CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

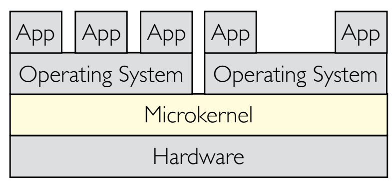
Week 1: Introduction, Assembly Language

Copyright Notice

Introduction and Goals

Origins

- For a long time, a group of us at PSU have been looking at the role that high-level programming languages can play in the construction of (very) low-level software.
- By using **high-level languages**, we can hope to increase programmer productivity, and improve software quality
- By focussing on very **low-level software**, we hope to provide strong foundations for the complete software stack

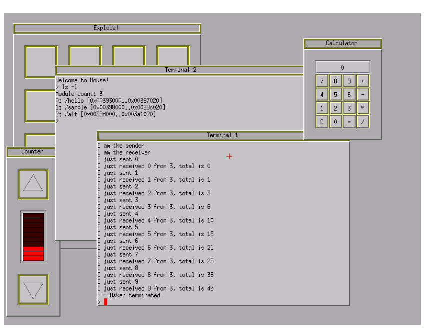


House (2005)

Kernel, GUI, drivers, network stack, and apps

Boots and runs in a bare metal environment

… all written in Haskell, a "purely functional" programming language



Why "House"?

"The Haskell User's Operating System Environment"

You are more secure in a house …

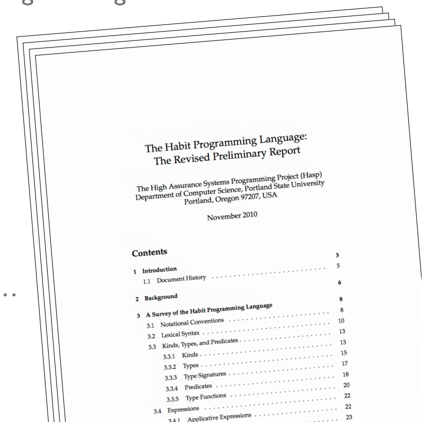
than if you only have Windows …

Performance concerns

- By design, higher-level languages abstract away from the details of how the underlying machine works
- Can we obtain the levels of performance and predictability that are typically required/expected in the systems programming domain?
- Can we write good systems software in a language that intentionally distances users from details of memory layout, representation, instruction selection, alignment, caching, etc.?
- Traditional approaches to building system software resort to using old, low-level languages like assembly and C
- Do "modern" languages have anything to offer in this area?

The Habit programming language

- "a dialect of Haskell that is designed to meet the needs of high assurance systems programming"
- How do you design a programming language for a specific domain?
- Experiment with existing languages
- Understand the domain …


The Habit Programming Language:
The Revised Preliminary Report

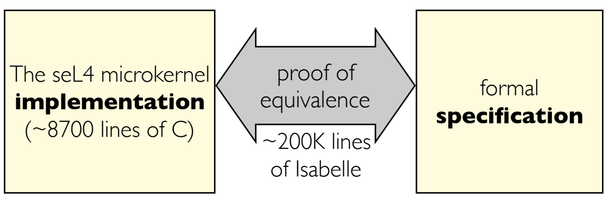
The High Assurance Systems Programming Project (Hasp)
Department of Computer Science, Portland State University
Portland, Oregon 97207, USA

November 2010

Contents

The seL4 experience

- In 2009, a group from NICTA, UNSW, and OK Labs in Australia announced seL4, as "the world's first operating system kernel with an end-to-end proof of implementation correctness and security enforcement."



- A landmark achievement for formal verification, and a strong foundation for building trustworthy systems

seL4 and capabilities

- Even without the verification result, the design of seL4 is interesting in its own right:
  - seL4 is a "capability enhanced" version of an earlier microkernel design called L4
  - The "capability" abstraction in seL4 provides facilities for implementing "least privilege" security policies and novel mechanisms for controlling resource usage

Safety properties for "free"?

- Security properties established in the seL4 verification include:
  - Absence of buffer overflows
  - Absence of null pointer dereferences
  - Absence of code injection attacks
  - …
- Many of these properties could be established for "free" if the implementation had been written in a "safer" language
- How might things be different if we built something like seL4 in Habit?

The CEMLaBS project

- "Using a Capability-Enhanced Microkernel as a Testbed for Language-Based Security"
- Started October 2014, funded by The National Science Foundation
- Three main questions:
  - **Feasibility**: Is it possible to build an inherently "unsafe" system like seL4 in a "safe" language like Habit?
  - **Benefit**: What benefits might this have, for example, in reducing verification costs?
  - **Performance**: Is it possible to meet reasonable performance goals for this kind of system?

