Lecturer: Simina Fluture,PhD

240 Week 3 – WINTER 23

Topics: **Macro Instructions**
Assembler directives
Input and Output
Reading and Printing Integers
Reading and Printing Strings

<u>Macro Instructions:</u>

- Actual Instructions
  - Each actual instruction represents a MIPS ML (machine language) instruction that the MIPS processor can execute
  - 1 actual instruction is translated to 1 ML instruction
- Pseudo Instructions (We will use pseudo-instrcutions ONLY IF THEY ARE SUPPORTED BY HARDWARE OR COMPILER)
  - E.g. pseudo instruction li $t1, 3 is translated into the ML instruction addi $t1, $zero, 3
  - Does not represent an actual ML instruction
  - The assembler translates each pseudo instruction into 1 or more ML instructions
    - Provided for convenience of programmer
    - Another eg: move $s0, $s1 add $s0, $zero, $s1

Note: SPIM has option of no pseudo instructions

Load Immediate **li $s1, value** Initialize registers with positive constants (<32768)
Load Address **la $s1, Label** Initialize pointers

When writing a program we might need to use assembler directives and labels:
assembler directives tell the assembler how to translate a program but do not produce machine instructions.

They are represented by names that begin with a period, for example
.data
.globl are **assembler directives**

<u>Assembler Directives:</u>
Give a programmer the ability to establish some initial data structures that will be accessed by the computer at the run time.

.text <addr> identifies the **Text segment**.
In SPIM these items may only be instructions or words. If the optional argument *addr* is present, subsequent items are stored starting at the address *addr.* By default, the program starts in the text segment (should be specified by .text directive), although we don't explicitly mention it.

.data <addr> tells the assembler that all the following data allocation directives should allocate data in a portion of memory called the **data segment**.
If the optional argument *addr* is present, subsequent items are stored starting at the address *addr*

Names followed by a colon, such as
str:
main:
are **labels** that name the next memory location.

More examples
Allocating space to various structures:
.space n allocates *n bytes* of space in the data segment.

For example to allocate space in memory for a one-dimensional array of 500 integers:
int ARRAY[500] in MIPS you do:

.data
ARRAY: .space 2000

.space requires that the amount of space must be specified in bytes.

.word initializes a word in main memory
For example to initialize a memory array named Pof2 (from power of two)
.data
Pof2: .word 1,2,4,8,16,32,64

.globl Symb declare that label **Symb** is global so it can be referenced from another file
.data

.asciiz string* store the string in memory and null-terminate it
ASCII null character (NUL) is an 8-bit binary value zero. Print String will stops printing characters when it finds the NUL

.ascii string* store the string in memory and not null-terminate it

Reading and Printing Integers / Strings

syscall stands for system call; asks for the assistance of the Operating System (kernel)

All I/O operations (read, write, input, output) need the assistance of the OS.

	code	
syscall	5	can be used to read an integer into register $v0
syscall	1	can be used to print out an integer stored in register $a0

Service	Code	Arguments	Result
print_int	1	$a0	*none*
print_float	2	$f12	*none*
print_double	3	$f12	*none*
print_string	4	$a0	*none*
read_int	5	*none*	$v0
read_float	6	*none*	$f0
read_double	7	*none*	$f0
read_string	8	$a0 (address), $a1 (length)	*none*
sbrk	9	$a0 (length)	$v0
exit	10	*none*	*none*

```
##      $v0, and $v1 are used to return values from functions
##      $a0 through $a3 are used to pass parameters to functions
```

```
li      $v0, 4          system call code for Print String
li      $v0,1           system call for print Integer
li      $v0,10          terminate program
```

To call the system service to print a string (in our case the one labeled as Prompt)
main:

```
        li      $v0,4           value 4 is loaded into $v0 to specify Print String
        la      $a0, Label      symbolic address of the memory location where the string of char
                                have been stored in memory must be loaded in $a0 ($a0 is a
                                pointer to the string)
        syscall                 system call for printing the string to console
```

