Chapter 03 程序的机器级表示

- 终端命令 `linux > gcc -Og -o p p1.c p2.c` 到底让机器做了什么?

  > `-Og` 表示符合原始C代码结构的优化等级
  > `-o p` 表示最终输出的可执行代码文件名为p

  1. **`C预处理器`** : 扩展源代码中的 `#include` 和 `#define`
     编译选项: `-E (Only run the preprocessor)`
  2. **`编译器`** : 分别产生 `p1.c` 和 `p2.c` 的**汇编代码** `p1.s` 和 `p2.s`
     编译选项: `-S (Only run preprocessor and Compiler)`
  3. **`汇编器`** : 将汇编代码转化成二进制**目标代码文件** `p1.o` 和 `p2.o` , 包含所有指令的二进制表示, 但没有填入全局值的地址.
     编译选项: `-c (Only run preprocessor, Compiler and Assemble steps)`
  4. **`链接器`** : 将两个目标文件代码, 外加**实现库函数**(如printf)的代码进行合并, 产生最终的**可执行代码文件** `p`

- **操作系统**负责管理虚拟地址空间, 将虚拟地址翻译成实际的物理地址.
- gcc 选项 `-S` 表示只进行预处理和编译, 从而方便我们查看汇编代码(.s)
- gcc 选项 `-c` 表示只进行预处理&编译&汇编, 产生 .o 文件
- x86-64的指令长度从1~15字节不等, 越常用的指令字节数越少.

  > 注: 1字节 ~ 2位16进制数. 比如指令 `48 89 d3` 的字节数为3.

- 指令的二进制编码满足: 从某个给定位置开始, 可以将字节**唯一地**解码成某条机器指令.
- 生成可执行代码需要对一组目标代码文件运行**链接器,** 这一组目标代码文件中必须包含一个main()函数.
- **链接器**为函数调用找到匹配的可执行代码的位置(地址).
- `nop` 指令表示 "no operation", 可用于让代码字节数"凑整"(比如凑成16字节).
- `%rip` (Instruction pointer) 程序计数器(Program counter), 存储**下一条指令**的地址.
  它在机器语言中不被编码! 因为它不是**GPR**(General Purpose Register, **通用寄存器**).

寄存器名称	名字最初由来	属性
%rsp	Stack Pointer	栈指针
%rax	Accumulator	返回值
%rdi	Destination Index	第1个参数
%rsi	Source Index	第2个参数
%rdx	Data	第3个参数

%rcx	Counting	第4个参数
%rbx	Base	**被调用者保存**
%rbp	Base Pointer	**被调用者保存**
%r8	-	第5个参数
%r9	-	第6个参数
%r10	-	调用者保存
%r11	-	调用者保存
%r12, %13, %r14, %r15	-	**被调用者保存(共4个)**

- 注意到 `%rdi` 和 `%rsi` 作为第一个参数和第二个参数, 这与 `strcpy(char* Dest, char* Source)` 的顺序相同.

- **向寄存器写入**1字节或2字节的指令会保持高位字节不变;
  而写入 **4字节** 的指令(movl, movzbl**等**)则会**将高位4个字节也置为0**.

  > 从而像 `movq mem %rax` 和 `movl mem %eax` 两个指令的行为是一样的, 不过由于后者是32位操作, 效率更高.
  > 这里注意 movq 要和 %rax 对应, movl 要和 %eax 对应.

- 向寄存器(如%rax)存入整数时, 此时寄存器的低位(如%al)就是**数学意义上的低位**.

  > 这无关于大端法还是小端法! 大端/小端只是**内存**中的概念!
  > 所以由于机器代码也是存储在内存中的, 所以其中的整数编码涉及小端法/大端法.

