



下载APP



45 | ARM新宠：苹果的M1芯片因何而快？

2021-08-20 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 20:01 大小 18.34M



你好，我是 LMOS。

前面两节课，我们一起学习了虚拟机和容器的原理，这些知识属于向上延展。而这节课我们要向下深挖，看看操作系统下面的硬件层面，重点研究一下 CPU 的原理和它的加速套路。

有了这些知识的加持，我还会给你说说，为什么去年底发布的苹果 M1 芯片可以实现高性能、低功耗。你会发现，掌握了硬件的知识，很多“黑科技”就不再那么神秘了。

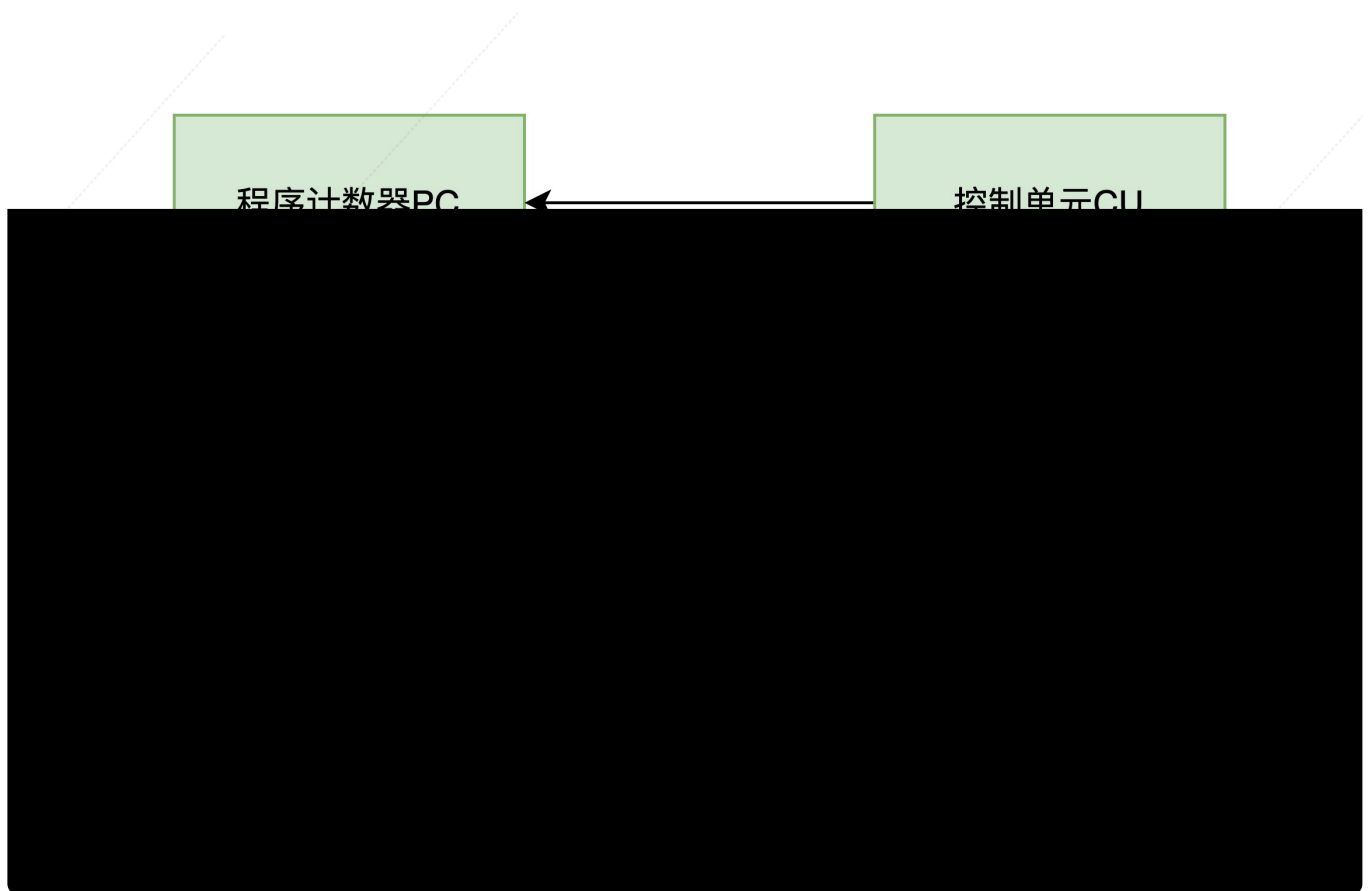


好，让我们正式开始今天的学习！

CPU 的原理初探

经过前面的学习，我们已经对操作系统原理建立了一定认知。从操作系统的位置来看，它除了能够向上封装，为软件调用提供 API（也就是系统调用），向下又对硬件资源进行了调度和抽象。我们通常更为关注系统调用，但为了更好地设计实现一个 OS，我们当然也要对硬件足够了解。

接下来，我们一起看一看硬件中最重要的一个硬件——CPU 是怎么工作的。让我们拆开 CPU 这个黑盒子，看一看一个最小的 CPU 应该包含哪些部分。不同架构的 CPU，具体设计还是有很大差异的。为了方便你理解，我这里保留了 CPU 里的共性部分，给你抽象出了 CPU 的最小组成架构。



