediate representation (IR) and then into VM code.

Translation Process:

1. Expression Handling
   - **Expression:** `2 + 3`
   - **Term:** `2` (VM: `push constant 2`)
   - **Term:** `3` (VM: `push constant 3`)
   - **Operation:** `+` (VM: `add`)
2. Assignment Handling
   - **Variable:** `a`
   - **Assignment Operator:** `=`
   - **Result of Expression:** is assigned to `a` (VM: `pop local 0`)

12.3.5 VM Code Output

Combine all the parts into the final VM code.

```
push constant 2 // Push the constant 2 onto
     the stack
push constant 3 // Push the constant 3 onto
     the stack
add // Add the top two elements of the stack
pop local 0 // Pop the result of the addition
      and store it in the local variable 'a'
```

12.4 Summary of all forms

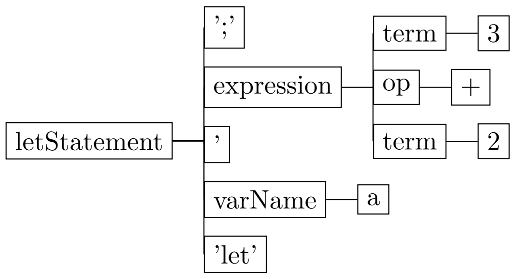
Source Code:

```
let a = 2 + 3;
```

Tokenized:

`let, a, =, 2, +, 3, ;`

Parsed:



Symbol Table:

Name	Type	Kind	#
a	int	local	0

Intermediate Representation:

1. Expression Handling
   - `push constant 2`
   - `push constant 3`
   - `add`
2. Assignment Handling
   - `pop local 0`

Final VM Code:

```
push constant 2
push constant 3
add
pop local 0
```

13 Chapter 12 - Operating System

The Jack Operating System (OS) is providing a high-level interface to interact with the hardware. The OS comprises eight main classes, each offering different services that are crucial for running Jack programs smoothly. These classes include Memory, Array, Math, String, Output, Screen, Keyboard and Sys. Each class encapsulates specific functionalities that simplify complex operations, making them accessible through simple commands.

13.1 The 8 main classes

13.1.1 Memory Management

Memory management is a fundamental aspect of the Jack OS. The `Memory` class provides functions to access and manipulate the computer's RAM. Key operations include allocating and deallocating memory segments. For example, the `alloc` function is used to request a block of memory, while `deAlloc` frees the previously allocated memory. This efficient memory handling is crucial for ensuring that the system runs smoothly without memory leaks.

13.1.2 Array Handling

The `Array` class simplifies the creation and manipulation of arrays. Arrays in Jack are dynamically allocated, and the `Array.new` function creates a new specified size. This class also includes methods to resize arrays and manage array elements, allowing for flexible data storage and manipulation within Jack programs.

13.1.3 Mathematical Operations

Mathematical computations are handled by the `Math` class, which includes basic operations like addition, subtraction, multiplication, and division. The `Math` class also provides more complex functions such as calculating the square root or trigonometric functions. These operations are implemented efficiently to support various computational needs within Hack programs.

13.1.4 String Manipulation

The `String` class provides tools for creating and manipulating strings. Functions like `String.new`, `String.appendChar` and `String.setChar` allow for dynamic string operations. This class is essential for handling text within Jack programs, enabling functionalities such as building messages, modifying text, and performing string comparisons.

13.1.5 Output Handling

The `Output` class is responsible for displaying text on the screen. Methods like `Output.printString`, `Output.printInt`, `Output.printChar` are used to print different types of data. Additionally, the `Output.moveCursor` method positions the cursor on the screen, and `Output.clearScreen` clears the display. This class is vital for creating user interfaces and displaying program output.

13.1.6 Screen Management

The `Screen` class provides low-level drawing operations, allowing programs to manipulate the screen at the pixel level. Functions include `Screen.drawPixel`, `Screen.drawLine` and `Screen.drawRectangle`. These methods are used to create graphical elements in Jack programs, supporting the development of games and graphical user interfaces.

13.1.7 Keyboard Input

The `Keyboard` class captures user input from the keyboard. Functions like `Keyboard.keyPressed` and `Keyboard.readKey` detect key presses and read the value of the pressed key. This class is essential for interactive programs that require user input to function.

