MIPS ASSEMBLY PROGRAMMING LANGUAGE PART V

Ayman Hajja, PhD

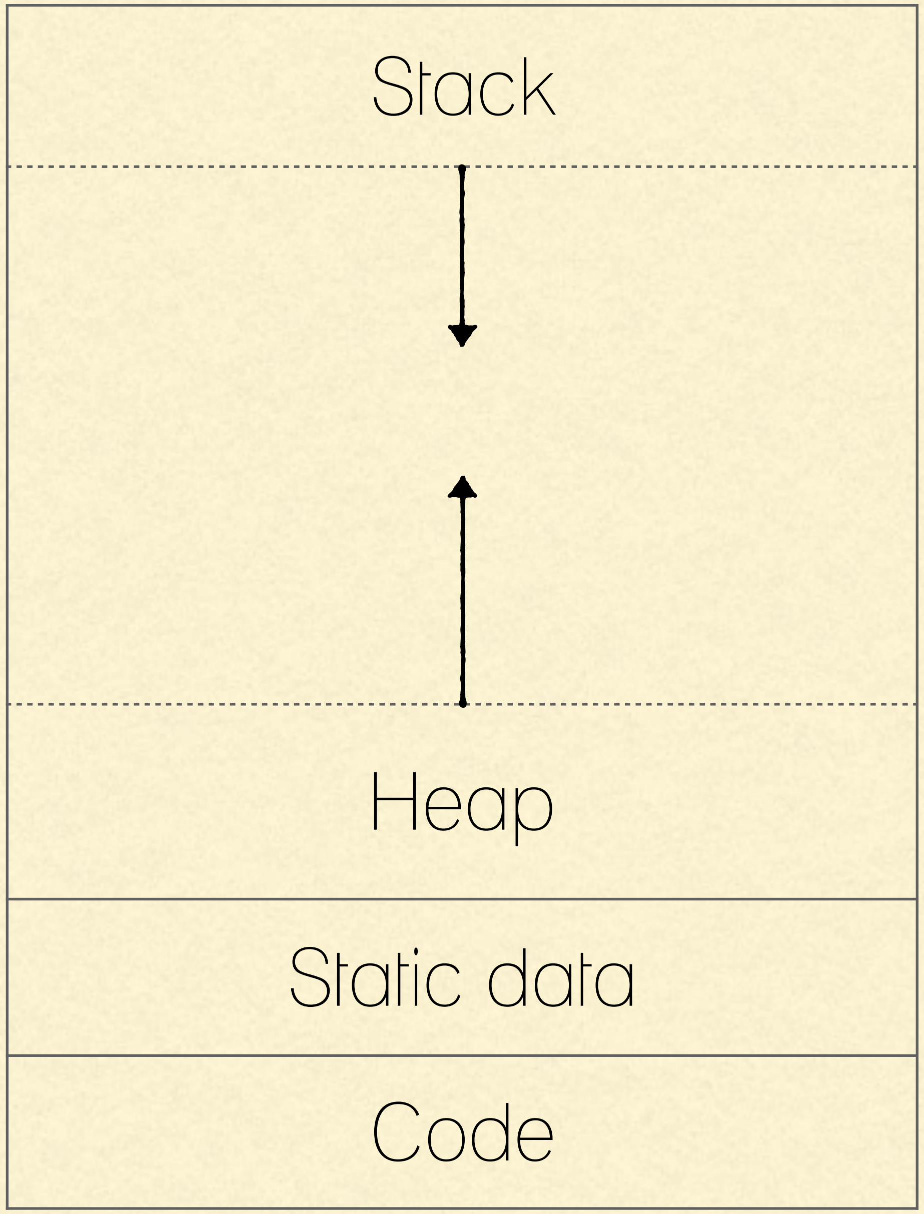
SYSCALL. EXITING PROGRAMS

- The 'syscall' instruction suspends the execution of your program and transfers control to the operating system.
- The operating system then looks at the contents of register $v0 to determine what it is that your program is asking it to do.
- Here's the proper way of telling MARS to stop executing your program:

```
addi $v0, $zero, 10
syscall
```

