CS230: Digital Logic Design and Computer Architecture

Lecture-8: MIPS Instructions-III

https://www.cse.iitb.ac.in/~biswa/courses/CS230/autumn23/main.html

Sequential execution and jumps

PC, PC+4, PC+8, …………….

PC, PC+4, {if condition here, TRUE} PC+32, ………

j instruction loads an immediate into the PC. It can be either specified as an offset or the label (assembler will convert this label into an offset).

Functions (Procedures)

```
int sum(int a, int b)
{
   int c=a+b;
   return c;
}
void main (void)
{
   int i=1;
   int j=2;
   int k = sum(i,j);
  // …..
}
```

Simple ☺

```
int sum(int a, int b)
{
   int c=a+b;
   return c;
}
void main (void)
{
   int i=1;
   int j=2;
   int k = sum (i,j);          //jump to function
   // .....
}
```

Simple ☺

```
int sum(int a, int b)
{
    int c=a+b;
    return c;
}
void main (void)
{
    int i=1;
    int j=2;
    int k = sum(i,j);
   // …..
}
```

How do you return? ☹

j sum

Awesome Instructions

- jal: Jump and Link and jr $ra

jal L1:

go to L1, the instruction that has to be executed next is in L1.

and

save the address of the next instruction in $ra. ra is an awesome register that stores the return address.

Awesome Instructions

- jal: Jump and Link and jr $ra

Go to instruction whose address is stored in ra (PC+4)

jal L1:

go to L1, the instruction that has to be executed next is in L1.

and

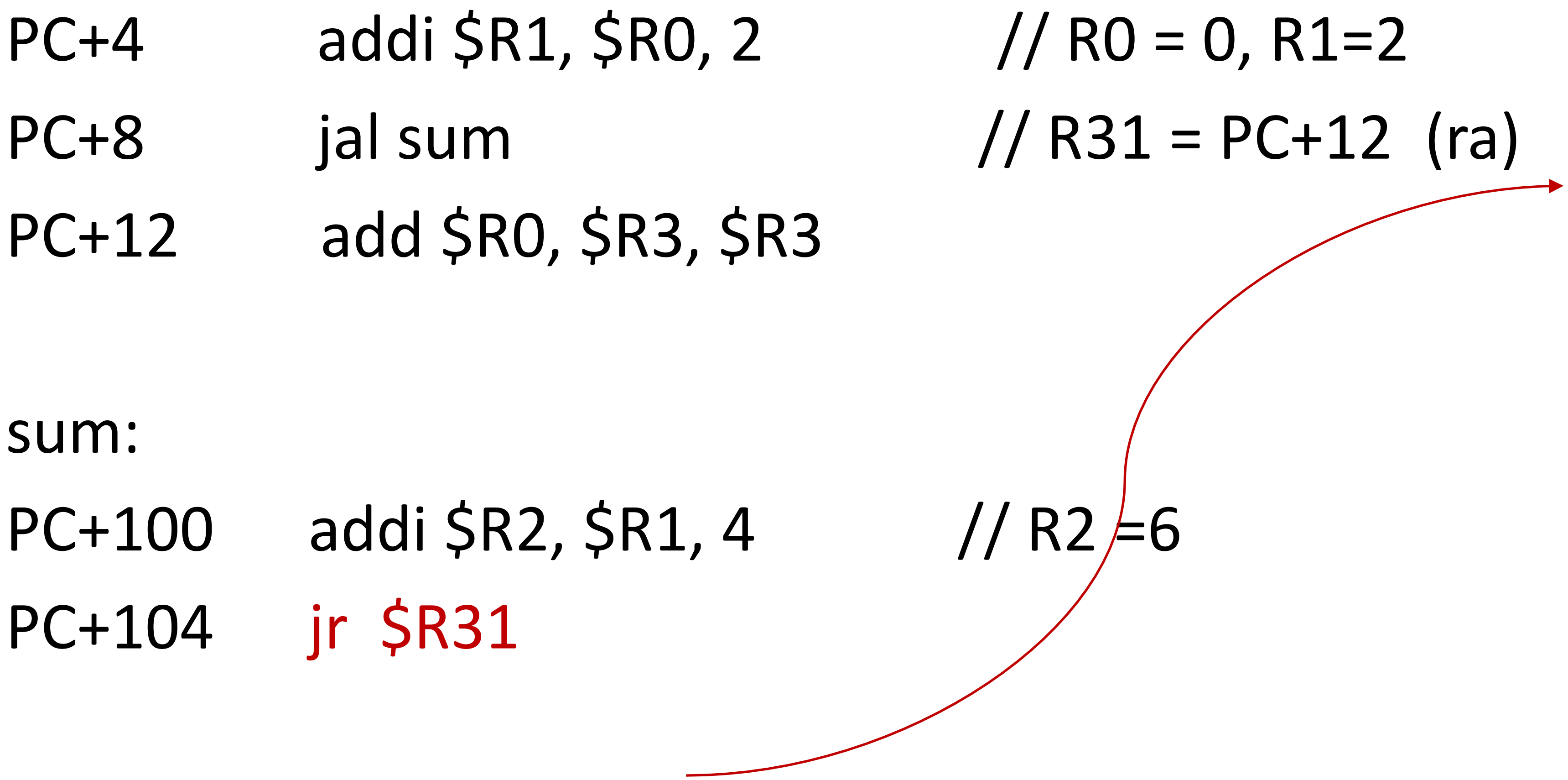
save the address of the next instruction in $ra. ra is an awesome register that stores the return address (ra).