CPU架构图

对照上图描绘的基本模块，我们可以把 CPU 运行过程抽象成这样 6 步。

1. 众所周知，CPU 的指令是以二进制形式存储在存储器中的（这里把寄存器、RAM 统一抽象成了存储器），所以当 CPU 执行指令的时候，第一步就要先从存储器中取出（fetch）指令。

2. CPU 将取出的指令通过硬件的指令解码器进行解码。

3. CPU 根据指令解码出的功能，决定是否还要从存储器中取出需要处理的数据。
4. 控制单元（CU）根据解码出的指令决定要进行哪些相应的计算，这部分工作由算术逻辑单元（ALU）完成。
5. 控制单元（CU）根据前边解码出的指令决定是否将计算结果存入存储器。
6. 修改程序计数器（PC）的指针，为下一次取指令做准备，以上整体执行过程由控制单元（CU）在时钟信号的驱动之下，周而复始地有序运行。

看了 CPU 核心组件执行的这 6 个步骤，不知道你有没有联想到 [第一节](#) 的图灵机的执行原理？没错，现代 CPU 架构与实现虽然千差万别，但核心思想都是一致的。

ALU 的需求梳理与方案设计

通过研究 CPU 核心组件的运行过程，我们发现，原来 CPU 也可以想象成我们熟悉的软件，一样能抽象成几大模块，然后再进行模块化开发。

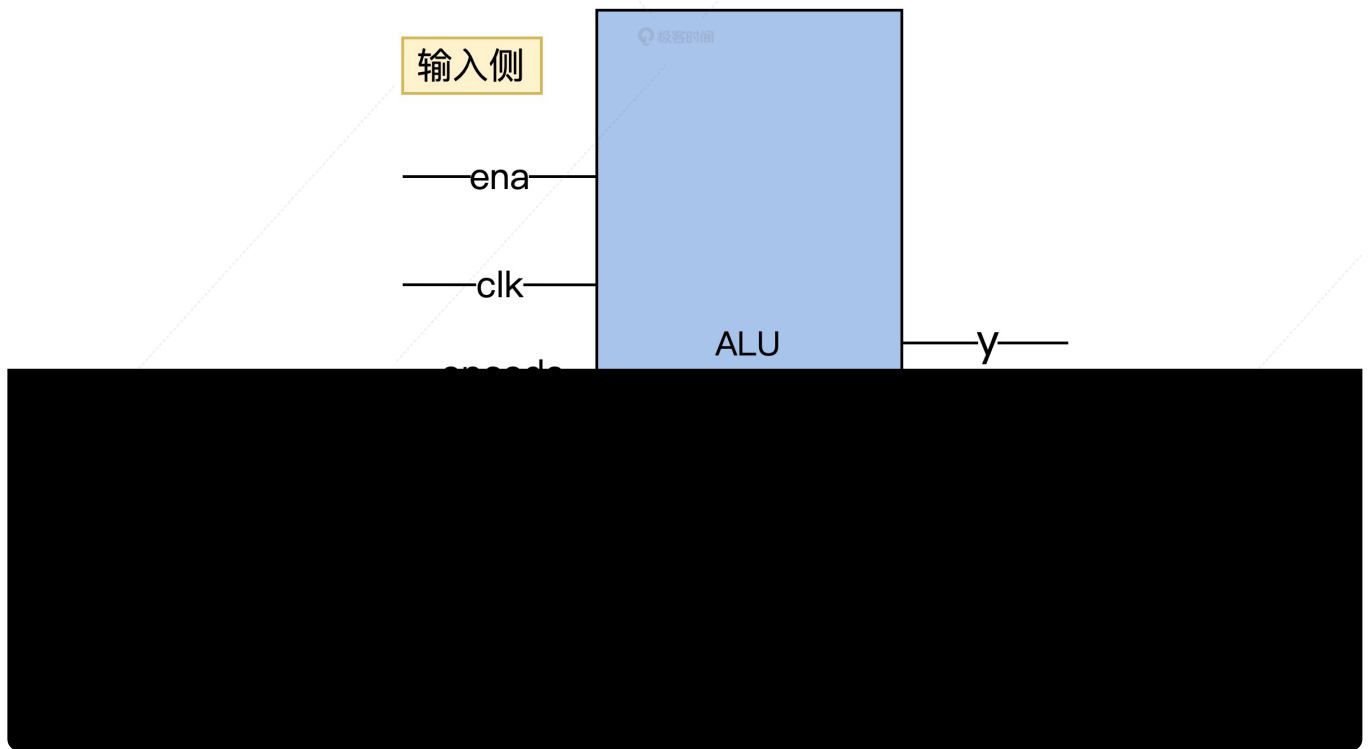
因为从零开始实现一款 CPU 的工程量还是不小的，所以在这里我带你使用 Verilog 语言实现一个可以运行简单计算的 ALU，从而对 CPU 具体模块的设计与实现加深一下认知。

首先，我们来思考一下，对于一个最简单的 ALU 这个模块，我们的**核心需求**是什么？

没错，聪明的你可能已经脱口而出了，我需要能**对两个 N 位的二进制数进行加减、比较运算**。等等，为啥这里没有乘除？还记得学生时代初学乘除法的时候，老师也同样先简化为加减法，方便我们理解。

这里也一样，因为乘除也可以转换为循环的加减运算，比如 2×3 可以转换成 $2+2+2$ ， $6/2$ 可以转换成 $6-2-2-2$ 。所以，只需要实现了加减运算之后，我们就可以通过软件操作 CPU，让它实现更复杂的运算了，这也正是软件扩展硬件能力的魅力。

