模型计算机的设计和调试

一、设计目标

1. 设计一个简单的 8 位计算机；
2. 使用 verilog 语言建立模型，并调试运行；
3. 提供对面向过程编程（C 语言）的支持。

二、总体设计思路

设计时主要参考了《数字设计和计算机体系结构》和 MIPS 架构。计算机字长为 8 位。为了支持面向过程编程，以及提供足够强大的功能，经过反复权衡，最后决定设置 16 个通用寄存器，因此需要 4 位寄存器地址。考虑到指令一般需要指定 1-3 个寄存器作为源/目的操作数，有些指令还需要 8 位的立即数，所以将所有指令设计为双字长，即 16 位。

为了便于电路的设计，采用了哈佛架构，即程序与数据分开存储，独立寻址。电路设计主要参考了《数字设计和计算机体系结构》中的单周期 CPU 设计。

三、架构和电路设计

（1）寄存器

通用寄存器共有 16 个，地址为 4 位，每个寄存器的用法按如下约定：

地址	记号	用法
0	$0	零寄存器，只读，恒为 0
1	$v	函数返回值
2-3	$a0 至$a1	函数参数
4-7	$t0 至$t3	临时变量

8-11	$s0 至$s3	保存变量
12	$gp	全局指针
13	$sp	栈指针
14	$fp	帧指针
15	$ra	函数返回地址

为了简化 ALU、寄存器寻址和指令的设计，没有设置标志寄存器，比较指令会像算术/逻辑指令一样把结果写入目的寄存器。也没有溢出标志位等设计。

（2）指令集

指令集的设计着重考虑了以下几个方面：

一是要能够实现尽可能完备的操作。比如，逻辑运算要选取完备集，保证任意逻辑可实现。且任意一个操作要可以使用任意一种寻址方式，也就是指令集结构的正交性（本架构中需要多条指令配合）。

二是要加快常用的操作。例如，存储和加载指令用于在寄存器和 RAM 之间转移数据，RAM 可能的寻址方式有立即数、寄存器值等。但最终决定采用基址（寄存器提供）+偏移（立即数）的方式。这是考虑到这两条指令最常见的用法是堆栈操作，以栈指针为基址，操作其附近的数。这样设计虽然增加了电路的复杂度，但堆栈操作可以以一条指令完成，加快了运行速度，同时也没有损失其正交性。

三是支持面向过程编程。在面向过程编程中，有几个重要的操作需要硬件的支持，即函数调用和返回、堆栈操作。为此设计了相关的寄存器（栈指针、函数返回地址等）和指令。

四是指令的操作码编码，和操作数的位置的设计。主要需要考虑的是使译码

器尽可能简单，尽可能复用数据通路，尽可能简化电路。不过设计过程主要凭经验和感觉，需要做很多方面的权衡，没有固定的好方法。

下面列出了全部指令（共 15 条）。指令长度均为 16 位，高 4 位是操作码，低 12 位是操作数。操作数中的 Rn 代表寄存器的地址，操作中的 Rn 代表对应寄存器的内容，Imm 代表立即数（4 或 8 位）。

算数/逻辑类

描述	助记符	指令				操作
		操作码	操作数			
加	add	0000	R1	R2	R3	R1=R2+R3
减	sub	0100				R1=R2+(~R3)+1
与	and	0001				R1=R2&R3
或非	nor	0010				R1=~(R2|R3)
移位	sft	0011				下述
大于	gt	0101				R1=(R2>R3)?1:0
小于	lt	0110				R1=(R2<R3)?1:0
等于	eq	0111				R1=(R2==R3)?1:0

nor 中令 R2 或 R3 为$0 可以实现非运算，add 中令 R2 或 R3 为$0 可以在寄存器间转移数据。选用与、或非两种逻辑运算一是为了完备，二是为了以尽可能快的速度实现最常用的与、或、非三种逻辑，也便于实现其他逻辑。

移位指令的操作为：将 R3 寄存器中的值看做补码，当其值为 0 ~ 7 时，将 R2 寄存器中值左移 0 ~ 7 位，当其值为-1 ~ -7 时，将 R2 寄存器中值算术右移 1 ~ 7 位，结果赋给 R1。其他值不建议使用（不保证结果）。

