CS354: Machine Organization and Programming

Lecture 14: Midterm1 Review

Monday the October 5th 2015

Section 2

Instructor: Leo Arulraj

Logical Machine Organization

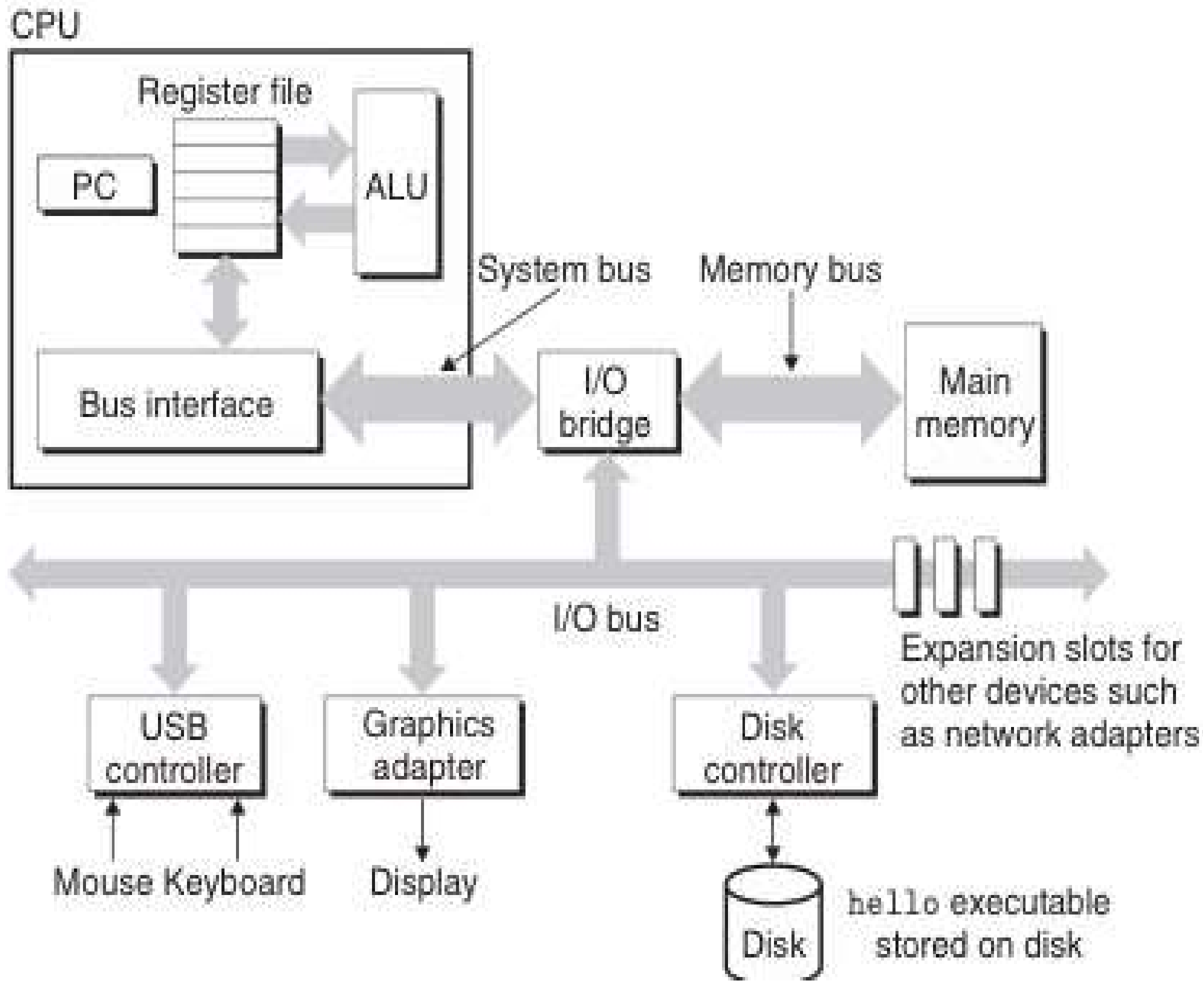


Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

Simple hello world Program

- What is C? A High Level Language
- What is Assembly?
- What is Machine Code?

Compilation Process Overview

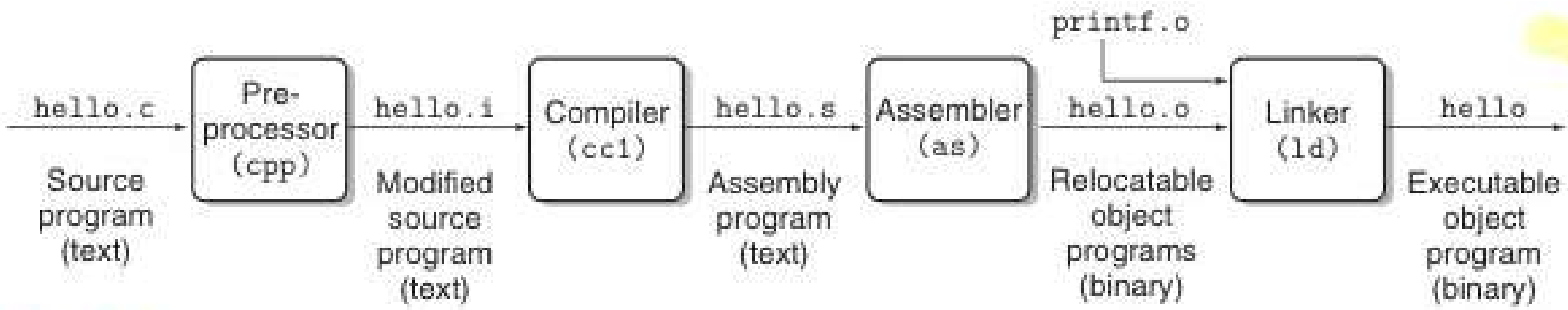


Figure 1.3 **The compilation system.**

Arithmetic Operators

Op.	Description	Example A=10,B=20
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Relational Operators

Op.	Description	Example A=10, B=20
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Op.	Description	Example A=true, B=false
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Op.	Description	**Example** A(60) = 0011 1100 B(13) = 0000 1101
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators 1

Op.	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A

Assignment Operators 2

Op.	Description	Example
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

Op.	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is integer, will return 4.
Unary &	Returns the address of an variable.	&a; will give actual address of the variable.
Unary *	Value of a pointer	*a; will value stored in the address a.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Integer Types

The actual size of integer types varies by implementation. Standard only requires size relations between the data types and minimum sizes for each.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating Point Types

The value representation of floating-point types is implementation-defined

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Void type

1	**Function returns as void** There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

Strings in C

- Strings in C are one dimensional arrays of characters terminated with a null character.

Examples: char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

char greeting[6] = “Hello”;

char* greeting = “Hello”;

Index	**0**	**1**	**2**	**3**	**4**	**5**
Content	H	e	l	l	o	\0
Memory Address.	0x88321	0x88322	0x88323	0x88324	0x88325	0x88326

Declarations

Global Variable: A global variable is a variable that is declared outside **all** functions.

Local Variable: A local variable is a variable that is declared inside a function.

Examples:

const int foo = 10;
// foo is const integer with value 10

char foo;
// foo is a char

double foo();
// foo is a function returning a double

If Statement

```
if(boolean_expression){
    /* statement(s) will execute if the
       boolean expression is true */
}
```

If-else Statement

```
if(boolean_expression){
      /* statement(s) will execute if the boolean
         expression is true */
}else{
      /* statement(s) will execute if the boolean
         expression is false */
}
```

Else-if Statement

```
if(expression){
    /*Block of statements;*/
}else if(expression){
    /*Block of statements;*/
}else{
    /*Block of statements;*/
}
```

Switch

```
switch(expression){
      case constant-expression1:   statements1;

      [case constant-expression2:   statements2;]

      [case constant-expression3:   statements3;]
      [default: statements4;]
}
```

While loop

```
while (expression) {
    Single statement or
    Block of statements;
}
```

For loop

```
for(expression1;expression2;expression3){
    Single statement   or
    Block of statements;
}
```

You can also skip expression1, expression2, expression3.

