Intel Cheat Sheet

Registers

- Each of the following registers is 32 bits in size

Name	Meaning of Acronym	Usage
EAX	Accumulator	General Purpose Register
EBX	Base	General Purpose Register
ECX	Counter	General Purpose Register
EDX	Data	General Purpose Register
ESI	Source Index	General Purpose Register
EDI	Destination Index	General Purpose Register
EBP	Base Pointer	Used as the base pointer in C code. Can be used as a general register in stand alone code
ESP	Stack Pointer	The stack pointer.

- You can access smaller portions of EAX, EBX, ECX, and EDX
  - The following example only shows the syntax for eax but it does apply to the the above mentioned registers

Syntax	Size
EAX	32 bits
AX	Lower 16 bits of EAX
AH	The upper 8 bits of AX
AL	The lower 8 bits of AX

AS Preprocesser Memory Allocations

Syntax	Effect	Example
.byte val	Make space for a byte and initialize it val	.byte 'h'
.word val	Make space for 2 bytes and initialize it val	.word 12
.long val	Make space for 4 byte and initialize it val	.long 1024
.float val	Make space for 4 bytes and initialize its value to the IEE floating point standard	.float 58.45
.string val	Make space for null terminated string and initialize it to val.	.string “Rock the boat”
.space N	Make space for N bytes of uninitialized memory	.space 16

GAS Preprocessor Directives

Syntax	Effect	Example
.equ label, value	Defines label to be value. All occurrences of label will be substituted with value	.equ wordsize, 4
.data	Marks the beginning of the data section	
.text	Marks the beginning of the instruction section	
label_name:	Creates the label with name label_name	hello_world:
.global label	Make label visible to linker	.global _start
.rept N code .endr	Repeat all code N times	.rept 4 addl %eax, %eax .endr
_start:	While not actually a preprocessor directive _start indicates where execution should begin. It is equivalent to main in C	

Operations

- In operations that require two operands the left is called the source and the right the destination

Operation	Result	Affects Flag Register?
mov src, dest	dest = src	No
add src, dest	dest = dest + src	Yes
sub src, dest	dest = dest - src	Yes
cmp src, dest	dest – src (result not stored)	Yes
inc dest	dest = dest + 1	Yes
dec dest	dest = dest - 1	Yes
and src, dest	dest = dest & src	Yes
or src, dest	dest = dest | src	Yes
xor src, dest	Dest = dest ^ src	Yes
not dest	dest = ~dest	Yes
shr src, dest	dest = dest >> src (brings in 0s)	Yes
sar src, dest	dest = dest >> src (brings in bit with same value as MSB	Yes
shl src, dest	dest = dest << src	Yes

sal src, dest	dest = dest << src (identical to shl)	Yes

- Constants can be specified for the source by placing a $ in front of the value
  - ie addl $4, %ecx
- All of the above operations need to have one of the following suffixes appended to them to determine the size of the unit they are operating on
- R is standing in for the variable name of each register ie (A,B,C,D)

Suffix	Size	Register Portion To Use
l (this is the letter L not 1)	32 bits	ERX
w	16 bits	RX
b	byte	RL

Multiplication and Division

Operation	Result	Type	Affects Flag Register?
mulb src	ax = al * src	unsgined	Yes
mulw src	dx:ax = ax * src	unsgined	Yes
mull src	edx:eax = eax * src	unsgined	Yes
divb src	ah = al / src al = ax % src	unsgined	Yes
divw src	ax = dx:ax / src dx = dx:ax % src	unsgined	Yes
divl src	eax = edx:eax / src edx = edx:eax % src	unsgined	Yes

- EDX:EAX means to concatenate the bits in EAX and EDX together and treat it as a 64 bit number
- To do signed multiplication/division append an I to the front of the instruction ie(imull, idivl)

“String” Instructions

Operation	Result	ECX	EDI	ESI
movsb	C(%edi) = C(%esi)	ECX--	If Direction EDI -= 1 else EDI += 1	If Direction ESI -= 1 else ESI += 1
movsw	C(%edi) = C(%esi)	ECX--	If Direction EDI -= 2 else EDI += 2	If Direction ESI -= 2 else ESI += 2
movsl	C(%edi) = C(%esi)	ECX--	If Direction EDI -= 4 else EDI += 4	If Direction ESI -= 4 else ESI += 4
stosb	C(%edi) = AL	ECX--	If Direction EDI -= 1 else EDI += 1	Unaffected
stosw	C(%edi) =AX	ECX--	If Direction EDI -= 2 else EDI += 2	Unaffected
stosl	C(%edi) = EAX	ECX--	If Direction EDI -= 4 else EDI += 4	Unaffected
cmpsb	C(%esi) - C(%edi)	ECX--	If Direction EDI -= 1 else EDI += 1	If Direction ESI -= 1 else ESI += 1
cmpsw	C(%esi) - C(%edi)	ECX--	If Direction EDI -= 2 else EDI += 2	If Direction ESI -= 2 else ESI += 2
cmpsl	C(%esi) - C(%edi)	ECX--	If Direction EDI -= 4 else EDI += 4	If Direction ESI -= 4 else ESI += 4
scasb	AL - C(%edi)	ECX--	If Direction EDI -= 1 else EDI += 1	Unaffected
scasw	AX - C(%edi)	ECX--	If Direction EDI -= 2 else EDI += 2	Unaffected
scasl	EAX - C(%edi)	ECX--	If Direction EDI -= 4 else EDI += 4	Unaffected

- Direction is controlled by the direction flag. To set the direction flag use the instruction STD. To clear the direction flag use CLD

