

**基于8086的模拟器**

兰英杰 PB15061297

**基于8086的模拟器**

**前言：**

因为选了吴俊敏老师的《计算机系统详解》，而且又学了《微型计算机原理与接口技术》。所以为了加深自己对计算机的了解，便打算自己写一个简单的CPU模拟器出来。对于8086来说指令集不是太多，而且《微型计算机原理与接口技术》主要讲的是8086，所以便决定实现一下8086。

**关键词：**

1. 8086
2. 模拟器
3. Java
4. 指令

**摘要：**

采用Java语言来实现简易的8086 CPU，并简单实现一下内存，并为其他部件写出相应接口以方便程序拓展。

**目录：**

* 正文
* 前期准备∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙2
* ALU的实现∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙3
* Register的实现∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙4
* Memory的实现∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙5
* Execution的实现∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙5
* 整合∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙7
* 测试与运行∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙8
* 总结∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙10
* 补充∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙10
* 附录∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙11
* 参考文献∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙11

**正文：**

**一、前期准备**

前期主要是写出相应的接口，并且对指令集进行录入。而且对于程序做一些规定。

首先最为主要的是如何在模拟器中实现一个字节。由于8086采用的是小端序，所以我希望能保留它，另外考虑到在做计算的时候如果用纯小端序势必会有不必要的麻烦，因此便对小端序做了一点点修改，如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| value=0x1234 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 8086CPU | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 模拟器 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | | | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

即在小端序的情况下对所有字节再按位反序。

为了实现防存，需要有read和write，因此有：

**public** **interface** IOInterface {

**public** **boolean** write(**long** address, **byte**[] value);

**public** **byte**[] read(**long** address);

}

即所以I/O只需要这两个方法即可。同理，对于硬件接口，我们可以有：

**public** **interface** DriverInterface {

**public** **byte**[] read(**byte**[] port);

**public** **boolean** write(**byte**[] port, **byte**[] value);

}

当然，二者基本互通，不过为了更好的处理，便使用了数组作为port口。

下面是对指令的录入，指令是比较令人头疼的，是在是太多了，不过8086指令集还算规整，指令前缀也不是太多，所以通过观察不难发现，绝大部分指令都会有(D)、(W)、(MOD)、(REG)、(R/M)、(SEG）。另外，所以指令都有一个规整的格式：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令前缀（可选） | 操作码 | 操作数 | 数据（可选） |

所以为了能更为方便的获取指令的信息，我便借助xml进行标记：

<excutecode>

<PackageName>

<category name="ClassName">

<subcategory opcode="opcode" operand="operand">describe</subcategory>

</category>

</segmentoverride>

</excutecode>

PackageName对应指令所对应的包名，ClassName对应指令的名称，opcode对应操作码，operand对应指令的操作数。从而便可以对所有代码通过一个方法确定相关的信息，简化处理过程。

**二、ALU的实现**

在实现ALU时，为了简化指令处理，便顺便修改了FLAGS寄存器。另外为了节省时间，便对所有的计算均使用JVM的运算，大致过程如下：

**public** **static** **void** OP(**byte**[] des, **byte**[] src) **throws** LengthNotMatchException {

**if** (des.length != src.length || des.length != LENGTH) {

**throw** **new** LengthNotMatchException("add16:" + des.length + ", " + src.length);

}

**long** a = MathUtils.*byteArrayToLong*(des, **false**, LENGTH);

**long** b = MathUtils.*byteArrayToLong*(src, **false**, LENGTH);

**byte**[] ret = MathUtils.*longToByteArray*(a OP b, **false**, LENGTH);

*setFlags*(des, src, ret);

MathUtils.*copyArray*(des, ret);

}

从而将指令集中的Logical Operation和Arithmetic Operation的操作全部进行实现。

结合《计算机系统详解上》对于数据存储的讲解，便可以做到对字位与对应数据的互转：

**public** **static** **byte**[] longToByteArray(**long** value, **boolean** signed, **int** length) {

**byte**[] b = **new** **byte**[64];

**for**(**int** i = 0;i < length;i++)

b[i] = (**byte**) ((value >> i) & 1);

