x86 Assembly Language Guide

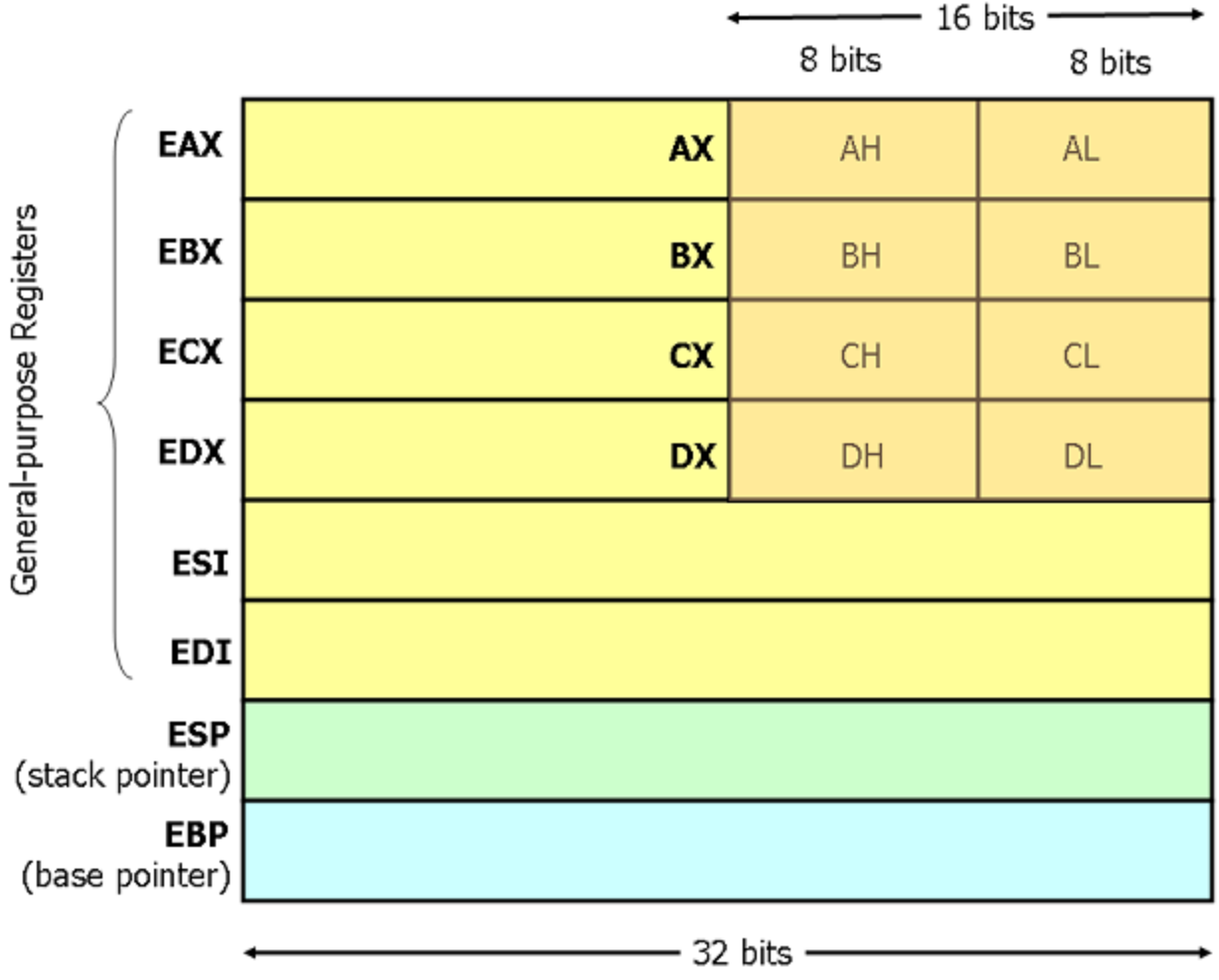
Lejun Yu

Beijing Normal University

- This guide describes the basics of 32-bit x86 assembly language programming in syntax of **AT&T**.
- A small but useful subset of the available instructions and assembler directives is covered.

Registers

eight 32-bit general purpose registers in a x86 processor



Memory and Addressing Modes

Declaring Static Data Regions

Static data regions (analogous to global variables)
Data declarations should be preceded by the *.data* directive.
Following this directive, the directives

- **.byte**,
- **.short**,
- **.long**

can be used to declare one, two, and four byte data locations, respectively.

Locations declared in sequence will be located in memory next to one another.

