A Tiny Guide to Programming in 32-bit x86 Assembly Language

by Adam Ferrari [1]

1. Introduction

This small guide, in combination with the material covered in the class lectures on assembly language programming, should provide enough information to do the assembly language labs for this class. In this guide, we describe the basics of 32-bit x86 assembly language programming, covering a small but useful subset of the available instructions and assembler directives. However, real x86 programming is a large and extremely complex universe, much of which is beyond the useful scope of this class. For example, the vast majority of real (albeit older) x86 code running in the world was written using the 16-bit subset of the x86 instruction set. Using the 16-bit programming model can be quite complex—it has a segmented memory model, more restrictions on register usage, and so on. In this guide we'll restrict our attention to the more modern aspects of x86 programming, and delve into the instruction set only in enough detail to get a basic feel for programming x86 compatible chips at the hardware level.

2. Registers

Modern (i.e 386 and beyond) x86 processors have 8 32-bit general purpose registers, as depicted in Figure 1. The register names are mostly historical in nature. For example, EAX used to be called the "accumulator" since it was used by a number of arithmetic operations, and ECX was known as the "counter" since it was used to hold a loop index. Whereas most of the registers have lost their special purposes in the modern instruction set, by convention, two are reserved for special purposes—the stack pointer (ESP) and the base pointer (EBP).
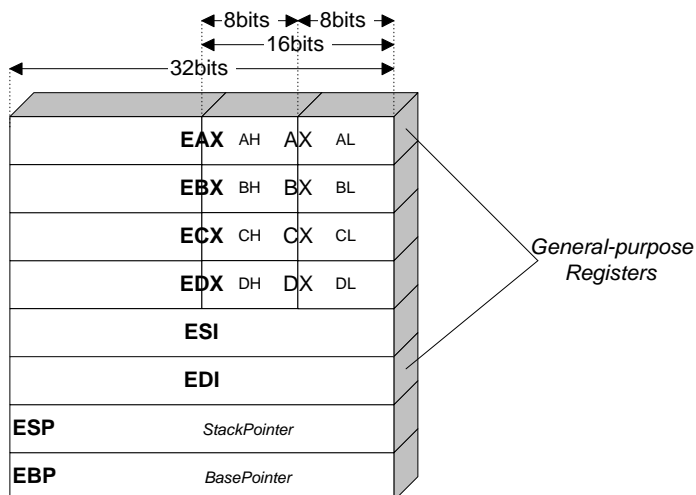

Figure 1. The x86 register set.

Insomecases,namelyEAX,EBX,ECX,andEDX,subsectionsoftheregistersmaybeused. Forexample,theleastsignificant2bytesofEAXcanbetreatedasa16-bitregistercalledAX. TheleastsignificantbyteofAXcanbeusedasasingle8-bitregistercalledAL,whilethemost significantbyteofAXcanbeusedasasingle8-bitregistercalledAH.Itisimportanttorealize thatthesenamesrefertothesamephysicalregister.Whenatwo-bytequantityisplacedintoDX, theupdateaffectsthevalueofEDX(inparticular,theleastsignificant16bitsofEDX).These "sub-registers"aremainlyhold-oversfromolder,16-bitversionsoftheinstructionset.However, theyaresometimesconvenientwhendealingwithdatathataresmallerthan32-bits(e.g.1-byte ASCIIcharacters).

WhenreferringtoregistersinassemblyLanguage,thenamesarenotcase-sensitive.Forexam-ple,thenamesEAXandeaxrefertothesameregister.

3.MemoryandAddressingModes

3.1.DeclaringStaticDataRegions

Youcandeclarestaticdataregions(analogoustoglobalvariables)inx86assemblyusingspe-cialassemblerdirectivesforthispurpose.Datadeclarationsshouldbeprecededbythe   .DATA directive.Followingthisdirective,thedirectives   DB, DW,and DDcanbeusedtodeclareone, two,andfourbytedatalocations,respectively.Declaredlocationscanbelabeledwithnamesfor laterreference-thisissimilartodeclaringvariablesbyname,butabidesbysomelowerlevel rules.Forexample,locationsdeclaredinsequencewillbelocatedinmemorynexttooneanother. Someexampledeclarationsaredepictedin Figure2.

```
        .DATA
var     DB 64    ; Declare a byte containing the value 64. Label the
                 ; memory location "var".
var2    DB ?     ; Declare an uninitialized byte labeled "var2".
        DB 10    ; Declare an unlabeled byte initialized to 10. This
                 ; byte will reside at the memory address var2+1.
X       DW ?     ; Declare an uninitialized two-byte word labeled "X".
Y       DD 3000  ; Declare 32 bits of memory starting at address "Y"
                 ; initialized to contain 3000.
Z       DD 1,2,3 ; Declare three 4-byte words of memory starting at
                 ; address "Z", and initialized to 1, 2, and 3,
                 ; respectively. E.g. 3 will be stored at address Z+8.
```
Figure2.Declaringmemoryregions

