Win32 Assembly Cheat Sheet

Binary operations (x = x *operation* y)

Add/Subtract 64 bits in edx:eax
```
ADD [lo], eax
ADC [hi], edx
SUB [lo], eax
SBB [hi], edx
```

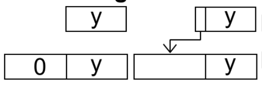
	MOV	reg, reg
+	ADD	reg, imm
–	SUB	reg, [mem]
	XOR	[mem], reg
	OR	[mem], imm
	AND	
x = x – y – cf	SBB	
x = x + y + cf	ADC	
set zf if (x and y) = 0	TEST	
flags ⇐ x – y	CMP	

Unary operations
(x = *operation* x)

	NOT	reg
x = –x	NEG	[mem]
x = x + 1	INC	
x = x – 1	DEC	

No operation and undefined instruction
NOP (db 90h)
UD2 (db 0Fh, 0Bh)

Zero or sign extension



MOVZX	reg32, reg8 | byte [mem]
MOVSX	reg32, reg16 | word [mem]
	reg16, reg8 | byte [mem]

dx = sign(ax)	CWD
edx = sign(eax)	CDQ
eq. to movsx ax, al	CBW
eq. to movsx eax, ax	CWDE

Memory [mem]
[reg + reg * (1|2|4|8) + abs_address]
Some parts may be omitted, e.g.,
[abs_address + reg] or [reg + reg * 4]

LEA instruction

eax = eax * 5	LEA eax, [eax + eax * 4]
ebx = eax + ecx	LEA ebx, [eax + ecx]
ebx = eax – 10	LEA ebx, [eax – 10]

Register

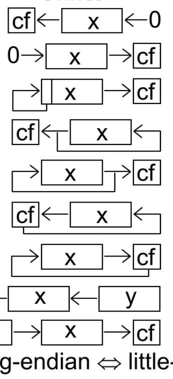
Register	reg		
STOS, LODS, DIV, MUL, CMPXCHG, XLAT; return value of function	eax	ax	ah, al
base for XLAT	ebx*	bx	bh, bl
counter for loops (REP, LOOP, and JECXZ)	ecx	cx	ch, cl
edx:eax holds 64-bit values for MUL, DIV, RDTSC, and CMPXCHG8B	edx	dx	dh, dl
base pointer for stack frame	ebp*	bp	
stack pointer (top of the stack)	esp*	sp	
source index for CMPS, LODS, MOVS	esi*	si	
destination for CMPS, STOS,SCAS,MOVS	edi*	di	

* should be preserved across function calls. Also clear df flag if you've set it.

Multiplication and division

			remainder:
ax = al * x	MUL	reg	
dx:ax = ax * x	IMUL	[mem]	
edx:eax = eax * x			
al = ax / x	DIV	reg	ah
ax = dx:ax / x	IDIV	[mem]	dx
eax = edx:eax / x			edx
x = x * y	IMUL	reg, reg | reg, [mem]	
x = y * z	IMUL	reg, reg, imm | reg, [mem], imm	

Shifts



SHL(SAL)	reg, imm
SHR	reg, cl
SAR	[mem], imm
ROL	[mem], cl
ROR	
RCL	
RCR	
SHRL	reg, cl
SHRD	[mem], reg, imm

Big-endian ⇔ little-endian BSWAP reg32
Requires 486

Conditions

if eax = ebx then G1
```
          CMP eax, ebx
inverted
condition!→JNZ @F
          [G1]
          @@:
```

if eax = ebx then G1 else G2
```
          CMP eax, ebx
          JNZ @F
          [G1]
          JMP end
@@:       [G2]
end:
```

Do-while loops

do G1 while eax ≠ ebx
```
@@: [G1]
    CMP eax, ebx
    JNZ @B
```

while eax ≠ ebx do G1
```
    CMP eax, ebx
    JZ skiploop
@@: [G1]
    CMP eax, ebx
    JNZ @B
skiploop:
```

Flags

carry/borrow for addition/ subtraction	cf	JC JNC
set if the result of the previous operation is 0	zf	JZ JNZ
set if the result is < 0	sf	JS JNS
set if there's a signed overflow	of	JO JNO
if set, string operations go in reversed direction	df	

df=1 STD	cf=1 STC
df=0 CLD	cf=0 CLC
	cf=not cf CMC
push flags PUSHF	
pop flags POPF	

ADD, SUB, TEST, CMP AND, and OR set all 4 flags.
MOV doesn't set any flags.
INC and DEC don't set cf.