Course description

- **An overview of conventional low-level programming techniques (1-5):**
  - Bare metal programming
  - Fundamental programmable hardware components
- **Case studies of practical microkernel implementations (6-8):**
  - OS abstractions (address spaces, threads, capabilities, …)
  - The L4 and seL4 microkernels
- **Reflections on the design of programming languages for this application domain (9-12):**
  - Assembly, C, Rust, Habit, domain specific languages, …

Course learning objectives

Upon the successful completion of this course, students will be able to:

1. Write simple programs that can run in a bare-metal environment using low-level programming languages.
2. Discuss common challenges in low-level systems software development, including debugging in a bare-metal environment.
3. Explain how conventional operating system features (multiple address spaces, context switching, protection, etc.) motivate the desire for (and benefit from) hardware support.

Course learning objectives, continued

4. Develop code to configure and use programmable hardware components such as a memory management unit (MMU), interrupt controller (PIC), and interval timer (PIT).
5. Describe the key steps in a typical boot process, including the role of a bootloader.
6. Describe the motivation, implementation, and application of microkernel abstractions for managing address spaces, threads, and interprocess communication (IPC).
7. Explain the use and implementation of capabilities in access control and resource management.
8. Develop programs using a capability abstraction, like the one provided by the seL4 microkernel.

Course learning objectives, continued

9. Illustrate the use of a range of domain specific languages in the development of systems software.
10. Use practical case studies to evaluate and compare language design proposals.
11. Describe features of modern, high-level programming languages—including abstract datatypes and higher-order functions—and show how they can be leveraged in the construction of low-level software.
12. Explain how the requirements of low-level systems programming motivate the desire for (and benefit from) language-based support.

The "programming languages" perspective

- We will survey and evaluate a range of programming languages during this course:
  - Low-level machine and assembly languages
  - Systems programming languages (e.g., C, Rust, …)
  - Object-oriented languages (e.g., the seL4 API)
  - Domain specific languages
  - Functional languages (e.g., Habit, Haskell, …)
- What are the driving needs of the systems domain?
- How can a programming language design best meet those needs?

Context

- Basic Platform: Generic "IBM PC" compatible
  - 32 bits … not 64
  - IA32 … not x86_64 or ARM
  - BIOS … not EFI or UEFI
  - `int` and `iret` … not `sysenter`/`sysexit`
  - PIC … not APIC
  - No PAE, PCI, ACPI, MMX, SSE, SMM, SMP, VTx, …
  - etc., …
- Already complicated enough for our purposes!
- Well supported by current hardware, emulators, and tools
- Underlying concepts still very broadly applicable

Development environment

- Ubuntu Linux
  - Week 1: using the lab machines (others also an option)
  - Weeks 2+: using a VirtualBox virtual machine, preconfigured with appropriate development tools (can be used on Linux, Mac OS, or Windows)
- Bare metal emulation using the QEMU emulator

Rough schedule

Week	Topic
1	Assembly language programming
2	Bare metal programming
3	Hardware support for OS abstractions
4	Memory management & protection
5	Case Study: L4 use & implementation
6	
7	Case Study 2: seL4 use & implementation
8	
9	Language design for low-level programming
10	

An introduction to IA32 assembly language programming

What is IA32?

- We'll be using the IA32 (x86) architecture as our main target:
  - A "32-bit" instruction set
  - Broadly adopted by:
    - processors from Intel, AMD, Via, ...
    - laptops, desktops, servers, gaming consoles, ...
    - Linux, Mac OS X, Windows, …
  - Arguably, a bit dated … but still very relevant, and a good platform for learning and exploration
  - (… and one of the architectures supported by seL4)

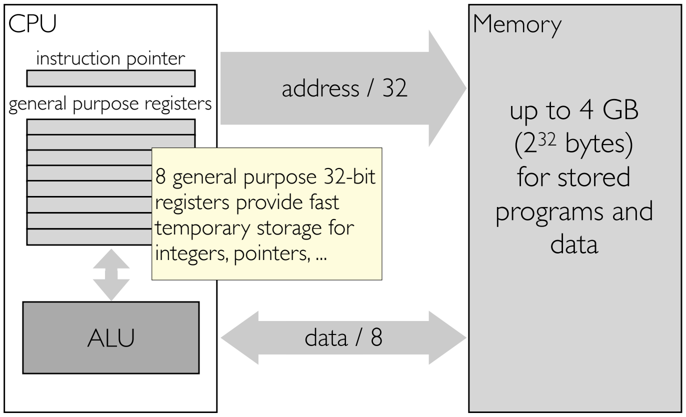
Other architectures:

- Not to be confused with:
  - x86-64/AMD64: a 64 bit architecture supported (in addition to IA32) by more recent AMD/Intel designs
  - IA64: a completely different 64-bit Intel architecture (Itanium)
  - ARM: widely used in phones, tablets, and more
  - IBM Power: used in Xbox 360, PS3, Wii, servers, and more
  - SPARC: used by some of the college's Unix servers
- Except for x86-64, you can't run IA32 code directly on a machine that uses one of these alternative instruction sets!

