Digital microsystems design

Marius Marcu

2020

Objectives

- Specific objectives
  - Instruction set architecture of x86 processors and assembly language

Outline

- Data types
- Memory Variables
- Constants
- Pointers

Data types

- Numeric data
  - Integer
  - Real
- Boolean data
  - TRUE/FALSE
- Characters data and strings
  - ASCII, UNICODE
- Address values
  - Reference, pointer

Data types

- Type – a specified set of values
- Data type – the set of values together with the operations allowed to be executed on the values

Data types

- Example:
  - Byte type
    - Values between 0 and 255
  - Byte data type
    - Values between 0 and 255
    - Operators +, -, *, /, %

Data types

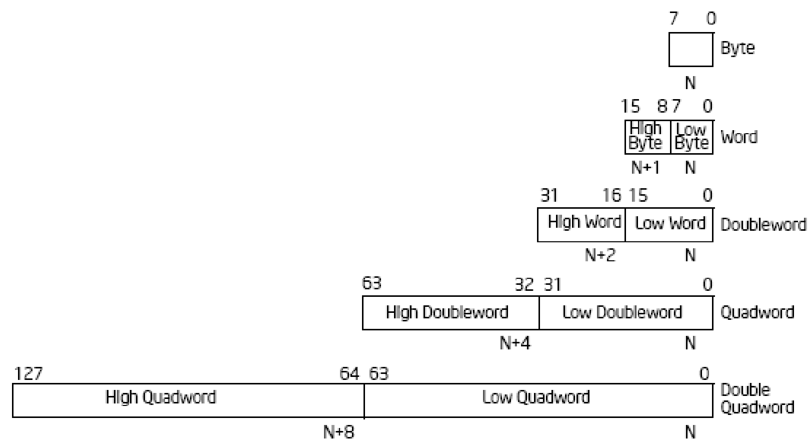
- In what ways data types are different each other?

Data types

- Primary data types attributes:
  - Size
    - Number of bytes needed to store any value of the type
  - Encoding
    - Binary encoding of values of the type
      - Signed vs. unsigned
      - Integer vs. floating point vs. BCD
      - ASCII vs. UNICODE
  - Operations
    - Instructions which have operands the values of the type

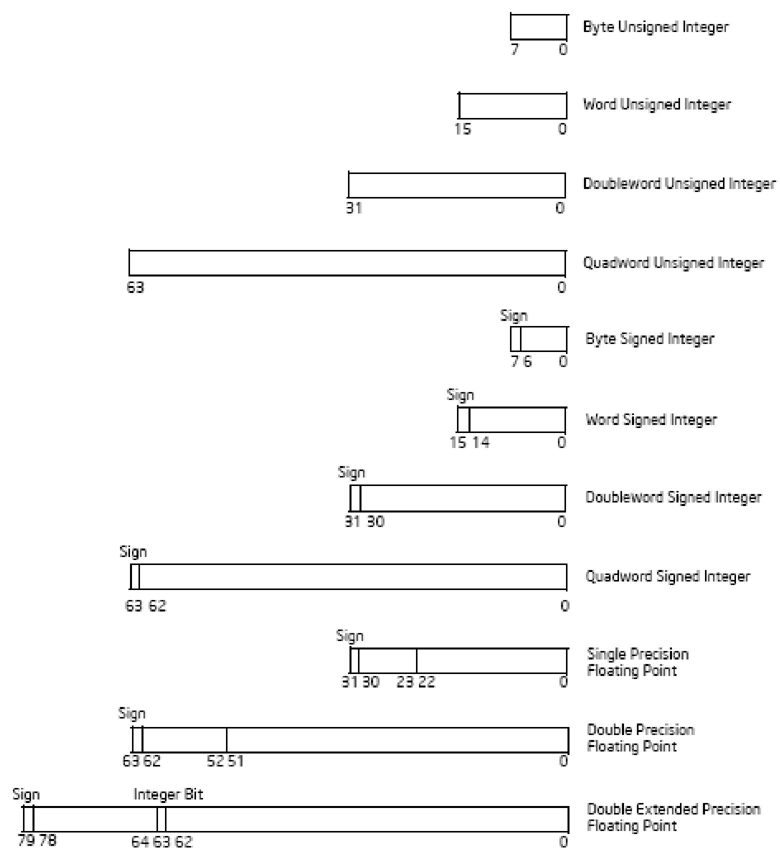
Data types

- Primary data types provided by IA-32
  - Byte
  - Word
  - Doubleword
  - Quadword
  - Tenbyte
  - Double quadword



Data types

- Instruction specific
  – Unsigned integer
  – Signed integer
  – Floating point



Data types

- Assembly primary data types

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Data types

- Size – range of values
  - unsigned int, unsigned long (32 bits)
    - [0, 2^32-1]
  - signed int, long (32 bits)
    - [-2^31, 2^31-1]
  - unsigned long long (64 bits)
    - [0, 2^64-1]
  - long long (64 bits)
    - [-2^63, 2^63-1]

Data types

- Encoding – single precision floating point
  - Standard: IEEE 754
  - Size: 32 bits
    - 1 sign bit (s)
    - 8 exponent bits (e), biased by 127
    - 23 mantissa bits (m), prefixed by 1, omitted
  - Number: s*m*2^e
    - (1-2*s)*(1+m)*2^(e-bias)

Data types

- Encoding – single precision floating point
  - Example:
    - `0 10000001 01001100110011001100110`
    - `s     e              m`
  - Exponent is biased by 127
    - e = 129 – 127 = 2
  - Mantissa is prefixed by 1.
    - m = 1.01001100110011001100110
  - 5.2 = (1+1/4+1/32+1/64+1/512+1/1024+…)*2^2
    - 5.2 = 101 + 0.001100110011…
    - 0.2 = 1/5 = 0.33333…h

Data types

- Encoding – single precision floating point
  - Example:
    - 1 01111100 11000000000000000000000
    - e = 124-127 = -3
    - m = 1.11000…b = 1 + 0.5 + 0.25 = 1.75
    - (-1)*1.75*2^(-3) = -0.21875

Memory variables

- The compiler reserves memory space for program variables

```
int a = 0; ->    a dd 0        ; 32 bits integer
```

– Use integer instructions to operate with integers

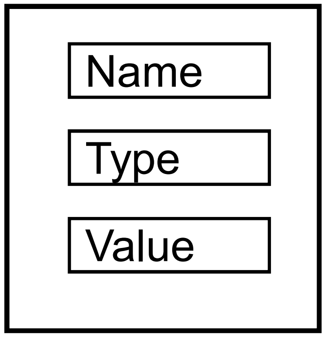
db, dw, dd, dq, dt

```
float f = 0.5; -> f dd 0.5f   ; single precision float
```

– Use FPU instruction set to operate with floating point numbers

Memory variables

- A program variable is an entity stored by the main memory whose values can be changed during the program execution
- A variable is a group of memory locations that will store the current value of the variable during program execution
- A variable is characterized by
  - Name
  - Type
  - Value



Memory variables

- The variable is characterized by:
  - *name (identifier):* associated to the memory address of the locations.
  - *value*: the information stored by memory locations
  - *type*: specifies the set of values the variable can take, their encoding, and the operations allowed to be executed upon the variable.
- Value encoding is specific to the data type used to define the variable
- Low level operations (instructions) are selected by the compiler according to the encoding

Memory variables

- Example

```
unsigned int a = 10, b=25, c;
float x=0.10, y=0.25, z;

c = a+b;     -> unsigned int ADD instruction
z = x+y;     -> floating point ADD instruction


c = c+z;     -> conversion needed
z = z+c;     -> conversion needed
```