**if**(signed)

**for**(**int** i = length;i < 64;i++)

b[i] = b[length - 1];

**return** b;

}

**public** **static** **long** byteArrayToLong(**byte**[] value,**boolean** signed,**int** length) {

**long** p = 0, t = 0;

**for**(**int** i = 0;i < length;i++) {

t = value[i];

p |= t << i;

}

**if**(signed)

**for**(**int** i = length;i < 64;i++)

p |= t << i;

**return** p;

}

至此，ALU基本写好。

**三、Register的实现**

通由于8086有14个寄存器，对每一个都编写相应的方法太过于繁琐，而且在访问的时候也会有很多问题，因此便抽象出一个基本的寄存器：

**public** **abstract** **class** BaseRegister {

**private** **final** **byte**[] DATA = **new** **byte**[Configer.*getRegisterSize*()];

**private** **static** **final** **byte**[] ***ONE*** = **new** **byte**[Configer.*getRegisterSize*()];

**static** {

***ONE***[0] = 1;

}

**public** **final** **void** setDATA(**byte**[] data) **throws** CopyArrayException;

**public** **final** **void** setHIGH(**byte**[] data) **throws** CopyArrayException;

**public** **final** **void** setLOW(**byte**[] data) **throws** CopyArrayException;

**public** **final** **byte**[] getDATA();

**public** **final** **byte**[] getHIGH();

**public** **final** **byte**[] getLOW();

**public** **final** **byte** indexOf(**int** index);

@Override

**public** String toString();

**public** **final** **void** dec();

**public** **final** **void** inc();

**public** **final** **boolean** isZero();

**public** **static** **void** sub16(**byte**[] des, **byte**[] src);

**public** **static** **void** add16(**byte**[] des, **byte**[] src);

}

从而所有寄存器便不用一一编写相应的处理逻辑。

对于FLAGS与IP这两个比较特殊的寄存器，只好分开对待。主要是IP一直在更新，FLAGS不适合上述访问，所以在设计IP的时候用Long类型的数据直接存储，在获得IP的数组类型时才进行转换。对于FLAGS内部定义出相应标志位的位置，以及提供setter与getter方法。