Declaring Static Data Regions

Example

```
.data
var:
        .byte 64 # a byte value, 64. ~location var

        .byte 10 # a byte value, 10, ~location is var+1.
 x:
        .short 42 # 2-byte value, 42, ~location x.
 y:
        .long 30000 # a 4-byte value, 30000, ~location y
```

Arrary

Arrays in x86 assembly language are simply a number of cells located contiguously in memory.

Arrary

Examples

```
S:
        .long 1, 2, 3 # Declare three 4-byte values,
                      # 1, 2, and 3.
                      # The value at (s + 8) will be 3.

barr:
        .zero 10 # 10 bytes starting at barr, 0.

str:
        .string "hello" # 6 bytes starting at
                        # the address str
```

Addressing Memory

A 32-bit system addressing up to 2^{32} bytes of memory.

Labels in assembly code are actually replaced by the assembler with 32-bit quantities that specify addresses in memory.

One of the registers can be optionally pre-multiplied by 2, 4, or 8.

Addressing Memory

Example

```
mov (%ebx), %eax  # Load 4 bytes from the memory address
                  # in EBX into EAX.

mov %ebx, var(,1) # Move the contents of EBX into the
                  # 4 bytes at memory address var.
                  # (Note, var is a 32-bit constant).

mov -4(%esi), %eax # Move 4 bytes at memory address
                   # ESI + (-4) into EAX.

mov %cl, (%esi,%eax,1) # Move the contents of CL into
                       # the byte at address ESI+EAX.

mov (%esi,%ebx,4), %edx  # Move the 4 bytes of data at
                         # address ESI+4*EBX into EDX.
```

Addressing Memory

Examples of **invalid** address calculations

```
mov (%ebx,%ecx,-1), %eax # Can only add register values.

mov %ebx, (%eax,%esi,%edi,1) # At most 2 registers
                             # in address computation.
```

Operation Suffixes

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction.

Ambiguity!

```
mov $2, (%ebx)
```

Will value 2 be moved into the single byte or 4 bytes at address EBX?

Operation Suffixes

The suffixes **b**, **w**, **l** indicate sizes of 1, 2, and 4 bytes respectively.

Example

```
movb $2, (%ebx) # Move 2 into the single byte
                # at the address stored in EBX.

movw $2, (%ebx) # Move the 16-bit integer
                # representation of 2 into the 2 bytes
                # starting at the address in EBX.

movl $2, (%ebx) # Move the 32-bit integer
                # representation of 2 into the 4 bytes
                # starting at the address in EBX.
```

Instructions

Machine instructions generally fall into three categories:

- **data movement**
- **arithmetic/logic**
- **control-flow**

Data Movement Instructions

- mov
- push/pop
- lea

mov — Move

The mov instruction: `mov op1 op2`

copies the data item refered to by op1 into op2

mov — Move

Syntax

```
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <con>, <reg>
mov <con>, <mem>
```

Examples

```
mov %ebx, %eax     # copy the value in EBX into EAX
movb $5, var(,1)   # store 5 into the byte at var
```

`push` — Push on stack

The `push` instruction places its operand onto the top of the hardware supported stack in memory.

> 1. **ESP** -= 4,
> 2. places the operand into the 32-bit location at address (%esp)

- **NOTE**: *stack grows down in the x86*

`push` — Push on stack

Syntax

```
push <reg32>
push <mem>
push <con32>
```

Examples

```
push %eax     # push eax on the stack
push var(,1)  # push the 4 bytes at var onto the stack
```

`pop` — Pop from stack

The `pop` instruction removes the 4-byte data element from the top of the stack into the specified operand (register or memory location).

> 1. copy the 4-byte data element from the top of the stack into operand.
> 2. ESP += 4.

`pop` — Pop from stack

Syntax

```
pop <reg32>
pop <mem>
```

Examples

```
pop %edi     # pop the top element of the stack into EDI.

pop (%ebx)   # pop the top element of the stack
             # into memory at the four bytes
             # starting at location EBX.
```

lea — Load effective address

The `lea` instruction places the address specified by its first operand into the register specified by its second operand.

Note: the contents of the memory location are not loaded, only the **effective address** is computed and placed into the register.

lea — Load effective address

Syntax

```
lea <mem>, <reg32>
```

Examples

```
lea (%ebx,%esi,8), %edi    # the quantity EBX+8*ESI
                           # is placed in EDI.

lea val(,1), %eax          # the value val is
                           # placed in EAX.
```

Arithmetic and Logic Instructions

- **Arithmetic instructions:**
  - add, sub
  - inc, dec
  - imul, idiv
- **Logic Instructions:**
  - and, or, xor
  - not
  - neg

add — Integer addition

The add instruction adds together its two operands, storing the result in its second operand.

