

Intel IA-32 Assembly

Simon Kågström

Department of Systems and Software Engineering
Blekinge Institute of Technology
Ronneby, Sweden

http://www.ipd.bth.se/ska/unix_programming.html



1 Motivation

- Introduction to the IA-32
- IA-32 History
- IA-32 Architecture

2 Assembly Language Introduction

- Compiling and Writing Assembly Language
- Handling of Data
- Instructions, Labels and Registers

3 Writing IA-32 Programs

- Interfacing with the Operating System
- Arithmetic/Logic Instructions
- Program Flow (branches)
- Referencing Memory

4 Assembly Laboration Assignment

1 Motivation

- Introduction to the IA-32
- IA-32 History
- IA-32 Architecture

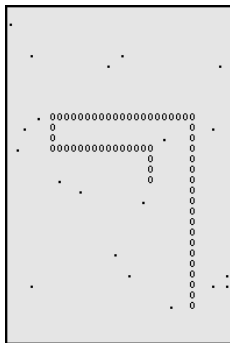
2 Assembly Language Introduction

- Compiling and Writing Assembly Language
- Handling of Data
- Instructions, Labels and Registers

3 Writing IA-32 Programs

- Interfacing with the Operating System
- Arithmetic/Logic Instructions
- Program Flow (branches)
- Referencing Memory

4 Assembly Laboration Assignment

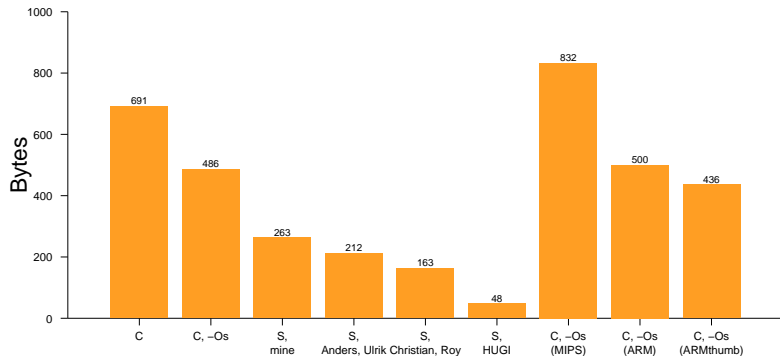


- Objective with these lectures: Introduce AT&T-style intel assembly
 - AT&T is a syntax-style used in the GNU tools.
 - The other main alternative is Intel syntax, used by NASM, MASM, TASM etc.
- Get some experience in writing assembly programs
 - ... And implement a Nibbles game
- For information, see idenet and www.ipd.bth.se/ska/unix_programming.html

Why should we use assembly?

- Speed?
 - Rarely. It is very hard to optimise better than the compiler.
 - Sometimes: Writing vectorised code for DSPs
 - Skilled assembly programmers can also (sometimes) beat the compiler, but you must know the machine internals in order to manage that
- Operating systems?
 - Not directly, almost no OS code is assembly.

Why use assembly: Size?



- You can reduce the size by assembly implementations.
- But as you can see, the C-version is fairly small as well
- This is very seldom a real problem

Why use assembly!

- Know your machine?
 - For writing an OS? Know your machine
 - Exploiting security holes? Know your machine
 - Optimizing performance? Know your machine
 - etc, etc.
- To know your machine: learn assembly.
- To learn assembly? know your machine.

Why use assembly!

- Know your machine?
 - For writing an OS? Know your machine
 - Exploiting security holes? Know your machine
 - Optimizing performance? Know your machine
 - etc, etc.
- To know your machine: learn assembly.
- To learn assembly? know your machine.
- Also very, very cool... So **learn assembly!**

Description

- **Machine instructions** - binary pattern which is parsed and executed by the CPU
- **Mnemonics** - symbolic representation of machine instructions
 - This is what we mean with programming in assembly
- Programming is done directly with machine instructions (but generally written in mnemonics)
- Data is stored in memory
- Operations are done on **registers** (not IA-32)

Example

if (a < 5)	cmpl \$5, a	0: 83 3d 00 00 00 00 05
b = 7;	jg nope	7: 7f 0a
a++;	movl \$7, b	9: c7 05 00 00 00 00 07
nope:	<nope>:	
	incl a	13: ff 05 00 00 00 00

Description

- **Machine instructions** - binary pattern which is parsed and executed by the CPU
- **Mnemonics** - symbolic representation of machine instructions
 - This is what we mean with programming in assembly
- Programming is done directly with machine instructions (but generally written in mnemonics)
- Data is stored in memory
- Operations are done on **registers** (not IA-32)

Example

```
if (a < 5)           cmpl $5, a           0: 83 3d 00 00 00 00 05
    b = 7;           jg nope           7: 7f 0a
a++;                movl $7, b          9: c7 05 00 00 00 00 07
    nope:           <nope>:
                    incl a           13: ff 05 00 00 00 00
```

Description

- **Machine instructions** - binary pattern which is parsed and executed by the CPU
- **Mnemonics** - symbolic representation of machine instructions
 - This is what we mean with programming in assembly
- Programming is done directly with machine instructions (but generally written in mnemonics)
- Data is stored in memory
- Operations are done on **registers** (not IA-32)

Example

if (a < 5)	<code>cmpl \$5, a</code>	0: 83 3d 00 00 00 00 05
b = 7;	<code>jg nope</code>	7: 7f 0a
a++;	<code>movl \$7, b</code>	9: c7 05 00 00 00 00 07
	<code>nope:</code>	<code><nope>:</code>
	<code>incl a</code>	13: ff 05 00 00 00 00

Description

- **Machine instructions** - binary pattern which is parsed and executed by the CPU
- **Mnemonics** - symbolic representation of machine instructions
 - This is what we mean with programming in assembly
- Programming is done directly with machine instructions (but generally written in mnemonics)
- Data is stored in memory
- Operations are done on **registers** (not IA-32)

