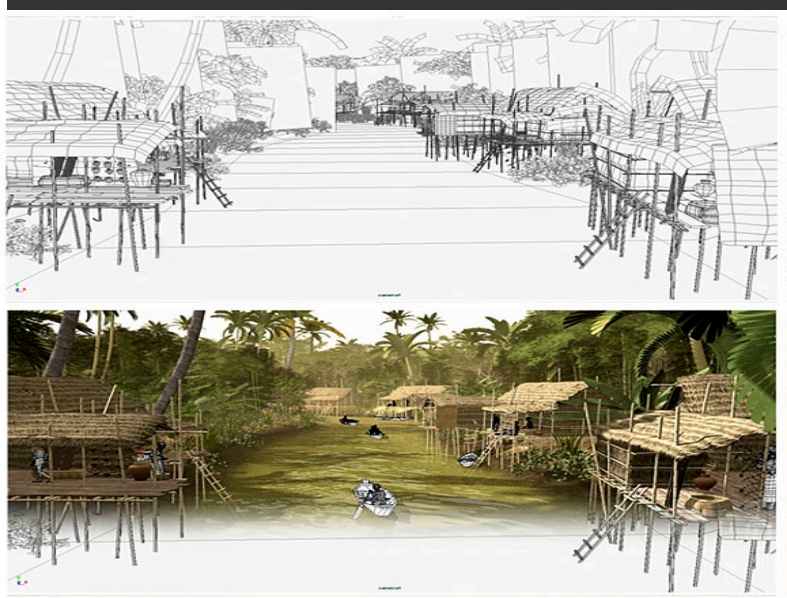
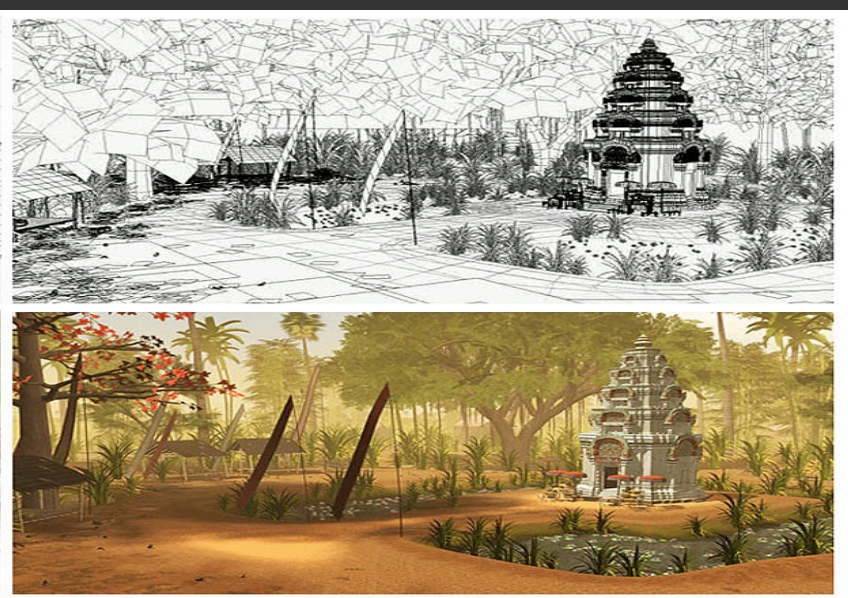
MONASH University

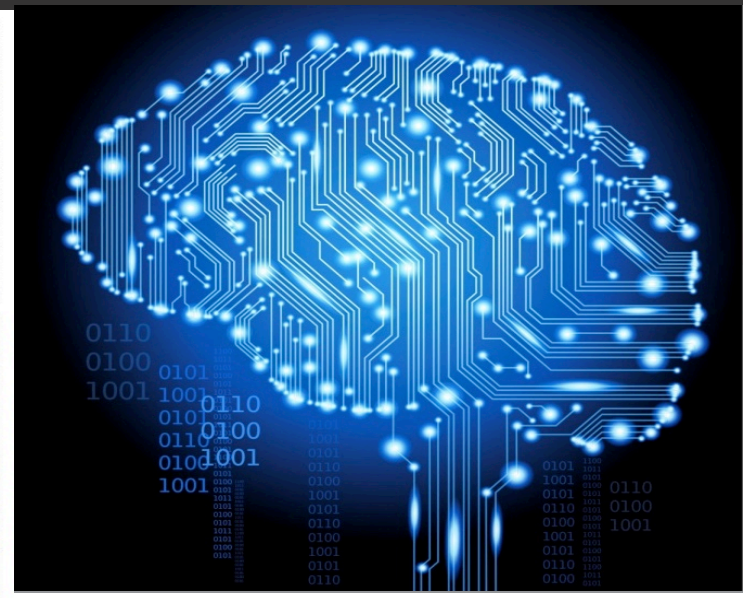
Information Technology

FIT1008/2085
MIPS – Arrays

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson







Where are we up to?

- Know the MIPS R2000 architecture and can name the main parts
- Understand the fetch-decode-execute cycle
- Able to use assembler directives
- Can program in assembly using the MIPS instruction set we will use
- Know the basics of the instruction formats
- Know how to translate simple programs with if-then-elses and loops
- Know how to do it faithfully

Learning objectives for this lecture

- To be able to translate into MIPS programs that contain arrays

MONASH University

Compiling arrays

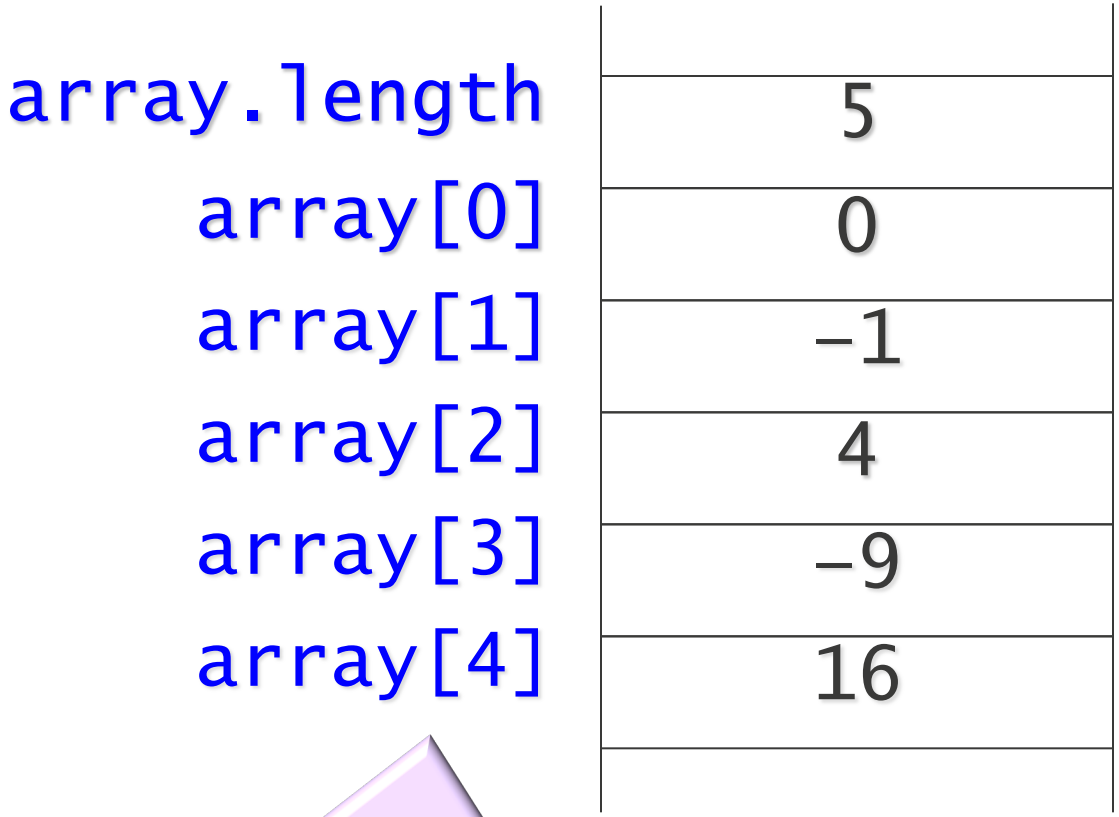
From Lists in Python to Arrays in MIPS

- **Python lists are implemented using arrays. Here we will use lists as arrays.**
- **When translating lists from Python to MIPS we will assume that:**
  - They have **fixed** length (they are really arrays, not lists)
    - Thus, once a list is created, no more elements can be added
    - In other words, don't use **`append`**!
  - They only have **integers** as elements
    - Elements are **automatically** initialized to 0 (rather than to None) by the system, so no need for you to do it (unless I ask you to do it)
  - The data kept for a given list is only:
    - Its **length**
    - Its **elements**