ThelastexampleinFigure2illustratesthedeclarationofanarray.Unlikeinhighlevellan-guageswherearrayscanhavemanydimensionsandareaccessedbyindices,arraysinassembly languagearesimplyanumberofcellslocatedcontiguouslyinmemory.Twoothercommonmeth-odsusedfordeclaringarraysofdataarethe   DUPdirectiveandtheuseofstringliterals.The   DUP directivetellstheassemblertoduplicateanexpressionagivennumberoftimes.Forexample,the statement" 4DUP(2) "isequivalentto"   2,2,2,2   ".Someexamplesofdeclaringarraysare depictedinFigure3.

```
bytes    DB 10  DUP(?)   ; Declare 10 uninitialized bytes starting at
                         ; the address "bytes".
arr      DD 100 DUP(0)   ; Declare 100 4 bytes words, all initialized to 0,
                         ; starting at memory location "arr".
str      DB 'hello',0    ; Declare 5 bytes starting at the address "str"
                         ; initialized to the ASCII character values for
                         ; the characters 'h', 'e', 'l', 'l', 'o', and
                         ; '\0'(NULL), respectively.
```
Figure 3. Declaring arrays in memory

3.2. Addressing Memory

Modern x86-compatible processors are capable of addressing up to 2^{32} bytes of memory; that is, memory addresses are 32-bits wide. For example, in Figure 2 and Figure 3, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 32-bit quantities that specify addresses in memory. In addition to supporting referring to memory regions by labels (i.e. constant values), the x86 provides a flexible scheme for computing and referring to memory addresses:

X86 Addressing Mode Rule - Up to two of the 32-bit registers and a 32-bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre-multiplied by 2, 4, or 8.

To see this memory addressing rule in action, we'll look at some example mov instructions. As we'll see later in Section 4.1, the mov instruction moves data between registers and memory. This instruction has two operands—the first is the destination (where we're moving data *to*) and the second specifies the source (where we're getting the data *from*). Some examples of mov instructions using address computations that obey the above rule are:

- `mov eax, [ebx]`   *;Move the 4 bytes in memory at the address contained in EBX into EAX*
- `mov [var], ebx`   *;Move the contents of EBX into the 4 bytes at memory address "var"*
                             ;(Note, "var" is a 32-bit constant).
- `mov eax, [esi-4]`   *;Move 4 bytes at memory address ESI+(-4) into EAX*
- `mov [esi+eax], cl`   *;Move the contents of CL into the byte at address ESI+EAX*
- `mov edx, [esi+4*ebx]`   *;Move the 4 bytes of data at address ESI+4*EBX into EDX*

Some examples of incorrect address calculations include:
- `mov eax, [ebx-ecx]`   *;Can only **add** register values*
- `mov [eax+esi+edi], ebx`   *;At most **2** registers in address computation*

3.3. Size Directives

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, in all of the above instructions, the size of the memory regions could be inferred from the size of the register operand—when we were loading a 32-bit register, the assembler could infer that the region of memory we were referring to was 4 bytes wide. When we were storing the value of a one byte register to memory, the assembler could infer that we wanted the address to refer to a single byte in memory. However, in some cases the size of a referred-to memory region is ambiguous. Consider the following instruction:

```
mov [ebx], 2
```

Shouldthisinstructionmovethevalue2intothesinglebyteataddressEBX?Perhapsitshould movethe32-bitintegerrepresentationof2intothe4-bytesstartingataddressEBX.Sinceeither isavalidpossibleinterpretation,theassemblermustbeexplicitlydirectedastowhichiscorrect. Thesizedirectives `BYTE PTR`, `WORD PTR`,and `DWORD PTR`servethispurpose.Forexample:

- `mov BYTE PTR [ebx], 2`   *;Move2intothesinglebyteatmemorylocationEBX*
- `mov WORD PTR [ebx], 2`   *;Movethe16-bitintegerrepresentationof2intothe2bytesstartingat*
  ;addressEBX
- `mov DWORD PTR [ebx], 2`   *;Movethe32-bitintegerrepresentationof2intothe4bytesstartingat*
  ;addressEBX

4.Instructions

Machineinstructionsgenerallyfallintothreecategories:datamovement,arithmetic/logic,and control-flow.Inthissection,wewilllookatimportantexamplesofx86instructionsfromeachcat-egory.Thissectionshouldnotbeconsideredanexhaustivelistofx86instructions,butrathera usefulsubset.

Inthissection,wewillusethefollowingnotation:

- <reg32>   -meansany32-bitregisterdescribedinSection2,forexample,   `ESI`.
  <reg16>   -meansany16-bitregisterdescribedinSection2,forexample,   `BX`.
  <reg8>   -meansany8-bitregisterdescribedinSection2,forexample   `AL`.
  <reg>   -meansanyoftheabove.
- <mem>   -willrefertoamemoryaddress,asdescribedinSection3,forexample   `[EAX]`,or
  `[var+4]`,or `DWORD PTR [EAX+EBX]`.
- <con32>   -meansany32-bitconstant.
  <con16>   -meansany16-bitconstant.
  <con8>   -meansany8-bitconstant.
  <con>   -meansanyoftheabovesizedconstants.

