Machine Architecture - Lecture 8

Ioannis Ivrissimtzis

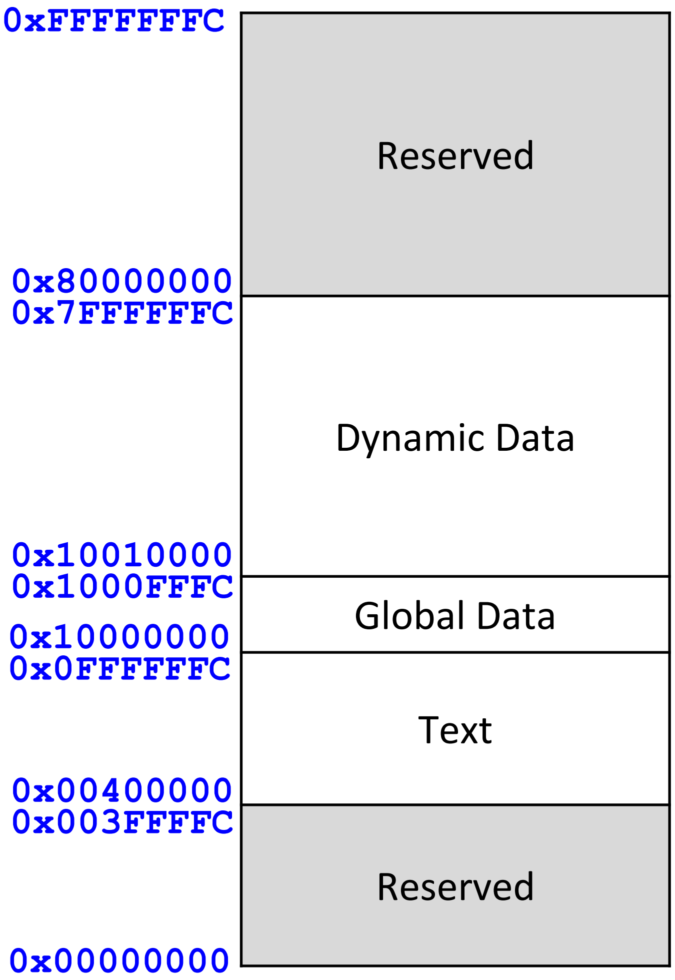
ioannis.ivrissimtzis@durham.ac.uk

MIPS - Compiling, assembling and loading

Slide material acknowledgements: Magnus Bordewich

The MIPS memory map

MIPS memory map



With 32-bit addresses, the MIPS address space spans

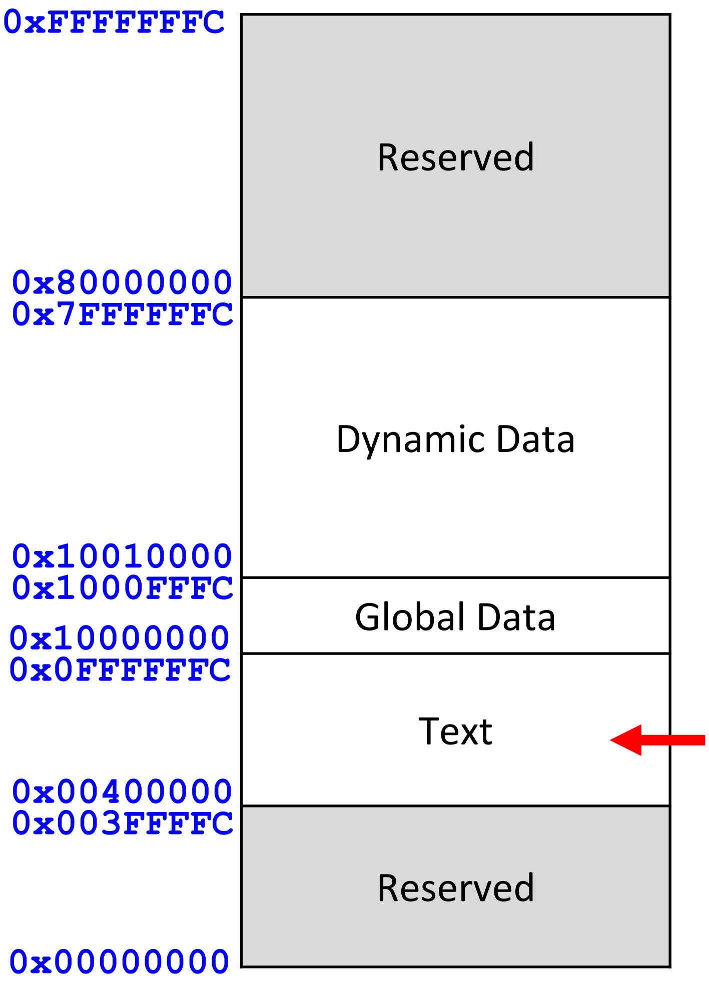
2^{32} bytes = 4 gigabytes (GB).

Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC.

The MIPS architecture divides the address space into four parts:

text segment
global data segment
dynamic data segment
reserved segments

The text segment

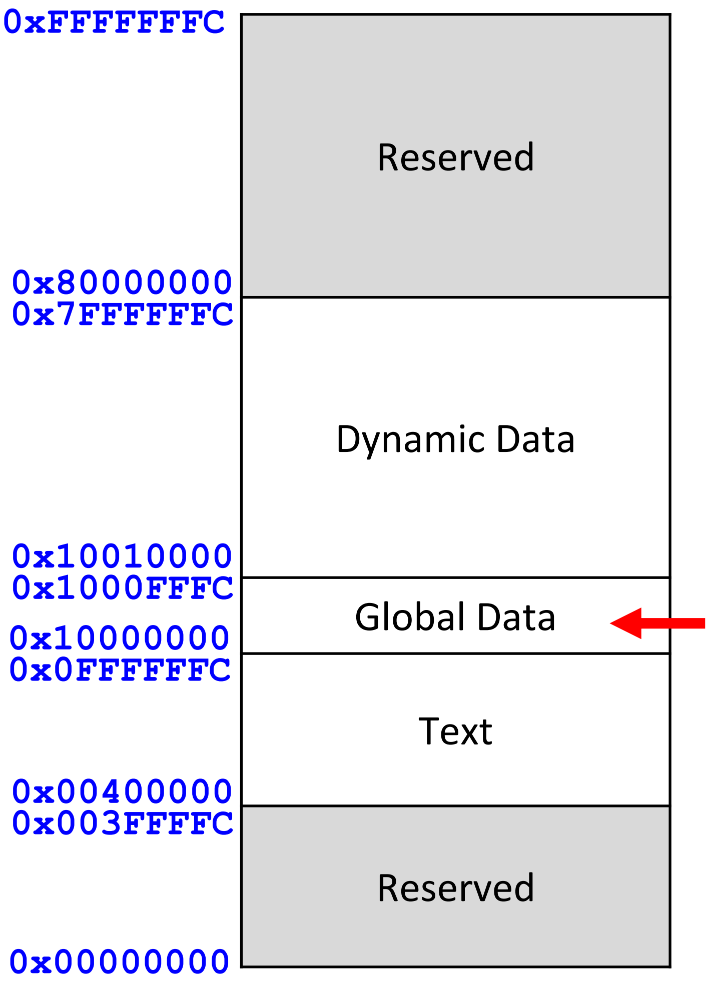


The text segment stores the machine language program.

It can accommodate almost 256 MB of code.

The four most significant bits of any word address in that segment are all 0, so the `j` instruction can directly jump to any address in the program.