In reality, a lot of object/class related info is stored with a Python list

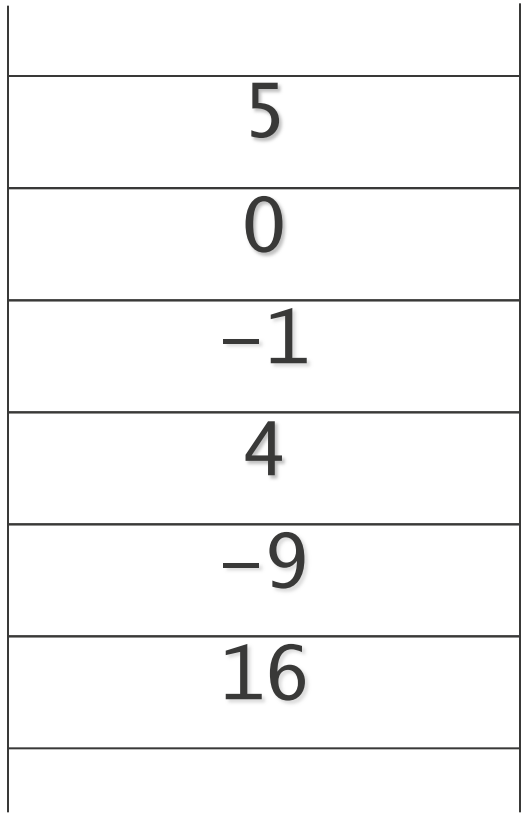
- **But how do we translate the access into each of its elements?**

From Arrays in Python to Arrays in MIPS



The high-level programmer's view: array **a** is accessed through indices 0, 1, 2, ...

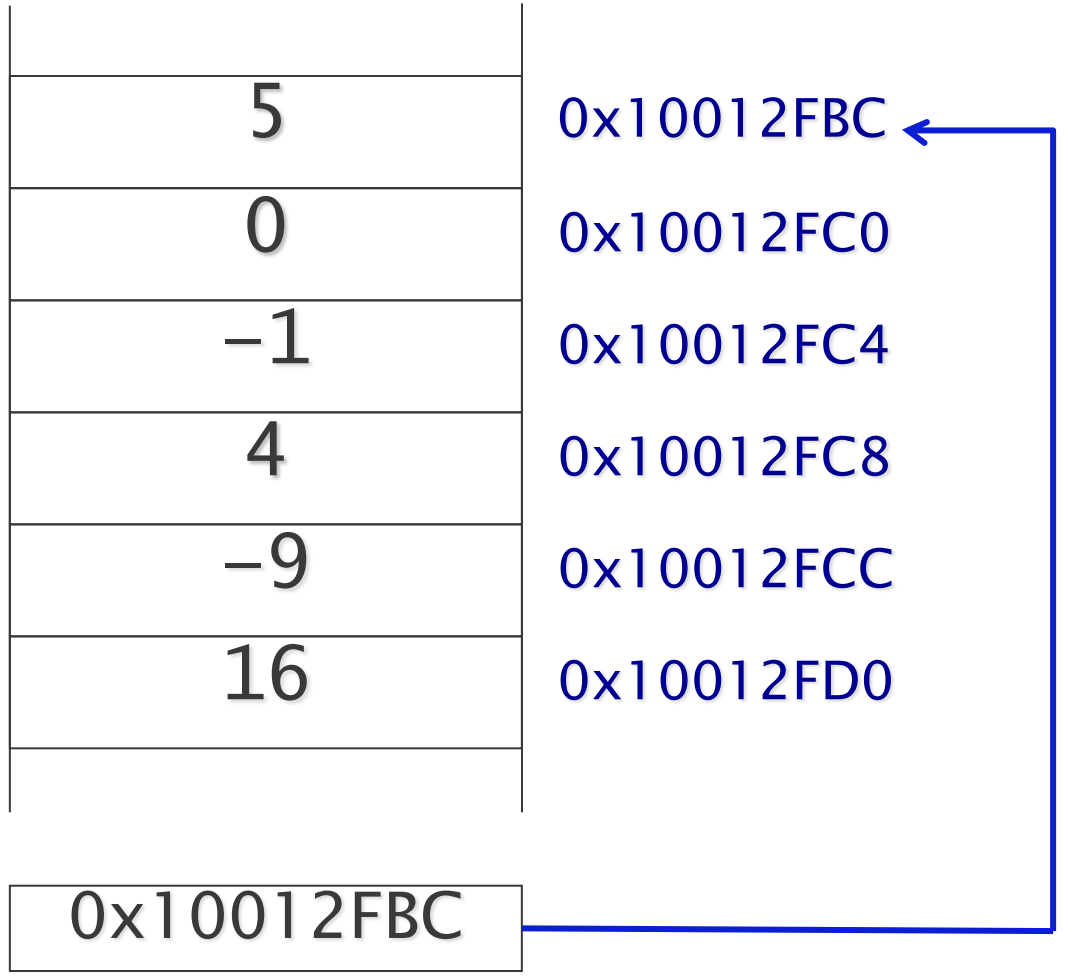
From Arrays in Python to Arrays in MIPS



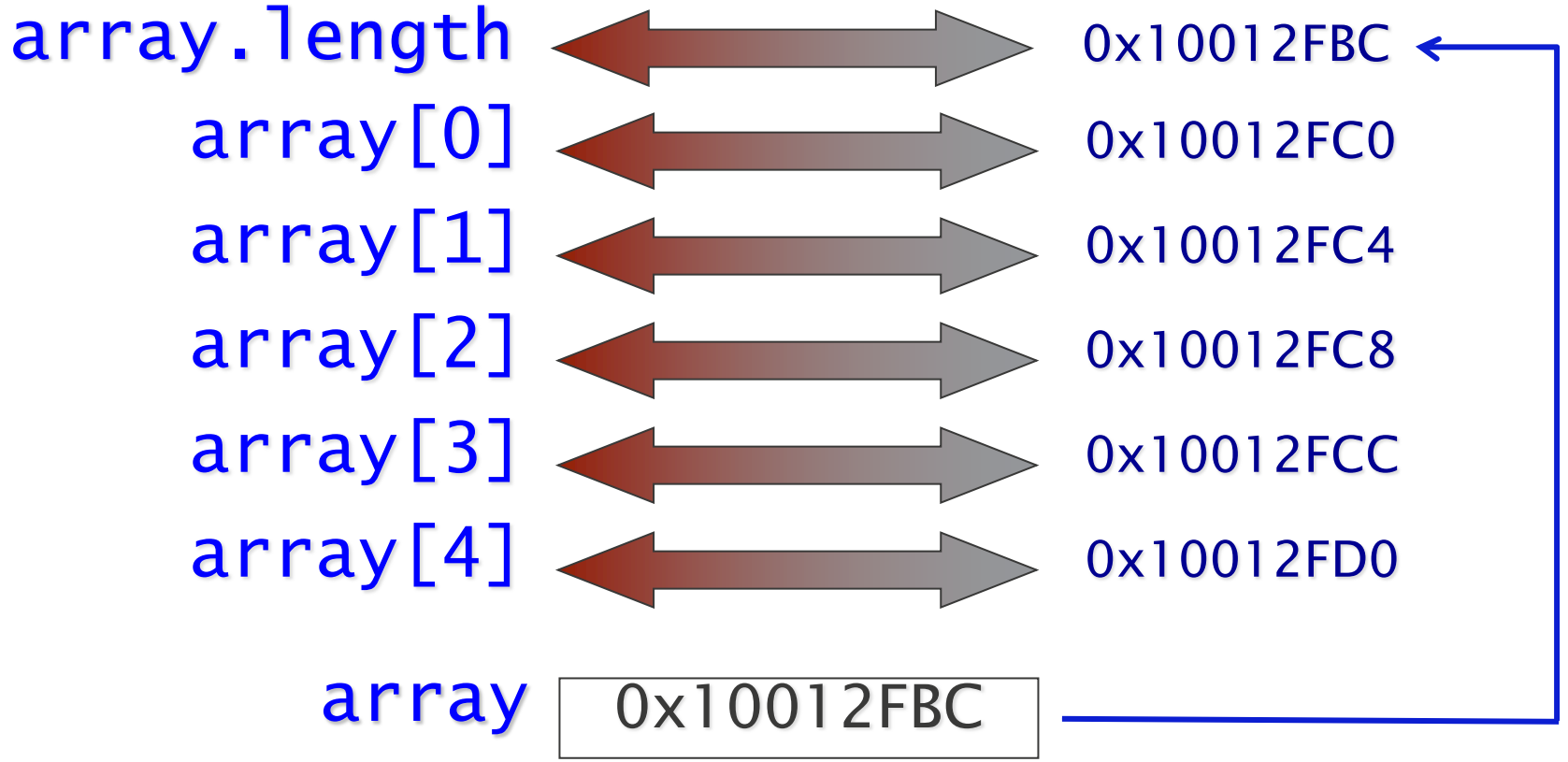
Remember: in FIT1008/2085 we only load/store 4 bytes. Thus, all addresses are multiples of 4 and, thus, the last two bits are always 0.