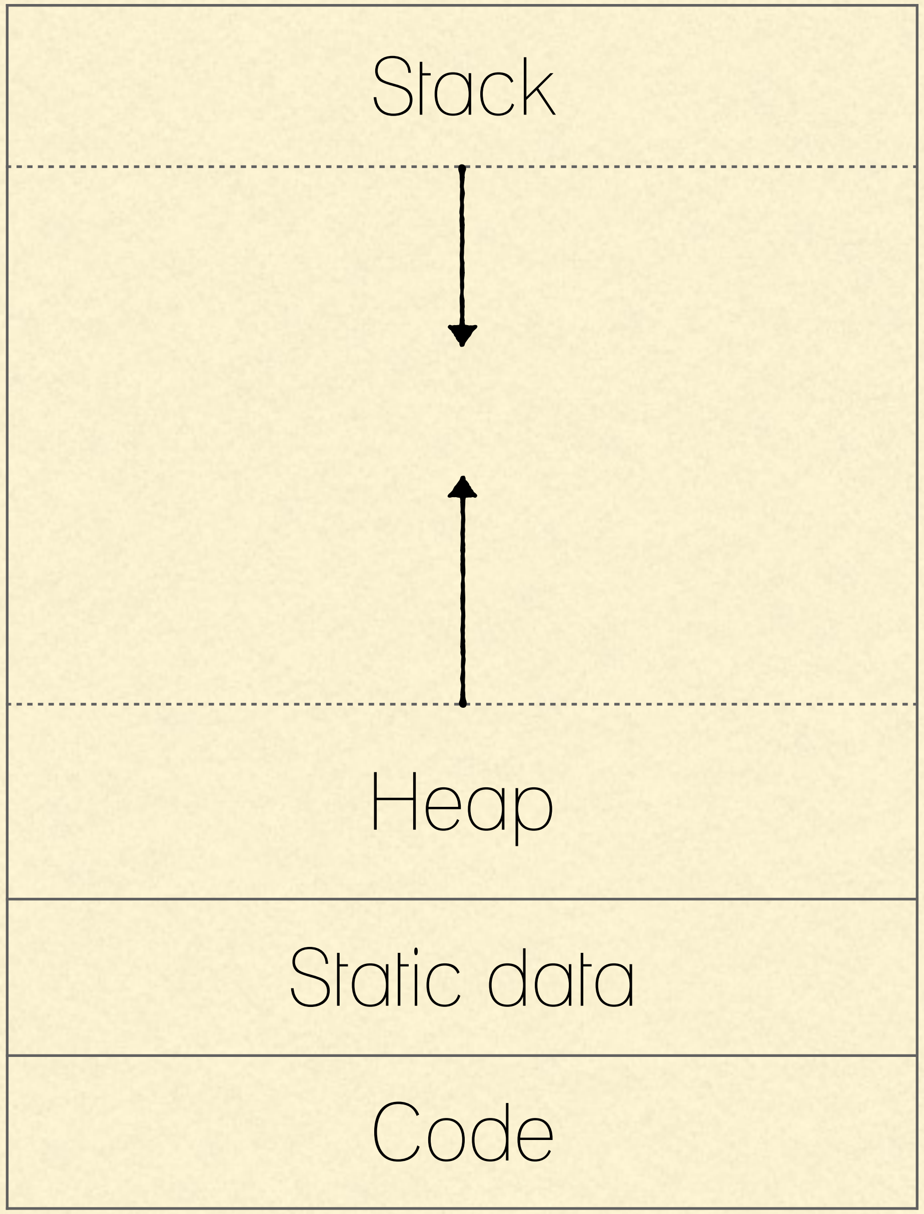
C/MIPS MEMORY MANAGEMENT

- Programs's address space contains 4 regions;
  - Stack; local variables inside functions, grows downward
  - Heap; space requested for dynamic data
  - Static data; variables declared outside functions, does not grow or shrink
  - Code; loaded when program starts — does not change