The global data segment

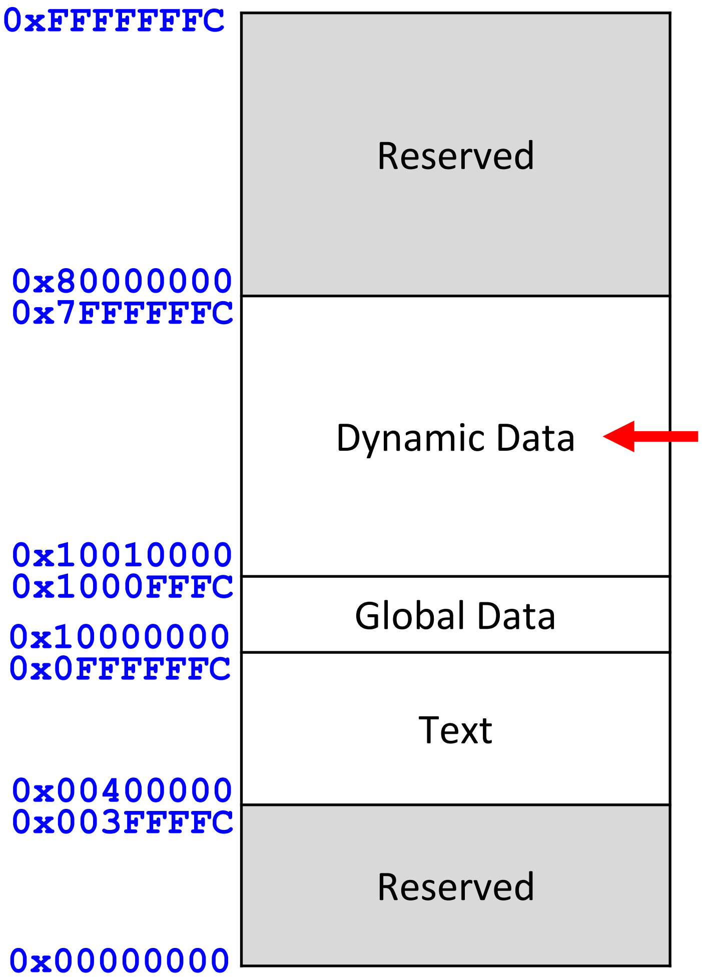


The global data segment stores global variables, which can be seen by all functions in a program. It can store 64 KB of data.

Global variables are accessed using the pointer $gp.

By convention, we initialise **$gp** at the middle of the global data segment at value **0x10008000**. The value of **$gp** stays constant throughout execution, and global variables are addressed as offsets from **0x10008000**.

