手把手教你构建 C 语言编译器（2）- 虚拟机

Table of Contents

这是“手把手教你构建 C 语言编译器”系列的第三篇，本章我们要构建一台虚拟的电脑，设计我们自己的指令集，运行我们的指令集，说得通俗一点就是自己实现一套汇编语言。它们将作为我们的编译器最终输出的目标代码。

手把手教你构建 C 语言编译器系列共有10个部分：

1. 手把手教你构建 C 语言编译器（0）——前言
2. 手把手教你构建 C 语言编译器（1）——设计
3. 手把手教你构建 C 语言编译器（2）——虚拟机
4. 手把手教你构建 C 语言编译器（3）——词法分析器
5. 手把手教你构建 C 语言编译器（4）——递归下降
6. 手把手教你构建 C 语言编译器（5）——变量定义
7. 手把手教你构建 C 语言编译器（6）——函数定义
8. 手把手教你构建 C 语言编译器（7）——语句
9. 手把手教你构建 C 语言编译器（8）——表达式

10. 手把手教你构建 C 语言编译器（9）——总结

计算机的内部工作原理

计算机中有三个基本部件需要我们关注：CPU、寄存器及内存。代码（汇编指令）以二进制的形式保存在内存中；CPU 从中一条条地加载指令执行；程序运行的状态保存在寄存器中。

内存

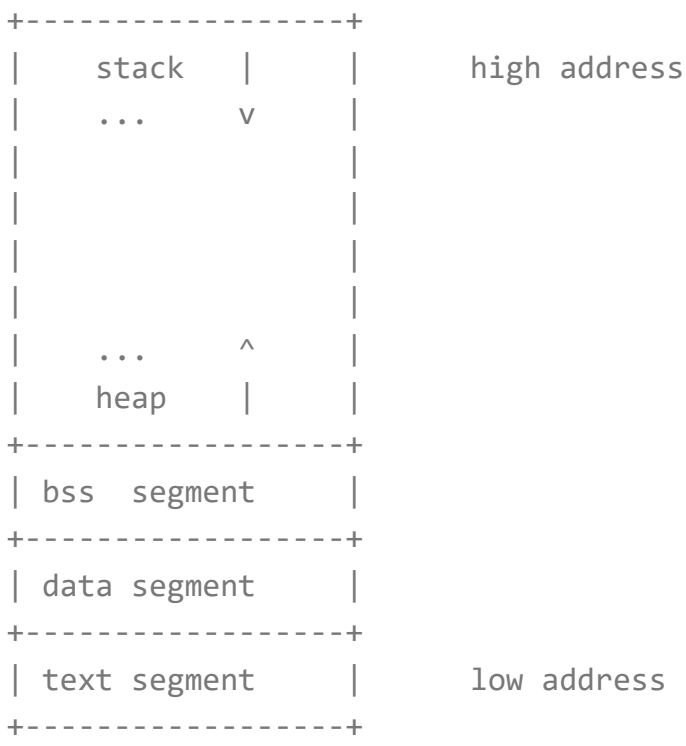
内存用于存储数据，这里的数据可以是代码，也可以是其它的数据。现代操作系统在操作内存时，并不是直接处理”物理内存“，而是操作”虚拟内存“。虚拟内存可以理解为一种映射，它的作用是屏蔽了物理的细节。例如 32 位的机器中，我们可以使用的内存地址为 `2^32 = 4G`，而电脑上的实际内存可能只有 `256 M`。操作系统将我们使用的虚拟地址映射到了到实际的内存上。

当然，我们这里并不需要了解太多，但需要了解的是：进程的内存会被分成几个段：

1. 代码段（text）用于存放代码（指令）。
2. 数据段（data）用于存放初始化了的数据，如 `int i = 10;`，就需要存放到数据段中。
3. 未初始化数据段（bss）用于存放未初始化的数据，如 `int i[1000];`，因为不关心其中的真正数值，所以单独存放可以节省空间，减少程序的体积。

4. 栈（stack）用于处理函数调用相关的数据，如调用帧（calling frame）或是函数的局部变量等。
5. 堆（heap）用于为程序动态分配内存。

   它们在内存中的位置类似于下图：



我们的虚拟机并不打算模拟完整的计算机，因此简单起见，我们只关心三个内容：代码段、数据段以及栈。其中的数据段我们只用来存放字符串，因为我们的编译器并不支持初始化变量，因此我们也不需要未初始化数据段。