Notes

- No prior or in-depth knowledge of IA32 programming will be assumed
- We will only use a small subset of the full instruction set
- If you're looking to become an expert on IA32 programming, you'll want to look for another class!
- We'll be using the *AT&T syntax* for IA32 assembly language rather than the *Intel syntax*. This is the default syntax used by the free GNU tools in Linux, MacOS, and DJGPP or Cygwin on Windows, and others

A greatly simplified view of IA32 computing



Programming for IA32

- In concrete terms, an IA32 program is just a collection of byte values (*machine code*)
- Once it has been loaded in to memory, the processor can *execute* a program by interpreting the byte values as *instructions* for the processor to act on
- For practical purposes, we will usually write IA32 programs in a textual format called *assembly language* that is easier to read than raw byte values
- The program that translates assembly language programs in to machine code is called an *assembler*

The GNU assembler, as

- Assembly code goes in files with a .s suffix
- We will typically use gcc to invoke the assembler

  `gcc -m32 –o output assemblyCode.s extras.c`

- You can also invoke the assembler directly: detailed documentation is available from:

  http://sourceware.org/binutils/docs/as/

  For IA32 programming, look in particular at the section on "80386 Dependent Features"

An assembly code listing

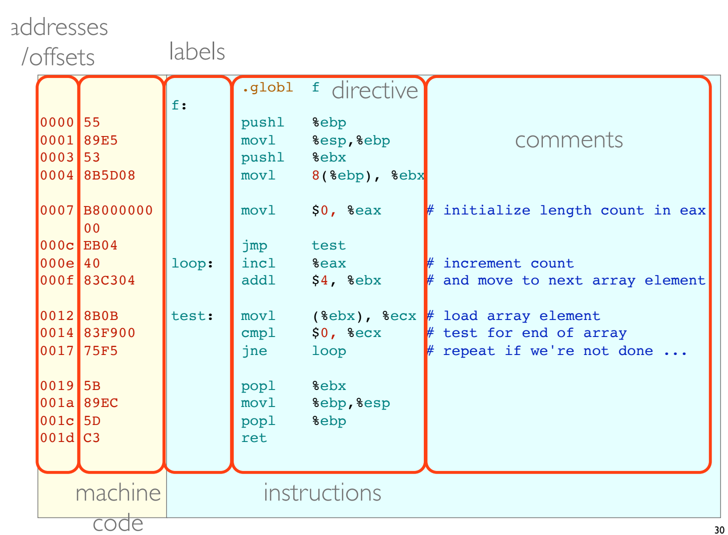
```
        .globl  f
f:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %ebx
        movl    8(%ebp), %ebx

        movl    $0, %eax      # initialize length count in eax

        jmp     test
loop:   incl    %eax          # increment count
        addl    $4, %ebx      # and move to next array element

test:   movl    (%ebx), %ecx  # load array element
        cmpl    $0, %ecx      # test for end of array
        jne     loop          # repeat if we're not done ...

        popl    %ebx
        movl    %ebp,%esp
        popl    %ebp
        ret
```

Assembly code

An assembly code listing

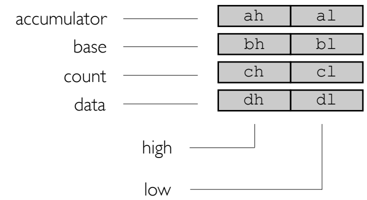
```
                         .globl  f
                 f:
0000 55                  pushl   %ebp
0001 89E5                movl    %esp,%ebp
0003 53                  pushl   %ebx
0004 8B5D08              movl    8(%ebp), %ebx

0007 B8000000            movl    $0, %eax      # initialize length count in eax
     00
000c EB04                jmp     test
000e 40          loop:   incl    %eax          # increment count
000f 83C304              addl    $4, %ebx      # and move to next array element

0012 8B0B        test:   movl    (%ebx), %ecx  # load array element
0014 83F900              cmpl    $0, %ecx      # test for end of array
0017 75F5                jne     loop          # repeat if we're not done ...

0019 5B                  popl    %ebx
001a 89EC                movl    %ebp,%esp
001c 5D                  popl    %ebp
001d C3                  ret
```

Machine code | Assembly code



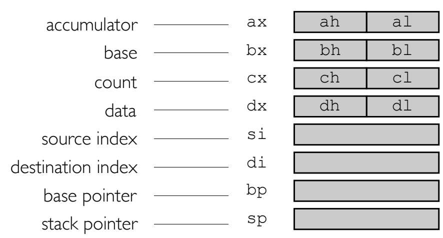
IA32 registers

8-bit registers (holding a single byte, 0-255)



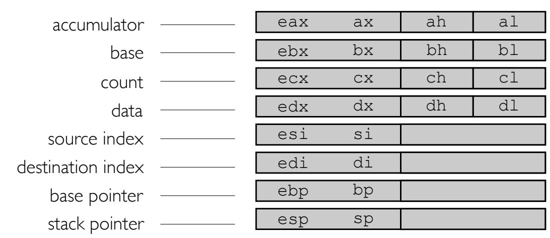
Introduced in 1978 as part of the 8086 architecture

16-bit registers ("word")



Introduced in 1978 as part of the 8086 architecture

32-bit registers ("double word")