Memory variables

- Example – global definition

```
unsigned int a = 10, b=0, c;
float x=10.0, y=5.2, z;
```

```
.data
a:      0a 00 00 00
x:      00 00 20 41
y:      66 66 a6 40
.bss
b:      00 00 00 00
c:      00 00 00 00
z:      00 00 00 00
```

Memory variables

- Example – local definition

```
unsigned int a = 10, b=0, c;
float x=10.0, y=5.2, z;
```

```
.stack
z:    ?? ?? ?? ??
y:    66 66 a6 40
x:    00 00 20 41
c:    ?? ?? ?? ??
b:    00 00 00 00
a:    0a 00 00 00
```

Memory variables

- Example

```
unsigned int a = 10, b=25, c;
float x=0.10, y=0.25, z;

c = a+b;     -> unsigned int ADD instruction

mov eax, a
add eax, b
mov c, eax
```

```
    c = a + b;
00F05292  mov          eax,dword ptr [a]
00F05295  add          eax,dword ptr [b]
00F05298  mov          dword ptr [c],eax
```

Memory variables

- Example

```
unsigned int a = 10, b=25, c;
float x=0.10, y=0.25, z;

z = x+y;     -> floating point ADD instruction
```

FPU

```
FLD x
FADD y
FST z
```

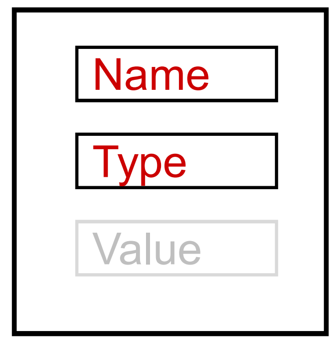
SSE

```
    z = x + y;
007F529B  movss       xmm0,dword ptr [x]
007F52A0  addss       xmm0,dword ptr [y]
007F52A5  movss       dword ptr [z],xmm0
```

Memory variables

- Variable declaration
  - Specifying a name and a data type
  - No memory allocation for the value
- Example

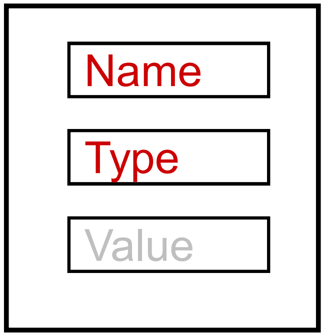
```
external int a;
```



Memory variables

- Variable definition
  - Memory allocation for the value
  - Memory location can be initialized or it may start uninitialized
- Example

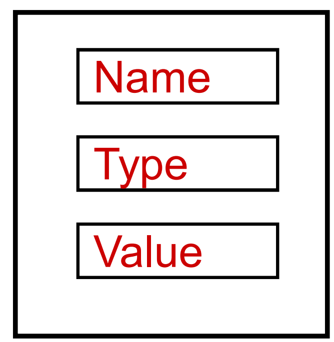
```
int a = 0;
int b;
```



Memory variables

- Variable initialization
  - Specify a value for the variable if the variable is already defined
  - Value assignment
- Example

```
b = 10;
```



Memory variables

- Defining variables in assembly language

```
    name_var    type  list_of_values
    name_str    type  N dup(value)

type: db, dw, dd, dq, dt

    var_int8    db    12H
    var_int16   dw    1234H
    var_int32   dd    12341234H
    var_int64   dq    1234123412341234H
```

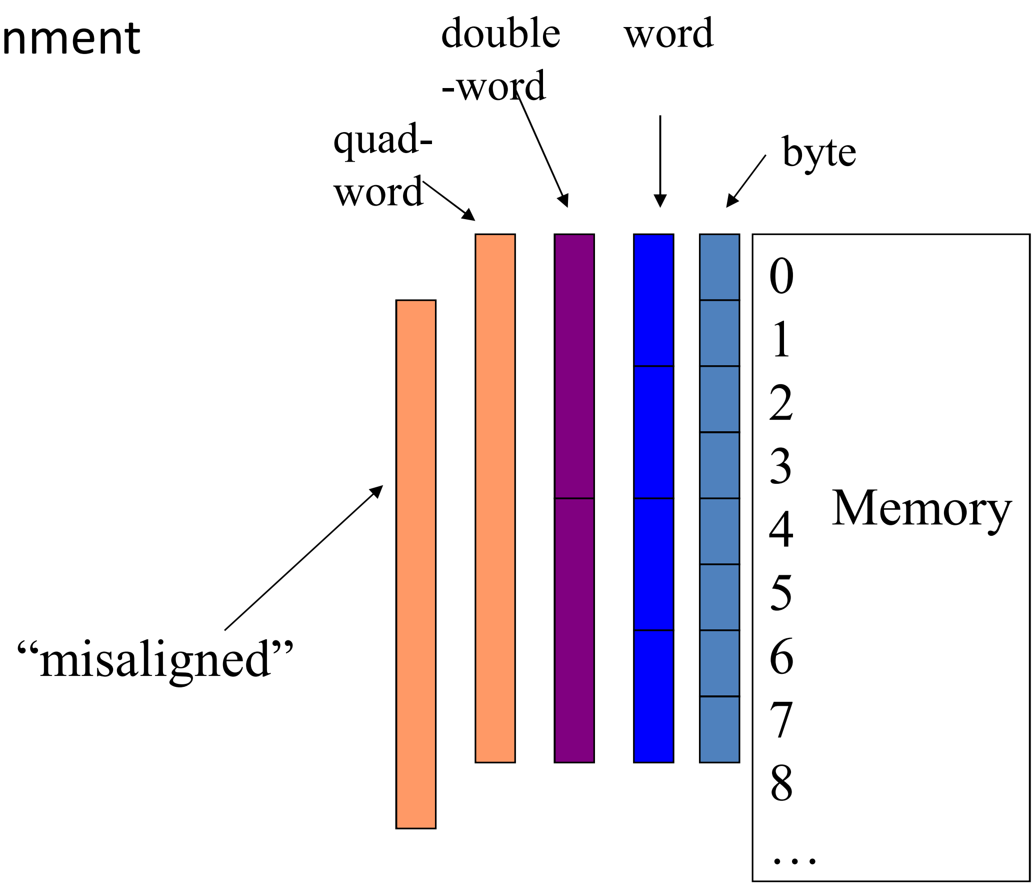
Memory variables

- Defining variables in assembly language

```
var_char    db    ‘A’
var_float   dq    0.123f

str_int8    db    1, 2, 3, 4, 5
str_int16   dw    1, 2, 3, 4, 5
str_int32   dd    5 dup(?)

var_str1    db    ‘ASCII’
var_str2    db    1000 dub(0)
var_str3    dw    L‘UNICODE’
```

