CSC34300
Lecture 06: MIPS Assembly and MARS Simulator

Instructor: Zheng Peng

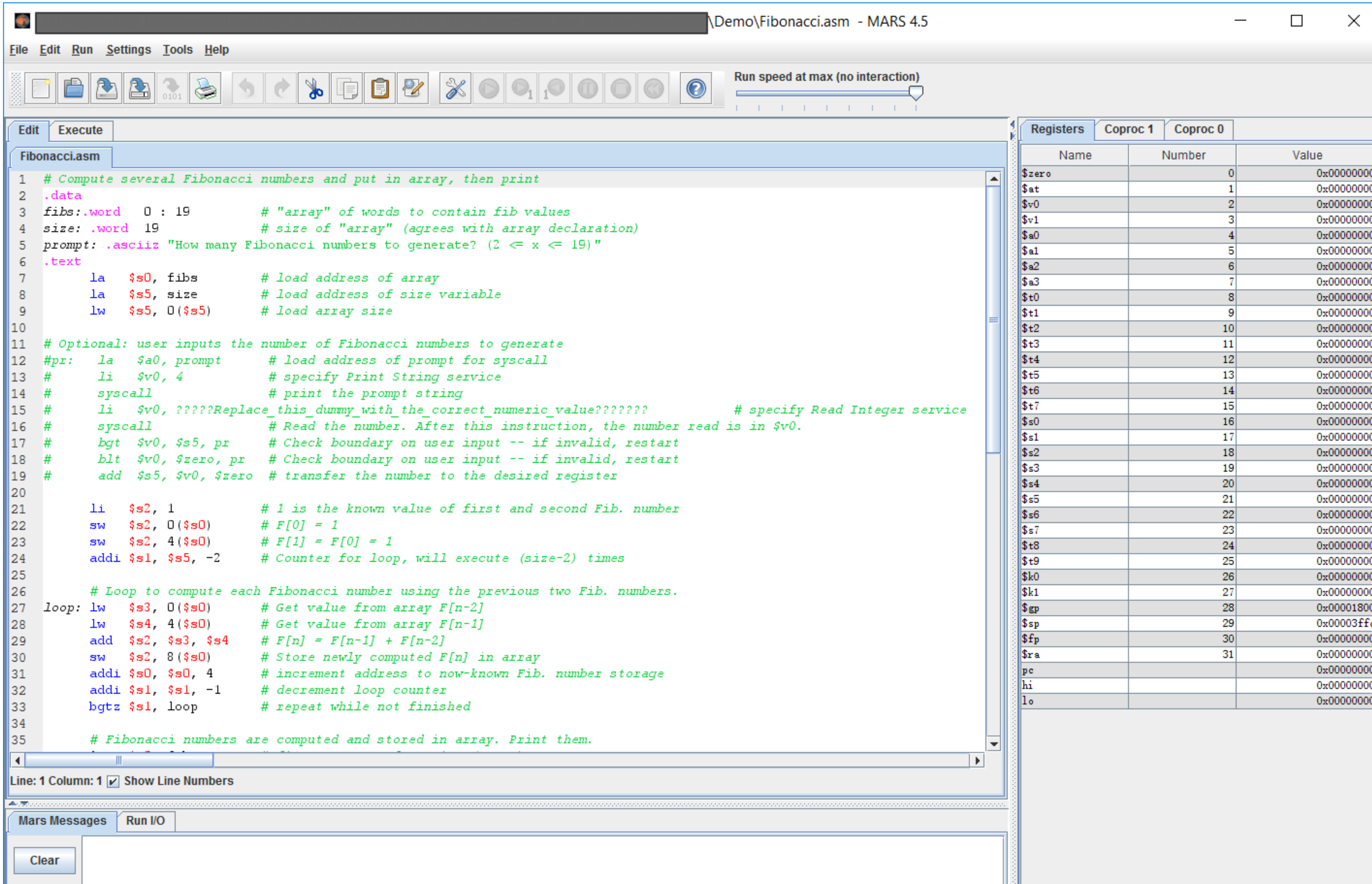
Computer Science Department

City College of New York

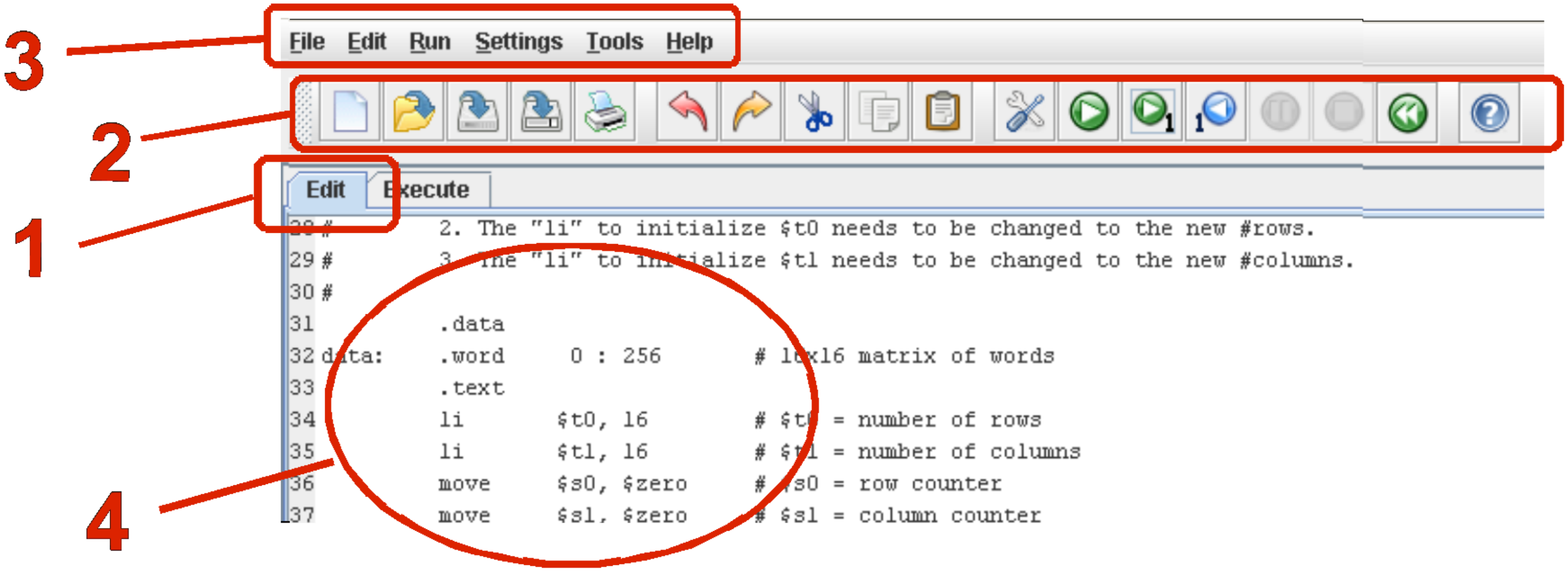
MARS

- **M**ips **A**ssembly and **R**untime **S**imulator
- A lightweight interactive development environment (IDE) for programming in MIPS assembly language
- Written in Java and requires Java Runtime Environment (JRE)
- Intended for educational user
- An open-source software
- http://courses.missouristate.edu/KenVollmar/mars/index.htm