What does this do ? `for(;;){printf("a\n");}`

Do while loop

```
do{
    Single statement    or
    Block of statements;
}while(expression);
```

Break; Continue; Statements

C provides two commands to control how we loop:

- **break --** exit form loop or switch.
- **continue --** skip 1 iteration of loop.

Goto and Labels

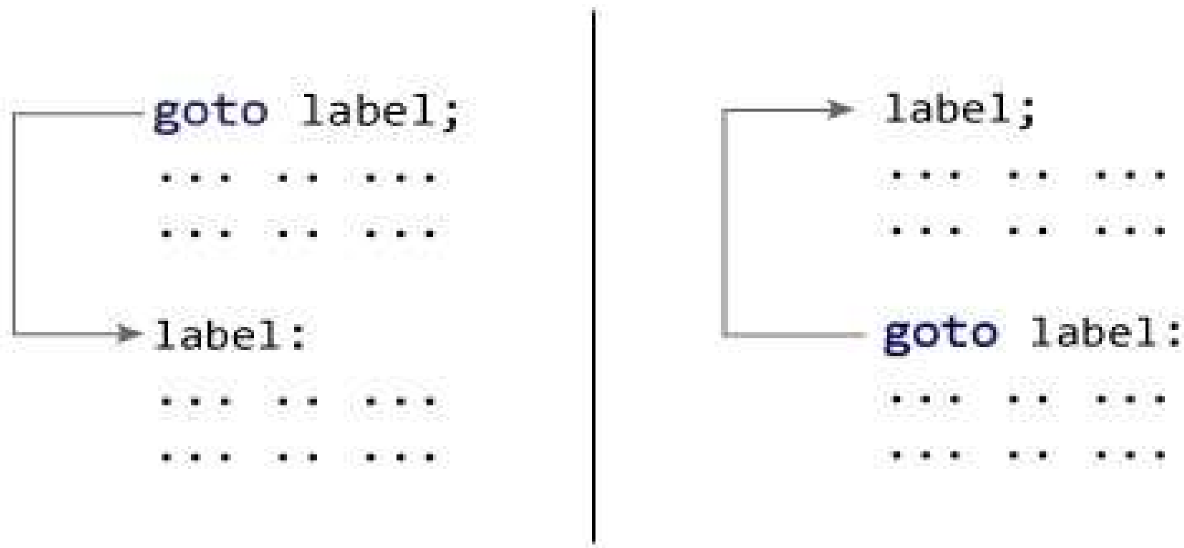
goto label;

………….

………….

………….

label: statement



You can have better label names
(e.g. mycalc, complexcalc etc.)

Functions 1

Function Prototype (Declaration):

return_type function_name(
        type(1) argument(1),….,type(n) argument(n));

Function Definition:

```
return_type function_name(
type(1) argument(1),..,type(n) argument(n))
{
//body of function
}
```

Functions 2

Function Call:

function_name(argument(1),....argument(n));

Return Statement:

return (expression);

C always passes arguments `by value': a copy of the value of each argument is passed to the function; the function cannot modify the actual argument passed to it.

Functions

C always passes arguments `by value': a copy of the value of each argument is passed to the function; the function cannot modify the actual argument passed to it.

```
#include <stdio.h>
int add(int a, int b);
int main(){
     ............
  sum=add(num1,num2);
    ...........
 }
     int add(int a, int b){
       ..................
       .................

    }
   Here,
         a=num1
         b=num2
```

Simple I/O Example

```
int b, a; long int b; char s[10], float d;

printf("%d\n",b);

scanf("%d", &a);

printf("%3d\n",b);

printf("%3.2f\n",d);

printf(”%ld\n",b);
```

Format String 1

Specifier	Description	Example
%i or %d	int	12345
%c	char	y
%s	string	“sdfa”
%f	Display the floating point number using decimal representation	3.1415
%e	Display the floating point number using scientific notation with e	1.86e6
%E	Like e, but with a capital E in the output	1.86E+06
%g	Use shorter of the 2 representations: f or e	3.1 or 1.86e6
%G	Like g, except uses the shorter of f or E	3.1 or 1.86E6

Arrays

Declarations: /* an array of 100 integers */

int ar[100];

Arrays are always allocated consecutively in memory.

Access:

ar[4] = 10; // 5th element set to value 10

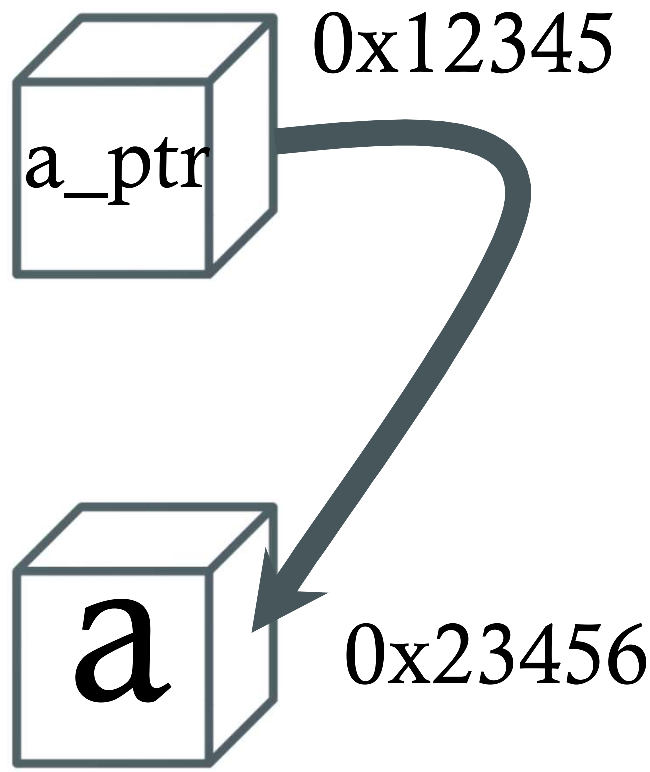
Pointers

A pointer is a memory address.

Simple example:

int a,b;

int*a_ptr = &a;

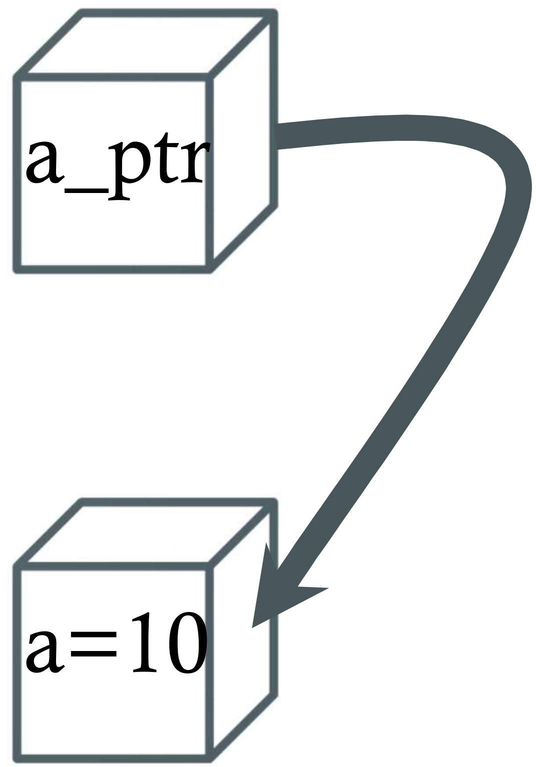


Pointers

A pointer is a memory address.