Example

if (a < 5)	cmpl \$5, a	0: 83 3d 00 00 00 00 05
b = 7;	jg nope	7: 7f 0a
a++;	movl \$7, b	9: c7 05 00 00 00 00 07
nope:	<nope>:	
	incl a	13: ff 05 00 00 00 00 00

- IA-32 (Intel Architecture / 32 bits) is a CISC-architecture
- The most popular 32-bit architecture so far (and probably in the future as well). Why?
 - IBM PC, MS-DOS and MS Windows
 - Backwards compatible:
 - *“One of the most important achievements of the IA-32 architecture is that the object code programs created for these processors starting in 1978 still execute on the latest processors in the IA-32 architecture family.”*
 - *(Intel Software Developers Manual)*
- RISC-systems better for compilers (and probably programmers)
- IA-32 now internally works RISC-like (microcode)
- AMD has implemented 64-bit extensions to IA-32, and the basic architecture is likely to stay around

- The Intel-architecture starts with the 8086 from the 1970s
- The 8086 is a 16-bit architecture (data bus)
- 16-bit registers (`ax`, `bx`, `cx`, ...)
- Memory addresses are 16-bit (i.e. possible to address 64KB)
- “640 KB should be enough for anyone!” Why not 64KB?

- The Intel-architecture starts with the 8086 from the 1970s
- The 8086 is a 16-bit architecture (data bus)
- 16-bit registers (ax, bx, cx, ...)
- Memory addresses are 16-bit (i.e. possible to address 64KB)
- “640 KB should be enough for anyone!” Why not 64KB?
 - Segmentation: Register containing a base, addresses offsets to that base
 - Segment registers 16-bit wide
 - Different segments for code, data and stack (cs, ds, ss).
 - Real address (“effective address”): $\text{segment} * 16 + \text{address}$
 - $64\text{KB} * 16 = 1\text{MB}$ (640KB is an MS-DOS limit)
- We will use 80386 and up (IA-32)

History, II

- From 80386 and onwards, Intel CPUs use a 32-bit architecture
- 386: 4GB address space, protection etc.
- 486: Caching, better FPU, instruction pipelining
- Pentium: More pipelining, branch prediction, faster data bus
- PPro: Redesign. Conditional moves, register renaming, 36-bit physical addresses, etc.
- P-II: Larger caches, MMX (integer vector execution)
- P-III: SIMD (floating point vector ops)
- P4: Redesign. Tracebuffer, hyperthreading
- x86-64/EM64T: 64-bit extension of IA-32, 8 extra registers, extended old registers, misc changes
- Itanium: Complete redesign, but emulates the old IA-32

- Four privilege levels: 0 (supervisor) ... 3 (user).
- Only 0 and 3 used in practice.
- In privilege level 0, all instructions are available (kernel mode). For instance: setting the page directory, accessing control registers etc.
- Standard paging is available, 4KB and 4MB pages.
- Device communication (`in/out`) instructions can be (but is usually not) available in user-mode.
- Software interrupts (the `int` instruction) is generally not available in user-mode, but can be used for syscalls.

Architecture, segment registers

- There are a number of segment registers, `cs`, `ds`, `ss` etc.
- The segment registers specify an index into a table.
- Each table (the GDT or LDT) entry contains:
 - A base address (32 bits)
 - A segment limit (20 bits, 1B/4KB granularity)
 - Privilege level (0..3)
 - Misc information.
- It is not possible to change to a segment with a higher privilege level than the current one.

1 Motivation

- Introduction to the IA-32
- IA-32 History
- IA-32 Architecture

2 Assembly Language Introduction

- Compiling and Writing Assembly Language
- Handling of Data
- Instructions, Labels and Registers

3 Writing IA-32 Programs

- Interfacing with the Operating System
- Arithmetic/Logic Instructions
- Program Flow (branches)
- Referencing Memory

4 Assembly Laboration Assignment

Compiling and ELF files

- Assembly source are put in .S-files
- The .S-files are compiled by the assembler into object files (.o)
- The object files are linked together by the linker into an executable file
- Note that this is exactly the same process as for C/C++ source!
- UNIX systems use the ELF object/exec format
 - Data section (.data): Initialised data (global variables)
 - BSS section (.bss): Uninitialised data (globals, zero)
 - Text section (.text): Code
 - Misc other sections

Assembly coding guidelines

- Program code is found in the `.text`-segment
- Code for GCC or GNU AS looks as below
- This is AT&T-style assembly which you can recognize by
 - %-prefixed registers,
 - \$-prefixed constants,
 - source, dest operations
- Regarding comments: Stick to one style...

Example

```
.text          # or .section .text
movl $5, %eax  # This is a comment
subl %eax, %ebx /* This is also a comment, C-style */
mull %ebx      // C++ are also possible.
```

Assembly coding guidelines

- Program code is found in the `.text`-segment
- Code for GCC or GNU AS looks as below
- This is AT&T-style assembly which you can recognize by
 - %-prefixed registers,
 - \$-prefixed constants,
 - source, dest operations
- Regarding comments: Stick to one style...

Example

```
.text          # or .section .text
movl $5, %eax  # This is a comment
subl %eax, %ebx /* This is also a comment, C-style */
mull %ebx      // C++ are also possible.
```

Assembly coding guidelines

- Program code is found in the `.text`-segment
- Code for GCC or GNU AS looks as below
- This is AT&T-style assembly which you can recognize by
 - %-prefixed registers,
 - \$-prefixed constants,
 - source, dest operations
- **Regarding comments: Stick to one style...**

Example

```
.text          # or .section .text
movl $5, %eax  # This is a comment
subl %eax, %ebx /* This is also a comment, C-style */
mull %ebx      // C++ are also possible.
```

Description

- Data can be in either the `.bss` (uninitialised) or the `.data` (initialised) section
 - Data in `.bss` will always be zero at program startup
- Dynamic memory must be allocated via the system (`malloc/sbrk`)
 - Seldomly used in assembly

