

计算机组成原理课程设计-虚拟机设计报告

姓名：董岩

班级：2016211310

学号：2016211225

时间：2018年5月22日

目录

计算机组成原理课程设计-虚拟机设计报告

目录

一、基本信息

1. 实验目的
2. 虚拟机基本信息

二、组成部件

三、处理器架构

1. 通用寄存器与寻址方式
 - i. 寄存器列表:
 - ii. 寻址方式:
2. 特殊功能寄存器
3. 机器状态
4. 指令类型及格式
5. 取指流程
6. 指令列表
 - i. 特殊指令
 - ii. 传送指令
 - iii. 双操作数算术与逻辑运算指令
 - iv. 单操作数算术与逻辑运算指令
 - v. 测试指令
 - vi. 跳转指令
 - vii. 堆栈操作
 - viii. 中断操作
7. 中断
 - i. 中断向量
 - ii. 中断操作的实现

四、虚拟存储系统

1. 预留空间
2. 用户代码区
3. 栈区
4. 用户数据区
5. 输入缓冲区
6. 输出缓冲区
7. 显示缓冲区

五、汇编语言

1. 语法单元
2. 语法要求
3. 函数调用
 - i. 寄存器内容的保存
 - ii. 函数参数
 - iii. 返回值
4. 库函数

六、虚拟机实现框架

1. 程序结构
2. 虚拟存储系统程序框架
 - i. AbstractFile接口
 - ii. ReadableFile接口
 - iii. WritableFile接口
 - iv. Memory类

v. Keyboard类

vi. TextOutput类

vii. Display类

viii. IOBridge类

3. 处理器程序框架

4. 主类框架

七、使用

1. 文件目录

2. 汇编器的使用

3. 虚拟机的使用

八、测试

1. 求素数程序

2. 生命游戏

九、问题与后续计划

1. 总结

2. 当前问题

3. 后续计划

4. 展望

一、基本信息

1. 实验目的

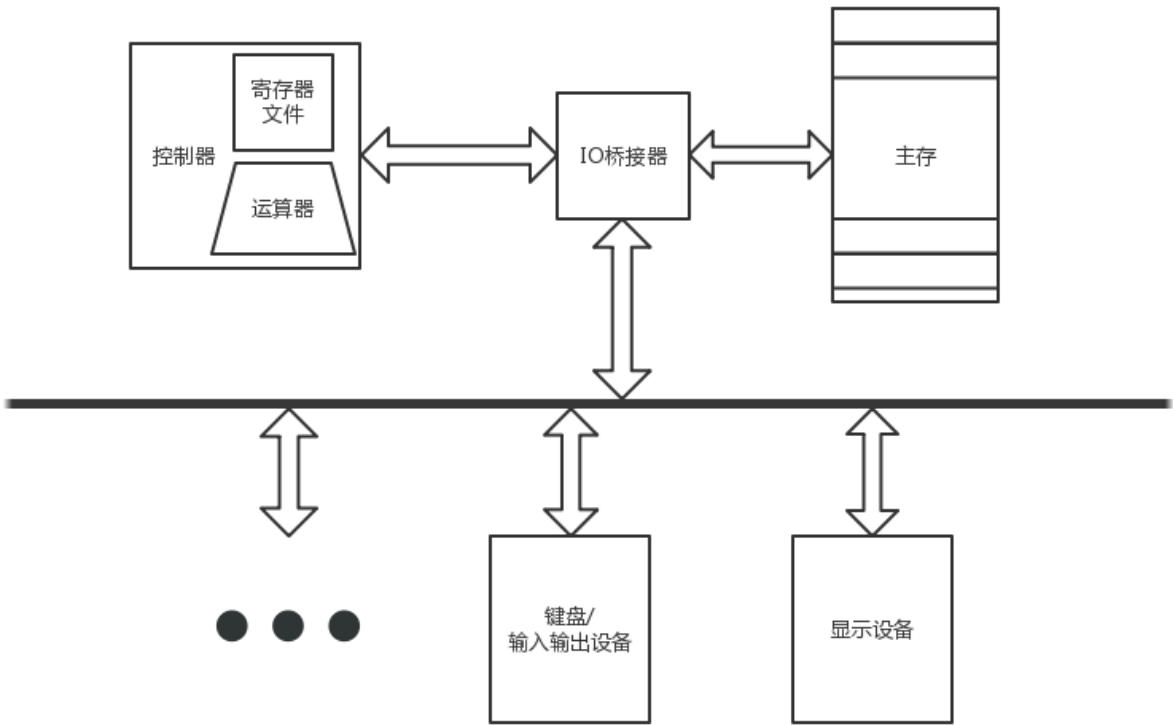
此虚拟机是计算机组成原理课程设计的一项重要内容，目的是通过高级语言实现软件，模拟冯诺依曼结构计算机系统及其工作原理，直观地显示机器运行的过程，综合运用软硬件知识，提升自身对计算机的认知水平。

2. 虚拟机基本信息

本次实验使用高级语言构建虚拟机，模拟冯诺依曼结构计算机的运行。此虚拟机使用Java 8(Intellij IDEA) 开发，测试环境为Windows 10 64bit，拥有文本和图形用户界面。存储系统包含寄存器与主存以及文件设备，不设缓存。指令系统源自Y86-64架构，目前仅支持有符号64位整型的算数与逻辑运算。

二、组成部件

结构示意图:



- **控制器**：通过函数来控制各部件的运行，模拟控制器的工作。
- **运算器**：用高级语言的计算功能模拟运算器的工作。
- **寄存器**：15个通用寄存器，3个专用寄存器。通用寄存器使用长度为15的64位整型向量来模拟；专用寄存器用成员变量保存。
- **IO桥接器**：提供虚拟地址映射。
- **存储器**：用字节数组模拟。
- **键盘输入部件**：用数组模拟输入缓冲区，与存储器共用虚拟地址编码。
- **文本输出部件**：用数组模拟输出缓冲区，与存储器共用虚拟地址编码。

- **显示设备**：用数组模拟显示缓冲区，与存储器共用虚拟地址编码
- **总线**：体现为控制函数中的参数和局部变量，起到传输数据的作用。

外观/设计图:



各部分说明:

- 左上角为**显示屏**，分辨率为200 X 125，默认黑色背景。支持RGBA颜色空间，对Alpha通道尚未提供完全的支持。
- 右上角为**处理器信息**，实时显示15个寄存器保存的数值以及程序计数器中保存的值。其下方6个按钮分别是：
 - 加载（Load）：从文件加载可执行目标文件；
 - 慢速运行（Run）：以约1000Hz的速率运行程序；
 - 快速运行（Fast）：以约15MHz的速率运行程序；
 - 暂停（Pause）：暂停运行；
 - 单步运行（Step）：单步运行程序；
 - 停机（Halt）：终止程序运行，并将程序状态恢复为刚载入时的状态。
- 左下角为**输入和输出窗口**，左侧为输入窗口，目前尚未实现；右侧为输出窗口，显示来自处理器的输出，不能手动修改。
- 右下角为**内存监视器**，最多支持10个条目的检测，每个条目中左侧是地址输入框，可以读取10进制或16进制整数，输入内存地址后右侧文本框会实时显示以输入地址为起始16字节的数据。

效果图:



三、处理器架构

DY64：源自Y86-64 ISA（CS:APP第四章所描述的指令集架构），在其基础上做了一些修改和扩充。

1. 通用寄存器与寻址方式

i. 寄存器列表：

编号	名称	功能
0	%rax	函数返回值
1	%rcx	第四个参数
2	%rdx	第三个参数
3	%rbx	被调用者保存
4	%rsp	栈指针寄存器
5	%rbp	被调用者保存
6	%rsi	第二个参数
7	%rdi	第一个参数
8	%r8	第五个参数
9	%r9	第六个参数
A	%r10	调用者保存
B	%r11	调用者保存
C	%r12	被调用者保存
D	%r13	被调用者保存
E	%r14	被调用者保存

ii. 寻址方式：

R 代表寄存器文件，M 代表存储器。

类型	格式	操作数值	名称
立即数	\$Imm	Imm	立即数寻址
寄存器	r_a	R[r_a]	寄存器寻址
存储器	Imm()	M[Imm]	绝对寻址
存储器	(r_b)	M[R[r_b]]	间接寻址
存储器	Imm(r_b)	M[Imm+R[r_b]]	基址+偏移量寻址

2. 特殊功能寄存器

- 程序计数器 (PC)：记录下一条取指地址
- 指令寄存器 (IR)：记录当前指令

- 条件码寄存器 (CC)：记录当前条件码 (溢出, 负数, 零, 中断等)
- 机器状态 (State)：记录当前机器状态 (正常, 各种类型异常)

3. 机器状态

机器状态 (State) 有如下类型：

no	标识符	含义
0	VM_OK	正常状态
1	VM_HLT	遇到停机指令
2	VM_INS	非法指令
3	VM_REG	非法寄存器表示
4	VM_ADR	非法内存引用
5	VM_LOG	逻辑错误
6	VM_PC	指令地址错误

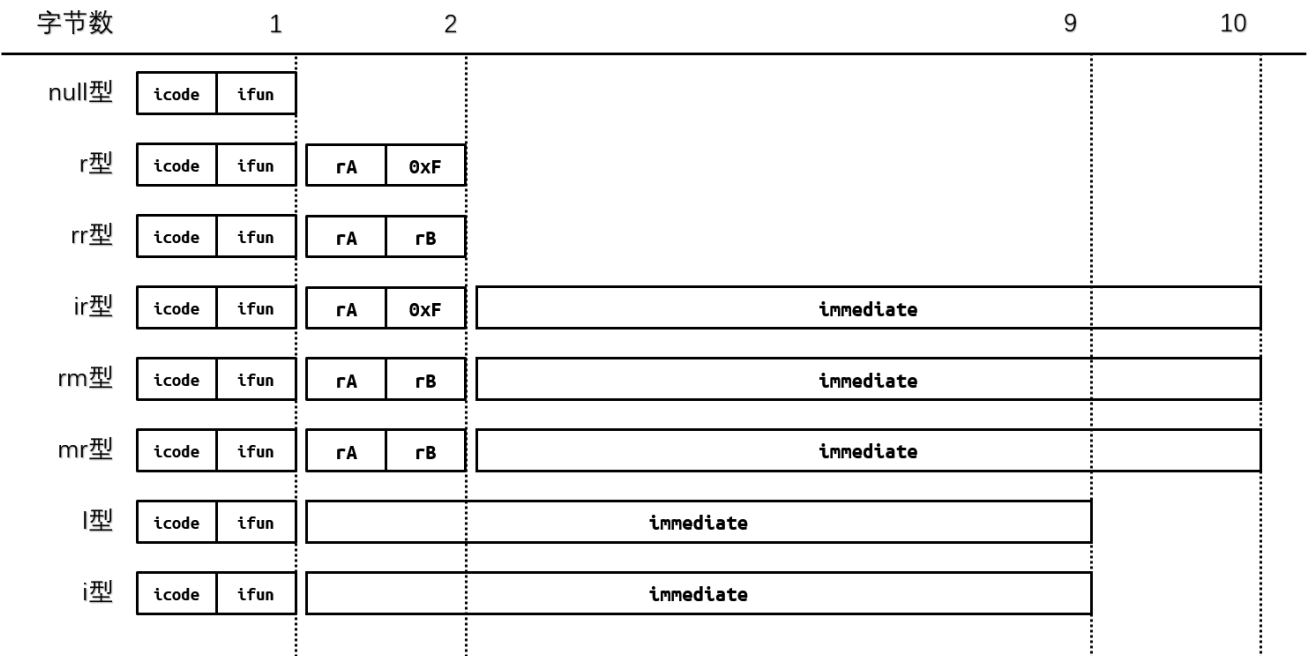
在目前的设计中，当程序运行出现异常后，处理器会直接终止运行，并将异常信息报告给外界。

4. 指令类型及格式

所有指令的第一个字节均为操作码，其中高4位 (icode) 代表指令种类，低4位 (ifun) 代表指令功能。所有包含寄存器ID的字节中，高4位代表第一个寄存器，低4位代表第二个寄存器（0xF 代表不选中寄存器）。立即数均为64位，采用小端法存储。

目前共有7种类型的指令：

1. **null型**：无操作数，长度为一个字节。
2. **r型**：单操作数，操作数源为寄存器，长度为2字节。第2个字节存放寄存器id。
3. **rr型**：双操作数，操作数源均为寄存器，长度为2字节。第2个字节存放寄存器id。
4. **ir型**：双操作数，第一个操作数源为立即数，第二个操作数源为寄存器，长度为10字节。第2个字节存放寄存器id，第3-10字节存放立即数。
5. **rm型**：双操作数，第一个操作数源为寄存器，第二个操作数源为存储器的某个位置，长度为10字节。第2个字节存放寄存器id，第3-10字节存放立即数。
6. **mr型**：双操作数，第一个操作数源为存储器的某个位置，第二个操作数源为立即数，长度为10字节。第2个字节存放寄存器id，第3-10字节存放立即数。
7. **l型**：单操作数，在汇编指令该操作数中是代表指令地址的标签，在机器指令中是指令地址，长度为9字节。第2-9字节存放立即数。编写汇编代码时操作数为标签，在汇编时标签会被翻译成立即数。
8. **i型**：单操作数，操作数为立即数。第2-9字节存放立即数。



5. 取指流程

每次取指时，处理器会先加载mem[PC]位置的第一个字节，根据此字节的内容判断指令类型 and 长度，再依据判断结果读取相应字节数的指令，将取出的指令存放在ir（指令寄存器）中。处理器执行指令时会根据ir中的数据进行操作。

