IA32 instruction-set architecture

course “Essentials of computing systems”

Feb.2022

contents

contents

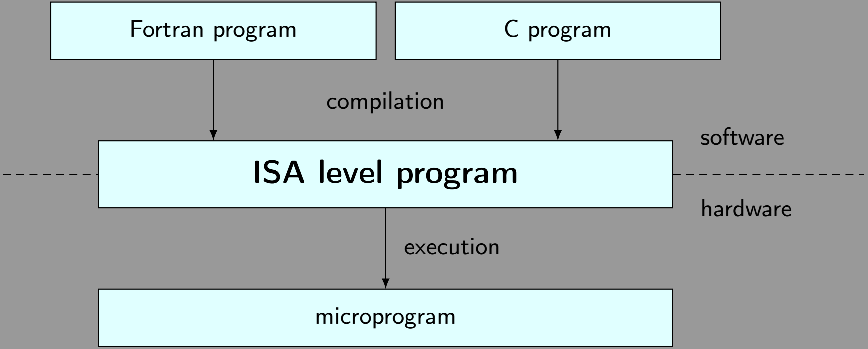
programming in high-level languages

- Programming in a high-level language is productive.
- The abstraction level is higher than the one provided by machine-level languages.
- A low-level code programmer must specify how memory is managed and which low-level instructions will be executed.
- Sophistication of the current compilers makes programming in assembly-language almost unneeded.
- It is difficult for a skilled assembly-language programmer to identify parts of the assembly code generated by a compiler that can be optimised.
- A program written in a high-level language is portable, while assembly code is specific to a reduced set of machines.
- The ability to read and understand assembly code is a relevant skill for professional programmers.

assembly and ISA-level code

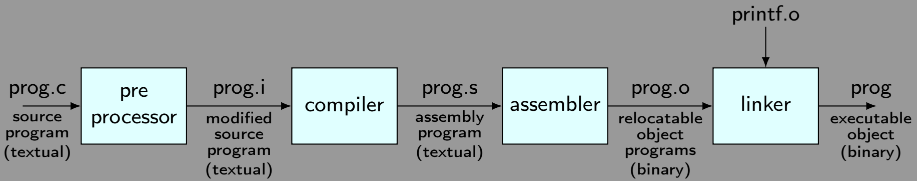
- By invoking the compiler with appropriate flags, it generates a file showing its output in assembly code.
- Assembly code is very close to the actual ISA-level code that computers execute.
- It is in a more readable textual format, compared to the binary format of object code.
- The ISA level is the interface between the software and the hardware.
- Programs in several high-level languages are compiled to the ISA level, whose programs can be directly executed by microprogrammed hardware.
- The ISA level constitutes the language that both compilers and the hardware must understand.

interface between software and hardware



compilation system

- To run a C program, each C statement must be translated into a sequence of low-level machine-language instructions.
- These instructions are packed in a form called an executable object program and stored as a binary file.
- The translation from source file `prog.c` to its corresponding executable object file can be accomplished by invoking the gcc compiler: `gcc prog.c -o prog`


printf.o
prog.c
source program (textual)
pre processor
prog.i
modified source program (textual)
compiler
prog.s
assembly program (textual)
assembler
prog.o
relocatable object programs (binary)
linker
prog
executable object (binary)

preprocessing phase

- In the preprocessing phase, the original C program is modified according to directives that begin with the # character.
- If an `#include <stdio.h>` directive is in the C program being compiled, the preprocessor inserts the contents of the header file `stdio.h` directly into the program text.
- If the program contains the `#define MAX` 30 directive, all instances of the string "MAX" are replaced by the string "30".
- The preprocessor produces another C program, typically with the .i suffix.
- The gcc compiler can be instructed to output the intermediate files that are produced, if the appropriate options are used.
- The file generated by the preprocessor can be obtained by invoking the gcc compiler: `gcc -E prog.c -o prog.i`

compilation phase

- During the compilation phase, the compiler translates the text file `prog.i` into the text file `prog.s`, which contains an equivalent assembly-language program.
- Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form.
- The assembly language is useful, since it constitutes a common language for compilers for different high-level languages.
- The `gcc` compiler can also generate a file with the assembly code: `gcc -S prog.c -o prog.s`