Example

```
.bss
NR_BSS:      .long 0          # Will be 0 at program startup
MORE:       .space 128      # 128 bytes of zeroes
.data
NR_DATA:    .long 0x123     # Hex value
            .asciz "hej"    # string (i.e. bytes 0x68656600)
.section .text
            movl NR_DATA, %eax # NR_DATA into %eax
            movl %eax, NR_BSS # Copy from NR_DATA to NR_BSS
```

Description

- Data can be in either the `.bss` (uninitialised) or the `.data` (initialised) section
 - Data in `.bss` will always be zero at program startup
- Dynamic memory must be allocated via the system (`malloc/sbrk`)
 - Seldomly used in assembly

Example

```
.bss
NR_BSS:        .long 0           # Will be 0 at program startup
MORE:         .space 128        # 128 bytes of zeroes
.data
NR_DATA:      .long 0x123       # Hex value
               .asciz "hej"    # string (i.e. bytes 0x68656600)
.section .text
movl NR_DATA, %eax # NR_DATA into %eax
movl %eax, NR_BSS # Copy from NR_DATA to NR_BSS
```

Description

- Data can be in either the `.bss` (uninitialised) or the `.data` (initialised) section
 - Data in `.bss` will always be zero at program startup
- Dynamic memory must be allocated via the system (`malloc/sbrk`)
 - Seldomly used in assembly

Example

```
.bss
NR_BSS:        .long 0           # Will be 0 at program startup
MORE:         .space 128        # 128 bytes of zeroes
.data
NR_DATA:      .long 0x123       # Hex value
              .asciz "hej"     # string (i.e. bytes 0x68656600)
.section .text
              movl NR_DATA, %eax # NR_DATA into %eax
              movl %eax, NR_BSS # Copy from NR_DATA to NR_BSS
```

Description

- Depends on the compiler
- With GCC the C preprocessor runs, so you can use `#define`.
- With `as`, constants are set using the `.set` keyword
 - Trick: Run `cpp` before `as` if you want to use `#define`
- NASM has its own preprocessor (Note the intel syntax!)

Examples

<code>#define</code>	<code>CONSTANT 16</code>	
<code>...</code>		GCC
	<code>movl \$CONSTANT, %eax</code>	
<hr/>		
<code>.set</code>	<code>CONSTANT, 16</code>	
<code>...</code>		GNU <code>as</code>
	<code>movl \$CONSTANT, %eax</code>	
<hr/>		
<code>%define</code>	<code>CONSTANT 16</code>	
<code>...</code>		NASM
	<code>mov eax, CONSTANT</code>	

Description

- Depends on the compiler
- With **GCC** the C preprocessor runs, so you can use `#define`.
- With `as`, constants are set using the `.set` keyword
 - Trick: Run `cpp` before `as` if you want to use `#define`
- **NASM** has its own preprocessor (Note the intel syntax!)

Examples

```
#define CONSTANT 16
```

```
...
```

GCC

```
movl $CONSTANT, %eax
```

```
.set CONSTANT, 16
```

```
...
```

GNU as

```
movl $CONSTANT, %eax
```

```
%define CONSTANT 16
```

```
...
```

NASM

```
mov eax, CONSTANT
```

Description

- Depends on the compiler
- With GCC the C preprocessor runs, so you can use `#define`.
- With `as`, constants are set using the `.set` keyword
 - Trick: Run `cpp` before `as` if you want to use `#define`
- NASM has its own preprocessor (Note the intel syntax!)

Examples

```
#define    CONSTANT 16
...
movl $CONSTANT, %eax
GCC
```

```
.set     CONSTANT, 16
...
movl $CONSTANT, %eax
GNU as
```

```
%define  CONSTANT 16
...
mov  eax, CONSTANT
NASM
```

Description

- Depends on the compiler
- With GCC the C preprocessor runs, so you can use `#define`.
- With `as`, constants are set using the `.set` keyword
 - Trick: Run `cpp` before `as` if you want to use `#define`
- **NASM** has its own preprocessor (Note the intel syntax!)

Examples

```
#define CONSTANT 16
```

```
...
```

```
movl $CONSTANT, %eax
```

GCC

```
.set CONSTANT, 16
```

```
...
```

```
movl $CONSTANT, %eax
```

GNU as

```
%define CONSTANT 16
```

```
...
```

```
mov eax, CONSTANT
```

NASM

Instruction format

Description

- Instruction format: `instr [op1, op2]`
- Instructions have 0, 1 or 2 operands
 - In AT&T syntax, `insn src, dst`, in Intel syntax `insn dst, src`
- The operands can be an immediate operand (a constant), a register or a memory reference.

Examples

```
.data
label:  .long 0
.text
movl $5, %eax      # register %eax = 5
neg %eax           # %eax = -%eax
movl $5, label     # put 5 in the memory at label
movl $label, %eax  # place the address of label in %eax
pushl (%eax)       # Push the value on label on the stack
cld                # clear direction flag
```

Description

- Labels point out memory addresses
- Compile-time replaced with absolute or relative addresses
- Local labels are numbers (labels without symbols)
 - Direction specified on reference
- Global labels can be seen from other files

Examples

```
.globl  fn          # Declare fn as global
fn:     # The fn label
lab:    # This is also a label, visible in this file
        jmp lab     # This is a jump to lab (infinite loop)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label
        jmp 1b      # Jump to 1 (backwards)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label again
```

Description

- Labels point out memory addresses
- Compile-time replaced with absolute or relative addresses
- **Local labels** are numbers (labels without symbols)
 - Direction specified on reference
- Global labels can be seen from other files

Examples

```
.globl  fn          # Declare fn as global
fn:     # The fn label
lab:    # This is also a label, visible in this file
        jmp lab     # This is a jump to lab (infinite loop)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label
        jmp 1b      # Jump to 1 (backwards)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label again
```

Description

- Labels point out memory addresses
- Compile-time replaced with absolute or relative addresses
- Local labels are numbers (labels without symbols)
 - Direction specified on reference
- Global labels can be seen from other files

Examples

```
.globl  fn          # Declare fn as global
fn:     # The fn label
lab:    # This is also a label, visible in this file
        jmp lab     # This is a jump to lab (infinite loop)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label
        jmp 1b      # Jump to 1 (backwards)
        jmp 1f      # Jump to 1 (forward)
1:      # Local label again
```