6. 指令列表

下列符号在指令中分别代表：

- `rA`, `rB`：寄存器ID
- `R`：寄存器，`R[rA]`表示寄存器rA中的值
- `m`：虚拟存储地址，寻址方式如上所述
- `M`：虚拟存储系统，可看作字节数组，`M[m]`表示地址m处的值
- `I`：立即数
- `Label`：标签，汇编器会将其翻译为立即数

i. 特殊指令

op码	指令名称与格式	功能
00	halt	停机
01	nop	空操作
02	ret	函数返回 <code>%rsp += 8, R[%rsp] -> pc</code>
03	iret	中断返回

ii. 传送指令

op 码	指令名称与格式	功能
10	irmovl i, rA	将立即数传送到寄存器 $I \rightarrow R[rA]$
20	rmmovq rA, m	将寄存器中64位数据传送到主存 $R[rA] \rightarrow M[m]$
21	rmmovl rA, m	将寄存器中低32位数据传送到主存 $R[rA] \& 0xffffffff \rightarrow M[m]$
22	rmmovw rA, m	将寄存器中低16位数据传送到主存 $R[rA] \& 0xffff \rightarrow M[m]$
23	rmmovb rA, m	将寄存器中低8位数据传送到主存 $R[rA] \& 0xff \rightarrow M[m]$
24	mrmmovq rA, m	将64位数据从主存传送到寄存器 $M[m] \rightarrow R[rA]$
25	mrmmovl rA, m	将32位数据从主存传送到寄存器 $M[m] \& 0xffffffff \rightarrow R[rA]$
26	mrmmovw rA, m	将16位数据从主存传送到寄存器 $M[m] \& 0xffff \rightarrow R[rA]$
27	mrmmovb rA, m	将8位数据从主存传送到寄存器 $M[m] \& 0xff \rightarrow R[rA]$
30	rrmov rA, rB	将数据从寄存器rA传送到寄存器rB $R[rA] \rightarrow R[rB]$
31	cmovl	条件传送, 当状态码为相等 (e) 时发生传送 $\text{if equal then } R[rA] \rightarrow R[rB]$
32	cmovne	状态码为不相等 (ne) 时发生传送 $\text{if not equal then } R[rA] \rightarrow R[rB]$
33	cmovg	状态码为大于 (g) 时发生传送 $\text{if greater then } R[rA] \rightarrow R[rB]$
34	cmovge	状态码为大于等于 (g) 时发生传送 $\text{if greater or equal then } R[rA] \rightarrow R[rB]$
35	cmovl	状态码为小于 (l) 时发生传送 $\text{if less then } R[rA] \rightarrow R[rB]$
36	cmovle	状态码为小于等于 (g) 时发生传送 $\text{if less or equal equal then } R[rA] \rightarrow R[rB]$

iii. 双操作数算术与逻辑运算指令

更新条件码

op 码	指令名称与格式	功能
40	add rA, rB	$R[rA] + R[rB] \rightarrow R[rA]$
41	sub rA, rB	$R[rA] - R[rB] \rightarrow R[rA]$
42	and rA, rB	$R[rA] \& R[rB] \rightarrow R[rA]$
43	or rA, rB	$R[rA] R[rB] \rightarrow R[rA]$
44	xor rA, rB	$R[rA] \wedge R[rB] \rightarrow R[rA]$
45	sar rA, rB	$R[rA] \ll (R[rB] \& 0x3f) \rightarrow R[rA]$ 左移量为R[rB]的低6位
46	sar rA, rB	$R[rA] \gg (R[rB] \& 0x3f) \rightarrow R[rA]$ 算术右移, 填充符号位
47	shr rA, rB	$R[rA] \gg (R[rB] \& 0x3f) \rightarrow R[rA]$ 逻辑右移, 填充0
48	mul rA, rB	$R[rA] * R[rB] \rightarrow R[rA]$
49	idiv rA, rB	$R[rA] / R[rB] \rightarrow R[rA], R[rA] \% R[rB] \rightarrow R[\%rax]$ 有符号数整除, 商存放在R[rA]中, 余数存放在R[%rax]中 (注: 如果 $rA == \%rax$, 余数会被抛弃, 不保存)

iv. 单操作数算术与逻辑运算指令

更新条件码

op码	指令名称与格式	功能
50	not rA	$\sim(R[rA]) \rightarrow R[rA]$
51	neg rA	$-(R[rA]) \rightarrow R[rA]$
52	inc rA	$R[rA]++$
53	dec rA	$R[rA]--$
54	cltq rA	对R[rA]的低32位进行符号拓展 $(int64)(int32)R[rA] \rightarrow R[rA]$
55	cwtq rA	对R[rA]的低16位进行符号拓展 $(int64)(int16)R[rA] \rightarrow R[rA]$
56	cbtq rA	对R[rA]的低8位进行符号拓展 $(int64)(int8)R[rA] \rightarrow R[rA]$
57	cqtl rA	保留R[rA]的低32位，其余位清零 $R[rA] \&= 0xffffffff$
58	cqtw rA	保留R[rA]的低16位，其余位清零 $R[rA] \&= 0xffff$
59	cqtb rA	保留R[rA]的低8位，其余位清零 $R[rA] \&= 0xff$

v. 测试指令

更新条件码

op码	指令名称与格式	功能
60	cmp rA, rB	将R[rA]与R[rB]比较大小 $\text{update cc with } R[rA] - R[rB]$
61	test rA, rB	将(R[rA] & R[rB])和0比较大小 $\text{update cc with } R[rA] \& R[rB]$

vi. 跳转指令

op 码	指令名称与格式	功能
70	jmp label	无条件跳转到label处 <code>label -> pc</code>
71	je label	比较结果为相等时跳转到label处 <code>if equal then label -> pc</code>
72	jne label	结果为不相等时跳转到label处 <code>if not equal then label -> pc</code>
73	jg label	结果为大于时跳转到label处 <code>if greater then label -> pc</code>
74	jge label	结果为大于等于时跳转到label处 <code>if greater or equal then label -> pc</code>
75	jl label	结果为小于时跳转到label处 <code>if less then label -> pc</code>
76	jle label	结果为小于等于时跳转到label处 <code>if less or equal then label -> pc</code>
77	call label(function name)	函数调用 <code>%rsp -= 8, pc -> R[%rsp], label -> pc</code>

vii. 堆栈操作

op 码	指令名称与格式	功能
80	push rA	将R[rA]压进栈中 <code>%rsp -= 8, R[rA] -> R[%rsp]</code>
81	pop rA	弹出栈顶并保存至R[rA] <code>%rsp += 8, R[%rsp-8] -> R[rA]</code>

viii. 中断操作

op 码	指令名称与格式	功能
90	int i	触发软中断（i为立即数，中断向量下标）

7. 中断

i. 中断向量

目前的中断向量中只有两个有效地址：

- 地址2：清空输出缓冲区
- 地址3：将显存的内容显示到屏幕上

ii. 中断操作的实现

当前版本的虚拟机没有真实地模拟中断操作，软件触发的陷阱（软中断）和普通指令的运行方式相同，没有设置中断标志位。此外，中断程序使用Java编写，而不是汇编语言，因为当前的指令集无法完备地支持虚拟机的所有操作。

四、虚拟存储系统

地址	分区
0x00000000 ~ 0x00ffffff	预留空间
0x01000000 ~ 0x01ffffff	用户代码区
0x02000000 ~ 0x02ffffff	栈区（读写）
0x03000000 ~ 0x0ffffff	用户数据区（读写）
0x10000000 ~ 0x1007ffff	输入缓冲区（只读）
0x10080000 ~ 0x100ffff	输出缓冲区（只写）
0x10100000 ~ 0x10200000	显示缓冲区（只写）



1. 预留空间

此部分空间为虚拟机预留的存储空间，存放了与IO和系统功能相关的信息，范围是[0x00000000, 0x01000000)。地址0x0 ~ 0x7存放的64位整型数据存储的是输入缓冲区的大小，用于读入操作的实现（尚未实现）。