The computer's view: the array is part of memory, accessed through addresses 0x10012FC0, 0x10012FC4, …

From Arrays in Python to Arrays in MIPS

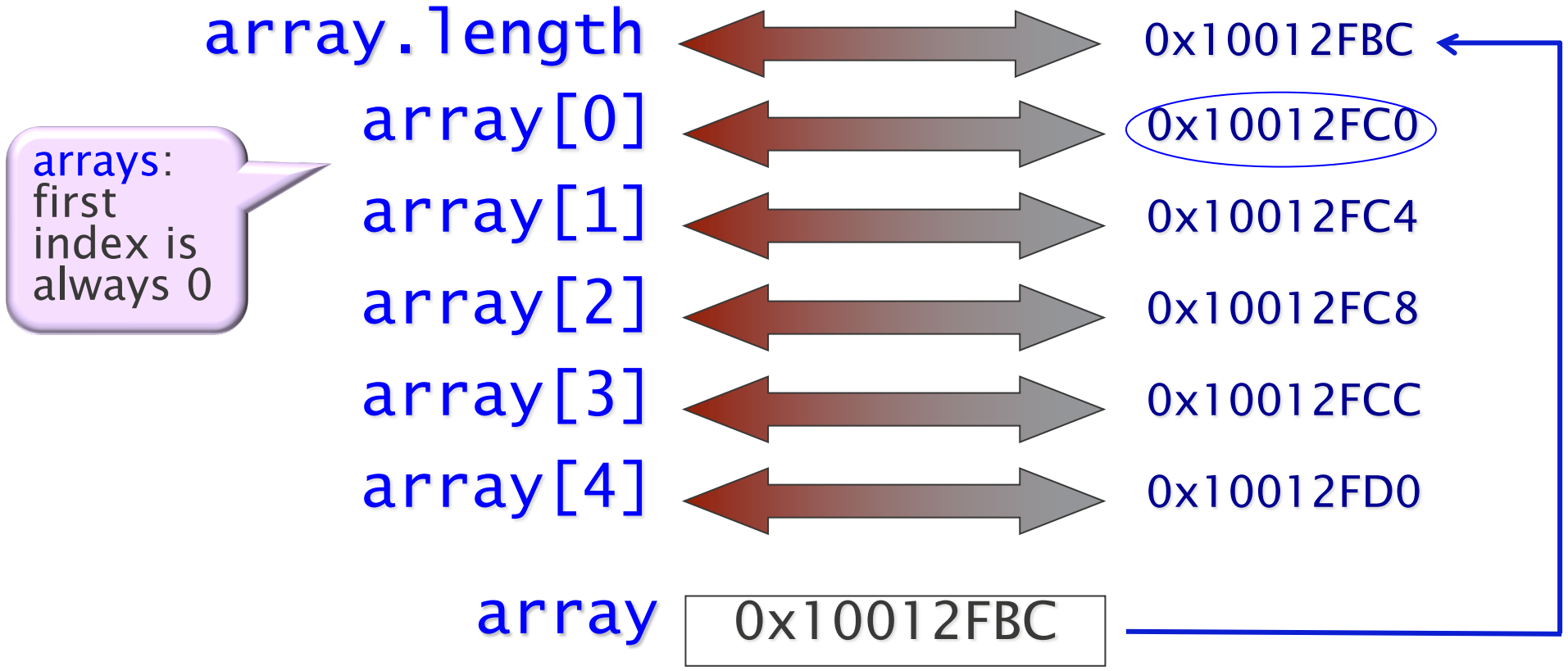


From Arrays in Python to Arrays in MIPS



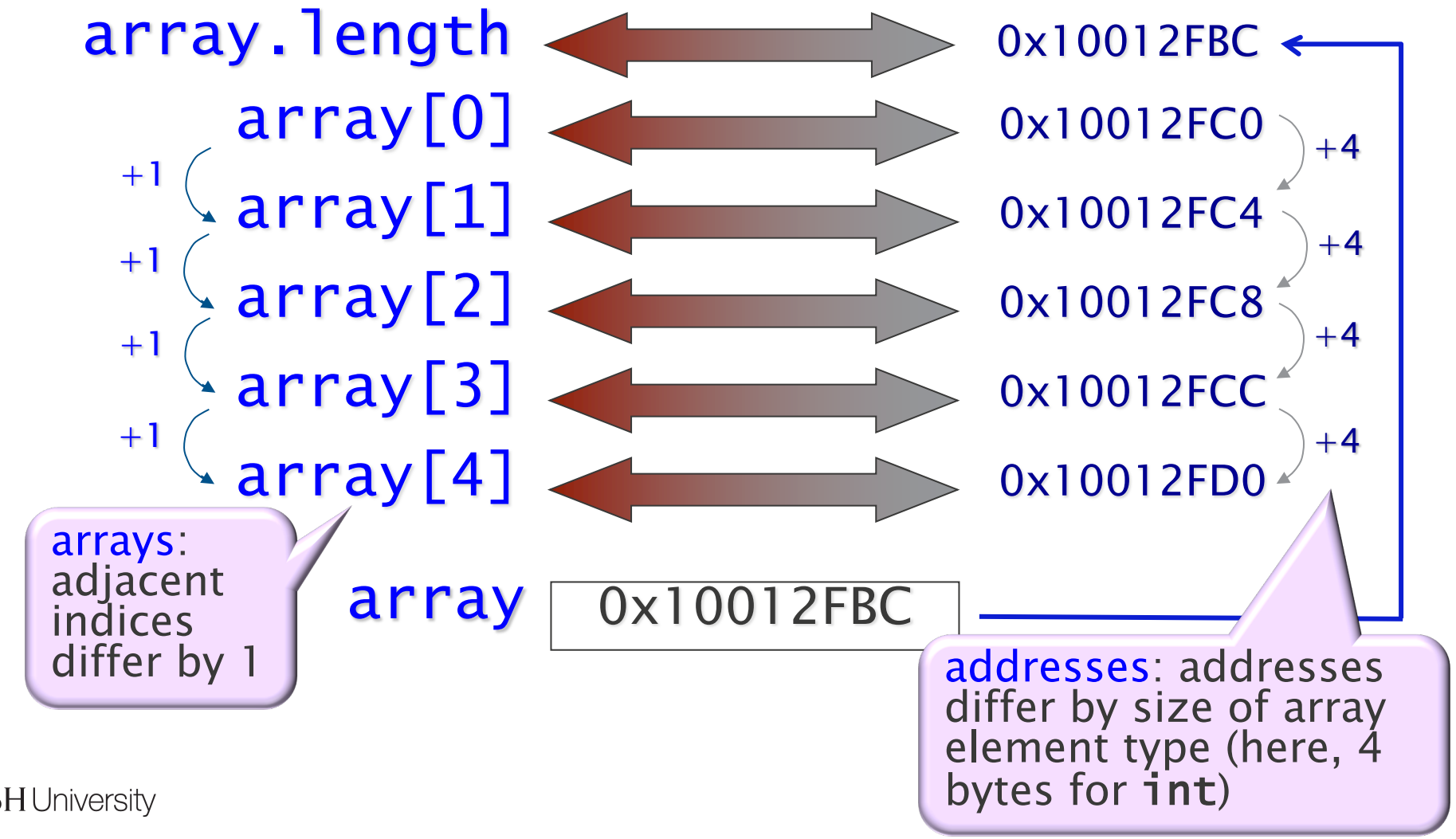
To program arrays in assembly language, need to understand their relationship, and how to convert from one two another.