Description

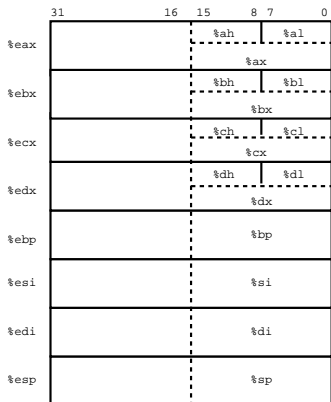
- Labels point out memory addresses
- Compile-time replaced with absolute or relative addresses
- Local labels are numbers (labels without symbols)
 - Direction specified on reference
- Global labels can be seen from other files

Examples

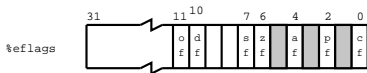
```
.globl  fn          # Declare fn as global
fn:     # The fn label
lab:    # This is also a label, visible in this file
        jmp lab    # This is a jump to lab (infinite loop)
        jmp 1f     # Jump to 1 (forward)
1:      # Local label
        jmp 1b     # Jump to 1 (backwards)
        jmp 1f     # Jump to 1 (forward)
1:      # Local label again
```

- IA-32 has 8 general-purpose registers
- The registers have special uses in some instructions
 - `%eax`: Accumulator, destination of `mul` etc
 - `%ebx`: (Sometimes pointer to data)
 - `%ecx`: Counter for loops
 - `%edx`: Additional destination register
 - `%esi`: Source for string ops
 - `%edi`: Destination for string ops
 - `%esp`: Stack pointer
 - `%ebp`: Frame pointer
- There are also some special registers:
 - `%eip`: Program counter (not directly accessible)
 - `%eflags`: Status flags, accessible through special instructions
 - Plus segmentation `cs`, `ds`, etc. Generally not usable outside the OS kernel

Registers, II



- Some registers have directly accessible 8 and 16-bit parts



1 Motivation

- Introduction to the IA-32
- IA-32 History
- IA-32 Architecture

2 Assembly Language Introduction

- Compiling and Writing Assembly Language
- Handling of Data
- Instructions, Labels and Registers

3 Writing IA-32 Programs

- Interfacing with the Operating System
- Arithmetic/Logic Instructions
- Program Flow (branches)
- Referencing Memory

4 Assembly Laboration Assignment

Examples, factorial calculation

Description

- Compute the factorial:

$$f(n) = \begin{cases} 1, & n = 0 \\ n * f(n - 1), & \textit{otherwise} \end{cases}$$

Example

```
fac:
    movl 4(%esp), %ecx    # Get the argument from the stack
    movl $1, %eax        # Init eax
l1:   mull %ecx           # eax = eax*ecx
    loop l1              # ecx--; if ecx != 0 then goto l1
    ret                  # eax contains the result
    ...
    pushl $5              # pass 5 as argument
    call fac
```

Examples, adding values

Description

- The example adds items together (end with 0) and loops around **lab**

Example

```
.section .data
NBRS:    .long 1           # NBRS point out address of 1
         .long 2
         .long 3
         .long 0

.section .text
adder:   xorl %eax, %eax   # zero %eax
         movl $NBRS, %ebx # The address of NBRS
lab:     movl (%ebx), %ecx # Get the number at (%ebx)
         cmpl $0, %ecx    # if %ecx == 0
         je out          # return
         addl %ecx, %eax  # (else) %eax = %eax + %ecx
         addl $4, %ebx    # Point %ebx to the next number
         jmp lab         # Loop
out:     ret
```

A UNIX program

Description

- A very simple UNIX program is the following (`simple.S`)
- Compile with `as -gstabs simple.S -o simple.o`
 - `-gstabs`: Supply debugging information for GDB.
 - Other arguments: See “`as`” man page
- Link with `gcc -nostdlib -lc -o simple simple.o`
 - `ld` can also be used
- Programs start at `_start` (entrypoint)

Example

```
.section .text
.globl _start      # Make start symbol visible
_start:
    pushl $2       # argument 2 to exit
    call exit      # Call exit
```

The UNIX program again

Description

- The **exit** system call which we used in the last program was called *indirectly* via **libc**
- To call it directly, we must resort to system dependent methods
- System calls on Linux/IA-32 are invoked with a software interrupt
- Other systems (even Linux itself!) use other methods, e.g., `sysenter`
 - This is heavily system dependent!

Example

```
.section    .text
.globl     _start          # Make start symbol visible
_start:

    movl   $1, %eax      # Exit
    movl   $2, %ebx      # argument 2 to exit
    int   $0x80          # Soft interrupt 0x80 (linux syscall)
```

Description

- Generally: **opsize src, dst**
 - Except mul,div
- Keep this slide under your pillow

Example

```
addl    %eax,    %ebx    # %ebx = %ebx + %eax
subl    $0xff,   %ebx    # %ebx = %ebx - 255
andl    $3,      %eax    # %eax = %eax & 3, i.e. zero bits 3..31
xorl    %eax,    %eax    # %eax = %eax ^ %eax (i.e. zero %eax)
orl     $4,      %ecx    # %ecx = %ecx | 4, i.e. set bit 3
shl     $2,      %eax    # %eax = %eax << 2, i.e. %eax = %eax * 4
subb    $8,      %ah     # %ah = %ah - 8 (sub 8 from %ah, byte-size)
addl    (%eax),  %ecx    # %ecx = %ecx + (%eax)
subw    $8,      8      # (8) = (8) - 8 (i.e. the address 8!, word-size)
mull    %ecx     # %edx:%eax = %eax * %ecx, %edx = bits 32..63
divl    %ebx     # %eax = %edx:%eax / %ebx, %edx = %edx:%eax MOD %ebx
mull    (%ebx)   # %edx:%eax = (%ebx) * %eax (memory references)
```