4.1.DataMovementInstructions

Instruction: `mov`

Syntax: `mov <reg>,<reg>`
                `mov <reg>,<mem>`
                `mov <mem>,<reg>`
                `mov <reg>,<const>`
                `mov <mem>,<const>`

Semantics: The movinstructionmovesthedataitemreferredtobyitssecondoperand(i.e. registercontents,memorycontents,oraconstantvalue)intothelocationreferred tobyitsfirstoperand(i.e.aregisterormemory).Whileregister-to-registermoves arepossible,directmemory-to-memorymovesarenot.Incaseswherememory transfersaredesired,thesourcememorycontentsmustfirstbeloadedintoaregis-ter,thencanbestoredtothedestinationmemoryaddress.

Examples: `mov eax, ebx`   ` ; transfer ebx to eax`
                `mov BYTE PTR [var], 5` ` ; store the value 5 into the byte at`
                                        ` ; memory location "var"`

Instruction: `push`

Syntax:	`push <reg32>`
	`push <mem>`
	`push <con32>`

Semantics: The pushinstructionplacesitsoperandontothetopofthehardwaresupported stackinmemory.Specifically, pushfirstdecrementsESPby4,thenplacesitsoperandintothecontentsofthe32-bitlocationataddress[ESP].ESP(thestack pointer)isdecrementedbypushsincethex86stackgrowsdown-i.e.thestack growsfromhighaddressestoloweraddresses.

Examples:
```
push eax    ; push the contents of eax onto the stack
push [var]  ; push the 4 bytes at address "var" onto the stack
```

Instruction: `pop`

Syntax:	`pop <reg32>`
	`pop <mem>`

Semantics: The popinstructionremovesthe4-bytedataelementfromthetopofthehardwaresupportedstackintothespecifiedoperand(i.e.registerormemorylocation).Specifically,popfirstmovesthe4byteslocatedatmemorylocation[SP]intothespecifiedregisterormemorylocation,andthenincrementsSPby4.

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.
```

Instruction: `lea`

Syntax:	`lea <reg32>,<mem>`

Semantics: The leainstructionplacesthe *address*specifiedbyitssecondoperandintotheregisterspecifiedbyitsfirstoperand.Note,the *contents*ofthememorylocationare notloaded—onlytheeffectiveaddressiscomputedandplacedintotheregister. Thisisusefulforobtaininga"pointer"intoamemoryregion.

Examples:
```
lea eax, [var]       ; the address of "var" is places in EAX.
lea edi, [ebx+4*esi] ; the quantity EBX+4*ESI is placed in EDI.
```

4.2.ArithmeticandLogicInstructions

Instruction: `add, sub`

Syntax:	`add <reg>,<reg>`	`sub <reg>,<reg>`
	`add <reg>,<mem>`	`sub <reg>,<mem>`
	`add <mem>,<reg>`	`sub <mem>,<reg>`
	`add <reg>,<con>`	`sub <reg>,<con>`
	`add <mem>,<con>`	`sub <mem>,<con>`

Semantics: The addinstructionaddstogetheritstwooperands,storingtheresultinitsfirst operand.Similarly,the subinstructionsubtractsitssecondoperandfromitsfirst. Note,whereasbothoperandsmayberegisters,atmostoneoperandmaybea memorylocation.

Examples:
```
add eax, 10    ; add 10 to the contents of EAX.
sub [var], esi ; subtract the contents of ESI from the 32-bit
               ; integer stored at memory location "var".
```