数据转移类

描述	助记符	指令				操作
		操作码	操作数			
加载立即数	loadi	1001	R	Imm		R=Imm
加载	load	1000	R1	Imm	R2	R1=*(R2+Imm)
存储	store	1111	Imm	R1	R2	*(R2+Imm)=R1

加载和存储是对数据存储器（RAM）进行的，R2 为基址，Imm 为偏移，Imm（4 位）在与 R2 相加之前符号扩展到 8 位。采用基址+偏移的方式，便于进行堆栈操作。

大部分指令均以寄存器为源/目的操作数，但配合上面三条数据转移指令，就可以实现以立即数、RAM 中数据为操作数，实现正交性。

跳转类

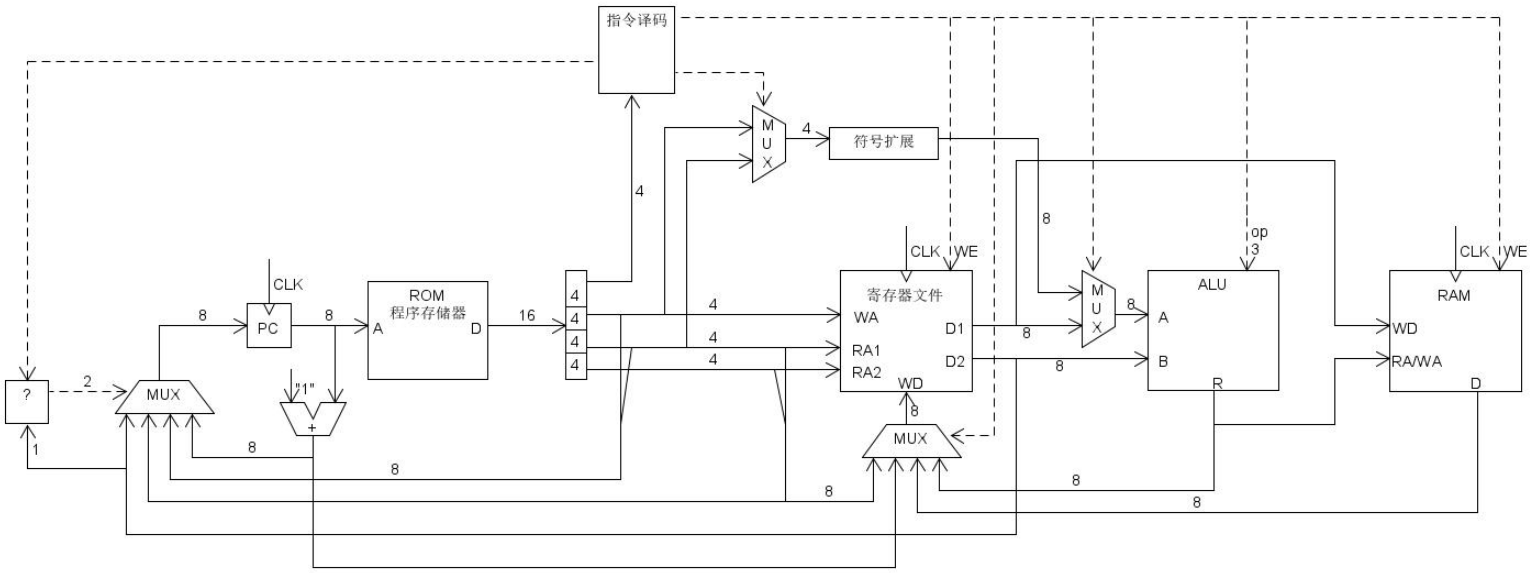
描述	助记符	指令				操作
		操作码	操作数			
跳转并链接	jal	1010	R	Imm		R=PC+1 PC=Imm
寄存器跳转	jr	1110	-	-	R	PC=R
1 条件跳转	jif	1101	Imm		R	if(R[0]==1)PC=Imm
0 条件跳转	jifn	1100	Imm		R	if(R[0]==0)PC=Imm

jal 中，令 R=$0，可以用作普通的无条件跳转（因为$0 只读），令 R=$ra，就是跳转并链接的标准用法——用于函数调用，跳转到被调用函数的入口，并把返回地址置于$ra 中。被调用函数返回时，用 jr $ra 就可以返回到调用指令的