当用户的程序需要分配内存时，理论上我们的虚拟机需要维护一个堆用于内存分配，但实际实现上较为复杂且与编译无关，故我们引入一个指令`MSET`，使我们能直接使用编译器（解释器）中的内存。

综上，我们需要首先在全局添加如下代码：

```
int *text,            // text segment
    *old_text,        // for dump text segment
    *stack;           // stack
char *data;           // data segment
```

注意这里的类型，虽然是`int`型，但理解起来应该作为无符号的整型，因为我们会在代码段（text）中存放如指针/内存地址的数据，它们就是无符号的。其中数据段（data）由于只存放字符串，所以是`char *`型的。

接着，在`main`函数中加入初始化代码，真正为其分配内存：

```
int main() {
    close(fd);
    ...

    // allocate memory for virtual machine
    if (!(text = old_text = malloc(poolsize))) {
        printf("could not malloc(%d) for text area\n", poolsize);
        return -1;
    }
    if (!(data = malloc(poolsize))) {
        printf("could not malloc(%d) for data area\n", poolsize);
        return -1;
    }
    if (!(stack = malloc(poolsize))) {
        printf("could not malloc(%d) for stack area\n", poolsize);
        return -1;
    }

    memset(text, 0, poolsize);
    memset(data, 0, poolsize);
    memset(stack, 0, poolsize);
```

```
    ...
    program();
}
```

寄存器

计算机中的寄存器用于存放计算机的运行状态，真正的计算机中有许多不同种类的寄存器，但我们的虚拟机中只使用 4 个寄存器，分别如下：

1. `PC` 程序计数器，它存放的是一个内存地址，该地址中存放着 **下一条** 要执行的计算机指令。
2. `SP` 指针寄存器，永远指向当前的栈顶。注意的是由于栈是位于高地址并向低地址增长的，所以入栈时 `SP` 的值减小。
3. `BP` 基址指针。也是用于指向栈的某些位置，在调用函数时会使用到它。
4. `AX` 通用寄存器，我们的虚拟机中，它用于存放一条指令执行后的结果。

要理解这些寄存器的作用，需要去理解程序运行中会有哪些状态。而这些寄存器只是用于保存这些状态的。

在全局中加入如下定义：

```
int *pc, *bp, *sp, ax, cycle; // virtual machine registers
```

在 `main` 函数中加入初始化代码，注意的是`PC`在初始应指向目标代码中的`main`函数，但我们还没有写任何编译相关的代码，因此先不处理。代码如下：

```
memset(stack, 0, poolsize);
...

bp = sp = (int *)((int)stack + poolsize);
ax = 0;

...
program();
```

与 CPU 相关的是指令集，我们将专门作为一个小节。

指令集

指令集是 CPU 能识别的命令的集合，也可以说是 CPU 能理解的语言。这里我们要为我们的虚拟机构建自己的指令集。它们基于 x86 的指令集，但更为简单。

首先在全局变量中加入一个枚举类型，这是我们要支持的全部指令：

```
// instructions
enum { LEA ,IMM ,JMP ,CALL,JZ  ,JNZ ,ENT ,ADJ ,LEV ,LI  ,LC  ,SI  ,SC
       OR  ,XOR ,AND ,EQ  ,NE  ,LT  ,GT  ,LE  ,GE  ,SHL ,SHR ,ADD ,SUB
       OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT };
```

这些指令的顺序安排是有意的，稍后你会看到，带有参数的指令在前，没有参数的指令在后。这种顺序的唯一作用就是在打印调试信息时更加方便。但我们讲解的顺序并不依据它。

MOV

`MOV` 是所有指令中最基础的一个，它用于将数据放进寄存器或内存地址，有点类似于 C 语言中的赋值语句。x86 的 `MOV` 指令有两个参数，分别是源地址和目标地址：`MOV dest, source`（Intel 风格），表示将 `source` 的内容放在 `dest` 中，它们可以是一个数、寄存器或是一个内存地址。

一方面，我们的虚拟机只有一个寄存器，另一方面，识别这些参数的类型（是数据还是地址）是比较困难的，因此我们将 `MOV` 指令拆分成 5 个指令，这些指令只接受一个参数，如下：