"e" for extended

sometimes referred to as "long word"s

Introduced in 1985 as part of the 80386 architecture

Special vs. general purpose registers

- `eip`: the instruction pointer register
- `esp`: the stack pointer register
- `eflags`: the flags register, stores information about the results of the most recent arithmetic or logic instruction

- Other registers can typically be used for any purpose (although some instructions—division, for example—work only with specific registers)

IA32 instructions

Instruction format

- A typical IA32 instruction has the form:

```
opcode src, dst
```

opcode: what to do; src: input source; dst: result destination

- A suffix on the opcode indicates the size of the data that is being operated on:
  - 32-bit values use the suffix **l**(ong)
  - 16-bit values use the suffix **w**(ord)
  - 8-bit values use the suffix **b**(yte)

Addressing modes

- **Register access**, reg:
  - `%eax`: the value in register `eax`
  - Can typically use any registers except `eip` and `eflags`
- **Memory access**, mem:
  - `var`: the value in memory at address var
  - `(%eax)`: the value in memory at the address in `eax`
  - `8(%eax)`: the value in memory at the address given by adding 8 to the value in `eax`
- **Immediate**, immed:
  - `$42`: the constant value 42 (decimal; use `$0x2A` for hex)
  - `$var`: the address of memory location var

Directives for "declaring" variables

```
        .data           # put variables in the "data" section
                        # (code usually goes in .text)

        .align  4       # make sure address is multiple of 4
myvar:  .long   42      # Simple variable, initialized to 42

        .global days    # A globally accessible array of ints
days:   .long   31, 28, 31, 30, 30, 30
        .long   31, 31, 30, 31, 30, 31

scratch:.space  4*100   # reserve uninitialized space

medium: .long   123     # a 32-bit integer (takes 4 bytes)
regular:.short  123     # a 16-bit integer (takes 2 bytes)
small:  .byte   123     # an 8-bit integer (takes 1 byte)
```


How values are stored in memory

- A double word holds 32 binary digits ("bits") (i.e., 4 bytes)

1011	1110	0001	1010	0011	1001	0001	0000
B	E	1	A	3	9	1	0

sign bit (first bit); most significant byte (BE) … least significant byte (10)

- 0xBE1A3910 can be interpreted as -1,105,577,712 (signed) or 3,189,389,584 (unsigned)
- Stored in memory with the least significant byte at the lowest address ("little endian"):

stored byte	0x10	0x39	0x1A	0xBE
address	400	401	402	403

IA32 instructions: data movement

Move instructions

- Copy data from a source to a destination (where `X` is one of the size suffixes: b,w,l):

```
movX src, dst
```

- Any of the following combinations of arguments is allowed:

```
movX reg,   (reg | mem)
movX mem,   reg
movX immed, (reg | mem)
```

- Note that you can't move mem to mem in one instruction

Examples

Suppose that the memory (starting at address 0) contains the following (four byte) values:

0	4	8	12	16	20	24	28	32	36	40	44	48
8	6	2	8	0	2	4	1	7	3	4	5	6

Then

instruction	contents of eax
`movl $12, %eax`	12
`movl (%eax), %eax`	8
`movl 8(%eax), %eax`	0

Zero and sign-extension

- Suppose we want to copy a value from a 16-bit register in to a 32-bit register

```
stu ....... xyz   ==>   ???????????????? | stu ....... xyz
      ax                               eax
```

- Two common strategies:
  - Zero extension: for unsigned values

```
stu ....... xyz   ==>   0000000000000000 | stu ....... xyz
      ax                               eax
```

  - Sign extension: for signed values

```
stu ....... xyz   ==>   ssssssssssssssss | stu ....... xyz
      ax                               eax
```


Move with sign, move with zero extension

- Copy from source to larger destination with sign extension:

```
movsFT src, dst
```

- Copy from source to larger destination with zero extension:

```
movzFT src, dst
```

- `F` and `T` are the "from" and "to" sizes (either `b`, `w`, or `l`)
- Valid combinations: `bw`, `bl`, or `wl`
- Examples:

```
movsbw %al, %dx     # byte to word
movzwl %ax, %edx    # word to long
```


Scaled indexed addressing

- [base] ([reg_1] , reg_2 [, index])

  a memory operand whose address is the value in reg_1, *plus* the specified base constant, *plus* the value of reg_2 *times* the index (which must be 1, 2, 4, or 8)

- Any of the parts in [...] can be omitted
- Examples:

`(eax,ebx,4)`	the **`ebx`**th element in the array of 32-bit words starting at the address in **`eax`**
`days(,ebx,4)`	the **`ebx`**th element in the array of 32-bit words starting at the address **`days`**

More examples

Suppose that the memory (starting at address 0) contains the following (four byte) values:

0	4	8	12	16	20	24	28	32	36	40	44	48
8	6	2	8	0	2	4	1	7	3	4	5	6

Then

instruction	eax	ebx
`movl $12, %eax`	12	
`movl 8(%eax), %ebx`	12	2
`movl 12(%eax,%ebx,4), %eax`	7	2

The `lea` (load effective address) instruction

- Load the address of the source operand (must be memory) to a destination (where `X` is one of the size suffixes: `b`,`w`,`l`):

```
leaX src, dst
```

- Can also be used to co-opt the addressing mode circuitry into performing arithmetic operations:

```
leal 4(%eax),%eax         # eax += 4
leal 1(%eax,%eax,2),%eax  # eax = 3*eax + 1
leal 1(%eax,%eax), %eax   # eax = 2*eax + 1
leal 4(,%eax,8), %eax     # eax = 8*eax + 4
```

- These instructions just do an address calculation and do not attempt to read the data at that address.