assembly phase

- In the assembly phase, the assembler:
  1. translates `prog.s` into machine-language instructions,
  2. packages them in a format designated as a relocatable object program,
  3. stores the result in the object file `prog.o`.
- This file is a binary file, whose bytes encode machine language instructions rather than characters.
- This file should not be open with a text editor, since it contains binary codes.
- To obtain the object file, `gcc` needs to be called with:

```
gcc -c prog.c -o prog.o
```

linking phase

- Usually, programs include other modules, like standard libraries.
- If `prog.c` calls `printf`, it needs to include the `stdio` library.
- The `printf` function resides in a separate precompiled object file (`printf.o`), which must be linked with the `prog.o` file.
- During the linking phase, the linker handles this merge.
- The result is the `prog` file, which is an executable.

loading phase

- To run the `prog` executable, the loader is invoked.
- The loader is part of the operating system and is responsible for loading programs and libraries.
- It loads programs into memory and prepares them for execution.
- Loading a program involves copying to the main memory the machine-level instructions that are saved in its executable file.
- Once loading is complete, the operating system passes the control to the loaded program code.

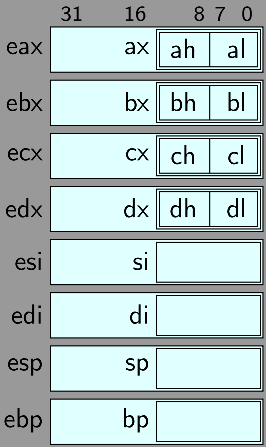
contents

important aspects

- IA-32 (Intel Architecture, 32-bit) is the 32-bit version of the x86 instruction set architecture, developed by Intel and first implemented in the 80386 microprocessor.
- An overview of the most important aspects of this machine-oriented language is provided:
  - the registers,
  - the data types,
  - the operands for the instructions,
  - different groups of instructions (data movement, arithmetic, logical, control),
  - the way procedures are handled, and
  - how data structures are represented.

registers

- IA32 CPUs contain eight 32-bit registers.
- These registers are used to store numeric values and pointers.
- The six registers eax-edi can be considered general-purpose.
- This is broken by some instructions that use some registers as sources/destinations.
- All eight registers can be accessed as either 16 bits (word) or 32 bits (double word).
- The two low-order bytes of the four registers eax-edx can be accessed independently.
- The two registers esp-ebp contain pointers to important places in the stack.



registers

- The low-order two bytes of the first four registers can be independently manipulated by byte-oriented instructions.
- This feature was provided in the 8086, a 16-bit microprocessor, to allow backward compatibility to the 8008 and 8080.
- When a byte instruction updates a single-byte register, the remaining three bytes of the full register remain unchanged.
- Similarly, the low-order 16 bits of each register can be read or written by word-oriented instructions.
- IA32 inherits many features of all its predecessors, namely 16-bit microprocessors.
- The instruction pointer (eip) indicates the address in memory of the next instruction to be executed.

data types

- The term "word" is used to refer to a 16-bit data type, since IA32 is an expansion from a 16-bit architecture.
- 32-bit quantities are referred to as "double words."
- The `MOV` instruction has three variants: `movb`, `movw`, and `movl`.

C data type	Intel data type	GAS suffix	size (bytes)
`char`	Byte	b	1
`short`	Word	w	2
`int`	Double word	l	4
`unsigned`	Double word	l	4
`long int`	Double word	l	4
`unsigned long`	Double word	l	4
`char *`	Double word	l	4

operands

- Instructions have operands, used to specify the source values for the operation and the destination where to store the result.
- There are three major forms for the operands.
- Immediate is used to constant values.
- Constants are written with a '\$' followed by an integer using standard C notation (`$-577` or `$0x1F`).
- Register is used to denote the contents of one of the registers.
- A register is specified with a leading '%' followed by its name.
- Memory is used to memory locations, according to an address that is calculated.
- The memory is viewed as an array of bytes.
- The notation $M[addr]$ refers to the value stored in memory starting at address *addr*.