Note, both operands may be registers, at most one operand may be a memory location.

add — Integer addition

Syntax

```
add <reg>, <reg>
add <mem>, <reg>
add <reg>, <mem>
add <con>, <reg>
add <con>, <mem>
```

Examples

```
add $10, %eax     # EAX is set to EAX + 10

addb $10, (%eax) # add 10 to the single byte
                 # stored at memory address stored in EAX
```

sub — Integer subtraction

The sub instruction stores in the value of its second operand the result of subtracting the value of its first operand from the value of its second operand. As with add, whereas both operands may be registers, at most one operand may be a memory location.

sub — Integer subtraction

Syntax

```
sub <reg>, <reg>
sub <mem>, <reg>
sub <reg>, <mem>
sub <con>, <reg>
sub <con>, <mem>
```

Examples

```
sub %ah, %al     # AL is set to AL - AH

sub $216, %eax   # subtract 216 from the value
                 # stored in EAX
```

inc, dec — Increment, Decrement

The `inc` instruction increments the contents of its operand by one.
The `dec` instruction decrements the contents of its operand by one.

inc, dec — Increment, Decrement

Syntax

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Examples

```
dec %eax      # subtract one from the contents of EAX
incl var(,1)  # add one to the 32-bit integer
              # stored at location var
```

imul — Integer multiplication

The imul instruction has two basic formats:

- two-operand (first two syntax listings above)
- three-operand (last two syntax listings above)

imul — Integer multiplication

- two-operand (first two syntax listings above)
  - stores the result in the second operand.
  - **The result (i.e. second) operand must be a register.**

Syntax

```
imul <reg32>, <reg32>
imul <mem>, <reg32>
```

imul — Integer multiplication

- three-operand (last two syntax listings above).
  - Storeing the result in its last operand.
  - **The result operand must be a register.**
  - **The first operand is restricted to being a constant value.**

Syntax

```
imul <con>, <reg32>, <reg32>
imul <con>, <mem>, <reg32>
```

imul — Integer multiplication

Examples

```
imul (%ebx), %eax       # EAX = EBX * EAX
imul $25, %edi, %esi    # ESI =  EDI * 25
```

idiv — Integer division

The `idiv` instruction:

- divident: 64 bit integer EDX:EAX
  - EDX: the most significant four bytes,
  - EAX: the least significant four bytes)
- by the * divisor: operand value.
- quotient $\longrightarrow$ EAX,
- remainder $\longrightarrow$ EDX.

idiv — Integer division

Syntax

```
idiv <reg32>
idiv <mem>
```

Examples

```
idiv %ebx     # EDX:EAX / EBX
              # EAX = quotient, EDX = remainder.

idivw (%ebx) # EDX:EAS / (EBX)
              # (EBX) is data whose memory addr. is in EBX.
              # Place the quotient in EAX
              # and the remainder in EDX.
```

and, or, xor — Bitwise logical operation

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, **placing the result in the first operand location**.

and, or, xor — Bitwise logical operation

Syntax of **and**

```
and <reg>, <reg>
and <mem>, <reg>
and <reg>, <mem>
and <con>, <reg>
and <con>, <mem>
```

and, or, xor — Bitwise logical operation

Syntax of **or**

```
or <reg>, <reg>
or <mem>, <reg>
or <reg>, <mem>
or <con>, <reg>
or <con>, <mem>
```

and, or, xor — Bitwise logical operation

Syntax of **xor**

```
xor <reg>, <reg>
xor <mem>, <reg>
xor <reg>, <mem>
xor <con>, <reg>
xor <con>, <mem>
```

and, or, xor — Bitwise logical operation

Examples

```
and $0x0f, %eax # clear all but the last 4 bits of EAX.
xor %edx, %edx  # set the contents of EDX to zero.
```

not — Bitwise logical not

Logically negates the operand contents (that is, flips all bit values in the operand).

Syntax

```
not <reg>
not <mem>
```

Example

```
not %eax — flip all the bits of EAX
```

neg — Negate

Performs the two's complement negation of the operand contents.