Unconditional jumps

Description

- Unconditional jumps are done with the `jmp` instruction
- The target can be either of:
 - An absolute address (a 32 bit value)
 - A relative address (± 128 bytes from the current `%eip`)
 - A relative address ($\pm 4G$ bytes from the current `%eip`)
 - A register (`jmp *%eax`)
- The compiler determines if a relative jump can be done and chooses the smallest encoding

Example

```
    jmp label
    ...
    movl $label, %eax    # The address of label in %eax
    jmp *%eax           # Jump to the contents of %eax
label:
```

Conditional jumps

Description

- Conditional jumps are done with the `jcc label` instructions where `cc` is a condition
 - The conditions come from flags in the `%EFLAGS` register
 - The `test` and `cmp` instructions can be used to set `EFLAGS` before a conditional jump
- There are many of these. Examples:
 - `jz`: Jump if zero
 - `jnz`: Jump if not zero
 - `je`: Jump if equal, same as `jz` (jump if zero-flag set)
 - `jne`: Jump if not equal
 - `jl`: Jump if less than
 - `jg`: Jump if greater than
 - `jle`: Jump if less than or equal
 - `jge`: Jump if greater than or equal
- The target of a conditional jump is *always* a label

Conditional jumps, if/else

Description

- If-else statements can be constructed with conditional jumps:
- `cmpl` performs a “mental” subtraction between the arguments and sets the flags (so a jump can be done)
- In the example, `1-%eax` is done, and the zero-flag is set if `%eax` is 1
- `test` performs a “mental” bitwise and instead
- How can we avoid the `jmp out` instruction in the false-branch?

Example

```
    cmpl $1, %eax    # if (%eax == 1)
    je then          #   %ebx = 3
    movl $2, %ebx    # else
    jmp out          #   %ebx = 2
then:  movl $3, %ebx
out:
```

Conditional jumps, if/else

Description

- If-else statements can be constructed with conditional jumps:
- `cmpl` performs a “mental” subtraction between the arguments and sets the flags (so a jump can be done)
- In the example, `1-%eax` is done, and the zero-flag is set if `%eax` is 1
- `test` performs a “mental” bitwise and instead
- How can we avoid the `jmp out` instruction in the false-branch?

Example

```
    cmpl $1, %eax    # if (%eax == 1)
    je then         #   %ebx = 3
    movl $2, %ebx   # else
    jmp out         #   %ebx = 2
then:  movl $3, %ebx
out:
```

Conditional jumps, if/else

Description

- If-else statements can be constructed with conditional jumps:
- `cmpl` performs a “mental” subtraction between the arguments and sets the flags (so a jump can be done)
- In the example, `1-%eax` is done, and the zero-flag is set if `%eax` is 1
- `test` performs a “mental” bitwise and instead
- How can we avoid the `jmp out` instruction in the false-branch?

Example

```
    cmpl $1, %eax    # if (%eax == 1)
    je then         #   %ebx = 3
    movl $2, %ebx   # else
    jmp out         #   %ebx = 2
then:  movl $3, %ebx
out:
```

```
    movl $3, %ebx
    cmpl $1, %eax
    je then
    movl $2, %ebx
then:
```

Conditional jumps, loops

Description

- Loops can be constructed with jumps
- the example performs `for(i=0; i<5; i++)`;
- Substitute the ... for your loop body
- If you can loop “backwards” (i.e. starting at 4 in this example), there is a better way!

Example

```
11:    xorl %eax, %eax    # zero %eax
      cmpl $5, %eax    # if (%eax == 0)
      je out          # exit loop
      ...             # Do something in the loop here!
      incl %eax       # %eax++
      jmp 11          # loop
out:
```

Conditional jumps, loops

Description

- You can also use the `loop` instruction for loops
- `loop` decrements `%ecx` with 1 and jumps if `%ecx` does not become 0
- There are also `loopcc`, where a loops are conditioned.
 - The example loops if `%ebx != 0` and `%ecx` does not become 0

Example

```
    movl $4, %ecx    # Init %ecx
1:   ...             # Do something here!
    loop 1b         # %ecx--; if (%ecx != 0) goto 1b
1:   ...
    cmpl $0, %ebx   # set %eflags according to 0-%ebx
    loopne 1b      # loop
```

Conditional jumps, loops

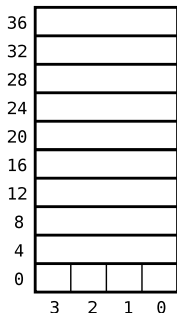
Description

- You can also use the `loop` instruction for loops
- `loop` decrements `%ecx` with 1 and jumps if `%ecx` does not become 0
- There are also `loopcc`, where a loops are conditioned.
 - The example loops if `%ebx != 0` and `%ecx` does not become 0

Example

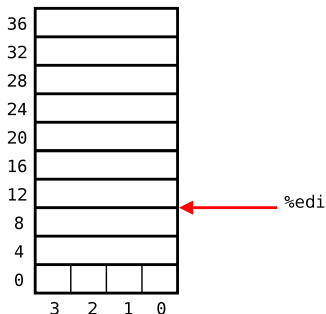
```
    movl $4, %ecx    # Init %ecx
1:   ...             # Do something here!
    loop 1b          # %ecx--; if (%ecx != 0) goto 1b
1:   ...
    cmpl $0, %ebx    # set %eflags according to 0-%ebx
    loopne 1b        # loop
```

Memory references



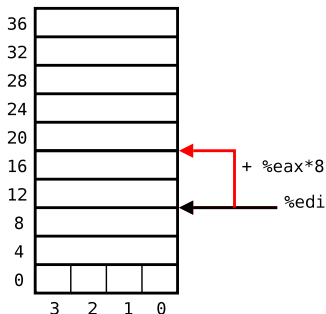
- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - `address = offset(base_reg, index_reg, scale)`
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`

Memory references



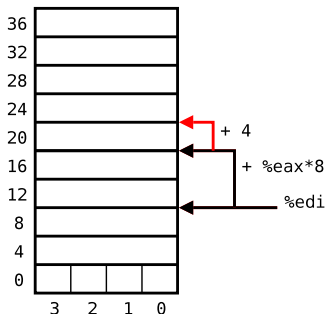
- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - `address = offset(base_reg, index_reg, scale)`
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`