```
                    add BYTE PTR [var], 10  ; add 10 to the single byte stored
                                            ;  at memory address "var".
```

Instruction: `inc, dec`

Syntax: `inc <reg>              dec <reg>`
                `inc <mem>              dec <mem>`

Semantics: The incinstructionincrementsthecontentsofitsoperandbyone,andsimilarly
                decdecrementsthecontentsofitsoperandbyone.

Examples: `dec eax                ; subtract one from the contents of EAX.`
                `inc DWORD PTR [var]    ; add one to the 32-bit integer stored at`
                `                       ; memory location "var".`

Instruction: `imul`

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

Semantics: The imulinstructionhastwobasicformats:two-operand(firsttwosyntaxlistings
                above)andthree-operand(lasttwosyntaxlistingsabove).
                Thetwo-operandformmultipliesitstwooperandstogetherandstorestheresultin
                thefirstoperand.Theresult(i.e.first)operandmustbearegister.
                Thethreeoperandformmultipliesitssecondandthirdoperandstogetherand
                storestheresultinitsfirstoperand.Again,theresultoperandmustbearegister.
                Furthermore,thethirdoperandisrestrictedtobeingaconstantvalue.

Examples: `imul eax, [var]   ; multiply the contents of EAX by the 32-bit`
                `                  ; contents of the memory location "var". Store`
                `                  ; the result in EAX.`
                `imul esi, edi, 25 ; multiply the contents of EDI by 25. Store the`
                `                  ; result in ESI.`

Instruction: `idiv`

Syntax: `idiv <reg32>`
                `idiv <mem>`

Semantics: The idivinstructionisusedtodividethecontentsofthe64bitintegerEDX:EAX
                (constructedbyviewingEDXasthemostsignificantfourbytesandEAXasthe
                leastsignificantfourbytes)bythespecifiedoperandvalue.Thequotientresultof
                thedivisionisstoredintoEAX,whiletheremainderisplacedinEDX.This
                instructionmustbeusedwithcare.Beforeexecutingtheinstruction,theappropri-
                atevaluetobedividedmustbeplacedintoEDXandEAX.Clearly,thisvalueis
                overwrittenwhenthe idivinstructionisexecuted.

Examples: `idiv  ebx   ; divide the contents of EDX:EAX by the contents of`
                `            ; EBX. Place the quotient in EAX and the remainder`
                `            ; in EDX.`
                `idiv DWORD PTR [var] ; same as above, but divide by the 32-bit`
                `                     ; value stored at memory location "var".`

Instruction: `and, or, xor`

Syntax: `and <reg>,<reg>   or <reg>,<reg>   xor <reg>,<reg>`
               `and <reg>,<mem>   or <reg>,<mem>   xor <reg>,<mem>`
               `and <mem>,<reg>   or <mem>,<reg>   xor <mem>,<reg>`
               `and <reg>,<con>   or <reg>,<con>   xor <reg>,<con>`
               `and <mem>,<con>   or <mem>,<con>   xor <mem>,<con>`

Semantics: Theseinstructionsperformthespecifiedlogicaloperation(logicalbitwiseand,or, andexclusiveor,respectively)ontheiroperands,placingtheresultinthefirst operandlocation.

Examples: `and   eax, 0fH    ; clear all but the last 4 bits of EAX.`
               `xor   edx, edx    ; set the contents of EDX to zero.`

Instruction: `not`

Syntax: `not <reg>`
               `not <mem>`

Semantics: Performsthelogicalnegationoftheoperandcontents(i.e.flipsallbitvalues).

Examples: `not BYTE PTR [var]; negate all bits in the byte at the memory`
               `                 ; location "var".`

Instruction: `neg`

Syntax: `neg <reg>`
               `neg <mem>`

Semantics: Performsthearithmetic(i.e.two'scomplement)negationoftheoperandcontents.

Examples: `neg eax     ; negate the contents of EAX.`

Instruction: `shl, shr`

Syntax: `shl <reg>,<con8>     shr <reg>,<con8>`
               `shl <mem>,<con8>     shr <mem>,<con8>`
               `shl <reg>,cl         shr <reg>,cl`
               `shl <mem>,cl         shr <mem>,cl`

Semantics: Theseinstructionsshiftthebitsintheirfirstoperand'scontentsleftandright(shl and shr,respectively),paddingtheresultingemptybitpositionswithzeros.The shiftedoperandcanbeshiftedupto31places.Thenumberofbitstoshiftisspeci- fiedbythesecondoperand,whichcanbeeitheran8-bitconstantortheregister CL.Ineithercase,shiftscountsofgreaterthen31areperformedmodulo32.

4.3.ControlFlowInstructions

Inthissection,wewillrefertolabeledlocationsintheprogramtextas   <label>.Labelscanbe insertedanywhereinx86assemblycodetextbyenteringalabelnamefollowedbyacolon.For example,considerthecodefragmentinFigure4.Thesecondinstructioninthiscodefragmentis labeled" begin".Elsewhereinthecode,wecanrefertothememorylocationthatthisinstructionis locatedatinmemoryusingthemoreconvenientsymbolicname"   begin"insteadofhavingtorefer

tothememoryaddressasaninteger.

