4

Register Usage & Procedures

Objectives

After completing this lab you will:

- know how the MIPS integer registers are used
- be able to write procedures in MIPS assembly language

Introduction

A convention regarding the use of registers is necessary when software is a team effort. In this case each member must know how registers are supposed to be used such that her piece of software does not conflict with others'.

Since almost nobody does assembly programming anymore, you may ask why such a convention is still necessary. Well, it's the compiler who needs to know about it. It is mostly because an executable can be created from pieces that are compiled separately: the compiler then makes the assumption that they all have been compiled using the same convention. To compile a procedure, the compiler must know which registers need to be preserved and which can be modified without worry.

These rules for *register-use* are also called *procedure call conventions*.

There is nothing to prevent you from ignoring these rules. After all they are not enforced by hardware mechanisms. But, if you choose to not follow the rules, then you call for trouble in the form of software bugs. And some of these bugs may be very vicious.

The following table presents the MIPS register usage convention.

Register number	Register name	Usage
`0`	`zero`	Always zero
`1`	`$at`	Reserved for the assembler
`2 - 3`	`$v0 - $v1`	Expression evaluation and procedure return results
`4 - 7`	`$a0 - $a3`	The first four parameters passed to a procedure. Not preserved across procedure calls
`8 - 15`	`$t0 - $t7`	Temporary registers. Not preserved across procedure calls

Register number	Register name	Usage
16 - 23	**$s0 - $s7**	Saved values. Preserved across procedure calls
24 - 25	**$t8 - $t9**	Temporary registers. Not preserved across procedure calls
26 - 27	**$k0 - $k1**	Reserved for kernel usage.
28	**$gp**	Global pointer (pointer to global area)
29	**$sp**	Stack pointer
30	**$fp**	Frame pointer[a]
31	**$ra**	Return address
	$f0 - $f2	Floating point procedure results
	$f4 - $f10	Temporary registers. Not preserved across procedure calls
	$f12 - $f14	The first two floating point parameters. Not preserved across procedure calls
	$f16 - $f18	Temporary floating point registers. Not preserved across procedure calls
	$f20 - $f30	Saved floating point values. Preserved across procedure calls

a. The MIPS compiler does not use a frame pointer (as gcc does). In this case register 30 is called `$s8` and is used like any of the registers `$s0` to `$s7`.

"Not preserved across procedure calls" means that the register may change value if a procedure is called. If some value is stored in that register before the procedure is called, then *you may not* make the assumption that the same value will be in the register at the return from the procedure.

"Preserved across procedure calls" means that the value stored in the register will not be changed by a procedure. It is safe to assume that the value in the register after the procedure call is the same as the value in the register before the call.

Laboratory 4: Prelab Exercise #1

Date 10/04/2023 Section 2

Name Pavithra Nagarle Guruswamy

Introduction

In this exercise we introduce the basic procedure call mechanism in MIPS. Only simple, non-nested, calls are presented.

We want to see in detail what happens when a procedure (the *caller*) calls another procedure (the *callee*), and when the control is transferred back from the callee to the caller. Along the way we introduce the instructions (`jal` and `jr`) that make the mechanism work.

The four steps needed to execute a procedure are:

a. save the return address (which is the address of the instruction just after the calling point)
b. call the procedure
c. execute the procedure
d. restore the return address and return

The MIPS architecture provides two instructions which are used in conjunction to perform the call (while saving the return address) and the return.

```
jal procedure_name
```

saves in **`$ra`** the address of the instruction following the `jal` (the return address) and then jumps to the instruction labeled 'procedure_name'. The format of `jal` is exactly the same with the format of `j` (the opcodes are different though), which means that the 26 least significant bits in the instruction represent the address of the instruction labeled 'procedure_name', divided by four. When the jump is executed two things happen:

- the address (26 bits) is left-shifted with two bits (which is equivalent to multiplying by 4) as to find the true address.
- *replace* the least significant 28 bits of the PC with the address.

The `jal` instruction implements the first two steps (a and b) in the above described sequence of steps needed to execute a procedure.

For the return (part d in the sequence of steps), we use

```
jr $ra
```

which jumps to the address stored in **$ra**. During execution the content of $ra replaces the content of the PC. The `jr` instruction can also be used with other registers and it is not necessary to be used only for the procedure return implementation.