Memory variables

- Memory variable alignment

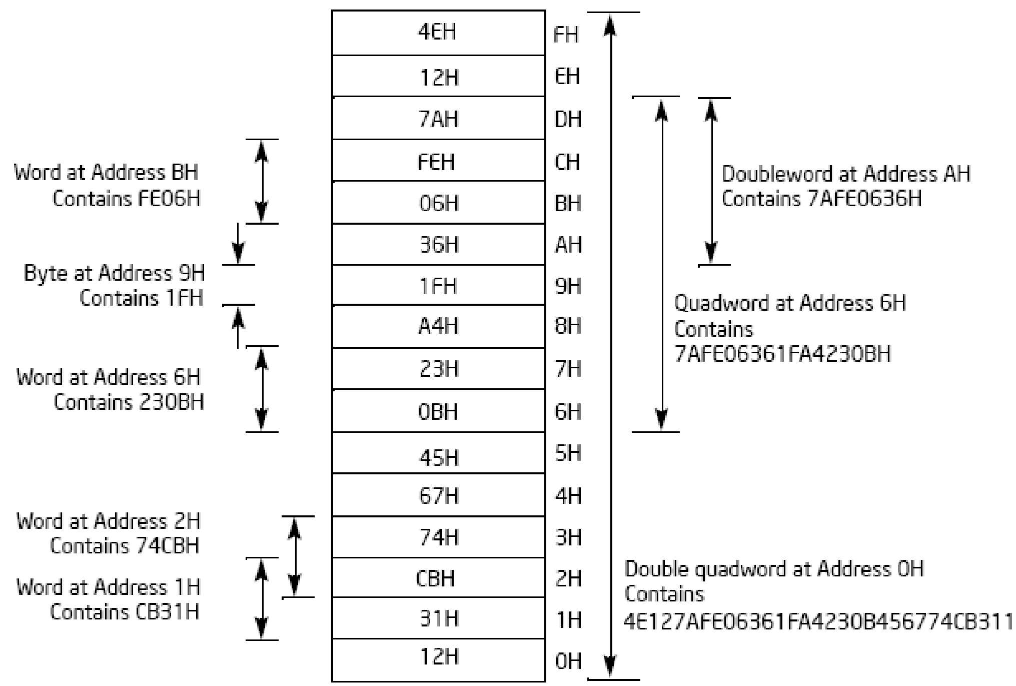


Memory variables

- Alignment
  - Some processors require aligned memory accesses only (RISC)
    - Memory variables should be located at some native boundaries (e.g. 4 or 8 bytes)
    - They will not be able to access data at arbitrary addresses
  - CISC processors do not require aligned memory accesses
    - They are able to access data at arbitrary addresses
    - However, using aligned addresses are more efficient than non-aligned addresses
    - Some data structures require aligned addresses

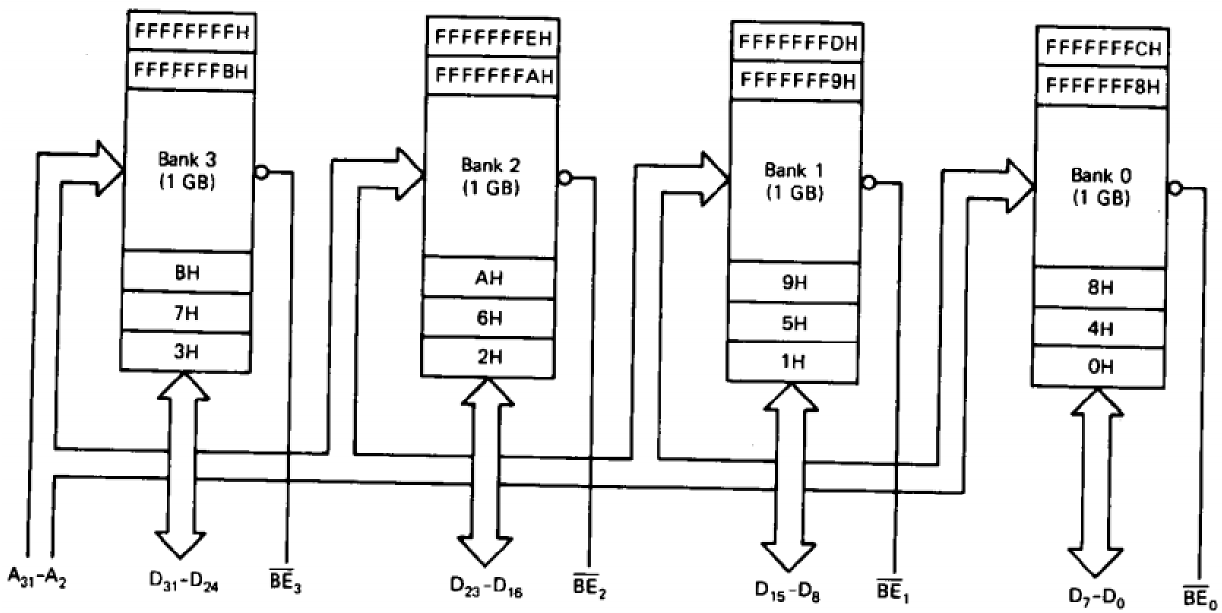
Memory variables

- Data alignment



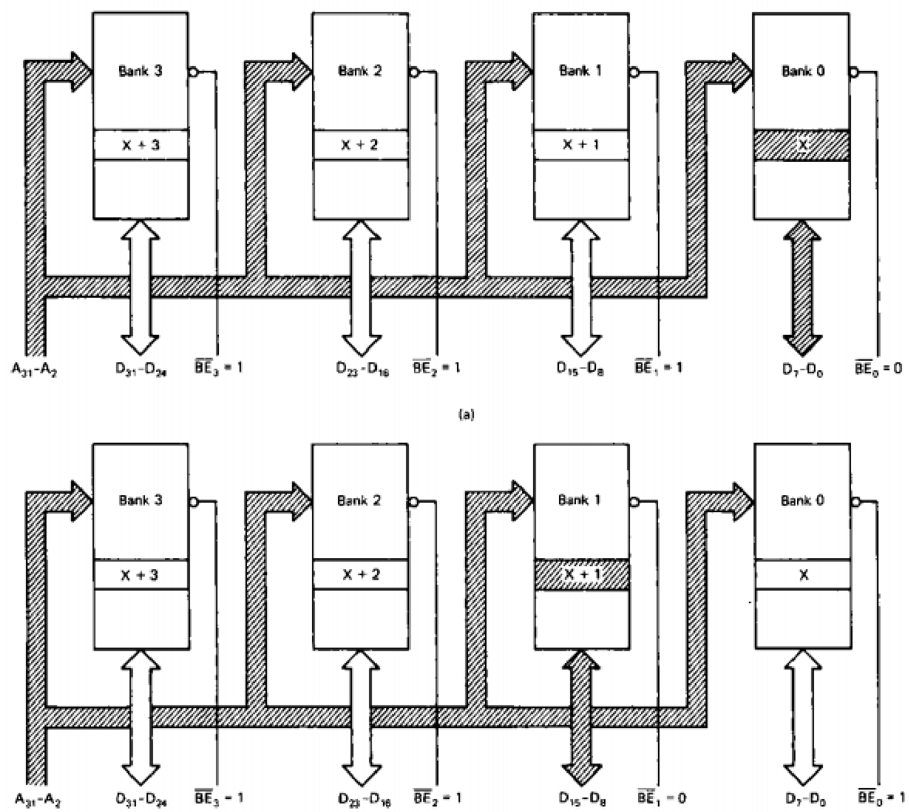
Data transfer

- Hardware organization of physical address space (32 bits)
  - 32 bits memory block