The dynamic data segment

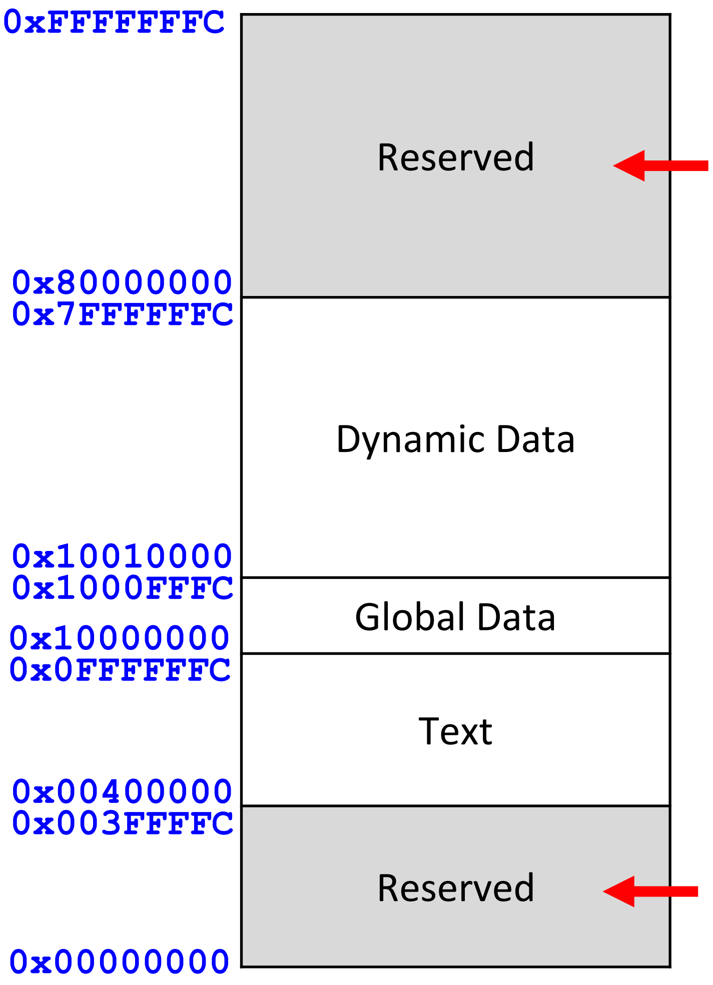


The dynamic data segment stores data that are dynamically allocated and deallocated throughout the execution of the program.

It is the largest segment of memory used by a program, spanning almost 2 GB of the address space.

Data in this segment are stored in a stack and a heap. These two data structures are covered in the ADS module.

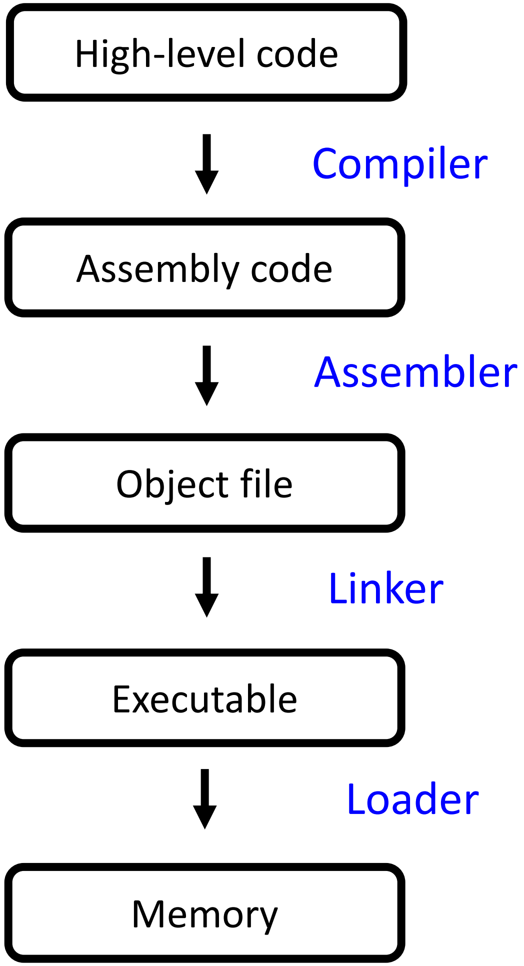
The reserved segments



The reserved segments are used by the operating system and cannot directly be used by the program.

Translating a program
from a high-level language
into machine language
and starting executing it

Translating and starting a program



The compiler translates the high-level code into assembly language.

The assembler turns the assembly code into machine language code (object file).

The linker combines the object file with other machine language code (e.g. from already compiled and assembled libraries).

The loader puts the executable into the memory.

Assembler

```
main:   addi  $sp, $sp, -4
        sw    $ra, 0($sp)
        addi  $a0, $0, 2
        sw    $a0, f
        addi  $a0, $0, 3
        sw    $sw, $a1, g
        jal   sum
        sw    $v0, y
        lw    $ra, 0($sp)
        addi  $sp, $sp, 4
        jr    $ra
sum:    add   $v0, $a0, $a1
        jr    $ra
```

The assembler makes two passes through the assembly code and turns it into the object file.

On the first pass, the assembler assigns instruction addresses and finds all symbols, such as labels and global variable names and makes a symbol table.

In this example, symbols are shown in red.