Ex 1:

```
        lui $t0, 0x40      # $t0 <- 0x00400000
        jr $t0             # the next instruction will be fetched from ...
                           # 0x00400000 ■
```

A general framework for procedure calls is presented below.

Ex 2:

```
                ....        # this is in the main program
                jal my_proc # call the procedure 'my_proc'
Return:         ....        # instruction following jal executed after jr $ra
                ....        # is executed in 'my_proc'
                ....

my_proc:        ....        # save $ra if other procedure will
                ....        # be called from this procedure
                ....        # do some useful work here
                ....        # restore here $ra if it was saved
                jr $ra      # return ■
```

Please note that we have already used this framework from the very first program (*lab1.1.asm*). The program that starts with the label 'main' is itself a procedure invoked from the start-up code. Therefore it obeys the same rules for procedure calls as any other procedure.

One **must** save the return address (**$ra**) upon entering a procedure if the procedure calls other procedures. Some of the programs we have written so far do call other procedures (we used syscall to do so) and we saved **$ra** in another register. Just a reminder that if you choose to save the return address in a register, then that register must be one which is preserved across procedure calls (**$s0** through **$s7**). Even better, save **$ra** on the stack.

Right now we do not deal with all the details involved in a procedure call. In particular we do not save any registers. This will be discussed in Exercise #2.

Step 1

Enter the program P.1 and save it as *lab4.1.asm*.

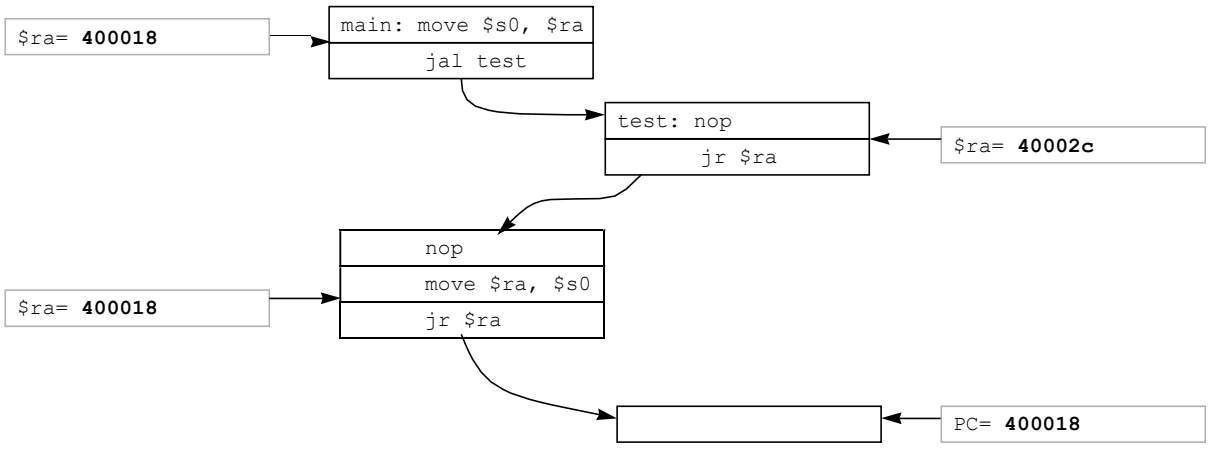
P.1:

```
                .text
                .globl main
main:           move $s0, $ra     # must save $ra since I'll have a call
                jal test          # call 'test' with no parameters
                nop               # execute this after 'test' returns
                move $ra, $s0    # restore the return address in $ra
                jr $ra            # return from main

# The procedure 'test' does not call any other procedure. Therefore $ra
# does not need to be saved. Since 'test' uses no registers there is
```

```
# no need to save any registers.

test:       nop         # this is the procedure named 'test'
            jr $ra      # return from this procedure
```

Step 2

Load *lab4.1.asm*. Use `step` to fill out the missing information in the figure.



Step 3

Explain what happens and why if you do not save **`$ra`** in **`$s0`** in 'main'.

