пару слов по арифметике

Есть у нас инструкция загрузки эффективного адреса очевидно что
lea ebx, [eax]; [eax] - указатель на память с адресом eax, поэтому ebx:=eax
но вспомним что адресные выражения могут быть разные и включать несколько регистров, тогда мы получаем инструкцию что может быстро выполнить арифметику за 1 такт сделав и умножение и сложение и не выставив флаги
пример:
lea eax, [eax+2*eax]
умножили eax на 3 так, как помним адресные выражения то можем умножить на степень двойки один из регистров прибавить убавить константу для себя умножить ebx на 5, сложите eax и ebx и прибавьте 4

Инструкции

зная вас вы возможно так и не поняли что за анонимные метки:
при компиляции все анонимные метки заменяются по аналогии с local метками в макросах на ??0000 в 16 сс

```
je @F; (Future)
    <код>
@@:
```

```
@@:
    <код>
je @B; (Back)
```

инструкции по работе с флагами все без операндов

cld	DF:=0 строковые операции на увеличение адресов
std	DF:=1 строковые операции на уменьшение адресов
clc	CF:=0
stc	CF:=1
cmc	CF:=not(CF)
cli	IF:=0 Interrupt Flag замаскировать прерывания (кроме №2) прерывания будут игнорироваться
sti	IF:=1 Interrupt Flag вернуть обычное поведение с прерываниями
lahf	загрузить в **ah** арифметические флаги AH := EFLAGS(SF,ZF,0,AF,0,PF,1,CF)
sahf	загрузить из **ah** арифметические флаги EFLAGS(SF,ZF,0,AF,0,PF,1,CF) := AH
pushfd	загрузить в стек 32 рязрядный EFLAGS
popfd	забрать 32 разрядный EFLAGS

инструкции перехода
напомню что они могут быть

1. короткий rel8
2. длинный rel16/rel16

rel - непосредственный
напомню что также можно сделать
jmp $+8 (переход на 8 байт вперёд по адресу) - если что компилятор сам так сделает если посчитает что это возможно и будет быстрее
и так тоже можно: jmp eax

инструкция	флаги
je, jz	ZF=1
jne, jnz	ZF=0
jg, jnle	SF=OF and ZF =0
jge, jnl	SF=OF
jl, jnge	SF<>OF
jle, jng	SF<>OF or ZF=1
ja, jnbe	CF=0 and ZF=0
jae, jnb, jnc	CF=0
jb, jnae, jc	CF=1
jbe, jna	CF=1 or ZF=1
js	SF =1
jns	SF =0
jo	OF =1
jno	OF =0
jcxz	CX = 0
jecxz	ECX = 0

напомню мнемонику: e - equal - для любых, g -greater больше, l - less меньше- для знаковых, a-above (выше) b - below (ниже) - для беззнаковых, n - not, cx ecx это и есть регистры)
loop - сначала уменьшит ecx потом выполнит переход если <> 0 то есть ecx := 15 тогда 15 раз код выполнится
условная пересылка инструкции *c***mov***cc* op1, op2
op1 - register 16 бит или register 32 бит, op2 - register/memory 16 бит или register/memory 32 бит соответственно сувать константу ошибка! все условия проверки аналогичны инструкциям условного перехода, отсутсвуют лишь условная пересылка по регистру ecx и cx пример

```
.code
    cmp bx, cx; напомню нельзя память память и константу первым параметром
    cmovb bx, cx; положит в bx max(bx, cx) беззнаково
```

следующие инструкции как по мне уже очевидны добавлю только то что размеры одинаковы и единственное что запрещено 1 -операнд - константа и оба операнда память:

mov

xchg

add

sub

inc - увеличить на 1 влияет на OF SF ZF но не CF r/m

dec - уменьшить на 1 влияет на OF SF ZF но не CF r/m

neg - берёт отрицание числа

adc - op1 := op1+(CF+op2) - для сложения 64 битных чисел допустим

sbb - op1 := op1-(CF+op2) - для вычитания

lea - взять адрес

mul AX=AL*r/m8 (8 битное)

mul DX:AX= AX*r/m16 (16 битное)

mul EDX:EAX= EAX*r/m32 (32 битное)

imul - знаковое умножение

div 8 битное беззнаковое: AL := AX div r/m8 AL := AX mod r/m8

div 16 битное беззнаковое: AX := DX:AX div r/m16 DX := DX:AX mod r/m16

div 32 битное беззнаковое: EAX := EDX:EAX div r/m32 EDX := EDX:EAX mod r/m32

idiv знаковое деление

movsx - знаковое расширение

movzx - беззнаковое расширение

Инструкции знакового расширения применяются чаще всего при знаковом делении!!