13.1.8 System Operations

The `Sys` class handles system-level operations, such as program initialization and termination. Methods like `Sys.init` are used to start the OS, while `Sys.halt` stops the execution of the program. This class ensures that Jack programs can be executed, managed, and terminated correctly.

13.2 Key Algorithms

All implementations of the used helper functions can be found in the appendix.

13.2.1 Multiplication Algorithm

Multiplication in Jack OS is implemented using repeated addition.

1. **Initialization:** Initialize the result to 0. This variable will store the final product.
2. **Repeated Addition:** Add the multiplication to the result the number of times specified by the multiplier.
3. **Termination:** Once the multiplier has been decremented to 0, the loop ends and the result holds the product.

Example with Jack Code

```
function Math.multiply 2
    push argument 0 // multiplicand
    push argument 1 // multiplier
    call Math.multiplyHelper 2
    return
```

13.2.2 Division Algorithm

Division in Jack OS uses repeated subtraction to determine the quotient and remainder.

1. **Initialization:** Initialize the quotient to 0.
2. **Repeated Subtraction:** Subtract the divisor from the dividend until the dividend is less than the divisor. Increment the quotient each time.
3. **Termination:** When the dividend is less than the divisor, the quotient holds the result, and the remaining dividend is the remainder.

Example with Jack Code

```
function Math.divide 2
    push argument 0 // dividend
    push argument 1 // divisor
    call Math.divideHelper 2
    return
```

13.2.3 Line Drawing Algorithm

The `Screen.drawLine` method uses Bresenham's algorithm to draw a line between two points. This algorithm determines which points in a grid should be plotted to form a close approximation to a staright line between the start and end points.

1. **Initialization:** Calculate the differences `dx` and `dy` between the start and end points. Determine the steps to increment `x` and `y`.
2. **Decision Parameter:** Initialize the dicision parameter to decide when to increment `y` while drawing the line.
3. **Plot Points:** Iterate over the `x`-coordinates, plotting points and updating the decision parameter to decide whether to increment the `y`-coordinate.

Example with Jack Code

```
function Screen.drawLine 0
    push argument 0 // x1
    push argument 1 // y1
    push argument 2 // x2
    push argument 3 // y2
    call Screen.drawLineHelper 4
    return
```

13.2.4 Heap Management

Heap management involves allocating and deallocating memory dynamically. The heap is managed by keeping track of free and used memory blocks.

1. **Allocation:** To allocate memory, the system searches for a free block large enough to fullfill the request. If found, it marks the block as used and returns a pointer to it.

2. **Deallocation:** To deallocate memory, the system marks the blocks as free, making it available for future allocations.

Example with Jack Code

```
function Memory.alloc 1
    push argument 0 // Requested size
    call Memory.allocHelper 1
    return
```

13.2.5 Text Output

Text output is managed by the `Output` class, which handles printing characters, strings, and integers to the screen.

1. **Printing Characters:** The `printChar` method prints a single character at the current cursor position.

2. **Printing Strings:** The `printString` method iterates over a string and prints each character.

3. **Printing Integers:** The `printInt` method converts an integer to its string representation and prints it.

Example with Jack Code

```
function Output.printString 1
    push argument 0 // String to print
    call Output.printStringHelper 1
    return
```

14 Chapter 13 - System Models (Theory)

Will maybe come in future.

15 Chapter 14 - Predictability (Theory)

Will maybe come in future.

16 Conclusions

I will write a conclusion as soon as I have implemented everything. I as well will then change bunch of stuff probably in the documentation that I found were missing or misleading after implementation. So check back at any point to view if there is a new version if you're interested or need it.