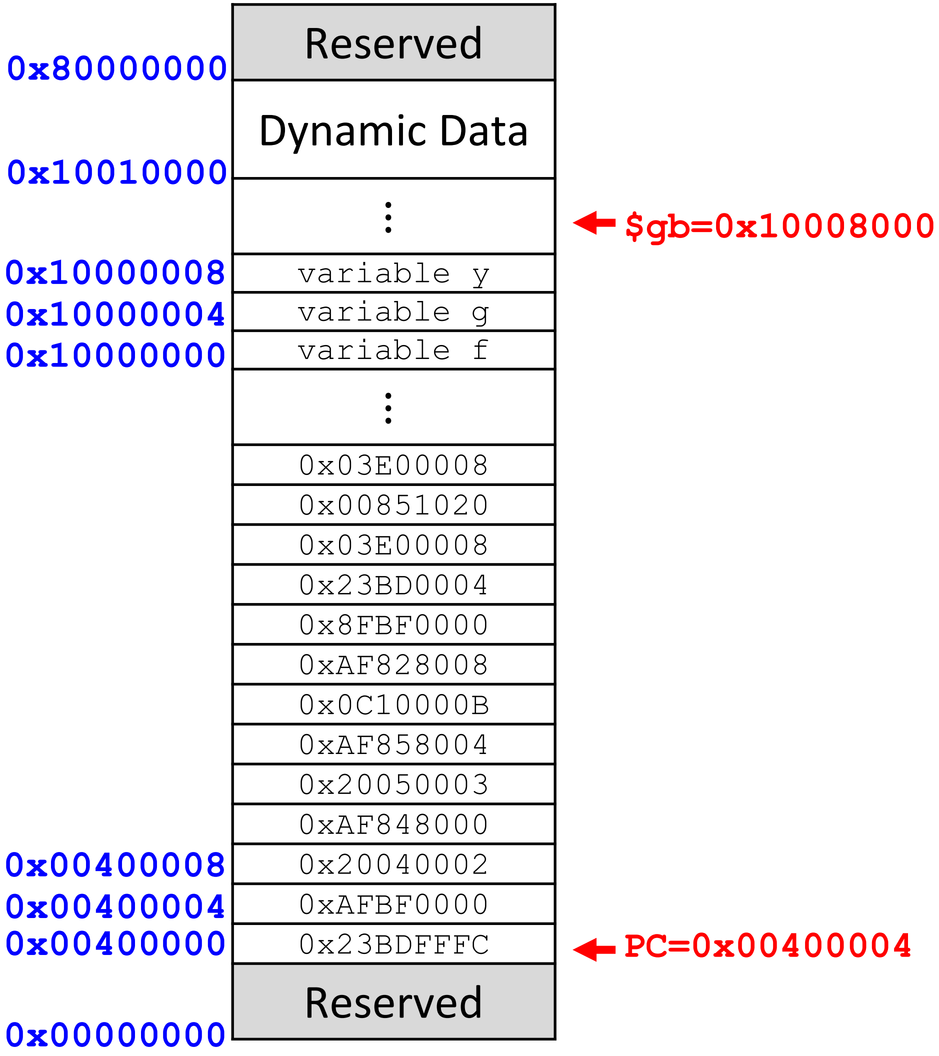
Assembler

```
0x00400000 main: addi  $sp, $sp, -4
0x00400004       sw    $ra, 0($sp)
0x00400008       addi  $a0, $0, 2
0x0040000C       sw    $a0, f
0x00400010       addi  $a0, $0, 3
0x00400014       sw    $sw, $a1, g
0x00400018       jal   sum
0x0040001C       sw    $v0, y
0x00400020       lw    $ra, 0($sp)
0x00400024       addi  $sp, $sp, 4
0x00400028        jr    $ra
0x0040002C sum:  add   $v0, $a0, $a1
0x00400030        jr    $ra
```

symbol	address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

After the first pass of the assembler, instruction addresses have been assigned and the symbol table has been created.