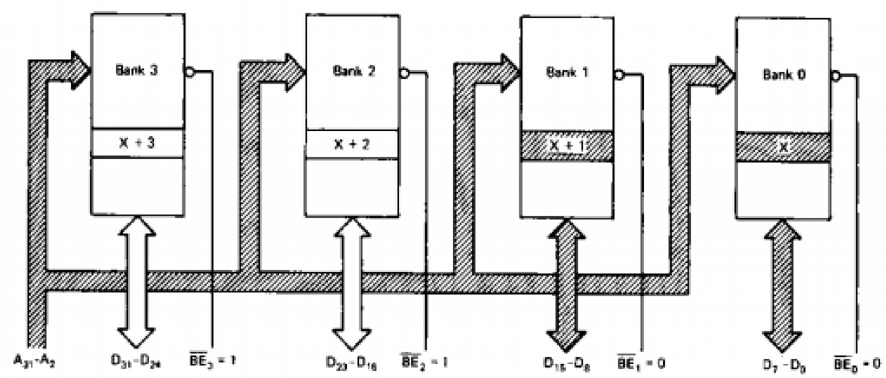
Data transfer

- 1 byte
  - Address 4xk
  - Address 4xk+1



Data transfer

- 1 word
  - Address 4xk

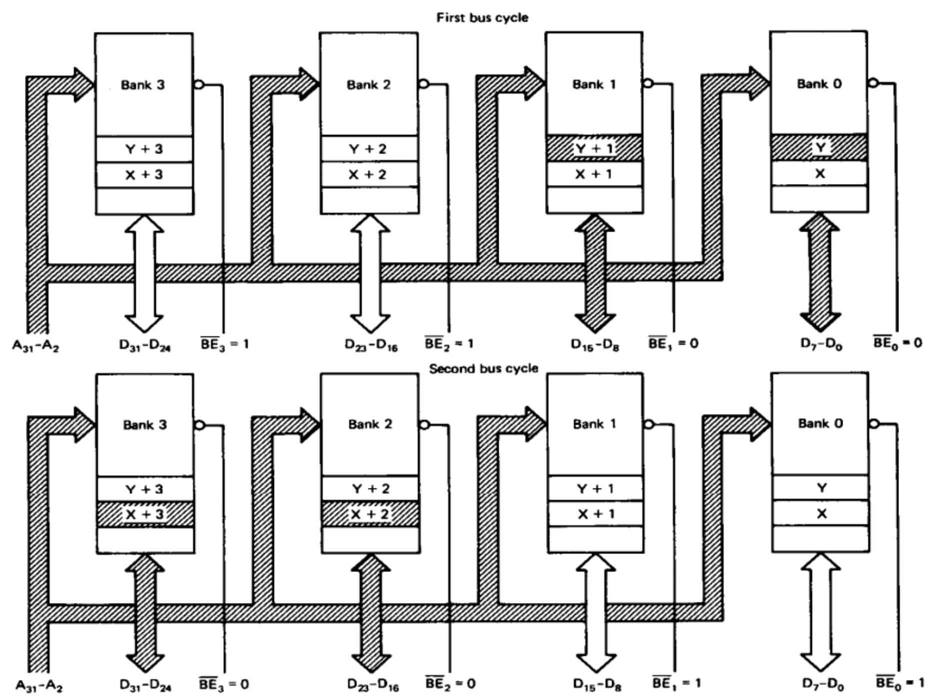


(c)

- 1 double word
  - Address 4xk

Data transfer

- Misaligned double-word data transfer
  - Address 4xk+2 – the processor will perform 2 bus cycles

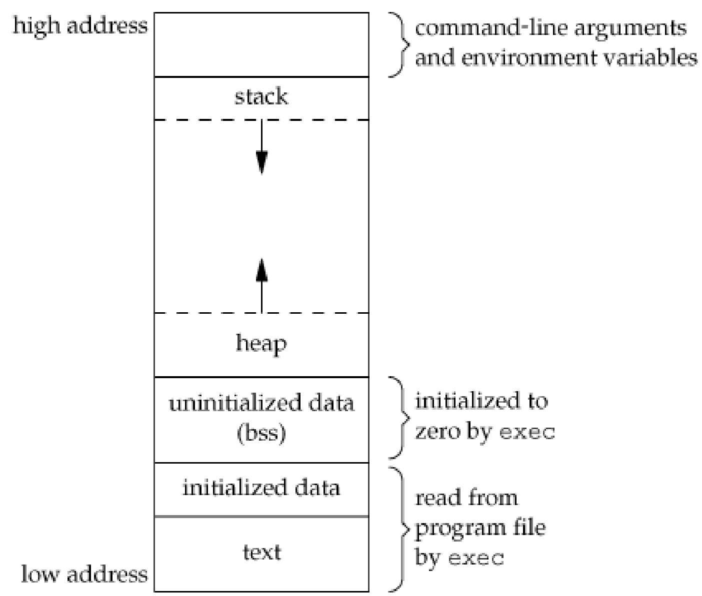


Memory variables

- Where exactly in memory the variables are allocated?
  - Static
    - Global variables – data segment
    - Local variables – stack
  - Dynamic
    - Pointers – heap
      - Dedicated memory region managed by the standard libraries or OS (kernel variables)

Memory sections

- .text – code section
- .data – data section (initialized)
- .bss – data section (uninitialized)
  - Static variables
  - Variables initialized to 0



Memory sections

- .stack
  - Local variables
  - Function arguments
  - Return addresses
  - Register save and restore
  - Stack segment

Memory sections

- .text – contains the instruction codes of the program (read only and execution permissions)
- .data – contains the initialized global and static variables and their values (read/write permissions)
- .rdata – contains constants and string literals (const in C)
- .bss – contains uninitialized global and static variables (it does not take any actual space in the object file)

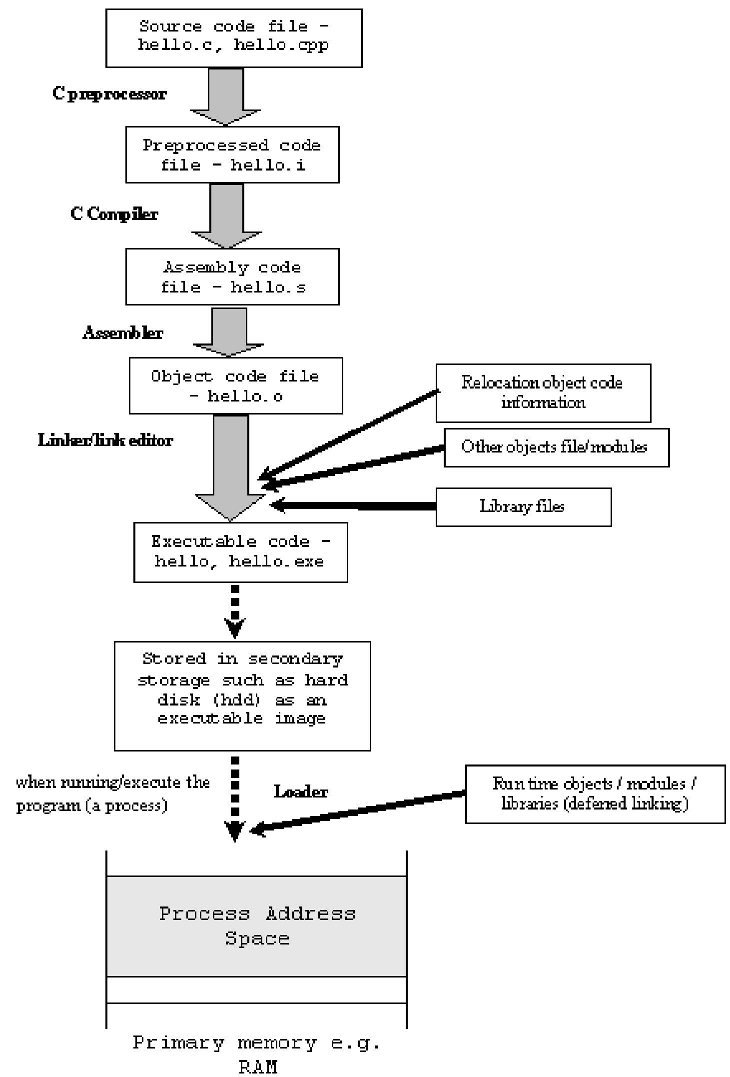
- Symbol table – contains information needed to locate and relocate program's symbols
  - Symbol – name and address
- Relocation records – mapping process between symbol references and their definitions