The exchange instruction

- Exchange data between two locations

xchgX (reg | mem), reg

- Consider the following instructions in a high-level language:

```
int tmp = x;
x       = y;
y       = tmp;
```

- If x and y are held in registers, then a "clever enough" compiler can translate this code into a single xchgl instruction

The instruction pointer, eip

- The eip register holds the address of the next instruction to be executed
- As the processor reads each instruction, it increments the value in eip by the appropriate number of bytes to point to the following instruction
- This mechanism allows the processor to execute a sequence of instructions stored in contiguous locations in memory
- What would happen if we "move" a different value in to eip?

Jumping and labels

- We can transfer control and start executing instructions at address addr by using a jump instruction

jmp addr

- Labels can be attached to instructions in an assembly language program:

```
►       jmp b
a:      jmp c
b:      jmp a
c:      ...
```

- Modern, pipelined machines work well with sequences of instructions that appear in consecutive locations. Jumps can be expensive: one of the goals of an optimizing compiler is to avoid unnecessary jumps.

IA32 instructions: arithmetic and logic operations

Arithmetic instructions

- Combine a given src with a given dst value and leave the result in dst:

```
► addX   src, dst  }
  subX   src, dst  } integer arithmetic (signed)
  imulX  src, dst  }
  andX   src, dst  }
  orX    src, dst  } bitwise arithmetic
  xorX   src, dst  }
```

- Similar to dst += src, dst -= src, etc.. in C/C++

Examples

- To compute $x^2 + y^2$ and store the result in z:

```
► movl  x, %eax
  imull %eax, %eax
  movl  y, %ebx
  imull %ebx, %ebx
  addl  %ebx, %eax
  movl  %eax, z
```

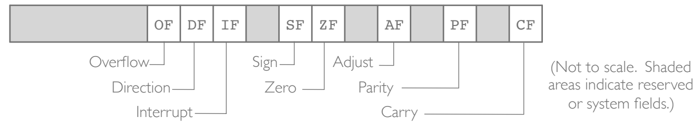
register	contents
eax	x^2+y^2
ebx	y^2

```
    .data
x:  .long  4
y:  .long  3
z:  .long  0
```

IA32 instructions: conditional execution

Flags

- In addition to performing the required operation, arithmetic instructions also change bits in the `eflags` register



- The flags record details about the last operation, such as:
  - Was the result zero?
  - Was the result positive?
  - Did a carry occur?
  - etc...

Conditional jumps, `jCC`

We can test these flags in *conditional jump* instructions

`jz addr`	(jump to `addr` if the zero flag is set)
`jnz addr`	(jump to `addr` if the zero flag is not set)
`je addr`	(jump to `addr` if equal; same as `jz`)
`jne addr`	(jump to `addr` if not equal; same as `jnz`)
`jl addr`	(jump to `addr` if less than) (signed)
`jnl addr`	(jump to `addr` if not less than) (signed)
`jg addr`	(jump to `addr` if greater than) (signed)
`jng addr`	(jump to `addr` if not greater than) (signed)
...	

Examples

```
subl    %eax,%ebx
jz      addr
```
jump to addr if ebx = eax

```
subl    %eax,%ebx
jnz     addr
```
jump to addr if ebx ≠ eax

```
subl    %eax,%ebx
jl      addr
```
jump to addr if ebx < eax

```
subl    %eax,%ebx
jnl     addr
```
jump to addr if ebx >= eax

If the specified condition does not apply, then execution just continues with the next instruction ...

The compare instruction

- The `cmpX` instruction behaves like `subX` except that the result is not saved; only the flags are changed
- For example:

```
cmpl  %eax,%ebx
jl    addr
```

will jump to `addr` if the value in `ebx` is less than the value in `eax`, but it will <u>not</u> change the values in either register

Other conditional instructions

- There are some other instructions that perform an action based on the conditional flags without the cost of a jump
- `setCC` reg8 sets the value in a specified 8-bit register to 0 or 1, based on the condition specified by `CC`:

```
cmpl     %ecx,%ebx    # set eax to 1 if
setl     %al          # ebx < ecx, or
movzbl   %al,%eax     # else to 0
```

- `cmovCC` src, dst copies data from the specified src to dst, but only if the condition specified by `CC` holds:

```
cmpl     %ebx,%eax    # set eax to the max of
cmovl    %ebx,%eax    # eax and ebx
```

condition code; no size suffix here!

