目录

硬件

CPU:

计算机的速度受到芯片的影响, 通常半导体大小越小, 速度也就越快
CPU 每条指令发出都需要时间, 而这个事件就由 CPU 的 Clock 来决定, 但每次发出信号, 并不是所有半导体都会相应, 且有些电路无法频繁开关(速度比 Clock 慢). 这个 Clock 发出信号的间隔就是 Clock period, $Clock\ rate = \frac{1}{Clock\ period}$

跟 CPU 的表现差不多, $performance = \frac{1}{work\ speed}$

CPI, IPS

但需要注意的是, 指令执行的速度和表现并不永远相关, 因为某些 CPU 的指令可能更加复杂, 所以一条指令做的计算更多, 而某些 CPU 指令执行的很快, 但每条指令都很简单
复杂指令的架构的代表就是 x86 (CISC), 而剩下来的架构(RISC)都是执行更多简单的指令
比如一段代码, x86 中的汇编代码需要 12 行, 而 MIPS 需要 27 行不止. 当然从行数不能判断到底谁快.

我们需要通过 CPI(Cycles per instruction)来判断到底快不快具体计算 CPI 是执行了 x 条指令花了多少 cycles:

$$CPI = \frac{cycles}{x}$$

通常时钟频率越快越好: 因为每次发出信号最多执行一条指令, 如果频率快, 这样就可以快速的执行更多的小指令, 不浪费时间, 费时的指令就可以使用多个 Cycles 来执行. 如果

我们把频率变慢， 如果很多指令比时钟间隔要短， 就浪费了很多时间(当然更快的频率也带来的更高的能耗) 在 2002 年之前很多的性能都是增加频率得到的， 但在之后就停在了 3GHz 左右， 因为再往上功耗会把 CPU 烧坏

如一个 3GHz CPU 每 4 个 Cycle 执行一条指令， 那 10 秒内能执行多少指令?
我们从 3GHz 中能知道这个 CPU 的时钟频率是$3,000,000$， 也就是一秒那么多 cycles. 可以得到一秒能执行$\frac{3,000,000}{4} = 750,000$条指令， 乘以 10 就得到了 10 秒执行的指令数量

而 IPS 表示的是 Instructions per second

$$\frac{Hz}{CPI} = \frac{cycles}{second} * \frac{instructions}{cycles} = IPS$$

制作 CPU

在我们生产 CPU 的时候， 需要把一整块硅切成晶圆， 再把晶圆切成一个一个的 Dies， 然后测试 Dies 后封装到 CPU 里再进行销售
但一点点的灰尘或者瑕疵都有可能导致芯片制作失败， 这些瑕疵可以看做是均匀分布在晶圆上的. 所以可以得出， 单个 Dies 的面积越大， 那么得到好的 CPU 的概率也就越小

$$yield = \frac{1}{(1 + die\ area * defect\ rate)^N}$$

这里的 N 是制作工艺的复杂度， 制作工序越多， 造成瑕疵的可能性也就越大

$$cost\ per\ die = \frac{cost\ per\ wafer}{dies\ per\ wafer * yield}, \quad dies\ per\ wafer \approx \frac{wafer\ area}{die\ area}$$

功耗墙

$$Dynamic\ Power = activity * capacitance * voltage^2 * frequency$$

为了增加 CPU 的频率，人们一直在尝试减小电容大小，电压大小，但电压最低就在 1v 左右，不能再降低了，所以频率也就停在那里了

多核

Response time：相应速度
Through put：单位时间内任务完成数量
对于一个完整的 CPU，一个单独的核心可以自己完成一个线程：有一个小的内存和一些逻辑电路，但有些部分也是多核共享的，可能如除法单元-因为单元比较大
一个多核 CPU 可以同时处理多个任务，直到某两个程序同时需要同一个部分去计算

对于一个 CPU 有三种提升的方法:

- 第一种是提高他的频率: 这样可以提高响应速度， 也表示提高了他的吞吐量(Through put)
- 如果我们把一个 CPU 做成多核的，它的 CPI 可能会提高（这样是不好的， 因为）他的相应速度变慢了， 但吞吐量会提高.
- 和多核相比，还有双 CPU: 也就是用两个相同的 CPU 放一起，这样虽然不会提升响应速度，但吞吐量会翻倍

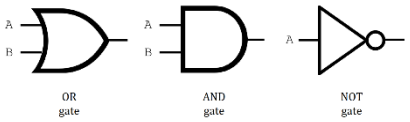
Amdahl's law: 对于程序的优化程度主要取决于没有受到优化的部分

$$T_{opt} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

比如一个视频滤镜每帧需要消耗 2ms 去处理，其中25%的时间花在了乘法上面，有人发明了一个算法，这样乘法的消耗从 15cycles 变成 3cycles， 也就是说它把乘法速度提升了 5 倍(improvement factor)， 被影响的部分是$2 * 25\% = 0.5ms(T_{affected})$， 这样我们就可以算出优化后这个滤镜所需要的时间:

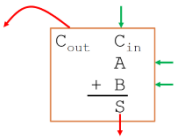
$$\frac{0.5}{5} + 1.5 = 1.6$$

Logical Gate

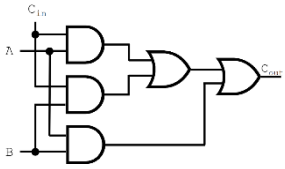


在这节课中， 我们并不会看晶体管层面的逻辑门， 而是看简化过后的:
这三种门就是 universal gate， 最常见的三种逻辑门

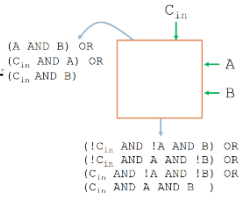
因为非门之接入一个电流， 所以我们可以经常在电路图中看到一个小圆圈， 那个就代表 not gate
计算也是用这三个门产生的: 我们用 and 门产生进一位， 用 xor 组合来检测当前位是多少.
当然， 如果只想用 universal gate 来产生加法， 那就麻烦一些了
右侧的是一个加法逻辑单元， 通过把C_{out}链接到C_{in}就可以产生一个多位 bit 加法器.



比如右侧的就是一个 carry out bit 电路图



对于一个 32 位的加法电路， 他会有 33 个输出. 其中 32 是位数的，还有一个输出是 carry out bit 的输出



做出这种电路图看起来很麻烦， 其实真正做起来却是也比较麻烦: 需要把所有输入的可能性列出来，画出一个 truth table，然后把整个 truth table 改成一个公式: 我们找出所有结果是 1 的，列出公式， 再把所有结果是 1 的公式用 or 连起来就可以. 这样做出来的公式又臭又长, (叫做 SoP, Sum of product，且里面的括号可以省略)需要简化一下.
用逻辑的一些交换， 结合， 分配律来简化， 下面是分配律的例子:

$$A\ and\ (B\ or\ C) = (A\ and\ B)\ or\ (A\ and\ C)$$

$$A\ or\ (B\ and\ C) = (A\ or\ C)\ and\ (A\ or\ B)$$

(在这里, and 用 “·” 表示, or 用 + 表示, not A 用 $\bar{A}$ 表示)

当然，还有一些小技巧，比如如果大部分的结果都是 1，我们可以写关于结果是 0 的 SoP，然后把整个式子求反

有些时候，结果的某些 bit 我们是不关心的，而我们可以自己决定这个 bit 是 0 或 1 来让电路变得更简单

还有一些是不太常见的 gate，如:

- mux: multiplexor. 它算是一种选择器，有 4 个或多个输入口，还有另外一个输入两个 bit 的输入口，它来决定输出这四个口中的哪个结果.
- Decoder: 它有点像 mux，它输入两个 bit，有 4 个输出，而这个两个 bit 的输入决定了这四个输出的哪个口是 1. 如输入 bit 是10，那么第三个端口就会输出 1. Decoder 可以用在 cpu 中用地址信息启用内存块，一个 decoder 有N个输入和2^N输出

ALU

ALU, arithmetic and logic unit 是 CPU 中很重要的一部分，他就是用来执行我们输入的各种指令，如 add, sub, and 等

并且，在构建电路的时候，我们需要把输入接入所有的运算单元，因为电路是静态的，虽然我们只能一条一条的执行指令，但电路都是同时连接的. 最后再用一个 mux 用指令编号来选择结果

之前我们做了加法，可以再来看看减法单元: $A - B = A + -B$，所以我们可以先把 b 取反再加 1，而这个加一也很简单，直接在加法单元的第一个加法器的 carry in bit 设成 1 就行. 这时候我们就可以把加法和减法单元混合在一起:
我们多一个输入，它决定了是加法单元还是减法单元，如果是加法单元，则是 0, mux 会输入 B, 0 也会输入进 carry in bit. 如果是 1, mux 就会选择 B 的反向，然后这个 1 也会输入进 carry in bit

多个这种指令组合在一起，就成为了一个 ALU

数据读取存储

在上面我们都在设计电路图，这些电路图都有独立的输入, A, B 等. 但在 CPU 中，都是存放在寄存器中，我们需要从那里面读取. 这又用到了 Mux: 把所有寄存器连在一起，然后用内存地址选择就行.