Memory sections

- How to transfer memory and code sections from compiler and linker to program execution?
- Executable file formats:
  - PE – Portable Executable (.exe)
  - ELF – Executable and Linkable format (.elf)

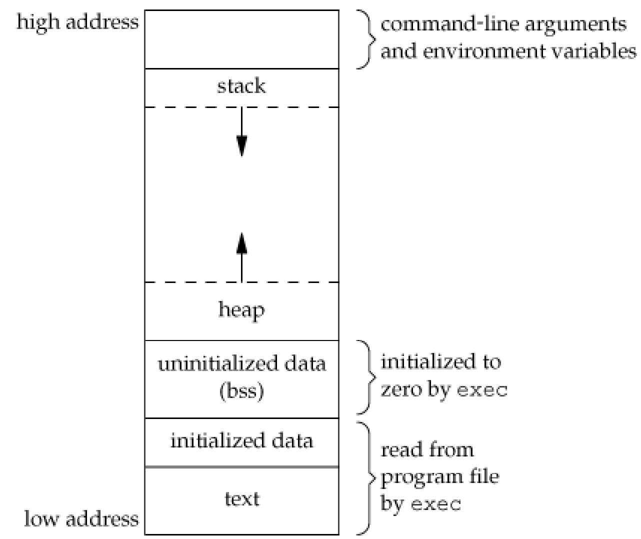
```
            0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  0123456789ABCDEF
0x00000000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ..............
0x00000010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@....... +10
0x00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................ +20
0x00000030  00 00 00 00 00 00 00 00  00 00 00 00 E0 00 00 00  ................ +30
0x00000040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ........!..L.!Th +40
0x00000050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno +50
0x00000060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS  +60
0x00000070  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....$....... +70
0x00000080  83 68 41 C9 C7 09 2F 9A  C7 09 2F 9A C7 09 2F 9A  .hA.../.../.../. +80
0x00000090  CA 5B F2 9A C5 09 2F 9A  CA 5B CF 9A D4 09 2F 9A  .[..../..[..../. +90
0x000000A0  CA 5B CE 9A C0 09 2F 9A  1A F6 E4 9A C4 09 2F 9A  .[..../......./. +A0
0x000000B0  C7 09 2E 9A FD 09 2F 9A  BA 70 CE 9A C5 09 2F 9A  ....../..p..../. +B0
0x000000C0  CA 5B F4 9A C6 09 2F 9A  BA 70 F1 9A C6 09 2F 9A  .[..../..p..../. +C0
0x000000D0  52 69 63 68 C7 09 2F 9A  00 00 00 00 00 00 00 00  Rich../......... +D0
0x000000E0  50 45 00 00 4C 01 07 00  D1 CE E4 59 00 00 00 00  PE..L......Y.... +E0
0x000000F0  00 00 00 00 E0 00 02 01  0B 01 0C 00 00 40 00 00  .............@.. +F0
0x00000100  00 40 00 00 00 00 00 00  78 10 01 00 00 10 00 00  .@......x....... +100
0x00000110  00 10 00 00 00 00 40 00  00 10 00 00 00 02 00 00  ......@......... +110
0x00000120  06 00 00 00 00 00 00 00  06 00 00 00 00 00 00 00  ................ +120
0x00000130  00 C0 01 00 00 04 00 00  00 00 00 00 03 00 40 81  ..............@. +130
0x00000140  00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  ................ +140
0x00000150  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  ................ +150
0x00000160  68 91 01 00 3C 00 00 00  00 A0 01 00 3C 04 00 00  h...<.......<... +160
```

Compiling tools



Memory segments

- Memory sections of programs are mapped to memory segments at execution
  - Text section – code segment
  - Data, bss, heap – data segment
  - Stack section – stack segment

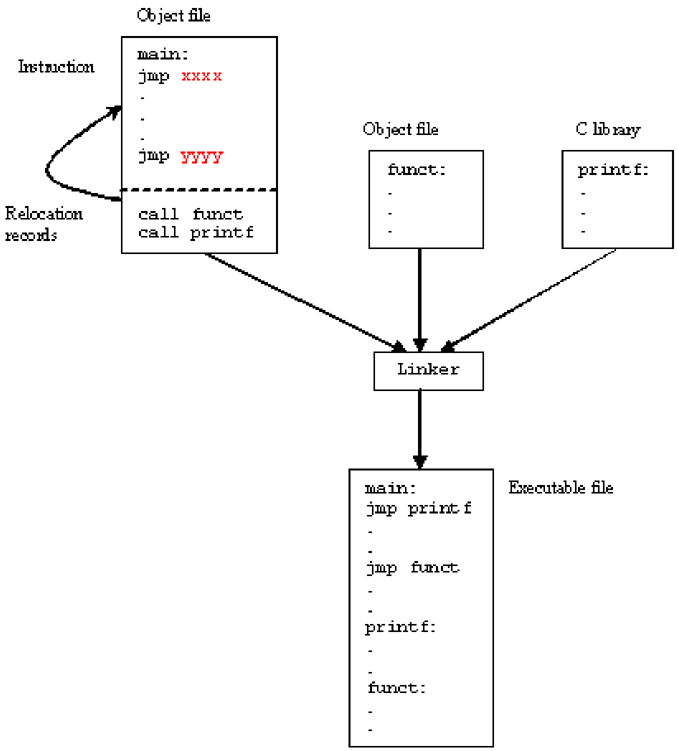


Compiling tools

- Linker
  - Various object files will include references to each others code and/or data
  - These shall need to be combined during the link time
  - After linking all of the object files together, the linker uses the relocation records to find all of the addresses that need to be filled in
  - It is accomplished by the symbol table that contains a list of names and their corresponding offsets in the text and data segments