operands

type	form	operand value	name
immediate	\$Imm	Imm	constant
register	%R$_a$	R_a	register
memory	Imm	$M[Imm]$	absolute
	(%R$_b$)	$M[R_b]$	indirect
	Imm(%R$_b$)	$M[R_b + Imm]$	base+offset
	(%R$_b$,%R$_i$)	$M[R_b + R_i]$	indexed
	Imm(%R$_b$,%R$_i$)	$M[R_b + R_i + Imm]$	indexed
	(,%R$_i$,s)	$M[R_i \cdot s]$	scale indexed
	Imm(,%R$_i$,s)	$M[R_i \cdot s + Imm]$	scale indexed
	(%R$_b$,%R$_i$,s)	$M[R_b + R_i \cdot s]$	scale indexed
	Imm(%R$_b$,%R$_i$,s)	$M[R_b + R_i \cdot s + Imm]$	scale indexed

data movement instructions

- Moving data is a fundamental operation in a computer.
- The term “move” is a synonym of “copy,” since the values in the source remain there unchanged.

instruction	effect	description
`MOV s, d`	$d \leftarrow s$	move
`movb`		move byte
`movw`		move word
`movl`		move double-word
`PUSH s`	$esp \leftarrow esp - size(s)$	push
`pushl`	$M[esp] \leftarrow s$	push double-word
`pushw`		push word
`POP d`	$d \leftarrow M[esp]$	pop
`popl`	$esp \leftarrow esp + size(s)$	pop double-word
`popw`		pop word

data movement instructions

❶	`movl $0x40, %eax`	Immediate–Register
❷	`movl %ebp, %esp`	Register–Register
❸	`movl (%edi,%ecx), %edx`	Memory–Register
❹	`movl $-15, (%esp,%eax)`	Immediate–Memory
❺	`movl %ebx, -12(%ebp)`	Register–Memory

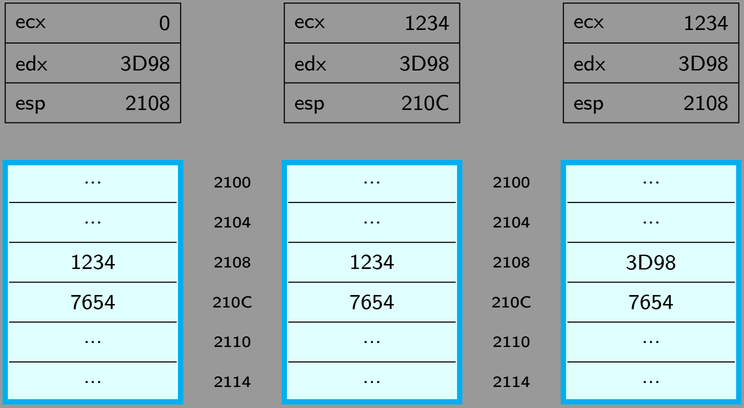
---

❻	`movb $0x40, %bl`	Immediate 8 bits
❼	`movw $0x40, %ax`	Immediate 16 bits
❽	`movb (%edi,%ecx), %dh`	Memory–Register
❾	`movw $-15, (%esp,%eax)`	Immediate–Memory
❿	`movw %bx, -12(%ebp)`	Register–Memory

stack

- The IA32 programming model also provides a stack, where temporary data can be stored.
- This stack is stored in some region of the main memory.
- It is accessed through the instructions `pushl` and `popl`.
- The stack pointer esp holds the memory address of the top stack element.

stack



initially popl %ecx pushl %edx

arithmetic instructions

instruction	effect	description
`leal s, d`	$d \leftarrow address(s)$	load effective address
`ADD s, d`	$d \leftarrow d + s$	add
`addl`		add double-words
`INC d`	$d \leftarrow d + 1$	increment
`incl`		increment double-word
`SUB s, d`	$d \leftarrow d - s$	subtract
`subl`		subtract double-words
`DEC d`	$d \leftarrow d - 1$	decrement
`decl`		decrement double-word
`NEG d`	$d \leftarrow -d$	two's-complement
`negl`		two's-complement double-word
`NOT d`	$d \leftarrow \sim d$	one's-complement
`notl`		one's-complement double-word

