《计算机组成与设计》

竝

Chapter 1 指令系统

- 寄存器
  - 寄存器都是以字节编址的。
  - X0 永远为 0，无法写入。
  - a0-a7（X10-X17） 寄存器用于参数传递，a0/a1 可以做返回值寄存器。
  - s0-s1 (x8-x9) 与 s2-s11 (x18-x27)：saved registers
  - PC寄存器存的是下一个指令的地址，一条指令占四个字节，所以一般顺序执行 PC=PC+4
  - sp：stack point 栈顶指针的值是由RISC-V确定的。
  - pc=pc+8 不需要free内存因为可以覆盖。
  - 小端规则：低位字节存储在低位地址，高位字节存储在高位地址。
  - 寄存器编号与用途：

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

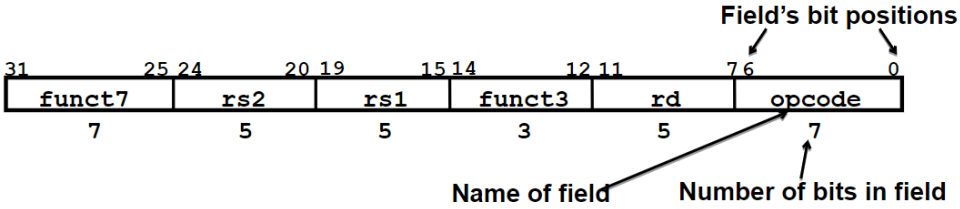
- 指令集
  - `add/sub x1, x2, x3`：x1=x2±x3
  - `addi/subi`：立即数加减
  - `lw/sw`：Load Word/Store Word 以 Word(4 Bytes) 为单位
  - `lb/sb`：Load Byte/Store Byte 以 Byte 为单位（lb 符号扩展，lbu 零扩展）
  - `beq/bne/bge/blt`：条件转移指令，相等/不相等/大于/小于则执行某操作
  - `j`：Jump 无条件转移

- `and/and;ior/ori;xor/xori`：按位与/或/异或
- `sll/srl`：Shift Left Logical/Shift Right Logical 逻辑左移/逻辑右移（左右移均补零）
- `sla/sra`：Shift Left Arithmetic/Shift Right Arithmetic 算术左移/算术右移（左移补零，右移符号位扩展）
- `slli/srli/slai/srai`：立即数左移/右移
- `mv rd, rs`：将 rs 的值保存到 rd 上
- `ls rd, 13`：令 rd 的值为 13

- 三个级别的语言
  - 机器语言：是计算机唯一可以直接识别和执行的语言。
  - 汇编语言：可以由汇编程序转换为机器语言。
  - 高级语言：需要经过编译程序转换为汇编语言（Assembly Language）；也可以直接由高级语言程序翻译成机器语言。
  - Compiler：编译程序会一次性直接转换成目标语言源文件
  - 解释程序逐句翻译并执行，不生成源文件。
- 内存分配
  - Static：静态区，直到程序运行结束才回收，用来保存全局变量、静态变量等
  - Heap：堆区，用 malloc 动态分配内存产生的变量
  - Stack：栈区，程序运行过程中用来保存寄存器值
  - Text Segment：程序段，用来存储程序的机器码，并且有 PC 指针指向
- 指令类型

> 硬件无法识别 add 这些字符，只能转化为将指令以二进制的形式存放在内存中。
>
> 指令同样是以 32bits 的 word 形式存储，同时指令被分成很多个数据块。RISC-V一共有6种指令格式：R、I、S、B、U、J。
> 一共32个寄存器器 x0~x31, 通过5bits来唯一确定。