17 Appendix

17.1 VM Translation to Hack table

VM Command	Hack Assembly Code

`push constant x`	`// Push constant x`
	`@x // Load constant x into A`
	`D=A // Store A in D`
	`@SP // Address stack pointer`
	`A=M // Address top of the stack`
	`M=D // Store D in the stack`
	`@SP // Address stack pointer`
	`M=M+1 // Increment stack pointer`

`push local x`	`// Push local x`
	`@LCL // Load LCL base address`
	`D=M // Store LCL base in D`
	`@x // Load x into A`
	`A=D+A // Address LCL[x]`
	`D=M // Store LCL[x] in D`
	`@SP // Address stack pointer`
	`A=M // Address top of the stack`
	`M=D // Store D in the stack`
	`@SP // Address stack pointer`
	`M=M+1 // Increment stack pointer`

`push argument x`	`// Push argument x`
	`@ARG // Load ARG base address`
	`D=M // Store ARG base in D`
	`@x // Load x into A`
	`A=D+A // Address ARG[x]`
	`D=M // Store ARG[x] in D`
	`@SP // Address stack pointer`
	`A=M // Address top of the stack`
	`M=D // Store D in the stack`
	`@SP // Address stack pointer`
	`M=M+1 // Increment stack pointer`

`push this x`	`// Push this x`
	`@THIS // Load THIS base address`
	`D=M // Store THIS base in D`
	`@x // Load x into A`
	`A=D+A // Address THIS[x]`
	`D=M // Store THIS[x] in D`
	`@SP // Address stack pointer`
	`A=M // Address top of the stack`
	`M=D // Store D in the stack`
	`@SP // Address stack pointer`
	`M=M+1 // Increment stack pointer`

VM Command	**Hack Assembly Code**

`push that x`	`// Push that x`
	`@THAT       // Load THAT base address`
	`D=M      // Store THAT base in D`
	`@x      // Load x into A`
	`A=D+A       // Address THAT[x]`
	`D=M      // Store THAT[x] in D`
	`@SP      // Address stack pointer`
	`A=M      // Address top of the stack`
	`M=D      // Store D in the stack`
	`@SP      // Address stack pointer`
	`M=M+1       // Increment stack pointer`

`push temp x`	`// Push temp x`
	`@5      // Load base address of temp`
	`D=A      // Store base address in D`
	`@x      // Load x into A`
	`A=D+A       // Address temp[x]`
	`D=M      // Store temp[x] in D`
	`@SP      // Address stack pointer`
	`A=M      // Address top of the stack`
	`M=D      // Store D in the stack`
	`@SP      // Address stack pointer`
	`M=M+1       // Increment stack pointer`

`push pointer x`	`// Push pointer x`
	`@3      // Load base address of pointer`
	`D=A      // Store base address in D`
	`@x      // Load x into A`
	`A=D+A       // Address pointer[x]`
	`D=M      // Store pointer[x] in D`
	`@SP      // Address stack pointer`
	`A=M      // Address top of the stack`
	`M=D      // Store D in the stack`
	`@SP      // Address stack pointer`
	`M=M+1       // Increment stack pointer`

`push static x`	`// Push static x`
	`@Foo.x       // Load static Foo.x address`
	`D=M      // Store Foo.x in D`
	`@SP      // Address stack pointer`
	`A=M      // Address top of the stack`
	`M=D      // Store D in the stack`
	`@SP      // Address stack pointer`
	`M=M+1       // Increment stack pointer`

VM Command	Hack Assembly Code
pop local x	// Pop local x
	@LCL // Load LCL base address
	D=M // Store LCL base in D
	@x // Load x into A
	D=D+A // Address LCL[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address LCL[x]
	M=D // Store D in LCL[x]
pop argument x	// Pop argument x
	@ARG // Load ARG base address
	D=M // Store ARG base in D
	@x // Load x into A
	D=D+A // Address ARG[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address ARG[x]
	M=D // Store D in ARG[x]
pop this x	// Pop this x
	@THIS // Load THIS base address
	D=M // Store THIS base in D
	@x // Load x into A
	D=D+A // Address THIS[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address THIS[x]
	M=D // Store D in THIS[x]

VM Command	Hack Assembly Code
pop that x	// Pop that x
	@THAT // Load THAT base address
	D=M // Store THAT base in D
	@x // Load x into A
	D=D+A // Address THAT[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address THAT[x]
	M=D // Store D in THAT[x]
pop temp x	// Pop temp x
	@5 // Load base address of temp
	D=A // Store base address in D
	@x // Load x into A
	D=D+A // Address temp[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address temp[x]
	M=D // Store D in temp[x]
pop pointer x	// Pop pointer x
	@3 // Load base address of pointer
	D=A // Store base address in D
	@x // Load x into A
	D=D+A // Address pointer[x]
	@R13 // Use R13 as temp storage
	M=D // Store address in R13
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@R13 // Address temp storage
	A=M // Address pointer[x]
	M=D // Store D in pointer[x]
pop static x	// Pop static x
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	@Foo.x // Address Foo.x
	M=D // Store D in Foo.x