Syntax

- neg
- neg

Example

```
neg %eax # EAX is set to (- EAX)
```

shl, shr — Shift left and right

These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros.
The shifted operand can be shifted up to 31 places.
The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL.
In either case, shifts counts of greater then 31 are performed modulo 32.

shl, shr — Shift left and right

Syntax

- shift left

```
shl <con8>, <reg>
shl <con8>, <mem>
shl %cl, <reg>
shl %cl, <mem>
```

- shift right

```
shr <con8>, <reg>
shr <con8>, <mem>
shr %cl, <reg>
shr %cl, <mem>
```

shl, shr — Shift left and right

Examples

```
shl $1, eax # Multiply the value of EAX by 2
            # (if the most significant bit is 0)

shr %cl, %ebx # EBX = EBX / (2^(n))
              # where n is the value in CL.
              # Note: for negative integers,
              # it is different from  the C semantics
              # of division!
```

Control Flow Instructions

The instruction pointer EIP register cannot be manipulated directly, but is updated implicitly by provided control flow instructions.

Label is location, but it is simplier than a 32-bit address.

Example

```
        mov 8(%ebp), %esi
begin:
        xor %ecx, %ecx
        mov (%esi), %eax
```

where **begin** is a label.

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax

```
jmp <label>
```

Example

```
jmp begin # Jump to the instruction labeled begin.
```

jcondition — Conditional jump

These instructions are conditional jumps that are based on the status of **condition codes** in the resigor call **machine status word**.

The contents of the machine status word include information about the last arithmetic operation performed.

Syntax

```
je <label>      # (jump when equal)
jne <label>     # (jump when not equal)
jz <label>      # (jump when last result was zero)
jg <label>      # (jump when greater than)
jge <label>     # (jump when greater than or equal to)
jl <label>      # (jump when less than)
jle <label>     # (jump when less than or equal to)
```

jcondition — Conditional jump

Example

```
cmp %ebx, %eax
jle done
```

If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*.

Otherwise, continue to the next instruction.

cmp — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately.

This instruction **is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.**

cmp — Compare

Syntax

```
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>
```

Example

```
cmpb $10, (%ebx)
jeq loop     # If the byte stored at the memory location
             # in EBX is equal to the integer constant
             # 10, jump to the location labeled loop.
```

call, ret — Subroutine call and return

These instructions implement a **subroutine** call and return.

call

The call instruction

1. first pushes the current code location onto stack in memory,
2. and then performs an unconditional jump to the code location indicated by the label operand.

Note, simple jump instructions do not saves the location for return.

ret

The ret instruction implements a subroutine return mechanism.

1. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details).
2. It then performs an unconditional jump to the retrieved code location.

call, ret

Syntax

```
* call <label>

* ret
```

Calling Convention

The calling convention is a protocol about how to call and return from routines.

- To allow separate programmers to share code and develop libraries for use by many programs, and to simplify the use of subroutines in general,
- Furthermore, given a set of calling convention rules, high-level language compilers can be made to follow the rules, thus allowing hand-coded assembly language routines and high-level language routines to call one another.

Calling convention of C language

The C calling convention is based heavily on the use of the stack.

- Subroutine parameters are passed on the stack.
- Registers are saved on the stack,
- and local variables used by subroutines are placed in memory on the stack.