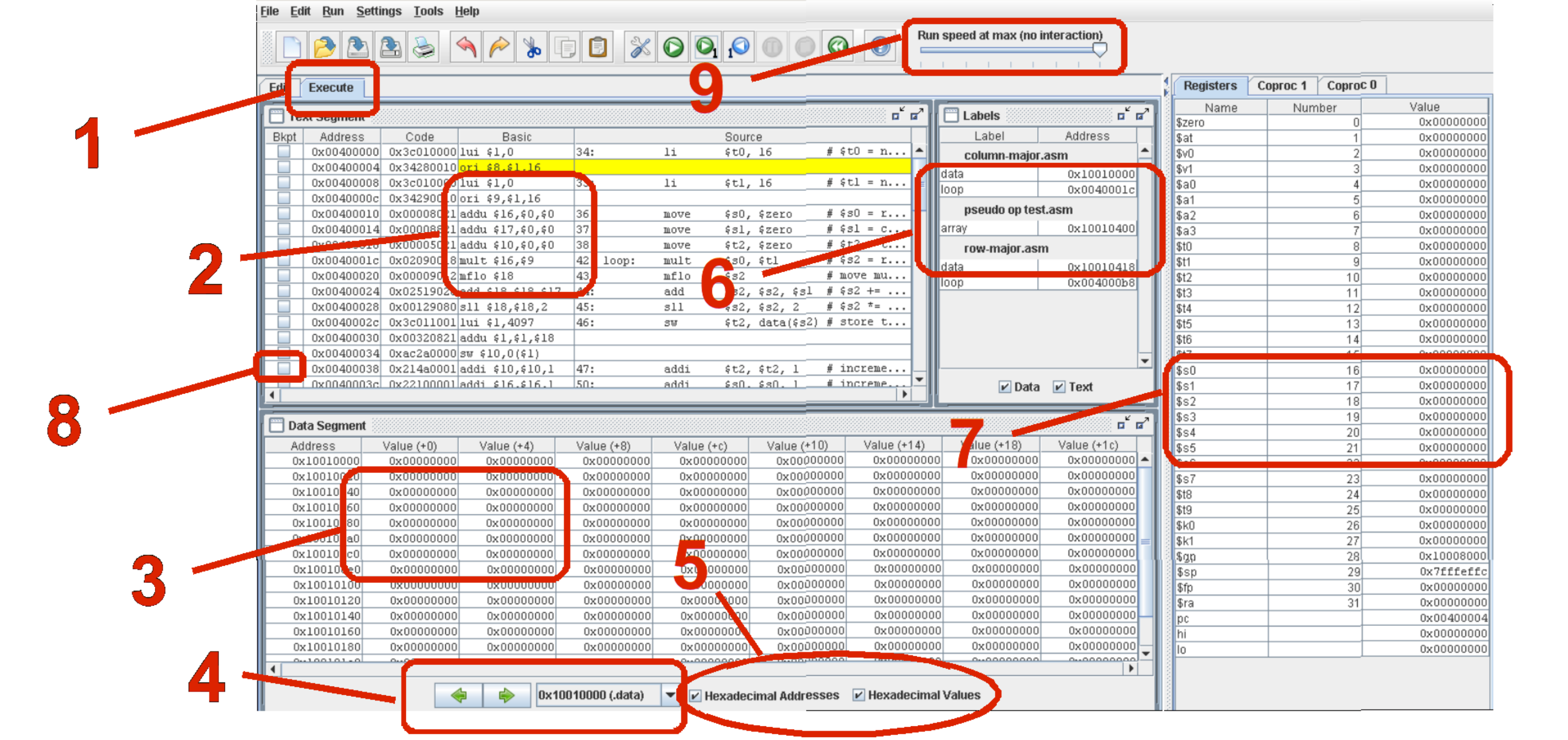
MARS Graphical User Interface



MARS Main and Toolbar



1. Edit display is indicated by highlighted tab.
2, 3. Typical edit and execute operations are available through icons and menus, dimmed-out when unavailable or not applicable.
4. WYSIWYG editor for MIPS assembly language code.



1. Execute display is indicated by highlighted tab.
2. Assembly code is displayed with its address, machine code, assembly code, and the corresponding line from the source code file. (Source code and assembly code will differ when pseudoinstructions have been used.)
3. The values stored in Memory are directly editable (similar to a spreadsheet).
4. The window onto the Memory display is controlled in several ways: previous/next arrows and a menu of common locations (e.g., top of stack).
5. The numeric base used for the display of data values and addresses (memory and registers) is selectable between decimal and hexadecimal.
6. Addresses of labels and data declarations are available. Typically, these are used only when single-stepping to verify that an address is as expected.
7. The values stored in Registers are directly editable (similar to a spreadsheet).
8. Breakpoints are set by a checkbox for each assembly instruction. These checkboxes are always displayed and available.
9. Selectable speed of execution allows the user to “watch the action” instead of the assembly program finishing directly.

Register List

- The right panel shows a list of registers
  - $zero - always has zero value, READ ONLY
  - $at - register is reserved for the assembler, DO NOT USE
  - $t0-$t9 - temporary registers, not preserved by subprograms
  - $s0-$s7 - general purpose registers, preserved by subprograms
  - $k0-k1 - Reserved for kernel, DO NOT USE
  - $sp - the stack pointer
  - $fp - the frame pointer
  - $ra - the return address
  - pc - program counter

MIPS “Hello World!”

```
# MIPS "Hello World!" example

        .data               # Data segment
msg:    .asciiz "\nHello, World!\n"

        .text               # Code segment
        .globl  main        # declare main to be global
main:
        li $v0, 4           # select a system call to print a string
        la $a0, msg         # load address of string to be printed into $a0
        syscall             # make the system call to perform operation

        li $v0, 10          # terminate program
        syscall
```

MIPS Directives*

- Assembler **directives** are instructions that direct the assembler to do something.
- Directives do many things: some tell the assembler to set aside space for variables, others tell the assembler to include additional source files, and others establish the start address for your program.