Describe what happens	Explain why this happens
If we do not save the value of $ra (return address) in $s0 register in the main function, it will lead to unexpected behavior when calling and returning from procedures and finally the program won't termimate.	Reason is $ra is used to store the return address of a function. When a function is called, the program execution jumps to the called function, and the return address is stored in $ra. This allows the called function to know where to return to after its execution is complete.

Laboratory 4: Prelab Exercise #2

Date 10/04/2023 Section 2

Name Pavithra Nagarle Guruswamy

The Stack

The `jal` instruction saves the return address in register **`$ra`**. Assume now that a procedure wants to call another procedure. Then the programmer must save the register **`$ra`** or else the new `jal` will clobber its value. Since the number of embedded calls is not limited (procedures may call procedures that call procedures and so on), we can not used a fixed location (be it a register or a memory location) for saving a register. We need a dynamic structure, something that allows us to store data without overwriting previously saved data. Moreover we know that the last saved will be the first we will want to use (the latest saved **`$ra`** will be the first to use when returning). This place is a *stack*.

The MIPS architecture describes a stack that has the bottom at a very high address (0x7fffff) in the main memory, and which grows downwards. Register **`$sp`** points always (if the program is correct) to the first empty location in the stack.

Since the stack is just a part of the main memory, it will be accessed using the same load and store instructions we use to access the data segment. The base register will be **`$sp`**.

To save a register into the stack (*push* the register into the stack) do

Ex 1:

```
add $sp, $sp, -4 # first update the stack pointer
sw $ra, 4($sp)    # push $ra into the stack
```

Note that we start by updating the stack pointer with the proper amount (-4 in this example since we want to store a word and a word has four bytes). Then we do the actual store using a positive displacement (4 in this example) as to properly point to the empty stack location **`$sp`** was pointing to before we modified it. The negative value we add to the stack pointer comes from the fact that the stack begins at a high address and grows toward smaller addresses. ■

Q 1:

Instead of adding a negative immediate value to **`$sp`**, one can subtract a positive value from the register. What instruction would do the same as `add $sp, $sp, -4` ?

sub $sp, $sp, 4

To restore a register from the stack (*pop* the register from the stack)

Ex 2:

```
lw $ra, 4($sp)     # first recover the data from the stack
add $sp, $sp, 4    # shrink the stack by one word ■
```

In case several registers need to be saved into the stack, we do not have to repeat the sequence of instructions in Example 1 for each push. Rather, we do as in Example 3.

Ex 3:

```
add $sp, $sp, -8 # want to store 2 words (which is 8 bytes)
sw $s0, 8($sp)    # push $s0
sw $s1, 4($sp)    # push $s1 ■
```

Parameter passing

There are two ways to pass parameters to a procedure

- in registers
- on the stack

The MIPS register-use convention specifies the first four parameters to a procedure will be passed in registers (**`$a0`** through **`$a3`**), and the remaining on the stack.

Passing parameters in registers is efficient since it avoids memory accesses. If the procedure calls another procedure, then those registers who contain a parameter need to be saved (**`$a0`** to **`$a3`** are not preserved across procedure calls). The saving can be done

- in registers that are preserved across procedure calls (**`$s0`** through **`$s7`**)
- on the stack

Calling a procedure

The **caller** does the following before calling a procedure

- Save any registers that are not preserved across procedure calls (**`$a0`** – **`$a3`** and **`$t0`** – **`$t9`**), and which the caller expects to use after the call.
- Pass parameters. The first four are passed in registers **`$a0`** through **`$a3`**. Any remaining parameters are passed on the stack.
- Jump to the procedure by executing a `jal`.
- After the call, adjust **`$sp`** as to point above the the arguments passed on the stack (this is like a pop except that nothing is read from the stack).