Delay

在我们切换输入的时候，因为物理原因，输出会在短时间内变得不太稳定(部分保留着上次执行的结果)，需要等一小会才可以稳定下来. 而每个小逻辑门都会让这个不稳定的时间稍微增加. 每个逻辑门的不稳定时间叫做 gate delay
而输出的不稳定的时间是电路里最长的那个 path 所有逻辑门的不稳定时间累加起来的总时间，这总时间叫做 Circuit Delay

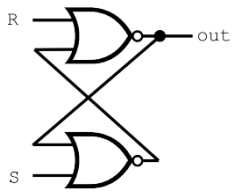
当然，每个小逻辑门虽然延迟有差别，但很细微，所以我们可以不看时间，而是看每个 branch 有多少 gate delay

正因为有 delay 的存在，我们在执行代码的时候可能出问题: 如$add\ \$s0, \$s0, \$t0$，这时候就会在寄存器和 ALU 之间构成了一个回路. 如果输变更，ALU 的 circuit delay 会输出不稳定的信号给寄存器，寄存器又因为不稳定的信号导致传给 ALU 的信号变更，就造成了一个死循环. 我们需要用某种方法把寄存器输出的东西固定一段时间，等我们说可以改变的时候再改变. 这就是 Clock，也就是我们说的时钟频率: 只有时钟发出一次脉冲，寄存器才会通过输入改变数据. 也因为脉冲频率有时间间隔，在下一次输入的时候 ALU 的信号已经稳定了.

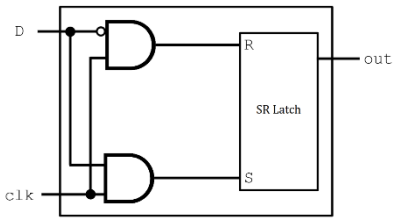
On Chip Memory

我们会来学习 latches 和 flip flops

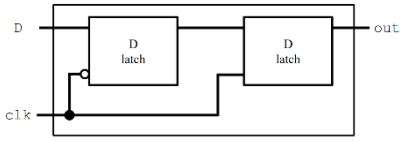
S-R latches 是来控制数据的电路: 他有两个输入 R,S，和一个输出. 如果 S 是 1，那么就会把输出 Set 到 1，如果 R 是 1，那么会把输出 reset 到 0，如果 S 和 R 都是 0，则保持输出的结果. 并且 RS 输入不可以同时为 1.
上面这种叫 volatile memory，因为只要一没电，内存数据直接没了，就像 DRAM 一样，并且它还是没有解决我们的需求: 如果电路信号在波动, S-R latch 也会一起波动



D-Latch 就可以解决这个问题: 它有一个 D(data)输入，还有一个输入是更新信号，如果更新信号是 1，则会更新数据 D，如果更新信号是 0，则忽略数据 D 输入，保持原本数据 D. 我们可以尝试一下: 如果 clk 是 0，则 R 和 S 输入永远是 0，如果是 1，则 R 和 S 取决于 D 的输入



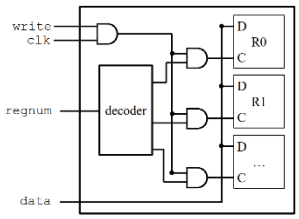
目前，我们似乎解决了问题，但又有了个新的问题: 如果数据刚好在更新信号开始的时候开始变化，这可不是我们想要的数据. 这就引出了 Flip Flop 电路，它是用两个 D-Latch 组成的. 这个电路的特性是只会在在更新信号从 0 变成 1 的瞬间才会捕获数据.



我们来试一下: out 原本储存的是 1，输入数据是 0，且现在更新信号是 0. 因为第一个 D-Latch 有个反向，所以第一个 D-Latch 捕获了输入数据 0. 接着，输入信号变成 1，第一个 D-

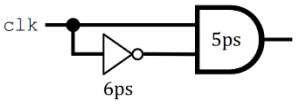
Latch 变成记忆模式，开始输出 0，然后第二个 D-Latch 变成捕获，所以就成功更新了数据(虽然没有变化)
但如果在更新信号是 1 的时候，数据波动了，变成了 1，第二个 D-Latch 还会保持 0，因为虽然第二个 Latch 是捕获状态，但第一个是记忆模式，所以不会把 1 传进来.

之前我么说过读取寄存器，直接全连起来再用 mux 选择就行，但储存就有些意思了：到底该怎么只修改我们指定的寄存器呢？答案就是多用一个 and gate：不仅需要是需要有更新信号，还需要有个 write 信号输入，只有我们想修改这个寄存器的时候，才会是 1. 而这个 write 信号是由 decoder 产生的.



对于寄存器来说，一般一个 cycle 能进行两次读取和一次写入.(这个读取时同时的，如一个 add 的两个 reg 参数)

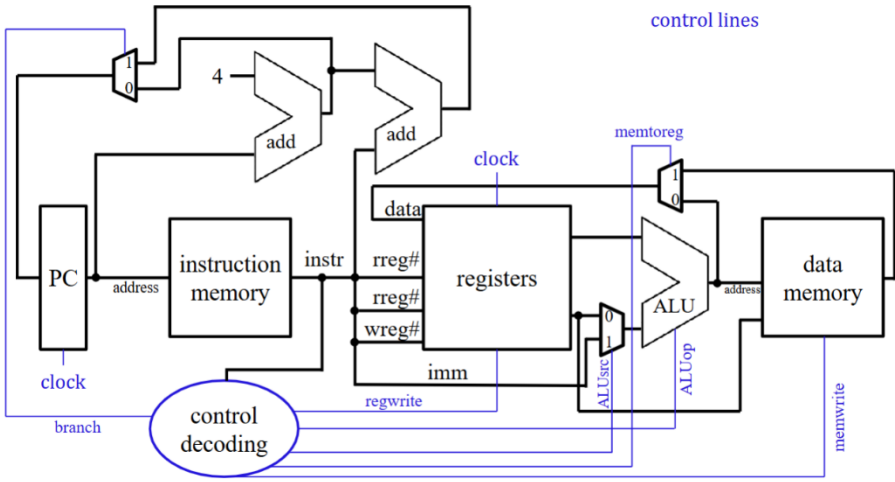
上面的电路看起来已经没有问题了，但还是会有 bug：一个 Cycle 的高频会有持续时间的，也就是说，如果 decoder 在 Cycle(更新信号)是 1 的时候变化了，则会在一个 cycles 中同时存入了两个寄存器(data 在这个时候可能还没更新). 我们需要用某种方法来把更新信号的 1 缩短到尽可能小(或者说是把 asymmetric square wave 变成不对称的).



答案就是用一个 Pulse Generator(右图)
仅靠这两个门的延迟，我们就可以做出一个发出一瞬间信号的频率

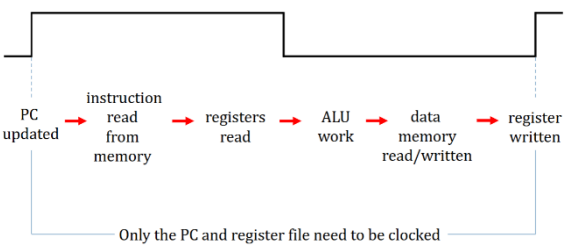
执行指令

对于执行指令，有下面的几个阶段:
1: 读取指令，用 program counter
2: ALU 操作
3: 读取/存储数据 (用 lw, sw)
4: 把数据存回寄存器

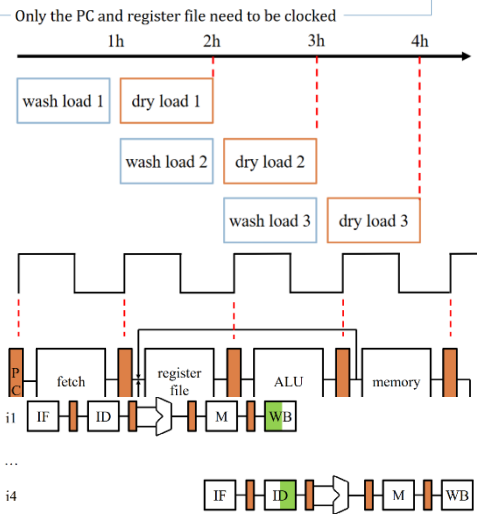


流水线

我们之前学的都是 one cycle CPU，也就是一个 Cycle 可以完成所有的任务. 但其实这样是很不好的: CPU 的频率必须跟着耗时最高的指令来设定.



比如，所有的任务加一起需要消耗 1ns，也就是说这个 CPU 只有 1GHz，但如果 ALU 只需要 200ps，则仅对于 ALU 来说，他能达到 5GHz. 我们可以把这个想象成 5 个不同的机器，他们可以独立工作，工作完以后传数据给下一个机器，所以如果用流水线的思想的话(就像右侧我们洗衣服和烘干衣服)，可以提升大约 5 倍的速度. 当然，我们不是在把每个指令加速，而是把 cpu 的频率加速了.