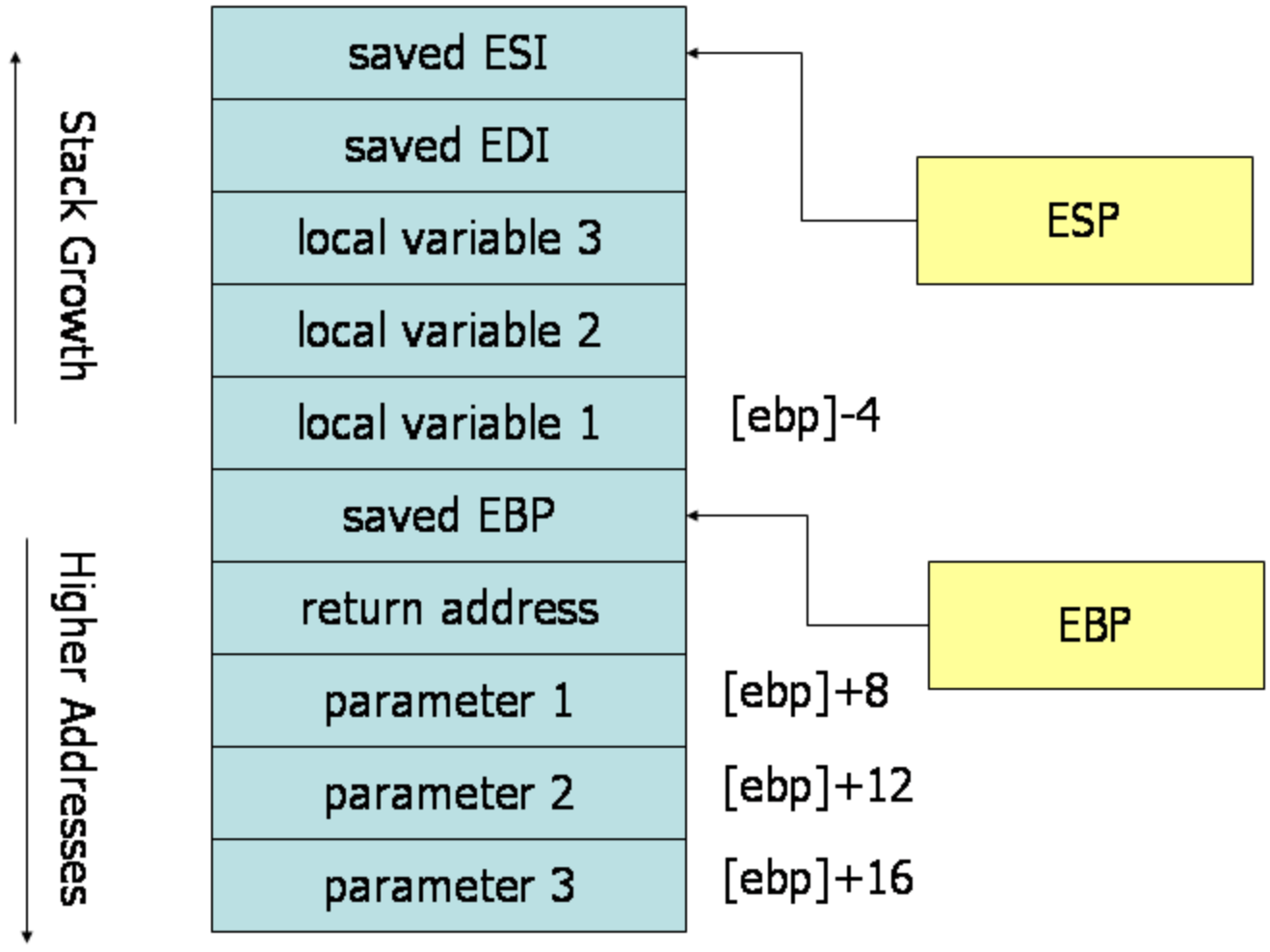
Ccalling conventions of other high-level languages are similar.

Calling convention

The calling convention is broken into two sets of rules.

- Caller Rules: the first set of rules is employed by the caller of the subroutine,
- Callee Rules: the second set of rules is observed by the writer of the subroutine (the callee).

It should be emphasized that mistakes in the observance of these rules quickly result in fatal program errors since the stack will be left in an inconsistent state; thus meticulous care should be used when implementing the call convention in your own subroutines.



Stack during Subroutine Call

The image above depicts the contents of the stack during the execution of a subroutine with three parameters and three local variables.
The cells depicted in the stack are 32-bit wide memory locations, thus the memory addresses of the cells are 4 bytes apart.
The first parameter resides at an offset of 8 bytes from the base pointer.
Above the parameters on the stack (and below the base pointer), the call instruction placed the return address, thus leading to an extra 4 bytes of offset from the base pointer to the first parameter.
When the retinstruction is used to return from the subroutine, it will jump to the return address stored on the stack.

Caller Rules

To make a subrouting call, the caller should:

1. Before calling a subroutine, the caller should save the contents of certain registers that are designated **caller-saved**.
    - The **caller-saved registers** are EAX, ECX, EDX.
      The called subroutine is allowed to modify these registers.
2. To pass parameters to the subroutine, push them (in inverted order) onto the stack () before the call.
3. Then invoke call the subroutine with **call** instruction.
    - The return address is placed on stack, and then branches to the subroutine code.

Caller Rules

After the subroutine returns (immediately following the call instruction), the caller can expect to find the **return value** of the subroutine in the register EAX.

To restore the machine state, the caller should:

1. Remove the parameters from stack. This restores the stack to its state before the call was performed.
2.
3. Restore the contents of caller-saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

Caller rules

Example

The caller is calling a function *myFunc* that takes three integer parameters.