地址0x8 ~ 0xf存放的64位整型数据存储的是输出缓冲区的大小，用于输出操作的实现。

地址0x10 ~ 0x13存放的32位整型数据存储的是随机数种子，在程序加载到内存中时会被自动置入，用于伪随机随机函数的实现。

2. 用户代码区

该段区域存放二进制用户代码，范围是[0x01000000, 0x02000000)。

读入的目标文件代码会被放置在以地址0x01000000为起始的连续内存空间中，代码若超出用户代码区的长度则超出部分会被截断。加载完目标文件后，PC会被设置为目标文件中main函数的地址。该区段理论上是只读的，但是目前的实现中还没有对此区域进行保护，若操作不当，可能会使代码被修改。

3. 栈区

该段区域用于存放函数参数、返回地址以及函数的局部变量等数据，范围是[0x02000000, 0x03000000)。

运行时，栈顶向低地址方向扩张，即：栈底为高地址，栈顶为低地址。成功加载目标代码后，栈指针会被初始化为0x03000000（栈区的上边界）。入栈操作会将栈指针的值减去8并存入数据，出栈操作会将栈指针的值加上8并取出数据。在调用函数或函数返回指令过程中，PC值出入栈的操作会自动执行。另外，栈中数据的读写不只局限于栈顶，从栈顶到栈底的任意位置都可随机访问读写。

理论上，应有栈溢出检测机制，但从该虚拟机实现的复杂性上考虑，没有加入该功能。若操作不当致使栈溢出，可能会覆盖用户代码区的内容，导致运行错误。

4. 用户数据区

此区域存放用户数据，范围是[0x03000000, 0x10000000)。

此区域用户可自由支配，用于存放全局变量，局部变量，数组等等。

5. 输入缓冲区

此区域存放来自键盘输入的临时数据，范围是[0x10000000, 0x10080000)。

缓冲区大小的数据（0x0 ~ 0x7）会实时反映缓冲区所含有效数据的字节数。目前输入尚未实现。

6. 输出缓冲区

此区域存放待输出的临时数据，范围是[0x10080000, 0x10100000)。

缓冲区大小的数据（0x8 ~ 0xf）**不会**实时反映缓冲区的有效字节数，但是在触发中断，**调用输出功能之时必须与缓冲区真实大小保持一致**，不然可能会导致输出结果异常。

7. 显示缓冲区

此区域存放显示数据，范围是[0x10100000, 0x10200000)。

该区存放的即是每个像素的颜色信息，每个像素信息占四字节，其中R(Red)、G(Green)、B(Blue)、A(Alpha)各占一字节。当前的设计中，对Alpha通道还无法提供完整支持。由于显示器的分辨率为200 X 125，实际有效的数据区域范围是[0x10100000, 0x101186a0)。访问此存储区时，必须保证数据对齐，即，地址为4的倍数，且访存指令只能为 `writel`。

五、汇编语言

此虚拟机使用的汇编指令完全基于本机的指令系统。

1. 语法单元

本机汇编语言共有5种语法单元：指令标识符、立即数、寄存器标识符、虚拟存储地址、地址标签。

指令标识符，唯一确定地标识指令，指令列表中所有指令的名称都是指令标识符。

立即数，用于立即数寻址，以字符 `$` 开头，支持十进制或十六进制表示（十六进制需加上 `0x` 前缀，字母大小写均可）。

寄存器标识符，唯一确定地标识寄存器文件，用于寄存器寻址。寄存器列表中的所有寄存器名称都是寄存器标识符。

虚拟存储地址，唯一确定地标识虚拟存储地址，用于基址寻址、偏移量寻址和基址+偏移量寻址。格式为 `I(rB)`，其中 `I` 为立即数，不需要以 `$` 开头，支持十进制或十六进制表示（十六进制需加上 `0x` 前缀，字母大小写均可）；同时，`I` 也可以是地址标签，汇编器会将地址标签替换为立即数；`rB` 为寄存器标识符，表示基址寄存器。所表示的地址为 `I+R[rB]`，`R[rB]` 代表寄存器 `rB` 存储的数值。立即数和寄存器标识符均不省略时，寻址方式为基址+偏移量寻址；省略立即数时，寻址方式变为基址寻址；省略寄存器标识符时，寻址方式变为绝对寻址；两者不允许同时省略。

地址标签，表示下一条指令的起始地址，方便跳转指令和虚拟存储地址的编写。地址标签分为两类，一类是目的标签，另一类是源标签。目的标签表示的是跳转语句将要跳转的目的地址，标签末尾要写上 `:`；源标签会在汇编过程中被替换为目的地址。函数名也是地址标签。汇编器提供了几个地址标签：

```
1 // 数据区段起始位置
2 data_sec_pos 0x03000000
3
4 // 存放输入缓冲区大小的位置
5 in_buf_size 0x00000000
6
7 // 输入缓冲区起始位置
8 in_buf_pos 0x10000000
9
10 // 存放输出缓冲区大小的位置
11 out_buf_size 0x00000008
12
13 // 输出缓冲区起始位置
14 out_buf_pos 0x10080000
15
16 // 显示缓冲区起始位置
17 disp_buf_pos 0x10100000
18
19 // 随机数种子的位置
20 random_seed_pos 0x00000010
```


2. 语法要求

汇编器的语法要求有：

- 每条指令占一行，地址标签占一行；
- 字符`;`为单行注释符，从`;`字符起始到行末都为注释内容；
- - 无操作数指令格式为 `instruction` ；
 - 单操作数指令格式为 `instruction <operand>` ，指令标识符与操作数之间必须间隔一个空格；
 - 双操作数指令格式为 `instruction <operand 1>, <operand 2>` ，指令标识符于第一个操作数之间必须间隔一个空格，操作数1个操作数2之间必须间隔一个`,` 和一个空格；
 - 目的标签末尾应加上`:`，如 `main:`，`.L3:` 等；源标签末尾不加`:`；源标签必须有唯一的目的标签相对应，目的标签则可以对应任意多个源标签；
- 每个汇编程序中必须有一个main函数，虚拟机将 `main:` 当作入口执行指令。

3. 函数调用

i. 寄存器内容的保存

函数调用过程中，调用者寄存器的值应当被保存。从设计上说，一部分寄存器应由调用者保存，这些寄存器是 `%rax, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11`，另一部分则由被调用者保存，这些寄存器是 `%rbx, %rbp, %r12, %r13, %r14`。`%rsp` 即栈指针寄存器，应由调用者和被调用者共同维护。

ii. 函数参数

函数的前六个参数顺序是 `%rdi,%rsi,%rdx,%rcx,%r8,%r9`，第七、第八乃至更多的参数则由栈来传递。所有参数应当在调用函数之前就存入相应的寄存器或栈中。超出6个参数的部分，应当以逆序入栈，即，最后一个入栈的应是第七个参数。在调用函数时，返回地址会被压入栈中，因此对于被调用者而言第七个参数地址是 `8(%rsp)`，第八个参数地址是 `16(%rsp)`。

iii. 返回值

函数返回值保存在 `%rax` 中，若有多余一个返回值，可以利用指针传递返回值。

4. 库函数

汇编器提供了一些库函数，实现了打印数据、绘图、随机数等功能。以下列举了几个函数：

```
1 println // 接收一个64位整型参数，将其当作有符号数打印并换行
2 draw    // 接受参数x, y, v, 在显示器(x, y)位置绘制颜色为v的像素点
3 repaint // 刷新显示屏，打印显示缓冲区的内容
4 random  // 随机数函数返回[0, 48271)之间的一个随机数
```