Simple example:

*a_ptr = 10;



"*" operator is called

"Indirection Operator"

Pointers

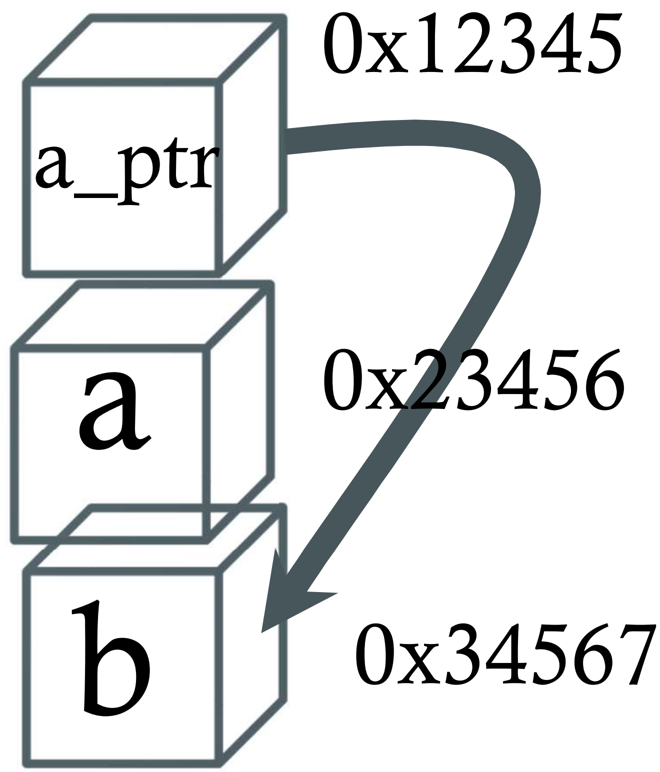
A pointer is a memory address.

Simple example:

a_ptr = &b;

"&" operator is called

"Address of" operator

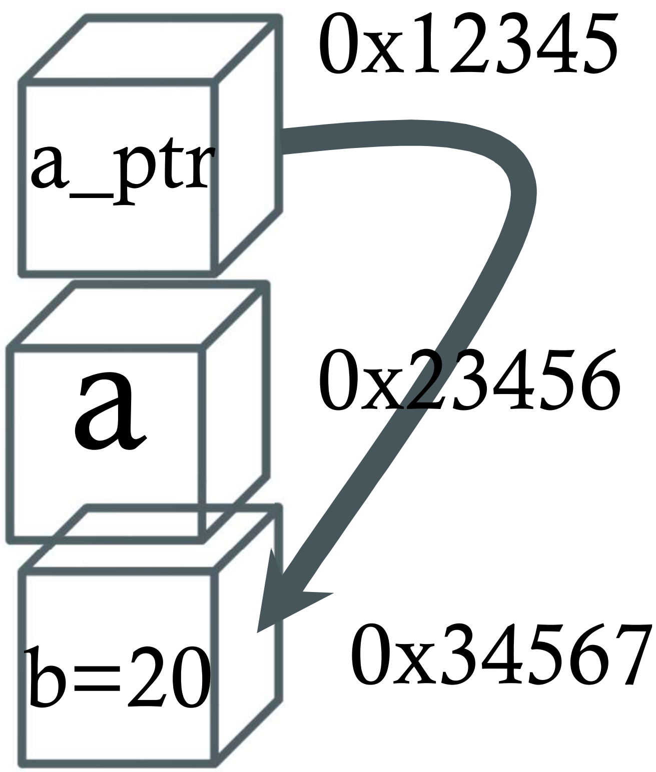


Pointers

A pointer is a memory address.

Simple example:

*a_ptr =20;



Pointers: Some Allowed Operations

1. Assignment to other pointers of the same type
2. Addition and subtraction of a pointer to an integer
3. Assignment of the value 0
4. Comparison to the value 0

Pointers: Some Allowed Operations

```
int a = 3; int b = 8; int c = 0;  /*declaration and
initialization */
int *ap; int *bp; int *cp;  /*declaration of pointers to
integers */
ap = &a; bp = &b; cp = &c;


c = *ap + *bp;
a = b + *cp;
(*bp)++;
cp++;
```

Pointers: Some Unwise Operations

1. Multiplication or division on a pointer
2. Addition or subtraction of two pointer values
3. Assignment of a value (a literal) other than 0 to a pointer

Swap two variables

```
void swap(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Swap two variables using pointers

```
void swap(int *px, int *py){
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Arrays vs. Pointers

Arrays and Pointers are often used interchangeably

Example:

```
int ar[100];  /* an array of 100 integers */
int *arptr = ar;
arptr[4] = 10; //sets the 5th element to 10
```

Arrays vs. Pointers

- And, we could now change the value of the 7th element of the array to 1000 with

```
*(arptr+6) = 1000;
```

- We can even do the same thing with

```
*(ar+6) = 1000;  /* 7th item is at offset of 6 from the element at index=0 */
```

Arrays vs. Pointers

- Stated a little more formally,
  a[i] is the same as ***(a+i)**
  and **&a[i]** is the same as **a+i**
- However, a pointer **is** a variable, but an array name **is not** a variable. So,
  arptr = arr is legal,
  but **arr = arptr** and **arr++** **are not** legal.
- Pointer can be used in place of an array. Array can not be used as a pointer in all scenarios.

Pointers increment with sizeof(type)

```
int ar[5]={0,6,-1,15,102};
int *ap = ar;
printf("ptr ap = %0x val *ap= %d\n",ap, *ap);
ap+=1;
printf("ptr ap = %0x val *ap= %d\n",ap, *ap);
```

Output:

ptr ap = a81b0d60 val *ap= 0

ptr ap = a81b0d64 val *ap= 6

Structures

Structures are a derived type that collect a set of variables under one type

For example,

```
struct line {
   int a, b, c;  /* line is  ax + by = c  */
};

struct line diagonal;
diagonal.a = 1;
diagonal.b = 1;
diagonal.c = 0;
```

The . (period) is an operator on a structure, to access the correct member of the structure.

Operations on Structures

- Copy it
- Assign to it (as a whole unit)
- Get its address (with the & operator)
- Access a member variable (using . operator)
- **CANNOT compare two structures even if they are of the same type.**

The -> operator

- We often have a pointer to a structure and want to access its members and it can be done with:

(*ptr).member

[parantheses needed because unary * is of lower precedence than . operator.]

- Convenient Alternative:

ptr->member

- The dot(.) and **->** operators are left to right associative and have highest precedence. So, use parentheses when needed.

malloc – Basic Memory Allocation

void * **malloc** *(size_t size)*   *[from stdlib.h]*

- returns a pointer to a newly allocated block *size* bytes long, or
- a null pointer if the block could not be allocated.

Example usage:

```
struct foo *ptr;

ptr = (struct foo *) malloc (sizeof (struct foo));

if (ptr == 0) abort ();

memset (ptr, 0, sizeof (struct foo)); //initialize to 0
```