From Arrays in Python to Arrays in MIPS



addresses: for a given array, address of first element is constant: address of array **a** + 4 bytes to skip the **int** length (here, 0x10012FBC+4 = 0x10012FC0)

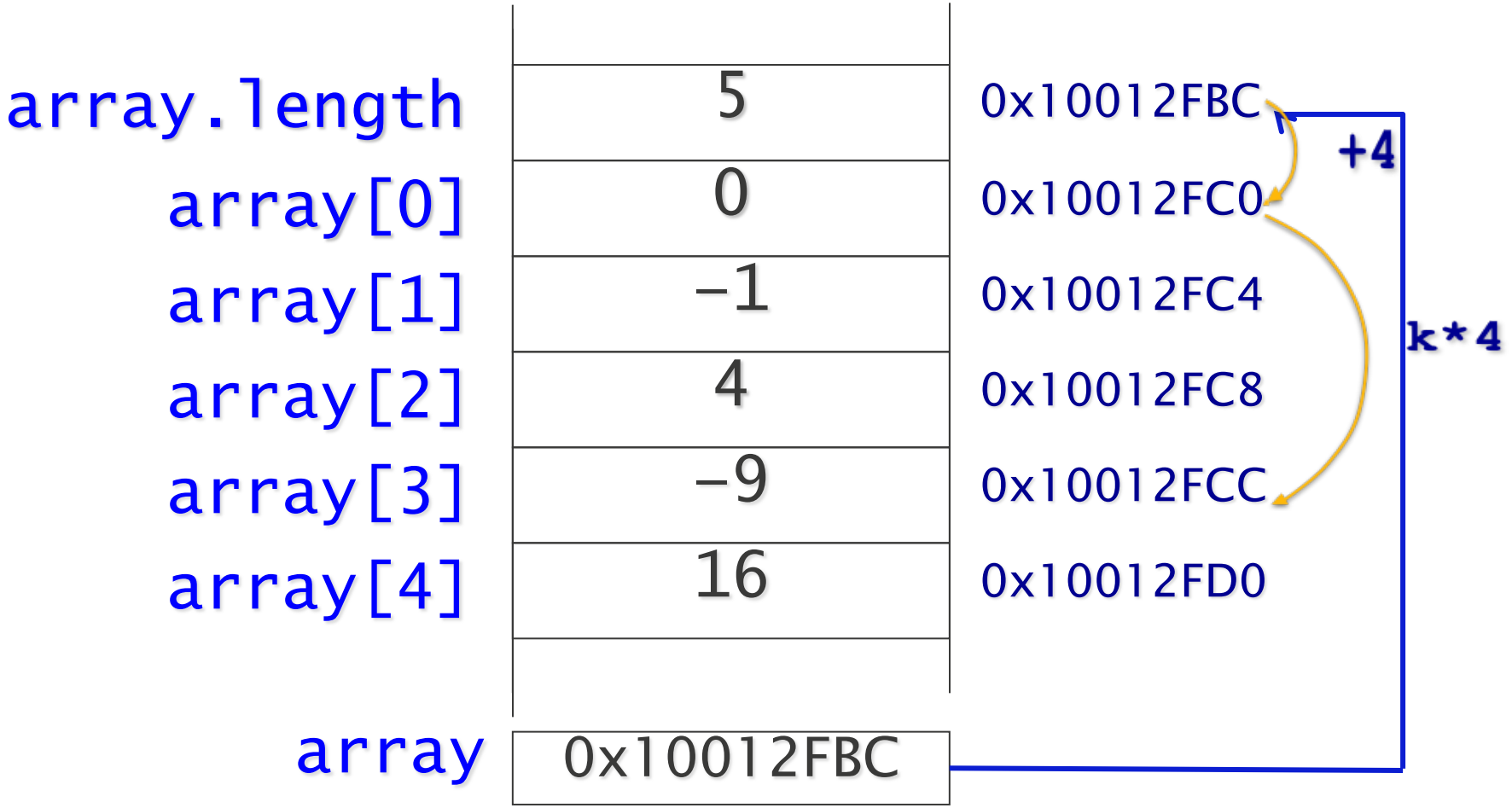
From Arrays in Python to Arrays in MIPS



From Arrays in Python to Arrays in MIPS

- **To compute address of `array[k]`**
  - Determine start address of the elements in the array
    - Address contained in **`array + 4`** (the 4 is needed to skip **`array.length`**)
    - Numerically smallest address of any element in the array
  - Determine size of one element of **`array`**
    - In bytes (we always assume an integer, so 4 bytes)
  - Compute k
    - Can be an arbitrary integer expression
- **In summary: the address of element k is start + (size * k)**
- **Important: need load/store instruction to access `array[k]`'s data**
  - Since the above calculation only computes the address of (i.e., a pointer to) `array[k]`

Computing the address of array[k]