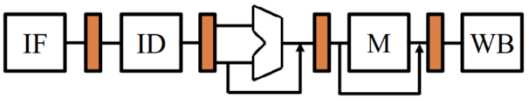


这样问题又来了，这五个机器并不是完全独立的，他们可能互相影响. 这就需要我们再加上多个储存来让数据稳定下来. 并且这些 latches 不仅会储存数据，还会储存指令来告诉下面的单元应该拿这数据干什么

还有一个问题就是解码指令和储存数据用的是同一个单元: 解码指令部分也是涉及到读取单元(同样负责储存). 然而这个问题我们可以不用考虑，因为这个单元工作很快，我们可以认为前半部分负责读取，后半个 cycle 就负责存储

Bypass

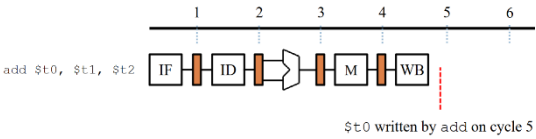
很多指令都用不着所有的单元，如 add，就用不到内存(M)单元，我们会 bypass 它，虽然电路还是在工作，并且时间也没有减少，但我们确实是略过了



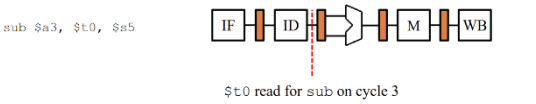
Data Dependency

Data Dependency 表示的就是上一条指令和下一条指令都用了相同的寄存器

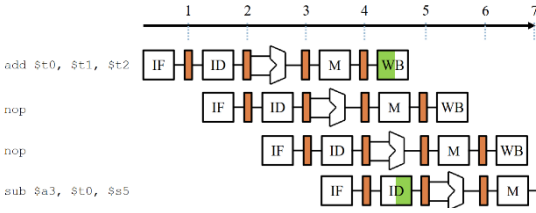
我们现在的都是 5 阶段 CPU 流水线，也就是说一个指令如 add 在 5 个 cycle 之后才会存储，但如果下面一个指令就要这个 add 的结果就会导致问题：我们读到了之前还没有加过的数值 – data hazards



为了解决这个问题，我们需要加上两个 no operation，也叫做 nop. 这个指令啥都不干，只是单纯的延迟.

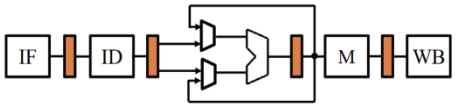


虽然这个办法解决了问题，但还是不是很好的解决办法：它造成了很多延迟，如一个 add 从需要额外两个 cycle 才能做完



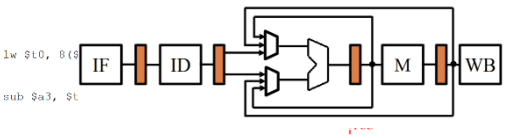
还有一种就是自动的检测数据是否有冲突，如果有冲突，就 hold 这个数据，直到前面指令执行完成. 当然，这也需要让 add 增加两个 cycle 才能完成，跟上面 nop 完全没区别.

但其实，我们是可以让这两个 nop 消失的：再来看上面的图，实际上数据已经在 ALU 之后就算出来了，并且第二行指令的 ALU 确实是在第一个指令 ALU 调用之后. 所以我们可以直接让 ALU 把数据喂给自己(上一条的指令直接传给下一条的指令) . 这种操作就叫 data forwarding



我们再来回顾一下 data hazards：只有当数据在生产(point of production – PoP)前被读取(Point of consumption – PoC)时，才会产生 data hazards，而上面的 Data forwarding 就是加速了 PoP 并延迟了 PoC.

但 data forwarding 不是万能的：如果上一条指令是 load，它是 M 阶段产生数据，但下一条需要在 ALU 就使用，我们无法直接传递给 ALU 了，这就叫做 load-use hazard. 当然，我们不用加多个 nop 来解决，其实只用加一个 nop，然后让上面的指令 M 在下面的指令的 ALU 前面就可以了



Instruction Ordering

之前我们都尝试在硬件解决问题，但编译器也可以对 data hazard 来提供一些帮助：如果三条指令其中两条有数据依赖，编译器会尝试把那个没有 data dependency 的指令夹在两条指令中间，这样就可以避免插入 nop

其他 Hazards

除了 data hazard，还有另外两个:

- Structural Hazard: 两条指令同时使用某个单元，但这个在我们的 CPU 中是不会发生的，因为我们不会允许出现长度小于 5 或大于 5 的指令，所以每个单元都是错开的. 就算需要，这个问题只需要增加更多单元就可以解决了
- Control Hazard: 当在计算 branch 的时候，我们需要加上 2 个额外的 nop，因为 branch 的算法是在 alu 里面执行的，所以结果在 ALU 之后才可以知道. 这就导致我们下条指令只能等 ALU 结果出来后才可开始执行. 不过我们可以额外加一个简单的判断来代替 ALU 在 Instruction Decoder 那里，这样可以节省一个 nop，毕竟时间比那么小小的一个 Adder 要重要

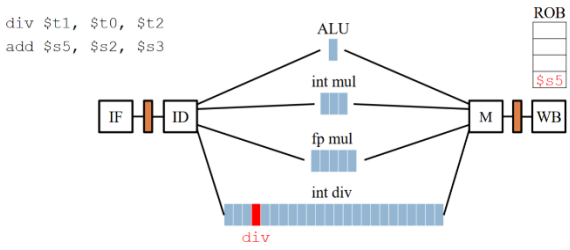
关于这个 Control Hazard，很烦人，因为还是有一个 nop，并且不仅是 branch，jump 啥的也需要用 nop. 我们可以有一种方法来稍微改进一下: 就直接假设所有的 branch 都不会转跳，继续执行下面的代码，如果猜对了，就继续执行，如果猜错了，就把错误的那条指令后面的代码都变成 nop(这是来得及的，因为在 decoder 就能知道对错结果). 用上面这种方法，我们节约了 50% nop 的消耗时间

还有一种方法是用 delayed branch，也就是跟之前的 lw 差不多: 我们在 branch 后面塞一个永远会执行的指令，再进行转跳. 当然这种指令是比较难找到的. 或者，如果一个函数有个 return，可以把最后一行指令塞到 return 后面节省一个 nop

ROB

因为我们把 CPU 变成流水线了，所以频率就上去了(每个小模块需要的时间更少)
但我们仍需要把频率设置成最大的那个模块的延迟.

但问题就来了: ALU 中的加法和除法消耗的时间有很大的差别: 可能加法需要 250 皮秒，但除法可能需要 6000 皮秒. 我们可不想因为一个除法来把频率设置的这么低.



这就引出了 Multicycle Instruction: 不同的指令我们会分配给它不同的 cycle 的数量，并且把除法从 ALU 中拆出来.

就像右边的图一样，div 需要执行很多 cycle，但加法已经做好了. 然而我们不想让 add 先比 div 修改数据，所以就要加个 buffer，来把 add 指令的结果存起来，直到除法做完. 并且如果下一个指令需要上一个指令的结果，那他也需要等着 div 做完.

Issue Buffer

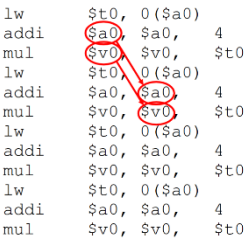
并且，还有一个功能叫做 Out of order execution：当在执行指令的时候，CPU 不仅在看当前的指令，还在看后面的后面几条指令，并放在 issue buffer 中．如果当前指令和下一条有 hazard 的话，CPU 就会尝试去下面的 issue buffer 中找没有 hazard 的指令给移上来

这个还可以跟上面的 ROB 搭配：如果我们要 branch 除法的结果，我们 branch 就要等好久．但我们可以先执行 branch 下面的指令，但不要存储到寄存器里，而是存到某个临时数据里然后在 push 到 ROB 中．直到除法指令执行完，再来进行 branch 判断．如果 branch 需要跳过，我么就把 ROB 那些指令删掉就行

Superscalar

既然上面我们把 ALU 和除法拆开了，当然还有一些其他耗时指令也被拆开了．这就可以让我们同时执行多个指令．但我们仍然得一个一个 fetch 指令，因为前面的两个模块只能一个一个输入指令．这就引出了 Superscalar CPU：它修改了 Instruction fetch 和 instruction decoder 等模块，就可以同时 fetch 多个指令． 但需要知道的是，这个并不是 multi core

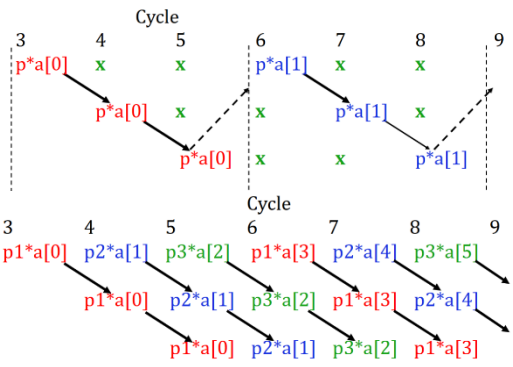
Superscalar & Loop



当 Superscalar 遇上循环，会更加有意思：如下面这个代码：

$$for(int\ i = 0; i < N; i++)\{mul *= arr[i]\}$$

假设我们的 CPU 可以同时输入多个指令，并可以同时执行多个加法，但乘法 ALU 需要 3 个循环(一共 7 循环)，最终算下来就是一个 loop 会需要 3 个循环(不考虑 branch，且乘法 3cycle 刚结束就可以给下个乘法指令)



