Additional Sheet #1:

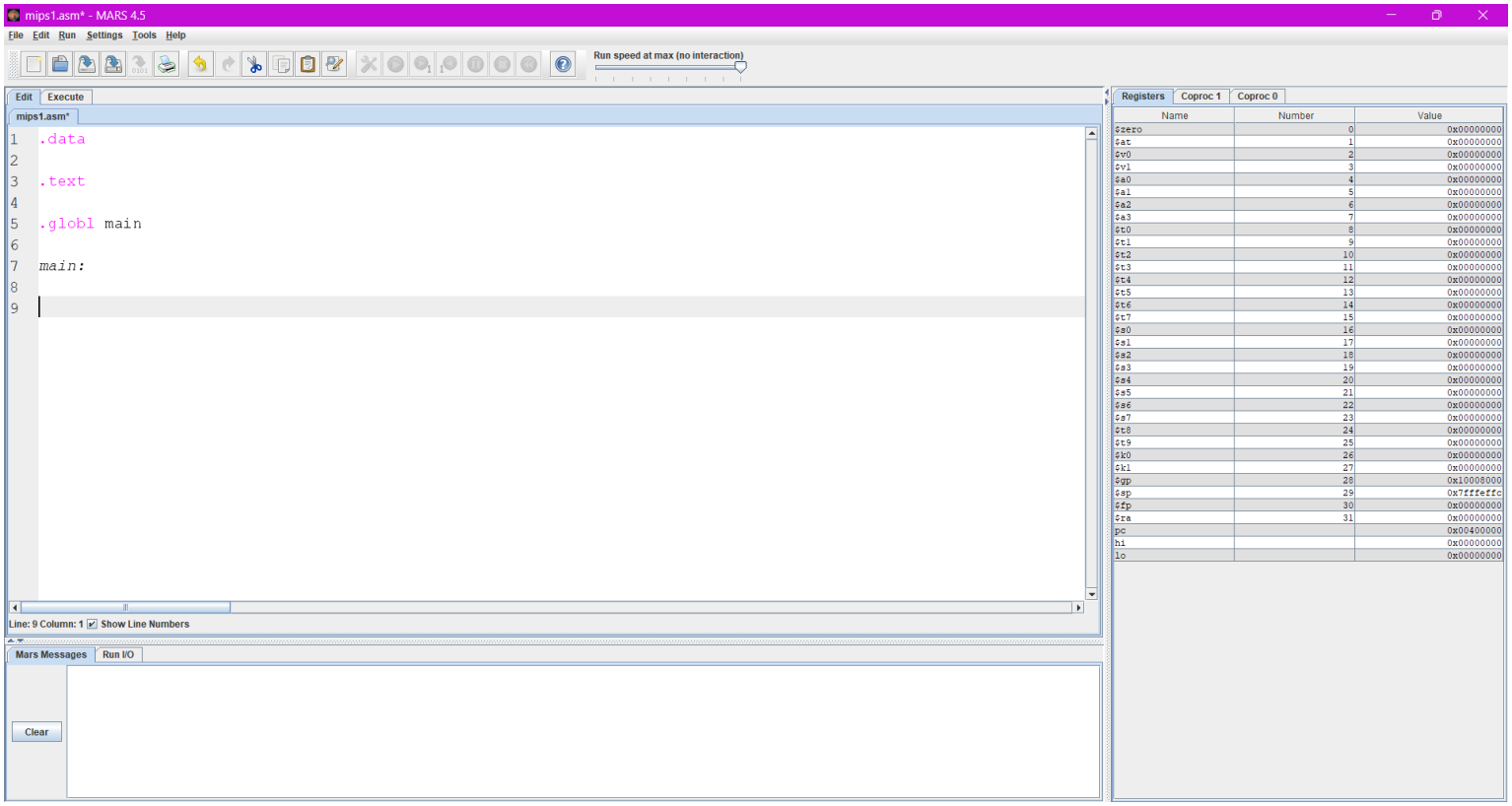
Let's begin with a small introduction:

Part 1: MIPS STRUCTURE

We have 3 main parts:

1) Data
2) Test
3) Main (inside the text)

Written in the following format:



Part 2: data

Syntax:

to declare a data variable in MIPS, we use the following format:

<label>: .<data type> <value>

In mars, we have many data types such as:

Data type	Description	Example	Explanation
word	Stores 32-bit data, which means it can store values from -2^31 to 2^31 – 1, mostly used for integers	wrd .word 100	wrd is an integer holding the value 100
half	Stores 16-bit data, which means that it can store values from -32,768 to 32,767, used for short integers	hlf .half 200	hlf is storing an integer of value 200
byte	Stores 8-bit data, which means that it can store values from 0 to 255, used for characters	bt .byte 'A'	bt stores the character A
ascii	Stores strings, not null terminated	str .ascii "hi"	str is a string that holds "hi", it makes some problems while printing
asciiz	Stores strings, null terminated (ends internally with \0)	s2 .asciiz "hello"	str is a string that holds "hi"
float	Stores 32-bit data, used for numbers with fractional parts	flt .float 3.51	flt is a float number with the value 3.51 (in java we call it double)
double	Stores 64 bit data, used as float is used but for more precision for big fractional numbers	dbl .double 3.1255262298745	It works the same as float, the difference is that it can fit more digits.
space	Space is a reserved word in MIPS that is used to reserve space in memory for data	spc .space 100	spc stores 100 bytes in memory, it can be used as initializing an empty array.

Let's write each example in the data and try to print them, the code now will look like this: (if you are copy pasting, rewrite the quotations)

Edit Execute

mips1.asm

```
1   .data
2   wrd: .word 100
3   hlf: .half 200
4   bt: .byte 'A'
5   str: .ascii "hi"
6   s2: .asciiz "hello"
7   flt: .float 3.51
8   dbl: .double 3.1255262298745
9   spc: .space 100
10
11  .text
12
13  .globl main
14
15  main:
16
```

Now, we will try and print them, unlike java and C++, each variable has a specific way to be printed.

For that, we have a table that will help us to print the variables that we want.

Note that this table should be always with you (will be provided later on in your exams)

This table includes many registers where each has its specific functionality.

In total, we have 31 registers. Let's see the registers before printing.

Part 3: registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
...		
15	t7	

16	s0	callee saves
...		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address
		caller saves

Part 4: system calls

Our first step is to print the variable we have done before, to do that we will write a code inside our main section.

Printing requires us to use something called “system calls” which begins by the keyword “syscall”

Inside the system calls, we have several functions to call where each function is responsible to print a certain data type. The functions are described as follows:

Service	Code (put in $v0)	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=addr. of string	

Let’s discuss in detail how to print the integer wrd

- Load wrd value inside a register (look at arguments)
- Make sure that 1 is inside the register $v0
- Make the system call in the end

There are a bunch of codes that we have not seen here, so also before printing, let’s go over some instructions.

Part 5: Instructions

Load instructions

Full instruction	Keyword meaning	Instruction description
lw $t0, wrd	Load word	Load the value of wrd in $t0
lh $t0, hlf	Load half	Load the value of hlf in $t0
lb $t0, bt	Load byte	Load the value of bt in $t0
l.s $t0, flt	Load float	Load the value of flt in $t0
l.d $t0, dbl	Load double	Load the value of dbl in $t0
la $t0, wrd	Load address	Load the address of wrd in $t0
li $t0, 1	Load immediate	Load the value 1 in $t0

Store instructions:

Full instruction	Keyword meaning	Instruction description
sw $t0, wrd	Store word	Store the value of $t0 in wrd
sh $t0, hlf	Store half	Store the value of $t0 in hlf
sb $t0, bt	Store byte	Store the value of $t0 in bt
sl $t0, dbl	Store double	Store the value of $t0 in dbl

Move instruction:

Full instruction	Keyword meaning	Instruction description
move $t0, $t1	Move	Copy the value of $t1 in $t0

Part 6: let's code!!