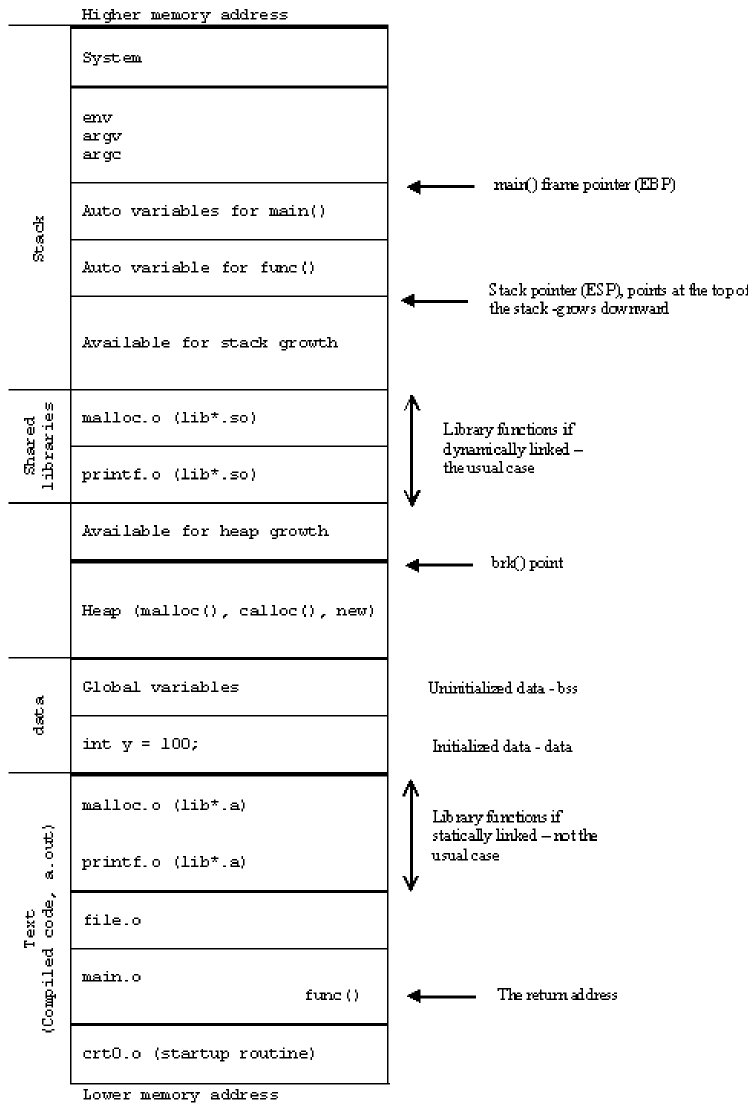
Compiling tools

- Linker
  - Symbol table



Compiling tools

- Memory layout



Compiling tools

- Static linking
  - Program and the particular library that it’s linked against are combined together by the linker at link time
  - The binding between the program and the particular library is fixed and known at link time before the program run
  - The drawback of this technique is that the executable is quite big in size, all the needed information need to be brought together

Compiling tools

- Dynamic linking
  - The program and the particular library it references are not combined together by the linker at link time
  - Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references
  - This means that the binding between the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.

Memory variables

- Summary
  - Variable name – memory address
  - Variable type
    - Memory size for variable
    - Encoding type for values: unsigned integer, complement of 2 integer, float, double
    - Operations used upon the values: unsigned integer, signed integer, floating point
  - Variable value
    - Memory allocation – store the value
      - Static – compiler
      - Dynamic – runtime (pointers)
    - Data alignment

Memory variables

- Exercises
  - How much memory is allocated to store each one of the following variables?
    - Assuming they are placed in contiguous blocks in the order they are defined
  - Which one are not aligned?

```
int a = 10;
unsigned short int b = 20;
unsigned long int c = 3000;
float x = .10;
char ch = ‘A’;
wchar str = L‘Unicode string!’
long array[5] = {0, 1, 2};
```

Constants

- Constants are entities that cannot change their value during program execution
- Implementation
  - Typed constants
  - Preprocessor definitions
- Constants can be characterized by:
  - name
  - type (typed constants only)
  - value

Constants

- Constants definition

```
name_const        equ   value

const_int         equ   10
const_char        equ   'A'
const_hex         equ   0abcdH
```

Constants

- The value of the constant is stored by the instruction code
- Immediate memory addressing mode is used

```
ASM:
      const_int8  equ   10
      var_int8    db    10

      mov   al, const_int8
      mov   al, var_int8
```

Constants

- Constant strings and typed constants are stored in .rdata section
- Preprocessor will replace macros before compiling

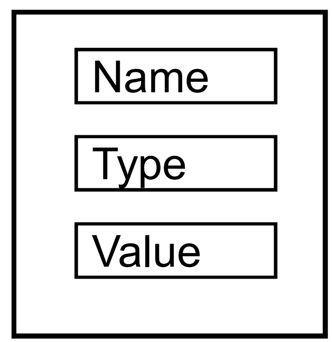
```
C:
   #define VAL      10
   int a;

   char *name = “qwerty”;
   const float pi = 3.14159

   a = VAL;
```

Pointers

- A pointer is a special memory variable whose value is the memory address where the variable is stored

Pointers

- Pointer variables
  - Name
  - Type
    - Size
    - Encoding
    - Operations
  - Value



Pointers

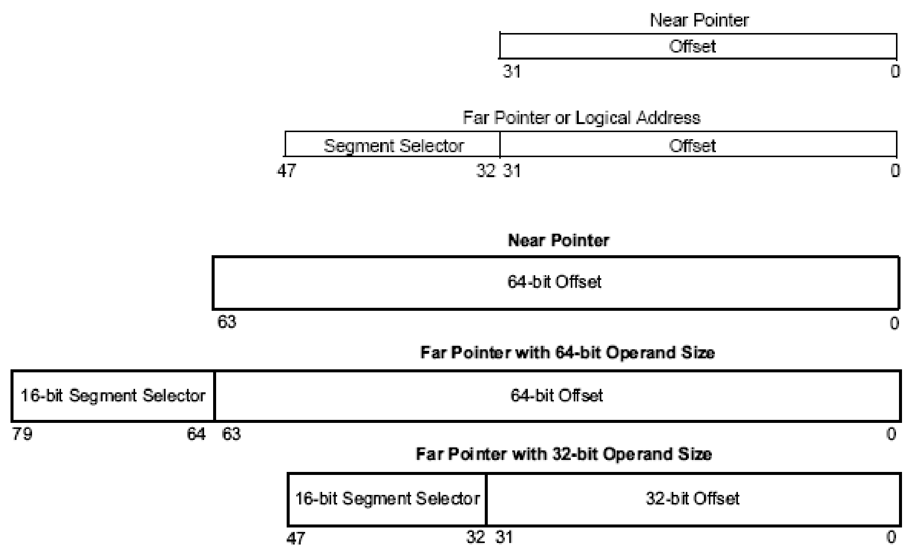
- Pointer variables
  – Size: size of the address (CPU and compiler specific address width)

  – Examples

```
int* p1;
short int* p2 = NULL;
float* p3 = &f;
void* p4 = (void*)p2;
```

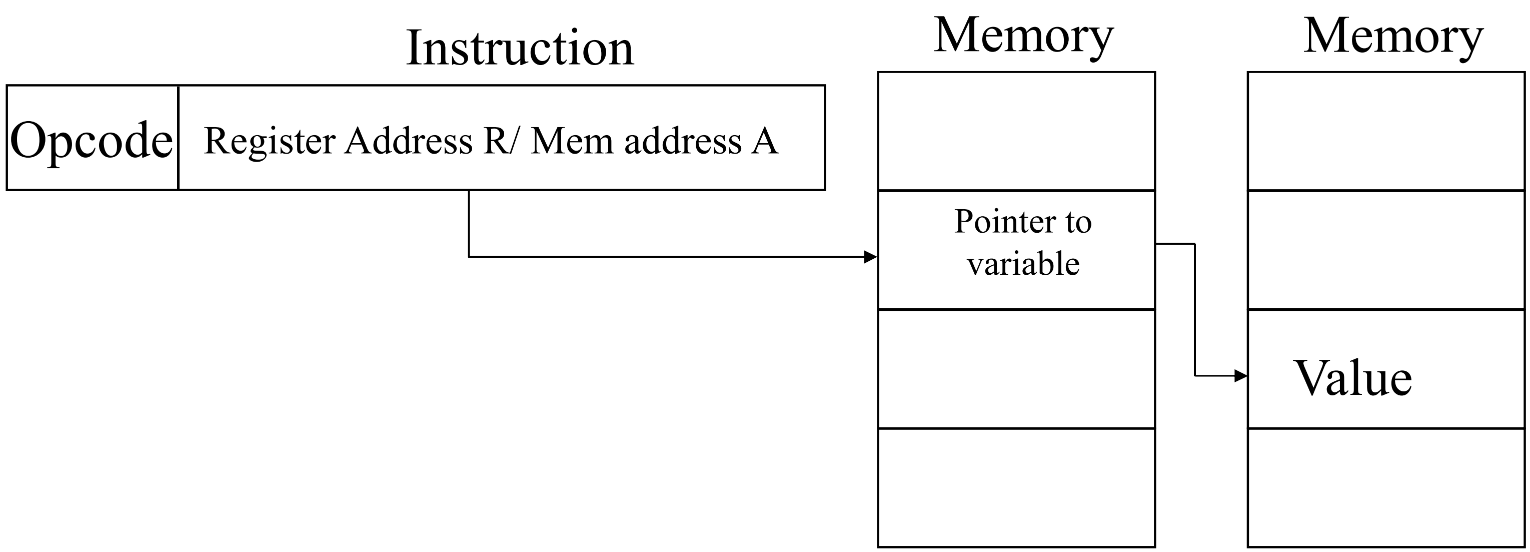
Pointers

- Size of the address
  - Near pointer (offset)
  - Far pointer (segment and offset)



