文档

小组成员

姓名	学号	班级
路已人	2018211125	2018211318
吕锐	2018211126	2018211318
陈程	2018211128	2018211318
范珈诚	2018210789	2018211318

文件清单

- `README.md`
- `文档.pdf`
- `encode.csv` 指令参考
- `arkcore` 仿真器
- `compiler` 编译器

- `tests` 内含可在仿真器下执行的程序`fib`和`sort`及其源代码。
- 其他：源码和项目构建脚本

运行环境

- Windows 10 64 bit

编译环境

- GNU G++ Complier
- CMake
- Make
- Windows 10 64 bit

方案

仿真器在整体设计上使用类MIPS设计，16位原型机，在细节上存在自主设计。

仿真器整体分为以下部分：

- CPU模拟
- 存储器模拟
- 数据通路模拟
- 适用于本仿真器的汇编编译器

指令

指令					
ADD	00000	rs	rt	rd	00
ADDU	00000	rs	rt	rd	01
SUB	00000	rs	rt	rd	10
SUBU	00000	rs	rt	rd	11
MUL	00001	rs	rt	rd	00
MULU	00010	rs	rt	rd	00
DIV	00011	rs	rt	rd	00
AND	00100	rs	rt	rd	00
OR	00101	rs	rt	rd	00
XOR	00110	rs	rt	rd	00
NOR	00111	rs	rt	rd	00
SLT	01000	rs	rt	rd	00

SLTU	01001	rs	rt	rd	00
SLL	01010	rs	rt	rd	00
SRL	01011	rs	rt	rd	00
SRA	01100	rs	rt	rd	00
SLLV	01101	rs	rt	rd	00
SRLV	01110	rs	rt	rd	00
SRAV	01111	rs	rt	rd	00
JR	10000	rs	rt	rd	00
ADDI	10001	rs	rt	imm(5)	
ADDIU	10010	rs	rt	imm(5)	
ANDI	10011	rs	rt	imm(5)	
ORI	10100	rs	rt	imm(5)	
XORI	10101	rs	rt	imm(5)	
LUI	10110	000	rt	imm(5)	
LW	10111	rs	rt	imm(5)	
SW	11000	rs	rt	imm(5)	
BEQ	11001	rs	rt	imm(5)	
BNE	11010	rs	rt	imm(5)	
SLTI	11011	rs	rt	imm(5)	
SLTIU	11100	rs	rt	imm(5)	
J	11101	imm(11)			
JAL	11110	imm(11)			
SWI	11111				

寄存器

通用寄存器共16个，分别为`R0~R15`。

- AR：地址寄存器
- DR：数据寄存器
- IR：指令寄存器
- PC：程序计数器

- X，Y：ALU寄存器

存储器

采用LRU算法的分页式内存。使用虚拟地址，数据总线宽度为16位，限制访问地址必须为2的倍数。

当内存未命中时，将通过硬中断通知CPU，并从仿真硬盘中读取相应页载入内存后继续工作。

时钟

仿真器分两个周期，取指令和执行两个周期。每个周期分为4拍。仿真器指令集所有指令均为单周期指令，全部指令可在4拍内完成。

- 取指令：从PC处读指令，装入IR。
- 执行指令：译码，运行。

信号

仿真器实现了信号的仿真。寄存器和主要部件存在写/读信号，另有特殊部件的其他信号。当CPU打开部件信号后，数据通路仿真将会自动模拟数据流动。例如

- 对于通用寄存器`Rx`，有输入信号`Rxi`，输出信号`Rxo`。
- 对于ALU，有令其进行指定运算的信号，有输出信号`ALUo`。
- 对于PC，有自增信号`ADPC`，装载信号`J`。
- 对于存储器，有读信号`RD`，写信号`WE`，AR写`ARi`，DR读写`DRo/DRi`。
- 对于IR，有IR写`IRi`。
- 有停机信号`PAUSE`。

微指令

仿真器采用了简单的微指令译码设计。所有的指令将会被译码为微指令后协调执行。

例如`ADD rs,rt,rd`，需要执行的微指令分别为`Rso,Xi`、`Rto,Yi`、`+,ALUo,Rdi`，微指令将会被传递给相应部件。例如`+,ALUo,Rdi`将会使得`ALU`执行`A+B`，结果经过数据总线写入`Rd`。

每条指令将会被解释为最多4拍微指令，在一个周期内执行完成。

数据通路

仿真器采用单总线设计。

在模拟时，使用图模型描述总线结构。通过控制各个输入/输出信号，由仿真器模拟数据的真实流动，而不是以代码的形式固化。

中断

仿真器目前实现了部分中断功能。

- 硬中断，例如来自存储器的硬中断。
- 软中断，例如`SWI`指令所导致的软中断。

在当前的策略下，通过调用中断可以实现仿真器的暂停和状态的输出。

解决问题

在单线程与多线程间的权衡

仿真器如何合理的模拟整个过程中CPU、存储器、数据通路的物理过程。在实际情况下，这几个部件均为共同工作。但在代码模拟时，必然会出现孰先孰后的问题。

起初，我们想使用多线程技术来解决。不同组件工作在不同的线程下。但是多线程下，存在大量锁和先后问题，给调试和编写代码都带了很大麻烦。后来我们意识到，多线程并没有触及到问题本质。问题的本质在于必须存在一个方法来同步各个部件的运行，这种办法不可能达到现实中的理想状况，但一定要效果等价。