- **立即数(Immediate)** 如: `$-577` `$0x1F` 注意前者是10进制, 后者是16进制.
- `$Imm` 是表示立即数的**唯一方法**, 而不包括$符号的 `Imm` 表示的是内存的**绝对寻址** $M[Imm]$
- $R[r_a]$ **表示寄存器** r_a **的值** (这里是把所有的寄存器**看作**了数组 R)
- r_a 表示从寄存器直接取值; 而 (r_a) 表示的是内存的**间接寻址** $M[r_a]$
- $M_b[Addr]$ **表示对内存地址** $Addr$ **的** b **个字节的引用**
- 最常用的**寻址模式(Addressing Mode)** 是 $Imm(r_b, r_i, s)$
  其中, Imm 是**立即数偏移**, r_b 是**64位基址寄存器(base)**, r_i 是**64位变址寄存器(index)**, s 是**比例因子(scale)**, $s = 1, 2, 4\ or\ 8$
  有效地址(Effective Address) 的计算公式为: $Imm + R[r_b] + s \cdot R[r_i]$

  > 其它**内存引用**的寻址模式是这种寻址方式的缺省形式,

> 注意如果缺省的是基址, 则 `,` 不可省略, 错误示范: ~~`leaq (%rdx, 1), %rdx`~~ , 应该改成: `leaq (, %rdx, 1), %rdx`

- **`MOV类`** 的格式为: `MOV Src, Dest` 表示将 Src 表示的操作数复制到存储 Dest 的位置.

  > Src 和 Dest 不能都是内存引用! 如果需要内存之间复制, 需要两个指令.

  > `movq` 指令如果需要立即数, 只能以32位补码作为立即数(当然, 是会符号拓展到64位的)
  > `movabsq` 指令以64位立即数作为源操作数, 且只能以寄存器作为目的.

- "小源(Mem/reg均可)到大**寄存器**"的移动指令类:
  - **`MOVZ类`** : 零拓展(Move Zero). 不存在所谓 ~~`movzlq`~~ 指令, 因为 `movl` 指令等价于它.
  - **`MOVS类`** : 符号拓展(Move Symbol). 其中 `cltq` (convert long to quad)无操作数, 完全等价于 `movslq %eax %rax` , 但编码更紧凑.
- **栈指针** **`%rsp`** 保存栈顶元素的地址, 和数算中的定义一样.
- `popq` 和 `pushq` 指令可以拆解为 `subq/addq` 和 `movq` 的组合:
  如 `pushq %rbp` 的行为等价于 `subq $8 %rsp; movq %rbp (%rsp)` , 但 `pushq` 的指令编码更紧凑 (1Byte, instead of 8Bytes)
  所以 `pop` 和 `push` 的操作数都得是寄存器.. 否则会出现 `mov` 两端都是内存的非法情形.
- 由于**程序栈向低地址生长**, `%rsp` 减小是腾出空间, `%rsp` 增加是减少空间.
- `leaq` 指令只能进行**有限形式的乘法**运算, 因为内存寻址模式 $Imm(r_b, r_i, s)$ 中的 $s = 1, 2, 4\ or\ 8$
- `SHL` `SHR` 表示 "Shift L/R"
  `SAL` `SAR` 表示 "Shift Arithmetic L/R"
  该指令的第一个操作数(无符号的!)必须要么是立即数, 要么是 `%cl` (即 `%rcx` 的最低字节), 且在为 `%cl` 的情况下, 位移量由 `%cl` 的低 $log2w$ 位决定.

  > 比如若操作数为 `%cl` , 其中存的是 `0xFF` , 则 `salb` 位移7位, `salw` 位移15位, `sall` 位移31位, `salq` 位移63位.
  > 当然, 7, 15, 31, 63 也是 char, short, int, long 的最多合法位移量了

- `rax` `eax` 的 r 是 register 的意思; e 是 extended 的意思, 即相对于 16位整数 的拓展.
- 加减法以及普通的截断乘法是不区分有符号or无符号的.
  这是因为有无符号的差别仅在于对最高位的理解, 两种解读会使得数学上相差 $2^{w-1} - (-2^{w-1}) = 2^w$, 然而由于bit截断, " $2^w == 0$ "
- 但产生128位数(**`八字`** **`oct word`**)的 **`全乘法`** 是区分有符号or无符号的!
  其中, 有符号乘法: **`imulq`** ; 无符号乘法: **`mulq`**
  由于产生oct word, 结果要用两个寄存器才存放得下,
  规定: **必须用** `%rdx(MS):%rax(LS)` 这两个寄存器存放结果, **必须用** `%rax` 作为其中一个乘数, 另一个乘数由唯一的操作数 **S** 给出.

因此, **全乘法反而是"单操作数"的**, 而普通的64位截断乘法(无论有无符号)是"双操作数的".

通过操作数数目可以判断 `imulq` 到底是全乘法还是截断乘法.