  - **R-Format**

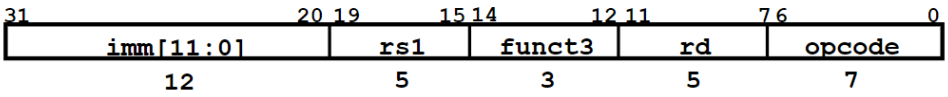


    `opcode`：操作码。R型的操作码均为`0100011`。

    `funct7+funct3`：与操作码字段一起，共同描述指令类型。

    `rs`：源操作数；`rd`：目的操作数。

  - **I-Format**



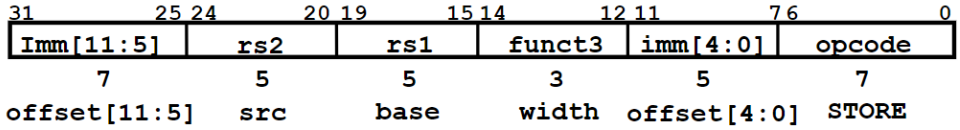
    与R型唯一不同的一点在于`funct7+rs2`被 12bits 的`imm[11:0]`代替，可以用于 $[-2048, 2047]$ 的立即数操作。

    立即数在操作前会被符号扩展至 32bits。

    `load`操作下，前 12bits 作为 Offset 与 rs1 相加得到最终地址，并将值赋给目的寄存器 rd。

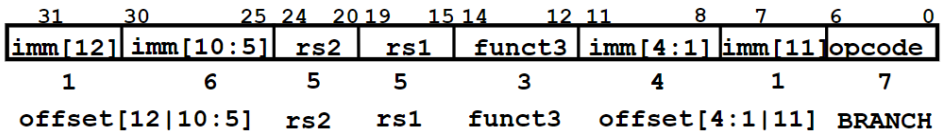
    `LH`：对最终地址里的 Halfword 进行符号拓展；`LHU`：进行无符号数的零拓展；`LW`不需要拓展。

  - **S-Format**



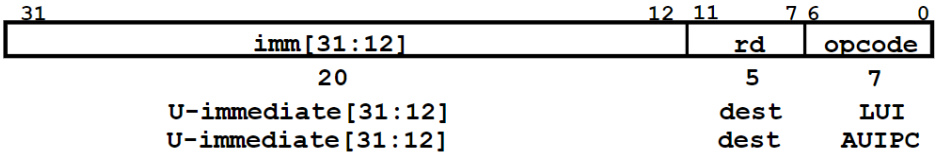
用于**Store**。不需要目的寄存器。

- **B-Format**



和S型很像，但这里Branch Offset 单位为 2bytes，12bits 的 Offset 实际表示范围为 $[-4096, 4095]$。

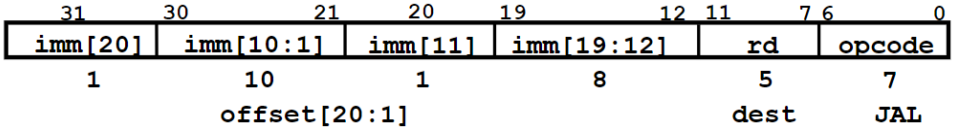
- **U-Format**



`LUI`：Load Upper Immediate （写入高20位并清空低12位），配合 `ADDI` 能够创造32位的值。当 `ADDI` 的立即数最高位为1时，需要高20位减去1，因为立即数是符号拓展的。解决方案中 `li x10, 0xDEADBEEF` 是需要软件来支持的。

`AUIPC`：Add Upper Immediate to PC 将高位立即数加到 PC 上，并将值写入目标寄存器中。

- **J-Format**



`JAL` 自动把 PC+4 存到 rd 寄存器，立即数是作为 PC 的 Offset。同样是以 2Bytes 为单位进行 Offset 计
算，与 Branch 类似，同样是减少硬件开销的手段。是 `PC-relative addressing`。

- **CPU性能指标**
  - cycle time：时钟周期。
  - frequecy：频率，即每秒多少个时钟周期。
  - CPI：平均一条指令占据时钟周期数。平均 CPI 的计算相当于把每条指令的 CPI 乘该指令使用的频率。
  - clock cycles：时钟周期数，等于 CPI * instructions
  - execution time：执行时间，等于 cycles * cycle time = CPI * instructions * cycle time