好了，搞清楚需求之后，先不用着急编码，我们先来根据需求梳理一下 ALU 模块功能简图。



ALU模块功能简图

首先，我们在模块左侧（也就是输入侧）抽象出了 5 根引脚，这五根引脚的作用分别是：

ena：表示使能信号，它的取值是 0 或 1 可以分别控制 ALU 关闭或开启。

clk：表示时钟信号，时钟信号也是 01 交替运行的方波，时钟信号会像人的心跳一样驱动 ALU 的电路稳定可靠地运行。

opcode：表示操作码，取值范围是 00、01、10 这三种值，用来区分这一次计算到底是加法、减法还是比较运算。

data1、data2：表示参与运算的两个 N 位数据总线。

现在我们再来看图片右侧，也就是输出侧的 y，它表示输出结果，如果是加减运算，则直接输出运算后的数值，而比较运算，则要输出 0、1、2，分别表示等于、大于、小于。

好了，有了方案，接下来就让我们想办法把方案变成可落地的实践吧。


自己动手用 Verilog 实现一个 ALU

Verilog 是一种优秀的硬件描述语言，它可以用类似 C 语言的高级语言设计芯片，从而免去了徒手画门电路的烦恼。

目前 Intel 等很多著名芯片公司都在使用 Verilog 进行芯片设计。我们为了和业界保持一致，也采用了这种 Verilog 来设计我们的 ALU。

在开发之前，你需要先进行一些准备工作，安装 VSCode 的 Verilog 语言支持插件、iverilog、gtkwave，这些工具安装比较简单，你可以自行 Google 搜索。

接下来，我们就来实现一下 ALU 的代码，也就是 alu.v，代码如下。

 复制代码

```
1  /*-----
2  Filename: alu.v
3  Function: 设计一个N位的ALU(实现两个N位有符号整数加 减 比较运算)
4  -----*/
5  module alu(ena, clk, opcode, data1, data2, y);
6      //定义alu位宽
7      parameter N = 32; //输入范围[-128, 127]
8
9      //定义输入输出端口
10     input ena, clk;
11     input [1 : 0] opcode;
12     input signed [N - 1 : 0] data1, data2; //输入有符号整数范围为[-128, 127]
13     output signed [N : 0] y; //输出范围有符号整数范围为[-255, 255]
14
15     //内部寄存器定义
16     reg signed [N : 0] y;
17
18     //状态编码
19     parameter ADD = 2'b00, SUB = 2'b01, COMPARE = 2'b10;
20
21     //逻辑实现
22     always@(posedge clk)
23     begin
24         if(ena)
25         begin
26             casex(opcode)
27                 ADD: y <= data1 + data2; //实现有符号整数加运算
28                 SUB: y <= data1 - data2; //实现有符号数减运算
29                 COMPARE: y <= (data1 > data2) ? 1 : ((data1 == data2) ? 0 : 2)
30                 default: y <= 0;
31             endcase
32         end
33     end
34 endmodule
35
```

对照上面的代码块，我帮你挨个解释一下。首先我们定义了 ALU 简图左侧的 5 个引脚，对应到代码上就是抽象成了 module 的 5 个参数（是不是看起来很像一个 C 语言的函数）。

其次，为了能够临时保存运算结果，我们定义了寄存器 y。

再然后，为了区别加、减、比较运算，我们定义了三种状态编码。代码中的 always@其实是 Verilog 中的一个语法特性，表示输入信号的电平发生变化的时候，下边的代码块将会被执行。所以，这里实现的就是当时钟信号发生变化的时候，ALU 就会继续执行。

再之后就是功能的实现啦，功能就是根据 opcode 将对应运算结果保存至寄存器 y。你看，总共才 30 多行代码，我们就实现了一个可以计算任意 N 位二进制数的 ALU，是不是很神奇？

验证测试 ALU

作为一个严谨的工程师，我们除了编码之外，肯定还是要编写对应的测试用例，提升我们的代码的健壮性和可靠性。我们这就来一起编写一下对应的测试代码 alu_t.v，代码如下。

 复制代码


```
1  /*-----
2  Filename: alu_t.v
3  Function: 测试alu模块的逻辑功能的测试用例
4  -----*/
5  `timescale 1ns/1ns
6  `define half_period 5
7  module alu_t(y);
8      //alu位宽定义
9      parameter N = 32;
10
11      //输出端口定义
12      output signed [N : 0] y;
13
14      //寄存器及连线定义
15      reg ena, clk;
16      reg [1 : 0] opcode;
17      reg signed [N - 1 : 0] data1, data2;
18
19      //产生测试信号
20      initial
21      begin
22          $dumpfile("aly_t.vcd");
23          $dumpvars(0, alu_t);
24          $display("my alu test");
```

```

25      //设置电路初始状态
26      #10 clk = 0; ena = 0; opcode = 2'b00;
27          data1 = 8'd0; data2 = 8'd0;
28      #10 ena = 1;
29
30      //第一组测试
31      #10 data1 = 8'd8; data2 = 8'd6; //y = 8 + 5 = 14
32      #20 opcode = 2'b01; // y = 8 - 6 = 2
33      #20 opcode = 2'b10; // 8 > 6 y = 1
34
35      //第二组测试
36      #10 data1 = 8'd127; data2 = 8'd127; opcode = 2'b00; //y = 127 + 127 =
37      #20 opcode = 2'b01; //y = 127 - 127 = 0
38      #20 opcode = 2'b10; // 127 == 127 y = 0
39
40      //第三组测试
41      #10 data1 = -8'd128; data2 = -8'd128; opcode = 2'b00; //y = -128 + -12
42      #20 opcode = 2'b01; //y = -128 - (-128) = 0
43      #20 opcode = 2'b10; // -128 == -128 y = 0
44
45      //第四组测试
46      #10 data1 = -8'd53; data2 = 8'd52; opcode = 2'b00; //y = -53 + 52 = -1
47      #20 opcode = 2'b01; //y = -53 - 52 = -105
48      #20 opcode = 2'b10; //-53 < 52 y = 2
49
50      #100 $finish;
51  end
52
53  //产生时钟
54  always #`half_period clk = ~clk;
55
56  //实例化
57  alu m0(.ena(ena), .clk(clk), .opcode(opcode), .data1(data1), .data2(data2)
58  endmodule