- `cqto` (convert quad to oct) 指令把 `%rax` **符号扩展**为 `%rdx:%rax` , 即用 `%rax` 的符号位 填充 `%rdx` 的所有位. (除法要用到~)
- 无论128位还是64位除法, 都采用类似全乘法的运作方式,

  即 **`idivq`** 是一个只有**单**操作数 **S** 的指令, 它将 **`%rdx:%rax`** 这个128位数作为**被除数(divident)**, 将 **S** 作为**除数(divisor)**,

  将 **商(quotient)** 存储在 **`%rax`** , 将 **余数(remainder)** 存储在 **`%rdx`** .

  这意味着, 执行普通的64位除法前, 要确保 **`%rdx`** 被置0(无符号除法)或置符号位(有符号除法)!

  前者通过 `movl $0, %rdx` 实现, 后者通过 `cqto` 指令实现.

- **条件码寄存器(CC寄存器)**:
  - **CF**(carry flag): 用于检测**无符号**的**溢出**(**包括 `carry` 和 `borrow`**), 比如 3u - 5u 的borrow会设置CF.
  - **OF**(overflow flag): 用于检测**补码**的溢出
  - **ZF**(zero flag): 结果为0
  - **SF**(symbol flag): 结果为负数
- **leaq**不改变条件码寄存器! 它是计算地址的.
- **CMP**的行为和**SUB**类似, **TEST**的行为和**AND**类似

  所以CMP指令本身并不包含"判断依据"这一层含义, CMP只是单纯地执行一次"减法".

  "判断依据"是由紧跟CMP的那条指令的后缀给出的.
- **setl**中的l表示的是less, 而非long word;

  而所谓less是指"a<b为真" , 因为setl检查的 `条件码组合为true` 对应于 `a-b 在数学意义上产生了负的结果(负or负溢出)` .

  所以当 `SET_` 紧随 `CMP b a` 时, l 就"字面意义上地"对应于表达式 a<b , g 就对应于表达式 a>b , 其它尾缀也是同理的.

  > 或许把SET指令的尾缀理解为"和0比如何如何"会更好一些.
  >
  > 比如把 `setl` 理解为 "结果比0小(less)吗?"
  >
  > (这里基于一个思想, 即判断 $a \ \mathrm{CMP}\ \mathrm{b}$, 在数学上等价于判断 $a - b \ \mathrm{CMP}\ 0$, 所以我们会去检测 $a - b$ 的符号. 但要注意汇编语言两个操作数顺序是反直觉的.)

- 设 a, b 为无符号数, 如果数学意义上 $a - b < 0$, 则 `CMP b a` 会设置 `CF=1` , 因为需要借位.

  从而无符号数的比较基于 **CF 和 ZF** . (**CF**: 是否<0?; **ZF**: 是否==0?)
- 有符号的SET指令中, 使用 **g**reater 和 **l**ess 表示大小;

  无符号的SET指令中, 使用 **a**bove 和 **b**elow 表示大小.

  此外, 尾缀 **s** 和尾缀 **ns** 表示 **s**ymbol 和 **n**ot **s**ymbol, 即 负数 和 非负数.
- **`条件尾缀`** 与 **`条件码组合`** 的对应关系如何**记忆**?

- 无符号中, 条件尾缀 `b (below, <)` 是"万物之源", 简单地对应于条件码组合 **`CF`** , 即**无符号减法结果产生溢出(借位)**.
- 有符号中, 条件尾缀 `l (less, <)` 是"万物之源", 对应于条件码组合 **`SF^OF`** , 即**有符号减法结果要么负, 要么负溢出**(到正数了).

只要记住了这两点(**都是关于"<"的!**), 其余的就只是这两者的变形, 即加入 **`ZF`** 来明确是否**取等**, 仅此而已!

- 逻辑操作(如XOR)会设置CF和OF为0;
  移位(Shift)操作会把CF设置为**最后一个被移出的位**, 把OF设置为0;
  INC和DEC会设置OF和ZF, 但不会改变CF. (有"补码专用"的意味...)
  (原因书中没有说明)

- **`jmp 无条件跳转`** : `jmp .L1` : **直接跳转** ; `jmp *%rax` , `jmp *(%rax)` : **间接跳转** .

  > `.L1` 这种标签只在汇编代码中可见, 在目标代码中已经被计算成实际地址.