1. `IMM <num>` 将 `<num>` 放入寄存器 `ax` 中。
2. `LC` 将对应地址中的字符载入 `ax` 中，要求 `ax` 中存放地址。
3. `LI` 将对应地址中的整数载入 `ax` 中，要求 `ax` 中存放地址。
4. `SC` 将 `ax` 中的数据作为字符存放入地址中，要求栈顶存放地址。
5. `SI` 将 `ax` 中的数据作为整数存放入地址中，要求栈顶存放地址。

你可能会觉得将一个指令变成了许多指令，整个系统就变得复杂了，但实际情况并非如此。首先是 x86 的 `MOV` 指令其实有许多变种，根据类型的不同有 `MOVB`, `MOVW` 等指令，我们这里的 `LC/SC` 和 `LI/SI` 就是对应字符型和整型的存取操作。

但最为重要的是，通过将 `MOV` 指令拆分成这些指令，只有 `IMM` 需要有参数，且不需要判断类型，所以大大简化了实现的难度。

在 `eval()` 函数中加入下列代码：

```
void eval() {
    int op, *tmp;
    while (1) {
        if (op == IMM)       {ax = *pc++;}
        else if (op == LC)   {ax = *(char *)ax;}
        else if (op == LI)   {ax = *(int *)ax;}
        else if (op == SC)   {ax = *(char *)*sp++ = ax;}
        else if (op == SI)   {*(int *)*sp++ = ax;}
    }

    ...
    return 0;
}
```

其中的 `*sp++` 的作用是退栈，相当于 `POP` 操作。

这里要解释的一点是，为什么 `SI/SC` 指令中，地址存放在栈中，而 `LI/LC` 中，地址存放在 `ax` 中？原因是默认计算的结果是存放在 `ax` 中的，而地址通常是需要通过计算获得，所以执行 `LI/LC` 时直接从 `ax` 取值会更高效。另一点是我们的 `PUSH` 指令只能将 `ax` 的值放到栈上，而不能以值作为参数，详细见下文。

PUSH

在 x86 中， `PUSH` 的作用是将值或寄存器，而在我们的虚拟机中，它的作用是将 `ax` 的值放入栈中。这样做的主要原因是为了简化

虚拟机的实现，并且我们也只有一个寄存器 `ax` 。代码如下：

```
else if (op == PUSH) {*--sp = ax;}
```

JMP

`JMP <addr>` 是跳转指令，无条件地将当前的 `PC` 寄存器设置为指定的 `<addr>`，实现如下：

```
else if (op == JMP)  {pc = (int *)*pc;}
```

需要注意的是， `pc` 寄存器指向的是 **下一条** 指令。所以此时它存放的是 `JMP` 指令的参数，即 `<addr>` 的值。

JZ/JNZ

为了实现 `if` 语句，我们需要条件判断相关的指令。这里我们只实现两个最简单的条件判断，即结果（ `ax` ）为零或不为零情况下的跳转。

实现如下：

```
else if (op == JZ)   {pc = ax ? pc + 1 : (int *)*pc;}
else if (op == JNZ)  {pc = ax ? (int *)*pc : pc + 1;}
```

子函数调用

这是汇编中最难理解的部分，所以合在一起说，要引入的命令有 `CALL`，`ENT`，`ADJ` 及 `LEV`。

首先我们介绍 `CALL <addr>` 与 `RET` 指令，`CALL` 的作用是跳转到地址为 `<addr>` 的子函数，`RET` 则用于从子函数中返回。

为什么不能直接使用 `JMP` 指令呢？原因是当我们从子函数中返回时，程序需要回到跳转之前的地方继续运行，这就需要事先将这个位置信息存储起来。反过来，子函数要返回时，就需要获取并恢复这个信息。因此实际中我们将 `PC` 保存在栈中。如下：

```
else if (op == CALL) {*--sp = (int)(pc+1); pc = (int *)*pc;}
//else if (op == RET)  {pc = (int *)*sp++;}
```

这里我们把 `RET` 相关的内容注释了，是因为之后我们将用 `LEV` 指令来代替它。

在实际调用函数时，不仅要考虑函数的地址，还要考虑如何传递参数和如何返回结果。这里我们约定，如果子函数有返回结果，那么就在返回时保存在 `ax` 中，它可以是一个值，也可以是一个地址。那么参数的传递呢？