```

在这个测试用例中，我们构造了一些测试数据来验证 ALU 模块功能是否正常，接下来我们就可以使用下面的命令对 verilog 源码进行语法检查，并生成可执行文件。

 复制代码

```
1 iverilog -o my_alu alu_t.v alu.v
```

生成了可执行文件之后，我们可以使用 vvp 命令生成.vcd 格式的波形仿真文件。

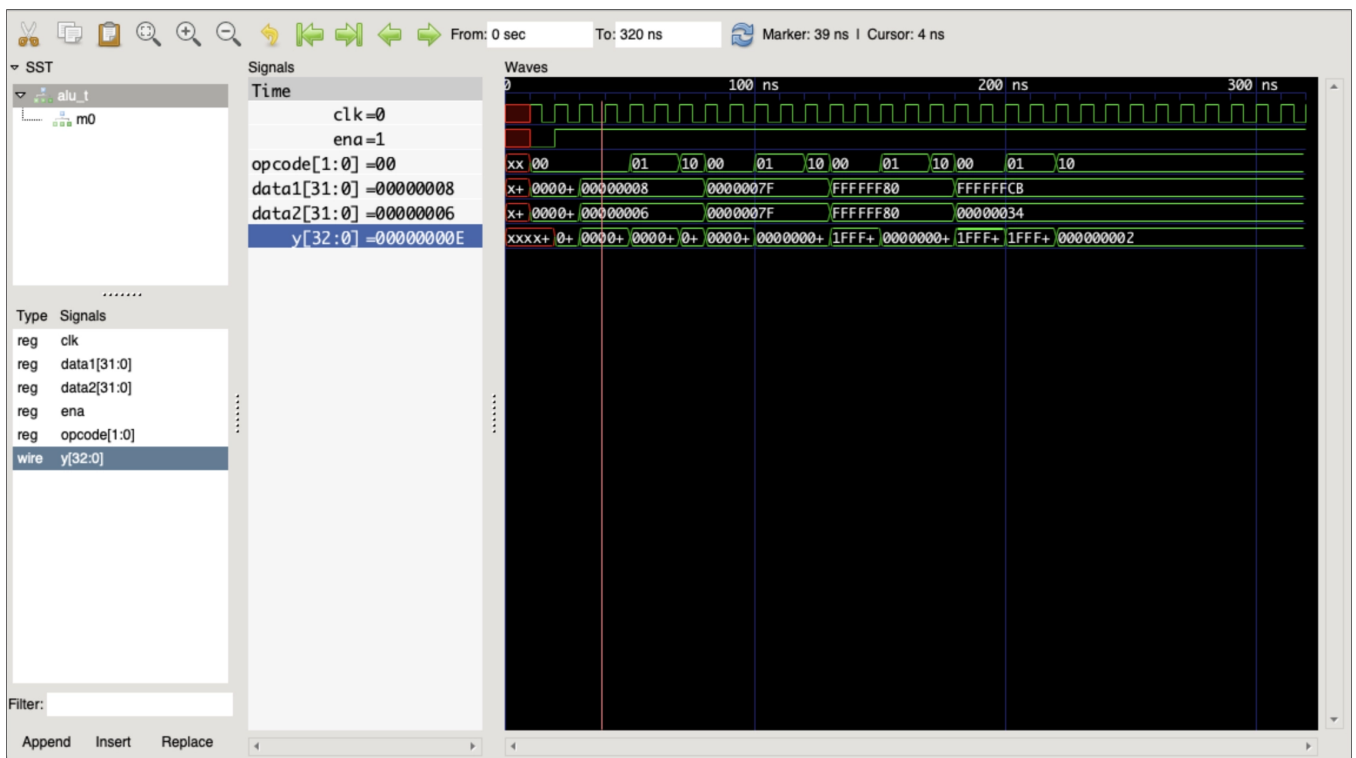
 复制代码

```
1 vvp my_alu
```

接下来，我们再把生成好的波形文件 `aly_t.vcd` 拖入 `gtkwave` 中，就能看到 ALU 模块仿真出的波形图了。

读到这里你可能会疑惑，难道 `verilog` 不支持像 C 语言一样动态调试每一行代码吗？为什么要仿真出波形文件呢？

其实 `verilog` 当然是支持动态调试的，只不过因为硬件芯片在实际运行过程中，有很多逻辑单元都可以并行，如果仅仅依靠动态调试来分析是很困难的。所以在实际开发过程中，我们会先模拟芯片真实运行时的信号波形来进行仿真，才能保证芯片的可靠性。



verilog仿真

在仿真图波行的 signals 信号窗口，我们可以看到，ALU 在每一个时刻入参和出参都和我们预期是一致的，这说明我们已经正确实现了一个 N 位的 ALU 模块。