- **`jCC 条件跳转`** : 条件跳转**只能是直接跳转**!
- 跳转目标 的 **`编码(而非书写在.S文件里)`** 方式:
  1. **`PC相对的`** (最常用): 将 (目标地址 — 跳转指令的**下一条指令的地址(因为%rip存的是下一条的)**) 作为编码. 编码为1/2/4字节, 且为有符号数(补码)!
  2. 直接给出4字节的绝对地址(为啥是4字节, Stack Overflow说是因为Intel认为你的目标文件怎么也不会超过2GB)
- 从 `if else` 到 `jCC` 的翻译中, 可以理解为什么 `&&` 具有"短路"性质.
- 条件跳转(如 `jne` , `jl` 等)在现代处理器的流水线模式下, 可能招致严重的 **`分支预测错误惩罚`** .

  > 分支预测错误惩罚 T_{punish} 的计算:
  > 记 T_{ran} , T_{random} 分别为执行代码所需时间和随机预测策略下平均所需时间,
  > 则可以注意到, $T_{random} = T_{Run} + 0.5 * T_{punish}$, 即有一半的情况下会被惩罚.
  > 从这个式子可以解出$T_{punish} = 2 * (T_{random} - T_{run})$

- 因此为了符合现代处理器的特性, 在***一些特定情况***下, 可以考虑用 `CMOV` (即**条件传送**指令) 来代替**条件跳转**指令.
  不同于 `MOV` 类, `CMOV` 类指令的尾缀是表明"传送条件"的, 而非"数据大小". 而"数据大小"可以从寄存器的类型推断得出.
  此外 `CMOV` 不支持单字节(即b)的传送.

  > ***特定情况*** 即 then_expr 和 else_expr 没有副作用, 且不会引发异常. 而且这两个都比较好计算.
  > 编译器需要对"分支预测错误惩罚"和"进行多余的运算"进行权衡.
  > 不过事实上, 对GCC而言, 只有当两个表达式都**非常**容易计算时, 比如只是一条加法, 才会使用 CMOV.

- > 不要把 MOV / CMOV 的行为 和 ADD 搞混了!

- `do-while` 是最自然实现的循环,
  `while` 的实现, 可以通过对 `do-while` 的汇编代码前加入一个 `jmp test` , 可以生成 `while循环` , 称为 `jumpToMiddle` 方法;
  也可以通过 `guarded-do` 方法, 即先执行一次判断, 如果是 `false` , 就跳过循环.
- 当 `for` 循环中出现 **continue** 语句时, 翻译成 `while` 时, **continue** 需要特别处理成 `goto update` , 而非直接沿用 **continue**.
  因为这里隐藏着一个细节: `for` 循环被 **continue** 后依然会进行 比如"i++" 的变量的 update 操作;
  而 `while` 循环被 **continue** 后则只是单纯地重新开始循环.
  所以如果不特别处理 **continue** (正确做法是**goto**到 Update: 那里, 而非**goto**到 Loop:) , 会造成死循环.

- **switch** 语句的翻译中, GCC会尝试使用**跳转表**提高效率 (而非简单的顺序遍历)
  比如某个**switch(expr)**语句中有100, 102, 103, 104, 106这几个**case** , 那GCC就会创建一个特殊的指针数组, 其中存放的是指向代码位置的指针, 比如 `static void* jt[7] = {&&loc_A, &&loc_def, &&loc_B, ..., &&loc_D}` (其中&&是取代码位置的操作符, 这是C语言标准之外的东西). 然后对于需要判断的**expr**, 先对它减去100, 然后判断它是否在[0,6]之间: 如果不在, 则**执行default后的语句(不是跳过switch! 别搞错了)**, 即跳到&&loc_def; 如果在, 则 goto 到 jt[expr] 对应的代码位置. (注意到jt[expr]是把expr作为下标来访问jt数组, 是O(1)的!)
  将这一过程落实到汇编, 则是声明一个 .rodata (read-only-data)的跳转表, 比如.L4是这个跳转表的地址. 于是C语言的 `goto jt[expr]` 翻译为汇编的 `jmp *L4(,%rsi,8)`
  总结一下就是: **case**后的数字 --减去100--> 指针数组下标index --*8后加上.L4--> 代码块地址

  > 不过如果case不够连续, 哪怕是100,200,300,...,900这种“虽然有规律的不连续”, 编译器就不会采用“地址表”, 但仍然会采用类似二分法的优化.