free –Allocating cleared space

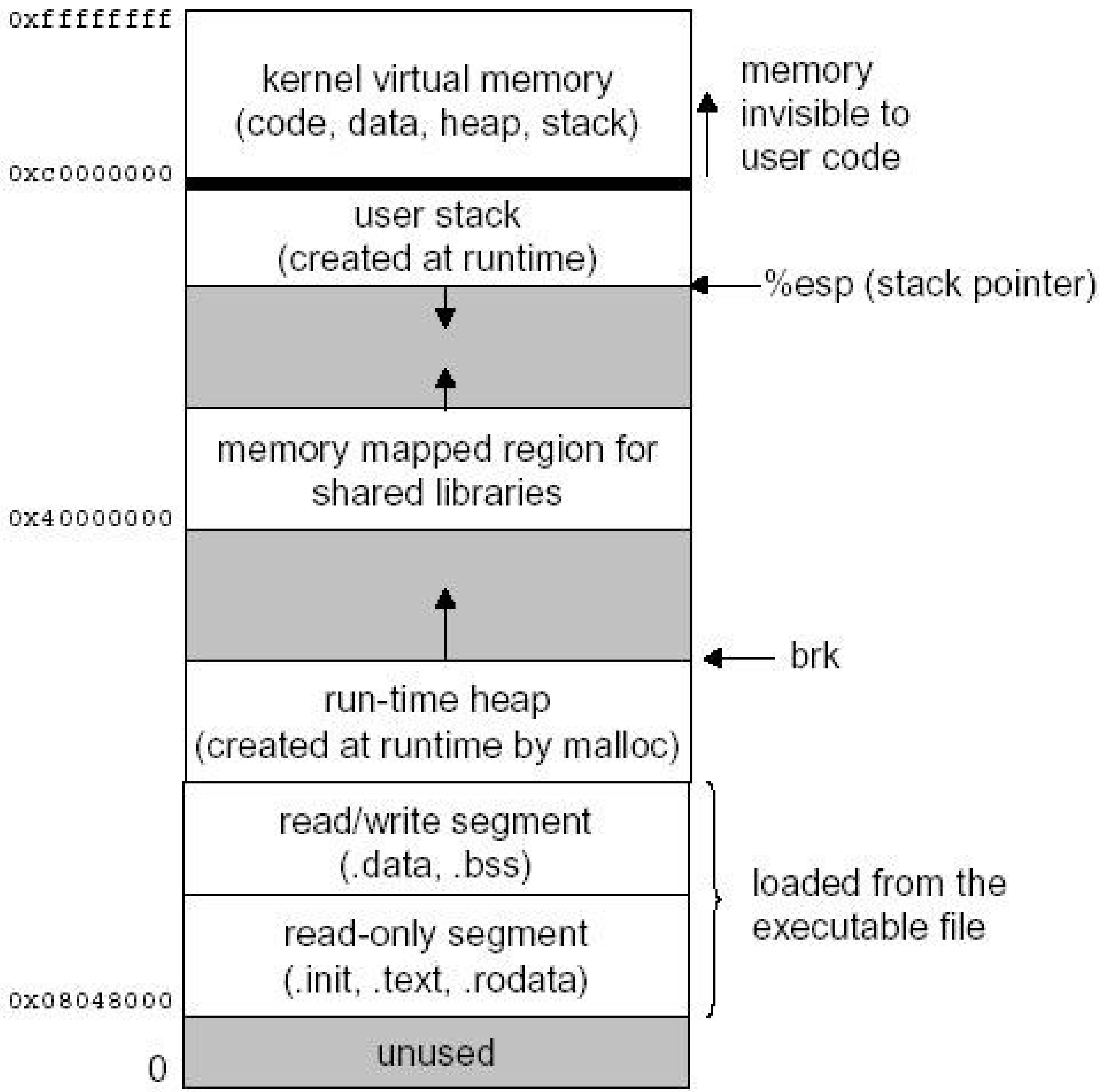
void **free** *(void *ptr)* *[from stdlib.h]*

- When you no longer need a block that you got with malloc or calloc, use the function free to make the block available to be allocated again
- The free function deallocates the block of memory pointed at by *ptr*.
- If you forget to call free, not the end of the world because all of the program's space is given back to the system when the process terminates.

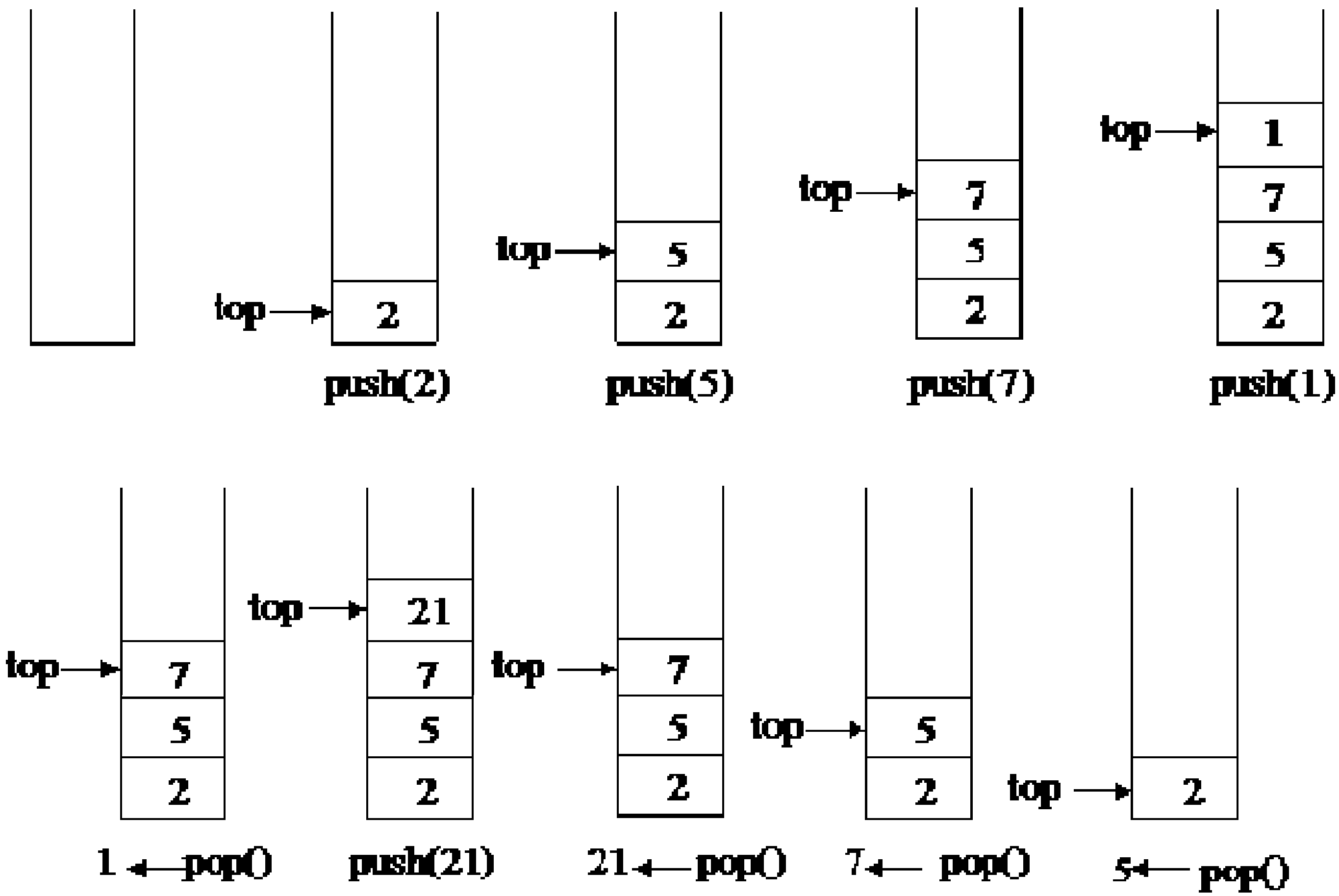
What and *Where* are

- program code (machine code)
- global variables (data)
- stack
- heap

Each can be thought of as residing in its own, separate section of memory. These sections are often identified as segments.