现代 CPU 加速的套路

可能动手体会之后你还是意犹未尽，这是因为这样实现的模块其实还只是一个入门级的低性能 ALU。上面的例子也只是为了帮你领会原理，因为追求极致的功耗、性能，所以现在我们使用的手机、电脑中的 CPU 基本上都不会设计得如此简单。

因此，如果想要更好使用 CPU 的机制来设计 OS，我们还需要知道真实的工业级 CPU 如何解决问题，看看它们是如何做到动辄几 GHZ 的超高性能的。我为你梳理了常见的五种加速套路：

更多的硬件指令

我们前面实现的 ALU 只实现了三种功能，然而实际真实的 CPU 还会实现乘法、除法、逻辑运算、浮点数运算等等很多硬件指令。这样就可以在一个时钟周期内实现更多的功能，从而提高效率。

通过缓存来提高数据装载效率

在现代计算机体系中，由于磁盘、RAM、CPU 寄存器之间的读写性能开销差别是非常大的，所以在现代 CPU 在设计的时候会在 CPU 内设计多级缓存，从而提高指令读写的速度。

流水线乱序执行与分支预测

我们发现，前面抽象出的 CPU 运行的 6 个步骤其实是串行执行的，而现实世界却不一样，其实计算机内的很多算法可以不按顺序并行执行的。

既然提到了并行，不难联想到我们之前讲的多线程技术。但是多线程开发显然需要对程序做出更多优秀的设计，才能充分利用多核的性能，想要实现比较困难。

那么有没有办法，在不改造程序的前提下充分利用多核的资源呢？答案就是用**空间资源换时间**。硬件层面把程序由解码器电路拆解成多步，调度到 CPU 的不同核心上并行、乱序执行。

比如，加法器在做加法运算的同时，乘法器不应该被闲置，应该也可以执行一些乘法指令。这样我们就可以把程序切分成多个可以并行运行的指令，以此来大幅提升性能了。

当然，形成流水线之后，理想情况就是所有被切分出来的指令都是正确的，这样就可以并行运算了。可惜事情并没有那么简单，因为我们的程序有可能走入了其他分支，后面的运算要依赖前边的结果才能运行。这时候，我们就需要引入分支预测器这个电路，尽可能猜对后面要执行的指令，这样正确切分指令从而提高并行度。

但一旦分支预测器预测失败，就需要重新刷新流水线，让指令顺序执行，这显然就会增加额外的时钟开销，造成性能损失。不过好消息是目前的分支预测器的准确率已经可以达到90%以上了。

多核心 CPU

随着单核心 CPU 的不断优化，我们会发现单核心下的 CPU 遇到了工艺等各种原因造成的瓶颈，很难再有更高的性能提升了。

所以，聪明的工程师又想到了提高并行度的经典套路，将多个 CPU 核心集成到了一颗芯片上。这时候每个 CPU 都有独立的 ALU、寄存器、L1-L3 多级缓存，但多个核心共用了同一条内存总线来操作内存。说到这里，反应快的同学可能会隐约感觉到哪里有些不妥了。

没错，因为内存中的数据被缓存到了 CPU 的多级缓存中，CPU 的多个核心是并行操作数据的，这时如果没有额外的设计的保障机制，就很可能导致并行读写数据引起的数据一致性问题，也就是出现脏数据。

为了解决缓存一致性问题，工程师们又发明出了 MESI、MOESI 等缓存一致性协议来解决这个问题。

超线程

我们发现前边整理的 CPU 核心组件的 6 个步骤，如果再进行进一步抽象，又可以简化的分为取指令和执行两部分。这时候我们发现，其实大部分指令在执行的过程中都不一定会占用所有的芯片资源的。

所以，出于尽可能的“压榨”硬件资源的考量，工程师们又设计了额外的逻辑处理单元用来保证多个可执行程序可以共享同一个 CPU 内的资源。当然，如果两个程序同时操作同一个资源（如某一个加法器）的时候，也是需要暂停一个程序进行避让的。

谈谈指令集

从前面 ALU 的设计过程中，我们发现如果设计一个芯片模块，首先是要根据分析的需求抽象出对应的 opcode 等指令，而众多约定好的指令则构成了这款芯片的指令集。那么常见的 CPU 指令集都有哪些呢，让我们一起来看看吧。

CISC

复杂指令集 Complex Instruction Set Computer 简称为 CISC，其实计算机早期发展的时候还是比较粗暴的，后来大家发现，让硬件实现更多指令可以有效降低软件运行时间，就疯狂地给硬件芯片设计工程师提需求。

于是越来越多的奇奇怪怪的指令被加入了 CPU，最后指令不但越来越多，还越来越复杂。并且为了实现这些指令不但占用了大量的硬件资源，而且长度还不一致，这些都给以后的扩展以及性能优化挖了不少的坑。

挖坑总要后面填坑的，甚至 Intel X86 系列这个经典的 CISC 指令集的 CPU，现在也是通过设计译码器，把变长的 IA32 指令翻译成简单的微代码，然后交给类似 RISC 的简单微操作来执行。这在某些层面上，也许也意味着 CISC 指令集巨头的一次叛逃。

RISC

精简指令集 Reduced Instruction Set Computer，简称为 RISC。经历了 CISC 指令集带来的问题，研究人员就对现代计算机运行的指令做了统计和分析，结果发现大部分的程序在大部分情况下，都在运行一小部分指令。