IA32 instructions: more arithmetic

Unary operations

- The following arithmetic operations have only one argument (which serves as both source and destination)

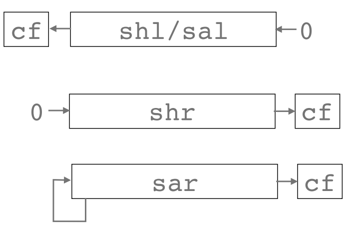
`negX`	(reg | mem)	negate
`notX`	(reg | mem)	complement
`incX`	(reg | mem)	increment
`decX`	(reg | mem)	decrement

- Like the binary operators, these instructions also set the flags for subsequent testing

Bitwise shift operations

- Shift operations are handled using instructions of the form:

`op` count, (reg | mem)



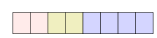
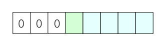
shift (logical/arithmetic) left

shift logical right

shift arithmetic right

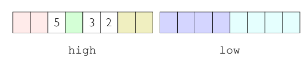
- count is either a constant or else the `%cl` register
- In all cases, the count value will be masked to 5 bits (0-31)

Example

- Given two 32 bit input values:
  - base: 
  - limit: 

Each box is one nibble (4 bits), least significant bits on the right

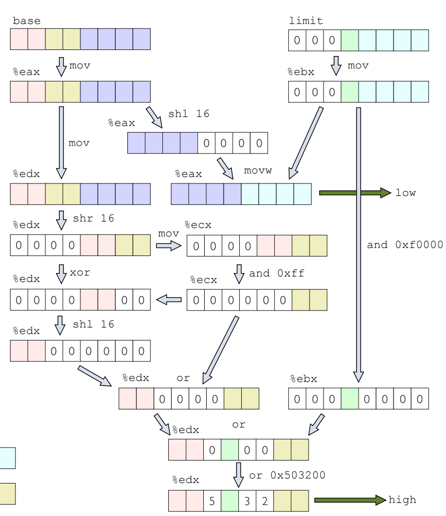
- Calculate a 64 bit descriptor:

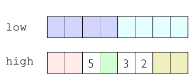


- (Needed for the calculation of "GDT entries")

Example

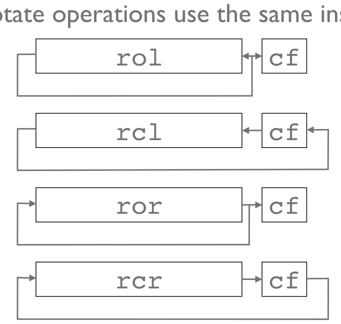
```
movl    base, %eax
movl    limit, %ebx

mov     %eax, %edx
shl     $16, %eax
mov     %bx, %ax
movl    %eax, low

shr     $16, %edx
mov     %edx, %ecx
andl    $0xff, %ecx
xorl    %ecx, %edx
shl     $16,%edx
orl     %ecx, %edx
andl    $0xf0000, %ebx
orl     %ebx, %edx
orl     $0x503200, %edx
movl    %edx, high
```





Bitwise rotate operations

- Rotate operations use the same instruction format:



rotate left

rotate left with carry

rotate right

rotate right with carry

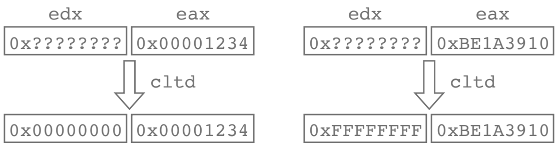
- [Aside: Curiously, "higher level" languages often include shift operators, but not rotates, even though the latter have more interesting/uniform behavior …]

Division

- Divide implicit destination (`edx:eax`) (a 64-bit quantity) by a specified argument with result in `eax` and remainder in `edx`

`idivl` (reg | mem)

- Often used in conjunction with the `cltd` instruction ("convert long to double", a.k.a. `cdq`), which converts a signed 32-bit value in `eax` into the corresponding signed 64-bit value in `edx:eax`.



Example 1

Divide 4,660 (i.e., 0x1234) by 25:

```
movl   $0x1234, %eax
cltd
movl   $25, %ecx
idivl  %ecx
```

Results: `eax` = `0xBA` (186)
`edx` = `0xA` (10)

Sure enough: 186*25 + 10 = 4,660

Example 2

Divide -1,105,577,712 (i.e., 0xBE1A3910) by 256

```
movl   $0xBE1A3910, %eax
cltd
movl   $256, %ecx
idivl  %ecx
```

Results: `eax` = `0xFFBE1A3A` (-4,318,662)
`edx` = `0xffffff10` (-240)

Sure enough: -4,318,662 * 256 - 240 = -1,105,577,712

Complications of division

- Division produces multiple results: a quotient and a remainder
- Division uses special registers: we'd better not store any other values in `eax` or `edx` if there's a chance that a division instruction might be executed
- Doesn't set flags: requires separate tests, for example, to determine whether quotient or remainder was zero
- Division can raise an exception if the src is zero (or -1)