各种编程语言关于如何调用子函数有不同的约定，例如 C 语言的调用标准是：

1. 由调用者将参数入栈。
2. 调用结束时，由调用者将参数出栈。
3. 参数逆序入栈。

事先声明一下，我们的编译器参数是顺序入栈的，下面的例子（C 语言调用标准）取自 维基百科：

```
int callee(int, int, int);

int caller(void)
{
        int i, ret;

        ret = callee(1, 2, 3);
        ret += 5;
        return ret;
}
```

会生成如下的 x86 汇编代码：

```
caller:
        ; make new call frame
        push    ebp
        mov     ebp, esp
        sub     1, esp          ; save stack for variable: i
        ; push call arguments
        push    3
        push    2
        push    1
        ; call subroutine 'callee'
        call    callee
        ; remove arguments from frame
        add     esp, 12
        ; use subroutine result
        add     eax, 5
```

```
; restore old call frame
mov     esp, ebp
pop     ebp
; return
ret
```

上面这段代码在我们自己的虚拟机里会有几个问题：

1. `push ebp`，但我们的 `PUSH` 指令并无法指定寄存器。
2. `mov ebp, esp`，我们的 `MOV` 指令同样功能不足。
3. `add esp, 12`，也是一样的问题（尽管我们还没定义）。

也就是说由于我们的指令过于简单（如只能操作`ax`寄存器），所以用上面提到的指令，我们连函数调用都无法实现。而我们又不希望扩充现有指令的功能，因为这样实现起来就会变得复杂，因此我们采用的方法是增加指令集。毕竟我们不是真正的计算机，增加指令会消耗许多资源（钱）。

ENT

`ENT <size>` 指的是 `enter`，用于实现 'make new call frame' 的功能，即保存当前的栈指针，同时在栈上保留一定的空间，用以存放局部变量。对应的汇编代码为：

```
; make new call frame
push    ebp
mov     ebp, esp
       sub     1, esp       ; save stack for variable: i
```

实现如下：

```
else if (op == ENT)  {*--sp = (int)bp; bp = sp; sp = sp - *pc++;}
```

ADJ

`ADJ <size>` 用于实现 'remove arguments from frame'。在将调用子函数时压入栈中的数据清除，本质上是因为我们的 `ADD` 指令功能有限。对应的汇编代码为：

```
; remove arguments from frame
add     esp, 12
```

实现如下：

```
else if (op == ADJ)  {sp = sp + *pc++;}
```

LEV

本质上这个指令并不是必需的，只是我们的指令集中并没有 `POP` 指令。并且三条指令写来比较麻烦且浪费空间，所以用一个指令代替。对应的汇编指令为：

```
; restore old call frame
       mov     esp, ebp
pop     ebp
; return
ret
```

具体的实现如下：

```
else if (op == LEV)  {sp = bp; bp = (int *)*sp++; pc = (int *)*sp++;}
```

注意的是，`LEV` 已经把 `RET` 的功能包含了，所以我们不再需要 `RET` 指令。

LEA

上面的一些指令解决了调用帧的问题，但还有一个问题是如何在子函数中获得传入的参数。这里我们首先要了解的是当参数调用时，栈中的调用帧是什么样的。我们依旧用上面的例子（只是现在用“顺序”调用参数）：

```
sub_function(arg1, arg2, arg3);

|    ....       | high address
+---------------+
| arg: 1        |    new_bp + 4
+---------------+
| arg: 2        |    new_bp + 3
+---------------+
| arg: 3        |    new_bp + 2
+---------------+
|return address |    new_bp + 1
+---------------+
| old BP        | <- new BP
+---------------+
| local var 1   |    new_bp - 1
+---------------+
| local var 2   |    new_bp - 2
```

```
+---------------+
|    ....       |  low address
```

所以为了获取第一个参数，我们需要得到 `new_bp + 4`，但就如上面的说，我们的 `ADD` 指令无法操作除 `ax` 外的寄存器，所以我们提供了一个新的指令：`LEA <offset>`

实现如下：

```
else if (op == LEA)  {ax = (int)(bp + *pc++);}
```

以上就是我们为了实现函数调用需要的指令了。

运算符指令

我们为 C 语言中支持的运算符都提供对应汇编指令。每个运算符都是二元的，即有两个参数，第一个参数放在栈顶，第二个参数放在 `ax` 中。这个顺序要特别注意。因为像 `-`，`/` 之类的运算符是与参数顺序有关的。计算后会将栈顶的参数退栈，结果存放在寄存器 `ax` 中。因此计算结束后，两个参数都无法取得了（汇编的意义上，存在内存地址上就另当别论）。