VM Command	**Hack Assembly Code**

`add`	`// Add top two stack values`
	`@SP      // Address stack pointer`
	`AM=M-1       // Decrement SP and address top`
	`D=M      // Store top of stack in D`
	`A=A-1      // Address next top of stack`
	`M=M+D      // Add D to new top of stack`

`sub`	`// Subtract top two stack values`
	`@SP      // Address stack pointer`
	`AM=M-1       // Decrement SP and address top`
	`D=M      // Store top of stack in D`
	`A=A-1      // Address next top of stack`
	`M=M-D      // Subtract D from new top of stack`

`neg`	`// Negate top stack value`
	`@SP      // Address stack pointer`
	`A=M-1      // Address top of stack`
	`M=-M      // Negate the value`

`eq`	`// Equal comparison`
	`@SP      // Address stack pointer`
	`AM=M-1       // Decrement SP and address top`
	`D=M      // Store top of stack in D`
	`A=A-1      // Address next top of stack`
	`D=M-D      // Subtract D from new top of stack`
	`@EQUAL       // Address EQUAL label`
	`D;JEQ      // If equal, jump to EQUAL`
	`@SP      // Address stack pointer`
	`A=M-1      // Address top of stack`
	`M=0      // Not equal, set to false`
	`@END      // Address END label`
	`0;JMP      // Jump to END`
	`(EQUAL)       // EQUAL label`
	`@SP      // Address stack pointer`
	`A=M-1      // Address top of stack`
	`M=-1      // Equal, set to true`
	`(END)      // END label`

VM Command	Hack Assembly Code
gt	// Greater than comparison
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	A=A-1 // Address next top of stack
	D=M-D // Subtract D from new top of stack
	@GREATER // Address GREATER label
	D;JGT // If greater, jump to GREATER
	@SP // Address stack pointer
	A=M-1 // Address top of stack
	M=0 // Not greater, set to false
	@END // Address END label
	0;JMP // Jump to END
	(GREATER) // GREATER label
	@SP // Address stack pointer
	A=M-1 // Address top of stack
	M=-1 // Greater, set to true
	(END) // END label
lt	// Less than comparison
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	A=A-1 // Address next top of stack
	D=M-D // Subtract D from new top of stack
	@LESS // Address LESS label
	D;JLT // If less, jump to LESS
	@SP // Address stack pointer
	A=M-1 // Address top of stack
	M=0 // Not less, set to false
	@END // Address END label
	0;JMP // Jump to END
	(LESS) // LESS label
	@SP // Address stack pointer
	A=M-1 // Address top of stack
	M=-1 // Less, set to true
	(END) // END label
and	// Bitwise AND
	@SP // Address stack pointer
	AM=M-1 // Decrement SP and address top
	D=M // Store top of stack in D
	A=A-1 // Address next top of stack
	M=M&D // AND top two stack values

VM Command	**Hack Assembly Code**
`or`	`// Bitwise OR`
	`@SP      // Address stack pointer`
	`AM=M-1      // Decrement SP and address top`
	`D=M      // Store top of stack in D`
	`A=A-1      // Address next top of stack`
	`M=M|D      // OR top two stack values`
`not`	`// Bitwise NOT`
	`@SP      // Address stack pointer`
	`A=M-1      // Address top of stack`
	`M=!M      // NOT top of stack value`

17.2 VM Implementation of Function Commands

17.2.1 call

```
@RETURN_ADDRESS // Push return address
D=A
@SP
A=M
M=D
@SP
M=M+1
@LCL // Save LCL
D=M
@SP
A=M
M=D
@SP
M=M+1
@ARG // Save ARG
D=M
@SP
A=M
M=D
@SP
M=M+1
@THIS // Save THIS
D=M
@SP
A=M
M=D
@SP
M=M+1
@THAT // Save THAT
D=M
@SP
A=M
M=D
@SP
M=M+1
@SP // Reposition ARG
D=M
@5
D=D-A
@2
D=D-A
@ARG
M=D
@SP // Reposition LCL
D=M
@LCL
M=D
@SimpleFunction // Transfer control
0;JMP
(RETURN_ADDRESS)
```