1. cbw - convert byte to word AL -> AX знаково
2. cwd - convert word to double word AX -> DX:AX
3. cdq - convert double to quadro EAX -> EDX:EAX
4. cwde - convert word to double EAX AX -> EAX

(скажу разве что первым параметром у всех операций сдвигов и логики может быть память)
Логические операции : and, or, xor, not
2 операнд сдвигов - imm8 (константа) или регистр CL!!
Сдвиги: shr, shl, sal, sar - **SH**ift **R**ight, **S**hift **A**rithmetic **L**eft (ну мнемоника надеюсь ясна)
shl и sal одинаково действую с заполнением нулями битов справа
shr заполняет слева нулями, sal заполняет слева или справо так чтобы сохранить знак
бит что был сдвинут отправляется в флаг CF
Циклические сдвиги rol,ror, rcl, rcr - циклические сдвиги
rol и ror цикл внутри операнда но сдвинутый бит оказывается в CF
rcl и rcr цикл внутри операнда+CF те он на примере rcl ax, 1: сделает сдвиг влево и в младший (правый разряд) положит содержие CF при этом в после инструкции в CF окажется старший бит что был сдвинут

векторные инструкции я где то писал но повторюсь:
<имя инструкции>[s|p][s|d] - первый выбор между s (scalar) и p (packed) означает будем ли мы обращаться с 1 элементом в xmm1 как правило элементом что в младшей части регистра или будем обращаться ко всем сразу второй выбор s (single precision) и d (double precision) за точность 32 битные числа или 64 битные

addss xmm, xmm/mem32 - сложить
addsd xmm, xmm/mem64
subss xmm, xmm/mem32 - вычесть
subsd xmm, xmm/mem64
mulss xmm, xmm/mem32 - умножить
mulsd xmm, xmm/mem64

divss xmm, xmm/mem32 - делить
divsd xmm, xmm/mem64
sqrtss xmm, xmm/mem32 - корень
sqrtsd xmm, xmm/mem64

по инструкциями с packed также можно брать из памяти но память должна быть выравнена то есть адрес начала m128 в 16cc в конце имеет 0 те если из стека использовать то сохраняем стек и выполняем and esp, FFFFFFF0h и потом уже пушим что хоти складывать если данные описаны в .data то используем align 16:

addps xmm, xmm/mem128 - сложить
addpd xmm, xmm/mem128
subps xmm, xmm/mem128 - вычесть
subpd xmm, xmm/mem128
mulps xmm, xmm/mem128 - умножить
mulpd xmm, xmm/mem128

divps xmm, xmm/mem128 - делить
divpd xmm, xmm/mem128
sqrtps xmm, xmm/mem128 - корень
sqrtpd xmm, xmm/mem128

Подробно про сравнения и cvt в ответах 25 числа напомню что в xmm не хранятся целочисленные их хранят в mmx! и что packed (cvtps2pi) преобразует лишь 2 числа (в младших битах) чаще всего использовать придётся cvtsi2ss и cvtss2si integer хранится в r/m32 число float в m32/xmm в память класть напрямую нельзя и иллюстрация shufps - очень важная инструкция если будет практическая задача

fld - положить на вершину стека fpu (st0) операнд - память
fstp - достать из вершины (st0) - модно преобразовывать в real10 так
movss, movsd, movups, movupd, movapd, movapd - mem|xmm, mem|xmm (память в память нельзя) последние 2 инструкции требует ровнения по 16, также как и с packed операциями для арифметики

Строковые команды

последняя буква - b|w|d - byte|word|double
как уже писали раннее DF=0 смотрим строку вперёд, DF=1 назад
std, cld - установить DF=1 и DF=0 соответственно
Мнемоника использования регистров:
ESI - Source
EDI - Destination
EAX|AX|AL - accumulator
ECX - counter

cmps[b|w|d] - сравнить esi с edi (псевдокод: cmp [esi],[edi])
movs[b|w|d] - переслать из esi в edi
scas[b|w|d] - Аккумулятор сравнивается с edi (псевдокод: cmp EAX,[edi])
lods[b|w|d] - загрузить в Аккумулятор из esi
stos[b|w|d] - Загрузить в edi Аккумулятор
rep - повторять до ecx = 0
repe - повторять пока равны и ecx <> 0
repne - повторять пока не равны и ecx <> 0
читайте лучше методичку по строковым она здесь лежит) там подробнее