

**课程小作业报告**

**计算机基本原理**

|  |  |
| --- | --- |
| **课程名称：** | **计算机与软件工程概论** |
| **学生姓名：** | **快要变成猪了QwQ** |
| **学生学号：** | **XXX** |
| **学生专业：** | **软件工程** |
| **开课学期：** | **2023-2024学年第1学期** |

**软件学院**

**2023年10月**

**课程作业1：计算机与计算思维**

## 问题1：计算机基本原理

以Pep/8虚拟计算机为基础：

1： 学习数据表示、二进制运算、布尔逻辑运算、门电路与加法器、CPU结构及冯.诺依曼型计算机体系结构及原理（程序存储及取指令-执行指令）、CPU指令、编程语言（Pep/8机器语言及其汇编语言）；

2：利用Pep/8计算机，求解如下问题; (注意，这里问题描述采用类c++语言)

int a, b, c; // 三个整型变量

a = 1;

b = 2;

c = a + b; // 加法，并输出c

### 知识与技术总结

报告要求，就解决问题过程中，学习到的知识与技术，按章节总结阐述，突出重点；

#### 数据表示

First of all, A basic problem at all levels of abstraction is the mismatch between the form of the information to be processed and the language to represent it. A program in machine language

processes bits. A program in a high-order language processes items such as arrays. So **matching the information to the language** is a basic problem at all levels of abstraction in the modeling process of problem solving.

**Binary** is a base-2 number system that uses only two digits, 0 and 1, to represent all numbers.

In binary, each digit’s place value is twice as much as that of the next digit to the right. **Signed integers** use two’s complement binary representation in which the first bit is the sign bit and the remaining bits determine the magnitude. For positive numbers, the two’s complement representation is identical to the unsigned representation. For negative numbers, however, the two’s complement of a number is obtained by taking 1 plus the ones’ complement of the corresponding positive number.

Similarly, **hexadecimal** (hereinafter referred to as “hex”) is a base-16number system whose digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. One hex digit corresponds to four binary digits and vice versa, e.g., (6 A)H=(0110 1010)B. Also hex numbers are often used for abbreviated ASCII and abbreviated memory address.

A **floating point number** is stored in a cell with three fields—a one-bit sign field, a field for the

exponent, and a field for the significand. Except for special values, numbers are stored in binary

scientific notation with a hidden bit to the left of the binary point that is assumed to be 1. The

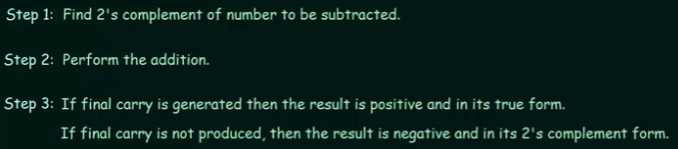
exponent is stored in an excess representation. Four special values are zero, infinity, NaN, and

denormalized numbers. The IEEE 754 standard defines the number of bits in the exponent and

significand fields to be 8 and 23 for **single precision**, and 11 and 52 for **double precision**.

* + 1. **二进制运算**

In the field of Digital Electronics, binary subtraction is usually performed using 2’s complement.

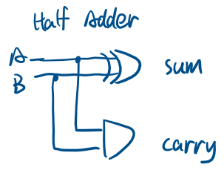
Steps are as followed:

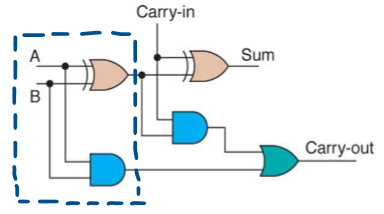
Specially, **ASL**, which stands for arithmetic shift left, multiplies a binary value by 2, and **ASR**, which stands for arithmetic shift right, divides a binary value by 2.