The Print String system service will print all the characters found in memory following the first character code until it finds the NULL.

declare a string

```
.data
integer1: .asciiz "\n Please input First negative integer betweeen [-30 and 0) : "
```

exit the program

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

read a String of characters from keyboard

```
main:
        li      $v0, 8
        la      $a0, Buffer     #$a0 is a pointer to an input Buffer
        li      $a1, 60         #specify the maximum buffer length
        syscall                 #read a string and store it into the buffer
```

We will also need an assembler directive (in the source code) to allocate space for the buffer in the memory data segment.
For example: **.space 60**

----
prompt the user for integer1
get integer1
prompt the user for integer2
get integer2
compute integer1 + 8*integer2
display a result message together with the result
exit

Topics: MIPS: Instructions for Making Decisions

<u>if-then-else into conditional branches</u>
C code: if (i == j) f = g + h;
else f = g – h;

f: $s0, g: $s1, h: $s2, i: $s3, j: $s4

Our initial solution:

Textbook solution:

```
bne $s3, $s4, Else          #branch if i!=j
add $s0, $s1, $s2
j Exit
Else:  sub $s0, $s1,$s2
Exit:
```

Usually there is a reversal of the relational operator; it typically occurs when translating pseudo-code to assembly language. Why? The conditional branch instruction transfers control to the *else* code. The code is more efficient if we test for the opposite condition to branch.

Branch Delay: the instruction after the jump or branch instruction starts being executed before the jump or branch condition can be evaluated, and the jump or branch is completely executed.
WHY : MIPS Pipeline

LOOPS

There are three types of loops in most of high level programming languages:

- while
- do … while
- for

Though there are multiple ways of writing a loop in MIPS, **conditional branch is the key to decision making.**

For a ***for loop*** we need the **slt** instruction: set on less than

```
slt     $t0, $s3, $s4          $t0 =1 if $s3 < $s4, $t0=0 otherwise
slti    $t0, $s2, 10           $t0 =1 if $s2 <10

sltu    set on less than unsigned
sltui   set on less than unsigned immediate
```

Example:
if (g < h) goto Less;
Use this mapping: g: $s0, h: $s1

MIPS code
Register $0 always contains the value 0; bne and beq are often used for comparison after a slt instruction.

Example:
C: if (g >= 1) goto Loop;

Example: For Loop

```
Pseudocode: int x;
#           for(x=0;x<10;x++){
#               printf("x=%d", x);
#           }
#
```

Example:
Case / Switch Statement

```
switch(k) {
case 0: f = i + j; break;   /* k = 0 */
case 1: f = g + h; break; /* k = 1 */
case 2: f = g – h; break;  /* k = 2 */
case 3: f = i – j; break;   /* k = 3 */
}
```

Example: C switch statement

Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if (k == 0) f = i + j;
   else if (k == 1) f = g + h;
      else if (k == 2) f = g – h;
          else if (k == 3) f = i – j;
```

Use this mapping: f: $s0, g: $s1, h:$s2, i: $s3, j: $s4, k:$s5

Lecturer: Simina Fluture,PhD

Topics: **ARRAYS**
WHILE LOOP AND ARRAY

Example 1: of using Arrays

```
# Print on one column the values of an array until a zero value is found.
################################################################################
$t0: contains the address of the element
$t1: contains the value of the element
################################################################################

.data
array:  .word 55
        .word 66
        .word 77
        .word 88
        .word 0

newline: .asciiz "\n"



.text
main:

#get the first element of the array;
# we need first to know the address of the element