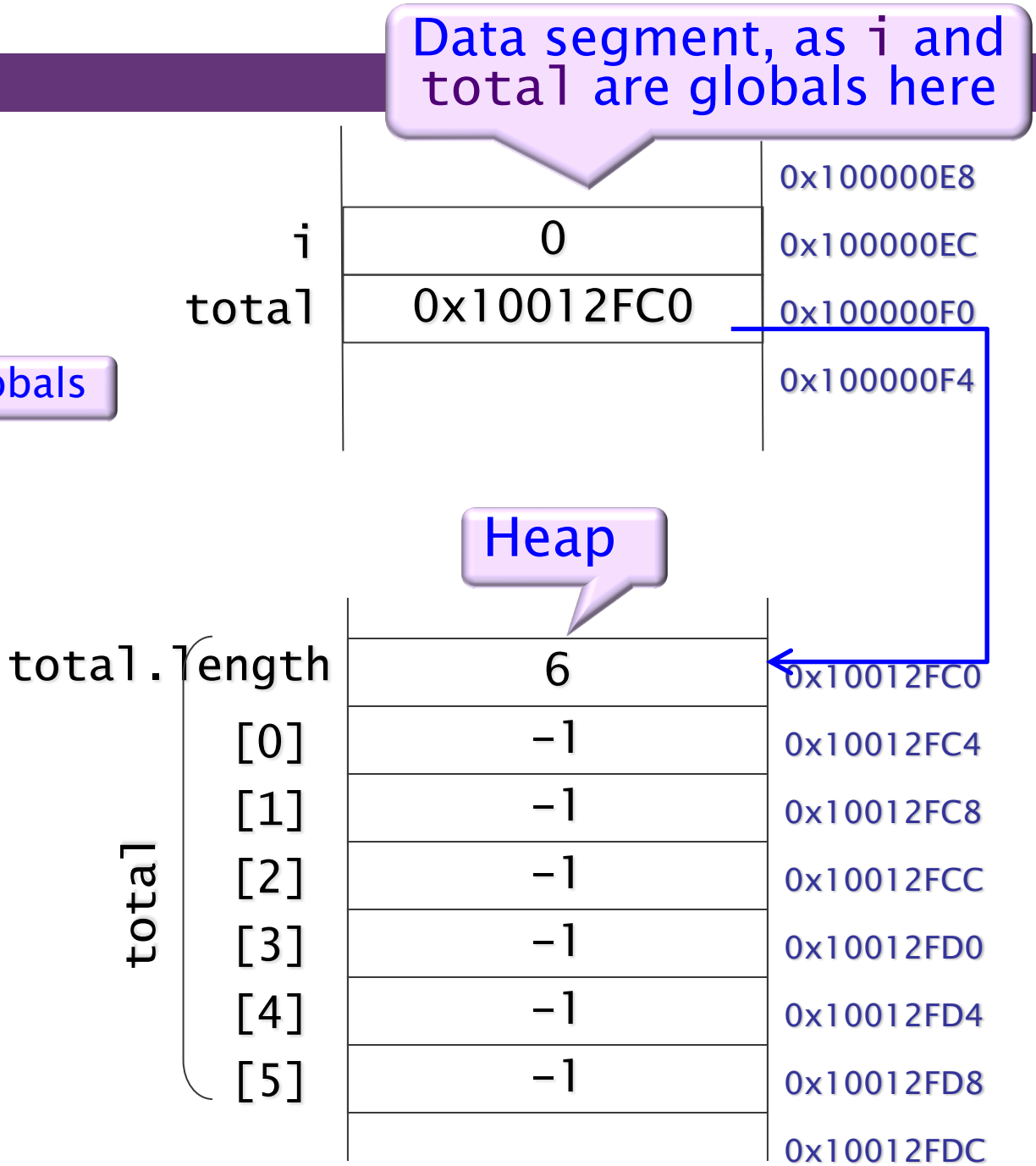
Address of array[k] = address in array + 4 + k*4

Address of array[3] = 0x10012FBC+ 4 + 12 = 0x10012FCC

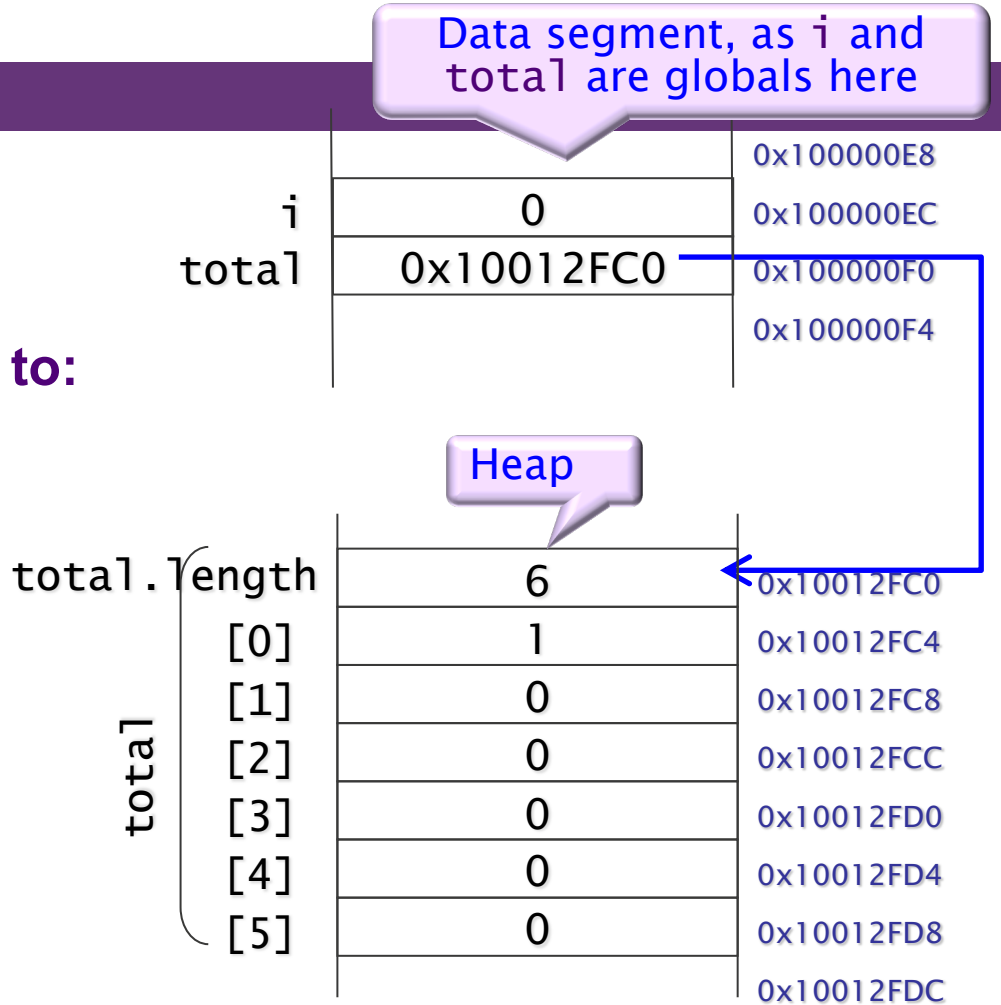
Example

```
# Array example.

i = 0
total = [-1,-1,-1,-1,-1,-1]
# MEM DIAG REPRESENTS UP TO HERE

# table of factorials
total[0] = 1
for i in range(1, 6, 1):
    total[i] =
        total[i - 1] * I

# Print final total
print(total[5])
```

We will treat i and total as globals



Confusing addresses?



Consider the following Python code, executed up to:

```
i = 0
total = [1,0,0,0,0,0]

# MEM DIAG REPRESENTS UP TO HERE...
```

Which results in the associated memory diagram.

What does this MIPS code do?

```
SOMEHERE IN THE .text part

la $t0, total          # $t0 = 0x100000F0, address of the total label
lw $t1, total          # $t1 = 0x10012FC0, value of global var total
lw $t2, total($0)      # $t2 = 0x10012FC0, same as it adds 0 to total
lw $t3, total+4($0)    # $t3 = ?, value of address 100000F4, garbage
lw $t4, ($t1)          # $t4 = 6, value of the heap address in total
lw $t5 4($t1)          # $t5 = 1, next value of the heap address
```

From Arrays in Python to Arrays in MIPS

- **To compute the start address we will use a syscall, as usual**
- **MARS syscall (sbrk): number 9**
  - It takes in `$v0` number 9,
  - In `$a0` the number of bytes to allocate in the heap
    - For arrays: the number of elements*4 + 4
  - Returns in `$v0` the address of the lowest address in the block

Reminder: Many ways to specify an address

- **Directly (or using a label), e.g.**

```
lw $t1, N  # loads from label N
```

- **Label plus offset, e.g.**

```
lw $t1, N+4      # loads from (label N + 4)
```

- **Using a GPR to store the address, e.g.**

```
lw $t1, ($s0)    # loads from address stored in $s0
```

- **GPR + offset, e.g.**

```
lw $t1, 4($s0)   # loads from (address stored in $s0)+4
```

- **Label, offset, and GPR, e.g.**

```
lw $t1, N+4($s0)# loads from (label N+4)+contents of $s0
```

Example

Same as = [-1,-1,-1,-1,-1,-1]

If the size is unknown, I must use a loop

```
i = 0
total = [-1]*6

# table of factorials

total[0] = 1
for i in range(1, 6, 1):
    total[i] =
        total[i - 1] * i

# Print final total
print(total[5])
```