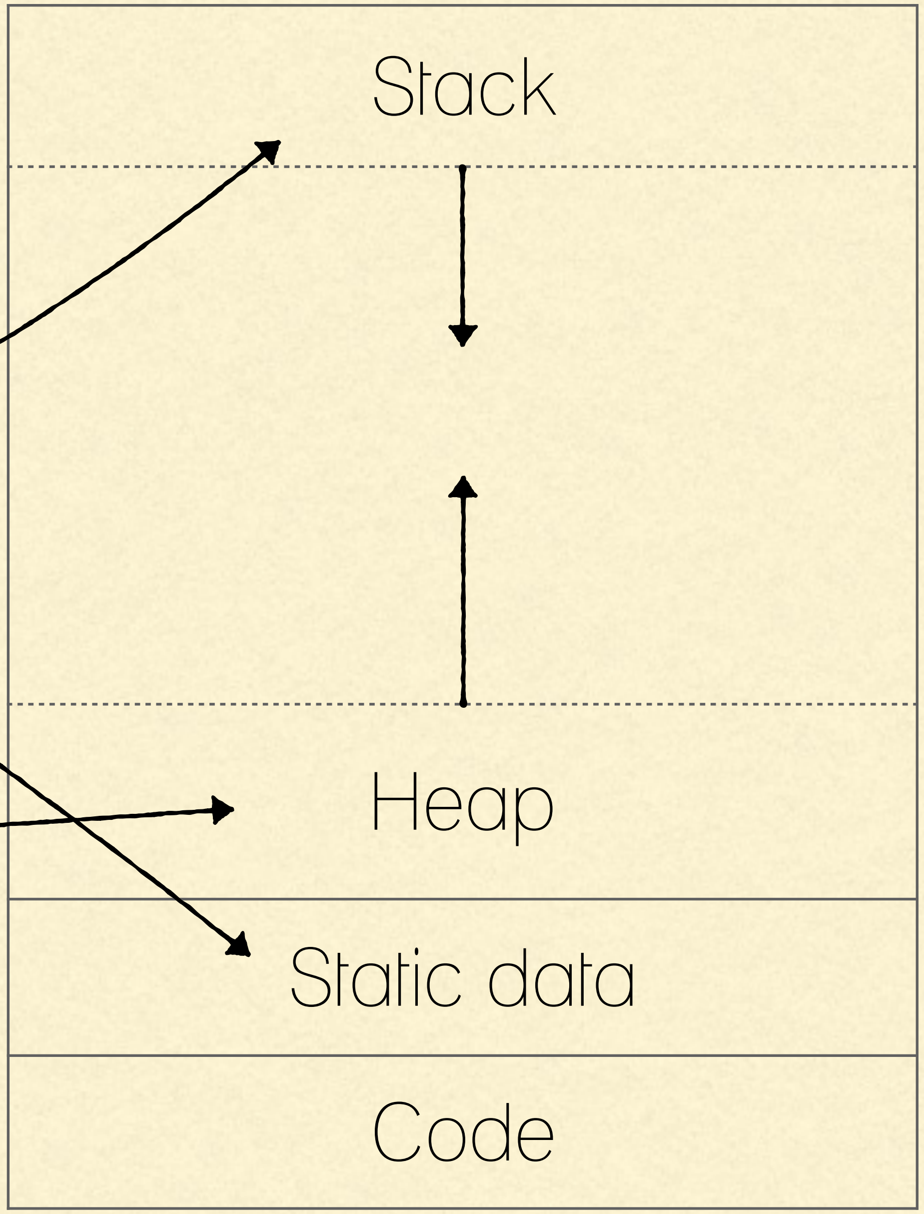
C/MIPS MEMORY MANAGEMENT

```
int arr_global[2];

int main()

{

    int arr_local[2];

    int *arr_dyn = malloc(sizeof(int) * 2);

}
```



C/MIPS MEMORY MANAGEMENT

```
int arr_global[2];

int main()

{

    int arr_local[2];

    int *arr_dyn = malloc(sizeof(int) * 2);

}
```



MIPS ASSEMBLER DIRECTIVES

- Directives are commands that are part of the assembler syntax but are not related to the instruction set.
    - All assembler directives begin with a period (.)
- .data and .text are both directives to the assembler.
    - .data tells the assembler that the upcoming section is considered data (static data section, or global data)
    - .text tells the assembler that the upcoming section is considered assembly language instructions

DATA DECLARATION

- A single declaration consists of:
  1. a label (identifier), followed by a colon. Each label corresponds to a unique address in memory, which the assembler determines.
  2. a storage type. MIPS has a weak sense of type, but it's the closest term to describe what's going on
  3. data values. Data values must be of the correct type (see next slide; 'var1' is numeric variable, 'array1' is an array of characters)

DATA DECLARATION; EXAMPLES

- var1: .word 3
    - Create a single integer variable and assign it the value 3
- array1: .byte 'a', 'b'
    - Create a 2-element character array
- array2: .space 40
    - Allocate 40 consecutive bytes, with store uninitialized (could be used as a 40-element character array, or as a 10-element integer array

ALLOCATING SPACE IN STATIC DATA

```
int arr[] = [2, 5, 10];

void main()

{

    What's here?

}
```

```
.data                               # Static data section
    arr: .word 2, 5, 10             # Space for 3 words
.text                               # Instruction section
    addi $s0, $zero, 4
    addi $s1, $zero, 10
    sw $s0, arr($zero)
    addi $t0, $zero, 4
    sw $s1, arr($t0)
```

ALLOCATING SPACE IN STATIC DATA

```
int arr[] = [2, 5, 10];

void main()

{

    arr[0] = 4;

    arr[1] = 10;

}
```

```
.data                           # Static data section
    arr: .word 2, 5, 10         # Space for 3 words
.text                           # Instruction section
    addi $s0, $zero, 4
    addi $s1, $zero, 10
    sw $s0, arr($zero)
    addi $t0, $zero, 4
    sw $s1, arr($t0)
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS FUNCTION CALL CONVENTIONS

1. Registers faster than memory, so use them
2. \$a0 to \$a3; four argument registers to pass parameters
3. \$v0–\$v1; two value registers to return values
4. \$ra; one return address register to return to the point of origin

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996
1000
1004
1008
1012
1016
...
2000
2004
2008
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996    main:
1000    ...
1004    ...
1008    ...
1012    ...
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996    main:
1000    ...
1004    ...
1008    ...
1012    ...
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