arithmetic instructions

instruction	effect	description
`IMUL s, d`	$d \leftarrow d \times s$	signed multiply
`imull`		signed multiply double-words
`IMUL d`	$d \leftarrow d \times R_a$	multiply
`imull`		signed multiply double-words
`IDIV s`	$R_d : R_a \leftarrow R_a / s$	signed division
`idivl`		signed division double-words
`SAR n, d`	$d \leftarrow d \gg n$	arithmetic right shift
`sarl`		arithmetic right shift double-word
`SAL n, d`	$d \leftarrow d \ll n$	arithmetic left shift
`sall`		arithmetic left shift double-word

arithmetic instructions

- The single-operand IMUL instruction executes a signed multiply of a byte, word, or double-word by the contents of a register (al, ax, or eax) and stores the product in a register (ax, dx:ax or edx:eax).

```
imulb 5(%edi)
```

- The two-operand IMUL executes a signed multiply of a register or memory word or double-word by a register word or double-word and stores the product in the latter.

```
imulw 8(%edi), %dx
```

- The three-operand instruction executes a signed multiply of a 16- or 32-bit immediate by a register or memory word or long and stores the product in a specified register word or long.

```
imull $12345678, 4(%edi), %eax
```

- The instruction MUL works in the same way as IMUL but executes an unsigned multiplication.

arithmetic instructions

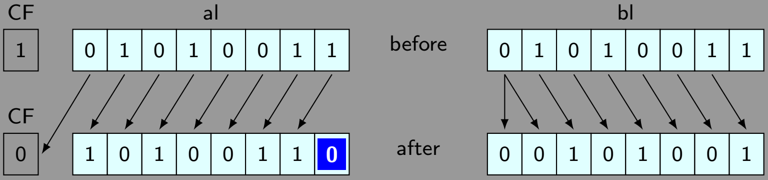
- `IDIV` executes signed division.
- It divides a 16-, 32-, or 64-bit register value (dividend) by a register or memory byte, word, or long (divisor).
- The size of the divisor implies the specific registers used as the dividend, quotient, and remainder.

divisor size	dividend	quotient	remainder
byte	ax	al	ah
word	dx:ax	ax	dx
double	edx:eax	eax	edx

- If the resulting quotient is too large to fit in the destination, or if the divisor is 0, an interrupt 0 is generated.
- The instruction `DIV` works in the same way as `IDIV` but executes an unsigned division.

arithmetic instructions

- SAL left shifts (multiplies) the operand for a count and stores the product in that operand.
- The MSB is shifted into the carry flag; the low-order bit is cleared.
- The instruction SAR right shifts (signed divides) the operand for a count, leaving the MSB unchanged.
- In both cases, the shift count is specified by an immediate value or by the contents of register ecl.



`salb $1, %al`   `sarb $1, %bl`

logical instructions

instruction	effect	description

`AND s, d`
`andl`
	d $\leftarrow$ d$\wedge$s	bitwise AND
		bitwise logical AND double-words

`OR s, d`
`orl`
	d $\leftarrow$ d$\vee$s	bitwise OR
		bitwise logical OR double-words

`XOR s, d`
`xorl`
	d $\leftarrow$ d$\veebar$s	bitwise logical XOR
		bitwise logical XOR double-words

`SHR n, d`
`shrl`
	d $\leftarrow$ d$\gg$n	logical right shift
		logical right shift double-word

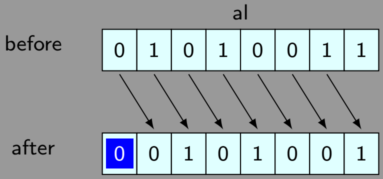
`SHL n, d`
`shll`
	same as `SAL`	logical left shift
		logical left shift double-word

logical instructions

al	0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 1 1 1 1
bl	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
result	0 0 0 0 0 1 0 1	0 1 0 1 1 1 1 1	0 1 0 1 1 0 1 0
	`andb %bl, %al`	`orb %bl, %al`	`xorb %bl, %al`

logical instructions

- SHL is the same operation of SAL. They are synonymous mnemonics.
- SHR right shifts (unsigned divides) the operand for a count.
- It sets the MSB to 0.
- The shift count is specified by an immediate value or by the contents of register ecl.



```
shrb $1, %al
```

control instructions

- By default, machine-level instructions are executed in sequence.
- Often, it is however necessary to break this default rule.
- This happens in constructs available in high-level programming languages like `IF-THEN-ELSE`, `FOR`, and procedure calls.
- IA32 includes various instructions that are relevant to control the flow of execution.
- Usually, the flow of control is dependent on a given condition, i.e., on the values of some variables.
- If that condition is true, the program jumps to an instruction that is not the next one, thus breaking the default behaviour.
- Otherwise, the next instruction is executed.