17.2.2 return

```
@LCL // Frame = LCL
D=M
@R13 // Save frame
M=D
@5
A=D-A
D=M
@R14 // Save return address in temp var
M=D
@SP // Reposition the return value for the
     caller
AM=M-1
D=M
@ARG
A=M
M=D
@ARG // Restore SP of the caller
D=M+1
@SP
M=D
@R13 // Restore THAT
AM=M-1
D=M
@THAT
M=D
@R13 // Restore THIS
AM=M-1
D=M
@THIS
M=D
@R13 // Restore ARG
AM=M-1
D=M
@ARG
M=D
@R13 // Restore LCL
A=M-1
D=M
@LCL
M=D
@R14 // Goto return address
A=M
0;JMP
```

17.3 Example Process of VM Parser and CodeWriter

1. Parser reads command:

```
    push constant 10
```

2. Parser Tokenizes the command:

   - Command Type: `push`
   - Segment: `constant`
   - Value: `10`

3. CodeWriter translates the command:

```
@10
D=A
@SP
A=M
M=D
@SP
M=M+1
```

This example illustrates how the Parser reads and tokenizes a VM command and how the CodeWriter translates it into Hack assembly code.

By clearly defining the responsibilities of the Parser and CodeWriter, the VM translator can effectively convert high-level VM commands into low-level assembly instructions.

17.4 Operating System Helpers

17.4.1 Multiply Helper Implementation

```
function Math.multiplyHelper 2
    push constant 0
    pop local 0 // result = 0
    label LOOP_START
    push argument 1
    push constant 0
    eq
    if-goto END_LOOP
    push argument 0
    push local 0
    add
    pop local 0 // result += multiplicand
    push argument 1
    push constant 1
    sub
    pop argument 1 // multiplier--
    goto LOOP_START
    label END_LOOP
    push local 0
    return
```

17.4.2 Divide Helper Implementation

```
function Math.divideHelper 2
    push constant 0
    pop local 0 // quotient = 0
    label DIV_LOOP_START
    push argument 0
    push argument 1
    lt
    if-goto END_DIV_LOOP
    push argument 0
    push argument 1
    sub
    pop argument 0 // dividend -= divisor
    push local 0
    push constant 1
    add
    pop local 0 // quotient++
    goto DIV_LOOP_START
    label END_DIV_LOOP
    push local 0
    return
```

17.4.3 Line Drawing Helper

```
function Screen.drawLineHelper 0
    // Calculate dx, dy, and initialize the
         decision parameter
    push argument 2
    push argument 0
    sub
    pop local 0 // dx = x2 - x1

    push argument 3
    push argument 1
    sub
    pop local 1 // dy = y2 - y1

    push local 0
    push constant 0
    lt
    if-goto NEGATIVE_DX
    goto POSITIVE_DX

    label NEGATIVE_DX
    push local 0
    neg
    pop local 0 // dx = -dx
    goto DX_HANDLED

    label POSITIVE_DX
    // do nothing, dx is already positive

    label DX_HANDLED

    push local 1
    push constant 0
    lt
    if-goto NEGATIVE_DY
    goto POSITIVE_DY

    label NEGATIVE_DY
    push local 1
    neg
    pop local 1 // dy = -dy
    goto DY_HANDLED

    label POSITIVE_DY
    // do nothing, dy is already positive

    label DY_HANDLED

    // Start plotting the line
    label PLOT_LINE
    // Decision parameter updates and
         plotting points go here
    // This part involves more detailed logic
          to implement the Bresenham
         algorithm correctly

    // For simplicity, let's assume the
         decision parameter is handled and
```

```
        the line is drawn
    push constant 1 // Dummy return value for
         simplicity
    return
```

17.4.4 Alloc Helper

```
function Memory.allocHelper 1
    // Simplified example of searching and
        allocating memory block
    push constant 1 // Dummy pointer to
        allocated memory
    return
```

17.4.5 Print String Helper

```
function Output.printStringHelper 1
    // Simplified example of printing each
        character in the string
    push constant 0 // Dummy return value for
         simplicity
    return
```