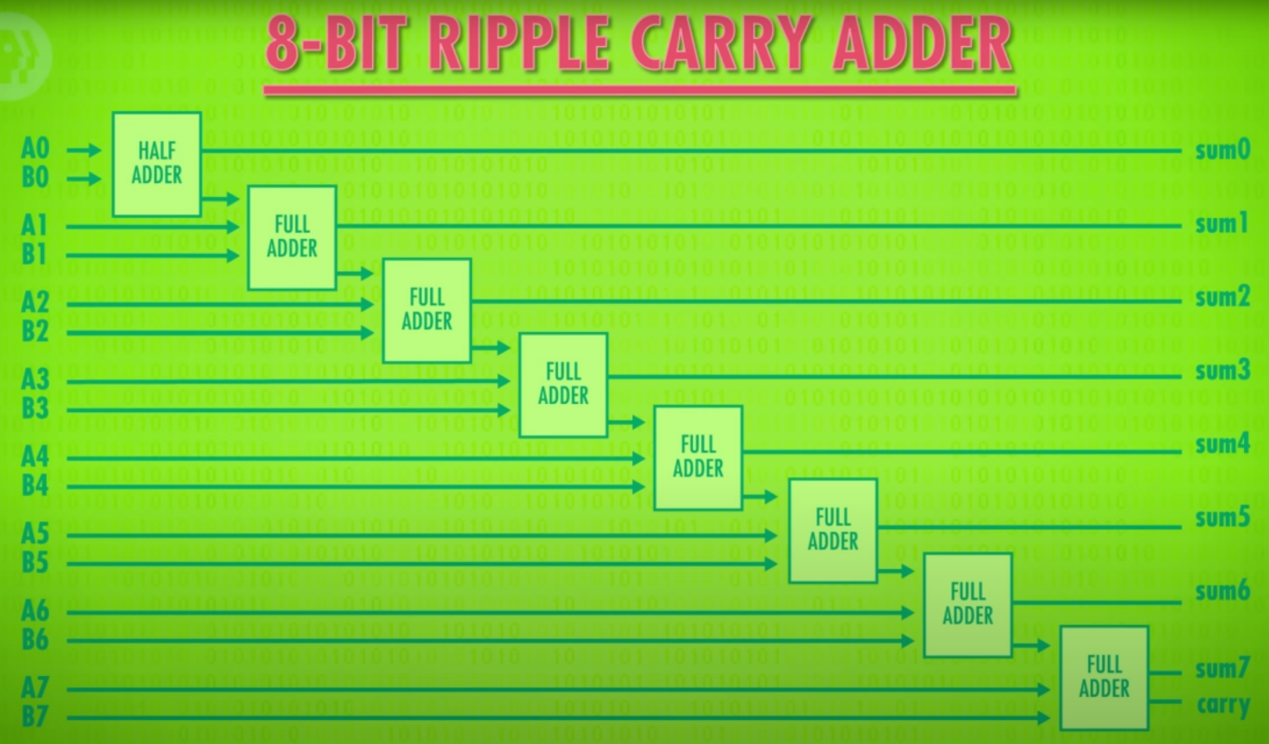
* + 1. **布尔逻辑运算和门电路与加法器**

**Boolean logical operators** on binary integers include AND, OR, XOR, NAND, NOR and NOT, all

of which have corresponding **truth tables** and notations referred to as **Gates**. We can perform single bit addition using **a Half Adder** which is composed of a AND gate and a XOR gate

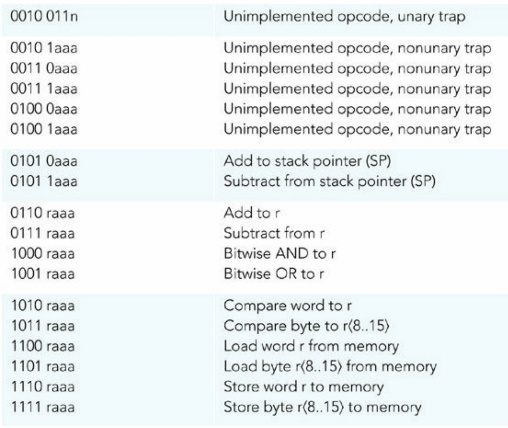
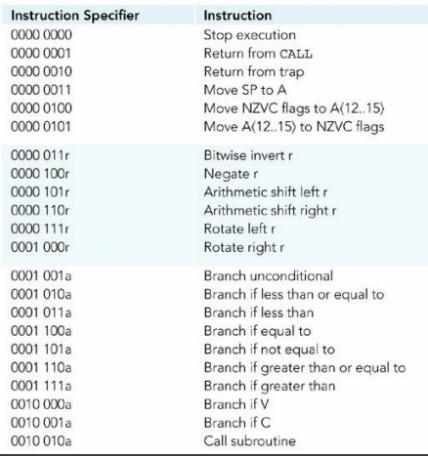
like this:  However, a Half Adder does not take the carry bit produced by the preceding addition into its account, so

we design **a Full Adder** to handle this issue:  And finally, we connect several Full Adders together to build **a Ripple Carry Adder**:

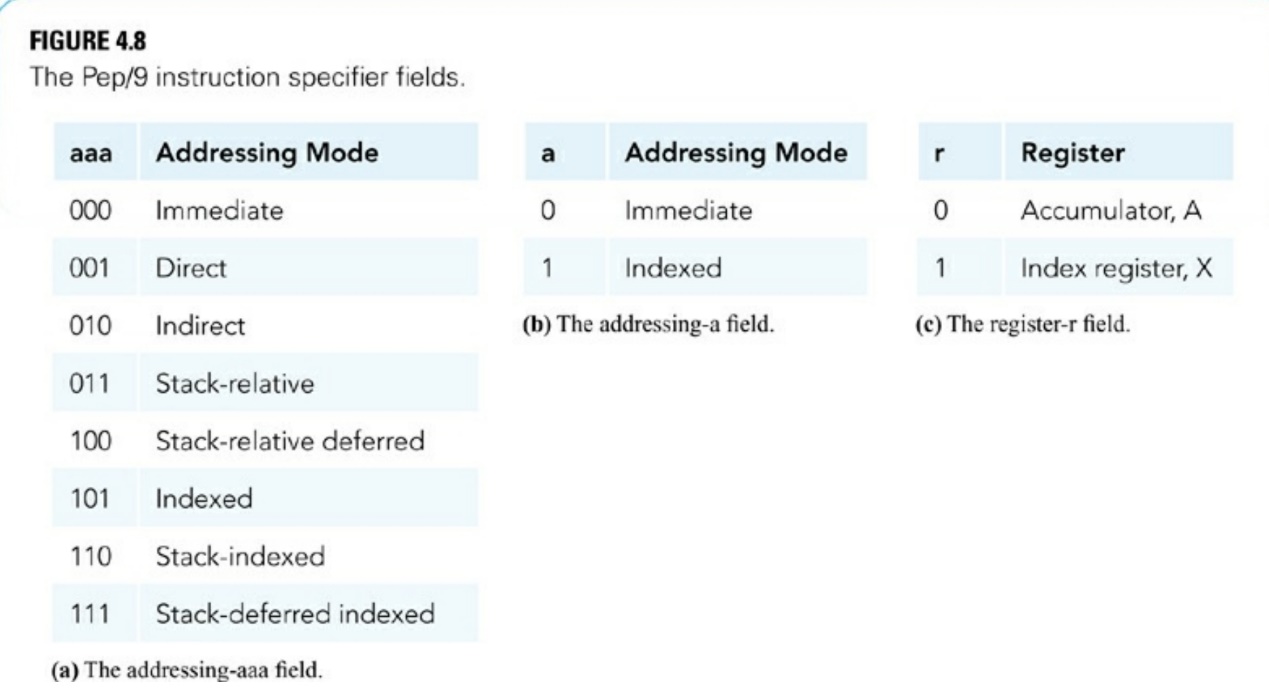


* + 1. **CPU结构及冯诺依曼型计算机体系结构及原理、CPU指令**

Virtually all commercial computers are based on the **von Neumann design principle**, in which main memory stores both data and instructions. The three components of a von Neumann machine are the **central processing unit** (CPU), **main memory** with **memory-mapped I/O devices**, and **disk**. The CPU contains a set of registers, one of which is the program counter (PC), which stores the address of the instruction to be executed next.

The CPU has an instruction set wired into it. **An instruction** consists of an instruction specifier and an operand specifier. The **instruction specifier**, in turn, consists of an opcode and possibly a register field and an addressing mode field. The **opcode** determines which instruction in the instruction set is to be executed. The **register field** determines which register participates in the operation. The **addressing mode field** determines which addressing mode is used for the source or destination of the data.