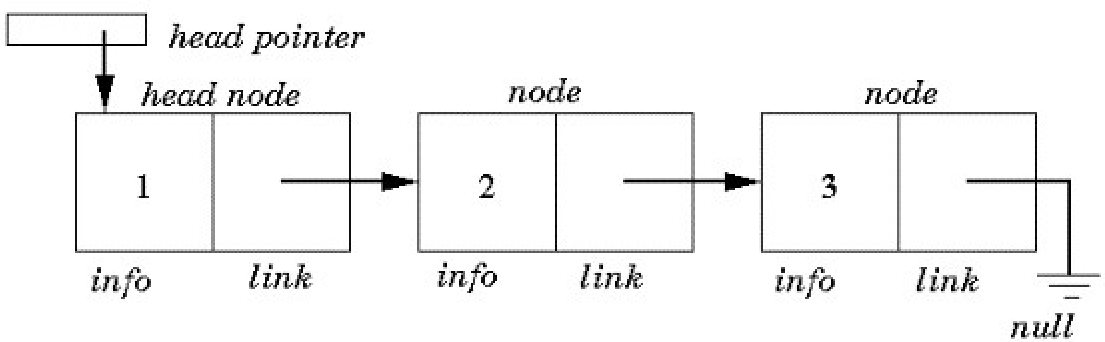

0xffffffff
kernel virtual memory
(code, data, heap, stack)
memory invisible to user code
0xc0000000
user stack
(created at runtime)
%esp (stack pointer)
memory mapped region for shared libraries
0x40000000
brk
run-time heap
(created at runtime by malloc)
read/write segment
(.data, .bss)
loaded from the executable file
read-only segment
(.init, .text, .rodata)
0x08048000
unused
0

Stack


top
2
push(2)
top
5
2
push(5)
top
7
5
2
push(7)
top
1
7
5
2
push(1)
top
7
5
2
1 pop()
top
21
7
5
2
push(21)
top
7
5
2
21 pop()
top
5
2
7 pop()
top
2
5 pop()

Singly Linked List

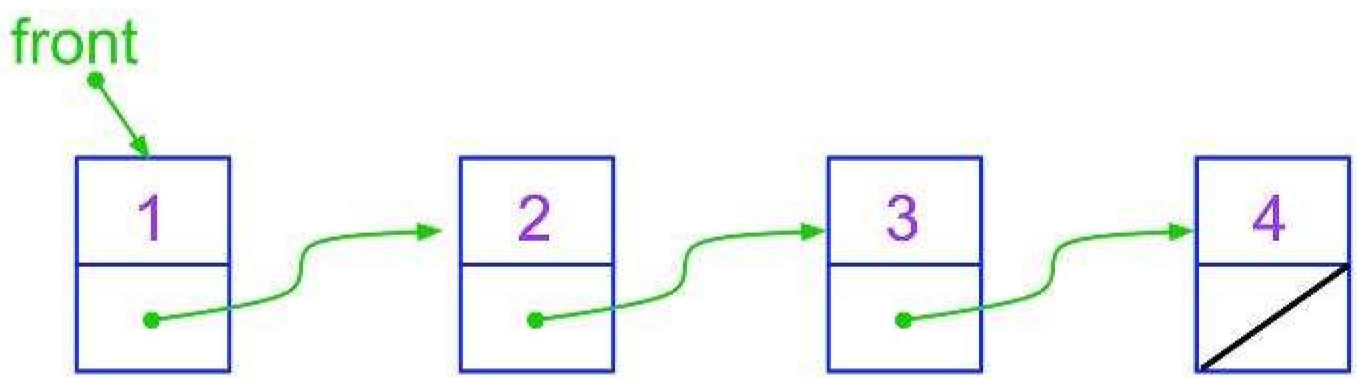
1. Linked list is made up of nodes.
2. Each node points to the next node.
3. The first node is called "**head**" of the linked list.
4. The last node is called **"tail"** of the linked list.



A Linked List

```
struct node {
    int theint;
    struct node *next;
};
```

SINGLY LINKED, BUT IN THE REVERSE ORDER
(ADD TO END OR BACK OF THE LIST)



With the correct code, what happens when this code is executed?

```
ptr = three.next;
ptr = ptr->next;
```

⟵

Runtime error: NULL pointer dereference

In Linux: Segmentation fault (core dumped)

Arithmetic Operations

- Arithmetic Operations
  - addition
  - subtraction
  - multiplication
  - division
- Each of these operations on the integer representations:
  - unsigned
  - two's complement

Addition Truth Table

Carry In	a	b	Carry Out	Sum Bit
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Unsigned Representation

$B2U_w(x_{vec}) = Sum_{i=0->w-1}\ x_i.2^i$

$B2U_4([0101]) = 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 5$

$B2U_w$ is a bijection:
- associates a unique value to each bit vector of length w
- each integer between **0 and 2^w-1** has a unique binary representation as a bit vector of length w

Unsigned Addition

Of two unsigned w bit values X & Y

X + Y equals:

- $X+Y$, if $(X+Y) < 2^w$
- $X+Y-2^w$, if $2^w <= (X+Y) < 2^{w+1}$

Addition

- Unsigned and 2's complement use *the same addition algorithm*
- Due to the fixed precision, throw away the carry out from the msb

```
   00010111
 + 10010010
 ----------
```

Addition

- Unsigned and 2's complement use *the same addition algorithm*
- Due to the fixed precision, throw away the carry out from the msb

```
  00010111
+ 10010010
----------
  10101001
```

Two's complement Representation

$B2T_w(x_{vec}) = -x_{w-1}2^{w-1} + Sum_{i=0->w-2}\ x_i2^i$

$B2T_4([1011]) = -1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = -5$

$B2T_w$ is a bijection:

- associates a unique value to each bit vector of length w
- each integer between **-2^{w-1} and $2^{w-1}-1$** has a unique binary representation as a bit vector of length w

Range of Values for Unsigned and 2's Complement (16 bits)

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

#include <limits.h> declares constants, e.g., ULONG_MAX, LONG_MAX, LONG_MIN

(Values platform specific)

4-bit Unsigned and 2's complement Integers

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

2's Complement Addition

Of two signed 2'complement w bit values X & Y

X + Y equals:

- $X+Y-2^w$, if $2^{w-1} <= (X+Y)$ Positive overflow
- $X+Y$, if $-2^{w-1} <= (X+Y) < 2^{w-1}$ Normal
- $X+Y+2^w$, if $(X+Y) < -2^{w-1}$ Negative overflow

Two's Complement Addition

	1	1	1	1	1	1	1	0	(-2)
+	0	0	0	0	0	0	0	1	(1)
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	(-1)

	1	1	1	1	0	0	0	0	(-16)
+	0	0	1	1	0	0	0	0	(48)
	0	**0**	**1**	**0**	**0**	**0**	**0**	**0**	(32)

Unsigned Overflow Detection

6-bit examples:

```
     0 0 1 1 1 1
   + 0 0 1 1 1 1
   -------------
     0 1 1 1 1 0
 0
```

No Overflow

```
     1 1 1 1 1 1
   + 0 0 0 0 0 1
   -------------
     0 0 0 0 0 0
 1
```

Overflow!

```
     1 0 0 0 0 0
   + 1 0 0 0 0 0
   -------------
     0 0 0 0 0 0
 1
```

Overflow!

Carry out from msbs is overflow in unsigned

Two's Complement Overflow Detection

When adding 2 numbers of like sign

+ to +

- to –

and the sign of the result is different!

+	–
+ +	+ –
–	+
Overflow!	Overflow!