# compare with zero; if zero then done; else print the element

# print the element (integer)


# go to next line


# update $t0 with the address of the next element;
```

Example 2: While Loop / Array

```
while (save[i] == k)
        i+=1;
```

i → $s3, k → $s5, base of array → $s6

$t0 for the address
t1 for the value

Lecturer: Simina Fluture,PhD

Topic: MIPS: Logical Operations – NOT COVERED IN WINTER 23

Logical Operations

Shift Left
Shift Right
Bit by bit AND / ANDI
Bit by bit OR / ORI
Bit by bit XOR / XORI

SHIFTS

Shift Left sll rd, rt, shamt

sll $t0, $s0,4 reg $t0 = reg $s0 << 4 bits

R-format

Op-code	rs	rt	rd	value	function	
6	5	5	5	5	6	
000000		00000				**000000**

Shift Right srl rd,rt,shamt

srl $t0,$s0,4 reg $t0 = reg$s0 >> 4 bits

R-format

Op-code	rs	rt	rd	value	function	
6	5	5	5	5	6	
000000		00000				**000010**

Problem – example(s): isolate byte1
isolate rs2 field

ANDs : **Mask a field: keep the field unchanged, set the rest to zero.**

Bit by Bit AND

AND and rd, rs, rt

R-format

Op-code	rs	rt	rd	value	function	
6	5	5	5	5	6	
000000					00000	100100

AND Immediate andi rd, rs, Imm

I-format

Op-code	rs	rt	Immediate

6	5	5	16
001100			

Problem example(s) Mask the last 8 bits of register $t0
Mask the first 8 bits of register $t0

ORs: OR can be used to force certain bits of a string to 1s.

OR **or rd, rs, rt**

R-format

Op-code	rs	rt	rd	value	function
6	5	5	5	5	6
000000					00000 100101