但这样还是不太好，因为可以看到绿叉的地方都是浪费的地方．然而硬件上就只能这样，但我们可以用一些技巧来更上一层楼：
也就是把一个乘法拆成多个：我们呢直接把循环一次+2，然后创建俩变量，同时记录$mul *= arr[i]$，$mul1 *= arr[i+1]$
或者可以拆成三个，因为一个 mul 需要 3 个 cycle 才可以执行完，并且它是支持 pipeline 的

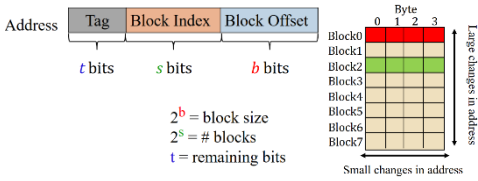
其实，这种操作一般编译器不会做，因为它会产生更多的指令．

Memory

CPU 上的储存是 SRAM，在内存上的是 DRAM，DRAM 上的访问需要一两百的 cycles，但 SRAM 很快，也很昂贵. 为了应对这个方法，我们就有了缓存：在 CPU 上专门有个地方是缓存，这个虽然比 DRAM 小很多，但还是有很多 MB 的，他会和 DRAM 进行交互，然后获取一堆数据，一般来说是 64Byte. 计算机会把内存分成很多小块，每次访问就会读取这一整个小块

CPU 如果访问数据的时候，会优先读取这个缓存里的. 而且设计电路的还会用一些原则进行优化：如 CPU 更有可能访问刚访问过的数据，或者跟这个数据放在一起的元素，而且，这种行为大部分情况下是无法通过程序操作的，如lw_{DRAM}或者lw_{CACHE}. 他们只有最普通的lw指令

当 CPU 把数据放在缓存中，可不是乱放的，他是按照一定的规律：有点像是 hash table，我们把地址转换成 cache 的地址. 比如我们想把$0x13$放入 cache，转换成二进制就是00010011，所以就是 block 4，offset 3：看右侧，它这个二进制读取的格式是按照 cache 大小来读取的：因为一个 block 只有 4 个 offset，所以 offset 只有两个 bit，有 8 个 block，就有 3 个 bit 区分 block



而且地址比 cache 大，而地址可能剩下的部分不一样，但 block index 和 offset 是一样的，我们就可以用 tag 来区分. 如果访问的地址在 cache 中有，就叫 hit，如果访问的地址在 cache 中没有或者是错误的，就叫 miss，$hit\ rate = \frac{hit\ time}{total\ time}$

Miss 也分为很多种：

- Compulsory miss (cold miss) 当第一次读取数据，永远都是 miss
- Capacity miss 当数据两次读取中间隔了很多其他数据的使用就会发生这种事情
- Conflict miss 当一行存储了过多的 block，数据被覆盖了

这就引出了我们的 hit rate：这个概率就描述了缓存的效率，一般来说，遍历一个 array 可以获得最高的 hit rate，因为每个 block line 只有一个 miss，剩下来的都是 hit.
但问题就是如果遍历 array 的时候，我们的步进是一个 block size，这不就纯纯找茬吗：hit rate 就会变成 0，或者一个 for loop 里访问两个地址相等但 tag 不同的 array，这也会导致 hit rate 变成 0. 这种情况就是 thrashing

为了解决 thrashing，我们可以重写程序，或者可以增大 cache. 但其实后者就算增大了，也是一样会有 thrashing 的出现.
还有一种方法就是 mapping：首先我们要知道每一行叫做 set，并且我们上面的 cache 是 direct map，我们无法改变地址的映射. 而高级的 cache 是 set associativity cache：它在一行

(一个 set)有两个 block. 但这个问题就会导致，如果我们有两个地址相同但 tag 不同的数据放进去的时候，一个 set 虽然可以同时存下他们，但我们并不知道到底应该把哪个数据放到哪个 block 中: 一般都是经行很多模拟，来决定放的方法.
比如有三个 tag 不同但地址相同的 array 在 for 中遍历. 第一个和第二个占用了一个 set 的两个 block，第三个 array 在读取的时候，我们可以放在第二个 block 中，这样至少在读取下个循环的第一个 array 的时候是 hit 了. 或者可以随机选一个 block 替换掉，或者可以把最近访问的保留(LRU)… 有很多种方法.
如果选择 LRU，我们就需要额外的 bit 来追踪每个 block 访问的次序，也就增大了我们 cache 的开销

或者，干脆直接直接把所有的 block 放在一行，叫做 fully associative cache. 当然它用在比较特殊的地方.

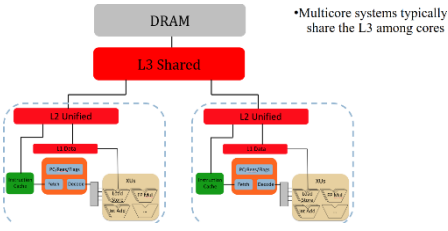
而且 cache 还会有个 valid bit，防止启动以后，cache 里是垃圾数据，或者切换程序导致数据变化.

总的来说，缓存就是让我们在空间和速度上面做取舍，这就是为什么寄存器之后有 L1, L2, L3 缓存，他们越来越大也越来越慢. 而 L3 后面就是 DRAM. 这种分级的缓存就叫 Hierarchy memory

在程序写入数据的时候，可能会发生 write hit: 如果要修改的数据在缓存中，则会先修改缓存的数据，然后缓存再会去 main memory 慢慢修改 main memory 的数据. 这个写两次数据的步骤叫做 write through. 虽然这样不会消耗时间，但会消耗 DRAM 的 buffer.

这就会有另外一个叫 write back 的方法: 如果修改了数据，我们会在 cache 里找数据，如果有，则只修改 cache 的数据，并且不修改 DRAM 的数据. 直到我们读取新数据并且准备替换掉 cache 的数据的时候，才会检查这个 cache 有没有被修改过，如果修改过，则把它 write back 到 DRAM 中. 至于如何记录有没有修改过，我们可以加一个 tag 叫 dirty 在 cache 中，如果修改过就把它改成 1

并且，在写入的时候，如果发生了 hit miss，我们也会把 DRAM 的数据读到 cache 中，就算我们只修改几个 byte.



上面说的都是 data cache，对于 instruction cache，就不太一样了，因为指令大部分时候都是在读，很少写. 指令缓存和 L1 是分开的，而指令缓存和 L1 共享 L2 缓存，而每个核心的 L2 共享同一个 L3 缓存

Cache Coherence

而这就会引出问题: Cache Coherence，比如看上面的 L3 缓存，如果用 write back 策略，如果 core 1 写了数据，但 core 2 读取，因为 Core 1 的数据还在 L1 或者 L2 呆着，没有写回共享内存，就导致 core 2 读取到旧的数据. 这就是没有 Cache Coherence 的问题. 为了解决这个问

题，现代 CPU 实现了一些协议来避免这种问题出现，而之前都是让程序员自己解决的

这个协议大多都基于 Snooping：这里来看 MSI 协议.
对于每个 cache，每行都追踪 M S I 三个数据
- M, modified：如果这个 cache 被修改了，则说明其他 cache 都可能没有一个正确的拷贝
- S, shared：表示了这个拷贝是个正确的拷贝，没有任何其他的 Cache 有对应的 M(修改过)的数据
- I, invalid：这个数据是错误的，说明其他的缓存是 M 或 S 状态

设计 CPU 的时候，把所有的核心的 Cache 用一个总线(bus)相连，然后一起连接最上级缓存(如 L3)

如果 Core1 想访问 x，则会发送消息到总线，内存会回复数据(如果这个数据没有在其他核心中) 然后把它存入 Core1，然后记录为 S 状态.
接着 Core 2 也想读取 x，他也请求到后写入到自己的核心(如 L2)缓存，然后记录为 S 状态
然后，Core 1 修改了 x 数据，他会发送数据到总线并把自己的数据设成 M 状态，其他的 cache 就会把 x 变成 I 状态. 但由于 Write back 策略，数据还在 Core1 自己的缓存中，没有写入三级缓存. 如果其他 core 想要 x 的数据，则就会让 Core 1 开始直接 write back，然后把数据从 M 降级为 S. 接着想要数据的 Core 就会读取新写入的新数据

需要注意的是，上面这种虽然解决了 Cache Incoherence，但它并不会解决 software race condition. 仍然需要在写代码的时候加上 lock (lock 是锁住 bus 不让其他线程去读写)

上面这种 MSI 协议会让总线负担很大，导致读写速度进一步变慢. 可以用 MESI，也就是多了一个 E 的信息. 也就是在修改数据的时候，并不会发送修改的信息，直到其他核心请求的时候，才会去同步.