Must be a register, not a label

lw, not la

```
        .data
i:      .word 0
total:  .word 0

        .text

        addi $v0, $0, 9  # allocate
        addi $a0, $0, 28 # (6*4)+4
        syscall
        sw $v0, total    #total=address
        addi $t0, $0, 6 #$t0 = 6
        sw $t0, ($v0)    #total.length=6

        lw $t0, total
        addi $t1, $0, -1
        sw $t1,  4($t0) # total[0]=-1
        sw $t1,  8($t0) # total[1]=-1
        sw $t1, 12($t0) # total[2]=-1
        sw $t1, 16($t0) # total[3]=-1
        sw $t1, 20($t0) # total[4]=-1
        sw $t1, 24($t0) # total[5]=-1
```

Often you will only need to initialise the size, not the elements

Example

Same as = [-1,-1,-1,-1,-1,-1]

If the size is unknown, I must use a loop

```
i = 0
total = [-1]*6

# table of factorials

total[0] = 1
for i in range(1, 6, 1):
    total[i] =
        total[i - 1] * i

# Print final total
print(total[5])
```

Must be a register, not a label

lw, not la

Often you will only need to initialise the size, not the elements

```
        .data
i:      .word 0
total:  .word 0

        .text

addi $v0, $0, 9  # allocate
addi $a0, $0, 28 # (6*4)+4
syscall
sw $v0, total    #total=address
addi $t0, $0, 6 #$t0 = 6
sw $t0, ($v0)    #total.length=6

lw $t0, total
addi $t1, $0, -1
sw $t1,  4($t0) # total[0]=-1
sw $t1,  8($t0) # total[1]=-1
sw $t1, 12($t0) # total[2]=-1
sw $t1, 16($t0) # total[3]=-1
sw $t1, 20($t0) # total[4]=-1
sw $t1, 24($t0) # total[5]=-1

# total[0] = 1
lw $t0, total    # total
addi $t1, $0, 1 # $t1 = 1
sw $t1, 4($t0)  # total[0]=1

# i = 1
addi $t0, $0, 1 # $t0 = 1
sw $t0, i       # i=1
```

Example

```
i = 0
total = [-1]*6

# table of factorials

total[0] = 1
for i in range(1, 6, 1):
    total[i] =
        total[i - 1] * i

# Print final total
print(total[5])
```

Could have been done better with sll to *4

We will use & to denote the address as opposed to the content of an array position, e.g., &(total[i-1])

```
# ... continued (part 2)
loop:  # i < 6
        lw $t0, i          # i
        slti $t0, $t0, 6 # is i < 6?
        beq $t0, $0, endloop # not i<6 ->end

        # shift left 2 used here
        # to scale by sizeof(int)

        # total[i-1] (careful with length)
        lw $t0, total      # total
        lw $t1, i          # i
        addi $t1, $t1, -1 # i-1
        addi $t2, $0, 4    # $t2 = 4
        mult $t1, $t2      # (i-1)*4
        mflo $t1           # $t1 = (i-1)*4
        add $t0, $t0, $t1 # &(total[i-1])-4
        lw $t2, 4($t0)     # $t2=total[i-1]

        # total[i-1]*i
        lw $t1, i          # i
        mult $t1, $t2      # total[i-1]*i
        mflo $t2

        # total[i]=
        lw $t0, total
        lw $t1, i          # i
        sll $t1, $t1, 2    # i * 4
        add $t0, $t0, $t1 # &(total[i])-4
        sw $t2, 4($t0)     # $t2=total[i]

# continued ...
```

Example

```
i = 0
total = [-1]*6

# table of factorials

total[0] = 1
for i in range(1, 6, 1):
    total[i] =
        total[i - 1] * i

# Print final total
print(total[5])
```

```
# ... continued (part 3)
    # i++
    lw $t0, i          # i
    addi $t0, $t0, 1 # i+1
    sw $t0, i          # i=i+1

    # Repeat loop
    j loop

endloop:
    # Print total[5]
    addi $v0, $0, 1   # print int
    # Allowed arbitrary
    # expression provided it
    # is constant
    lw $t0, total       # total
    lw $a0, 4+5*4($t0) # &total[5]
    syscall

    # exit
    addi $v0, $0, 10
    syscall
```

Summary

- **Now able to program in MIPS with simple arrays, and translate high-level code with arrays into MIPS**
- **Further reading (see afterwards):**
  - Simple program optimisations
  - The compilation process in more detail

Another Example

```
read(size)
the_list = [0]*size
for i in range(size):
    read(the_list[i])
```

No need to initialise the elements to 0, just initialise the size

```
        .data

        i: .word 0
     size: .word 4
the_list: .word 4

        .text
        # read(size)
        addi $v0, $0, 5
        syscall
        sw $v0, size

        # create a list of size elements
        addi $v0, $0, 9  #allocate
        lw $t0, size
        addi $t1, $0, 4  # $t1 = 4
        mult $t0, $t1    # size*4
        mflo $t1         # $t1 = size*4
        addi $a0, $t1, 4 #(size*4)+4
        syscall
        sw $v0, the_list #the_list=address
        sw $t0, ($v0) #the_list.length=size

        sw $0, i
 loop: # while i < size
        lw $t0, i           # i
        lw $t1, size        # size
        slt $t0, $t0, $t1   # is i < size?
        beq $t0, $0, end    # not i<size->end

        # read(the_list[i])

        # continued ...
```

Another Example

```
read(size)
the_list = [0]*size
for i in range(size):
    read(the_list[i])
```

```
    # read(the_list[i])
    lw $t0, i           # i
    lw $t1, the_list    # the_list
    addi $t2, $0, 4     # $t2 = 4
    mult $t0, $t2       # i*4
    mflo $t0            # $t0 = i*4
    add $t0, $t0, $t1   # &(the_list+i*4)
    addi $v0, $0, 5     # read item
    syscall
    sw $v0, 4($t0)      # the_list[i]=item

    # i += 1
    lw $t0, i          #i
    addi $t0, $t0, 1   # i+1
    sw $t0, i          #i=i+1

    # restart the loop
    j loop

end:


    # rest of the program ...
```

MONASH
University

Some compiler optimisations (for those interested)

Some simple optimizations

Assume `y` is a global variable

- **Constant folding**
  - `s = 60 * 60 * 24` ⇨ `s = 86400`
- **Replace multiplication/division by power of two with shift**
  - `x = y * 8` ⇨ `x = y << 3`

Bitwise shifting in Python (you can do it at high level too!)

```
lw $t0, y
addi $t1, $0, 8
mult $t0, $t1
mflo $t0
sw $t0, x
```

⇨

```
lw $t0, y
sll $t0, $t0, 3



sw $t0, x
```

Some simple optimizations

- **Re-ordering expressions to use fewer registers**
  - `x = 5 + 6 * y` ⇨ `x = 6 * y + 5`

```
addi $t0, $0, 5
addi $t1, $0, 6
lw $t2, y
mult $t1, $t2
mflo $t1
add $t0, $t0, $t1
sw $t0, x
```

⇨

```
addi $t1, $0, 6
lw $t2, y
mult $t1, $t2
mflo $t1
addi $t0, $t1, 5
sw $t0, x
```

Some complex optimizations

- **Keeping and re-using values in registers**
- **Extracting invariant expressions from loop and evaluating once before loop entry**
  - `while i < x*y:` ⇨
    `n = x*y; while i < n:`

Loop invariant expression: an expression whose value does not change inside the loop

- **Removing redundant variables**
  - `b = c; a = b + 3` ⇨ `a = c + 3`

- **Introducing other variables**
  - `x = a.hard() + 5; y = a.hard() - 5` ⇨
    `temp = a.hard(); x = temp + 5; y = temp - 5;`

Is this always safe?

Some complex optimizations

- **Changing loop exit conditions**
  - `while i < 10:` ⇨ `while not i = 10:`

Is this always safe?

Assumes `i` is a global variable

```
lw $t0, i
slti $t1, $t0, 10
beq $t1, $0, end
```

⇨

```
lw $t0, i
addi $t1, $0, 10
beq $t1, $t0, end
```

Same number of instructions. So what's the gain?