Chapter 2 控制与流水线

- 流水线
  - IF：取指
  - ID：指令译码
  - EX：执行指令
  - MEM：数据内存访问
  - WB：写回

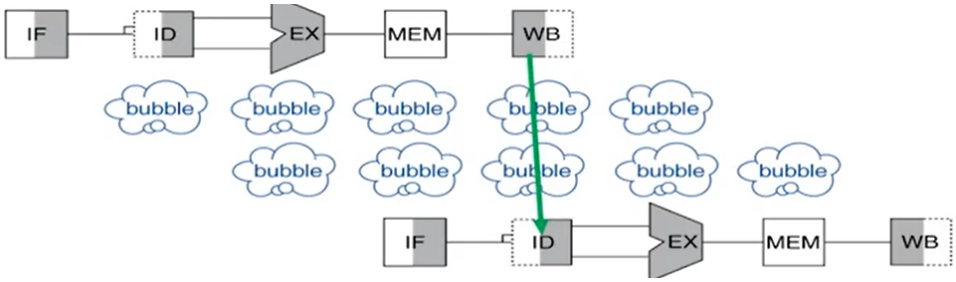
Phase	Pictogram
Instruction Fetch	
Reg Read	
ALU	
Memory	
Register Write	
$t_{instruction}$	

reg 的读/写均只需要半个时钟周期。

- 冲突类型
  - 数据冲突 **Data Hazards**

    存在必须等前一条指令执行完才能执行下一条指令的情况。解决方法如下：

    ① stalling：把遇到数据相关的指令及其后续指令都暂停一到几个时钟周期。

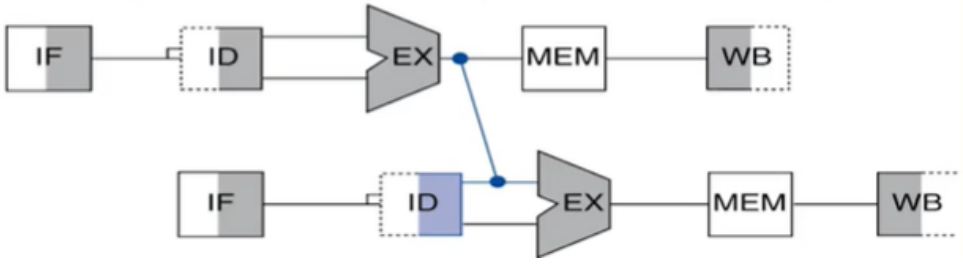
    

    如

    ```
    lw x5,0(x6)
    lw x7,0(x5)
    ```

    写后读 Read After Write

    ② 设置相关专用通路，即数据旁路技术，上一条指令生成的数据在 MEM 段前就进入下一指令的 EX 段。

    ③ 通过编译器对数据相关的指令编译优化的方法，调整指令顺序。

    注意事项：

    - WB 时期更新寄存器内容。
    - 若某指令紧跟着使用 `lw` 的结果，则需要 stall one cycle。
    - 若源操作数与上一条指令目的操作数相同，则需要 Stalling
    - X0 寄存器的数据冲突无需转发。
  - 控制冲突 **Control Hazards**

    一条指令要确定下一条指令的位置。解决方法如下：

① 对转移指令进行分支预测，尽早生成转移目标地址。② 预取转移成功和不成功两个控制流方向上的目标指令。③ 加快和提前形成条件码。④ 提高转移方向的猜准率。

- **提高处理器性能的方法**
  - 加快时钟频率
  - 改进流水线设计
  - 超标量处理器

Chapter 3 Cache

- **半导体随机存取存储器**
  - **静态SRAM存储器**

    存取速度快，但集成度低，功耗大，一般用来组成 Cache。
  - **动态DRAM存储器**

    集成度高，功耗小，但存取速度慢，一般用来组成 大容量主存系统。
- **Cache 的基本工作原理**
  - 主存地址结构：`标记位 + 行/组号 + 块内地址`
  - 结构：`有效位 + 脏位（修改位） + 替换算法控制位 + 标记位（主存高位）+ 数据位（行长）`
  - 有效容量：`行长 × 行数`
  - 总容量：`结构地址位数 × 行数`
  - Cache—主存系统效率

    效率 e =（访问 Cache 的时间 / 平均访问时间）×100%

    设命中率为 h，访问 Cache 时间为 t_c，访问主存的时间为 t_m，则

    $e = \frac{t_c}{h \times t_c + (1-h) \times t_m} \times 100\%$
  - 主存数据以块为单位放入 Cache 中
- **Cache 和主存之间的映射方式**
  - **直接映射**