Pointers

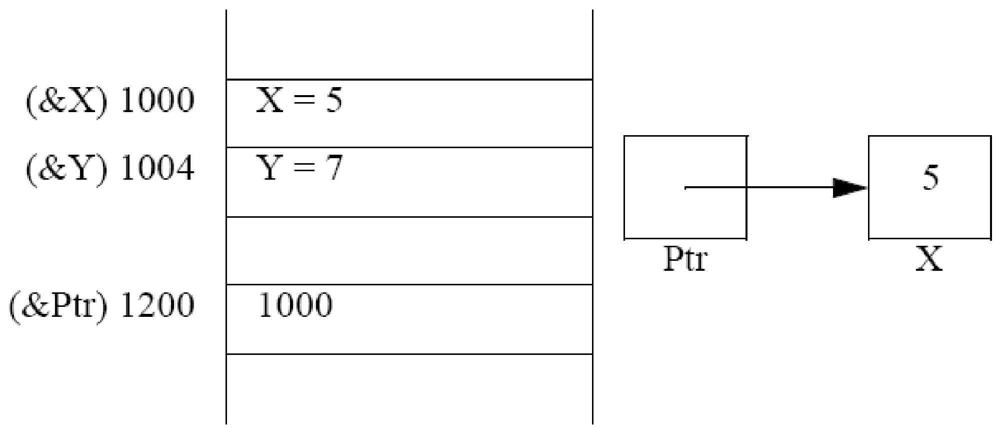
- Indirect memory addressing mode
  - Memory indirect memory addressing



Pointers

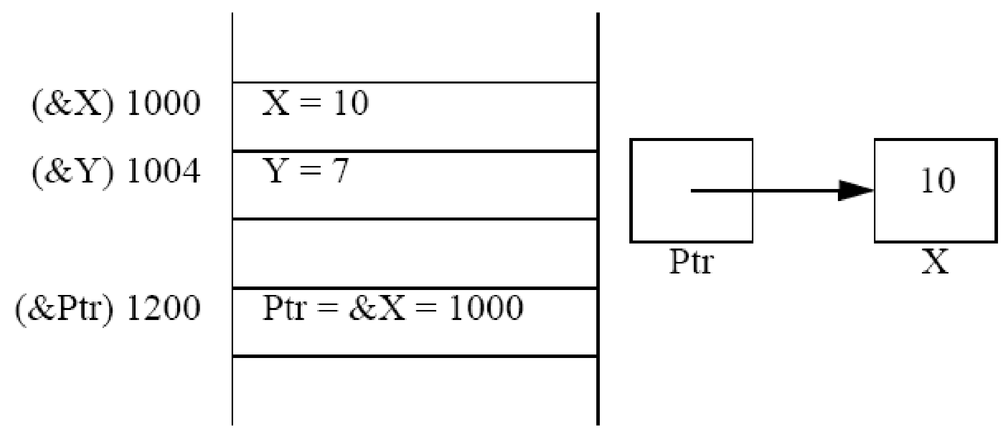
- Example

```
int X = 5;
int Y = 7;
...
int* Ptr = &X;
```



Pointers

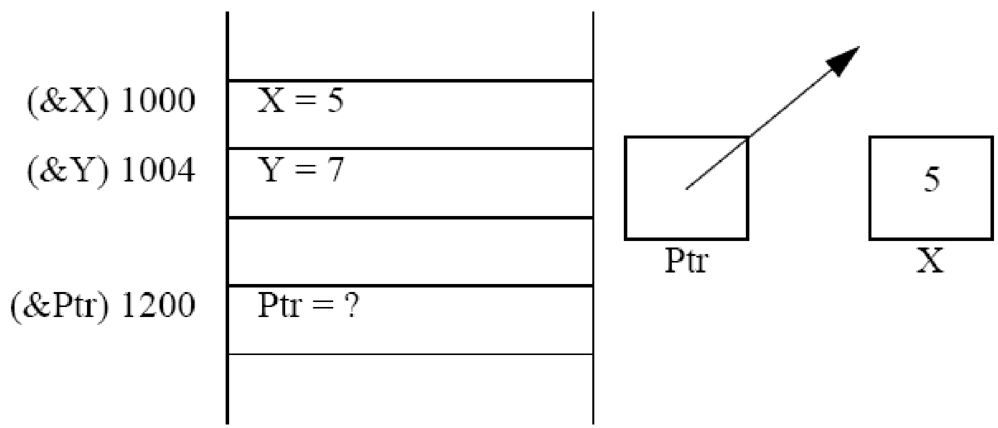
- Example
  - Result of: *Ptr = 10



Pointers

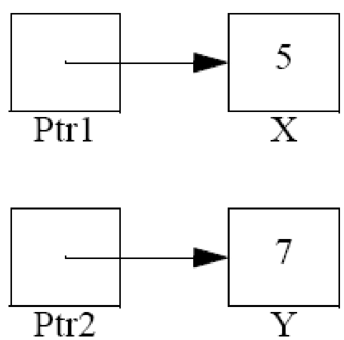
- Uninitialized pointer

```
int *Ptr;
```