```
                        mov esi, [ebp+8]
            begin:   xor ecx, ecx
                        mov eax, [esi]
```

Figure4.Alabeledcodelocation

Instruction: `jmp`

Syntax: `jmp <label>`
Semantics: Transfersprogramcontrolflowtotheinstructionatthememorylocationindicated
bytheoperand.
Examples: `jmp begin`

Instruction: `jCC`

Syntax: `je <label>`   –Jumpwhenequal
`jne <label>`   –Jumpwhennotequal
`jz <label>`   –Jumpwhenlastresultwaszero
`jg <label>`   –Jumpwhengreaterthan
`jge <label>`   –Jumpwhengreaterthanorequalto
`jl <label>`   –Jumpwhenlessthan
`jle <label>`   –Jumpwhenlessthanorequalto

Semantics: Theseinstructionsareconditionaljumpsthatarebasedonthestatusofaseto
conditioncodes thatarestoredinaspecialregistercalledthe *machinestatusword* .
Thecontentsofthemachinestatuswordincludeinformationaboutthelastarith-
meticoperationperformed.Forexample,onebitofthiswordindicatesifthelast
resultwaszero.Anotherindicatesifthelastresultwasnegative.Basedonthese
conditioncodes,anumberofconditionaljumpscanbeperformed.Forexample,
the jzinstructionperformsajumptothespecifiedoperandlabeliftheresultofthe
lastarithmeticoperation(e.g.   add, sub,etc.)waszero.Otherwise,controlproceeds
tothenextinstructioninsequenceafterthe   jz.Theseconditionaljumpsarethe
underlyingsupportneededtoimplementhigh-levellanguagefeaturessuchas"if"
statementsandloops(e.g."while"and"for").
Anumberoftheconditionalbranchesaregivennamesthatareintuitivelybasedon
thelastoperationperformedbeingaspecialcompareinstruction,   cmp(seebelow).
Forexample,conditionalbranchessuchas   jleand jnearebasedonfirstperforming
a cmpoperationonthedesiredoperands.

Examples:
```
cmp eax, ebx      ; if the contents of eax are less than or equal
jle done          ; to the contents of EBX, jump to the code
                  ; location labeled "done".
```

Instruction: `cmp`

Syntax: `cmp <reg>,<reg>`

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

Semantics: Comparesthetwospecifiedoperands,settingtheconditioncodesinthemachine statuswordappropriately.Infact,thisinstructionisequivalenttothe subinstruction,excepttheresultofthesubtractionisdiscarded.

Examples:
```
cmp DWORD PTR [var], 10 ; if the 4 bytes stored at memory location
jeq loop                ; "var" equal the 4-byte integer value 10,
                        ; then jump to the code location labeled
                        ; loop
```

Instruction: `call`

Syntax: `call <label>`

Semantics: Thisinstructionimplementsasubroutinecallthatoperatesincooperationwiththe subroutinereturninstruction, ret,describedbelow.Thisinstructionfirstpushesthe currentcodelocationontothehardwaresupportedstackinmemory(seethe push instructionfordetails),andthenperformsanunconditionaljumptothecodeloca-tionindicatedbythelabeloperand.Theaddedvalueofthisinstruction(ascom-paredtothesimple jmpinstruction)isthatitsavesthelocationtoreturntowhen thesubroutinecompletes.

Examples: `call my_subroutine`

Instruction: `ret`

Syntax: `ret`

Semantics: Incooperationwiththe callinstruction,the retinstructionimplementsasubroutine returnmechanism.Thisinstructionfirstpopsacodelocationoffthehardwaresup-portedin-memorystack(seethe popinstructionfordetails).Itthenperformsan unconditionaljumptotheretrievedcodelocation.

Examples: `ret`

5.BasicProgramStructure

Giventheaboverepertoireofinstructions,youareinapositiontoexaminethebasicskeletal structureofanassemblylanguagesubroutinesuitableforlinkingintoC++code.UnlikeC++, whichisoftenusedforthedevelopmentofcompletesoftwaresystems,assemblylanguageismost oftenusedincooperationwithotherlanguagessuchasFortran,C,andC++.Commonly,mostof aprojectisimplementedinthemoreconvenienthigh-levellanguage,andassemblylanguageis usedsparinglytoimplementextremelylow-levelhardwareinterfacesorperformance-critical "innerloops."Thus,inadditiontounderstandinghowtoprograminassemblylanguage,itis equallyimportanttounderstandhowtolinkassemblylanguagecodeintohigh-levellanguage programs.

Beforeexaminingthelinkageconventions,wemustfirstexaminethebasicstructureofan assemblylanguagefile.Todothis,wecancompareaverysimpleassemblylanguagefiletoan equivalentC++file.InFigure5weseetwofiles,oneinC++,theotherinx86assembly.Eachfile

includesafunction(albeitanuglyone)toreturntheintegervalue2.

```
int var = 2;                              .486
                                          .MODEL FLAT