- **过程的栈帧(stack frame)**
  当 P 调用 Q 时, 会把返回地址压入栈中. 返回地址被规定为是 P 的栈帧内的一部分. 而再之后则是 Q 的栈帧了.

栈底(高地址)	栈帧所有者
. . .	(更早的过程)
...	P
第n个参数	P
...	P

第7个参数	P
返回地址 (标志着P的栈帧的结束, 以下是Q的了)	P
Q保存在栈内的 **被调用者保存寄存器**	Q
局部变量	Q
调用子过程时的参数构造区 (如果6个寄存器够用, 就没有这个区域了)	Q
栈顶(低地址)	

- `call Q` : 把 call 的下一条指令的地址压入栈中, 然后将 PC 设置为地址 Q , 和 jmp 的参数类似, Q 可以是直接的, 也可以是间接的.
- `ret` : 从栈顶弹出地址, 记为 A, 然后把 PC 设置为地址 A .
- **被调用者保存寄存器** : `%rbx` `%rbp` `%r12` `%r13` `%r14` `%r15` 共6个.

  > cpy记忆: 不传参的2个字母寄存器, 和靠后的2个数字寄存器是自己调用别的函数时的数据保险柜

- **调用者保存寄存器** : 除了以上6个和 `%rsp` 之外的全部: 包括6个用来传参的, 返回值 `%rax` , 和 `%r10` `%r11`

  > 以下叙述均假设 main 调用 P, P 调用 Q.
  > 所谓被调用者保存寄存器, 就是说当 P 调用子过程 Q 时, Q 必须保证这些寄存器刚进入 Q 时什么样, 从 Q 返回 P 时就是什么样.
  > 当然, 父进程 P 也会认定这些寄存器一定不会被 Q 修改, 是 P 心目中万无一失的"保险柜".
  > 另一个例子. 假设当 P 调用 Q 时, P 需要把某个变量x存储在 `%rbx` 中, 但此时要注意, P 作为调用者的同时也是 main 的被调用者!!! 所以 P 在存储 x 之前, 需要先把 main 在 `%rbx` 中存储的数据压入 P 自己的栈中, 在返回 main 之前 pop 回 `%rbx` .

- > cpy:
  > 被调用者保存寄存器: "父进程存储变量的保险柜"
  > 调用者保存寄存器: 各个过程公用的, 谁都可以随意修改, 所以仅用于加速数据读取, 即临时变量不必反复从内存访问.

- P想调用Q, 但Q需要的整数or指针参数多于6个. 由于只有6个整数寄存器, 参数7~n需要用栈来传递.
  其中参数7放在栈顶. 且所有的数据大小向8的倍数对齐. 当参数到位后, 程序将执行 call Q .
- 过程通过**减小栈指针** `%rsp` 的方式, 在栈上分配空间.
- > cpy: 论一个进程P对自己的栈帧的使用:
  > 首先, 对于P会用到的 `被调用者保存寄存器` , 使用 `push` 的方法保存在 P 的栈帧.

> 然后, 计算P的局部变量所需空间, 以及需要传递给子过程 Q 的实参所需的空间, 通过减小 `%rsp` , 一次性分配够.
> 最后, 从P返回main之前, 先把 `%rsp` 加回去, 然后把保存的寄存器数据 `pop` 回去.

- **数组与高维数组**:
  int A[3]; // 声明一个长度为3的以 `int` 为元素的数组
  int B[5][3]; // 声明一个长度为5的, 以 `长度为3的int数组` 为元素的数组

  > 所以其实 int B[5][3] 应该把 B[3] 看做一个整体, 而5表示有多少个这样的整体.

- **数据对齐**:
  - 任何 K 字节的 `基本对象` 的首地址必须是 K 的倍数.
  - 结构体本身的地址也要满足最大成员(比如long)的对齐量(比如8的倍数).
  - 所以结构体的结尾也可能会将一些空间用于对齐. 这是因为考虑到结构体在内存中连续排列成数组时也应满足对齐.

    > 对于 struct，将成员按照其 `数据大小` 从小到大(或从大到小)的顺序进行排列, 才能使内存占用最小.