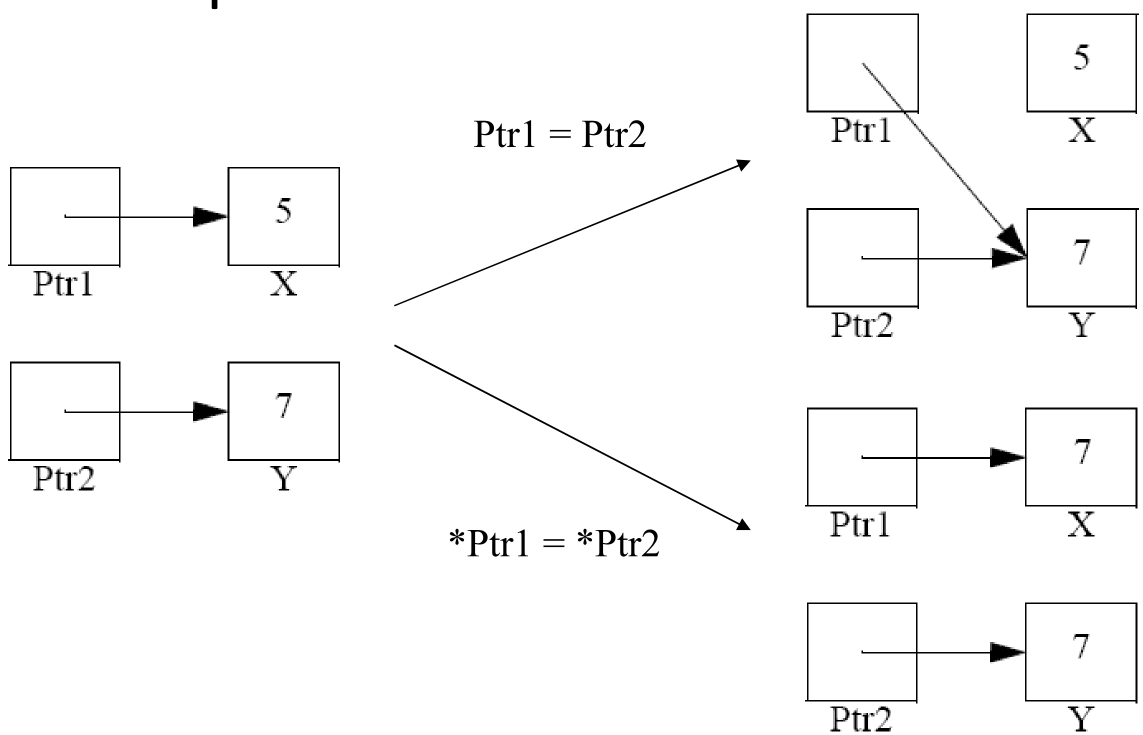
Pointers

- Operations with pointers
  - What happens when Ptr1 = Ptr2 gets executed?
  - What happens when *Ptr1 = *Ptr2 gets executed?



Pointers

- Operations with pointers



Pointers

- A pointer is a variable that stores the logical address of the target location where the actual value is located

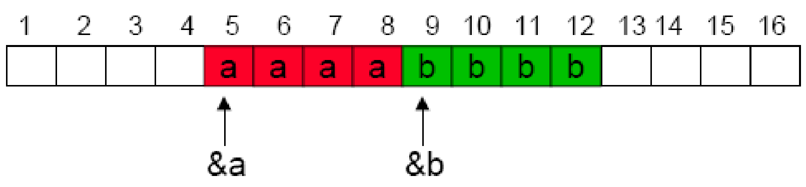
```
int a = 1;
int b = 2;
int *pa = &a;
int *pb;
pb = pa + 1;
*pb = *pa + *pb;
```

What the variables contains after the code is executed?

Pointers

- Pointer arithmetic is recommended only for elements of arrays.
- Variable definition by compiler will use different places in memory and different alignments and paddings – compiler specific

```
int a = 1;
int b = 2;
int *pa = &a;
int *pb;
pb = pa + 1;
*pb = *pa + *pb;
```



Pointers

- Memory allocation:

```
int *pa = new int;        // C++
int *pb = malloc(sizeof(int));
int *pc = new int[10];    // C++
```

- Memory deallocation:

```
delete pa;                // C++
free(pb);
delete[] pc;              // C++
```

Pointers

- Problems of operations with pointers
  - Usage of uninitialized pointers

```
int *pa;
*pa = 10;
```

```
int *pa = NULL;
*pa = 10;
```

  - Incorrect initialized pointers

```
int *pa;
pa = 10;
*pa = 10;
```

Pointers

- Problems of operations with pointers
  - Accessing a released pointer:

```
int *pa = new int;
*pa = 10;
delete pa;
(*pa)++;
```

  - Erroneous usage of pointers operations:

```
int a = 10;
int *pa = (&a)+1;
```

```
int a = 10;
float b;
int *pa = (&a)+1;
```

Pointers

- Problems of operations with pointers
  - Pointer usage out of its scope:

```
int *pa;
if( ... )
{
    int a = 10;
    pa = &a;
}
(*pa)++;
```

```
int* f(void)
{
    int a = 10;
    return &a;
}
```

Pointers

- Problems of operations with pointers
  - Dangling pointers:

```
int *pa = malloc(4), *pb = pa;
free(pa);
*pb = 10;
```

Pointers

- Problems of operations with pointers
  - Memory leaks

```
for( i = 0; i < 100000; i++ )
{
    int *pi = new int(i);
    *pi = rand();
    printf("%d", *pi);
}
```

Pointers

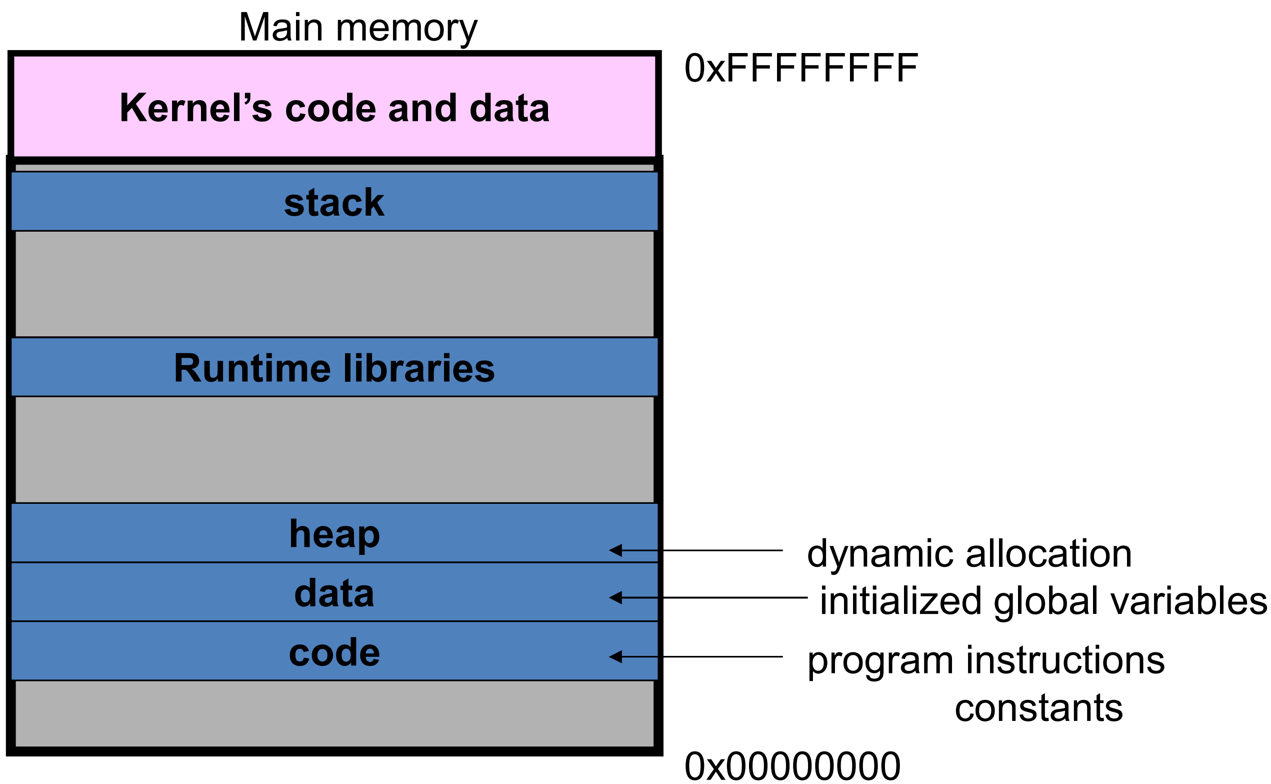
- Problems of operations with pointers
  - Multiple releases

```
int *pa = malloc(4), *pb = pa;
free(pa);
free(pb);
```

Pointers

- Pointer usage recommendations
  – The value addressed by the pointer is valid (points to an allocated memory block)
  – All dynamic variables allocated should be released once they are no longer needed
  – Avoid releasing memory multiple times
  – Be careful when using pointer arithmetics

Summary


Main memory
Kernel's code and data
stack
Runtime libraries
heap
data
code
0xFFFFFFFF
0x00000000
dynamic allocation
initialized global variables
program instructions
constants