六、虚拟机实现框架

1. 程序结构

本程序使用java实现，主要包含以下代码文件：

- Main.java：完成虚拟机逻辑层面的组装与运行；
- Processor.java：实现处理器功能；
- AbstractFile.java：接口，为文件设备提供抽象接口；
- ReadableFile.java：接口，为可读设备提供抽象接口；
- WritableFile.java：接口，为可写设备提供抽象接口；
- IOBridge.java：IO桥接器，实现ReadableFile和WritableFile接口，连接一切文件设备；
- Memory.java：主存，实现ReadableFile和WritableFile接口；
- Keyboard.java：模拟键盘设备，实现ReadableFile接口；
- TextOutput.java：文本输出设备，实现WritableFile接口；
- Display.java：模拟显示设备，实现WritableFile接口。

注：下列代码只是源代码的抽象示意，许多具体的实现细节没有在其中体现。

2. 虚拟存储系统程序框架

i. AbstractFile接口

定义了getMaxSize(), isReadable(), isWritable()抽象方法。

```
1 interface AbstractFile {
2     methods:
3         getMaxSize()
4         isReadable()
5         isWritable()
6 }
```

ii. ReadableFile接口

继承AbstractFile接口，定义了readq(), readl(), readw(), readb()四个读方法。

```
1 interface ReadableFile extends AbstractFile {
2     methods:
3         readq()
4         readl()
5         readw()
6         readb()
7 }
```

iii. WritableFile接口

继承AbstractFile接口，定义了writeq(), writel(), writew(), writeb()四个写方法。

```
1 interface WritableFile extends AbstractFile {
2     methods:
3         writeq()
4         writel()
5         writew()
6         writeb()
7 }
```

iv. Memory类

模拟主存，实现ReadableFile和WritableFile接口，用一维字节数组模拟主存存储空间。

```
1 class Memory implements ReadableFile, WritableFile {
2     variables:
3         byte ram[MAX_SIZE] //随机访问存储器
4
5     methods:
6         @Override {           // 继承自接口的方法
7             getMaxSize()
8             isReadable()
9             isWritable()
10            readq()
11            readl()
12            readw()
13            readb()
14            writeq()
15            writel()
16            writew()
17            writeb()
18        }
19
20        class MemoryPane extends Pane {
21            monitors[10] //10个内存监视器条目
22        }
23 }
```

v. Keyboard类

模拟键盘输入设备，实现ReadableFile接口，用一维字节数组模拟输入缓冲区。

```
1 class Keyboard implements ReadableFile {
2     variables:
3         byte ram[MAX_SIZE] //随机访问存储器
4
5     methods:
```

```

6      @Override {          // 继承自接口的方法
7          getMaxSize()
8          isReadable()
9          isWritable()
10         readq()
11         readl()
12         readw()
13         readb()
14     }
15
16     class KeyboardPane extends Pane {
17     }
18 }

```

vi. TextOutput类

模拟文本输出设备，实现WritableFile接口，用一维字节数组模拟输出缓冲区。

```

1  class TextOutput implements WritableFile {
2  variables:
3      byte ram[MAX_SIZE] //随机访问存储器
4
5  methods:
6      @Override {          // 继承自接口的方法
7          getMaxSize()
8          isReadable()
9          isWritable()
10         writeq()
11         writel()
12         writew()
13         writeb()
14     }
15
16     class TextOutputPane extends Pane {
17         TextArea textArea //输出文本框
18
19     methods:
20         print()            //输出缓冲区内容
21     }
22 }

```

vii. Display类

模拟显示输出设备，实现WritableFile接口，用一维字节数组模拟显示缓冲区。

```

1  class Display implements WritableFile {
2  variables:
3      byte ram[MAX_SIZE] //随机访问存储器

```

```

4
5 methods:
6     @Override {          // 继承自接口的方法
7         getMaxSize()
8         isReadable()
9         isWritable()
10        writeq()
11        writel()
12        writew()
13        writeb()
14    }
15
16    class DisplayCanvas extends Canvas {
17    methods:
18        paint()          //绘图
19    }
20 }

```

viii. IOBridge类

模拟IO桥接器，用来链接CPU与文件设备，实现了ReadableFile和WritableFile接口。以下是IOBridge的简易框架：

```

1 class IOBridge implements ReadableFile, WritableFile {
2 variables:
3     Memory memory
4     Keyboard keyboard
5     TextOutput textOutput
6     Display display
7
8     Processor processor
9
10 methods:
11     loadObject()          // 加载目标文件的方法
12
13     @Override {          // 继承自接口的方法
14         getMaxSize()
15         isReadable()
16         isWritable()
17         readq()
18         readl()
19         readw()
20         readb()
21         writeq()
22         writel()
23         writew()
24         writeb()
25     }
26 }

```

3. 处理器程序框架

处理器包含15个通用寄存器，和专用寄存器：PC、IR、CC、State（如上所述）。考虑到设计复杂度和拟真程度等因素，该处理器使用非流水线化的结构以使运行流程变得更加简单、清晰。

其运行流程是：每一指令周期开始时，先从PC所给地址取出指令，保存到IR中，然后运行指令，期间可能会更改CC中的值，或者读写内存和通用寄存器，最后更新PC的值。如运行过程中出现异常，处理器会更新State寄存器的内容并抛出异常。如果运行正常则进入下一指令周期。

以下是Processor的简易框架：

```
1  class Processor {
2  variables:
3      IOBridge ioBridge
4
5      long regs[15] //寄存器用一维64位整型数组模拟
6      int pc //程序计数器
7      byte cc //条件寄存器
8      byte state //机器状态
9
10     byte ir[10] //指令寄存器用字节数组模拟
11
12 methods:
13     void fetch() //取指 + 计算下一PC地址
14     // PC保存的值为目标文件中指令的绝对地址，在取指过程中会将PC映射到真实地址
15     void exec() //译码 + 执行 + 访存 + 更新PC
16
17     class ProcessorPane extends Pane {
18         Button load      // 加载程序
19         Button run        // 运行程序
20         Button fast       // 快速运行程序
21         Button pause      // 暂停
22         Button step       // 单步运行
23         Button halt       // 停机并复原初始状态
24
25         registers & pc messages //寄存器和PC的值
26     }
27 }
```

4. 主类框架

```
1  class Main extends Application {
2  variables:
3      Processor processor
4      IOBridge ioBridge
5      Memory memory
6      Keyboard keyboard
7      TextOutput textOutput
```

```

8      Display display
9
10     methods:
11         void start()
12     }
13
14     Main.start() {
15         //链接文件设备到处理器
16         link memory, keyboard, textOutput, display to ioBridge
17         link ioBridge to processor
18
19         init a scene
20         add memoryPane, KeyboardPane, TextOutputPane,
21         DisplayCanvas and ProcessPane to the scene
22
23         add the scene to the stage
24         show the stage
25     }

```

七、使用

1. 文件目录

```

1  DyVm.jar
2  |-- asm
3  |-- assembler
4  |-- docs
5  |-- hex
6  |-- info
7  |-- assemble.bat
8  |-- DyVM.jar