OR Immediate **ori rd, rs, Imm**

I-format

Op-code	rs	rt	Immediate
6	5	5	16
001101			

Problem example(s) set shift amount to all 1s (leave the rest unchanged)

set Byte 2 to 1s (leave the rest unchanged)

XOR: used for flipping the bits (one-s complement)

XOR **xor Rd, Rt, Rs**
XORI **xori Rd, Rt, Imm**

Problem example(s): complement the lower 12 bits of register $t0 but leave all the other bits unchanged
Complement Byte 3

Lecturer: Simina Fluture,PhD

Topics: Supporting Procedures in Computer

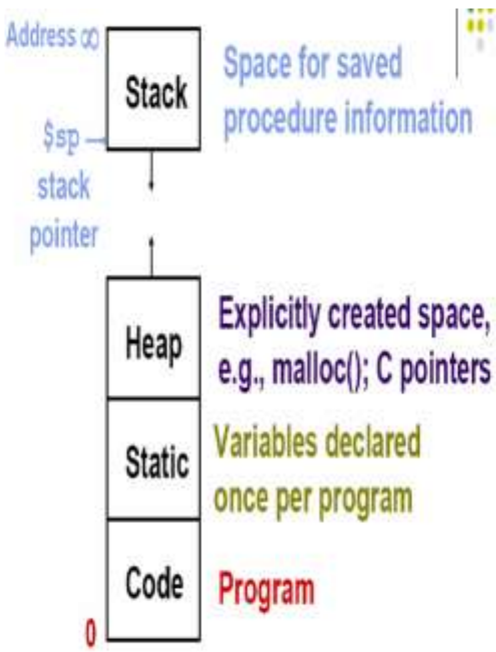
<u>Memory Allocation</u>

When a C program is run, there are 3 important memory areas allocated:

Static:

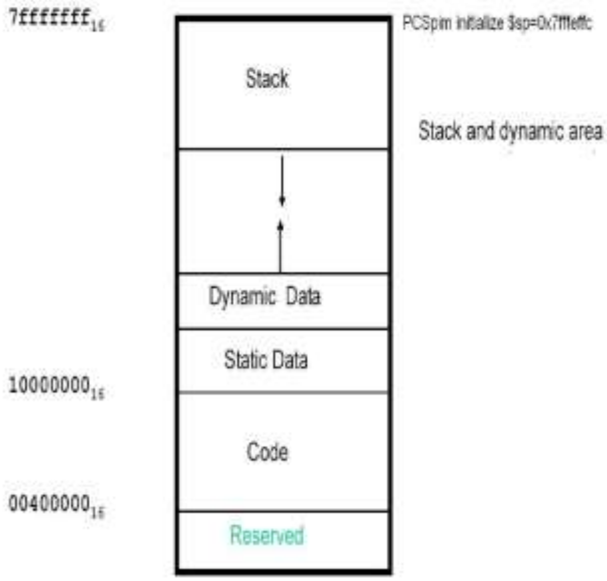
Heap:

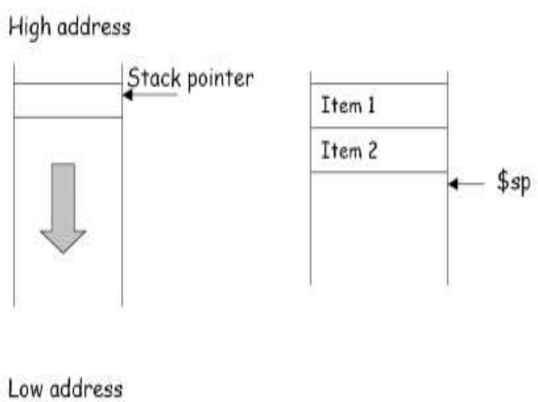
Stack:



Lecturer: Simina Fluture,PhD

MIPS Memory Layout





The stack occupies a part of the main memory. In MIPS, it grows from high address to low address as you push data on the stack. Consequently, the content of the stack pointer ($sp) decreases.

	Caller	Callee:

	--------	-------------
	--------	jr $ra
1000	Jal Callee	

Register Conventions:
Return address: $ra

The first 4 in parameters are passed to a function in: $a0, $a1, $a2, $a3
Return value: $v0, $v1

Jump-and-link jal ProcedureAddress
#jumps to an address and saves the addr of the next instruction (PC+4) into $ra

jump register jr $ra
#jump to the address specified in a register

Implement a function call