Snooping 协议对于大约 8 核以下的 CPU 都有余力应对，如果更多的核心就需要 Directory-based 协议

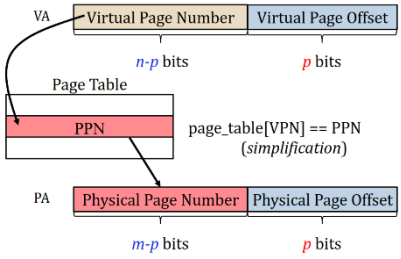
Virtual Memory

在内存布局那里我们提到过内存的格式：但有一个问题就是，如果多个程序都一起运行在一个电脑上，那内存不久互相篡改了吗?
这就引出了虚拟内存：大部分程序虽然得到的是这个内存布局，但这整个内存都是虚拟出来的，也就是说所有程序都是有一块自己的独立的内存. 所以程序使用的那些地址也是虚拟的 – 都是对于虚拟内存的地址. 那些虚拟指令都会被一个叫 MMU (memory management unit) 来转译成真正的物理地址，再访问实体的内存. 程序无法得到那些真正的物理地址.

不同程序的地址可能有不同的虚拟地址，这些虚拟地址可能映射到同一块内存，或者不同的一块内存上，甚至虚拟地址还能映射到硬盘上面，以防所有程序把所有的内存给用光了

为了 cache 读取更快，肯定不能随机把每个 bit 都映射到不同的地方，真正的虚拟内存一般都是有一个个的分页，每个分页大概 4kb. 每个分页会被映射到不同的内存的上面，虽然在虚拟内存上是连续的，但是在硬件的内存里面是分散的一个个的 4kb 的分页

对于现代 x64 处理器，用的都是 48bit 的虚拟内存地址. 这可以覆盖 256TB 大小的内存. 因为这已经足够了，根本不需要用完全部的 64bit 的内存地址.



如果想把虚拟地址 VA 映射到实体内存地址 PA 上，就是右边这种计算过程. 有点像是 cache lookup 的过程. 其中的 Page table 就是放在内存布局里面的 Kernel Memory 部分(OS). OS 占了虚拟地址的一半，大概是 128TB.

Page Table 的格式是这样的:

VPN	PPN	Valid	W	X
虚拟内存	实体内存	1 在 DRAM,0 在 Disk	写的权限	执行的权限
0x123	0xab9	1	0	1

如果我们尝试写一个不允许写的内存，或者执行不允许执行的内存，就会产生 segmentation fault. 这个非常常见，因为这就是空指针产生的原因：尝试读取 0 的地址内存. 当发生这个错误, OS 就会终止这个程序.
如果读取不在 DRAM 而是在 disk 上的内存，就会产生 page fault. 内存会把数据从 disk swap DRAM 中，然后再让程序经行读取.

并行

之前，我们在编程中(类似 CS3500)学过并行，那个是叫 task level parallelism. 因为它是利用多核 CPU，用多个线程同时完成多个任务. 而今天我们要来看的 Data level parallelism 不太一样

Data Level Parallelism

最简单的一个例子是 SIMD – Single Instruction Multiple Data: 也就是一个指令修改多个数. 比如我们有两个向量，里面有多个数字，但用一个 add 指令，就能把两个向量中的每个元素相加起来得到一个新的向量，就是数据级别的并行.

有了这个骚操作，就可以让计算快好多倍：比如计算灰度，需要把 RGB 值加一起，然后除以三．像以前，就只能遍历每个像素，然后取 RGB，但用向量，可以一下遍历多个，比如 4 个像素，然后把 4 个 RGB 值同时存进一个 4 维向量中．最后再除以一个全是 3 的向量．这样就使计算快了大约 4 倍

对于现代的 CPU，很多都可以做 8-way 单精度浮点数据并行或者 4-way 双精度浮点数据并行．而 GPU 通常可以做 32-way 的数据并行(尽管这个已经可以不叫 SIMD 了，而是叫 SIMT).

至于为什么想数据并行而不是任务并行(如可以同时加 32 数据，而不是搞 32 个核心同时加一个数据)，这是因为我们只想让数据处理的部分复制，如果搞很多的数据，流水线的其他部分也会被复制，如 IF, ID⋯.

而且 SIMD 无法做一些有分支的任务，比如有个 if 分支，不同的数据会走不同的分支.

在绝大多数情况下，编译器都不会去主动使用 SIMD，就算做了，带来的提升也不是很大.

Single Instruction Multiple Thread

GPU 使用的就是 SIMT，本质上就是一堆线程，每个线程都是 32-wide SIMD．在某个时间，并不是所有的线程都在工作, GPU 核心会找到闲置的线程，然后使用它.

Simultaneous multithreading

通常发生的情况是，我们有多个 Add 的单元，可以同时执行多个 add，但问题是这个线程上没有那么多的 add 需求．就浪费算力了．然而有一种技术，就是这个 Simultaneous multithreading，或者被因特尔叫做 hyper threading，这种技术就是去其他线程中找有没有多出来的指令去求，然后去完成那些指令.

Task Level Parallelism

这种并行就是我们更熟悉的概念：同时执行多个任务的时候就会使用 task level parallelism. 我们有多种方法可以实现这种功能：比如可以是双 CPU 主板，或者更常见的就是多核 CPU.

SMT

上面说了，每个核心能完成一个线程，但打开任务管理器，会发现线程数量是核心数的两倍. 这个是因为 SMT 导致的, SMT 可以让一个核心同时处理两个线程.

瓶颈

上面说了那么多的优化，然而 CPU 还是没有提升多少，其实有很多的原因是内存的锅：内存太慢了，以至于很多时候, CPU 的任务早就做完了，就在那里等内存来传输数据

Race Condition

Race condition 是表示当两个线程同时访问一个寄存器会出现的情况. 请注意，这个跟 cache coherence 不同，因为这是在修改不同线程的寄存器，只有在 sw 指令后才会修改缓存. 而寄存器是每个线程里自己的东西.

Critical Section 表示的就是那些可能会出现 Race condition 的地方，我们需要做出一些保护措施如加线程锁. 代码中加锁了之后, CPU 就会产生一个特殊指令，去避免 critical section

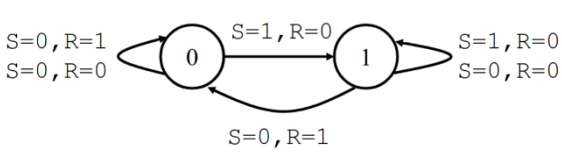
而且，不仅加锁，还要加个 fence. 这个 fence 就是会阻挡 cpu 读取 fence 后面的指令，这样就会不会让编译器/CPU 重新排序指令导致 lock 失效了

State Machine

状态机(SM)就是一种机器，他接受并输出数据，但会根据数据切换状态. 由于这个状态的存在，不同的状态也会导致不同的结果输出

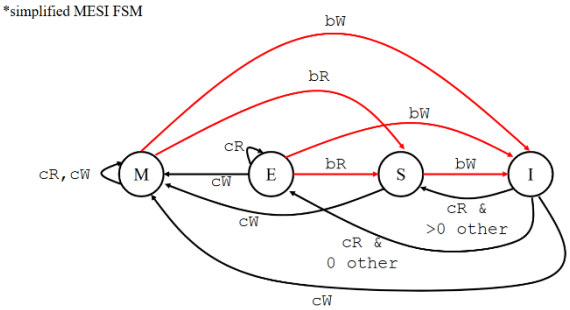
比如，就可以把刷卡门禁抽象成状态机：有两种状态，开门和锁门，如果刷卡，就会从锁门变成开门，在开门状态下刷卡还是从开门变成开门状态，如果有人通过，状态就会变成锁门.

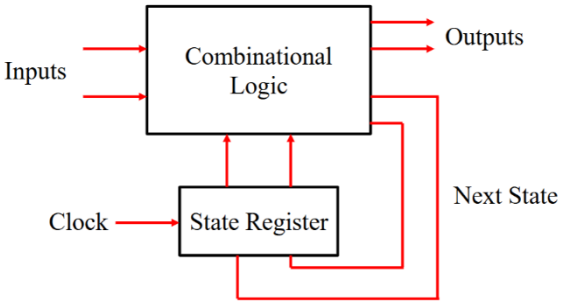
像之前的 SR latch，也可以抽象成 FSM – 有限状态机. 他有 0 和 1 的状态，会根据 S 和 R 的数据来修改他的数据.



更复杂的是上面提到的 MESI，也同样可以抽象成 FSM

至于如何搭建一个 FSM，其实 SR latch 就是 FSM – 也就是可以仅仅用门来搭建出 FSM. 更复杂的 FSM，可以用下面这个格式来搭建：本质上就是一堆逻辑门，和一个记录当前状态的 Register，然后用那些逻辑门来修改当前的状态





软件

数字在二进制中的表示

在二进制 int 中的表达负数，可能第一时间想到的是在第一个（最大的那位）bit 表示正负：这样很直接但也带来了一些问题：如果想表示 0，那么就会有正 0 和负 0，并且如果这样表达还会带来算法的不一样：一正一负相加就成了两个负数相加-因为负数的那个 bit 影响了结果. 如果我们想解决这种问题就得设计新的电路来专门算减法，这样太麻烦了.