IA32 instructions: using the stack

Stack

- The IA32 includes features that allow the programmer to use a region of memory as a simple stack:
  - the `esp` (stack pointer) register
  - special instructions like `push`, `pop`, `call`, `ret`, ...
- There is no obligation for the programmer to use these features, but it is often convenient to do so:
  - for temporary/scratch storage when a calculation needs more storage than the CPU registers can provide
  - to support calling and returning from functions

A typical memory layout

- A typical operating system reserves an area of scratch memory for each program, and sets the `esp` register to point to the end of this region when the program begins

- The stack pointer moves
  - down (decreases) as values are pushed on to the stack
  - up (increases) as values are popped off of the stack
- So long as they never overlap, the data and stack areas can grow or shrink as necessary as the program runs

Stack operations

- Push a value onto the stack

  `pushl` (reg | mem | immed)

- Pop a value of the stack

  `popl` (reg | mem)

- Roughly speaking:

```
pushl src   =   subl $4, %esp;   movl src, (%esp)
popl dst    =   movl (%esp), dst;   addl $4, %esp
```


Spilling temporaries on the stack

- The stack is often used for saving the contents of a register on the stack ("spilling") so that the register can be used, temporarily, for some other reason
- For example:

```
pushl  %eax
pushl  %edx
... code that changes eax and/or edx ...
popl   %edx
popl   %eax
```

(pop values in reverse order that was used to push them!)

- Note that values on the stack can still be accessed, from memory, using `(%esp)`, `4(%esp)`, `8(%esp)`, `12(%esp)`, ...

Call and return

- There is a special instruction for calling a function

```
call addr     ≃       pushl  $lab
                      jmp    addr
                  lab: ...
```

- And a special instruction for returning from a function

```
ret           ≃       popl   %eax
                      jmp    *%eax
```

(assuming **eax** isn't being used for something else ...)

(special syntax: jump to the address given by the contents of **eax**)

- In practice, additional instructions are often needed to deal with parameter passing, etc. ...

Functions and the System V ABI

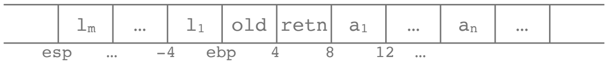
Implementing functions

- How do we pass arguments to a function?
- How does a function return a result?
- How do we handle local variables?
- In principle, especially in a bare metal setting, we can implement these features any way we like, using the basic tools that the IA32 instruction set provides
- But there are some existing standards we can follow, notably the "System V IA32 Application Binary Interface (ABI)":

  http://www.sco.com/developers/devspecs/abi386-4.pdf
  particularly Section 3-9

Stack frames

The code for any given function/procedure call runs in the context of a <u>stack frame</u> of the form:



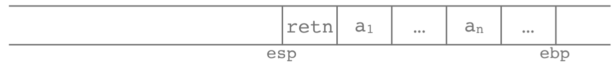
- <u>Frame (base) pointer</u>: `ebp` points to the stack frame; the caller's frame pointer is stored in `old` (i.e., `(%ebp)`)
- <u>Return address</u>: `retn` is the return address
- <u>Actual parameters</u>: a_1, … ,a_n are the function's arguments. We can access a_1 as `8(%ebp)`, etc…
- <u>Local variables</u>: l_1, … ,l_m are the function's local variables. We can access l_1 as `-4(%ebp)`, etc…

Building the stack frame … in the caller

- 

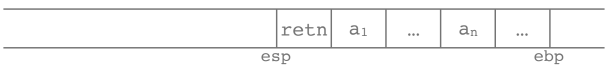
- The <u>caller</u> starts by pushing the arguments:

- Then it executes a `call` instruction, which pushes the return address:

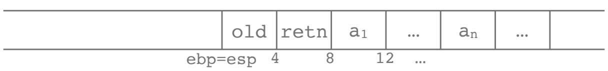


- … and jumps to the code for the callee …

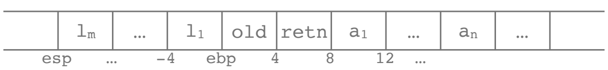
Building the stack frame … in the callee

- 

- The <u>callee</u> saves the old frame pointer, and sets a new value:



- Then it decrements the stack pointer to reserve space for any local variables:



- … and now the callee can start work …

Function prologue

- The code that builds the stack frame at the start of a function body is called the <u>prologue</u>:
  - At the beginning of a function body, the parameters and return address have already been pushed on to the stack. We need to:

```
pushl %ebp          # save old frame pointer
movl  %esp, %ebp    # and set new value
```

  - If local variables taking `M` bytes of storage are required, then we need to reserve space for them:

```
subl   $M, %esp   # allocate space for
                  # locals (skip if M=0)
```


Function epilogue

- When a function completes, we must dismantle the stack frame and return the machine to the state it was in before the call. The code to do this is called the <u>epilogue</u>:
  - Running the previous process in reverse:

```
movl   %ebp, %esp  # discard locals/temps
popl   %ebp        # restore frame pointer
ret                # return to caller
```

  - The first two instructions here can be replaced with the more efficient, but otherwise equivalent `leave` instruction