Addition

Overflow detection: 2's complement

6-bit examples

```
  111111 (-1)
+ 111111 (-1)
  ------
  111110 (-2)
```

```
  100000 (-32)
+ 011111 (31)
  ------
  111111 (-1)
```

```
  011111 (31)
+ 011111 (31)
  ------
  111110 ( -2)
```

2’s Complement Inverse

Additive inverse of a 2’complement w bit value X equals:

-2^{w-1}, if $x = -2^{w-1}$

$-X$, if $X > -2^{w-1}$

2's Complement Inverse: Easy Techniques

1) **Toggle all bits and then add 1:**

E.g. Inverse of 0101 (5) is 1011 (-5)

Inverse of 1000 (-8) is 1000 (-8)

2) **Toggle all bits until (not including) the rightmost 1 bit:**

E.g. Inverse of 0111 (7) is 1001 (-7)

Inverse of 1010 (-6) is 0110 (6)

Sign Extension

The operation that allows the same 2's complement value to be represented, but using more bits.

```
      0 0 1 0 1 (5 bits)
0 0 0 0 0 1 0 1 (8 bits)

        1 1 1 0 (4 bits)
1 1 1 1 1 1 1 0 (8 bits)
```

Zero Extension

The same type of thing as sign extension, but used to represent the same unsigned value, but using more bits

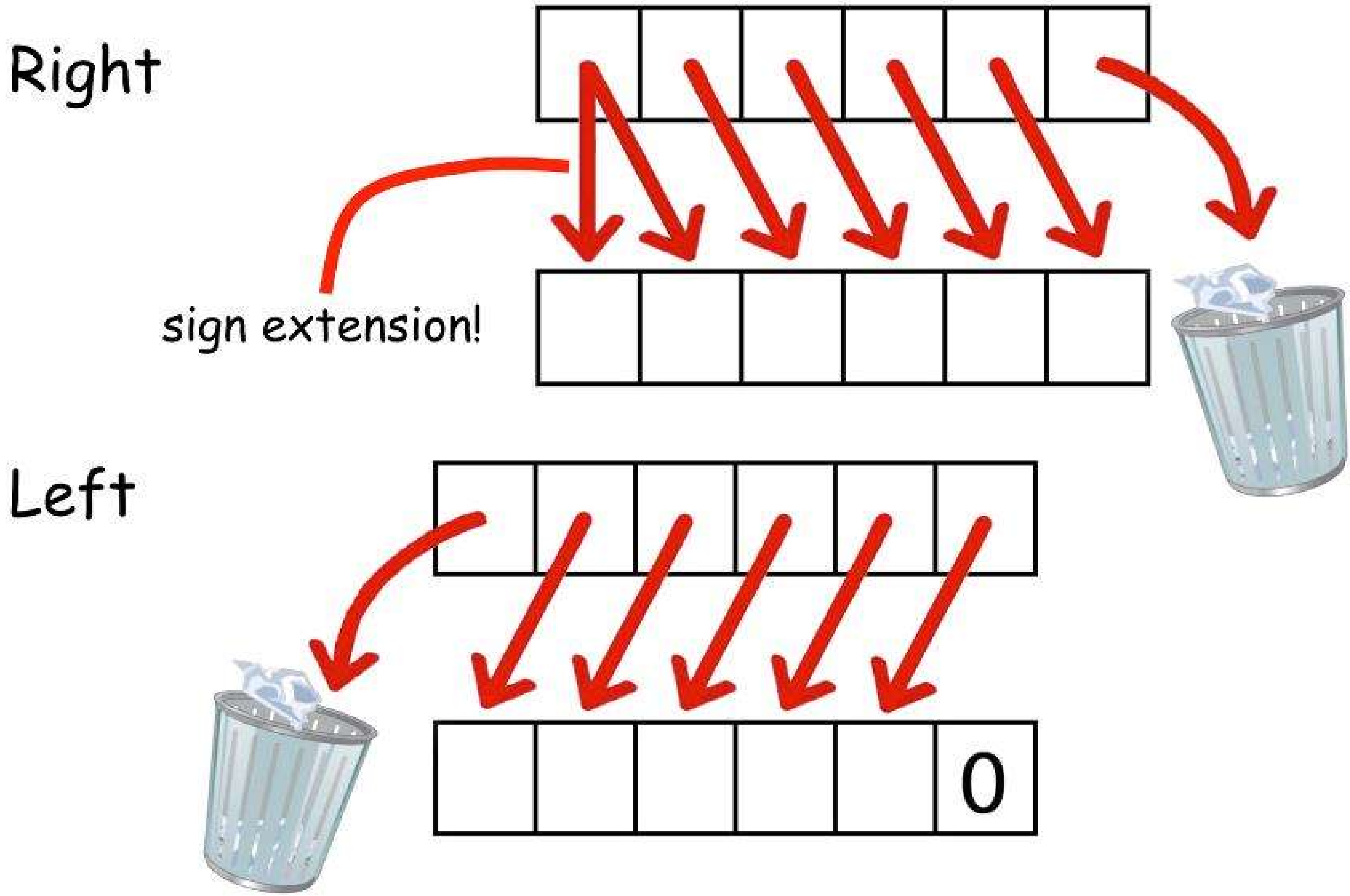
```
      0 0 1 0 1 (5 bits)
0 0 0 0 0 1 0 1 (8 bits)

        1 1 1 1 (4 bits)
0 0 0 0 1 1 1 1 (8 bits)
```

Truth Table for a Few Logical Operations

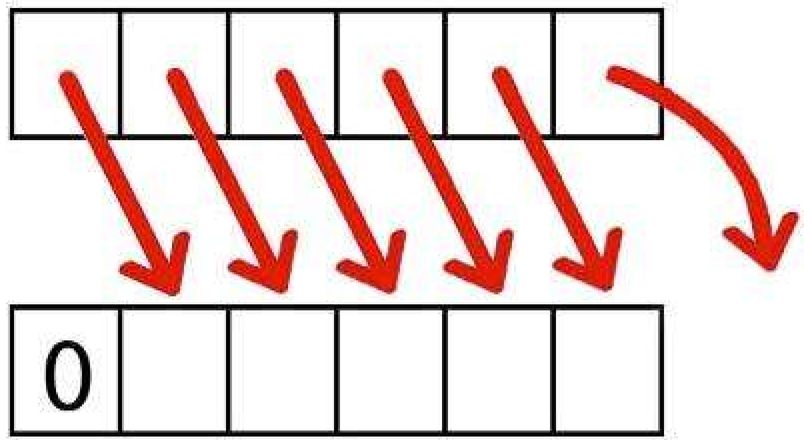
X	Y	X and Y	X nand Y	X or Y	X xor Y
0	0	0	1	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	0