所以工程师就提出了一个大胆的假设，我们通过少部分相对简短且长度统一的指令集来替代 CISC，这样同样能满足所有程序的需求。经过大量论证和实验后，人们发现这样不但解决了 CISC 指令带来的痛点，还带来了不少性能提升。

ARM 与 M1 芯片

后来 ARM 应运而生，ARM，是 Advanced RISC Machine 的缩写。看名字我们就知道。这是一个精简指令集的 CPU。

早期很多 CPU 都是封闭的，要想设计一款新的 CPU 只能从头设计，这显然需要极高的成本投入。这时候 ARM 公司就抓住了市场痛点，ARM 公司只做指令集和 CPU 的设计，然后付费授权（当然，授权费还是挺贵的）给各个厂商，由厂商根据自己的需求再去定制和生产。

由于 ARM 相对开放的态度，以及 RISC 指令集带来的高性能、低功耗、低成本特点，让它迅速从嵌入式领域杀进了移动设备、PC，甚至超级计算机领域。在 2020 年末，M1 芯片

一经上市测评数据便刷爆朋友圈，以致于 Intel、AMD 这些传统 CPU 在相同功耗的情况下性能被完全吊打，那么苹果到底使用了什么黑科技呢？

首先，苹果的 M1 芯片也是基于 ARM 架构的，它采用了 AArch64 架构的 ARMv8-A 指令集，是由台积电采用 5nm 工艺代工生产的，在芯片内集成了 160 亿个晶体管。显然，它在继承了 ARM 优点的同时，还能享受到更先进的芯片制程带来的高性能与低功耗。

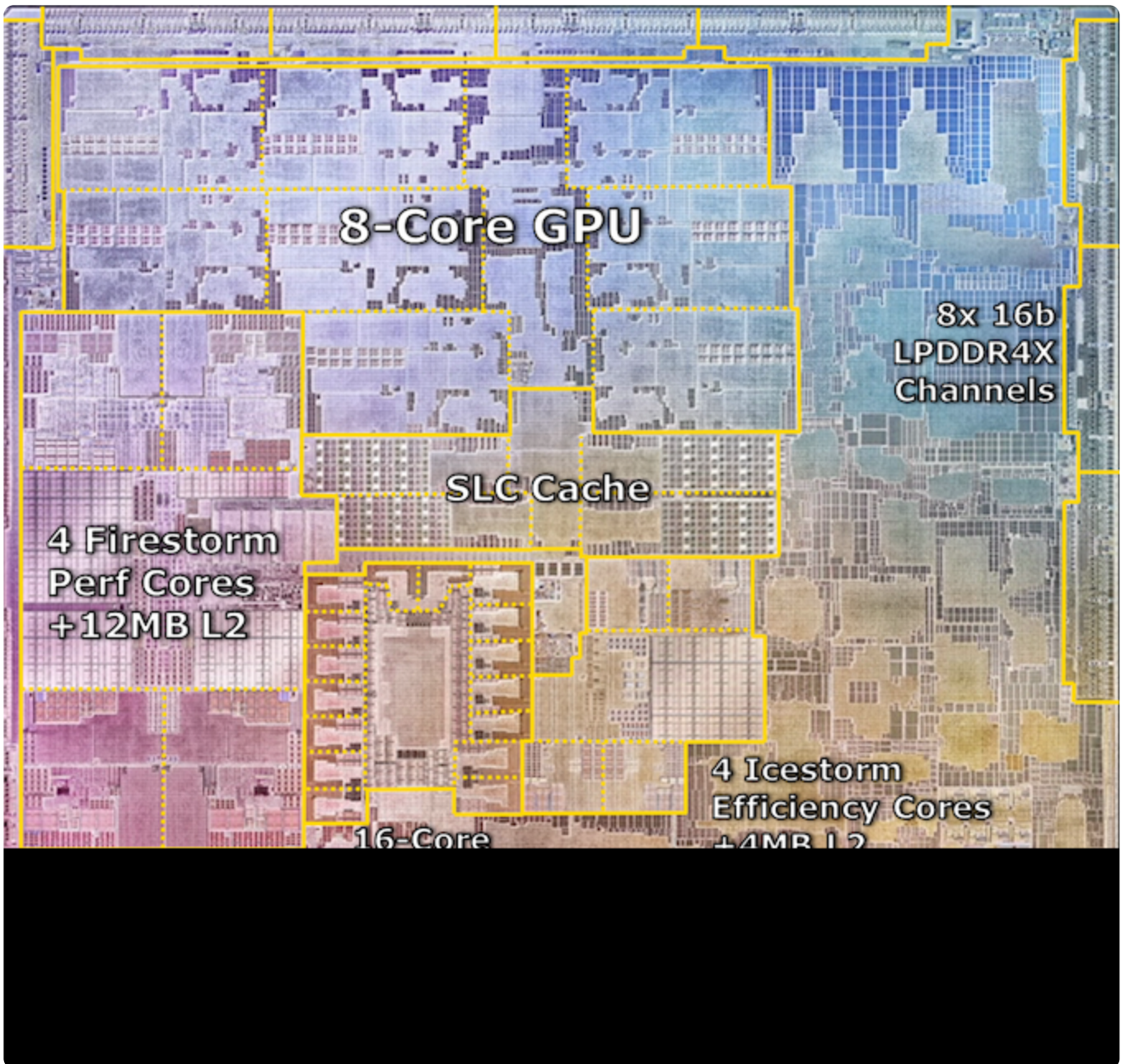
而仅仅单纯继承 ARM 的优势其实还是不够的，因此 M1 芯片还额外引入了如增加解码器电路、统一内存架构、MCU 等多种优化方式来进行设计。接下来，让我们来看一下苹果具体是如何做的：