下一条指令。令 R 为其他值违反约定，因此不建议使用。1/0 条件跳转与前面的大于/小于/等于指令配合使用，可以产生大于等于/小于等于/不等于逻辑。

（3）数字电路

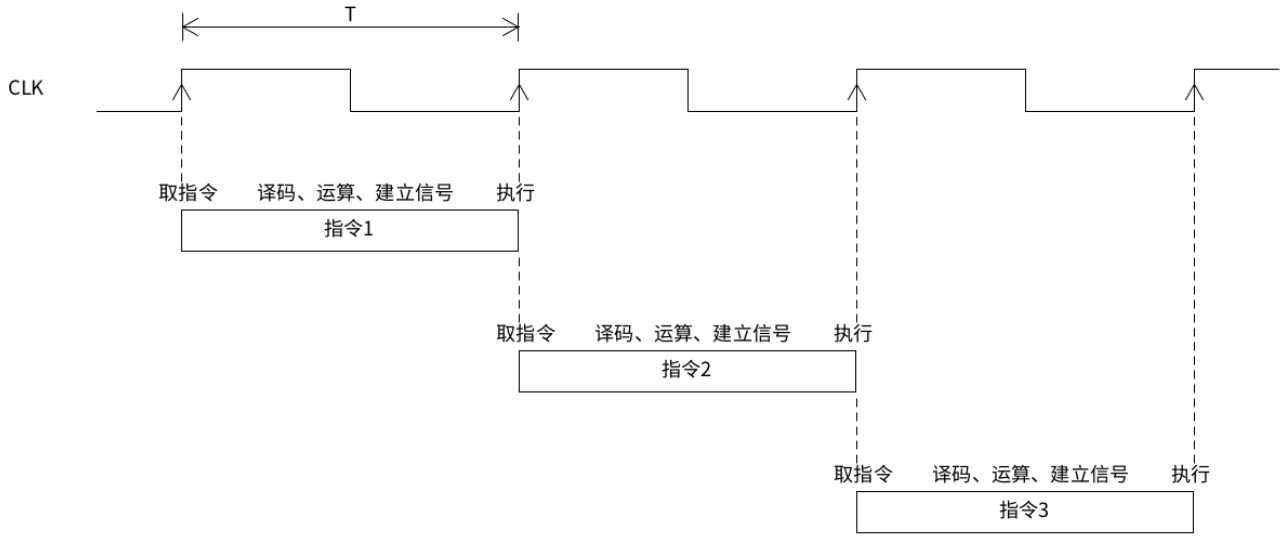
数字电路的设计主要参考了《数字设计和计算机体系结构》中的单周期 CPU 设计。下面是总体电路图：



图中实线为数据，虚线为控制信号，线上标注的数字是位宽。

寄存器文件提供的接口有 WA（写地址），RA1（读地址 1），RA2（读地址 2），D1（读数据 1），D2（读数据 2），WD（写数据），WE（写使能）。D1 和 D2 输出 RA1 和 RA2 指定的寄存器的数据。在写使能的情况下，在时钟信号 CLK 的驱动下将 WD 写入 WA 指定的寄存器。RAM 的接口类似。

下面的时序图说明了指令执行的时序：



在每一个时钟的上升沿（也可定义为下降沿），前一条指令完成执行，同时后一条指令被取出，在两个沿之间，译码、运算等操作将会完成，所有信号建立，为执行做好准备。每个周期执行一条指令。

四、程序设计和测试

开发环境：编译器和仿真器 Icarus Verilog（一般简称 iverilog），波形查看器 WaveTrace（VSCode 插件）。

按照上面的电路图，用 Verilog 实现了模型计算机，各个源代码文件主要内容如下表：

main.v	顶层模块，将所有部件组合起来
ALU.v	ALU 模块
adder.v	全加器模块，多处用到
sl.v	左移模块，实现左移，用在 ALU 中
sr.v	右移模块，实现算术右移，用在 ALU 中

PC.v	程序计数器模块
decoder.v	译码器模块
reg_file.v	寄存器文件模块
ROM.v	ROM 模块，程序存储在其中
RAM.v	RAM 模块，数据存储器