```
###################################################################
# Name:
#
# Date:
#
# Program Name: YourFullName_function.s
#
# Description:  This program illustrates how to implement a simple function call
#
# Pseudocode:
#               add_four(g,h,i,j){
#
#               return  f=(g+h)-(i+j);
#
#               }
#
# Registers:    s0: variable g
#               s1: variable h
#               s2: variable i
#               s3: variable j
#               s4: variable f
#
###################################################################
```

```
# START OF PROGRAM
```

```
        .data                    # Data declaration section; it can be empty

        .text

main:

        #declare and initialize variables (with some values)
        # load your specific registers with your choice of integer values; li ….


        #move arguments into registers for function call (param in registers)



        #call function


        #the return value of the procedure will go in $v0
        #store return value locally


        #output the return value (print int)



        #program exit

############ END OF CALLER ##########################################



################### STARTING THE CALLEE ###########################
```

 # Start of function

```
add_four:

        # Move the param in into temp registers (t registers)



        # compute


        # the result should go into $v0


        #return
```

Program Design and Documentation

Main Program Header

```
######################Example Main Program
Header#################################
#	Program name:
#	Programmer:
#	Submission Date:
############################################################################
#	Overall Program Functional Description:
############################################################################
#########
#	Register Usage in Main:
#
############################################################################
#########
#	Pseudocode Description:
#
############################################################################
```

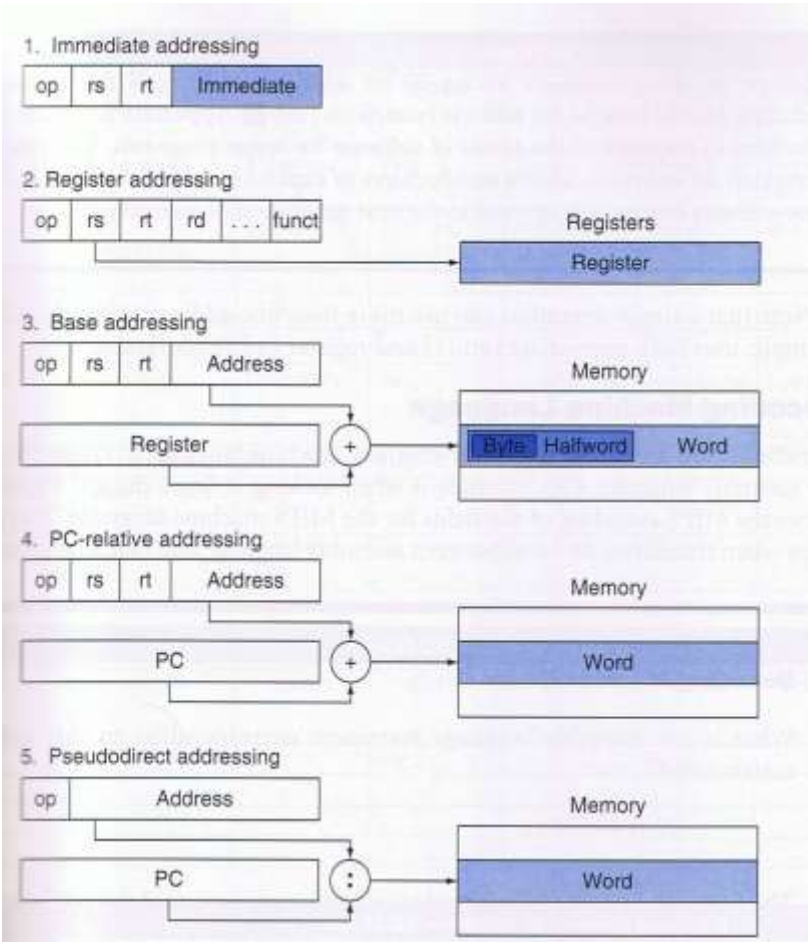
Each MIPS function should be immediately preceded by a header such as the following one for a function:

```
#######################Example Function
Header#########################################
#	Function Name:
############################################################################
#########
#	Functional Description:
#
############################################################################
#	Explain the parameters passed to the function
#
#	Explain what values are returned by the function
#
############################################################################
#	Register Usage in Function:
#
###########################################################################
#	Pseudocode Description
#
############################################################################
```

The use of in-line documentation is also useful. Example:

```
	li	$a1, 4		# initialize length parameter
	jal	Sum		# Call sum function
```

MIPS addressing modes (asynchronous lecture ???? MAYBE)



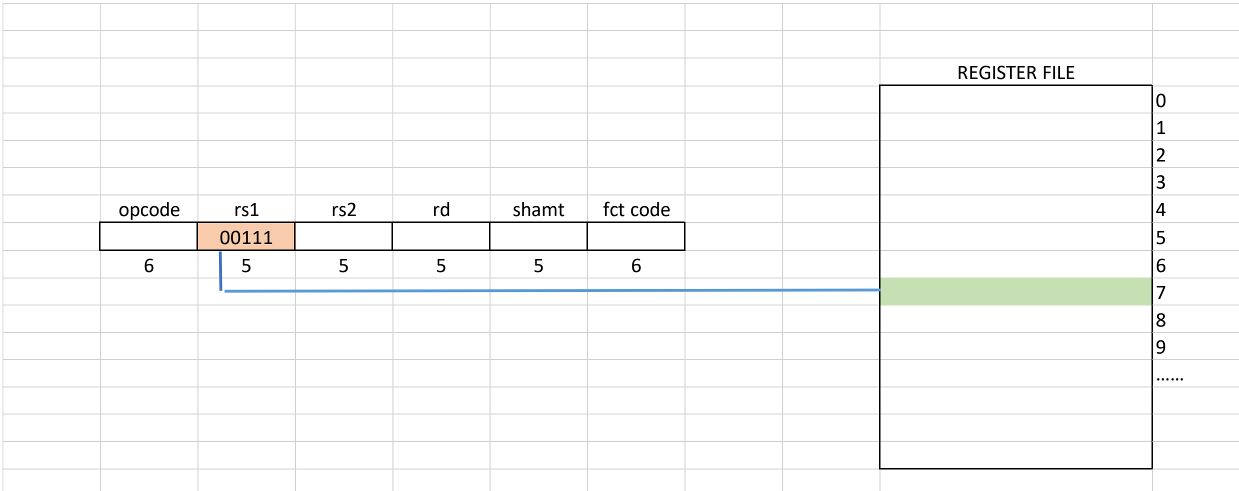
Immediate: the operand is a constant
Register addressing: operand is in a register
Base or displacement: address - sum of a register (address) and a constant in the instruction
PC-relative addressing: address is the sum of the PC and a constant in the instruction
Pseudo-direct addressing: the jump address is the 26 bits of the instruction concatenated 00 and with the 4 upper bits of the PC register

.

Lecturer: Simina Fluture,PhD

R-Format Instruction

Register Addressing



Address = register number
Data is in the register

I-Format Instruction

6	**5**	**5**	**16**
Opcode	**rs**	**rt**	**immediate**

Immediate addressing

Immediate arithmetic operations: **addi, slti, sltiu**,
The immediate field: is **sign extended** to 32 bits

addi $s1, $s2, -20 One of the operands is given immediately in decimal format. No address.

Immediate logical operations: andi, ori, xori
The immediate field: is **zero extended** to 32 bits

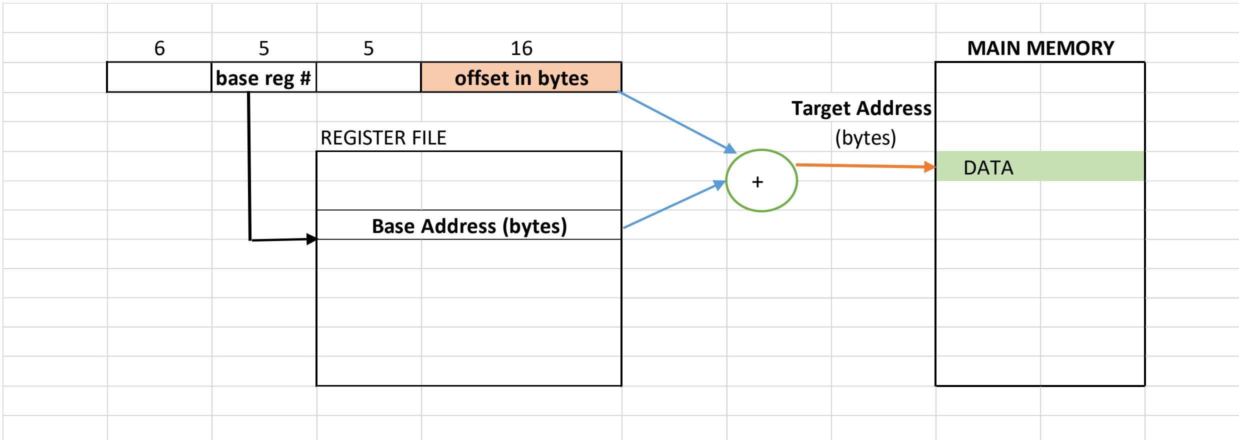
ori $s1, $s2, 0x3fac One of the operands is given immediately in Hexa format. No address.

<u>Base addressing</u>

6	5	5	16 bits
	base reg #		**offset in bytes**

sw $s0, 16($s1) **Target address:**

lw $s0, 20($s1) **$s1 = base register**



Branches: PC-relative addressing

Since the address of the next instruction is already stored in the PC (part of the fetch phase) , it is easier to use this value as the base for computing the branch target address.

The offset (immediate field) of the branch instruction represents the number of lines (instructions) from the next instruction (the instruction following the branch) to the Label.

6	5	5	16 bits
	rs1	**rs2**	**offset in lines**

beq $t1, $t2, Label **Target address = [PC+4] + offset$_{bytes}$**

MIPS code:

```
Loop:   beq $s1, $s2, End        ###  offset in lines for beq is 4          lines
        add $t1, $$t2, $t3
        addi $s0, $s4, -1
        bne $a0, $a1, Loop      ###    offset in lines for bne  is   -4      lines
        j Loop
End:
```

If bne is at the address 1000, then the target address will be:

TA: 1004 -16 = 988

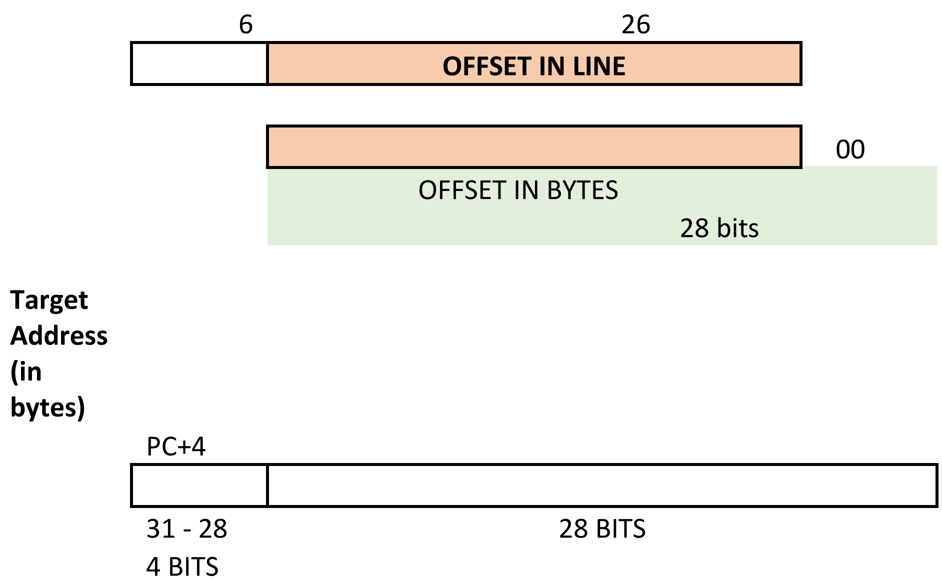
Lecturer: Simina Fluture,PhD

Jump Format

Pseudo-direct addressing

The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC+4 to the 26 address field in the jump instruction and adding 00 as the 2 low-order bits.

j Label



Topics: Recursive Functions

Recursive Functions

Programmer must save into the stack the contents of all registers relevant to the current invocation of the function before a recursive call is executed. Upon returning from the recursive function call the values saved on the stack must be restored to the correct registers.
The caller pushes any argument registers ($a0-$a3) or temporary registers ($t0-$t9) that are needed after the call.

Factorial Function

```
int fact(int n)
{
        if (n == 0) return (1);
        else return (n*fact(n – 1));
}
```

Let's consider that n is in register $s0. Since n represents the function's parameter, before we implement and call the function, n must be moved into the $a0 register.

```
move $a0, $s0
----------------------------------------------------------------------------------------------------------------
fact:

#PROLOGUE                               #allocate space and save in the stack
                                        #space for 2 words
                                        #save return address
                                        #save argument n

#BASE CASE
# if (n==0) return(1)
                                        #$v0=1
                                        #if (n==0)

#RECURSIVE CASE
# n != 0
                                        #$a0=n-1
                                        #fact(n-1)
                                        # need to retrieve correct n first; we will use a temp file $t0
                                        # return fact(n-1)*n
                                        # only if used for printing result
```

#EPILOGUE
#when done restore
done:

#restore argument n
#restore return address
#restore stack pointer
#return to caller