The **callee** does

- Save **`$ra`** if the procedure will call another procedure
- Save any of the registers **`$s0`** – **`$s8`** which the procedure modifies
- Some work
- If the procedure returns a value, then place it in **`$v0`**
- Restore all saved registers (pop them from the stack)

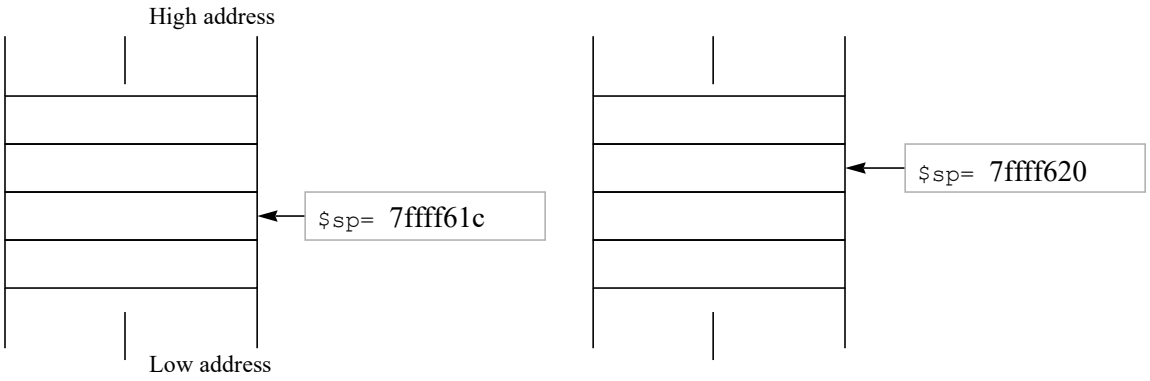
- Return by executing `jr $ra`

Step 1

Modify *lab4.1.asm* as to save **`$ra`** on the stack instead of **`$s0`**. Save your work as *lab4.2.asm*. Load and run *lab4.2.asm*. If your attempt to run results in an error or if the `(spim)` prompt does not appear, then you may want to check to see if you have the proper sequences of instructions for push (when you enter 'main') and for pop (just before you return from main).

Show a few words of storage in the stack in two different instances:

- right after you have pushed **`$ra`** into the stack in 'main'
- right after the `jr $ra` which returns from main



After pushing `$ra` into the stack

After returning from 'main'

Q 2:

As you can see in the figure above, after returning from main, the stack pointer points to a location in the stack where something different from zero is stored. Do you think the stack will behave properly in these conditions? Explain why.

Once a function completes its execution, the stack pointer is reverted to its prior position. If the stack pointer is directed to a spot in the stack containing a non-zero value, it might lead to complications when the subsequent function is invoked. This is due to the fact that the new function will assume the stack pointer is directed to an unoccupied area in the stack and may overwrite any existing value in that location.

Step 2

Based on *lab4.2.asm* create the program *lab4.3.asm* as follows:

- in 'main' prompt the user to enter two integers; store them in `$t0` and `$t1`
- call a procedure named 'Largest' whose parameters will be the two integers read from the user, and which returns the largest of them
- pass parameters in registers
- prints a message and the value returned by 'Largest'

Make sure the program runs properly. Use the table below to indicate what numbers you have used for testing

Test cases for *lab4.3.asm*

First number	Second number	Expected result	Obtained
0	0	0	0
-1	2	2	2
-3	5	5	5
2	7	7	7
-5	9	9	9

Step 3

Based on *lab4.3.asm* create the program *lab4.4.asm* which does the same thing, except that the parameters for 'Largest' are all passed on the stack instead of registers.

Make sure the program runs properly. Use the table below to indicate what numbers you have used for testing

Test cases for *lab4.4.asm*

First number	Second number	Expected result	Obtained
0	0	0	0
-1	2	2	2
-3	3	3	3
-4	6	6	6
-8	1	1	1

Laboratory 4: Inlab

Date 10/04/2023 Section 2

Name Pavithra Nagarle Guruswamy

Recursive Procedures

A recursive procedure call is a nested call where the procedure calls itself. Yet a recursive procedure can not call itself always, or it would never stop. Therefore we must keep in mind that an essential ingredient of any recursive procedure is the *termination condition*. The termination condition specifies when the procedure can cease to call itself. Another point to make is that the procedure calls itself every time with different arguments (parameters) or else the computation never stops.

The most often cited function to be computed in a recursive way is the factorial. Its inductive definition

- Basis: $0! = 1$
- Induction: $N! = (N-1)! \cdot N$, for $N \geq 1$

can be easily translated in a recursive program

Ex 1:

```
int factorial(int num)
{
int fact;
      if (num == 0) return 1;              /* the termination condition */
      fact = factorial(num-1) * num;       /* recursive call */
      return(fact);
}
```

The function 'factorial' calls itself, every time with a smaller argument, and it has a termination condition in which it directly calculates a result. ■

Q 1:

What happens if the function factorial() is called with a negative argument?