最后，我们决定弃用多线程。通过合理安排模拟过程以及自动机状态转移的方法来模拟每一拍应该进行的动作。这种类似生成器的办法更加符合逻辑，同时提高了性能，减少了多线程带来的复杂的竞态条件。

如何模拟数据通路

如何模拟数据通路，这是首先遇到的问题。这不是逻辑层面上，而是代码设计上的问题。以面向对象的设计方法，将各个存储单元分散到相应部件内，将会给未来数据通路的模拟、信号开关模拟带来很大麻烦。需要在面向对象的基础上添加额外的机制。

通过IoC控制反转的设计方法，设计一专门模块，使用图模型描述数据通路构成，统一由该代码模块管理并模拟存储单元之间的数据流通，并通过该模块向所有其他模块注入它们所需要的存储单元。

如此设计后，信号的模拟也变得简单，只需要在该模块修改指定存储单元的开关，无需再处理其他问题，代码变得简约。

……

使用及测试说明

仿真器在使用上可以分为两部分：

- 仿真器：主体
- 编译器：将汇编代码编译为可以被仿真器加载的可执行文件

一般情况下，可以直接使用已经准备好的仿真器和编译器。

仿真器与编译器的源码也已经同时提供，在必要情况下，可以选择从源码编译执行。

使用方法

首先，需要准备经过编译的可以在仿真器下运行的程序。

使用本仿真器指令集编写汇编代码，保存后调用仿真器提供的编译器，对汇编代码进行编译。

```
./compiler <汇编文件> <编译后输出文件>
# 例如，您准备了`fib.asm`，接下来使用命令
# ./compiler fib.asm fib
# 编译生成fib可执行文件，该文件可以在仿真器下运行。
```

生成能够在仿真器下运行的二进制码后，可以装载入仿真器运行。

我们在`tests`目录下准备了两个测试文件`fib`和`sort`，二者分别为求解斐波那契数列和排序的程序。

假设您的可仿真程序文件名为`fib`。使用命令行运行

```
./arkcore fib
```

仿真器将装载程序`fib`进入内存，而后启动仿真。仿真器会在每条指令运行后输出内存和寄存器状态。当遇到停机指令后，现行策略为暂停仿真运行，此时您将看到如下提示

```
paused. Input register name or memory address to watch its value.
Format: Rx or ADDR [n -the number of words]
```

此时，您可以依照提示输入查询并回车，查看指定寄存器或者内存的数据。例如使用`0 10`查看内存中动态生成的斐波那契数列。

当不输入任何内容并回车时，仿真器将继续执行指令。

- 对于自带的`fib`程序。在运行后，第一次暂停时，输入查询指令`0 10`来查看计算后保存在内存中的前10项斐波那契数列。
- 对于自带的`sort`程序。在运行后，第一次暂停时，可以输入查询指令`0 10`查看排序前的数组；不输入内容并回车继续运行，在第二次暂停后输入`0 10`查看排序后的数组。

编译仿真器

一般情况下，无需自行编译仿真器。如果确定要这么做，请继续阅读本部分。

在这一步，我们将编译仿真器。

请注意将仿真器的编译与可仿真程序的编译区分来开。

我们假设你的系统为`Windows`，需准备构建工具：

- GNU G++ Complier
- CMake
- Make

请转到项目所在目录，执行：

```
cmake CMakeLists.txt -G "MinGW Makefiles"
make
```

在理想情况下，目录下将生成可执行文件`arkcore`。若出现错误，这可能是由于缺少上述构建工具，或工具版本过低引起，请根据提示排错。

> 如果你的系统为`Linux`，请运行`cmake CMakeLists.txt`和`make`编译。

编译编译器

一般情况下，无需自行编译编译器。如果确定要这么做，请继续阅读本部分。

在这一步，我们将编译仿真器。

请注意将编译器的编译与可仿真程序的编译区分来开。

需准备构建工具：

- GNU G++ Complier

构建过程：

```
g++ compiler.cpp -o compiler -std=c++14
```

在理想情况下，目录下将生成可执行文件`compiler`。若出现错误，这可能是由于缺少上述构建工具，或工具版本过低引起，请根据提示排错。

体会

在完成以上内容的时候，我们遇到了很多问题，在解决这些问题时学到了很多。

例如我们了解了中断指令和特权指令以及异常处理的实现和意义。发现其大部分都与操作系统密不可分。例如中断被大量使用于系统调用、异常处理的过程调用，异常处理除硬件支持外还需要软件的配合。而整个部分还存在名为IDT中断描述符表的概念。在这个过程中，我们还了解了内核态与用户态的概念及其在硬件层面上的体现，了解了在访问虚拟地址的全过程中硬件与软件的分工配合流程，了解特权指令的概念。因此，能够较为合理地安排我们仿真器各个功能的实现方法，从中找到平衡点。

同时，我们也研究了高级语言向我们的仿真器编译的可能性。发现有一定的难度。例如使用LLVM框架，为高级语言编写LLVM中间汇编语言到目标机器（仿真器）的后端编译，需要更加深入的了解系统与硬件间的问题，明确从哪部分开始该在编译完成。此后是IR的格式，LLVM提供的平台无关代码生成框架，语言特性等问题，而这里列出的问题都需要深入学习，最终才能够让仿真器以通用且较为合规的标准接受高级语言。

通过这次作业，增进了我们对计算机内部的组成和运行原理的了解，也增进了对指令、信号、周期、数据通路、存储器、软硬件接口的理解，强化了我们的代码技能与团队协作能力。十分感谢这次作业带来的机会。