Recall the steps above, let's start with them:

Step 1:

- Load wrd value inside a register

Here we can choose any register we want, but if you look closely to the table, it turns out that the integer should passed as an argument using the $a0 register, so we can in one shot load the value of wrd inside $a0 using the following code:

lw $a0, wrd

this is called direct addressing mode, since the value is actually inside the variable wrd not its address or anything else.

Step 2:

- Make sure that 1 is inside the register $v0

In this step, we want to add directly immediately the value 1 to $v0 without storing the 1 in a variable. (If you are still not sure why we put 1 inside $v0, please go back and look at the table). For this we will write the following code:

li $v0, 1

this is called immediate addressing mode, since we have 1 (or actual value) as an operand in the code (written inside the code without a variable)

Step 3:

- Make the system call in the end

Simply just write on a third line:

syscall

And like that, you have written the whole code :D

```
#print the integer
  lw $a0, wrd
  li $v0, 1
  syscall

#print the half:
  lh $a0, hlf
  li $v0, 1
  syscall

#print asciiz (string):
  la $a0, s2
  li $v0, 4
  syscall

#print float
  l.s $f12, flt
  li $v0, 2
  syscall

#print double
  l.d $f12, dbl
  li $v0, 3
  syscall
```

Part 7: Read input

Service	Code (put in $v0)	Arguments	Result
read_int	5		int in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0=buffer, $a1=length	

To read an integer for example, you should do the following:

- Load 5 inside $v0
- Write the “syscall” keyword.
- Check the value inside $v0 that contains the integer entered by the user.

```
#read integer
  li $v0, 5
  syscall


#read float
  li $v0, 6
  syscall


#print float after reading
  mov.s $f12, $f0
  li $v0, 2
  syscall


#read double
  li $v0, 7
  syscall


#print double after reading
  mov.d $f12, $f0
```

```
li $v0, 3
syscall
```

We will keep the strings for next time :D

Part 8: Math (until now)

Operation	MIPS code	Explanation
Addition	add $t0, $t1, $t2	Adds the values in registers $t1 and $t2 and stores the result in register $t0.
	addi $t0, $t1, im	Adds the immediate value im to the value in register $t1 and stores the result in register $t0.
Multiplication	mul $t0, $t1, $t2	Multiplies the values in registers $t1 and $t2 and stores the result in register $t0.
	muli $t0, $t1, im	Multiplies the immediate value im by the value in register $t1 and stores the result in register $t0.
Division	div $t0, $t1, $t2	Divides the value in register $t1 by the value in register $t2 and stores the quotient in register $t0.
	divi $t0, $t1, $t2	Divides the value in register $t1 by the immediate value im and stores the quotient in register $t0.

Part 9: Arrays

Let's consider that we have an array of type word containing 2 values: 10 and 20. We just want to print the elements of the array, let's see how to do that:

To make this array write the following:

array: .word 10, 20

This array is stored in data, to use it in our code we should:

1) Load the address of the array
2) Load each word in a temporary register
3) Print the words

Here is the code for this program:

```
la $t0, array
```

```
lw $a0, 0($t0)
li $v0, 1
syscall
lw $a0, 4($t0)
li $v0, 1
syscall
```

This type of loading values is called indirect memory addressing, since we are loading the memory address of the array then accessing each element inside.

Note that we are jumping 4 by 4 not 1 by 2 because the word is 32 bits which is 4 bytes, so each 4 bytes represent an integer, that's why we need to jump by 4 bytes.

If the array is of type half, we should jump by 2 bytes, and if it was of type byte, we jump 1 byte

(the math behind it is number of bites / 8)

Part 10: exit

In MIPS, there is a code to exit the program, for this you should use the following table:

Service	Code (put in $v0)	Arguments	Result
exit	10		

So we are only loading the value 10 inside the $v0 register, and of course you end the code by writing the syscall keyword.

```
li $v0, 10
syscall
```