condition codes

- A condition code (status flag) is the mechanism used by processors to save a specific attribute related to the last arithmetic or logic operation.
- By testing a subset of these flags, conditional jumps to specific locations into the code can be performed.
- IA32 provides the following condition codes:
  - **CF carry flag**: The most recent operation generated a carry out of the MSB. This flag is used to detect overflow for unsigned operations.
  - **OF overflow flag**: The most recent operation caused a two's-complement overflow, either negative or positive.
  - **SF sign flag**: The most recent operation yielded a negative value.
  - **ZF zero flag**: The most recent operation yielded zero.

test instructions

instruction	effect	description
`CMP s2,s1`	s_1-s_2	compare
`cmpb`		compare bytes
`cmpw`		compare words
`cmpl`		compare double-words
`TEST s2,s1`	$s_1 \wedge s_2$	test
`testb`		test bytes
`testw`		test words
`testl`		test double-words

SET instructions

instruction	effect	set condition
`sete d`	$d \leftarrow ZF$	equal / zero $(=0)$
`setne d`	$d \leftarrow \neg ZF$	not equal / not zero $(\neq 0)$
`sets d`	$d \leftarrow SF$	negative (<0)
`setns d`	$d \leftarrow \neg SF$	nonnegative (≥ 0)
`setg d`	$d \leftarrow \neg(SF \veebar OF) \wedge \neg ZF$	greater (signed $>$)
`setge d`	$d \leftarrow \neg(SF \veebar OF)$	greater or equal (signed $\geq$)
`setl d`	$d \leftarrow SF \veebar OF$	less (signed $<$)
`setle d`	$d \leftarrow (SF \veebar OF) \vee ZF$	less or equal (signed $\leq$)
`seta d`	$d \leftarrow \neg CF \wedge \neg ZF$	above (unsigned $>$)
`setae d`	$d \leftarrow \neg CF$	above or equal (unsigned $\geq$)
`setb d`	$d \leftarrow CF$	below (unsigned $<$)
`setbe d`	$d \leftarrow CF \vee ZF$	below or equal (unsigned $\leq$)

JUMP instructions

instr.	condition	description
`jmp l`		Direct jump
`je l`	ZF	Jump if equal / zero (=0)
`jne l`	$\neg$ZF	Jump if not equal / not zero ($\neq$0)
`js l`	SF	Jump if negative (<0)
`jns l`	$\neg$SF	Jump if nonnegative ($\geq$0)
`jg l`	$\neg$(SF$\veebar$OF) $\wedge$ $\neg$ZF	Jump if greater (signed >)
`jge l`	$\neg$(SF$\veebar$OF)	Jump if greater or equal (signed $\geq$)
`jl l`	SF$\veebar$OF	Jump if less (signed <)
`jle l`	(SF$\veebar$OF)$\vee$ZF	Jump if less or equal (signed $\leq$)
`ja l`	$\neg$CF$\wedge\neg$ZF	Jump if above (unsigned >)
`jae l`	$\neg$CF	Jump if above or equal (unsigned $\geq$)
`jb l`	CF	Jump if below (unsigned <)
`jbe l`	CF$\vee$ZF	Jump if below or equal (unsigned $\leq$)

JUMP instructions

```
if (a==b)
   c+=3;
else
   c++;
```

```
       cmpl %eax, %ebx
       jne .lb2
.lb1:  addl $3, %ecx
       jmp .lb3
.lb2:  addl $1, %ecx
.lb3:  ...
```

Variables `a`, `b`, and `c` are saved in registers eax, ebx, and ecx, resp.

labels

- A label constitutes a human convenience and is represented by a symbolic name followed by a colon (:).
- Whenever the assembler finds a new label, its memory address is calculated and that information is saved in an auxiliary table.
- When the object code is generated, the assembler uses that data and encodes the targets as part of the jump instructions.
- Direct jumps are written in assembly code by giving a label as the jump target.