Each addressing mode corresponds to a relationship between the operand specifier (OprndSpec) and the operand (Oprnd). In the direct addressing mode, the operand specifier is the address in main memory of the operand. In mathematical notation, Oprnd = Mem[OprndSpec].



To execute a program, a group of instructions and data are loaded into main memory, and then the von Neumann execution cycle begins. The von Neumann execution cycle consists of the following steps: **(1) fetch the instruction specified by PC, (2) decode the instruction specifier, (3) increment PC, (4) execute the instruction fetched, and (5) repeat by going to Step 1.**

* + 1. **汇编语言**

An **assembler** is a program that translates a program in assembly language into the equivalent program in machine language. The von Neumann design principle calls for instructions as well as data to be stored in main memory. Corresponding to each of these bit sequences are two types of assembly language statements. For program statements, assembly language uses **mnemonics** in place of opcodes and register-r fields, **hexadecimal** instead of binary for the operand specifiers, and **mnemonic letters** for the addressing modes. For data statements, assembly language uses **pseudo-ops, also called dot commands**.

With **direct addressing**, the operand specifier is the address in main memory of the operand. But with immediate addressing, the operand specifier is the operand. In mathematical notation, Oprnd = OprndSpec. **Immediate addressing** is preferable to direct addressing because the operand does not need to be stored separately from the instruction. Such instructions execute faster because the operand is immediately available to the CPU from the instruction register.

Assembly language **symbols** eliminate the problem of manually determining the addresses of data and instructions in a program. **The value of a symbol is an address**. When the assembler detects a symbol definition, it stores the symbol and its value in a **symbol table**. When the symbol is used, the assembler substitutes its value in place of the symbol.

A **variable** at the high-order language level (Level HOL6) corresponds to a memory location at the assembly level (Level Asmb5). An **assignment statement** at Level HOL6 that assigns an expression to a variable translates to a load, followed by an expression evaluation, followed by a store at Level Asmb5. **Type compatibility** at Level HOL6 is enforced by the compiler with the help of its symbol table, which is more complex than the symbol table of an assembler. At Level Asmb5, the only type is bit, and any operation can be performed on any bit pattern.

### 问题求解

**Version1 (Machine Language)**: The two numbers to be added are 5 and 3. The program stores them at Mem[000D] and Mem[000F]. The first instruction loads the 5 into the accumulator, and then the second instruction adds the 3. At this point the sum is in the accumulator.

**Version2 (Assembly Language)**: Since **mnemonics** and **pseudo-ops** at Assembly Language level add convenience to programming, we want our v2 program to obtain the ability to add two given numbers without storing them in the first place, which means the program can take two integer values from input peripheral and calculate their sum.

We have three valuables, a, b, c, to store num1, num2 and their sum. The first two instructions read num1, num2 and store them in a, b and then the follow three instructions perform the addition and store the result in c. And finally, we output the sum.

### Pep/8程序

**Version1 (Machine Language):**

0000 C1000D ;A <- first number

0003 61000F ;Add the two numbers

0006 910011 ;Convert sum to character

0009 F1FC16 ;Output the character

000C 00 ;Stop

000D 0005 ;Decimal 5

000F 0003 ;Decimal 3

0011 0030 ;Mask for ASCII char

**Version2 (Assembly Language)**:

BR main ;branch to main

a: .WORD 0 ;define a to store the 1st num

b: .WORD 0 ;define b to store the 2nd num

c: .WORD 0 ;define c to store the sum

;

main: DECI a,d ;input a

DECI b,d ;input b

LDWA a,d ;A <- a

ADDA b,d ;perform a+b

STWA c,d ;store the sum in c

DECO c,d ;output the sum

STOP

.END

### 总结

* + 1. **遇到的问题与解决方案**

**Version1 (Machine Language)**: We want to output this result, but the only output instruction for this Level ISA3 machine is to store a byte in ASCII format to the output device at Mem[FC16]. The problem is that our result is 0000 1000 (bin). If the store byte instruction tries to output that, it will be interpreted as the backspace character.