Removing the parameters

- Once we return to the caller, the result of the function is in `eax`, but the parameters are still on the stack:

- We restore the stack pointer to its original value by adding on the number of bytes that are used by the parameters:

```
addl  $N, %esp
```

- If no parameters were passed, then this step can be omitted

Example: a leaf function

```
int g(int u) {
  return u*u;
}
```

```
g: pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %eax
   imull %eax, %eax
   movl  %ebp, %esp
   popl  %ebp
   ret
```

…	old	retn	u	…
	esp=ebp	ebp+4	ebp+8	

Example: multiple parameters + call

```
int f(int x,
      int y,
      int z) {
  return g(x+y);
}
```

```
f: pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %eax
   addl  12(%ebp), %eax
   pushl %eax
   call  g
   addl  $4, %esp
   movl  %ebp, %esp
   popl  %ebp
   ret
```

…	old	retn	x	y	z	…
	esp=ebp	ebp+4	ebp+8	ebp+12	ebp+16	

Example: spilling

```
int h(int x,
      int y,
      int z) {
    return g(x)+g(y);
}
```

```
h: pushl %ebp
   movl  %esp, %ebp
   pushl 8(%ebp)
   call  g
   addl  $4,%esp
   pushl %eax -- spill
   pushl 12(%ebp)
   call  g
   addl  $4, %esp
   popl  %ecx -- unspill
   addl  %ecx, %eax
   movl  %ebp, %esp
   popl  %ebp
   ret
```

spill	old	retn	x	y	z
esp	ebp	ebp+4	ebp+8	ebp+12	ebp+16

Observations

- There is a four instruction overhead for each function that uses the frame pointer
  - Increases execution time
  - Prevents use of ebp as a general purpose register
- For larger functions, the four instruction overhead is less of an issue
- For small functions, we would prefer to inline rather than copy
- Nevertheless, it is common to produce code that doesn't use ebp as a frame pointer (e.g., `-fomit-frame-pointer` in gcc)

Caller and callee saves

We (System V) can designate some registers as:

- **caller saves** (eax, ecx, and edx)
  - can be freely used by the callee
  - the caller is responsible for saving (and later restoring) the value of a caller save register before a call
- **callee saves** (ebp, ebx, esi, and edi)
  - can be freely used by the caller
  - the callee is responsible for saving (and later restoring) the value of a callee saves register before using it to store temporary values

Revisiting the previous example: h

```
int h(int x,
      int y,
      int z) {
    return g(x)+g(y);
}
```

```
h: pushl %ebp
   movl  %esp, %ebp

   pushl 8(%ebp)  -- x
   call  g
   addl  $4,%esp
   pushl %eax -- spill
   pushl 12(%ebp) -- y
   call  g
   addl  $4, %esp
   popl  %ecx -- unspill
   addl  %ecx, %eax

   movl  %ebp, %esp
   popl  %ebp
   ret
```

instead of having the compiler save this value on the stack …

Revisiting the previous example: h

```
int h(int x,          h: pushl %ebp
      int y,             movl  %esp, %ebp
      int z) {
   return g(x)+g(y);     pushl 8(%ebp)  -- x
}                        call  g
                         addl  $4,%esp
                         movl  %eax, %esi
                         pushl 12(%ebp) -- y
                         call  g
                         addl  $4, %esp

                         addl  %esi, %eax

                         movl  %ebp, %esp
                         popl  %ebp
                         ret
```

… we can move it to a callee saves register, esi

g will preserve the value in esi, if necessary

so it will still contain the correct value here…

Revisiting the previous example: h

```
int h(int x,          h: pushl %ebp
      int y,             movl  %esp, %ebp
      int z) {           pushl %esi
   return g(x)+g(y);     pushl 8(%ebp)  -- x
}                        call  g
                         addl  $4,%esp
                         movl  %eax, %esi
                         pushl 12(%ebp) -- y
                         call  g
                         addl  $4, %esp

                         addl  %esi, %eax
                         popl  %esi
                         movl  %ebp, %esp
                         popl  %ebp
                         ret
```

… now h has to save the value in register, esi

one save in h is better than one saves in each of two calls to g

empirically, more than 50% of calls are to leaf functions

Closing thoughts

Assembly "Language"?

- Highly imperative, primitive instructions, no expressions
- No high-level abstractions, but all the building blocks:
  - No arrays, records, variants, objects, closures, …
  - No loops, switch statements, functions, local variables, …
- Type System?
  - Values classified by size (e.g., 8 vs 32 bits) and storage class (e.g., memory, flag, integer register, floating point register, …)
  - Limited protection against common programming mistakes
  - Programmer has full control over data representation

Summary

- IA32 provides a very basic programming language:
  - A fixed set of registers
  - Instructions for moving and operating on data
  - Instructions for testing and control transfer
- In programming language terms:
  - Low-level, primitive instructions, loosely typed
  - No high-level abstractions, but all the building blocks
  - Very close to the metal, low-level control, "predictable" performance
- Let's write some programs!