```

说明：

- 文件夹asm，存放汇编代码；
- 文件夹assembler，存放汇编器程序；
- 文件夹docs，存放文档和图片；
- 文件夹hex，存放汇编器输出的目标代码；
- 文件夹info，存放虚拟机所需要的一些信息；
- 文件assemble.bat，批处理文件，双击可调用汇编器。使用前需打开文件制定输入文件和输出文件路径。该文件第二行默认有一条程序调用指令。第一个参数是汇编程序的路径，第二个参数是待汇编的代码的路径，第三个参数输出目标文件的路径。打开此批处理文件时请不要修改第一个参数，第二个参数应该为汇编代码的路径（汇编程序应该放在asm目录下），第三个参数应改为输出文件的路径（应指定在hex目录下）；
- DyVM.jar，虚拟机程序，可双击运行；

2. 汇编器的使用

汇编器存放在assembler目录下。库函数文件存放在assembler/asm目录下，汇编器源代码存放在assembler/src目录下，compile.bat是用来编译汇编器的批处理文件。assembler/main.exe即是汇编程序，它接受两个参数，第一个参数是输入文件路径，第二个参数是输出文件路径，如果汇编过程没有错误，则程序不会有显示输出；如果出现错误，会显示错误信息和出错位置。err_log.txt保存错误信息，如果汇编过程没有错误，则err_log.txt会保存显示了地址的16进制目标文件。

3. 虚拟机的使用

双击DyVM.jar即可运行虚拟机。按下虚拟机面板中的Load按钮会弹出选择文件的窗口，默认路径在hex目录下选择文件后虚拟机会自动加载目标文件到内存。按下Run按钮，虚拟机会进入慢速执行程序状态；按下Fast按钮，会进入快速运行状态；按下Step按钮，会进入单步运行状态；按下pause，虚拟机会暂停运行；按下halt，虚拟机会终止程序的运行，并恢复程序被载入时的初始状态。

八、测试

为了检验虚拟机的正确性和性能，我编写了一些汇编程序进行测试。

1. 求素数程序

该程序打印100以内的素数。汇编文件是prime.asm，对应的目标文件名prime.hex。

```
1 ; program prime
2 ; print all prime numbers below 100
3
4 ; function prime(n) returns boolean
5 prime:
6     irmov $2, %rdx ; i = 2
7     irmov $1, %rax
8     cmp %rdi, %rax ; n <= 1
9     jg .L3
10    xor %rax, %rax ; return 0
11    ret
12 .L3:
13    cmp %rdx, %rdi
14    jge .L4
15    rrmov %rdx, %rsi
16    mul %rsi, %rdx
17    cmp %rsi, %rdi ; i * i > n ?
18    jg .L4
19    rrmov %rdi, %r8
20    idiv %r8, %rdx ; n / i
```



```

21     test %rax, %rax ; test n % i
22     je .L4
23     irmov $1, %rax
24     inc %rdx          ; i++
25     jmp .L3
26 .L4:
27     ret
28
29 ; function main
30 main:
31     irmov $2, %rcx ; i = 0
32     irmov $100, %r10 ; limit = 0
33 .L1:
34     rrmov %rcx, %rdi
35     call prime          ; prime(i)
36     test %rax, %rax
37     je .L2
38     rrmov %rcx, %rdi
39     push %rcx
40     push %r10
41     call println
42     pop %r10
43     pop %rcx
44 .L2:
45     inc %rcx
46     cmp %rcx, %r10
47     jl .L1
48     halt

```

运行结果：

文本输出框输出了1~100内的全部素数。

2. 生命游戏

该程序模拟康威生命游戏的运行。生命游戏是一种二维元胞自动机，具体规则可参考[维基百科-康威生命游戏](#)。其中细胞的颜色会随周期型变化。汇编文件名是GOL.asm，对应的目标文件名是GOL.hex。

```
1 ; function dx(x, d)
2 dx:
3     push %rbx
4     add %rdi, %rsi
5     rrmov %rdi, %rax
6     rrmov %rdi, %rbx
7     irmov $200, %rsi
8     add %rdi, %rsi
9     sub %rbx, %rsi
10    cmp %rax, %rsi
11    cmovge %rbx, %rax
12    test %rax, %rax
13    cmovl %rdi, %rax
14    pop %rbx
15    ret
16
17 ; function dy(y, d)
18 dy:
19     push %rbx
20     add %rdi, %rsi
21     rrmov %rdi, %rax
22     rrmov %rdi, %rbx
23     irmov $125, %rsi
24     add %rdi, %rsi
25     sub %rbx, %rsi
26     cmp %rax, %rsi
27     cmovge %rbx, %rax
28     test %rax, %rax
29     cmovl %rdi, %rax
30     pop %rbx
31     ret
32
33 ; function init
34 init:
35 ; initialize the cell matrix
36     push %rbx
37     irmov $25000, %rbx
38     irmov $7, %r8
39     irmov $3, %r9
40     irmov $1, %r10
41     rrmov %rbx, %rdx
42     xor %rcx, %rcx
43 .L2:
44     call random
45     and %rax, %r8
46     xor %rdi, %rdi
```

```

47     cmp %rax, %r9
48     cmovl %r10, %rdi
49     rmmovb %rdi, data_sec_pos(%rdx)
50     inc %rcx
51     inc %rdx
52     cmp %rcx, %rbx
53     jl .L2
54     pop %rbx
55     ret
56
57 ; function next
58 next:
59     push %rbx
60     push %rbp
61     push %r12
62     push %r13
63     push %r14
64     irmov $25000, %rbx
65     irmov $50000, %rbp
66     xor %rcx, %rcx
67 ; calculate
68     irmov $200, %r8
69     irmov $125, %r9
70     irmov $1, %r12
71     irmov $-1, %r11
72     xor %rcx, %rcx
73     xor %r10, %r10
74     rrmov %rbp, %rdx
75 .L3:
76     ; store x + 1
77     rrmov %rcx, %rdi
78     rrmov %r12, %rsi
79     call dx
80     push %rax ; push x + 1
81     ; store x - 1
82     rrmov %rcx, %rdi
83     rrmov %r11, %rsi
84     call dx
85     push %rax ; push x - 1
86     push %rcx ; push x
87     ; store y + 1
88     rrmov %r10, %rdi
89     rrmov %r12, %rsi
90     call dy
91     push %rax ; push y + 1
92     ; store y - 1
93     rrmov %r10, %rdi
94     rrmov %r11, %rsi
95     call dy
96     push %rax ; push y - 1
97     push %r10 ; push y
98     ; count
99     xor %r14, %r14 ; count = 0