extern "C" int returnTwo();
int returnTwo()                           .DATA
{                                         var   DD    2
   return var;
}                                         .CODE
                                          PUBLIC _returnTwo
                                          _returnTwo PROC
                                            mov   eax, [var]
                                            ret
                                          _returnTwo ENDP
                                          END
```

Figure5.Equivalentfunctionsthatreturnthevalue2,
implementedinC++(left)andx86(right).

Thetopoftheassemblyfilecontainstwodirectivesthatindicatetheinstructionsetandmemorymodelwewilluseforallworkinthisclass(note,thereareotherpossibilities—onemightuse onlytheolder80286instructionsetforwidercompatibility,forexample).

Next,whereintheC++filewefindthedeclarationoftheglobalvariable"   var",intheassemblyfilewefindtheuseofthe.   DATAand DDdirectives(describedinSection3.1)toreserveand initializea4-byte(i.e.integer-sized)memoryregionlabeled"   var".

Nextineachfile,wefindthedeclarationofthefunctionnamed   returnTwo.IntheC++filewe havedeclaredthefunctiontobe   extern"C".ThisdeclarationindicatesthattheC++compiler shoulduseCnamingconventionswhenlabelingthefunction   returnTwointheresultingobjectfile thatitproduces.Infact,thisnamingconventionmeansthatthefunction   returnTwoshouldmapto thelabel _returnTwointheobjectcode.Intheassemblycode,wehavelabeledthebeginningof thesubroutine _returnTwousingthe PROCdirective,andhavedeclaredthelabel   _returnTwoto bepublic.Again,theresultoftheseactionswillbethatthesubroutinewillmaptothesymbol _returnTwointheobjectcodethattheassemblergenerates.

Thefunctionbodiesarestraight-forward.AswewillseeinmoredetailinSection6,return valuesforfunctionsareplacedintoEAXbyconvention,hencetheinstructiontomovethecontentsof" var"intoEAXintheassemblycode.

Giventheseequivalentfunctiondefinitions,useofeitherversionofthefunctionisthesame.A samplecalltothefunction   returnTwoisdepictedinFigure6.ThisC++codecouldbelinkedto eitherdefinitionofthefunctionandwouldproducethesameresults(note,wecouldnotlinkto bothdefinitions,orthelinkerwouldproducea"multiplydefinedsymbol"error.Themechanicsof programlinkingwillbediscussedinanassociateddocumentthatrelatestothespecificprogram-

mingenvironmentthatyouwillusetoassembleandrunprograms.

```
#include <iostream.h>
extern "C" int returnTwo();
int main()
{
   if(returnTwo()!=2) {
      cerr << "Does not compute!\n";
   }
}
```

Figure 6. Calling returnTwo from C++.

6. Subroutine Calling Convention

6.1. What is a Calling Convention?

InSection5wesawasimpleexampleofasubroutinedefinedinx86assemblylanguage.In fact,thissubroutinewasquitesimple—itdidnotmodifyanyregistersexceptEAX(whichwas neededtoreturntheresult),anditdidnotcallanyothersubroutines.Inpractice,suchsimple functiondefinitionsarerarelyuseful.Whenmorecomplexsubroutinesarecombinedinasingle program,anumberofcomplicatingissuesarise.Forexample,howareparameterspassedtoa subroutine?Cansubroutinesoverwritethevaluesinaregister,ordoesthecallerexpecttheregis- tercontentstobepreserved?Whereshouldlocalvariablesinasubroutinebestored?Howshould resultsbereturnedfromfunctions?

Toallowseparateprogrammerstosharecodeanddeveloplibrariesforusebymanyprograms, andtosimplifytheuseofsubroutinesingeneral,programmerstypicallyadoptacommon *calling convention.*Thecallingconventionissimplyasetofrulesthatanswerstheabovequestionswith- outambiguitytosimplifythedefinitionanduseofsubroutines.Forexample,givenasetofcalling conventionrules,aprogrammerneednotexaminethedefinitionofasubroutinetodeterminehow parametersshouldbepassedtothatsubroutine.Furthermore,givenasetofcallingconvention rules,high-levellanguagecompilerscanbemadetofollowtherules,thusallowinghand-coded assemblylanguageroutinesandhigh-levellanguageroutinestocalloneanother.

Inpractice,evenforasingleprocessorinstructionset,manycallingconventionsarepossible. Inthisclasswewillexamineanduseoneofthemostimportantconventions:theClanguagecall- ingconvention.Understandingthisconventionwillallowyoutowriteassemblylanguagesubrou- tinesthataresafelycallablefromCandC++code,andwillalsoenableyoutocallClibrary functionsfromyourassemblylanguagecode.

6.2. The C Calling Convention

TheCcallingconventionisbasedheavilyontheuseofthehardware-supportedstack.To understandtheCcallingconvention,youshouldfirstmakesurethatyoufullyunderstandthe push, pop, call,and retinstructions—thesewillbethebasisformostoftherules.Inthiscalling convention,subroutineparametersarepassedonthestack.Registersaresavedonthestack,and localvariablesusedbysubroutinesareplacedinmemoryonthestack.Infact,thisstack-centric implementationofsubroutinesisnotuniquetotheClanguageorthex86architecture.Thevast majorityofhigh-levelprocedurallanguagesimplementedonmostprocessorshaveusedsimilar callingconvention.

The calling convention is broken into two sets of rules. The first set of rules is employed by the caller of the subroutine, and 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; thus meticulous care should be used when implementing the call convention in your own subroutines.

6.2.1. The Caller's Rules

The caller should adhere to the following rules when invoking a subroutine:

1. Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*. The caller-saved registers are EBX, ECX, EDX. If you want the contents of these registers to be preserved across the subroutine call, push them onto the stack.
1. To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first)—since the stack grows down, the first parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).
2. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.
3. After the subroutine returns, (i.e. immediately following the call instruction) the caller must remove the parameters from stack. This restores the stack to its state before the call was performed.
4. The caller can expect to find the return value of the subroutine in the register EAX.
5. The caller restores the contents of caller-saved registers (EBX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

6.2.2. The Callee's Rules

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

1. At the beginning of the subroutine, the function should push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

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

   The reason for this initial action is the maintenance of 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. Essentially, when any subroutine is executing, the base pointer is a "snapshot" of the stack pointer value from when the subroutine started executing. Parameters and local variables will always be located at known, constant offsets away from the base pointer value. We push the old base pointer value at the beginning of the subroutine so that we can later restore the appropriate base pointer value for the caller when the subroutine returns. Remember, the caller isn't expecting the subroutine to change the value of the base pointer. We then move the stack pointer into EBP to obtain our point of reference for accessing parameters and local variables.
2. Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented. The amount by which the stack pointer is decremented depends on the number of local variables needed. For

example, if 3 local integers (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables. I.e:

```
sub esp, 12
```

As with parameters, local variables will be located at known offsets from the base pointer.

3. Next, the values of any registers that are designated *callee-saved* that will be used by the function must be saved. To save registers, push them onto the stack. The callee-saved registers are EDI and ESI (ESP and EBP will also be preserved by the call convention, but need not be pushed on the stack during this step).

   After these three actions are performed, the actual operation of the subroutine may proceed. When the subroutine is ready to return, the call convention rules continue:

4. When the function is done, the return value for the function should be placed in EAX if it is not already there.

5. The function must restore the old values of any callee-saved registers (EDI and ESI) that were modified. The register contents are restored by popping them from the stack. Note, the registers should be popped in the inverse order that they were pushed.

6. Next, we deallocate local variables. The obvious way to do this might be to add the appropriate value to the stack pointer (since the space was allocated by subtracting the needed amount from the stack pointer). In practice, a less error-prone way to deallocate the variables is to move the value in the base pointer into the stack pointer, i.e.:

```
mov esp, ebp
```

   This trick works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.

7. Immediately before returning, we must restore the caller's base pointer value by popping EBP off the stack. Remember, the first thing we did on entry to the subroutine was to push the base pointer to save its old value.

8. Finally, we return to the caller by executing a `ret` instruction. This instruction will find and remove the appropriate return address from the stack.

It might be noted that the callee's rules fall cleanly into two halves that are basically mirror images of one another. The first half of the rules apply to the beginning of the function, and are therefor commonly said to define the *prologue* to the function. The latter half of the rules apply to the end of the function, and are thus commonly said to define the *epilogue* of the function.

6.2.3. Call Convention Example

The above rules may seem somewhat abstract on first examination. In practice, the rules become simple to use when they are well understood and familiar. To start the process of better understanding the call convention, we now examine a simple example of a subroutine call and a subroutine definition.

In Figure 7 a sample function call is depicted. Note how the caller pushes the parameters onto

thestackininvertedorderbeforethecall.The   callinstructionisusedtojumptothebeginningof

```
; Want to call a function "myFunc" that takes three
; integer parameters. First parameter is in EAX.
; Second parameter is the constant 123. Third
; parameter is in memory location "var"

  push  [var]  ; Push last parameter first
  push  123
  push  eax    ; Push first parameter last

  call _myFunc ; Call the function (assume C naming)

; On return, clean up the stack. We have 12 bytes
; (3 parameters * 4 bytes each) on the stack, and the
; stack grows down. Thus, to get rid of the parameters,
; we can simply add 12 to the stack pointer