在访问寄存器的时候，注意到指令中会通过(MOD)、(R/M)、(W)、(SEG)等作为参数，因此便编写了一个管理者：

**public** **class** RegisterMgr {

**public** **static** **final** **void** setDATA(String REG, **boolean** W, **byte**[] value) **throws** Exception;

**public** **static** **final** **byte**[] getDATA(String REG, **boolean** W) **throws** Exception;

}

从而方便了指令中访问寄存器的步骤。

至此，寄存器部分基本结束。

**四、Memory的实现**

对于Memory这一块，为了实现与内部数据格式匹配，所有I/O必须遵从模拟器内部数据表示的格式要求。当然，必须实现读写操作方法。

Memory在实现的时候只是用了一个很大的字符串，现实中的内存也可以看做是一个巨大的字符串。

当然，为了提高适配性，对于内存又提供了管理者，也提高了可扩展能力。

至此，内存实现基本结束。

**五、Execution的实现**

对于Execution的实现真的是太麻烦了，8086指令虽然比较规整，但是还是相当多。不过借助于指令集表(xml)，便可以省掉一大部分功夫去解析出指令的信息：

**public** **class** Execution {

**private** **static** **final** String ***REGEX*** = "\\(([A-Z\_a-z\_0-9\_/]\*)\\)\*";

**private** **static** **final** Pattern ***PATTERN*** = Pattern.*compile*(***REGEX***);

**private** String opcode;

**private** String operand;

**private** String describle;

**private** **int** index;

**public** List<String> operandName = **new** ArrayList<>();

**public** List<String> operandValue = **new** ArrayList<>();

**public** Execution(String opcode, String operand, String describle, **int** index);

**private** **void** init();

**public** **boolean** match(String c1, String c2) **throws** Exception;

**public** String getOperand(**int** index);

**public** String getOperand(String index);

**public** String getDescrible();

**public** **int** getIndex();

**public** **void** exec() **throws** Exception;

@Override

**public** String toString();

**public** **static** **boolean** match(String c1, String c2, Execution exc) **throws** Exception;

**public** **static** **byte**[] arrayConcat(**byte**[] low,**byte**[] high);

**public** **static** **void** arraySplit(**byte**[] high,**byte**[] low,**byte**[] src);

**public** **static** **byte**[] longToByteArray(**long** value, **boolean** signed, **int** length);

**public** **static** **long** byteArrayToLong(**byte**[] value,**boolean** signed,**int** length);

**public** **static** **byte**[] add16(**byte**[] des, **byte**[] src);

**public** **static** **byte**[] add8(**byte**[] des, **byte**[] src);

**public** **static** **byte**[] sub16(**byte**[] des, **byte**[] src);

**public** **static** **byte**[] sub8(**byte**[] des, **byte**[] src);

}

从而对所有指令公平看待，通过解析xml获得里面的(D)、(W)、(MOD)、(REG)、(R/M)、(SEG）信息。建立起对应的指令信息如下：

{

String packageName = "PackageName";

String usingPackage = "cn.jay.computer.exec." + packageName + “.”;

Elements es = *document*.getElementsByTag(packageName);

Elements cs = es.get(0).children();

**for** (**int** i = 0; i < cs.size(); i++) {

String name = cs.get(i).attr("name");

Elements ss = cs.get(i).children();

**for** (**int** j = 0; j < ss.size(); j++) {

Execution ec = **null**;

**try** {

ec = (Execution) Class.*forName*(usingPackage + name).getConstructors()[0].newInstance(ss.get(j).attr("opcode"), ss.get(j).attr("operand"),ss.get(j).text(), j);

} **catch** (Exception e) {

}

**if** (ec != **null**)

*addExecuter*(name, ec);

}

}

}

另外对于寻址，大部分指令都会通过(W)、(MOD)、(R/M)进行寻址，因此便有：

**public** **class** RM\_MOD\_Analyzer {

**public** **static** **byte**[] analyze(String MOD,String RM,**boolean** W);

**private** **static** **byte**[] RM\_Analyze(String RM);

**public** **static** **byte**[] add(**byte**[] right, **byte**[] left,**int** length);

**public** **static** **byte**[] arrayConcat(**byte**[] low,**byte**[] high);

}

调用后会根据(MOD)、(RM)、(W)获得相应的地址。当然，如果MOD == “11”则返回null，以表示访问的是寄存器。

对于指令执行的时候，会传入两个参数(c1,c2)，代表指令的前两个字节。正是由于8086所有指令操作码都不会超过两个字节，所以解析变得比较轻松。首先根据xml中的opcode进行匹配对于的操作指令，然后根据xml中的operand去解析operand中对应的指令信息并存储。然后根据操作信息去解析相应的功能，至此，一条指令执行结束。

如：MOV CL,[BX + 1234H]，机器码为：8A 8F 34 12

首先，由xml中的opcode去匹配8A 8F，发现在MOV中的

<subcategory opcode="100010\*\*\*\*\*\*\*\*\*\*" operand="(D)(W)(MOD)(REG)(R/M)">MOV reg/mem to/from reg</subcategory>

与之对应。然后根据operand中的信息进行匹配，将(D = 1)(W = 0)(MOD = 10)(REG = 001)(R/M = 111)存储起来。然后在执行的时候，根据上述信息，知道是双字节指令，会自动读取一条指令，更改IP位置。然后调用RM\_MOD\_Analyzer去获得寻址地址，返回(address=[BX] + D16)(D16为16位立即数，内部会自动更新IP)，从而获得EA，然后读出数据并将结果传送至对于的寄存器(CL)。

**六、整合**

在此阶段主要是对上述部件进行整合起来。根据CPU分为BIU和EU，我们也进行相应的分类，BIU用于取指，EU用于执行。

考虑到指令执行需要一至两个字节，因此传参的时候为：

(BIU.getInstruction(),BIU.peekInstruction());