为了测试模型计算机，同时验证其对面向过程编程的支持，设计了一个 C 语言程序 main.c。其内容为用递归方法计算斐波那契数列的第 10 项（55），其中涉及了复杂的函数调用（递归）以及堆栈管理。为了在模型计算机上运行这段程序，首先对这段 C 语言程序进行手动编译，得到了适用于本架构的汇编语言代码 code.s。之后用 Python 编写了适用于本架构的汇编器 assembler.py，从汇编代码生成机器码 code.exe。将机器码复制到 ROM.v 中，之后就可以进行仿真，观察输出波形。以上和 main.c 相关的文件均放在./test 目录下。

汇编命令（在./test 中执行，生成 code.exe，之后复制到 ROM.v 中）

python3 assembler.py code.s

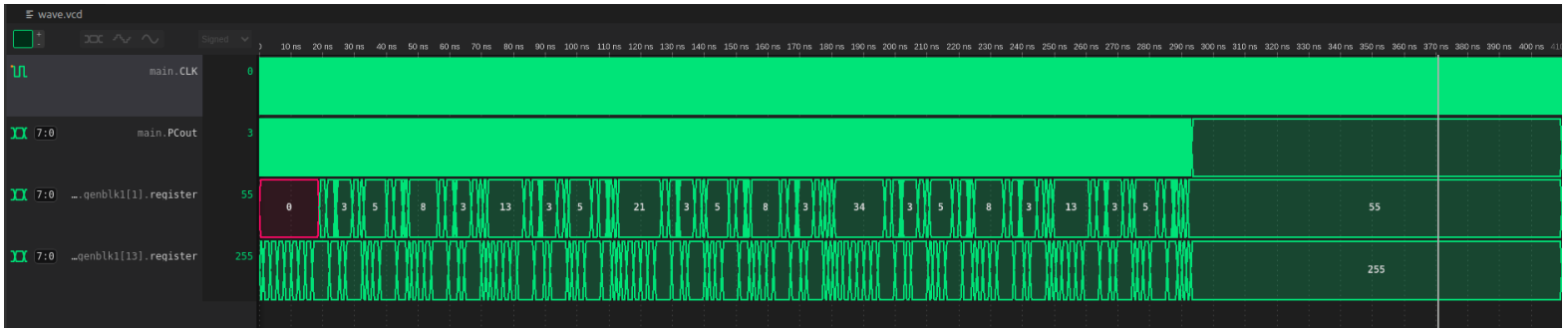
编译命令（生成 a.out）

iverilog main.v

仿真命令（生成 wave.vcd）

./a.out

用 WaveTrace 查看 wave.vcd，如下图



第一行是 CLK，第二行是 PC，第三行是$v，第四行是$sp。通过波形可以看出，程序最终在 PC=3 的位置进入死循环，此时$v=55，即 fibbo(10)的返回值，斐波那契数列第 10 项为 55。在运行过程中，栈指针$sp 从栈顶$sp=255 开始不断变化，也就是栈向下增长，随函数的调用（入栈）和返回（出栈），动态调整，最后返回栈顶。程序运行正常，证明了模型计算机可以正常工作，且对面向过程编程有充分的支持。

五、不足之处和改进

模型计算机尚存在许多不足：

1. 没有标志寄存器，加法运算等难以进行位数扩展；
2. 没有设计 I/O；
3. 缺乏总线设计；
4. 没有定时器、中断系统、MMU 等（对操作系统而言必要的支持）。

对于没有 I/O 的问题，一种可行的改进方案是：采用内存映射 I/O。即将一部分内存（RAM）的地址空间分割出来，映射到 I/O 上。当 CPU 尝试读/写一个地址时，如果这一地址在内存空间中，则内存响应请求，如果在 I/O 空间中，则 I/O 设备响应请求。这样做事实上也会引入总线。

六、总结

本项目设计了一个简单的 8 位计算机，并用 Verilog 语言进行了实现，在其上成功运行了 C 语言程序，证明了设计的可行性和对面向过程编程的良好支持。

与参考设计（Heartbreak 架构）相比，本架构寄存器资源极大丰富，指令集更加精简和高效，功能更加强大，时序也更加简单。在参考设计上，由于寄存器数量过少，且缺少必要的链接并跳转功能，所以甚至无法运行简单的 C 语言

程序，而本架构很好地克服了这一点。但缺点是没有总线和 I/O 设计，这一问题可以考虑通过内存映射 I/O 来弥补。