The **`addi`** is only executed once (i.e., outside the loop)

MONASH
University

Compilation Process
(for those interested)

Programming Languages

- **A programming language (PL) must be:**
  - Universal: any computable problem must be programmable in that language (basically: I/O, basic data manipulation and recursion)
  - Implementable: every well formed program must be able to be executed
- **They are at the core of computer science, as the principal tool of the programmer**

> The limits of my language are the limits of my world – Ludwig Wittgenstein

How are they specified?

- **Syntax: what the basic elements are, how to combine them to form valid sentences**
- **Semantics: what these elements and their combinations "mean". For example:**
  - Operational: simplified execution model
  - Denotational: mathematical functions
  - Axiomatic: mathematical logic
- **Example: X = X+1 is syntactically valid for both C and Prolog, but has very different semantics**

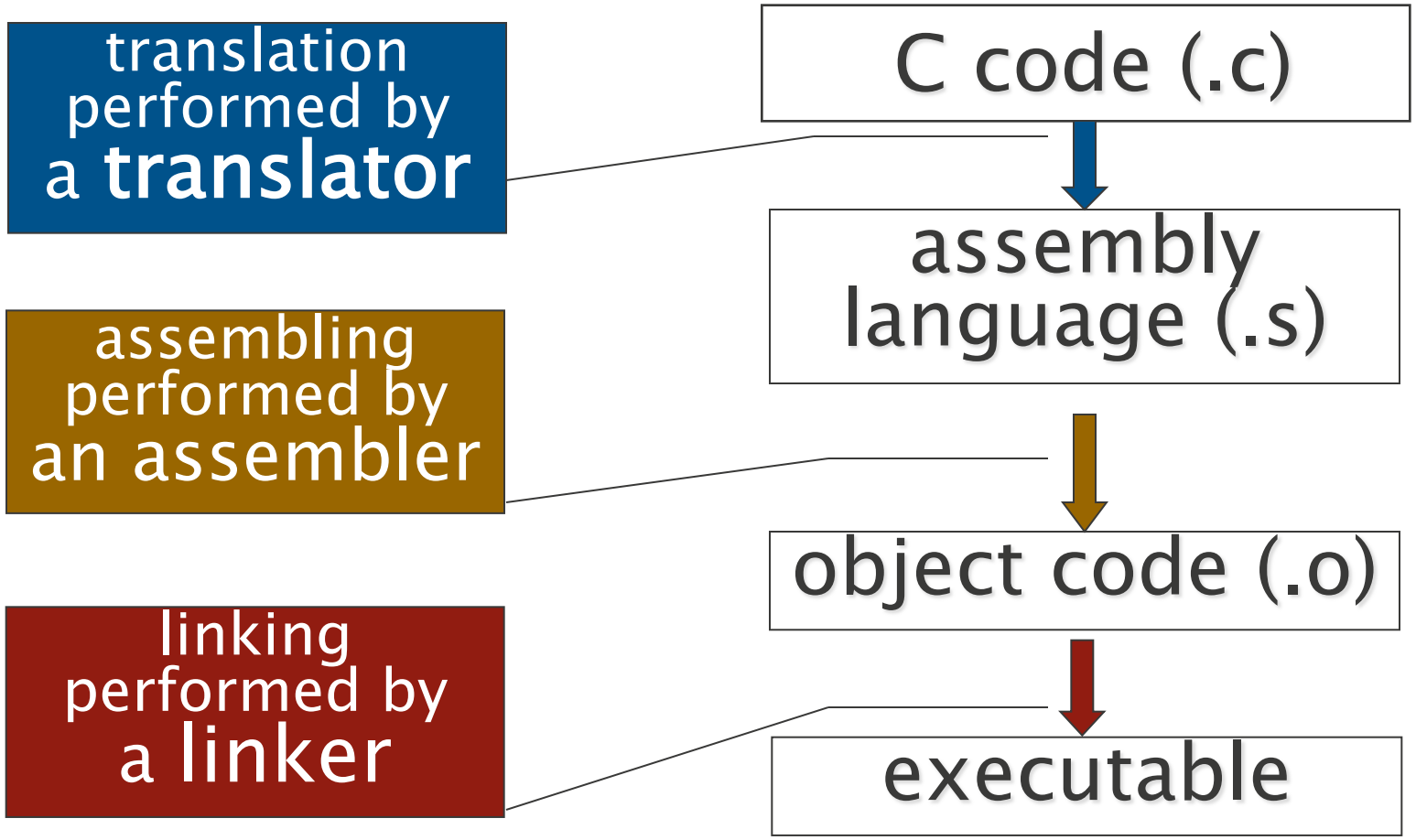
How are these languages implemented?

- **The programming language implementation must:**
  - Check the validity of the syntax
  - Ensure the execution follows the semantics
- **Can be performed by a compiler, interpreter, a hybrid or all these:**
  - Compiler: translates the program into some other language. Traditionally, machine language; nowadays can be C, Java, etc.
  - Interpreter: executes the program directly (no other program created). Acts as a software simulation of the machine (HTML)
  - Hybrid: programs are compiled into lower-level virtual machine code, which is then interpreted (most Prolog, Java, Python)
- **Nowadays distinction between compiler/interpreter is a bit fuzzy**
  - For example, some people say Python is interpreted. But Cpython does create bytecode files (.pyc).

Traditional view of Compilers

- **Set of tools (programs) that convert higher-level code to machine language**
- **Traditional model:**
  - Translator: from high level code to assembly language
  - Assembler: from assembly language to object code
  - Linker: from object code to executable program
- **Most compilers can perform all the above steps**
  - Often, they have options that can halt processing at any stage

A traditional compiler for C



Object Code versus Machine Code

- **Some programs may refer to variables or methods defined elsewhere**
  - E.g., in Python the `print` method is a built-in, not defined by the user
- **So, how do we know where in the text segment its definition is?**
- **Object code: code compiled as far as it can be without resolving these references**
  - Mainly machine language, but with additional data identifying symbols (like `print`) that still need resolving
- **Linking: combining object modules and resolving these references so that they refer to the correct memory addresses**
  - Produces pure machine language executable

Separate compilation and Linking

- **Why should several object modules be combined?**
- **Because programs may be constructed from many source files (modules, classes, etc)**
- **Separate files can be compiled to the object code stage "independently" of each other**
  - If one file is changed, other files don't need recompiling
- **This is referred to as separate compilation**
- **All required object files are then linked simultaneously to form executable code**
  - Including libraries written by OS vendor
- **Nowadays, linking is often done at run-time (dynamic linking)**

Compilers

one.c

```
/* Variable foo, defined elsewhere. */
extern int foo;
/* Function func defined elsewhere. */
void func(void);

/* main uses both foo and func */
int main() {
  /* ... */ foo++;
            func(); /* ... */
}
```

two.c

```
#include <stdio.h>
/* Variable foo is defined here. */
int foo;
/* Variable secret too, but is private*/
static int secret;

/* Function func is defined here */
void func(void) {
  printf( /* ... */ );
}
```