.align n	Align the next datum on a 2^n byte boundary.
.ascii str	Store the string in memory, but do not null-terminate it.
.asciiz str	Store the string in memory and null-terminate it.
.data <addr>	The following data items should be stored in the data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.
.globl sym	Declare that symbol sym is global and can be referenced from other files.
.space n	Allocate *n* bytes of space in the current segment
.text <addr>	The next items are put in the user text segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

*Reference: http://students.cs.tamu.edu/tanzir/csce350/reference/assembler_dir.html

SYSCALL functions in MARS (1)

- A number of system services, mainly for input and output, are available for use by your MIPS program.
- How to use SYSCALL system services
  - Step 1. Load the service number in register $v0.
  - Step 2. Load argument values, if any, in $a0, $a1, $a2, etc.
  - Step 3. Issue the SYSCALL instruction.
  - Step 4. Retrieve return values, if any, from result registers as specified.

All information are from:
http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

SYSCALL functions in MARS (2)

Service	$v0	Arguments	Result
print integer	1	$a0 = integer to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	
terminate execution	10		

Complete list available at:
http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

SYSCALL functions in MARS (3)

- Example:
  - Display the value stored in $t0 on the console

```
li $v0, 1             # service 1 is print integer
add $a0, $t0, $zero   # load desired value into argument
                      # register $a0, using pseudo-op
syscall               # make the system call
```

  - Read a string from the console

```
        .data
str:    .byte 20

        .text
        .globl main
main:
        la   $a0, str     # $a0, buffer address
        li   $a1, 20      # $a1, size of the buffer
        li   $v0, 8       # read a string
        syscall

        la   $a0, str     # $a0, buffer address
        li   $v0, 4       # print a string
        syscall

        li   $v0, 10      # terminate program
        syscall
```

Another MIPS Programming Example

- Generating Fibonacci Numbers:
  - Fibonacci number: every number after the first two is the sum of the two preceding ones: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
  - $F_n = F_{n-1} + F_{n-2}$
  - In a HLL the such a function might resemble the following

```
void PrintTwelveFibonacci()
{
    int Fibonacci[9];
    Fibonacci[0] = 1;
    Fibonacci[1] = 1;

    for (int k = 2; k <= 9; k++)
        Fibonacci[k] = Fibonacci[k - 1] + Fibonacci[k - 2];

    for (int k = 0; k <= 9; k++)
        print(Fibonacci[k]);
}
```

Another MIPS Programming Example

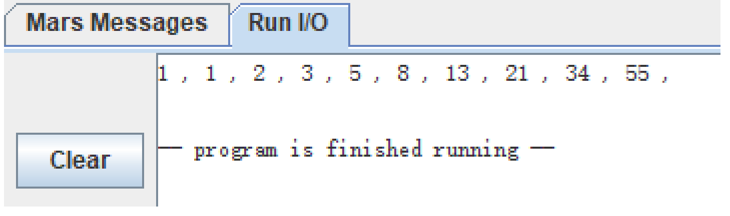
```
        .data               # data segment
nl:     .asciiz "\n"        # ASCII for a new line
        .align 2            # aligned at word boundary
comma:  .asciiz " , "       # note the space
        .align 2

        .text               # Code segment
        .globl  main        # declare main to be global
main:   # $t2: F(n-2), $t1: F(n-1), $t0: F(n)
        li      $t2,0           # $t2=0; initial value of F(n-2)
                                # in this case, F(0)
        li      $t1,1           # $t1=1; initial value of F(n-1)
                                # in this case, F(1)
        # -----------------
        move    $a0,$t1         # $a0= $t1, which is F(1)
        li      $v0,1           # system call, type 1, i.e. integer
        syscall                 # make system call to print the value of F(1)
        # -----------------
        la      $a0,comma       # $a0 = address of comma
        li      $v0,4           # system call, type 4, i.e. string
        syscall                 # make system call to print string
        # -----------------
        li      $t3,9           # $t3 is the counter to be decremented
```

