

# 1. Required Knowledge to Write in Assembly

## 1. Application Binary Interface

→ (ABI): Function/OS interoperation

- (a) Argument passing
- (b) Stack handling
- (c) Register conventions

## 2. Instruction Set Architecture

→ (ISA): ISA actually hex inst formats, but most assemblers use suggested mnemonics

→ These are the instructions that you must build programs out of

## 3. Registers/flags

## 4. Assembler used (gas/MASM):

- (a) Assembler directives (prefixed by `.`)
- (b) Operand order (`src`, `dest`)
- (c) Const identifier (`$`)
- (d) Register identifiers (`%`)
- (e) x86/gas suffixes commands with precision qualifier:
  - i. `b` : 1 byte int
  - ii. `w` : 2 byte int
  - iii. `l` : 4 byte int
  - iv. `q` : 8 byte int
  - v. `l` : 8 byte float
  - vi. `s` : 4 byte float

## 2. Further Resources

- 80x80 Assembly Language and Computer Architecture by Richard C. Detmer
  - Doesn't cover SSE[1-3], but good basic ref
- ATLAS assembly page (from [links] on class homepage):
  - <http://math-atlas.sourceforge.net/devel/assembly/>
- ATLAS architecture page (from [links] on class homepage):
  - <http://math-atlas.sourceforge.net/devel/arch/>

### 3. x86 Calling Sequence and Stack Frame

- Stack grows downward in memory
- Caller puts callees' args in its frame
- Frame 4-byte (32 bit) aligned
- If callee needs no scratch space, can leave SP unmodified
- Otherwise, subtract frame size from SP, keeping 4-byte aligned

	Caller's frame
	last arg
	:
$\%esp+4$	1st arg
$\%esp$	return address

**Stack frame passed to callee**

# 4. x86 Integer Registers

## general purpose

## index

31 ..... 0

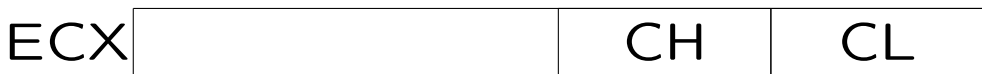
31 ..... 0



AX



BX



CX



DX

- All but `eax`, `edx`, `ecx` callee-saved
- 8 regs of every size
- `esp` is stack pointer
- `ebp` can be frame ptr
- `eax` is integer return val reg
  - `int64:edx←hi32, eax←lo32`

Also have:

- `EIP/IP` – instruction ptr
- `EFLAGS/FLAGS` – control & status bits

## 5. x86 Assembly Overview

- Two operand assembler:
  - **gas**: `<mnem> <src> <dest>`
  - **masm**: `<mnem> <dest> <src>`
- ⇒ Dest is input/output!
  - \* `add src, dst → dst = dst + src;`
- Most ops take (mem)/reg/immed, rdest
  - call this `mrisc, rd`
    - \* constants begin with \$, regs with %, mem is address
- Ops with memory operands need size suffix (b,w,l,q,[l,s])
- Usually only one operand can be from memory
- CISC format: diff inst of different sizes/efficiencies
- Generally, arithmetic affects status word, movement does not

## 6. x86 Addressing Modes

### CONST(breg, nreg, mul)

- $@ = \text{val}(\text{breg}) + \text{val}(\text{nreg}) * \text{mul} + \text{CONST}$
- CONST is signed integer constant:
  - $[-128, 127]$  : 8 bit offset
  - else : 32 bit offset
- breg: reg holding base address
- nreg: index reg
- mul: amount to mul index value by:
  - Can be 1, 2, 4, 8

### examples

- `120(%esp)`
  - Load item 120B above SP
- `(%eax,%edi,4)`
  - `X[i]` assuming `edi=i`, `eax=X` (single)
- `16(%eax,%ecx,8)`
  - `A[2+1*lda]` assuming `ecx=lda`, `eax=A` (double)

## 7. x86 Integer Movement/Init Operations

Mnemonic	Operands	Action
mov	rs, rd	rd = rs ( <i>register copy</i> )
mov	rs, (mem)	*(mem) = rs ( <i>store</i> )
mov	(mem), rd	rd = *(mem) ( <i>load</i> )
mov	const8/16/32, rd	rd = const; ( <i>reg init from inst</i> )
xchg	r0, r1	Exchanges reg contents ( <i>register swap</i> )
xchg	r0, eax	Exchanges reg contents; eax faster
xchg	r0, (mem)	Exchanges reg and mem contents

## 8. Common Integer Arithmetic Operations

Mnemonic	Operands	Action
add	mris, rd	rd += mris
sub	mris, rd	rd -= mris
inc	mrd	mrd++ (unsigned) (does not set CF)
dec	mrd	mrd-- (unsigned) (does not set CF)
neg	mrd	mrd = -mrd
mul	mrs	(unsigned) eax = lo(eax*mrs); edx = hi(eax*mrs)
imul	mrs	(signed) eax = lo(eax*mrs); edx = hi(eax*mrs)
imul	mris, rd	(signed) rd = lo(rd*mrs); hi(rd*mrs) == 0
imul	imm, mrs, rd	(signed) rd = lo(imm*mrs); hi(imm*mrs) == 0
idiv/div	mrs	eax ← (hi32(edx):lo32(eax))/(mrs), dx ← rem
lea	(mem), rd	rd = &(mem) (extremely flexible!)