- **函数指针**: `(返回值类型)(*指针名)(形参表)` 例: `(int*)(*fp)(int, int)`

- **标量数据**: 整数, 长整数, 指针, 浮点数, 即不是任何聚合形式(包括数组, struct, union)的数据.
- `objdectdump -d mstore.o` 反汇编, 默认打印到屏幕. 也可以重定向到文件: `objdectdump -d mstore.o > mstore.d`
- **GDB调试器**
- **对抗缓冲区溢出攻击**:
  - **程序员**: 使用更安全的实现, 使用 fgets, strncpy 等更安全的函数.

    ```c
    //更安全的字符读入方法:
    fgets(s1, 5, stdin);
    scanf("%5s", s1);
    strncpy(s2, s1, 2);
    ```

  - **操作系统**: 1.栈随机化. 2.限制可执行代码区域, 即引入内存的 `NX(No-Execute)` 位, 将 `可读` 和 `可执行` 分开对待.
  - **编译器**: 设置**金丝雀值**(canary): 在P的返回地址和Q的char[]缓冲区之间设置一个canary, 在从Q返回P之前, 检查canary是否被篡改了. (不设置金丝雀值: `-fno-stack-protector`)

- **变长栈帧**: 比如声明一个变长数组, 或者使用标准库函数 `alloca` , 都会在栈帧上分配空间, 空间大小在编译期间是不可知的!
- 为了管理变长栈帧, 即在调用结束后释放 `alloca` 分配的空间,

可以使用寄存器 `%rbp` 作为**帧指针**(base_pointer), 具体地:

首先, 由于 `%rbp` 是一个被调用者保存寄存器, 所以在最开始得先把之前 `%rbp` 的值push到栈帧存储.

然后, 将 `%rbp` 的值设置为 "保存之前 `%rbp` 的值的栈帧" 的地址, 并且在函数的整个执行过程中**不再改变**.

... ...

最后, 在函数执行的末尾, 使用 **leave** 指令: 效果是将 `栈指针%rsp` 设置为 `%rbp` , 并把存在栈里的 `%rbp` 值恢复回寄存器.

(即 **leave**指令等价于: `movq %rbp,%rsp` , `popq %rbp`)

- `YMM` 是256位的, `XMM` 是128位的, `MM` 是64位的. 可以存放浮点数, 也可以是整数.
- `%xmm0` ~ `%xmm7` 用于传递最多8个浮点参数, 更多的参数使用栈来传递.
  其中 `%xmm0` **同时承担**返回浮点数返回值的作用.
- 所有 `XMM` 寄存器都是 `调用者保存(Caller-Saved)` 的. 即子进程可以"随便修改随便用".
- AVX浮点操作**不能**以立即数作为操作数, 编译器必须为常量也分配和初始化存储空间.
  这些浮点数在汇编中写作 `.long 3435973837 .long 1073532108` 的形式(这里一个.long表示long word, 分别编码了低和高4个字节(由于是小端法!)). 而要想理解它代表什么浮点数, 则需要先把这两个十进制数转成16进制, 然后转成2进制, 最后用浮点数的IEEE定义翻译成浮点数.
- 浮点的位操作: `vxorps` `vandps` , 同样地, `vxorps` 可以用于置零某个 `%XMM` 寄存器, 这样立即数0既可以用 `.long` 定义, 也可以用 `vxorps` 生成.
- 浮点比较指令 `vucomiss` 和 `vucomisd` 会设置 `ZF` 和 `CF` 以及 `PF` 位, (并置零 `SF` 和 `OF`).
  `PF` 的本意是低8位的奇偶校验, **然而在浮点数中**, `PF` 改用于标志浮点比较中是否有某个操作数是 `NaN` .
  根据C语言惯例, 在浮点比较中, 如果有一个操作数为 `NaN` , 则认为比较失败了, 返回 0 .
- `addss` 表示 add scalar single 单精度标量相加
  `addsd` 表示 add scalar double 双精度标量相加
  这两个指令只利用 XMM 的低字节, 高字节不被使用.
- XMM 寄存器没有历史包袱, 即没有兼容 `低字节使用` 的问题. 所以不用指明到底是用整个, 还是只用低字节. 一律写作 `%xmm_` 即可.
- `vcvtsi2ssq` : convert scalar integer to scalar single quad
  `vynpcklpd` : unpack low packed double

End Of Note, 444 Lines.