根据我们之前提到了流水线和乱序执行的原理不难推断，**解码器和 CPU 指令的缓冲空间大小会影响 CPU 的程序并行计算能力。**

所以，苹果工程师在设计的时候，将解码器增加到了 8 个（而 AMD、Intel 的解码器一般只有 4 个）。同时，M1 芯片的指令缓冲空间也比常见的 CPU 大了 3 倍。你可能会好奇，为啥 X86 系列的 CPU 不能多增加点解码器呢？

其实这就是 ARM 的 RISC 指令集的优势了。因为在 ARM 中，每条指令都是 4 个字节解码器，进行切分处理很容易；而 X86 的每条指令长度可以是 1 到 15 字节。这就导致了解码器不知道下一条指令是从哪里开始的，需要实际分析每条指令才可以，这就增加了解码器电路的复杂度。

有了提高并行能力的基础，多核心也是必须的。根据 [@AnandTech](#) 分享的资料来看，M1 芯片内包含了 4 个 3.2GHz 的高性能 Firestorm 核心和 4 个 0.6 ~ 2.064 GHz 的低功耗 Icestorm 核心，这也为 M1 芯片在各种功耗下进行并行计算提供了基础。



M1芯片

我们观察上图可以发现 M1 芯片还集成了苹果自行设计的 8 个 GPU 核心。对手机芯片有了解的同学可能会觉得，高通之类的芯片也集成了 GPU 呀，这里有什么区别呢？其实这里引入了**统一内存**（Unified memory）的设计。

传统的做法是如果 CPU 要和 GPU 之间传输数据，需要通过 PCIe 总线在 CPU 和 GPU 的存储空间内来回传递。

这就好比你有两个水杯，但互相倒水只能靠一个很细的吸管。而统一内存则是可以让 CPU 和 GPU 等组件共享同一块内存空间，这时候 CPU 要想传递数据，只需要写入内存之后通知 GPU 说：“嗨，哥们儿，你要的数据在某个地址空间，你自己直接用就好了。”这样就避免了通过 PCIe 总线传递数据的开销。

最后，我想提醒你注意这一点，它非常重要，**严格讲，M1 芯片其实并不是 CPU**。M1 芯片其实是包含了 CPU、GPU、IPU、DSP、NPU、IO 控制器、网络模块、视频编解码器、安全模块等很多异构的处理器共同组成的系统级（SOC）芯片。

这样做的好处就是不需要在主板上通过各种总线来回传输数据，同时也避免了额外的信号、功耗开销。既然 SOC 的思路这么好，传统厂商为什么没有跟进呢？

原因在于商业模式不同，传统厂商生产 CPU，但 GPU、网卡、主板等模块是交由其他厂商生产，最终由专门的公司组装成一台计算机才对外销售。而 Apple 为代表的厂商的业务模式则是自己就有全产业链的整合能力，可以直接设计、交付整机。所以，不同的业务模式最终催生出了不同技术的方案。

重点回顾

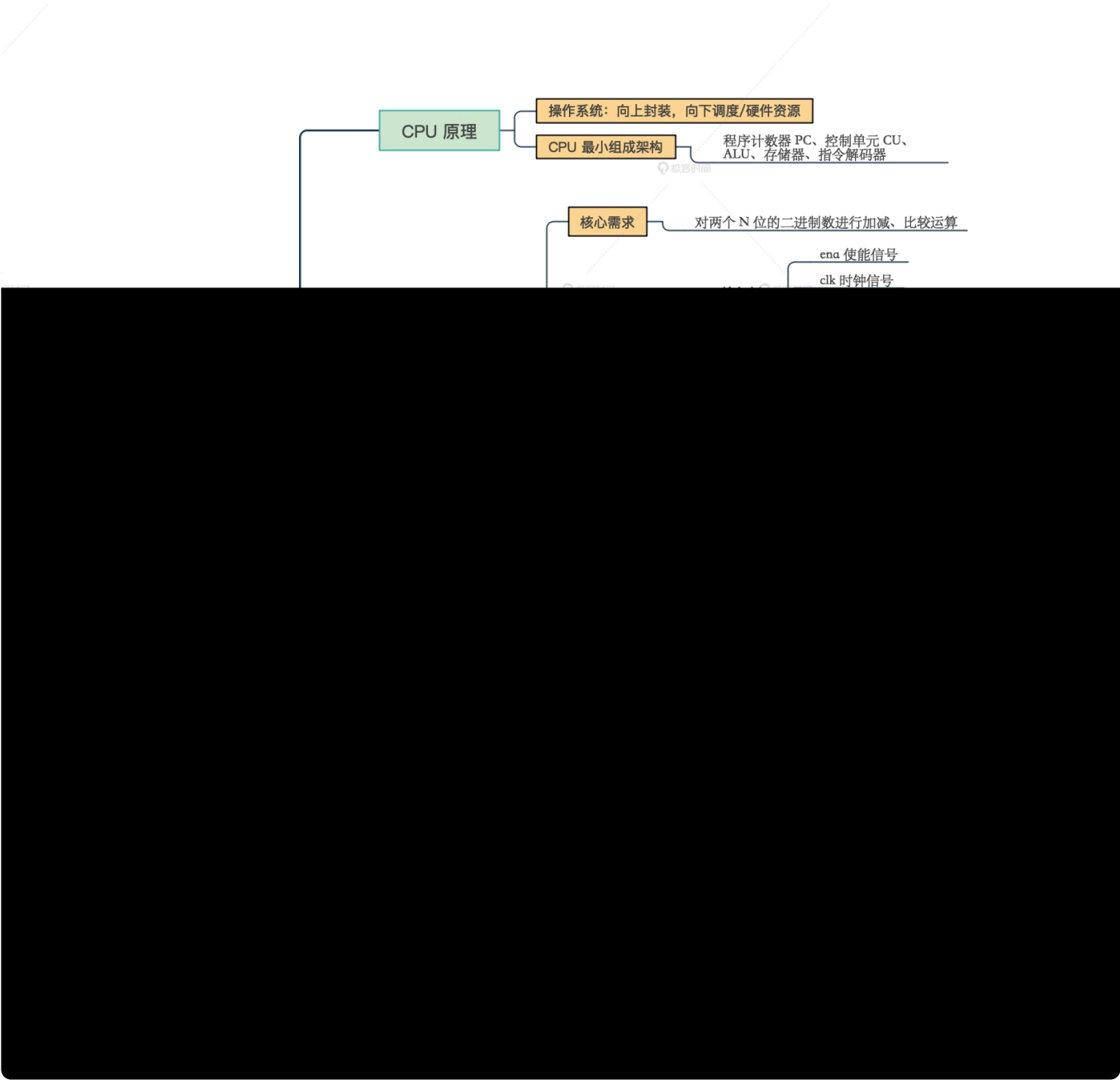
通过这节课的学习，我们明白了对于设计一款操作系统而言，对硬件的理解与把控能力非常重要。而硬件中很关键的一个组件就是 CPU，我们一起分析了一个 CPU 的基本组成和运行步骤。