Arithmetic Shift

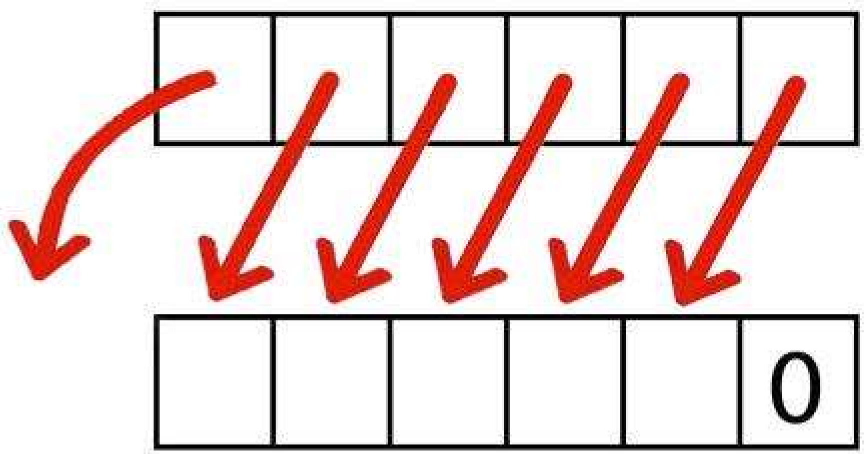


Logical Shift

Right

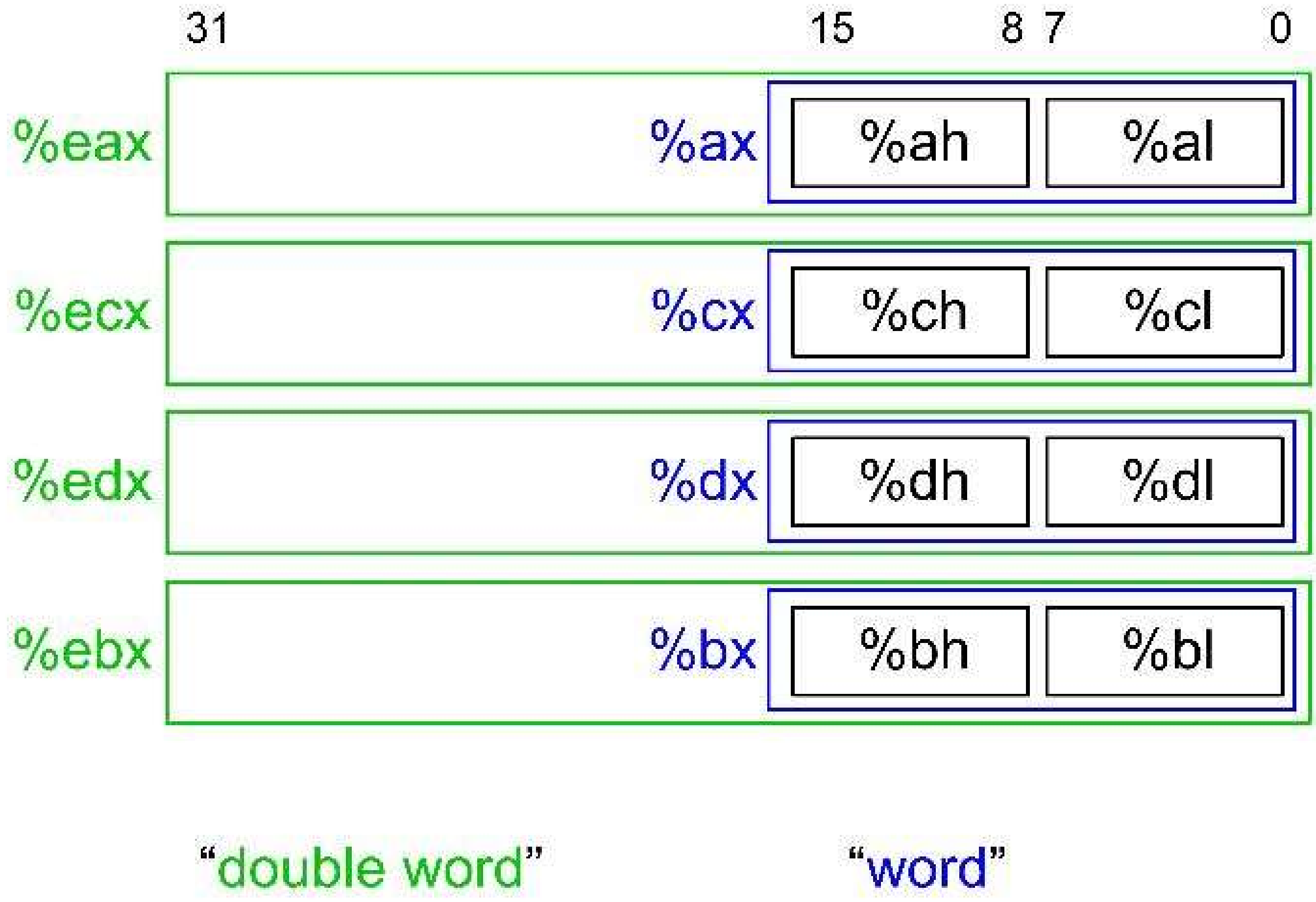


Left

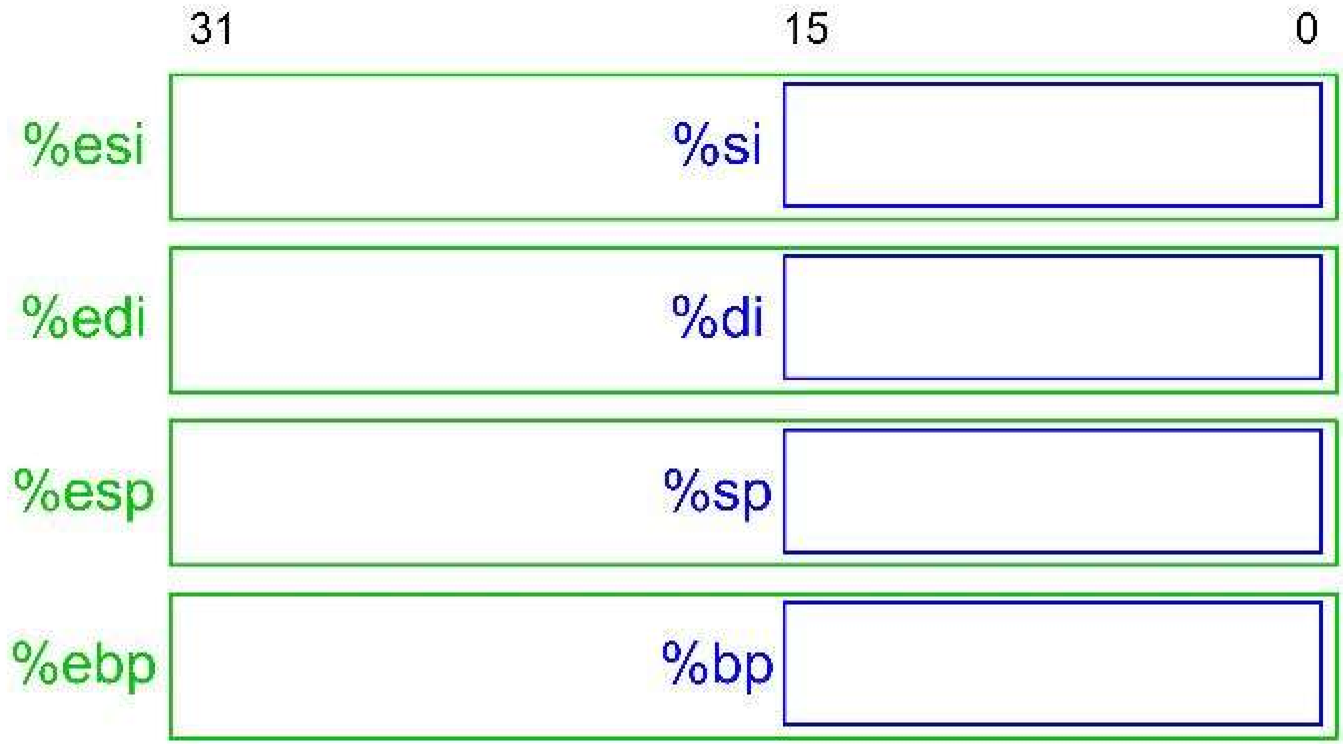


Logical left is the same as arithmetic left.

Registers


31
15
8 7
0
%eax
%ax
%ah
%al
%ecx
%cx
%ch
%cl
%edx
%dx
%dh
%dl
%ebx
%bx
%bh
%bl
"double word"
"word"

4 More Registers



Operands

Syntax	Addressing mode name	Effect
$Imm	immediate	value in machine code
%R	register	value in register R
Imm	absolute	address given by Imm
(%R)	register direct (incorrect in textbook)	address in %R
Imm(%R)	base displacement	address is Imm + %R

Some more operand formats in IA32

(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Cannot do memory to memory transfer with a single instruction

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value
%eax	____
0x104	____
$0x108	____
(%eax)	____
4(%eax)	____
9(%eax,%edx)	____
260(%ecx,%edx)	____
0xFC(,%ecx,4)	____
(%eax,%edx,4)	____

?