jump instructions

```
while (a<50) {              .lb1: cmpl $50, %eax
     ...                          jge .lb2
     a++;                         ...
}                                 incl %eax
                                  jmp .lb1
                            .lb2: ...
```

```
do {                        .lb1: ...
     ...                          incl %eax
     a++;                         cmpl $50, %eax
} while (a<50)                    jl .lb1
                            .lb2: ...
```

jump instructions

- IA32 uses two major encodings for jumps.
- In an absolute jump, the instruction directly indicates the location in the memory to where the program jumps.
- This location occupies four bytes.
- In a relative jump, the value indicated in the instruction is a signed offset that is added to the address of the instruction following the jump instruction to calculate the destination.
- The use of the address of the next instruction results from the fact that the processor updates the IP as one of the first steps in executing an instruction.

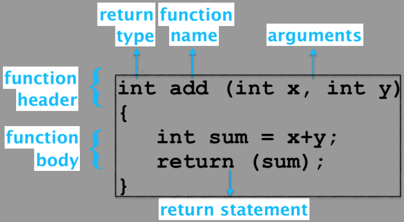
```
jg .lb1    7F 0C
jg .lb2    0F 8F 02 01 00 00
```

jump instructions

```
       movl %edi, %eax                  0: 89 C7
       jmp .lb1                         2: EB 01
.lb2:  decl %eax                        4: 48
.lb1:  testl %eax, 400(%esi)            5: 85 46 90 01 00 00
       jg .lb2                          B: 7F F7
       incl %ebx                        D: 43
```

procedures

- A procedure is a mechanism used to encapsulate and reuse code.
- Assume that procA calls procB.

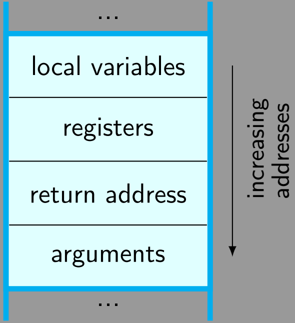


```
int add (int x, int y)
{
    int sum = x+y;
    return (sum);
}
```

- This call implies the following steps:
  1. **Control**: IP must be set to the starting address of the code for procB upon entry; when procB ends, the IP must be set to the instruction in procA that follows the call to procB.
  2. **Arguments and return value**: procA must be able to provide the arguments to procB, which must be able to return a value back to procA.
  3. **Local memory**: procB may need to allocate space for local variables when it starts executing and then free it before returning back to procA.

stack frame

- The structure created in the stack to support the execution of each procedure call is called stack frame.
- It includes space for:
  1. the arguments,
  2. the return information,
  3. the registers, and
  4. the local variables.
- Some parts may be omitted if not needed, to optimise the program for size or time.



procedures

- Before procA calls procB, it needs to put the arguments into the stack frame.
- This can be accomplished by pushing their values into the stack with `PUSH` instructions.
- Often, arguments are passed in registers.
- Afterwards, procA can pass the control to procB, by setting the IP to the starting address of the code for procB.
- This is similar to a `jump`, but the processor records the code location where it should resume the execution of procA.
- This information is recorded by invoking procedure procB with the instruction `call procB`.
- This instruction pushes an address onto the stack, called the return address, and sets the IP to the beginning of procB.
- The return address is computed as the address of the instruction immediately following the `call` instruction.

instructions that support procedures

instruction	effect	description
`call l`	pushl eip; $eip \leftarrow l$	Procedure call
`call *o`	pushl eip; $eip \leftarrow *o$	Procedure call
`leave`	$esp \leftarrow ebp$; popl ebp	Procedure exit
`ret`	popl eip	Return from call

saving registers

- procB is then responsible for allocating the space required for saving the values of registers and the local variables.
- Registers, whose values need to be preserved, are just pushed onto the stack.
- Not all registers need to be saved, as there is a convention that all programmers and compilers are expected to follow.
- A register is classified according to two types:
  1. A callee-saved register is a register whose value must be preserved by the callee procedure;
  2. A caller-saved register is a register whose value must be preserved by the caller procedure.