```

```

100         ; x-1, y-1
101     mrmovq 32(%rsp), %rdi
102     mrmovq 8(%rsp), %rsi
103     mul %rdi, %r9
104     add %rdi, %rsi
105     add %rdi, %rbx
106     mrmovb data_sec_pos(%rdi), %r13
107     add %r14, %r13 ; count++ if alive
108     ; x-1, y
109     mrmovq 32(%rsp), %rdi
110     mrmovq (%rsp), %rsi
111     mul %rdi, %r9
112     add %rdi, %rsi
113     add %rdi, %rbx
114     mrmovb data_sec_pos(%rdi), %r13
115     add %r14, %r13 ; count++ if alive
116     ; x-1, y+1
117     mrmovq 32(%rsp), %rdi
118     mrmovq 16(%rsp), %rsi
119     mul %rdi, %r9
120     add %rdi, %rsi
121     add %rdi, %rbx
122     mrmovb data_sec_pos(%rdi), %r13
123     add %r14, %r13 ; count++ if alive
124     ; x, y-1
125     mrmovq 24(%rsp), %rdi
126     mrmovq 8(%rsp), %rsi
127     mul %rdi, %r9
128     add %rdi, %rsi
129     add %rdi, %rbx
130     mrmovb data_sec_pos(%rdi), %r13
131     add %r14, %r13 ; count++ if alive
132     ; x, y+1
133     mrmovq 24(%rsp), %rdi
134     mrmovq 16(%rsp), %rsi
135     mul %rdi, %r9
136     add %rdi, %rsi
137     add %rdi, %rbx
138     mrmovb data_sec_pos(%rdi), %r13
139     add %r14, %r13 ; count++ if alive
140     ; x+1, y-1
141     mrmovq 40(%rsp), %rdi
142     mrmovq 8(%rsp), %rsi
143     mul %rdi, %r9
144     add %rdi, %rsi
145     add %rdi, %rbx
146     mrmovb data_sec_pos(%rdi), %r13
147     add %r14, %r13 ; count++ if alive
148     ; x+1, y
149     mrmovq 40(%rsp), %rdi
150     mrmovq (%rsp), %rsi
151     mul %rdi, %r9
152     add %rdi, %rsi

```

```

153     add %rdi, %rbx
154     mrmovb data_sec_pos(%rdi), %r13
155     add %r14, %r13 ; count++ if alive
156         ; x+1, y+1
157     mrmovq 40(%rsp), %rdi
158     mrmovq 16(%rsp), %rsi
159     mul %rdi, %r9
160     add %rdi, %rsi
161     add %rdi, %rbx
162     mrmovb data_sec_pos(%rdi), %r13
163     add %r14, %r13 ; count++ if alive
164     ; restore rsp
165     irmov $48, %rax
166     add %rsp, %rax
167     push %r8
168     push %r9
169     irmov $2, %r8
170     irmov $3, %r9
171     cmp %r14, %r9
172     jg .L6
173     cmp %r14, %r8
174     jl .L6
175     jg .L7
176     sub %rdx, %rbx
177     mrmovb data_sec_pos(%rdx), %rax
178     add %rdx, %rbx
179     test %rax, %rax
180     jne .L7
181 .L6:
182     xor %rax, %rax
183     jmp .L8
184 .L7:
185     irmov $1, %rax
186 .L8:
187     rmmovb %rax, data_sec_pos(%rdx)
188     pop %r9
189     pop %r8
190     ; increment
191     inc %rdx
192     inc %r10
193     cmp %r10, %r9
194     jl .L3
195     xor %r10, %r10
196     inc %rcx
197     cmp %rcx, %r8
198     jl .L3
199 ; copy
200     xor %rcx, %rcx
201 .L4:
202     rrmov %rcx, %rdx
203     add %rdx, %rbp
204     mrmovb data_sec_pos(%rdx), %rax
205     sub %rdx, %rbx

```

```

206     rmmovb %rax, data_sec_pos(%rdx)
207     inc %rcx
208     cmp %rcx, %rbx
209     jl .L4
210     pop %r14
211     pop %r13
212     pop %r12
213     pop %rbp
214     pop %rbx
215     ret
216 ; end function next
217
218 ; function paint(v) v:color
219 paint:
220     push %rbx
221     irmov $25000, %rbx
222     irmov $0x000000ff, %r10
223     rrmov %rdi, %r9
224     rrmov %rbx, %rax ; address
225     xor %rcx, %rcx ; pos = 0
226 .L1:
227     mrmovb data_sec_pos(%rax), %r8
228     rrmov %r9, %rsi
229     test %r8, %r8
230     cmovl %r10, %rsi
231     rrmov %rcx, %rdi
232     call _draw
233     inc %rcx ; pos++
234     inc %rax ; address++
235     cmp %rcx, %rbx
236     jl .L1
237     pop %rbx
238     call repaint
239     ret
240
241 ; function color: a scheme of changing color
242 color:
243     irmov $0xff000000, %r8
244     irmov $0x00ff0000, %r9
245     irmov $0x0000ff00, %r10
246     rrmov %rdi, %rsi
247     and %rsi, %r8
248     cmp %rsi, %r8
249     je .L21 ; (255,x,x)
250     rrmov %rdi, %rsi
251     and %rsi, %r9
252     cmp %rsi, %r9
253     je .L22 ; (x,255,x)
254     rrmov %rdi, %rsi
255     and %rsi, %r10
256     cmp %rsi, %r10
257     je .L23 ; (x,x,255)
258 .L21:

```

```

259     rrmov %rdi, %rsi
260     and %rsi, %r9
261     cmp %rsi, %r9
262     je .L32                ; (255,255,0)
263     rrmov %rdi, %rsi
264     test %rsi, %r10
265     je .L24                ; (255,x,0)
266 .L31:                    ; reduce blue
267     irmov $0x00000500, %rsi
268     sub %rdi, %rsi
269     jmp .L27
270 .L22:
271     rrmov %rdi, %rsi
272     and %rsi, %r10
273     cmp %rsi, %r10
274     je .L33                ; (0,255,255)
275     rrmov %rdi, %rsi
276     test %rsi, %r8
277     je .L25                ; (0,255,x)
278 .L32:                    ; reduce red
279     irmov $0x05000000, %rsi
280     sub %rdi, %rsi
281     jmp .L27
282 .L23:
283     rrmov %rdi, %rsi
284     and %rsi, %r8
285     cmp %rsi, %r8
286     je .L31                ; (255,0,255)
287     rrmov %rdi, %rsi
288     test %rsi, %r9
289     je .L26                ; (x,0,255)
290 .L33:                    ; reduce green
291     irmov $0x00050000, %rsi
292     sub %rdi, %rsi
293     jmp .L27
294 .L24:                    ; add green
295     irmov $0x00050000, %rsi
296     add %rdi, %rsi
297     jmp .L27
298 .L25:                    ; add blue
299     irmov $0x00000500, %rsi
300     add %rdi, %rsi
301     jmp .L27
302 .L26:                    ; add red
303     irmov $0x05000000, %rsi
304     add %rdi, %rsi
305 .L27:
306     rrmov %rdi, %rax
307     ret
308
309 ; function main
310 main:
311     call init

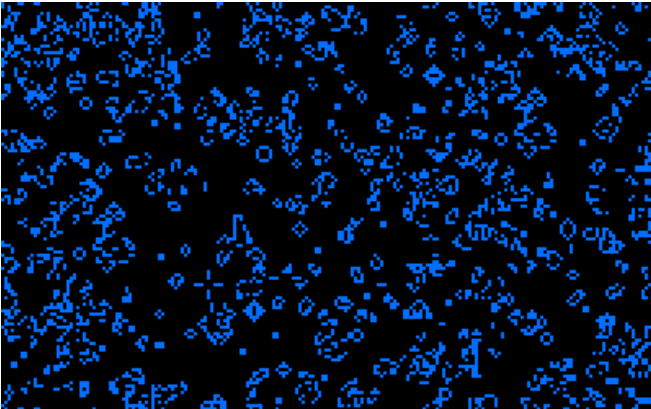
```



```
312     irmov $0x00ffffff, %rdi
313     push %rdi
314     call paint
315 .L0:
316     call next
317     pop %rdi
318     call color
319     push %rax
320     rrmov %rax, %rdi
321     call paint
322     jmp .L0
323     halt
```