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

Data Movement Instructions

movb movw movl	S, D	nondestructive copy of S to D
movsbw movsbl movswl	S, D	sign-extended, nondestructive copy of S to D byte to word byte to double word word to double word
movzbw movzbl movswl	S, D	zero-extended, nondestructive copy of S to D byte to word byte to double word word to double word
pushl	S	push double word S onto *the* stack
popl	D	pop double word off *the* stack into D

pushl and popl

- pushl %ebp is equivalent to:

```
subl $4, %esp
movl %ebp, (%esp)
```

- popl %eax is equivalent to:

```
movl (%esp), %eax
addl $4, %esp
```

Arithmetic Instructions

`leal`	S, D	(load effective address) D gets the address defined by S
`inc`	D	D gets D + 1 (two's complement)
`dec`	D	D gets D - 1 (two's complement)
`neg`	D	D gets -D (two's complement additive inverse)
`add`	S, D	D gets D + S (two's complement)
`sub`	S, D	D gets D - S (two's complement)
`imul`	S, D	D gets D * S (two's complement integer multiplication)

More Arithmetic Instructions, with 64 bits of results

`imull`	S	%edx||%eax gets 64-bit *two's complement* product of S * %eax
`mull`	S	%edx||%eax gets 64-bit *unsigned* product of S * %eax
`idivl`	S	*two's complement* division of %edx||%eax / S; %edx gets remainder, and %eax gets quotient
`divl`	S	*unsigned* division of %edx||%eax / S; %edx gets remainder, and %eax gets quotient

Notice *implied* use of %eax and %edx.

`leal` is commonly used to calculate addresses. Examples:

```
leal 8(%eax), %edx
```

- 8 + contents of eax goes into edx
- used for pointer arithmetic in C
- very convenient for acquiring the address of an array element

```
leal (%eax, %ecx, 4), %edx
```

- contents of eax + 4 * contents of ecx goes into edx
- *even more convenient* for addresses of array elements, where eax has base address, ecx has the index, and each element is 4 bytes

Examples

Assume %eax is x and %ecx is y
and %edx=10, address 10 has value 100

1. leal 6(%eax), %edx :: ?
2. leal 9(%eax,%ecx,2), %edx :: ?
3. addl %ecx, (%edx) :: ?
4. decl %ecx :: ?

Examples

Assume %eax is x and %ecx is y
and %edx=10, address 10 has value 100

1. leal 6(%eax), %edx :: 6+x
2. leal 9(%eax,%ecx,2), %edx :: 9 + x + 2y
3. addl %ecx, (%edx) :: (y +100) stored @ address 10
4. decl %ecx :: (y-1) stored in %ecx

Examples

Assume x at %ebp+8, y at %ebp+12, z at %ebp+16

1 movl 16(%ebp), %eax	*z*
2 leal (%eax,%eax,2), %eax	*z*3*
3 sall $4, %eax	*t2 = z*48*
4 movl 12(%ebp), %edx	*y*
5 addl 8(%ebp), %edx	*t1 = x+y*
6 andl $65535, %edx	*t3 = t1&0xFFFF*
7 imull %edx, %eax	*t4 = t2*t3*

Logical and Shift Instructions

`not`	D	D gets ~D (complement)
`and`	S, D	D gets D & S (bitwise logical AND)
`or`	S, D	D gets D | S (bitwise logical OR)
`xor`	S, D	D gets D ^ S (bitwise logical XOR)
`sal` `shl`	k, D	D gets D logically left shifted by k bits
`sar`	k, D	D gets D arithmetically right shifted by k bits
`shr`	k, D	D gets D logically right shifted by k bits

Examples

Assume x at %ebp+8, y at %ebp+12, z at %ebp+16

1 movl 12(%ebp), %eax	y
2 xorl 8(%ebp), %eax	t1 = x ^ y
3 sarl $3, %eax	t2 = t1 >> 3
4 notl %eax	t3 = ~t2
5 subl 16(%ebp), %eax	t4 = t3-z

Condition Codes

a register known as EFLAGS on x86

CF: **carry flag**. Set if the most recent operation caused a carry out of the msb. Overflow for unsigned addition.

ZF: **zero flag**. Set if the most recent operation generated a result of the value 0.

SF: **sign flag**. Set if the most recent operation generated a result that is negative.

OF: **overflow flag**. Set if the most recent operation caused 2's complement overflow.

Instructions related to EFLAGS

sete setz	D	set D to 0x01 if ZF is set, 0x00 if not set (place zero extended ZF into D)
sets	D	set D to 0x01 if SF is set, 0x00 if not set (place zero extended SF into D)
		. . . many more set instructions . . .
cmpb cmpw cmpl	S2, S1	do S1 - S2 to set EFLAGS
testb testw testl	S2, S1	do S1 & S2 to set EFLAGS

Control Instructions

`jmp`	`label`	goto `label`; %eip gets `label`
`jmp`	`*D`	indirect jump; goto address given by `D`
`je` `jz`	`label`	goto `label` if ZF flag is set; jump taken when previous result was 0
`jne` `jnz`	`label`	goto `label` if ZF flag is *not* set; jump taken when previous result was *not* 0
`js`	`label`	goto `label` if SF flag is set; jump taken when previous result was negative
`jns`	`label`	goto `label` if SF flag is *not* set; jump taken when previous result was *not* negative

More Control Instructions

`jg` `jnle`	`label`	goto `label` if EFLAGS set such that previous result was greater than 0
`jge` `jnl`	`label`	goto `label` if EFLAGS set such that previous result was greater than or equal to 0
`jl` `jnge`	`label`	goto `label` if EFLAGS set such that previous result was less than 0
`jle` `jng`	`label`	goto `label` if EFLAGS set such that previous result was less than or equal to 0

“if” and “if else” Stmts in Assembly

Overview of “if” and “if else” statement:

if(condition){
    statements;
}
	if(condition){
	    statements1;
	}else{
	    statements2;
	}

General Approach:

1. Use compare instructions to set the condition codes
2. Then use the jump instructions to execute the right set of instructions

"if else" example

```c
if(x<y){
    return y-x;
}else{
    return x-y;
}
```

x at %ebp+*8, y at* %ebp+*12*

```
movl 8(%ebp), %edx        Get x
movl 12(%ebp), %eax       Get y
cmpl %eax, %edx           Compare x:y
jge .L2                   if >= go to L2
subl %edx, %eax           result = y-x
jmp .L3                   Goto done
.L2:
subl %eax, %edx           result = x-y
movl %edx, %eax           %eax  = result
.L3: done: Begin completion code
```

"while" example

```
result = 1;
while(n>1){
   result*=n;
   n = n-1;
};
```

Argument: n at %ebp+*8*
Registers: n in %edx, *result in* %eax

```
movl 8(%ebp), %edx      get n
movl $1, %eax           result = 1
cmpl $1, %edx           compare n:1
jle .L7                 If <=, goto done
.L10:                 loop:
imull %edx, %eax        result *= n
subl $1, %edx           decrement n
cmpl $1, %edx           compare n:1
jg .L10                 If >, goto loop
.L7:                  done:
```

Return result

FOR LOOP EXAMPLE

$$\sum_{i=1}^{N} i$$

```
sum = 0;
for (i = 1; i <= N; i++) {
    sum = sum + i;
}
```

gcc's implementation (mostly):

```
        movl   N, %ecx
        movl   $0, %eax        sum in eax
        movl   $1, %edx        i in edx
        jmp    .L2
.L3:    addl   %edx, %eax      sum = sum + i
        incl   %edx
.L2:    cmpl   %ecx, %edx
        jle     .L3          jump when i-N is less than
                              or equal to 0
```

Conditional Move Instructions

Instruction		Synonym	Move condition	Description
cmove	*S*, *R*	cmovz	ZF	Equal / zero
cmovne	*S*, *R*	cmovnz	~ZF	Not equal / not zero
cmovs	*S*, *R*		SF	Negative
cmovns	*S*, *R*		~SF	Nonnegative
cmovg	*S*, *R*	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	*S*, *R*	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	*S*, *R*	cmovnge	SF ^ OF	Less (signed <)
cmovle	*S*, *R*	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	*S*, *R*	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	*S*, *R*	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	*S*, *R*	cmovnae	CF	Below (unsigned <)
cmovbe	*S*, *R*	cmovna	CF | ZF	below or equal (unsigned <=)

Figure 3.17 **The conditional move instructions.** These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.