计组易错点

SID: 12012919

Name: Leo_Adventure

Chapter 1

- CPU 运行时间 = CPI * #指令 * 时钟周期
- 响应时间：计算机完成某任务所需的总时间
- 吞吐率：单位时间内完成的任务数量
- Cycles = clock rate × execution time
- Dynamic Power = 1/2 × $Capacitiveload\times Voltage^2 \times clockrate$

Chapter 2

- 在书写汇编语言的时候需要注意MIPS是按照字节进行编址的，而一个寄存器可以容纳 4 个字节，所以计算数组偏移量的时候需要 * 4
- 二进制补码的英文是 two's complement，二进制反码的英文是 one's complement
- MIPS 字段：

R 型（算术逻辑指令 (register addressing)）

op (6 bit) 操作码 + rs (5 bit) 第一个源操作数寄存器 + rt (5 bit) 第二个源操作数寄存器 + rd (5 bit) 存放操作结果的寄存器 + shamt (5 bit) 位移量 + funct (6 bit) 功能码

I 型 (立即数（immediate addressing） 、数据传送(Base addressing) 、条件分支（PC-Relative & register addressing）指令)

op (6 bit) 操作码 + rs (5 bit) 第一个源操作数寄存器 + rt (5 bit) 第二个源操作数寄存器 + const or address (16 bit)

J 型（跳转指令(Pseudo-direct addressing))

op (6 bit) 操作码 + addr (26 bit) 地址

在存取字指令中，rt 字段用于指明接收取数结果的**目的寄存器**，rs 字段用于指明 **基址寄存器**

- 栈增长是按照地址从高到低的顺序进行的，比如要存入三个临时变量，则

  ```
  addi $sp, $sp, -12
  sw $t1, 8($sp)
  sw $t0, 4($sp)
  sw $s0, 0($sp)
  ```

  返回的时候：

  ```
  lw $s0, 0($sp)
  lw $t0, 4($sp)
  lw $t1, 8($sp)
  addi $sp, $sp, 12
  ```

- 递归程序写法

```
int fact(int n){
    if(n < 1) return 1;
    else{
        return n * fact(n - 1);
    }
}
```

汇编语言：

```
fact:
    addi $sp, $sp, -8
    sw $ra, 4($sp) # 保留调用 fact 的地址
    sw $a0, 0($sp) # 保留 n
slti $t0, $a0, 1 # 判断 n ?< 1
beq $t0, $zero, L1

addi $v0, $zero, 1 # return 1
addi $sp, $sp, 8 # 恢复栈空间，由于 n < 1 时，没有改变 $ra 和 $a0 的值，所以不进行加载
jr $ra # 回到调用地址

L1:
    addi $a0, $a0, -1 # 将 n - 1 作为新参数
    jal fact # 重新调用 fact，并将 $ra 设置成下一条指令

lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
```

```
mul $v0, $a0, $v0
jr $ra
```

- 使用 lui 和 ori 将**32**位常量加载到寄存器 $\$s0$
- PC 相对寻址是相对于**下一条指令**的地址（PC + 4），将PC 和 指令中的常数相加作为寻址结果，并且由于所有的 MIPS 指令都是 4 字节长，所以在PC 相对寻址时所加的常数被设计为字地址而不是字节地址，这样对于 条件分支指令 就可以使用 16 位 来表示 18位的字节地址，对于跳转指令就可以使用 26 位来表示 28位的字节地址。 -> 这样，通过计算得到的目标地址就是 **PC + 4 + const * 4**； 并且由于 PC 是 32 位，高四位保持不变，跳转指令中的常数 代替 PC 的低 28 位。
- 条件分支指令是相对于下一条 PC 地址的寻址。跳转指令是采用完整的地址进行寻址，例如 需要跳转到 80000 地址，对应的跳转指令的 常数字段 填写 20000， 因为 20000 * 4 = 80000.
- sll $t1, $s3, 2 的机器码表示，由于 rs 是空的，所以置零， -> 0 0 19 9 2 0
- R 型指令的操作码是 0
- 寻址模式：
  - Immediate addressing（立即数寻址）: addi, subi, andi, ori, lui
  - Register addressing: add, addi, sub, subi, lw, lui
  - Base/displacement addressing: **lw, sw**, lb, sb
  - PC-relative addressing: beq, bneq
  - Pseudodirect addressing: j jal
- 对于一个浮点数，求 range 的时候，格式：
  $\pm 1.000000000 \times 2^{-14}$到 $\pm\ 1.111111111 \times 2^{15}, \pm 0, \pm \infty, NaN$
- speedup 算法：改进前的运行时间/ 改进后的运行时间
- 条件分支范围：距离当前PC + 4 的位置前后各 2^{18} 的地址（字节编址，所以乘4）
- 直接跳转范围：任何的 $256M$ 地址，然后由 PC 提供高四位