测试结果：

显示窗口的输出即是生命游戏的画面。



%rax123

%rcx24

%rdx53124

%rbx25000

%rsp50331544

%rbp50000

%rsi123

%rdi3248

%r8200

%r9125

%r10124

%r11-1

%r121

%r130

%r141

PC2a5

Load

Run

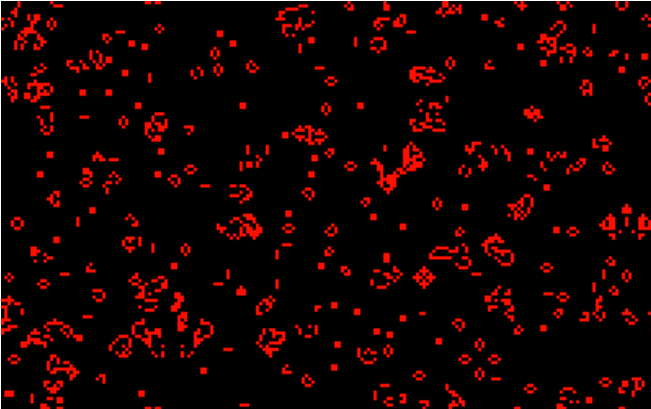
Fast

Pause

Step

Halt

0x08	00 00 00 00 00 00 00 00 a2 09 00 00 00 00 00 00
50331632	99 05 00 00 00 00 00 00 ff ff 6e 00 00 00 00 00
50331544	7c 00 00 00 00 00 00 00 7b 00 00 00 00 00 00 00
0x300b000	01 01 00 01 01 00 01 00 00 00 00 00 00 00 01 01
0x300d000	00 00 00 01 00 01 01 00 00 00 00 00 00 01 00 00



%rax123

%rcx162

%rdx70374

%rbx25000

%rsp50331560

%rbp50000

%rsi125

%rdi248

%r8200

%r9125

%r10124

%r11-1

%r121

%r130

%r140

PC1c9

Load

Run

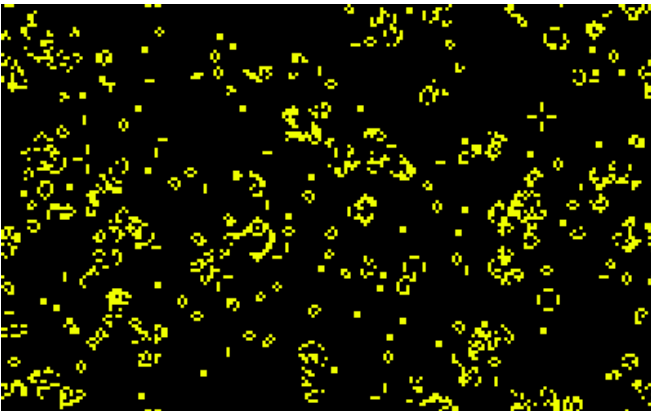
Fast

Pause

Step

Halt

0x08	00 00 00 00 00 00 00 00 a2 09 00 00 00 00 00 00
50331632	99 05 00 00 00 00 00 00 ff 00 0f ff 00 00 00 00
50331544	a8 61 00 00 00 00 00 00 c9 01 00 00 00 00 00 00
0x300b000	00 00 00 01 00 00 00 00 00 01 00 00 00 00 00 00
0x300d000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00



%rax78

%rcx194

%rdx74329

%rbx25000

%rsp50331544

%rbp50000

%rsi79

%rdi49204

%r8200

%r9125

%r1079

%r11-1

%r121

%r130

%r140

PC219

Load

Run

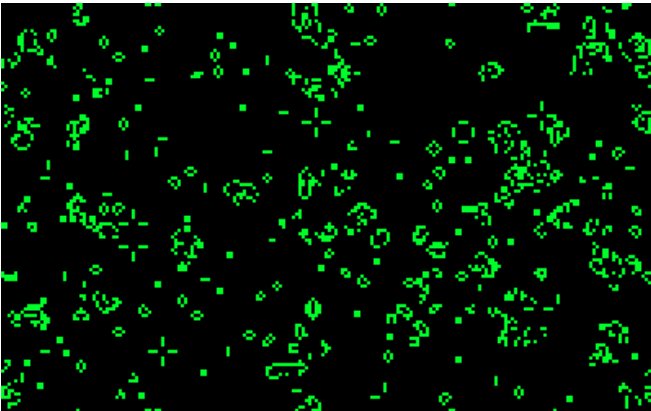
Fast

Pause

Step

Halt

0x08	00 00 00 00 00 00 00 00 a2 09 00 00 00 00 00 00
50331632	99 05 00 00 00 00 00 00 ff 00 ff f5 00 00 00 00
50331544	4f 00 00 00 00 00 00 00 4e 00 00 00 00 00 00 00
0x300b000	00 00 00 01 01 01 00 01 00 01 01 00 00 00 01 01
0x300d000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00



%rax0

%rcx22268

%rdx72268

%rbx25000

%rsp50331592

%rbp50000

%rsi0

%rdi25000

%r8200

%r9125

%r100

%r11-1

%r121

%r130

%r140

PC3ac

Load

Run

Fast

Pause

Step

Halt

0x08	00 00 00 00 00 00 00 00 a2 09 00 00 00 00 00 00
50331632	99 05 00 00 00 00 00 00 ff 23 ff 00 00 00 00 00
50331544	7c 00 00 00 00 00 00 00 7b 00 00 00 00 00 00 00
0x300b000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x300d000	00 00 00 00 00 00 01 01 00 00 00 00 00 00 00 00

九、问题与后续计划

1. 总结

本次实验中，我基本完成了自己预期的设计效果，完成了处理器的正常功能，虚拟存储系统的设计和实现。此外，完成了重要的文本输出和图形显示部分，尤其是图形显示部分，该部分的实现使得本机可以运行较为复杂的图形程序，产生更绚丽的显示效果。

2. 当前问题

当前的设计中仍存在一些缺陷：

1. 当前的虚拟机未完成输入部件的制作；
2. 只支持64位整数操作，虽然提供了一些不同长度数据转换和传送的指令，仍然无法很好的适应32位或更低位程序的运行；
3. 缺乏浮点数操作指令，无法模拟实数运算；
4. 输入与输出缓冲区并不是真正意义上在内存中存在的缓冲区，而是相应设备中的存储区，不利于可能出现的复杂操作的实现；
5. 目前的设计中没有真实地模拟中断操作，在处理软中断时采用的策略是将其当作普通指令去运行，而没有设置中断标志；
6. 异常检测机制和应对措施不够健全。

3. 后续计划

如果未来时间充裕，可能会有一下改进计划：

1. 完成输入部件的开发与测试；
2. 在内存中开辟真正的输入和输出缓冲区，并提供相应的库函数；
3. 真实模拟终端操作，在处理器中设置中断标志位；
4. 完善异常检测机制。

4. 展望

这个虚拟机已经提供一套能完成计算机基本功能的指令集，未来可以为这套指令集系统开发编译器，之后便可用高级语言在此机器上编程。如果未来此机器上实现了输入功能，便可再此机器上开发文本编辑器等应用程序。这个虚拟机有不少可拓展的空间，如果有文件管理系统和操作系统，该机将变得更加完善。