实现如下：

```
else if (op == OR)  ax = *sp++ | ax;
else if (op == XOR) ax = *sp++ ^ ax;
else if (op == AND) ax = *sp++ & ax;
else if (op == EQ)  ax = *sp++ == ax;
else if (op == NE)  ax = *sp++ != ax;
```

```
else if (op == LT)  ax = *sp++ < ax;
else if (op == LE)  ax = *sp++ <= ax;
else if (op == GT)  ax = *sp++ >  ax;
else if (op == GE)  ax = *sp++ >= ax;
else if (op == SHL) ax = *sp++ << ax;
else if (op == SHR) ax = *sp++ >> ax;
else if (op == ADD) ax = *sp++ + ax;
else if (op == SUB) ax = *sp++ - ax;
else if (op == MUL) ax = *sp++ * ax;
else if (op == DIV) ax = *sp++ / ax;
else if (op == MOD) ax = *sp++ % ax;
```

内置函数

写的程序要"有用"，除了核心的逻辑外还需要输入输出，例如 C 语言中我们经常使用的 `printf` 函数就是用于输出。但是 `printf` 函数的实现本身就十分复杂，如果我们的编译器要达到自举，就势必要实现 `printf` 之类的函数，但它又与编译器没有太大的联系，因此我们继续实现新的指令，从虚拟机的角度予以支持。

编译器中我们需要用到的函数有：`exit`，`open`，`close`，`read`，`printf`，`malloc`，`memset` 及 `memcmp`。代码如下：

```
else if (op == EXIT) { printf("exit(%d)", *sp); return *sp;}
else if (op == OPEN) { ax = open((char *)sp[1], sp[0]); }
else if (op == CLOS) { ax = close(*sp);}
else if (op == READ) { ax = read(sp[2], (char *)sp[1], *sp); }
else if (op == PRTF) { tmp = sp + pc[1]; ax = printf((char *)tmp[-1], t
else if (op == MALC) { ax = (int)malloc(*sp);}
else if (op == MSET) { ax = (int)memset((char *)sp[2], sp[1], *sp);}
else if (op == MCMP) { ax = memcmp((char *)sp[2], (char *)sp[1], *sp);}
```

这里的原理是，我们的电脑上已经有了这些函数的实现，因此编译编译器时，这些函数的二进制代码就被编译进了我们的编译器，因此在我们的编译器/虚拟机上运行我们提供的这些指令时，这些函数就是可用的。换句话说就是不需要我们自己去实现了。

最后再加上一个错误判断：

```
else {
    printf("unknown instruction:%d\n", op);
    return -1;
}
```

测试

下面我们用我们的汇编写一小段程序，来计算 `10+20`，在 `main` 函数中加入下列代码：

```
int main(int argc, char *argv[])
{
    ax = 0;
    ...

    i = 0;
    text[i++] = IMM;
    text[i++] = 10;
    text[i++] = PUSH;
    text[i++] = IMM;
    text[i++] = 20;
    text[i++] = ADD;
    text[i++] = PUSH;
    text[i++] = EXIT;
    pc = text;
```

```
    ...
    program();
}
```

编译程序 `gcc xc-tutor.c`，运行程序：`./a.out hello.c`。输出

```
exit(30)
```

另外，我们的代码里有一些指针的强制转换，默认是 32 位的，因此在 64 位机器下，会出现 `segmentation fault`，解决方法（二选一）：

1. 编译时加上 `-m32` 参数：`gcc -m32 xc-tutor.c`
2. 在代码的开头，增加 `#define int long long`，`long long` 是 64 位的，不会出现强制转换后的问题。

注意我们的之前的程序需要指令一个源文件，只是现在还用不着，但从结果可以看出，我们的虚拟机还是工作良好的。

小结

本章中我们回顾了计算机的内部运行原理，并仿照 x86 汇编指令设计并实现了我们自己的指令集。希望通过本章的学习，你能对计算机程序的原理有一定的了解，同时能对汇编语言有一定的概念，因为汇编语言就是 C 编译器的输出。

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-1 https://github.com/lotabout/write-a-C-interpreter
```

实际计算机中，添加一个新的指令需要设计许多新的电路，会增加许多的成本，但我们的虚拟机中，新的指令几乎不消耗资源，因此我们可以利用这一点，用更多的指令来完成更多的功能，从而简化具体的实现。