    主存中的字块只能放在 Cache 中指定位置 $j = i \mod 2^c$。若已有内容，则直接替换（不考虑替换算法）。

    `主存字块标记tag + Cache块号地址index + 主存块内地址offset`

    速度快，结构简单，但Cache 占用率低，冲突率高。
  - **全相联映射**

    主存中的字块可以被放到 Cache 的任何一个字块中，但成本太高而不能采用。

    `主存字块标记tag + 主存块内地址offset`

    方式灵活，冲突率低，增加了利用率。增加了标识位位数，判断开销大。
  - **组相联映射**

    组内全相连映射，将 Cache 分成 Q 组，主存中的字块只能放到指定组中 $j = i \mod Q$

    `主存字块标记tag + 组地址index + 主存块内地址offset`

    X 路组相联：每组 X 行。
- **Cache 中主存块的替换算法（主要是 LRU）**

选择近期内长久未访问过的存储行换出；对2路组关联：在每组里单独设置一位用来指示哪个单元刚被访问过

- **Cache 写策略**
    - 写直通法（全写法） **Write Through**

      写操作时数据既写入 Cache 又写入主存，写操作时间就是访问主存的时间。Cache 块退出时，不需要对主存执行写操作，更新策略比较容易实现。时刻保持两者内容一致。
    - 写回法 **Write Back**

      写操作时数据只写入 Cache 不写入主存，当 Cache 数据被替换时才写入主存。写操作时间就是访问 Cache 的时间。缺点是可能有一致性的隐患（采用此方法时，Cache 中一定有一位“脏位”）
- 多级 **Cache**

  example：

  - Assume
    - L1 Hit Time = 1 cycle
    - L1 Miss rate = 5%
    - L2 Hit Time = 5 cycles
    - L2 Miss rate = 15% (% L1 misses that miss)
    - L2 Miss Penalty = 200 cycles
  - L1 miss penalty = 5 + 0.15 * 200 = 35
  - Avg mem access time = 1 + 0.05 x 35
    = 2.75 cycles
- 写分配与不按写分配
    - 写分配：在发生写缺失时，内存的块被读到 Cache 中，然后执行上个写命中的操作。与读缺失类似。
    - 不按写分配：仅修改底层存储器的该块而不将该块取到 Cache 中，因而不影响 Cache 内容。

      例题 假设有一个全相联的Cache，采用写回策略，刚开始时Cache为空。下面5个存储器操作（方括号中为地址）：
      Write Mem[100];
      Write Mem[100];
      Read Mem[200];
      Write Mem[200];
      Write Mem[100];
      分别求写分配和不按写分配情况下的命中次数，缺失次数。

      解：不按写分配：地址为100的单元不在Cache中，而Cache对写操作不产生分配，所以前两个写操作是缺失的。200单元也不再Cache中，读操作缺失，但该单元所在的块被取到Cache中，故下一条写操作命中，最后一条写100单元操作仍然发生缺失。不按写分配在最终结果是4次缺失和1次命中。

      写分配：对100号单元和200号单元第一次操作都发生缺失，其所在的块同时被读到Cache中，则剩下的操作都是命中的。故写分配总共有2次缺失和3次命中。

Chapter 4 虚存

- 页式虚拟存储器

  以页为基本单位的虚拟存储器，虚拟地址为：`虚拟页号 + 页内地址`

  页表是一张存放在主存中的虚页号到实页号的对照表。

  `有效位（装入位） + 脏位（修改位） + 引用位（使用位） + 物理页号/磁盘地址`

CPU执行指令时，根据虚拟地址高位的页号寻找页表项，再根据对应项装入位执行对应操作，如已装入，则取出对应物理页号，与虚拟地址低位的页内地址拼接，形成实际物理地址（主存地址）；反之，则说明缺页，进行缺页操作。

- **快表TLB**

  放在Cache中（主存中称为Page 慢表），通常采用全相联或组相联，每个TLB项由 `标记字段 tag + 页表表项内容` 组成。

  全相联时，tag = 虚页号；组相联时，tag = 虚页号高位，虚页号低 $\log_2 r$ 位作为组索引。（设分成 r 组）

  `TLB`是`Page`的一个副本，若`TLB`命中，则`Page`一定命中。

- **内存查询顺序**

  TLB->Page

  Cache->Memory