- First parameter is in EAX,
- the second parameter is the constant 216;
- the third parameter is in the memory location stored in EBX.

```
push (%ebx)     # Push last parameter first
push $216       # Push the second parameter
push %eax       # Push first parameter last
call myFunc     # Call the function (assume C naming)

add $12, %esp
```

Caller rules

Example

```
push (%ebx)
push $216
push %eax
call myFunc

add $12, %esp  # Cleans up the parameters in stack
```

Note that after the call returns, the caller cleans up the stack using the add instruction. (12 bytes = 3 parameters * 4 bytes each)

The result produced by *myFunc* is now in the register EAX.

Callee Rules

prologue

The definition of the subroutine should adhere to the following rules at the beginning of the subroutine:

1. Push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

```
    push %ebp
    mov  %esp, %ebp
```

    - This initial action maintains the **base pointer**, EBP.
    - The base pointer is used by convention as a point of reference for finding parameters and local variables on the stack.

Callee Rules

prologue

2. Next, allocate local variables by making space on the stack.
3. Next, save the values of the **callee-saved** registers that will be used by the function. To save registers, push them onto the stack.
    - The callee-saved registers are EBX, EDI, and ESI.
    - (ESP and EBP will also be preserved by the calling convention, but need not be pushed on the stack during this step).

After these three actions are performed, the body of the subroutine may proceed.

Callee Rules

epilogue

When the subroutine is returns, it must follow these steps:

1. Leave the return value in EAX.
2. Restore the old values of any callee-saved registers (EDI and ESI) that were modified.
    - The register values are restored by popping off the stack.
3. Deallocate local variables.
    - one way: add the appropriate value to the stack pointer.
    - **better way**: `mov %ebp, %esp`. (This works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.)

Callee Rules

epilogue

4. Immediately before returning, restore the caller's base pointer value by popping EBP off the stack.
    - Recall that the first thing we did on entry to the subroutine was to push the base pointer to save its old value.
5. Finally, return to the caller by executing a ret instruction.
    - This instruction will find and remove the appropriate return address from the stack.

Callee Rules

Example

```
/* Start the code section */
.text

/* Define myFunc as a global (exported) function. */
.globl myFunc
.type myFunc, @function

myFunc:
/* Subroutine Prologue */
push %ebp       # Save the old base pointer value.
mov %esp, %ebp  # Set the new base pointer value.

sub $4, %esp    # Make room for one 4-byte
                # local variable.

push %edi       # Save the values of registers
                # that the function will modify.
push %esi       # This function uses EDI and ESI.

/* (no need to save EBX, EBP, or ESP) */
```

Example (Cont.)

Here is an example function definition that follows the callee rules:

```
/* Subroutine Body */
mov 8(%ebp), %eax    # Move parameter 1 into EAX.
mov 12(%ebp), %esi   # Move parameter 2 into ESI.
mov 16(%ebp), %edi   # Move parameter 3 into EDI.

mov %edi, -4(%ebp)   # Move EDI into the local variable.
add %esi, -4(%ebp)   # Add ESI into the local variable.
add -4(%ebp), %eax   # Add the contents of the local
                     # variable into EAX (final result).

/* Subroutine Epilogue */
pop %esi         # Recover register values.
pop %edi
mov %ebp, %esp # Deallocate the local variable.
pop %ebp         # Restore the caller's base pointer.

ret
```

Example (Cont.)

Subroutine prologue

The subroutine prologue performs the standard actions of

1. saving a snapshot of the stack pointer in EBP (the base pointer),
2. allocating local variables by **decrementing the stack pointer**,
3. and saving register values on the stack.

Example (Cont.)

Subroutine prologue

Allocation of parameters to subroutine

Parameters were placed onto the stack before the subroutine was called, they are always located below the base pointer (i.e. at higher addresses) on the stack.

- The 1st parameter is at memory location (EBP+8),
- The 2nd is at (EBP+12),
- The 3rd is at (EBP+16).

Example (Cont.)

Subroutine prologue

Allocation of parameters to subroutine

Local variables are allocated above the base pointer (i.e. at lower addresses) on the stack.

- The 1st local variable is always located at (EBP-4),
- The 2nd is at (EBP-8),
- and so on.

This conventional use of the base pointer allows us to quickly identify the use of local variables and parameters within a function body.

Example (Cont.)

Subroutine epilogue

The function epilogue is mirror image of the prologue.

1. The caller's register values are recovered from the stack,
2. the local variables are deallocated by resetting the stack pointer,
3. the caller's base pointer value is recovered,
4. and the ret instruction is used to return to the appropriate code location in the caller.

Credits

This guide was originally from Yale University.

http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html.