Function calls and argument passing

__stdcall (stdcall, invoke)	*__cdecl (ccall, cinvoke)*	*__fastcall*	*thiscall*
PUSH arg2	PUSH arg2	PUSH arg4	PUSH arg2
PUSH arg1	PUSH arg1	PUSH arg3	PUSH arg1
CALL func	CALL func	MOV edx, arg2	MOV ecx, \ this
	ADD esp, 8	MOV ecx, arg1	CALL func
		CALL func	
func: ...	func: ...	func: ...	func: ...
ret 8	ret	ret 8	ret 8

Conditional jumps

JE(JZ)	x = y	
JNE(JNZ)	x ≠ y	
signed		*unsigned*
JG(JNLE)	x > y	JA(JNBE)
JL(JNGE)	x < y	JB(JNAE)
JGE(JNL)	x ≥ y	JAE(JNB)
JLE(JNG)	x ≤ y	JBE(JNA)

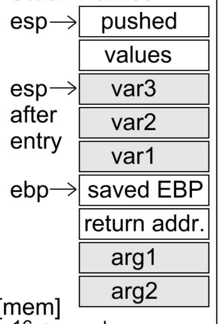
Branchless code

if cc then x=1 else x=0	SETcc reg8 | byte [mem]
if cc then x=y	CMOVcc reg$\frac{16}{32}$, reg$\frac{16}{32}$ | $\frac{\text{word}}{\text{dword}}$[mem]

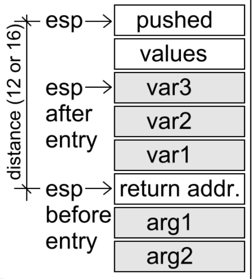
Check support for CMOVcc with CPUID. Requires Pentium pro or Athlon.

set cf if x < y (unsigned)	CMP x, y
make bitmask from cf	SBB eax, eax
if cf then increment eax	ADC eax, 0

Stack frames



```
            arg1 equ ebp + 8
            arg2 equ ebp +12
            var1 equ ebp – 4
            var2 equ ebp – 8
            var3 equ ebp –12
eq. to      PUSH ebp
ENTER 12,0  MOV ebp, esp
            SUB esp, 12
            MOV eax, [arg1]
            MOV [var1], eax
            ...
eq. to      MOV esp, ebp
LEAVE       POP ebp
```



```
arg1 equ esp + 4
arg2 equ esp + 8
var1 equ esp – 4
var2 equ esp – 8
var3 equ esp – 12
SUB esp, 12 ; vars
MOV eax, [12+arg1]
MOV [12+var1], eax
; Distance changes:
PUSH eax
MOV [16+var1], eax
...
ADD esp, 12
```

For loops

for ecx=5 to 1 do G1
```
 MOV ecx, 5
@@: [G1]
 DEC ecx
 JNZ @B
```

eax = 0
for ecx=0 to 9 do
eax = eax + a[ecx]
```
 XOR eax, eax
 MOV ecx, –(4 * 10)
@@: ADD eax, \
    [a + 40 + ecx]
 ADD ecx, 4
 JNZ @B
```

Switch...case

if eax=0 then G0 else if eax=1 then G1 else GD
```
 CMP eax, 2
 JAE def
 JMP [t + eax * 4]
case0: [G0]
 JMP @F
case1: [G1]
 JMP @F
def:   [GD]
@@:
t dd case0, case1
```

Spin-Wait Loop
```
acq: XOR eax, eax
  INC eax
  XCHG eax, [cs]
  OR eax, eax
  JNZ spin_loop
[shared data access]
spin_loop: PAUSE
  CMP [cs], 0
  JNE spin_loop
  JMP acq
```

String operations

fill ecx elements from [edi] with aq	REP STOSq
move ecx elements from [esi] to [edi]	REP MOVSq
find aq in ecx elements at [edi]	REPE SCASq
find non-aq in ecx elements at [edi]	REPNE SCASq

After execution of the instruction, [edi] will point at the found value.

compare [esi] with [edi] (ecx elem's)	REPE CMPSq

zf will be set if equal. Use JA/JB to find which string is greater.

find the first equal elements in [esi] and [edi] (ecx elem's)	REPNE CMPSq

[esi] and [edi] will point at the equal elements afterwards.
q is B(yte), W(ord), or D(word). Correspondingly, aq is al, ax, or eax.

Synchronization

exchange: x=y, y=x	XCHG [mem] | reg,reg
x=x+y, y=x	LOCK XADD [mem] | reg, reg
if eax=x then x=y, zf=1 else zf=0	LOCK CMPXCHG [mem] | reg, reg
if edx:eax=x then x=ecx:ebx, zf=1 else zf=0	LOCK CMPXCHG8B qword[mem]

You should check support for CMPXCHG8B with CPUID. Requires Pentium or higher.

Time measurement
edx:eax = clocks RDTSC
Requires Pentium or Athlon. Flush the pipeline with CPUID.

CPUID instruction
eax=0 ⇒ eax=maxeax, ebx=$\frac{\text{'Auth'}}{\text{'Genu'}}$, edx=$\frac{\text{'enti'}}{\text{'ineI'}}$, ecx=$\frac{\text{'cAMD'}}{\text{'ntel'}}$
eax=1 ⇒ eax8:11=family, eax4..7=model, edx:0=sse3,
ecx:4=rdtsc, 15=cmov, 23=mmx, 25=sse, 26=sse2
eax=8000_0000h ⇒ eax=max eax for extended functions
eax=8000_0001h ⇒ edx:31=3Dnow, edx:30=3Dnow-ext
eax=8000_0002h, ..3h, ..4h ⇒ eax:ebx:ecx:edx=namestring
Requires Pentium (may work on some 486's).