Chapter3

- 除法 乘法 硬件实现以及改进
- 偏阶为 $2^{s-1}-1$
- 注意指数 = s - 偏阶，尾数 = 1 + frac
- 浮点加法
  - 将有较小指数的数向有较大指数的数对齐
  - 将有效位相加
  - 调整至规格化表达
  - 舍入

- 指数 全零 和 全1 都被保留
- 上溢（overflow）：正的指数太大导致指数域放不下的情况； 下溢（underflow）：负的指数太大导致指数域放不下的情况

Chapter 4

- 除了跳转指令之外的指令（存储访问、算术逻辑、分支）在读取寄存器之后都会使用 ALU
- 数据通路功能部件包括两个逻辑单元：处理数据值的**组合单元(combinational unit)** 和 存储状态的 **状态单元(state unit)**
- **边沿触发方法**支持状态单元在同一个时钟周期内同时读写而不会因为竞争而出现中间数据。
- 存储器需要读控制信号，寄存器则不需要；PC不需要写控制信号，因为每个周期都会进行写入。
- RegDst 为 1 时，控制 写入寄存器编号字段 来自 rd 而不是 rt
- **ALUSrc 为1 时，第二个 ALU 操作数来自指令低16位的符号扩展(constant)**
- PCSrc 为 1 时，PC 被分支目标地址替代

指令\控制信号	RegWrite	RegDst	ALUSrc	MemtoReg
addi	1	0	1	0
lw	1	0	1	1
add	1	1	0	0

- 跳转指令的实现：[当前 PC + 4 的 高 4 位 : 跳转指令的 26 位立即数字段 : 低位 00]
- 流水线带来的性能提高是 **增加指令的吞吐率**
- 结构冒险：缺乏硬件支持导致指令不能在预定的时钟周期内执行的情况
- 数据冒险：无法提供指令执行所需数据而导致指令不能再预定的时钟周期内执行的情况 -> 解决办法： forwarding / bypassing.
- 控制冒险：取到的指令不是所需要的，导致指令不能在预定的时钟周期内执行 ---> 解决办法：分支预测
- 在 EX，MEM，和 WB 级如果将所有 9 个控制信号全部清除（置零），就会产生一个空指令，相当于在流水线插入气泡，事实上只需要将 MemWrite 和 RegWrite 置零即可
- 动态分支预测：通过查找指令的地址观察上一次执行该指令时分支是否发生，如果上次执行时分支发生，就从上一次分支发生的地方开始取新的指令。一种实现方法是：分支预测缓存（branch prediction buffer）
- 增加指令集并行的两种方式：增加流水线的深度 & 多发射（multiple issue）
- 发射包（issue packet）可以由编译器静态生成，也可由处理器动态生成，是在一个时钟周期内发射的多条指令的集合
- 循环展开（loop unrolling）：一种从存取数组的循环中获取更多性能的技术

- 寄存器重命名(renaming)：由编译器或硬件对寄存器进行重命名以消除反相关
- 动态流水线调度：对指令进行重排序(reordering)以避免阻塞的硬件支持
- 乱序执行（out-of-order commit）：执行的指令被阻塞时不会导致后面的指令等待
- 静态——编译器（软件）；动态——处理器（硬件）
- 解决结构冒险的方法有：分离指令和数据之间的缓存
- 寄存器重命名（register renaming）——静态和动态都有；动态是通过使用保留站以及重排序缓存区实现的