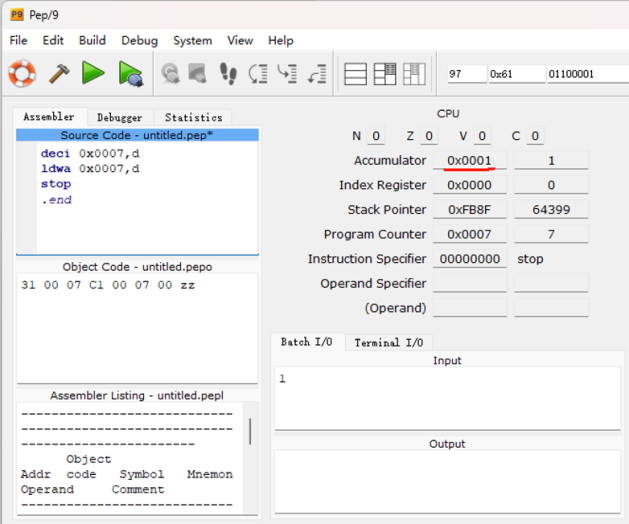
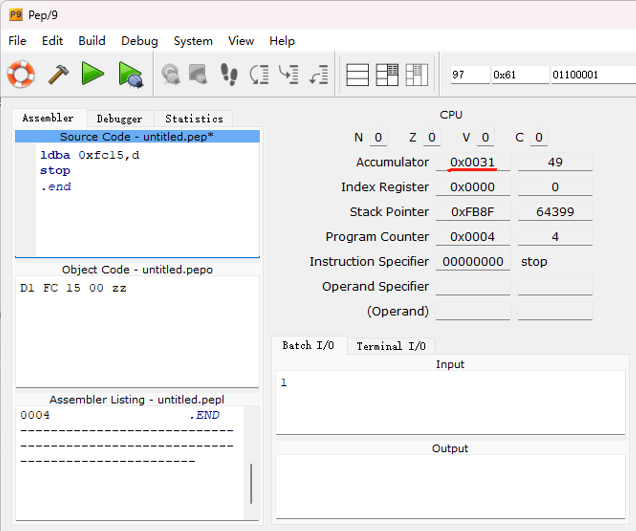
So, the program must convert the decimal number 8, 0000 1000 (bin), to the ASCII character 8, 0011 1000 (bin). The ASCII bits differ from the unsigned binary bits by the two extra 1’s in the third and fourth bits. To do the conversion, the program inserts those two extra 1’s into the result by **ORing** the accumulator with the **mask** 0000 0000 0011 0000.

The accumulator now contains the correct sum in ASCII form. The store byte instruction sends it to the output device.

**Version2 (Assembly Language)**: The program uses “**.WORD 0**” to allocate 2 bytes and initiate its value to 0x00 with a **symbol** denoting its address for the sake of usability and readability. And **the reason why we put the data part at the top of the program** is that if we don’t and we want to insert or delete an instruction, every single subsequent instruction that references a data by its address has to be modified to suit a new address, which is tedious and annoying.

However, note that in the von Neumann execution cycle, **PC** was initialized to 0x0000 before everything kicks off and it will make data mistaken for an instruction. So, we insert a “**BR main**” instruction at address 0x0000 to tell our computer to bypass the data part.

Below is a simple example that explains how Pep/9 inherently sees its input as a character



and it’s the same for the output. So, **traps** like “**DECI**” and “**DECO**” are wired into the CPU, allowing the CPU to handle exceptions or interrupts. No more masks are needed to convert a decimal number to its corresponding ASCII character.

### 参考

[1] Computer Systems, Fifth Edition. https://computersystemsbook.com/

（下载Pep/8虚拟机）

[2] J. 斯坦利·沃法德 - 计算机系统：核心概念及软硬件实现（原书第4版） (2015, 机械工业出版社)

（阅读相关章节）

[3] J. Stanley Warford - Computer Systems (2016, Jones \_ Bartlett Learning)

[4] Neso Academy - [Binary Subtraction using 2's Complement](https://www.youtube.com/watch?v=L_m7jBvtzpQ)

[5] CrashCourse - [How Computers Calculate - the ALU: Crash Course Computer Science #5](https://www.youtube.com/watch?v=1I5ZMmrOfnA&list=PL8dPuuaLjXtNlUrzyH5r6jN9ulIgZBpdo&index=6)