还有一种方法来记录，在知道这种方法之前，我们需要知道一个特性：对于二进制来说，最大的那个 bit 永远比之前所有 bit 加起来还要多 1：比如四个 bit 最高能表示 0-15，最大的那个 bit 就能表是 0 或者 8. 而前面的三个 bit 表示的是 0-7. 根据这个特性，我们可以让最大的那个 bit 等于负数，再让剩下的 bit 表示正数，最终结果就是正数和负数相加的数. 如 4 个 bit 就能表示-8~7 因为当1000就是-8，当0111就是 7，1111就是-1.
这种方法不仅可以表示正负，还可以让正数和负数相加了

如果我们想把一个数从正数变成负数，也是比较简单的：我们只需要把正数的所有的 bit 反转，再加上 1 就行，如$3 = 0011, -3 = 1101$
通过这种方法，就可以做出减法了：$a - b = a + (-b)$

我们可以发现，当两个负数相加，最大 bit 就会有两个 1，这样就会造成了 over flow. 如一个字节是 8 个 bit，表示最大值是 255，当再加 1 就变成了 0. 对于带正负值的 int，它的 overflow 是 127 变成-128（对于一个 byte 大小）

那带小数的数字呢，它该怎么存进去？

其实跟十进制的思想是差不多的：对于十进制，在小数点后面的分别是$\frac{1}{10}, \frac{1}{10^2}$… 二进制也是

如此，在二进制小数点后面也是$\frac{1}{2}, \frac{1}{2^2}, \frac{1}{2^3}$… 比如在二进制中$11.011 = 3.375$

具体的计算方法是：如果我们想知道小数点后第一位是否是 1，就把小数点的数*2，如果得到的数≥ 1，则说明这个小数点的数是 1. 反之亦然. 比如 0.2，可以得到前两位是00， 此时 0.2 已经变成了 0.8，下面*2 就变成了 1.6，这时候就把个位去掉留 0.6 接着之前的操作. 但我们会发现$0.6 * 2 = 1.2$，就成了循环，这就是有些数二进制永远无法准确表达

但内存是有限的：我们需要考虑小数点到底在哪：如果在太靠左，就不能表示太大的整数部分，但如果靠右，又不能表示太精确的小数部分. 这就引入了浮点数：小数点是可变动的：
在浮点数中 bits 被分为了 3 部分：

- Sign bit
- Fraction
- Exponent

$$(-1)^{sign} * fraction * 2^{exponent}$$

这就像科学计数法一样-5.4×10^6， 前面是正负号和整数和小数，后面是决定数的大小. 二进制浮点也是一样$2^{exponent}$就是负责修改小数点的，在 IEEE 标准中，对于一个单精度浮点（4byte, 32bit）中里面有 8bit 负责 exponent， 1bit 负责正负号，剩下的 23bits 是负责

Fraction

想要转换成浮点，我们需要先 Normalize 这个小数，拿1101.01举例，我们需要把1101.01变成1.10101×2^3，$0.011 = 1.0 \times 2^{-2}$. 至于为什么要 Normalize 数，是因为这样我们可以省下来一个 bit：在小数点前面永远只有一个 1.

最后完整的公式就是

$$(-1)^{sign} * (1 + fraction) * 2^{exponent-127}$$

如-3.5，他的二进制是 11.10，归一化是1.110所以小数部分就是 11，我们把小数点往前移了 1 位，所以 exp 应该是 128（128-127=1）这个 127 就是 bias，求得公式是$2^{k-1} - 1$这个 k 就是 exp 的 bit 数量，这里是 8，所以得 127

最终存储在数据中是

$$1\ 10000000\ 11000000000000000000000$$

而在浮点中，全部是 0 表示的就是 0，这是个特殊情况，公式并不适用

下面是浮点数的一些特殊情况

Exponent	Fraction	Value	Example
0	0	$+/-0$	0
0	Non zero	$+/-0$ renormalized number	Under flow 数字太小了
255	0	Inf	overflow
255	Non zero	NaN	$\sqrt{-1}$

汇编：

指令

指令可以分为下面这三种

- Logic – 加减法
- Datta – 操纵内存
- Control – 从某行代码跳到另一行

下面用 MIPS 举例：

Operation	Instruction	Meaning
Addition	$add\ a,\ b,\ c$	$a = b + c$
Subtraction	$sub\ a,\ b,\ c$	$a = b - c$

这种指令一次只能加一个元素，不可以$x = y + z + w$

如在 JAVA 中一行指令：$a = x - y + z$

换成 MIPS 汇编指令就是$sub\ a,\ x,\ y\ add\ a,\ a,\ z$

或者在 MIPS 中两行汇编是$add\ t,\ x,\ x$　　$add\ x,\ x,\ t$
变成高级语言就是$x = 2x + x$或者$x = 3x$ 可以看到，在汇编中，我们需要产生一个中间变量，不然就无法把一个数乘以 3

- 机器语言：
  表示的就是计算机处理数据时默认的 bit 大小. 如 MIPS 的机器语言就是 32 位的，也就是里面的数据默认是 32bit. 这个机器语言很重要的一点就是影响了内存大小，如 32bit 的地址，只有 4GB 大小，而 64 位的地址能存储 16PB 的内存地址

Register

在 CPU 中，有这三个模块：
- Register 是在芯片上的一个高速储存器， 用来存放临时数据和指令
- PC（Program Counter）一个特殊的 Register，记录了指令执行到内存的哪里了
- Functional Unit 计算电路，如加法电路

在 MIPS 中，有 32 个寄存器，其中$\$t0$~$\$t9$是临时的，$\$s0$~$\$s7$是保存的 Registers，主要是为了存放变量，$\$zeor$是永远存放 0 的一个 register.

当我们进行计算的时候，如果给一个 register 附上新值，那被赋值的旧值就永远消失了，除非把它放到内存中.
如:

$$add\ \$t0,\quad \$t1,\quad \$t2$$

在执行完后，旧值就永远消失了.

- Immediate Operator: 我们也可以直接把一个数加到寄存器中， 也就是硬编码进去，不过需要用$subi$或$addi$指令， 这个指令里必须出现一个寄存器

$$subi\ x, b, 9$$

Memory

计算机中所有类型都需要大小的，如 int 通常占用 4bytes，而 array 的大小是

$$array\ Size = element\ size * element\ num$$

当我们想访问数组中的一个元素时，就可以用$base\ Address + i * element_size$
这个基址就是数组中的第一个元素的地址

如果我们需要操作内存，就需要lw, sw指令，分别是 load word 和 store word.

$$lw\ dest,\quad offset(base)$$

$$lw\ \$s1,\quad 8(\$t0)$$

这个代码的意思是数据在在基址$st0偏移量 8 中，取出数据再存入$s1中
并且地址其实也是一串数字，我们可以直接把地址$s1加上 4 转到下一个 int（假设这数据都是 int）
例:
$arr[4]+= 1$， 假设这是个 int 数组，且$s3储存着数组的基址， 写出这行代码对应的汇编代码

$$lw\ \$t0,\quad 16(\$s3)$$
$$addi\ \$t0,\quad \$t0,\quad 1$$
$$sw\ \$t0,\quad 16(\$s3)$$

需要注意的是， 这种指令只能是$int(\$sx)$， 也就是前面是数字， 括号里面是基址. 我们不可以交换位置或者是把数字换成寄存器， 如果想用寄存器的值进行偏移， 就需要让基址和偏移进行相加， 然后数字那里写个 0 就行

Load byte 也是一个常见的指令，如果 array 中存的是 char，就可以这么写↓

$$lb\ \$t2,\quad 0(\$s2)$$
$$lb\ \$t3,\quad 1(\$s2)$$

Data & Text Segments

Data segment 是表示给变量分配内存的汇编代码， 如

$$x:$$
$$.word\ 65$$
$$str:$$
$$.asciiz\ "hello!"$$

x就是告诉编译器这段内存地址就代表x， 可以把他当作变量， 然后里面存的数据是 4 bytes 的 int(.word 表示)， 初始化为 65.
然后下一段内存叫做 str， 初始化为"$hello!$"

而 text segment 就记录了我们所有汇编的代码. 如 lw， add 等. 我们也可以在 text segment 中使用 data segment 里创建的变量， 不过这就需要用到lw指令了， 因为我们是把它创建在内存当中的， 不能像在寄存器那样直接使用. 并且， 在使用lw之前， 还需要得到这个变量的地址， 就需要la指令， 也就是 load address， 用上面的 x 指令， 假设我们想把 x 的地址存到$s0
中:

$$la\ \$s0,\quad x$$

Bitwise operator

这种操作符很像编程中的逻辑:

$$0|1 = 1$$
$$0\&1 = 0$$
$$1\&1 = 1$$
$$\sim(0\ 0\ 1\ 1) = 1\ 1\ 0\ 0$$

其中需要注意的是 Exclusive Or: 也就是它值允许当其中一个是 true 的时候才是 true, 如$01, 10$, 而$00, 11$会返回 0, 如果我们用$xor\ b,\ 1$, 得到的会是$\sim b$

其中一个应用就是求单双: 如果我们在汇编中使用%是非常昂贵的, 但如果我们用 and 指令, 就可以直接得到单双: 如 5 的二进制是101, 让他 and 1 的话得到的就是 1. 通过最后的结果就可以知道是单还是双.