LEA (Load Effective Address) can be used for multiply, shift, and/or add, without needing scratch regs:

- takes any valid indexing mode, puts target @ in rdest
- $ecx = 8 + 4*eax + ebx \Rightarrow lea\ 8(ebx, eax, 4),\ ecx$

## 9. Common Bit-Level Operations

Mnemonic	Operands	Action
and	mris, rd	rd &= mris
or	mris, rd	rd  = mris
xor	mris, rd	rd ^= mris, (if rd=rs, zero!)
not	rd	rd = ~rd
shl	imm, mrd	mrd << (imm%32)
shl	cl, mrd	mrd << (ecx%32)
shr	imm/cl, mrd	mrd >> (imm%32) 0 fill right
sar	imm/cl, mrd	mrd >> (imm%32) sign bit fill right

## 10. x86 Integer Condition Codes

### unsigned iop:

cmp result	CF	ZF
dst < src	1	0
dst = src	0	1
dst > src	0	0

### signed iop:

cmp result	ZF	SF, OF
dst < src	?	SF $\neq$ OF
dst = src	1	0
dst > src	0	SF = OF

Conditions signaled in flag register:

- **ZF** (bit 6): result of op is zero
- **CF** (bit 0): (*unsigned only*) set to 1 if unsigned addition overflows or unsigned subtraction borrows from beyond integer
- **SF** (bit 7): set to most sig bit of result (1 means neg number)
- **OF** (bit 11): most sig bit of dest different than both src sig bits
- **PF** (bit 2) : set to 1 if there's an even number of 1s in least sig byte of dest, otherwise 0.

## 11. Common x86 Comparison Instructions

Mnemonic	Operands	Action
cmp	mris, rd	Set CC as if $rd - mris$
cmp	rs, mrid	Set CC as if $mrid - rs$
test	ris, mrid	Set CC as if $mrid \& rs$
bt	imm, rd	Set CF to value of imm bit from rd
btc	imm, rd	Set CF to value of imm bit from rd toggle (compliment) imm bit in rd
btr	imm, rd	Set CF to value of imm bit from rd set imm bit in rd to 0
bts	imm, rd	Set CF to value of imm bit from rd set imm bit in rd to 1

- Can do comparison early, branch later  
→ Must be no intervening iops that set the same CC bits

## 12. Common x86 Comparison & Branching Instruction

Mnemonic	Operands	Action	CC Check
jumps for signed cmp			
jg/jnle	label	jump if (dst > src)	SF=0, ZF=0
jge/jnl	label	jump if (dst >= src)	SF=OF
jl/jnge	label	jump if (dst < src)	SF ≠ OF
jle/jng	label	jump if (dst <= src)	ZF=1 or SF ≠ OF
jumps for unsigned cmp			
ja/jnbe	label	jump if (dst > src)	CF=0 and ZF=0
jae/jnb	label	jump if (dst >= src)	CF=0
jb/jnae	label	jump if (dst < src)	CF=1
jbe/jna	label	jump if (dst <= src)	CF=1 or ZF=1
bit-level jumps			
je/jz	label	jump if (dst == src)	ZF=1
jne/jnz	label	jump if (dst != src)	ZF=0
jc	label	jump if CF=1	CF=1
jnc	label	jump if CF=0	CF=0

## 13. x87 Floating Point Registers

79 .....	0
	ST(0)
	ST(1)
	ST(2)
	ST(3)
	ST(4)
	ST(5)
	ST(6)
	ST(7)

- ST(0) aka ST (stack top)
- For most fp ops, one operand must be ST(0)
  - other operand register or memory
- As new values are loaded, values pushed down in stack
- Two regs swapped via FXCH
- Values added by push ops (implicit/explicit)
- Values removed by pop ops (implicit/explicit)

## 14. Common x87 Stack Manipulation Instructions

Mnemonic	Operands	Action
<code>finit</code>	none	init FPU & clear stack
<code>fld</code>	<code>(mem@)</code>	<code>*(mem@)</code> pushed onto stack
<code>fld</code>	<code>st(x)</code>	<code>ST(x)</code> pushed onto stack
<code>fld1</code>	none	1.0 pushed onto stack
<code>fldz</code>	none	0.0 pushed onto stack
<code>fst</code>	<code>st(x)</code>	$ST(x) = ST$
<code>fstp</code>	<code>st(x)</code>	$ST(x) = ST$ ; <code>ST</code> popped
<code>fst</code>	<code>(mem@)</code>	$*(mem@) = ST$
<code>fstp</code>	<code>(mem@)</code>	$*(mem@) = ST$ ; <code>ST</code> popped
<code>fxch</code>	none	swap <code>ST</code> and <code>ST(1)</code>
<code>fxch</code>	<code>st(x)</code>	swap <code>ST</code> and <code>ST(x)</code>