  add   esp, 12

; The result produced by "myFunc" is now available for
; use in the register EAX. No other register values
; have changed
```

Figure7.Examplefunctioncall,caller'srulesobeyed

thesubroutineinanticipationofthefactthatthesubroutinewillusethe   retinstructiontoreturn whenthesubroutinereturns.Whenthesubroutinereturns,theparametersmustberemoved fromthestack.Asimplewaytodothisistoaddtheappropriateamounttothestackpointer(since thestackgrowsdown).Finally,theresultisavailableinEAX.

Relativetothecaller'srules,thecallee'srulesaresomewhatmorecomplex.Anexamplesub-routineimplementationthatobeysthecallee'srulesisdepictedinFigure8.Thesubroutinepro-logueperformsthestandardactionsofsavingasnapshotofthestackpointerinEBP(thebase pointer),allocatinglocalvariablesbydecrementingthestackpointer,andsavingregistervalues onthestack.

Inthebodyofthesubroutinewecannowmoreclearlyseetheuseofthebasepointerillus-trated.Bothparametersandlocalvariablesarelocatedatconstantoffsetsfromthebasepointerfor thedurationofthesubroutinesexecution.Inparticular,wenoticethatsinceparameterswere placedontothestackbeforethesubroutinewascalled,theyarealwayslocatedbelowthebase pointer(i.e.athigheraddresses)onthestack.Thefirstparametertothesubroutinecanalwaysbe foundatmemorylocation[EBP+8],thesecondat[EBP+12],thethirdat[EBP+16],andsoon. Similarly,sincelocalvariablesareallocatedafterthebasepointerisset,theyalwaysresideabove thebasepointer(i.e.atloweraddresses)onthestack.Inparticular,thefirstlocalvariableis alwayslocatedat[EBP-4],thesecondat[EBP-8],andsoon.Understandingthisconventionaluse ofthebasepointerallowsustoquicklyidentifytheuseoflocalvariablesandparameterswithina functionbody.

Thefunctionepilogue,asexpected,isbasicallyamirrorimageofthefunctionprologue.The caller'sregistervaluesarerecoveredfromthestack,thelocalvariablesaredeallocatedbyreset-

tingthestackpointer,thecaller'sbasepointervalueisrecovered,andtheretinstructionisusedto returntotheappropriatecodelocationinthecaller.

```
.486
.MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
    ; *** Standard subroutine prologue ***
  push ebp     ; Save the old base pointer value.
  mov ebp, esp ; Set the new base pointer value.
  sub esp, 4   ; Make room for one 4-byte local variable.
  push edi     ; Save the values of registers that the function
  push esi     ; will modify. This function uses EDI and ESI.
               ; (no need to save EAX, EBP, or ESP)

    ; *** Subroutine Body ***
  mov eax, [ebp+8] ; Put value of parameter 1 into EAX
  mov esi, [ebp+12]; Put value of parameter 2 into ESI
  mov edi, [ebp+16]; Put value of parameter 3 into EDI

  mov [ebp-4], edi ; Put EDI into the local variable
  add [ebp-4], esi ; Add ESI into the local variable
  add eax, [ebp-4] ; Add the contents of the local variable
                   ; into EAX (final result)

    ; *** Standard subroutine epilogue ***
  pop esi        ; Recover register values
  pop edi
  mov esp, ebp ; Deallocate local variables
  pop ebp        ; Restore the caller's base pointer value
  ret
_myFunc ENDP
END
```
Figure8.Examplefunctiondefinition,callee'srulesobeyed

Agoodwaytovisualizetheoperationofthecallingconventionistodrawthecontentsofthe nearbyregionofthestackduringsubroutineexecution.Figure9depictsthecontentsofthestack duringtheexecutionofthebodyof   myFunc(depictedinFigure8).Notice,loweraddressesare depictedlowerinthefigure,andthusthe"top"ofthestackisthebottom-mostcell.Thiscorre- spondsvisuallytotheintuitivestatementthatthex86hardwarestack"growsdown."Thecells depictedinthestackare32-bitwidememorylocations,thusthememoryaddressesofthecellsare 4bytesapart.Fromthispictureweseeclearlywhythefirstparameterresidesatanoffsetof8 bytesfromthebasepointer.Abovetheparametersonthestack(andbelowthebasepointer),the callinstructionplacedthereturnaddress,thusleadingtoanextra4bytesofoffsetfromthebase pointertothefirstparameter.
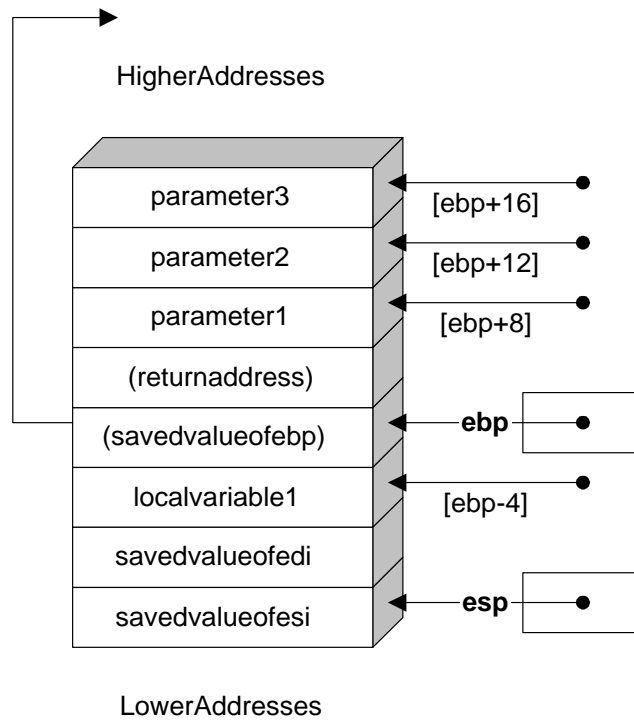
Figure9.Apictureofthestackinmemoryduringthe
executionofthebodyof   myFunc.