Another MIPS Programming Example

```
loop:
            add     $t0,$t1,$t2 # F(n) = F(n-1) + F(n-2)
            # ----------------
            move    $a0,$t0     # $a0= $t0, which is F(n)
            li      $v0,1       # system call, type 1, i.e. integer
            syscall             # print the value of F(n)
            # ----------------
            la      $a0,comma   # $a0 = address of comma
            li      $v0,4       # system call, type 4, i.e. string
            syscall             # make system call to print string
            # ----------------
            addi    $t4,$t4,1   # $t4 = $t4 + 1; increment n
            move    $t2,$t1     # $t2 = $t1; previous F(n-1) becomes F(n-2)
            move    $t1,$t0     # $t1 = $t0; previous F(n)   becomes F(n-1)
            addi    $t3,$t3,-1  # $t3 = $t3 - 1; decrement the counter
            bne     $0,$t3,loop # continue if $t3 is not 0
Exit:       # ----------------
            la      $a0,nl      # $a0= address of "nl"
            li      $v0,4       # system call, type 4, print an string
            syscall             # print a newline
            # ----------------
            li      $v0,10      # System call, type 10, standard exit
            syscall             # ...and call the OS
```

Another MIPS Programming Example

- To run the program, press F3 and then F5 in MARS
- Then the output, if any, will show up at the bottom window of the MARS simulator



Assembling the code

- Once this code is loaded into MARS we can assemble it … click Run > Assemble or press F3
  - Our window will change to display the Text and Data segments

Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	6: la $t0, fibs # load address of array
☐	0x00400004	0x34280000	ori $8,$1,0x00000000	
☐	0x00400008	0x3c011001	lui $1,0x00001001	7: la $t5, size # load address of size variable
☐	0x0040000c	0x342d0030	ori $13,$1,0x00000030	
☐	0x00400010	0x8dad0000	lw $13,0x00000000($13)	8: lw $t5, 0($t5) # load array size
☐	0x00400014	0x240a0001	addiu $10,$0,0x0000...	9: li $t2, 1 # 1 is first and second Fib. number
☐	0x00400018	0x46241000	add.d $f0,$f2,$f4	10: add.d $f0, $f2, $f4
☐	0x0040001c	0xad0a0000	sw $10,0x00000000($8)	11: sw $t2, 0($t0) # F[0] = 1
☐	0x00400020	0xad0a0004	sw $10,0x00000004($8)	12: sw $t2, 4($t0) # F[1] = F[0] = 1
☐	0x00400024	0x21a9fffe	addi $9,$13,0xfffffffe	13: addi $t1, $t5, -2 # Counter for loop, will execute (size-...
☐	0x00400028	0x8d0b0000	lw $11,0x00000000($8)	14: loop: lw $t3, 0($t0) # Get value from array F[n]

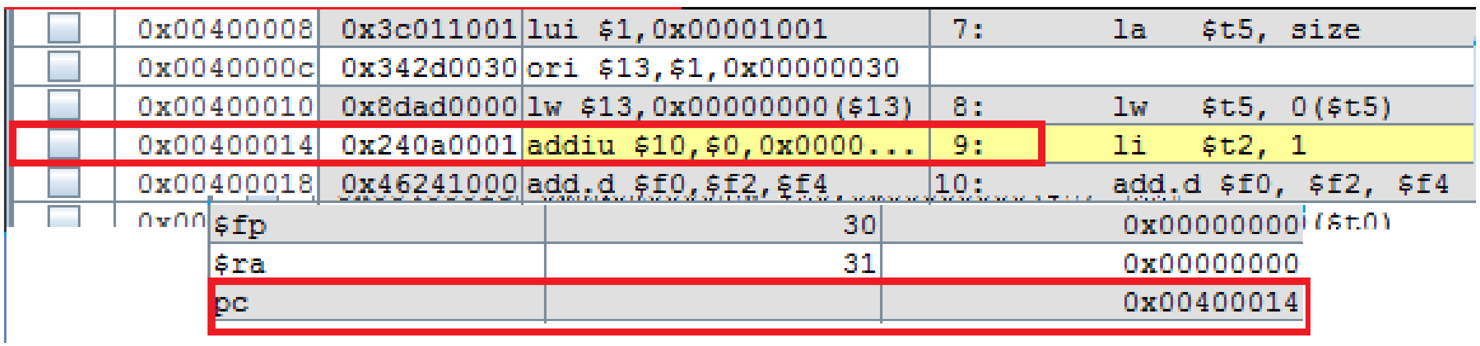
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000c	0x68540020	0x69462065	0x616e6f62
0x10010040	0x20696363	0x626d756e	0x20737265	0x3a657261	0x0000000a	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