Memory references



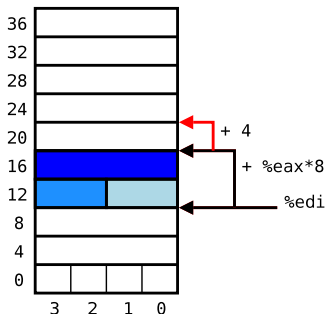
- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - $\text{address} = \text{offset}(\text{base_reg}, \text{index_reg}, \text{scale})$
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`

Memory references



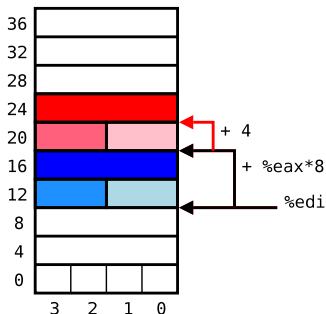
- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - $\text{address} = \text{offset}(\text{base_reg}, \text{index_reg}, \text{scale})$
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`

Memory references



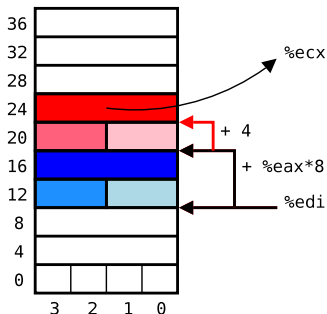
- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - $\text{address} = \text{offset}(\text{base_reg}, \text{index_reg}, \text{scale})$
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`
- Useful for handling data structures

Memory references



- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - $\text{address} = \text{offset}(\text{base_reg}, \text{index_reg}, \text{scale})$
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`
- Useful for handling data structures

Memory references



- Most IA-32 instructions can do memory references
- Memory references are quite complex:
 - $\text{address} = \text{offset}(\text{base_reg}, \text{index_reg}, \text{scale})$
- Example:
 - `movl 4(%edi, %eax, 8), %ecx`
- Useful for handling data structures
- There are shorter forms:
 - `addb $1, (%edi)`: (only base register)
 - `movl 4(%esp), %eax`: (base and offset)

Description

- You can only have one memory reference per instruction
- Memory references can also be absolute (1,2,4)
- but beware: (2 and 3) looks similar, but
 - (1,2, and 4) copies the value from an address
 - (3 and 5) copies the *address*

Examples

```
1  movl 0x1000, %eax    # Copy 4 bytes from address 4096
2  movl label, %eax    # Copy 4 bytes from address 'label'
3  movl $label, %eax   # Copy the value label into %eax
4  movl 128, %eax      # Copy 4 bytes from address 128
5  movl $128, %eax     # Copy the value 128 into %eax
```

Memory references, more

Description

- You can only have one memory reference per instruction
- Memory references can also be absolute (1,2,4)
- but beware: (2 and 3) looks similar, but
 - (1,2, and 4) copies the value from an address
 - (3 and 5) copies the *address*

Examples

```
1  movl 0x1000, %eax    # Copy 4 bytes from address 4096
2  movl label, %eax     # Copy 4 bytes from address 'label'
3  movl $label, %eax   # Copy the value label into %eax
4  movl 128, %eax      # Copy 4 bytes from address 128
5  movl $128, %eax     # Copy the value 128 into %eax
```

Memory references, more

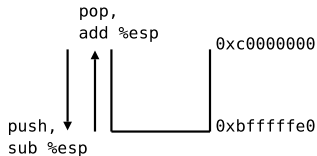
Description

- You can only have one memory reference per instruction
- Memory references can also be absolute (1,2,4)
- but beware: (2 and 3) looks similar, but
 - (1,2, and 4) copies the value from an address
 - (3 and 5) copies the *address*

Examples

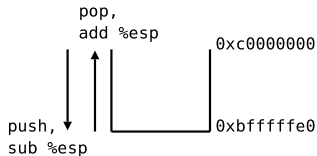
```
1  movl 0x1000, %eax    # Copy 4 bytes from address 4096
2  movl label, %eax    # Copy 4 bytes from address 'label'
3  movl $label, %eax   # Copy the value label into %eax
4  movl 128, %eax      # Copy 4 bytes from address 128
5  movl $128, %eax     # Copy the value 128 into %eax
```

The stack



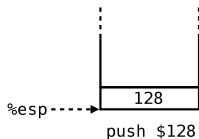
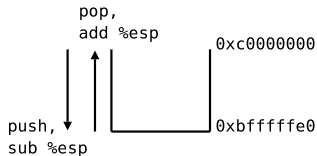
- The stack is where “automatic” (local) variables are stored
- The stack top is stored in `%esp`
 - `addl $8, %esp` **decreases** the stack size with 8 bytes
 - `subl $8, %esp` **increases** the stack size with 8 bytes
- `push` pushes a value on the stack
- `pop` pops a value from the stack

The stack



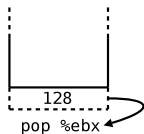
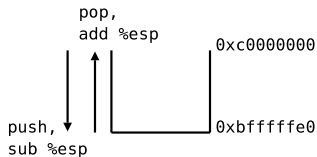
- The stack is where “automatic” (local) variables are stored
- The stack top is stored in `%esp`
 - `addl $8, %esp` **decreases** the stack size with 8 bytes
 - `subl $8, %esp` **increases** the stack size with 8 bytes
- `push` pushes a value on the stack
- `pop` pops a value from the stack

The stack



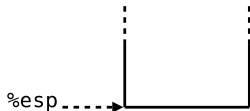
- The stack is where “automatic” (local) variables are stored
- The stack top is stored in `%esp`
 - `addl $8, %esp` **decreases** the stack size with 8 bytes
 - `subl $8, %esp` **increases** the stack size with 8 bytes
- `push` pushes a value on the stack
- `pop` pops a value from the stack