还有就是 masking: 我们可以把一个数的某些 bit 给提取出来: 只需要和我们想要提取的地方做 and 计算就可以. 或者我们可以用 or 把某些地方全都设置成 1

Bitwise Shifting

我们还可以通过某些方法把所有 bit 向左向右偏移: 这种还有区别, 因为我们要考虑有没有正负号, 当我们把所有 bit 向右移动, 我们要考虑要不要把左边新出来的数始终设为 1(如果原始是 1) , 如果我们向左移动, 我们不需要考虑这个事情. (如果考虑就是 algorithm)

具体的应用就是能快速的计算出一个数乘上2^N或除以2^N后的结果
Shift left logical: $sll\ a,\ b,\ shamt$
Shift right algorithm: $sra\ a,\ b, shamt$
Shift right logical: $srl\ a, b, shamt$

另外的应用就是把大数放到寄存器中. 如 immediate 指令, 它允许的数只有 16 位大小. 如果我们想往一个 32 位数据中存放一个 32bit 大小的数, 就会遇到问题. 我们可以用 bit shift 来解决这个问题:

$$addi\ \$t0,\quad \$zero,\quad 0xAABB$$
$$sll\ \$t0\ \$t0,\quad 16$$
$$ori\ \$t0,\quad 0xCCDD$$

通过这样的方法, 就可以得到$0xAABBCCDD$. 因为这个操作非常频繁, 有一个指令能一步干前两个指令:

$$lui\ \$t0,\quad 0xAABB\quad \#\ \$t0 = 0xAABB0000$$

Pseudo Instructions

这个是一种伪指令， 当我们写在程序中后， 汇编程序会把它转换成真正的指令.
如 li， la 指令

Operation	Instruction	Meaning
Load immediate	$li\ a, imm$	$a = imm$
Load address	$la\ a, label$	$a = address\ of\ label$
Move	$move\ a, b$	$a = b$
Not	$not\ a, b$	$a = \sim b$
Subtract with immediate	$subi\ a, b, imm$	$a = b - imm$

System Call

有很多指令， 并不是代码自身能做到的， 需要系统层面的帮助: 如 new object， 监听键盘等.
他就是一个函数$syscall$， 直接写在汇编代码中， 这行会调用系统的函数， 来干一些事情, 然后返回一些值.
只写一个词系统肯定不知道干什么， 我们就需要给他传递 id:
如 print int 指令在 MARS 虚拟 OS 中 id 是 1.
我们需要把 id 的值放在寄存器$\$v0$中， 把函数参数放在$\$a0$~$\$a3$当中

$$
\begin{aligned}
&li\ \$v0, \quad 1\ \#\ procedure\ ID\ 1: print\ an\ int\\
&li\ \$a0, \quad -5\ \#\ the\ value\ to\ print\ (-5)\\
&syscall
\end{aligned}
$$

这个就是让 system 输出-5

Control Instruction

通过控制指令， 可以做到 if else 效果

Operation	Instruction	Meaning
Branch on equal	$beq\ a, b, label$	$if(a == b) goto\ label$
Branch on not equal	$bne\ a, b, lable$	$if(a != b) goto\ label$
Set less than	$slt\ a, b, c$	$if(b < c) a = 1;$ $else\ a = 0;$

Label 告诉 CPU 这些指令到底在哪:

$$label1:$$

$xxxx$

$lable2$:

$xxxx$

不仅可以用 if， 我们可以直接只用j指令(jump)来直接转跳到某些地方

JAVA	MIPS
$if(x == y)x++;$ $y = 10;$	$bne\ \$s0, \$s1,\quad skip$ $addi\ \$s0,\quad \$s0, 1$ $skip$: $li\ \$s1,\quad 10$

可以看到， 在汇编中， 我们一般都是使用高级语言中相反的指令(如 not equal)， 因为在高级语言中我们想要的是:当 xx 的时候执行， 而在汇编中我们需要的是:当不是 xx 的时候跳转，并且上面这种情况是没办法调换指令位置， 因为当 x=y 的时候， 两条指令都会执行， 而指令之间可能需要有先后顺序

JAVA	MIPS
$if(x == y)x++;$ $else\ y = 10;$	$bne\ \$s0, \$s1,\quad skip$ $addi\ \$s0,\quad \$s0, 1$ $j\ done$ $skip$: $li\ \$s1,\quad 10$ $done$:

JAVA	MIPS
$while(x != y)\{$ $x++;$ $\}$	$start$: $beq\ \$s0,\quad \$s1,\quad done$ $addi\ \$s0,\quad \$s0,\quad 1$ $j\ start$ $done$:

JAVA	MIPS
$for(x = 0; x != y; x++)$ $statement;$	$li\ \$s0,\quad 0$ $start$: $beq\ \$s0,\quad \$s1,\quad done$ $statement$ $addi\ \$s0,\quad \$s0,\quad 1$ $j\ start$ $done$:

JAVA	MIPS
$if(x < y)x++;$ $y = 10;$	slt $t0, $s0, $s1 beq $t0, $zero, skip addi $s0, $s0, 1 skip: li $s1, 10

需要注意的是， 我们只有一个小于操作符， 所以其他的都需要进行变换:

Operation	Equivalent Using <
x > y	y < x
x >= y	!(x < y)
x <= y	!(y < x)

而且需要注意的是， 在汇编中， true 是0001， false 是0000， 所以不能简单的用 not true 表示 false. 需要用的是 xor， 也就是0001^1 = 0000

Operation	Equivalent Using <	MIPS
a = x > y	a = y < x	slt a, y, x
a = x >= y	a = !(x < y)	slt a, x, y xori a, a, 1
a = x <= y	a = !(y < x)	slt a, y, x xori a, a, 1

Operation	Equivalent Using <	MIPS
if(x > y)	if(y < x)	slt a, y, x beq a, 0, skip
if(x >= y)	if(!(x < y))	slt a, x, y bne a, 0, skip
if(x <= y)	if(!(y < x))	slt a, y, x bne a, 0, skip

Function

如果汇编想实现函数的功能: 开始执行某段代码, 再返回一个数值, 就说明我们举要 jump 指令, 还需要一些可以存储数值的地方. 但问题就来了: jump 目前只能 jump 到 label, 也就是说我们如果想把函数返回的 jump 地址硬编码进代码, 那么函数只能用一次.

这就用到了 return address: 我们之前说过汇编代码也是存在内存中的, 所以也有对应的地址. 而有一个寄存器专门储存了一个 function call 的下一个代码的地址. 然后我们用$\$ra$就可以得到这个代码的地址

然而不仅仅 jump 时候 func call, 何况要是函数里面也有 jump 那就难受了. 所以我们要告诉电脑什么时候才是执行了 function: $jal\ label$ (Jump and link), 这个是等价于$j\ label$的, 并且它还会去把下一行的代码地址存放到$\$ra$中. 想要使用这个地址, 那就需要$jr\ \$ra$指令

这时候如果在函数中调用其他函数, 那么就会有问题: $\$ra$会被覆写掉, 但这个不是很大的问题, 如果需要调用其他函数, 我们就需要手动记录$\$ra$

接下来该解决的问题就是返回值了: 我们在调用函数的时候会把函数返回值存在不同的地方. 这就需要一些统一的规则不造成混乱:

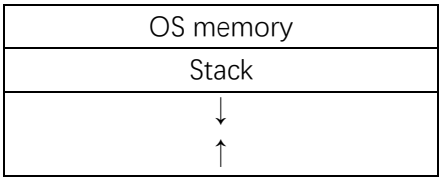
- 如果我们调用函数传递参数的时候, 前四个参数会被放进$\$a0$~$\$sa3$, 多出来的函数会放在内存中
- 而函数的返回值会到$\$v0$寄存器中.

这就是为什么我们可以分开编译函数和代码, 因为他们根本就没有关联, 他们有一个统一的规则来讲参数和返回值到底放哪里.