If the factorial function ( ) is invoked with a negative input, it will become stuck in an endless loop. This occurs because the stopping condition (if num == 0) will never be satisfied, causing the function to repeatedly call itself with a decreasing value of num. Since num will never reach 0, the function will continue to call itself endlessly.

The factorial function can be easily coded in MIPS assembly as indicated by the next example.

Ex 2:

```
# This procedure computes the factorial of a non-negative integer
# The parameter (an integer) received in $a0
# The result (a 32 bit integer) is returned in $v0
# The procedure uses none of the registers $s0 - $s7 so no need to save them
# Any parameter that will make the factorial compute a result larger than
# 32 bits will return a wrong result.

Factorial:
            subu $sp, $sp, 4
            sw $ra, 4($sp)          # save the return address on stack

            beqz  $a0,  terminate   #  test  for  termination
            subu $sp, $sp, 4        # do not terminate yet
            sw $a0, 4($sp)  # save the parameter
            sub $a0, $a0, 1         # will call with a smaller argument
            jal Factorial
# after the termination condition is reached these lines
# will be executed
            lw $t0, 4($sp)          # the argument I have saved on stack
            mul $v0, $v0, $t0       # do the multiplication
            lw $ra, 8($sp)          # prepare to return
            addu $sp, $sp, 8        # I've popped 2 words (an address and
            jr $ra                  # .. an argument)

terminate:
            li $v0, 1               # 0! = 1 is the return value
            lw $ra, 4($sp)          # get the return address
            addu $sp, $sp, 4        # adjust the stack pointer
            jr $ra                  # return
```

■

Step 1

Based on *lab4.2.asm* create the program *lab4.5.asm* as follows:

- in 'main' prompt the user to enter an integer; store it in **`$t0`**
- check if the number entered is negative: if it is negative, then print a message saying so and prompt the user again for a number
- call the procedure named 'Factorial' whose parameter will be the integer read from the user, and which returns the factorial of that number
- pass the parameter in a register
- prints a message and the value returned by 'Factorial'

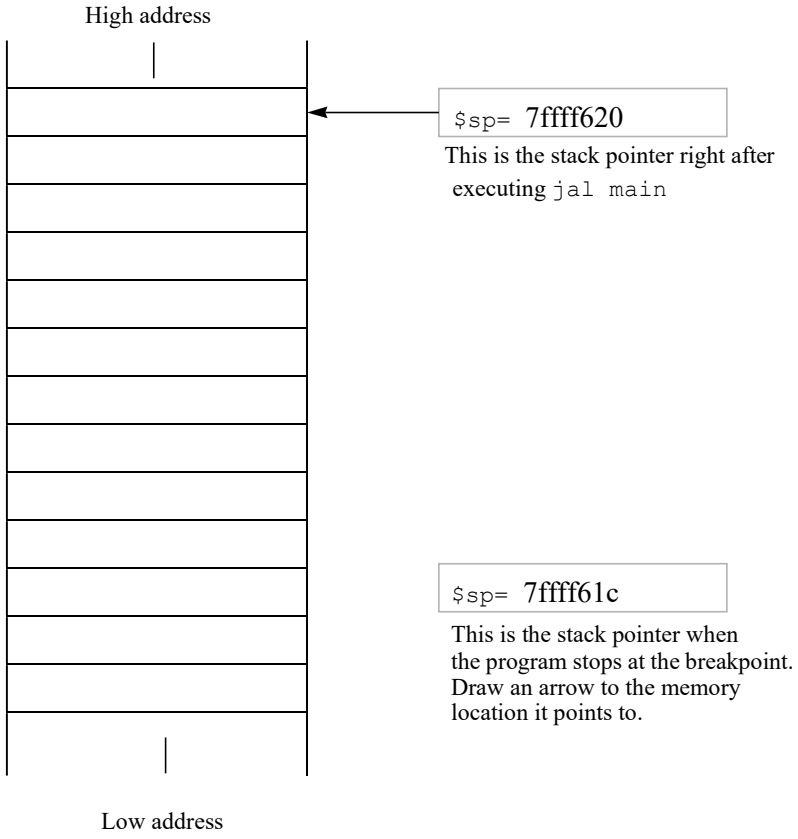
Run your program and make sure it works correctly.