The stack



- The stack is where “automatic” (local) variables are stored
- The stack top is stored in %esp
 - `addl $8, %esp` **decreases** the stack size with 8 bytes
 - `subl $8, %esp` **increases** the stack size with 8 bytes
- `push` pushes a value on the stack
- `pop` pops a value from the stack

Function calls, C calling convention



Description

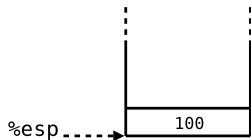
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

Function calls, C calling convention



Description

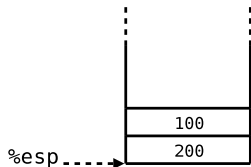
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100  
pushl $200  
call fn  
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax  
     movl 8(%esp), %edx  
     ...  
     movl $0, %eax  
     ret
```

Function calls, C calling convention



Description

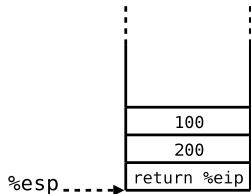
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100  
pushl $200  
call fn  
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax  
     movl 8(%esp), %edx  
     ...  
     movl $0, %eax  
     ret
```

Function calls, C calling convention



Description

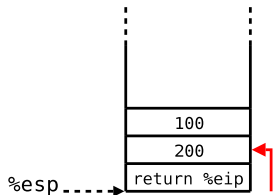
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

Function calls, C calling convention



Description

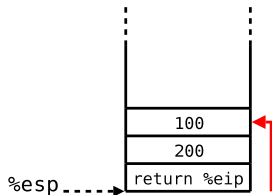
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

Function calls, C calling convention



Description

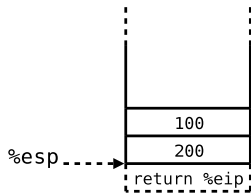
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

Function calls, C calling convention



Description

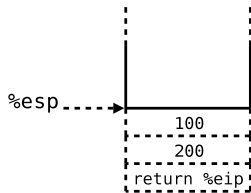
- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

Function calls, C calling convention



Description

- Arguments are pushed in reverse order on the stack
- The PC is pushed on `call` and popped on `ret`
- `%eax` contains the return value
- The stack pointer is restored by the caller

Example

```
pushl $100
pushl $200
call fn
addl $8, %esp
```

```
fn:  movl 4(%esp), %eax
     movl 8(%esp), %edx
     ...
     movl $0, %eax
     ret
```

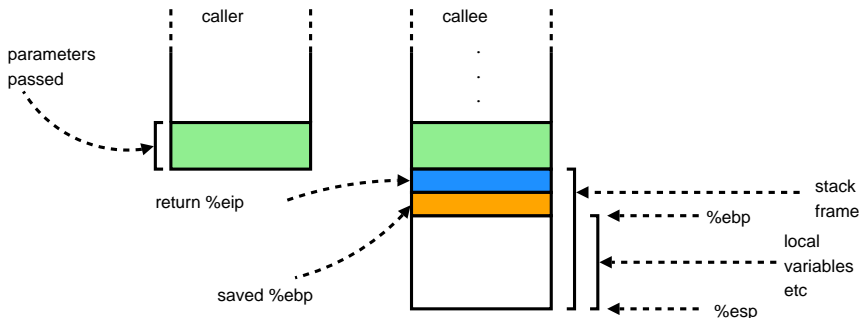
Function calls, C calling convention (2)

- Arguments are always passed on the stack in reverse order.
- `%esp`, `%ebp`, `%ebx`, `%esi`, `%edi` must *not* be changed by the function
- `%eax` contains the result of the function.
- Passed arguments may be overwritten by the function
- The caller must restore the stack (i.e. pop the arguments)

C, frame/base pointer

Description

- A stack frame consists of:
 - Local variables, parameters for other procedures
 - Old `%ebp`, `%ebx`, return address etc.
- `%ebp` is used to access function arguments and local variables



C, frame/base pointer, example

Description

- `leave` is an instruction that destroys a stack frame, i.e.

```
movl %ebp, %esp # Set %esp to point to old %ebp
popl %ebp      # Restore old %ebp
```
- there is a corresponding `enter` instruction, unused by GCC

Example

<pre>void proc(int a) { int b = a; }</pre>	<pre>proc: pushl %ebp # save old %ebp movl %esp,%ebp # set stack frame subl \$24,%esp movl 8(%ebp),%eax # Get 'a' movl %eax,-4(%ebp) # save at 'b' leave # leave the stack frame ret</pre>
--	---

C, frame/base pointer, example

Description

- `leave` is an instruction that destroys a stack frame, i.e.

```
movl %ebp, %esp # Set %esp to point to old %ebp
popl %ebp       # Restore old %ebp
```
- there is a corresponding `enter` instruction, unused by GCC

Example

<pre>void proc(int a) { int b = a; }</pre>	<pre>proc: pushl %ebp # save old %ebp movl %esp,%ebp # set stack frame subl \$24,%esp # movl 8(%ebp),%eax # Get 'a' movl %eax,-4(%ebp) # save at 'b' leave # leave the stack frame ret</pre>
--	--

C, frame/base pointer, example

Description

- `leave` is an instruction that destroys a stack frame, i.e.