## 15. Common x87 Floating Point Computation Instructions

Mnemonic	Operands	Action
fadd(fmul)	none	replace ST & ST(1) with their sum (product)
fadd(fmul)	(mem)	replace ST with its sum (product) with *(mem)
fadd(fmul)	st, st(x)	replace ST(x) with sum (product)
fadd(fmul)	st(x), st	replace ST with sum (product)
faddp(fmulp)	st, st(x)	replace ST(x) with sum (product), pop ST
fsub	none	replace ST & ST(1) with ST(1)-ST
fsub	(mem)	replace ST with ST - *(mem)
fsub	st, st(x)	ST(x) -= ST
fsub	st(x), st	ST -= ST(x)
fsubp	st, st(x)	ST(x) -= ST; pop ST
fsubr	none	replace ST & ST(1) with ST-ST(1)
fsubr	(mem)	replace ST with *mem - ST
fsubr	st, st(x)	ST(x) = ST - ST(x)
fsubr	st(x), st	ST = ST(x) - ST
fsubpr	st, st(x)	ST(x) = ST - ST(x); pop ST
fabs	none	ST =  ST
fchs	none	ST = -ST

## 16. x87 Comparison Instructions

Mnem	Ops	cmp ST against?
fcom	none	ST(1)
fcom	st(x)	ST(x)
fcom	mem	*(mem)
ftst	none	0.0
fcomp	none	ST(1), pop ST
fcomp	st(x)	ST(x), pop ST
fcomp	mem	*(mem), pop ST
fcompp	none	ST(1), pop ST(0) & ST(1)
fstsw	ax	ax = fp status word

- use fstsw store fp status word to ax, then use test or bt to trigger branches:

Result of fcom	14	10	8
	C3	C2	C0
ST >	0	0	0
ST <	0	0	1
ST =	1	0	0
fcomi:	ZF	PF	CF
	<b>fp status bits set</b>		

- From PPRO on, have fcomi for all all-register fcom variants that directly sets int flags as shown in table

## 17. Three Sections for Assembly Routines

### 1. **prologue:**

- Figures local frame size
- Moves stack pointer
- Saves all used callee-saved registers
- Loads arguments from previous frame

### 2. **body:**

- Function body

### 3. **epilogue:**

- Restores saved registers (including SP)
- Sets any return values
- returns to caller

⇒ If this separation enforced, func body unchanged as ABI changes

## 18. Simple DAXPY in x86 Assembly

```
#define N      %eax
#define X      %edx
#define Y      %ecx
#define II     %edi
#define FSIZE 4
/*
void ATL_UAXPY
    4          8          16
(const int N, double alpha, TYPE *X,
 int incX, TYPE *Y, int incY)
    20         24         28
*/
# Prologue
.text
.global ATL_UAXPY
ATL_UAXPY:
    finit          # clear ST
    subl    $FSIZE, %esp # get frame space
    movl    %edi, (%esp) # save reg
# Load paras
    movl    FSIZE+4(%esp), N
    fldl    FSIZE+8(%esp)    # ST = {alpha}
    movl    FSIZE+16(%esp), X
    movl    FSIZE+24(%esp), Y

# for (i=0; i < N; i++)
#     Y[i] = alpha * X[i]
    cmp     $0, N
    je     DONE
    xorl    II, II
LOOP1:
    fldl    (X,II,8)    # ST = {X[i], alpha}
    fmul    %st(1), %st # ST = {alpha*X[i], alpha}
    faddl    (Y,II,8)    # ST = {Y[i]+alpha*X[i], alpha}
    fstpl    (Y,II,8)    # ST = {alpha}
#
# Increment II, and jmp back to top of loop
#
    addl    $1, II
    cmp     II, N
    jne    LOOP1
# Epilogue: restore regs and return
DONE:
    fstp    %st(0)      # ST = {}
    movl    (%esp), %edi
    addl    $FSIZE, %esp
    ret
```

## 19. Avoiding Loop Comparison

- On x86, all int ops do implicit comparison to 0
- Can often save a comparison instruction by running loop backwards

```
# for (i=N; i; i--)
#     Y[N-i] = alpha * X[N-i]
    cmp     $0, N
    je     DONE
    movl   N, II
    neg    II           # II = -N
    lea   (X, N, 8), X   # X+=N;
    lea   (Y, N, 8), Y   # Y+=N;
LOOP1:
    fldl   (X,II,8)      # ST = {X[i], alpha}
    fmul   %st(1), %st   # ST = {alpha*X[i], alpha}
    faddl  (Y,II,8)      # ST = {Y[i]+alpha*X[i], alpha}
    fstpl  (Y,II,8)      # ST = {alpha}
# Increment II, and jmp back to top of loop
    incl   II
    jnz    LOOP1
```