saving registers

- Registers ebx, esi, edi, and ebp are callee-saved.
- When procA calls procB, procB must preserve the values of these registers, ensuring that they have the same values when procB returns to procA as they had when procB was called.
- procB can preserve a register value by:
  1. not changing its value,
  2. pushing the original value on the stack, using it, and popping the old value from the stack before returning to procA.
- Register ebp is used as a reference pointer for accessing all the elements in the stack frame.
- ebp is a callee-saved register, so all called procedures must save this register.
- Registers eax, ecx, and edx are caller-saved.
- The caller procedure must save these registers, to guarantee that they are not affected by a calling procedure.

space for local variables

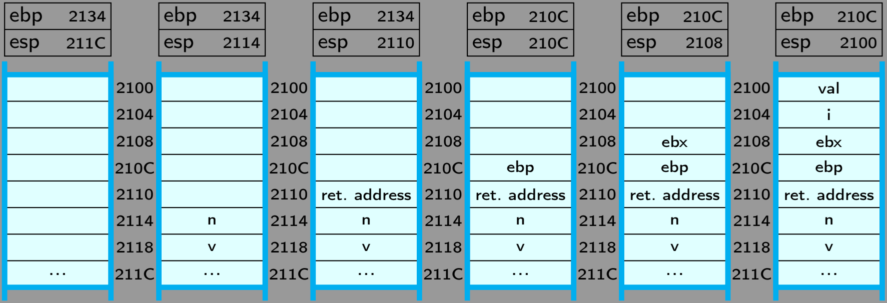
- Space for local uninitialised data can be allocated on the stack by decrementing the stack pointer by an appropriate amount.
- The size of the space for local variables must be at least equal to the sum of the sizes of each local variable.
- Otherwise, the initial values of the local variables can be directly pushed to the stack.
- Each variable must be associated with a specific part of that space.
- Often, local variables are assigned to registers.
- When these two allocations (registers and local data) are finished, procedure procB can start its execution.

return to calling procedure

- When procB is finished, the control must be passed to procA.
- To this end, procB passes the return value, if present, through register eax.
- It also needs to “clean” its stack frame, since it is no longer needed.
- The space for local variables can be deallocated simply by incrementing the stack pointer.
- The next elements to be freed are the registers, by popping their values from the stack.
- The last register to be popped is the IP (eip in IA32).
- This occurs when the instruction `ret` is executed, which pops the return address from the stack and copies it to eip.

stepwise construction of the stack frame

```
int sum (int n, int v) {
    int i, val=0;
    for (i=1; i<n; i++)
        val = val + v + i;
    return (val);
}
```



ebp	2134	2134	2134	210C	210C	210C
esp	211C	2114	2110	210C	2108	2100
2100						val
2104						i
2108					ebx	ebx
210C				ebp	ebp	ebp
2110			ret. address	ret. address	ret. address	ret. address
2114		n	n	n	n	n
2118		v	v	v	v	v
211C	…	…	…	…	…	…

scalar variables

- A scalar variable contains only a value of a primitive data type, like a character, an integer, or a floating-point number.
- Since a scalar variable occupies a reduced number of bytes, it can be saved in the memory or in a given register.
- In general, global variables are allocated in memory.
- The address of each memory address is calculated by the linker, when all modules are merged into an executable program.
- In assembly, the global variables are referred by their symbolic name (i.e., by the respective label).
- Global variables can also be stored in registers, but this is not generalisable, as the number of registers is very reduced.

structured variable

- A structured variable is composed of a set of scalar variables and other structured variables.
- Examples of variables in this category are structs (in C) and arrays/lists.
- The size of these variables does not permit their allocation to registers, so they are stored in the memory.
- C uses a simple implementation of arrays, so the translation into machine code is straightforward.
- In C, one can use pointers to access the elements of the arrays and perform arithmetic with the pointers.