但这个又会导致问题: 如果函数和 main 函数都修改了$\$s0$, 但寄存器只有一个$\$s0$寄存器, 这就会出现数据覆写. 这个问题就可以用 stack 解决

内存布局

编译代码的第一行会存在$0x0040000$, 也就是 heap 内存的下面
Heap 和 stack 的内存增长方向是相对的
并且 stack 的增长方向才是 stack 的顶部, 但是对于整体来说, 代码越新, 地址越小



Heap
Global variable, machine code

会有一个特殊的寄存器，寄存着 stack 顶部的地址，叫$sp，也就是 stack pointer，所以我们给栈增加空间就是给$sp加一个负数
比如一个函数需要 50 个 char，6 个 int 的变量，我们就至少需要给 stack 分配 74byte 的空间

并且用 stack 还能解决上面的数据覆写问题. 如果两个寄存器同时被调用者函数和被调用的函数使用，那么调用者会在调用前冲突的寄存器存到栈上，在函数执行后取出.
在被调用函数执行的时候，也需要存一边冲突的数据，然后再函数结尾再 load 出来数据
在 MIPS 对应的指令叫sw和lw，对应的基址就是栈顶指针

Name	Use	Type
$\$v0 - \$v1$	Function results	Temporary (caller)
$\$a0 - \$a3$	Arguments	Temporary (caller)
$\$t0 - \$t9$	Temporaries	Temporary (caller)
$\$s0 - \$s7$	Saved register	Preserved (callee)
$\$gp$	Global pointer	Preserved (callee)
$\$sp$	Stack pointer	Preserved (callee)
$\$fp$	Frame pointer	Preserved (callee)
$\$ra$	Return Address	Preserved (callee)

对于不同的数据，储存的任务也是不同的代码干的，如函数参数，temp 寄存器都是调用函数的代码来存储，剩下来的寄存器都是被调用的函数来存储. 当然，所有的代码都应该至少被算作 callee，就算是 main.
并且可以看到，在 callee 中，我们也要存储$\$ra$，这也就自然解决了调用函数的问题

```
foo() {
  int p = 22;
}
```

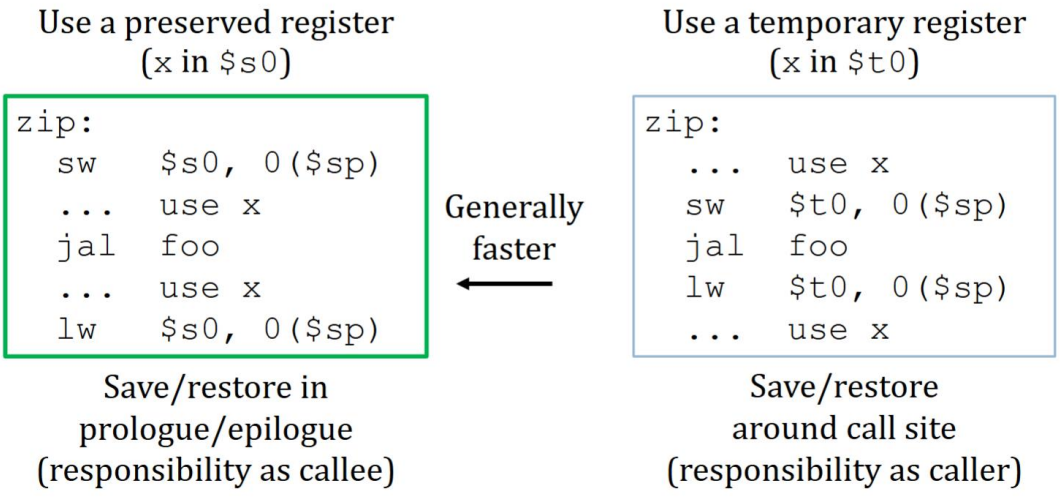
```
foo :
  addi $sp, $sp, -4     # allocate stack space
  sw   $s0, 0($sp)      # save $s0 on stack

  li   $s0, 22          # overwrite and use $s0
  ...
  lw   $s0, 0($sp)      # restore $s0
  addi $sp, $sp, 4      # deallocate stack space
  jr   $ra              # return
```

就像上面这个代码， 其实只有一行， 其他的都是用来分配内存和存储数据
当然， 上面的代码卵用没有， 我们编译的时候完全可以把它当作 temp 数据， temp 数据并不需要 callee 来存储， 所以可以大大简化代码

```
foo() {
  int p = 22;
  …
}
```

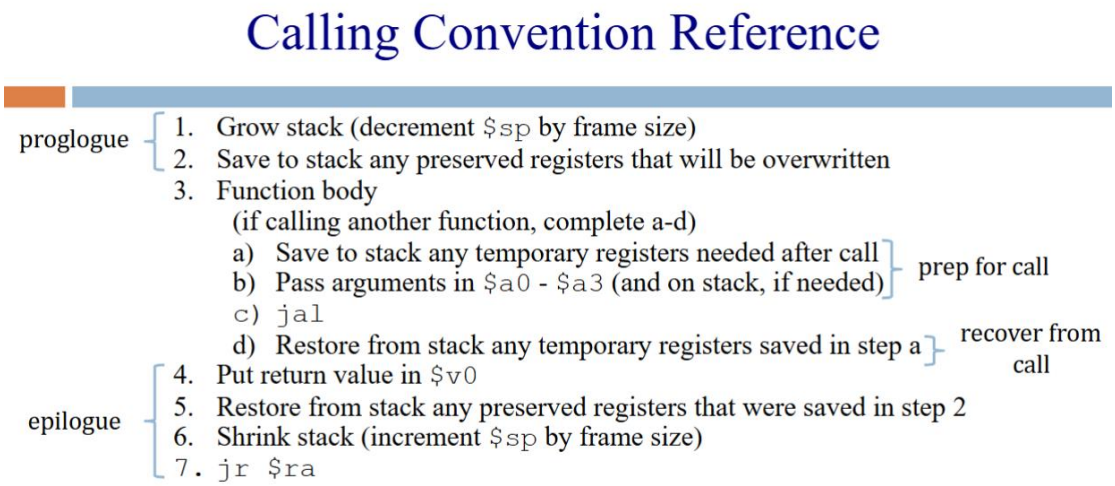
```
foo :
  li   $t0, 22      # use a temporary for p
  …
  jr   $ra          # return
```

储存读取是非常耗时的， 如果我们需要经常用某个变量， 最好把它放在$s0等寄存器中，而不是 temp 寄存器， 因为每次调用函数， 我们必定都要存储读取 temp 寄存器， 尤其是在循环中



通常我们选择存放变量在寄存器的某个位置的规则:

- 如果一个变量的使用寿命横跨了调用的函数， 那么就让函数去存这个变量(这样如果函数不用这个变量， 那么就省去了存储的消耗)
- 否则就可以直接用 temp 寄存器， 因为我们根本不需要存它 – 未来都用不上了也没存的必要了
- 如果某个函数内不会调用任何其他函数， 尝试把所有变量都放在 temp 寄存器中



Frame Pointer

Frame pointer 跟 stack pointer 有些类似， 但 frame pointer 指向的是函数的最后一个指令，在汇编中很少用到， 在 Debug 的时候比较有用

字体

很有名的两种字体格式是 UTF-8 和 ASCII， 后者值用 7 个 bit 来表示字母， 所以存储的数也很少， 当然表示英文字母已经没有什么问题了.
而 UTF8 用的是多个 byte， 所以连 emoji 也能表示.

在汇编中， 我们就用 ASCII 了: 一个 string 就是一串 byte， 每个字母是一个字节. 而 string 的结尾就是一个 0(也是一字节). 在 C 语言中也是差不多的: 我们创建了一个 5 个长度的 char 数组， 其实是 6 个长度， 最后一个元素是 0.

32	Blank space
48-57	0-9
65-90	A-Z
97-122	a-z

当写汇编代码的时候， 我们可以用 lb 和 sb 来存储和读取字节 如果字节里的不是数字， 可以用 lbu， 也就是无正负读取字节.
问题是 1 字节是 8bit， 而一个寄存器是 32bit， 所以放进去的时候还要有一些变化:
如果是 lb: 那么在读取的时候， 指令在复制完以后， 前面还有 24 个 bit 是空的， 指令会把这个 byte 的最大一位全部给赋值到前面 24bit 上， 这样负数和正数在这样操作过后数值都

是正确的.$11111110 = -1 = 11111111111111111111111111111110$
而 lbu 不会这么干

指令

指令有 R-Type 和 I-type: 前者是当指令的三个参数都是寄存器， 后者是参数中有 immediate.
但不管哪种指令， 都是占用 32bit.

之前我们说过， MIPS 中有 32 个寄存器， 所以就需要 5 个 bit 来区分这些寄存器.

R-type:

因为参数中有三个寄存器， 所以需要 15 个 bit 去区分., 还有 5 个 bit 去区分函数的种类, 还有 5 个 bit 去区分 shift amount(有些函数不需要) 最后还有一个 opcode 是不需要的部分

31-26	25-21	20-16	15-11	10-6	5-0
opcode	rs 第一参数	rt第二参数	rd目标 reg	Shift amount	Func

I-type:

I-Type 因为有个数字作为函数， 所以他们需要有 16 个 bit 去储存那个数字: $5 + 5 + 16 = 26$， 剩下来的 6 个 bit 就是用来表示这个函数的编号的.

对于 branch 也是 I-type. 虽然它后面是个 label， 但其实它这个 label 只是给我们看的， 实际上在编译的时候， 他会变成转跳多远的指令. 如跳过两个指令就是 2. 如果往上跳跃就是负数.

31-26	25-21	20-16	15-0
opcode	rs 第一参数	rt第二参数	数字参数

可以看到， 16 个 bit 根本不够用， 因为汇编可能成千上万行. 因为这是相对位置转跳， 根本没有设计转跳那么远， 这就需要 J-Type

J-Type

J-Type 有 26 个 bit 都是用来表示地址， 剩下来的 6 个 bit 是表示指令编号.

31-26	25-0

opcode	Address

如果还不够的话， 就需要 large jump， 也就是 jr