Repetition Prefixes

Operation	Supported Prefixes
movs	rep
stos	rep
cmps	rep, repe, repne
scas	rep, repe, repne

- Appending *rep* to any of the instructions mentioned causes them to continue to execute until ECX reaches 0
- repe (repeat while equal) causes the instruction to be repeated until either ECX reaches 0 or the comparison results in an inequality
- repne (repeat while not equal) causes the instruction to be repeated until either ECX reaches 0 or the comparison results in an equality
- For repe and repne to check what caused the termination of the instruction you can use
  - JECXZ: Jump if ECX is 0, to check if ECX's value is 0
  - JZ/JNZ: to check if the comparison resulted in an equality/inequality

Jumps

Instruction	Description	Comparison Type
jmp label	Unconditional jump	NA
jl label	Jump if less than 0	Signed
jg label	Jump if greater than 0	Signed
jle label	Jump if less than or equal 0	Signed
jge label	Jump if greater than or equal 0	Signed
jb label	Jump if below 0	Unsigned
ja label	Jump if above 0	Unsigned
jbe label	Jump if below or equal 0	Unsigned
jae label	Jump if above or equal 0	Unsigned
jz label	Jump if zero	Signed or Unsigned
jnz label	Jump if not zero	Signed or Unsigned
jc label	Jump if carry	NA
jnc label	Jump if no carry	NA
jo label	Jump if overflow	NA
jno label	Jump if no overflow	NA

- Jumps are used to control the flow of execution
- All jumps are based off the last instruction to set the sign flag

Stack Operations

Operation	Result	Affects Flag Register?
pushl src	C(esp) = src, esp -= 4	No
popl dest	dest = C(esp), esp += 4	No

- C means the value of the contents of memory at the given
- All operations to the stack must be of word size so you can't do pushw, popb, etc

Call and Return

Operation	Result
Call label	Push PC, jmp label
ret	Pop PC

Indexing Syntax

- displacement(base, index, scale)
  - Result access memory at location $displacement + base + index * scale$

Field	Type	Value if not specified
displacement	Constant	0
base	register	0
index	register	0
scale	One of (1,2,4, and maybe 8). If index is not give scale should not be provided	1

  - Displacement, base, index,a and scale are all optional.
  - If you want the address that is computed by the advanced indexing syntax you can use the **leal** instruction
    - leal dispalcement(base, index, scale), %dest
      - dest = displacement + base + (index * scale)

C Conventions

- Arguments are pushed in reverse order of their appearance (ie right to left)
  - Example: int foo(int a, int b, int c)
    - First c is pushed, then b, then a, and then the function is called
- Local variables are pushed in order of their appearance (ie left to right)
  - myfun()
    int x,y,z;
    int a,b,c;
  - The push order would be x,y,z,a,b,c
- On entry to a function registers EAX, ECX, and EDX will not have live values
  - This means in your function you are free to overwrite these registers without saving their values
  - This also means that if you call a C function in your assembly you must be sure to save EAX, ECX, and EDX before you make the call as that function is likely going to overwrite those values
  - If you want to use any other registers you must make sure to save their values before overwriting them and restore them to their original values before leaving your function
- The return value of a function is placed in EAX
  - 64 bit return values are placed in EDX:EAX

Inline Assembly

- It is possible to insert assembly code directly into either C or C++ code by using the __asm__ construct
- The format using __asm__ is
  - __asm__(assembly code : outputs : inputs : clobber list);
  - outputs, inputs, and the clobber list are all optional
  - If you don't have outputs the call looks like __asm__(assembly code : : inputs : clobber list)
- Assembly code is a string containing the assembly instructions that you would like executed.
  - Each instruction has to be separated by a valid delimiter
    - For us that is going to be ;
    - Registers have 2 % signs placed in front of them
      - ie %%eax, %%ebx, %%esi, etc
    - Arguments have a single % placed in front of them
  - Outputs and inputs are how we map variables that are in the C code into our inlined assembly. Each one has the form
    - ['['symbolic name']'] “modifiers* constraints+” (C variable name)
      - [] denote optional
      - * means 0 or more
      - + means 1 or more
      - if something is enclosed in ' ' it is literally that character

Constraint	Meaning
a	EAX
b	EBX
c	ECX
d	EDX
S	ESI
D	EDI
r	Any general purpose register. The compiler chooses which one to use
m	memory
g	Any general purpose register or memory

Modifiers	Meaning
'='	This variable is write only. The C value variable associated with this inline assembly value will be overwritten with this value
+	This variable is both read and write
&	This variable will be written to before all inputs are consumed (early colobber)

    - Symbolic names allow you to refer to this argument more conveniently in the assembly code. To refer to an argument in the assembly you do %[symbol name]

        - Symbolic names do not have to be the same as the name of the C variable but they can be.
    - If you do not give symbolic names you can refer to the argument by its position. %0 is the first argument, %1 the second, %2, the third and so on
- The clobbered list contains the registers, memory, and condition codes that could change but are not listed as either inputs or outputs

Name	Description
“%eax”, “%ebx”, “%ecx” ,...	EAX, EBX, ECX, ...
memory	Memory
cc	The condition codes. In the intel case this is the eflags register

- Short example. For more examples see basicInline.cpp and inlineFun.cpp

```
int a = 10;
int b = 20;
int c = 30;
int x,y;
__asm(
“movl %3, %%eax;” //set x to a's value
“addl %[fun], %[newc];” //c+= b
“movl %[newc], %1” // y = c
:
“=a” (x), “=m” (y), [newc] “+r” (c) :
[a] “g” (a), [fun] “b” (b):
“cc”
);
```