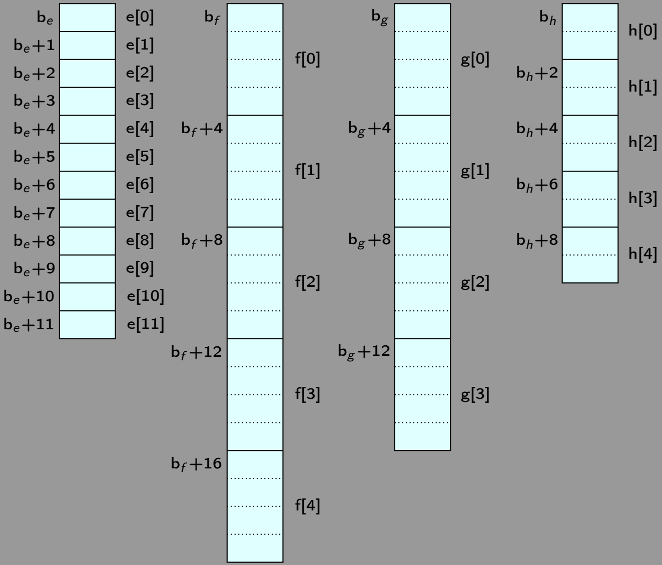
arrays

```
datatype arr[N];
```

- This declaration allocates a contiguous region of $N \times S$ bytes in memory, where S is the size in bytes of the data type.
- It also introduces an identifier `arr` that can be used as a pointer to the beginning of the array.
- The value of this pointer is denoted as b_{arr}.
- The array elements are accessed using an index that ranges from 0 to N-1.
- Array element i is stored at address $b_{\text{arr}} + i \times S$.

arrays

```
char e[12];
int f[5];
char *g[4];
short int h[5];
```



access to arrays

- The addressing modes of IA32 ease the access to arrays.
- If registers ebx and esi store the base address of array `e` and the index *i*, the next instruction initialises `e[i]` with value 0.

```
movb $0, (%ebx, %esi, 1)
```

- If esi is equal to 2, the memory address that is affected is given by $b_e + 2 \times 1$, that is two positions below the base address which is the location for `e[2]`.
- For array `f`, the next instruction initialises `f[i]` with value 10, if registers ebx and esi store the base address of the array `f` and the index *i*.

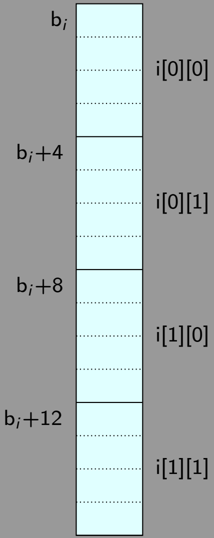
```
movl $10, (%ebx, %esi, 4)
```

multidimensional arrays

- The general mechanism used for unidimensional arrays also applies for multidimensional arrays.

```
int i[2][2];
```

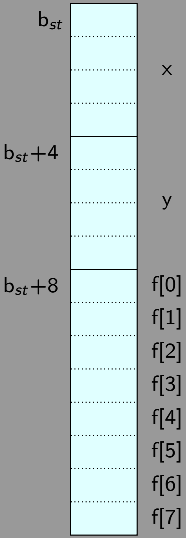
- This array has four integers, so it occupies 16 bytes.
- The array elements are stored in the memory in row-major order.
- All the elements of row 0 go first, which are followed by all elements of row 1.



structs

- With respect to a `struct` declaration, its implementation is similar to that of arrays:
  - all its components are stored in a contiguous area of the memory,
  - the base pointer to that structure is the address of its first byte.
- The compiler uses information about each structure type to store the offsets of all fields, which are displacements with respect to the base address.

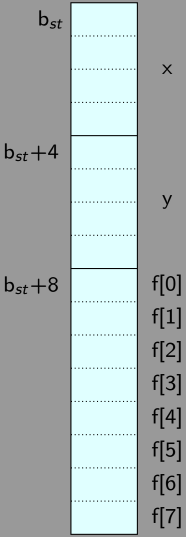
```
struct rec {
    int x;
    int y;
    char f[8];
}
```



structs

- The addressing modes of IA32 are convenient to access structures.
- Suppose that variable `st` is of type `struct rec`.
- If register ebx contains the base address of `st`, the following instructions access `st.x`, `st.y` and `st.f[5]`.

```
movl $10, (%ebx)          /* st.x = 10 */
movl $15, 4(%ebx)         /* st.y = 15 */
movb $75, 13(%ebx)        /* st.f[5] = 'K' */
```


b_{st}
$b_{st}+4$
$b_{st}+8$
x
y
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]