```
Address (shown in decimal)
0996    main:
1000    add $a0, $s0, $zero     # $a0 = $s0 (a)
1004    add $a1, $s1, $zero     # $a1 = $s1 (b)
1008    ...
1012    ...
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996    main:
1000    add $a0, $s0, $zero        # $a0 = $s0 (a)
1004    add $a1, $s1, $zero        # $a1 = $s1 (b)
1008    ...
1012    ...
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

```
Address (shown in decimal)
0996    main:
1000    add $a0, $s0, $zero        # $a0 = $s0 (a)
1004    add $a1, $s1, $zero        # $a1 = $s1 (b)
1008    ...
1012    j sum                      # jump to sum
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. <u>Acquire (local) storage resources needed for function</u>
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

```
Address (shown in decimal)
0996    main:
1000    add $a0, $s0, $zero        # $a0 = $s0 (a)
1004    add $a1, $s1, $zero        # $a1 = $s1 (b)
1008    ...
1012    j sum                      # jump to sum
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. <u>Perform desired task of the function</u>
5. <u>Put result value in a place where calling program can access it and restore any registers you used</u>
6. Return control to point of origin, since a function can be called from several points in a program

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996    main:
1000    add $a0, $s0, $zero         # $a0 = $s0 (a)
1004    add $a1, $s1, $zero         # $a1 = $s1 (b)
1008    ...
1012    j sum                       # jump to sum
1016    ...
...
2000    sum:
2004    ...
2008    ...
```

INSTRUCTION SUPPORT FOR FUNCTIONS

In main(), we call sum(a, b); /* a in $s0, b in $s1 */

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

Address (shown in decimal)

```
0996    main:
1000    add $a0, $s0, $zero        # $a0 = $s0 (a)
1004    add $a1, $s1, $zero        # $a1 = $s1 (b)
1008    ...
1012    j sum                      # jump to sum
1016
...
2000    sum:
2004    add $v0, $a0, $a1
2008    ...
```

CALLING A FUNCTION

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. <u>Return control to point of origin, since a function can be called from several points in a program</u>

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
        return x + y;
}
```

Address (shown in decimal)

```
0996    main:
1000    add $a0, $s0, $zero         # $a0 = $s0 (a)
1004    add $a1, $s1, $zero         # $a1 = $s1 (b)
1008    ...
1012    j sum                       # jump to sum
1016
...
2000    sum:
2004    add $v0, $a0, $a1
2008    ...
```

INSTRUCTION SUPPORT FOR FUNCTIONS

```
In main(), we call sum(a, b);  /* a in $s0, b in $s1 */
int sum(int x, int y)
{
    return x + y;
}
```

```
Address (shown in decimal)
0996    main:
1000    add $a0, $s0, $zero      # $a0 = $s0 (a)
1004    add $a1, $s1, $zero      # $a1 = $s1 (b)
1008    addi $ra, $zero, 1016
1012    j sum                    # jump to sum
1016
...
2000    sum:
2004    add $v0, $a0, $a1
2008    jr $ra
```

INSTRUCTION SUPPORT FOR FUNCTIONS

In main(), we call sum(a, b); /* a in $s0, b in $s1 */

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

> Question;
> Why use jr here? Why not use j (num)?

Address (shown in decimal)

```
0996    main:
1000    add $a0, $s0, $zero        # $a0 = $s0 (a)
1004    add $a1, $s1, $zero        # $a1 = $s1 (b)
1008    addi $ra, $zero, 1016
1012    j sum                      # jump to sum
1016
...
2000    sum:
2004    add $v0, $a0, $a1
2008    jr $ra
```

INSTRUCTION SUPPORT FOR FUNCTIONS

- Single instruction to jump and save return address:
  - jump and link (jal)
  - Before:
    - 1008 addi $ra, $zero, 1016 #$ra=1016
    - 1012 j sum #goto sum
    - 1016 …
  - Now:
    - 1008 jal sum # $ra=1012, goto sum

INSTRUCTION SUPPORT FOR FUNCTIONS

- Single instruction to jump and save return address:
  - jump and link (jal)
  - Before:
    - 1008 addi \$ra, \$zero, 1016 #\$ra=1016
    - 1012 j sum #goto sum
    - 1016 ...
  - Now:
    - 1008 jal sum # \$ra=1012, goto sum

Why 1012 not 1016?