接着，为了把原理落地，我们一起实现了一个 ALU，带你加深了对 CPU 原理的理解。之后我们还了解了现代 CPU 的发展历程以及设计思路，并分析了 CISC、RISC 指令的区别，以及基于 ARM 指令集的 M1 芯片的特点。

苹果的 M1 芯片，它在继承了 ARM 优点的同时，还做了很多优化，比如增加解码器提高并行计算能力，利用提高指令缓存空间的机制提升了指令加载与计算的效率，还引入了**统一内存**的巧妙设计。

在看到这些优势的同时，我们不妨发散思维，想一想为什么这些想法之前没有实现，这其实和业务模式息息相关。

最后，我特意为你梳理了这节课的导图，帮你巩固记忆。



思考题

除了 ARM 指令集，如果想开发一款 CPU，我们还有更好的 RISC 指令集可选么？

欢迎你在留言区和我交流。也欢迎你把这节课分享给有需要的朋友，说不定就能帮他搞懂 CPU 的原理。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 44 | 容器：如何理解容器的实现机制？

下一篇 46 | AArch64体系：ARM最新编程架构模型剖析

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



精选留言 (5)

💬 写留言



neohope 置顶

2021-08-24

老师设计芯片太豪放了哦，输入输出位数引脚太多了。30位就十分奢侈啦：

ena, 1位；clk, 1位；opcode, 2位

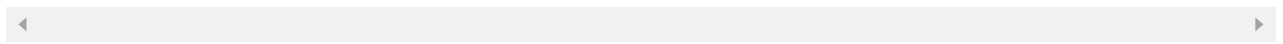
data1, 8位；data2, 8位；y, 10位；

Icarus Verilog : <http://iverilog.icarus.com/home>

Icarus Verilog for Windows : <http://bleyer.org/icarus/>

展开 ∨

作者回复: 是的 这样更简单明了



1

**Fan**

2021-08-20

哈哈，再来几篇Verilog。🐶📖

展开 ∨

作者回复：哈哈



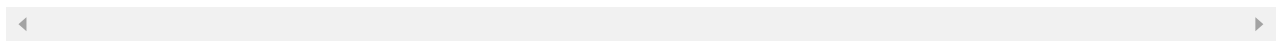
2

**pedro**

2021-08-20

risc-v最近太火了，没有历史包袱，设计又足够精妙，谁不爱呢。

作者回复：哈哈



1

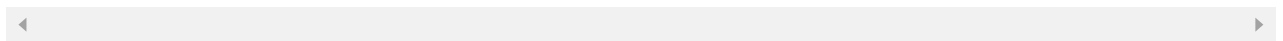
2

**springXu**

2021-08-20

请教东哥，intel那款20年前的IA64(安腾)处理器，也是CISC的指令集么？目前主流RISC指令集就ARM了，mips已经被市场淘汰。RISC-V是开源的指令集，被众多公司所追捧了。还有IBM的PowerPC指令集。

作者回复：安腾不是

**苏流郁宓**

2021-08-20

mips？risc-v？不过，risc指令集实际实现某些功能的步骤比cisc多，苹果的cpu优秀离不开先进的工艺，假如intel的cpu也用5nm工艺，就台式机的cpu的价格估计会翻不少价，当然性能也会好到爆。问题是市场需要吗，认可吗？如同作者所言，这也是商业模式有关，不同的模式决定不同的选择？

展开 ∨

作者回复: RISC在并行流水线上有优势 当然工艺先进是绝对的 可以集成更多晶体管了 就可以实现更多逻辑了