对应所有的错误，在指令执行阶段都进行抛出，统一在ExecuterMgr中进行捕获。以便后续处理。

至此，8086模拟器基本组装成型。不过时钟模块还不知道如何设计，现在仅仅是在每一段时间后去自动执行后续代码。

**七、测试运行**

测试的时候采用的是Dos6.22-5.25.img，初始化CS=0，IP=0,。将代码load进内存中，以0为起始位置，然后start()。例如运行如下代码(CS:IP=0000：0200H，使用TASM编译，CS:IP经查看二进制代码后得到)：

CODE SEGMENT

ASSUME CS:CODE

START:

MOV AX,12

MOV CX,12

CALL FARB

HLT

;DL为入口参数

SHOW PROC

PUSH DX

PUSH AX

MOV AX,DX

MOV DX,0FFFFH

OUT DX,AL

POP AX

POP DX

RET

ENDP

FARB PROC

PUSH AX

PUSH BX

PUSH CX

PUSH DX

MOV CX,4

MOV AX,1

MOV BX,1

PUSH BX

PUSH AX

MOV DX,AX

CALL SHOW

REP\_S:

POP AX

POP BX

MOV DX,AX

CALL SHOW

ADD AX,BX

PUSH DX

PUSH AX

LOOP REP\_S

POP CX

POP CX

POP DX

POP CX

POP BX

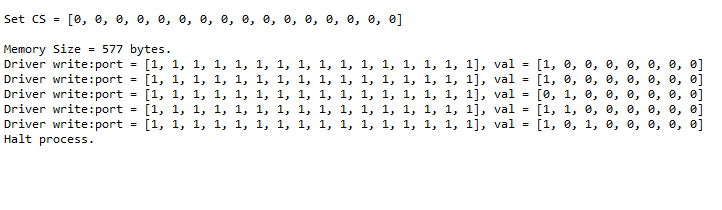
POP AX

RET

ENDP

CODE ENDS

END START

结果如下：

与预期符合。

**总结：**

从运行结果来看，还是比较令人满意的，可能内部还会有诸多Bug，但是至少对所给示例运行成功了。

但是由于在刚开始设计的时候由于设计缺陷，导致好多工作都要一一重复。最典型的的就是没有做到数据的存取模块化，所有的数据存取都是在每条指令中进行。导致一处出错处处错的情况。而且在设计ALU的时候没有考虑调用的简单易行性，把字操作与字节操作写成两个方法来调用，导致内部处理的时候无法根据(W)信息直接调用。明显与内存读取、寄存器访问过程脱节，因此有需要进行重写。

另外在时钟设计上也没有进行完全按照8086的时序进行，每条基本指令基本都在一个周期内执行结束。中断调用等CPU控制指令也没有进行相应的编写。还有预期是各个部件通过TCP与CPU进行交互，不过这个工作量过大，而且需要更为详细的8086CPU以及外设资料，所以便没有进行相应编写。

**补充：**

目前对于该模拟器还有好多地方需要补充，而且还有好多地方需要重构甚至是重写，特别是没有完全做到OOP，好多工作分工不明确，耦合度很高！所以一时难以全部完成。在编写该模拟器的时候出现过许许多多的问题，比如在刚开始写的时候，所有指令执行时的异常均被我在指令执行代码出捕获了，而且也没有去打印输出，导致在初运行的时候好多错误都没有显示，只好在更改指令执行的代码，将异常一一抛出。另外，对于测试样例，所调用的指令很少，所以还有相当大的一部分代码没有进行测试，所以无法确定该模拟器是否能正常运行。

不过总体来说，我的目标是完全用Java语言实现出一个8086CPU，封装出同样的40个引脚，以便能与现有硬件或者是另外编写的虚拟硬件互联。从而更好的理解8086CPU的内部原理。

**附录：**

所有代码均已上传至<https://github.com/lanyj/8086CPU>上，后续将将逐步修复内部的Bug并进行相应拓展。

**参考文献：**

* 《深入理解计算机系统》ISBN 978-7-111-54493-7
* 《微型计算机原理与接口技术》ISBN 978-7-312-03191-5
* 《8086 16-BIT HMOS MICROPROCESSOR》<http://studylib.net/doc/8824721>