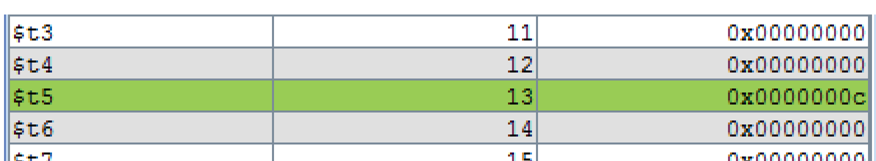
0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Execution – Single Step

- We can click Run > Step or press F7 to single step through the code
- As you step through the code you will notice the line you are about to execute is highlighted in yellow, an the value of the program counter matches the address value of that line

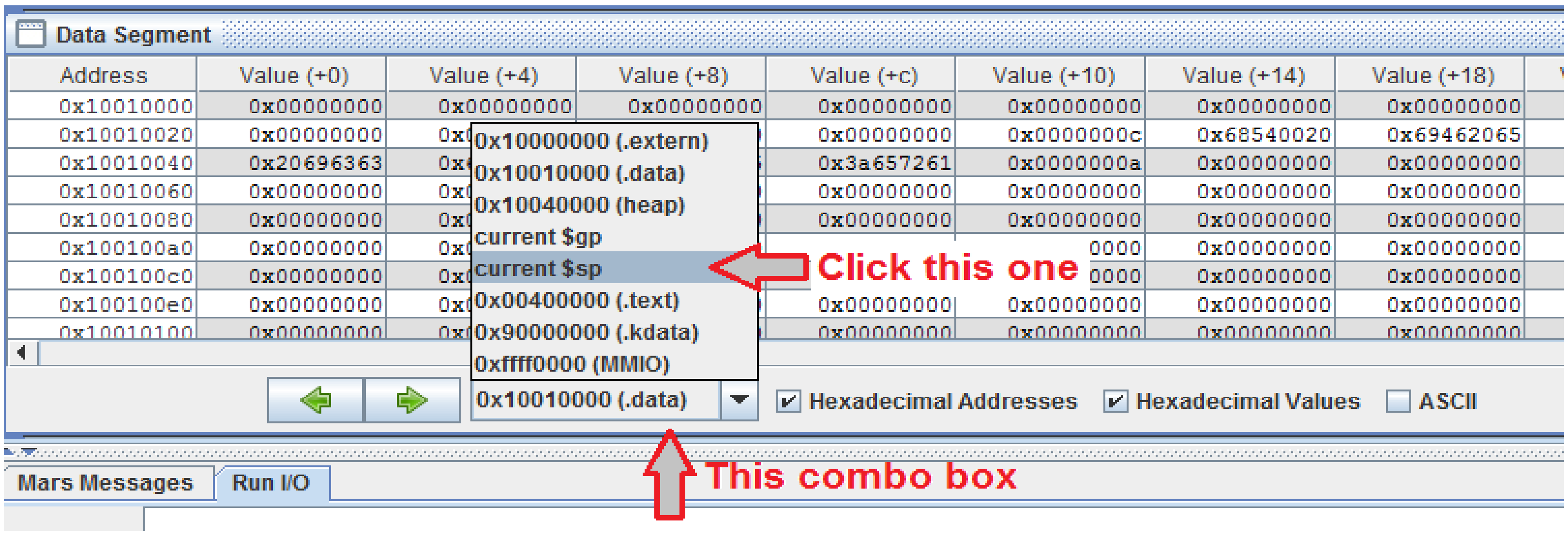


- You may have also noticed in the registers window registers are being highlighted in green, this is to indicate the contents of that register have changed



Execution – Viewing the Stack

- To see the data currently on the stack, click the combo box in the Data Segment window, and change the value to current $sp



- This fibonacci.asm does not use the stack at all, so the view will revert back to data as you step through the code. So we need to rewrite the code so that it incorporates the stack – or you can download it from the course website