Step 2

To do this step you need an integer between 5 and 9 from your instructor.

n = 7

Reload *lab4.5.asm*. Step until you encounter the `jal main` instruction. Write down the address in **`$sp`**. Set a breakpoint at the address where the instruction labeled 'terminate' is. Run the program. Use the number provided by your instructor when prompted for a number. When the program stops at the breakpoint, peek into the stack and fill the blanks in the following figure



Q 2:

In the figure above the stack seems to hold some addresses that point to code (i.e. point somewhere in the text segment). What instructions exactly do they point to?

Address	Instruction	Comment
0040007c	addiu $29,$29,-4	subu $sp, $sp, 4
00400080	sw $4, 4($29)	sw $a0, 4($sp)

Address	Instruction	Comment
00400084	addi $4, $4, -1	sub $a0, $a0, 1

Q 3:

How many recursive calls are made in the program?

8

Q 4:

Every time a recursive call is made, the stack grows. By how many bytes per call?

8 bytes

Q 5:

What is the maximum integer you may enter such that the factorial still returns a correct value?

12

Q 6:

Since the stack grows at every recursive call, it may be possibly exhausted. Assuming that the size of the stack segment is 2,000,000 bytes, what input value would generate so many calls as to exhaust the stack?

250000

Q 7:

In your opinion, what limits the ability of *lab4.5.asm* to calculate the factorial of very large numbers?

The support is only for 32bits in the MIPS version which we use.

Laboratory 4: Postlab

Date 10/04/2023 Section 2

Name Pavithra Nagarle Guruswamy

Recursive Procedures (cont'd)

Now that you have enough experience with the procedure call mechanism in MIPS it is time to try something harder on your own.

A famous example of a recursive function is *Ackermann's function* $A(x, y)$:

- if $x = 0$ then $y+1$
- else if $y = 0$ then $A(x\text{-}1, 1)$
- else $A(x\text{-}1, A(x, y\text{-}1))$

Step 1

Write a program in C/C++ (named *lab4.6.c*) which:

- asks the user for two non-negative integers, x and y
- ouputs the value of the Ackermann's function for those two values, $A(x, y)$

Run the program and fill out the missing parts of the following test table

Test plan for the C/C++ Ackermann's function

x	y	$A(x, y)$
0	0	1
0	2	3
1	0	2
2	3	9
2	4	11
2	5	13
2	6	15
3	1	13
3	2	29

Test plan for the C/C++ Ackermann's function

x	y	$A(x, y)$
3	3	61

Note: Do not take large values (especially for x) if you want your program to finish in a reasonable amount of time (or to finish at all). It takes 128 seconds to compute $A(3, 12)$ on an Indigo^2 running at 250 MHz, with 64 MB main memory, with the code compiled using the cc compiler and the -O2 flag for optimization. On the same machine $A(4, 2)$ exhausts not only the main memory space but also the 100 MB disk swap space.

Step 2

Create a program named *lab4.7.asm* which

- in 'main' prompts the user to enter two non-negative integers; store them in `$t0` and `$t1`
- check if any of the numbers entered is negative: if it is negative, then print a message saying so and prompt the user again for numbers
- call the procedure named 'Ackermann' whose parameters will be the integers read from the user, and which returns the value of the Ackermann's function for those two integers
- pass the parameters in registers
- prints a message and the value returned by 'Ackermann'

Run your program and fill out the missing spaces in the following test plan

Test plan for the MIPS assembly Ackermann's function

x	y	$A(x, y)$
0	0	1
0	2	3
1	0	2
2	3	9
2	4	11
2	5	13
2	6	15
3	1	13
3	2	29
3	3	61

Make sure you obtain the proper values when running your program.

Q 1:

How many recursive calls are made to calculate *A*(2, X) where X is the first digit of your SSN?

X = 1	Recursive calls = 14

Q 2:

Every time a recursive call is made, the stack grows. By how many bytes per call?

8 bytes

Step 3

Return to your lab instructor copies of *lab4.6.c* and *lab4.7.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopy), e-mail or both.