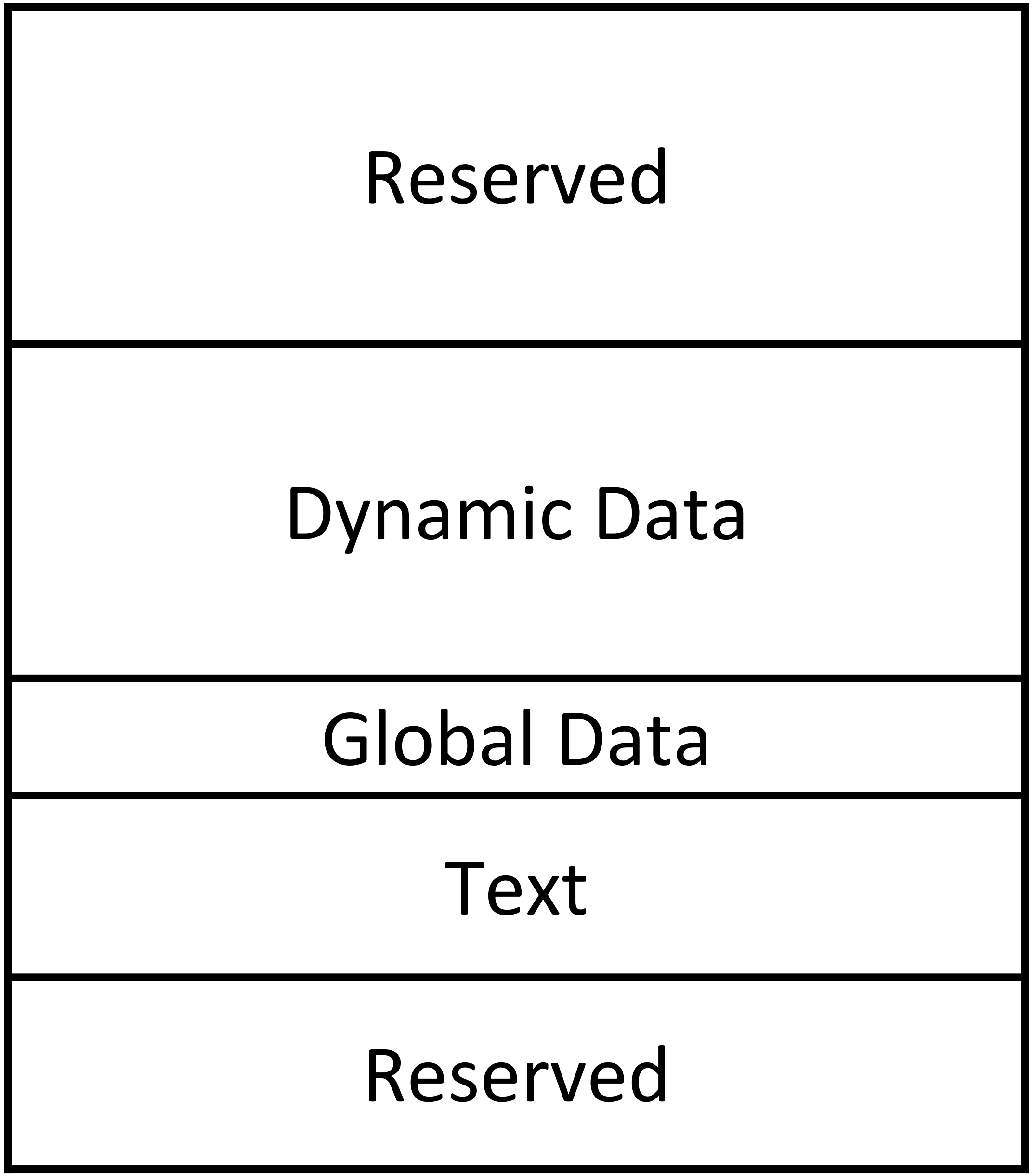
Assembler, linker and loader



The second pass of the assembler generates the object file.

The linker creates the executable by combining the object file with other machine language code, e.g. code corresponding to libraries.

The loader puts the executable into the memory and its execution can start.



Reserved

Dynamic Data

Global Data

Text

Reserved