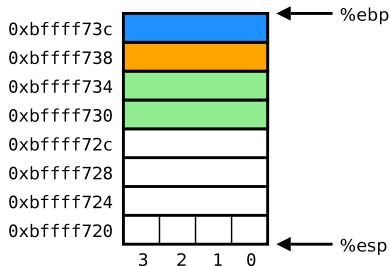
```
movl %ebp, %esp # Set %esp to point to old %ebp
popl %ebp      # Restore old %ebp
```

- there is a corresponding `enter` instruction, unused by GCC

Example

<pre>void proc(int a) { int b = a; }</pre>	<pre>proc: pushl %ebp # save old %ebp movl %esp,%ebp # set stack frame subl \$24,%esp # movl 8(%ebp),%eax # Get 'a' movl %eax,-4(%ebp) # save at 'b' leave # leave the stack frame ret</pre>
--	---

C, frame/base pointer: buffer overflow



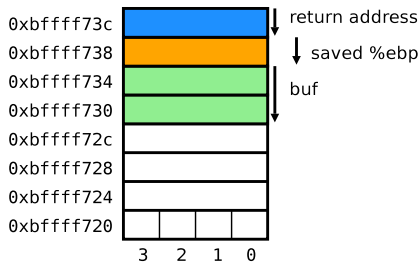
Description

- strcpy can be used to overwrite the return address
- Idea: pass a long string to fn which return to your own code

Example

```
static void fn(char *str) {  
    char buf[8];  
    strcpy(buf, str);  
}  
  
int main(int argc, char *argv[]) {  
    fn(argv[1]);  
    return 0;  
}
```

C, frame/base pointer: buffer overflow



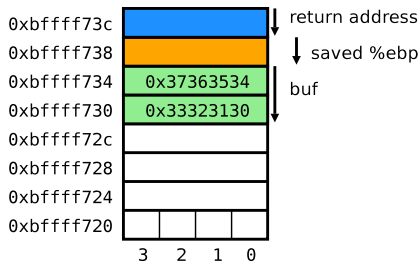
Description

- strcpy can be used to overwrite the return address
- Idea: pass a long string to fn which return to your own code

Example

```
static void fn(char *str) {  
    char buf[8];  
    strcpy(buf, str);  
}  
  
int main(int argc, char *argv[]) {  
    fn(argv[1]);  
    return 0;  
}
```

C, frame/base pointer: buffer overflow



```
$ ./overflow 01234567
```

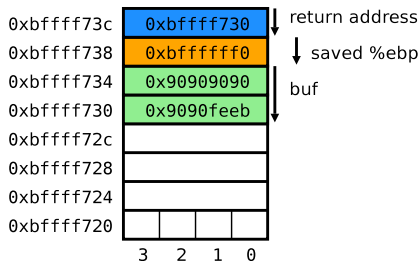
Description

- strcpy can be used to overwrite the return address
- Idea: pass a long string to fn which return to your own code

Example

```
static void fn(char *str) {  
    char buf[8];  
    strcpy(buf, str);  
}  
  
int main(int argc, char *argv[]) {  
    fn(argv[1]);  
    return 0;  
}
```

C, frame/base pointer: buffer overflow



```
$ ./overflow 01234567
```

```
$ ./overflow 'cat attack.bin'
```

```
1:      jmp 1b      # Loop forever
.fill   6, 1, 0x90 # 6 nops (0x90)
.long   0xbffffff0 # old %ebp
.long   0xbffff730 # return %eip
```

Description

- strcpy can be used to overwrite the return address
- Idea: pass a long string to fn which return to your own code

Example

```
static void fn(char *str) {
    char buf[8];
    strcpy(buf, str);
}

int main(int argc, char *argv[]) {
    fn(argv[1]);
    return 0;
}
```

1 Motivation

- Introduction to the IA-32
- IA-32 History
- IA-32 Architecture

2 Assembly Language Introduction

- Compiling and Writing Assembly Language
- Handling of Data
- Instructions, Labels and Registers

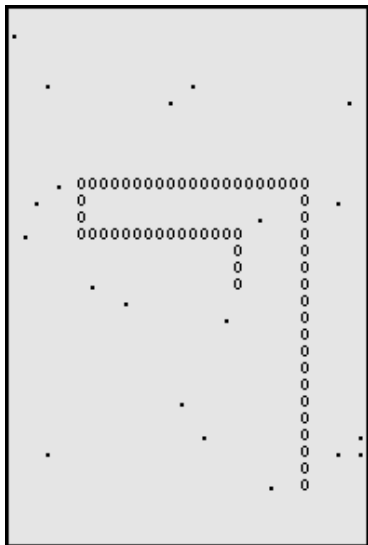
3 Writing IA-32 Programs

- Interfacing with the Operating System
- Arithmetic/Logic Instructions
- Program Flow (branches)
- Referencing Memory

4 Assembly Laboration Assignment

- A good debugger is very important when coding assembly.
- Use GDB!
 - Breakpoints
 - Step to next instruction
 - Check register values
 - Examine stack and data
 - Etc etc.
- Examples on
http://www.ipd.bth.se/ska/unix_programming.html

The Nibbles lab



- You are to produce a working “Nibbles” game written in IA-32 assembly
- The game should be a playable, complete Nibbles implementation
- You should book a time for examination by sending an email to ska@bth.se
- See idenet for more information

The Nibbles lab, competition

- There is also a competition associated with the lab!
- The goal of the competition is to produce the smallest nibbles implementation possible
 - The size is measured as the size of the `.text` plus the `.data` sections
- You are **encouraged** to use dirty tricks to take down the size
 - For example: the `aaa`-instruction was used in a similar competition for something completely different than intended
 - Tricks with operand sizes can be beneficial
- The winners will receive a **secret price** (and eternal fame)!

The Nibbles lab, competition winners



Nr	Names	Year	Bytes
1.	Christian Lindblom, Roy Sandgren	2004	163
2.	Anders Olofsson, Ulrik Mikaelsson	2003	212
3.	Magnus Hellfalk, Rickard Katz	2004	228
	Simon Kågström	2003	276
	Simon Kågström, C implementation	2003	486

"Hugi Size Coding Competition"

1.	Altair / ODDS, Finland	48
2.	Maxx, Sweden	51
2.	Mode 19 (group), Singapore	51

- See <http://www.hugi.scene.org/compo/> for "Hugi Size Coding Competition"

Description

- On PPro+, there are conditional move instructions
- These can be used to execute conditional statements without jumps

Example

```
if (ebx == 11)    movl $10, %ebx    # Init ebx to something
    eax = 15      movl $15, %ecx    # ecx = 15
else              movl $5, %eax    # eax = 5 (assume else)
    eax = 5

                  cmpl $11, %ebx  # if ebx == 11, set flags
                  cmovzl %ecx,%eax # if (zero flag set) eax = ecx

                  pushl %eax
                  call exit
```