gcc -c one.c -o one.o

gcc -c two.c -o two.o

one.o

```
..01.. foo ..11.. main
..00.. func ..10..
```

gcc one.o two.o -o abc

two.o

```
..01.. foo ..11.. func
..11.. printf ..01..
```

library

```
..00.. printf ..11..
strlen ..11.. lots ...
```

abc

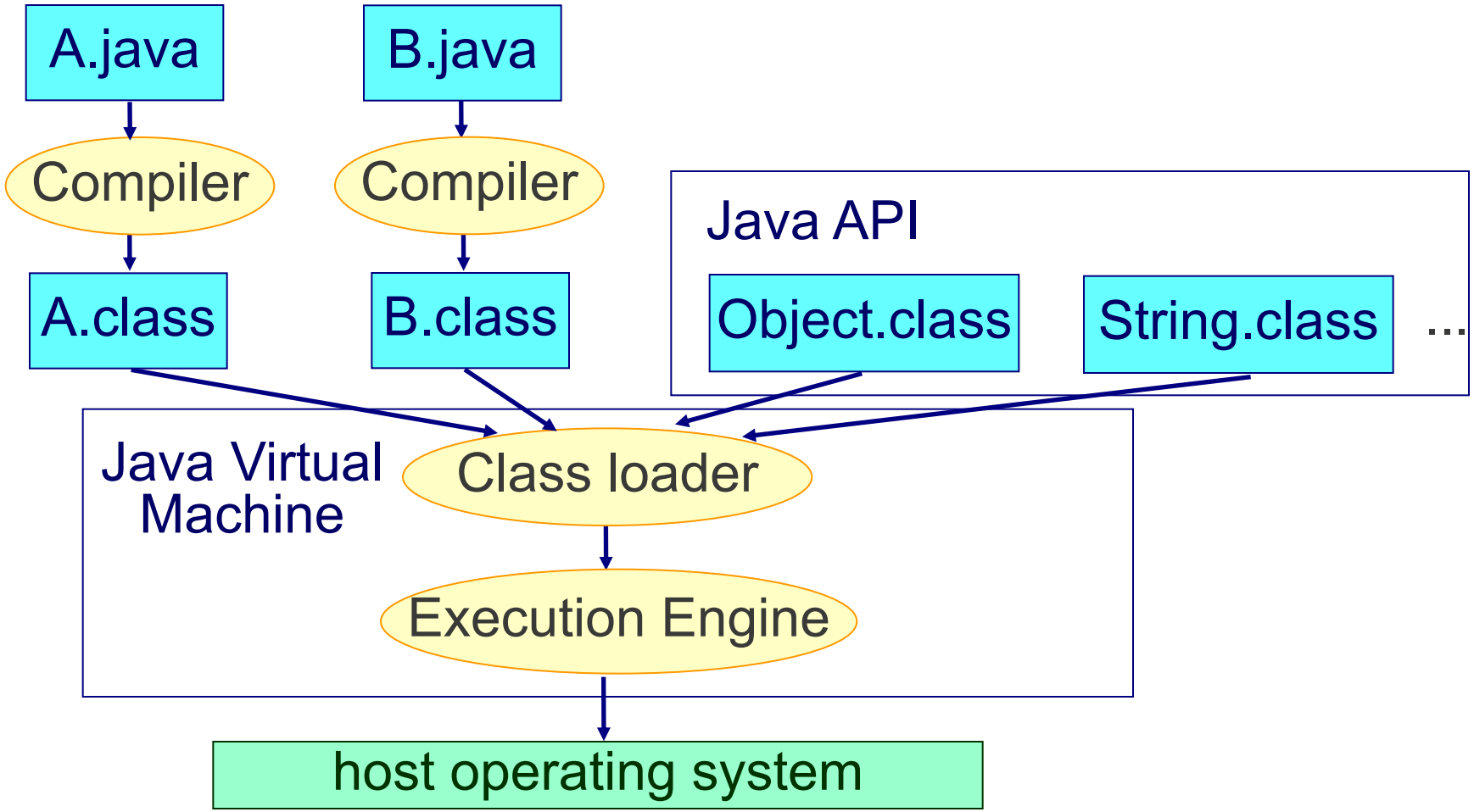
```
01101010010010010110
11001001100001101010
```

-c option stops at object code stage

More modern view: Virtual Machines

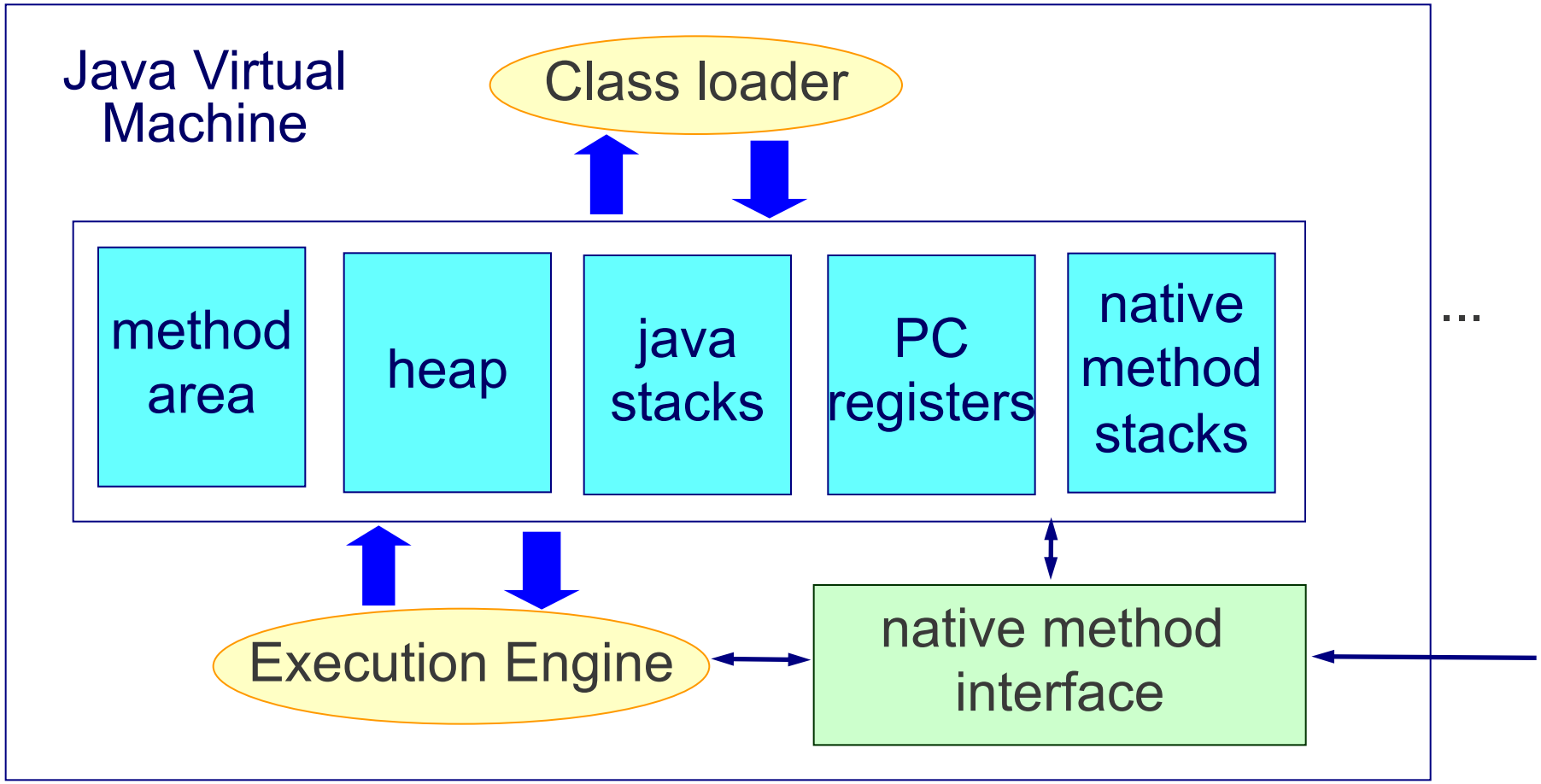
- **Most modern compilers rely on a virtual machine**
- **Aim: provide a platform-independent environment**
  - Allows a program to execute in the same way on any platform
- **Interpreted or compiled, a virtual machine is just an application:**
  - Written in some language and capable of running in any computer
- **Both Java and Python use virtual machines**
- **Most Java implementations translate Java to JVM bytecode:**
  - JVM: Java Virtual Machine
- **The most popular Python implementations translate Python to:**
  - Python byte code which is then interpreted by PVM (Cpython)
  - Java byte code which is then interpreted by JVM (Jython)
  - Python byte code which is then interpreted by PVM (PyPy)
  - .Net byte code which is then interpreted by CLR (IronPython)

Example: Java Virtual Machine



From Bill Venners' "Inside the Java 2 Virtual Machine"

Example: Java Virtual Machine



From Bill Venners' "Inside the Java 2 Virtual Machine"