Let’s Have a Complete Picture

```
PC+4        addi $R1, $R0, 2            // R0 = 0, R1=2
PC+8        jal sum                     // R31 (ra) = PC+12
PC+12       add $R0, $R3, $R3

sum:
PC+100      addi $R2, $R1, 4
PC+104      jr
```

Let’s Have a Complete Picture



```
PC+4        addi $R1, $R0, 2          // R0 = 0, R1=2
PC+8        jal sum                   // R31 = PC+12  (ra)
PC+12       add $R0, $R3, $R3

sum:
PC+100      addi $R2, $R1, 4          // R2 =6
PC+104      jr  $R31
```

Let’s Have a Complete Picture

```
PC+4        addi $R1, $R0, 2           // R0 = R3 = 0, R1=2
PC+8        jal sum                    // R31 = PC+12  (ra)
PC+12       add $R0, $R3, $R3          // R0 = 0

sum:
PC+100      addi $R2, $R1, 4           // R2 =6
PC+104      jr  $R31
```

Let’s Have a Complete Picture

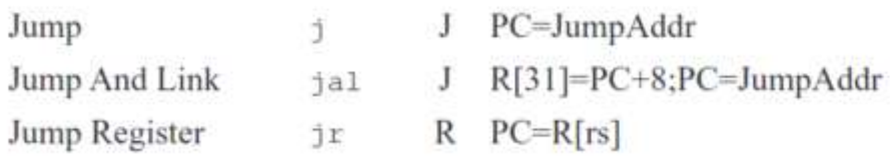
```
PC+4      addi $R1, $R0, 2        // R0 = R3 = 0, R1=2
PC+8      jal sum                 // R31 = PC+12  (ra)
PC+12     add $R0, $R2, $R2       // R0 = 12

sum:
PC+100    addi $R2, $R1, 4        // R2 =6
PC+104    jr  $R31
```

JAL: Jump and Link, What’s wrong?

```
PC+4     addi $R1, $R0, 2
PC+8     jal sum                  // R31 = PC+12  (ra)
PC+12    add $R0, $R2, $R2
```

Well



Jump	j	J	PC=JumpAddr
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr
Jump Register	jr	R	PC=R[rs]

As per MIPS specification, Check P&H MIPS sheet ☹

PC: jal label ra = PC + 8

PC+4:

PC+8:

PC+4 or PC+8? Why this

PC+4 at the moment

PC+8 after a month or so ☺

PAUSE: Quick recap

Usage of j, jr, jal, and $ra

MIPS provides

Upto four arguments can be passed from the caller to the callee while using jal. It uses registers $a0 to $a3

A callee can return upto two values to the caller. It uses registers $v0 and $v1

What if?

```
main(){
a = a + f1(a);
}
f1(a) {
        a = a -  f2(a);  return a;}
f2(a) {
        a = a + f3(a); return a;}
f3(a) {
        a = a + 1;      return a;}
```

```
f1:
        f2’s argument in $a0 to $a3
                jal f2
```

What if?

```
f1:
    f2’s argument in $a0 to $a3
    jal f2
…
f2:
    f3’s argument in $a0 to $a3
    jal f3
…
```

What is the big deal?

```
f1:
    f2’s argument in $a0 to $a3
    jal f2
…
f2:
    f3’s argument in $a0 to $a3
    jal f3
…
```

What is the big deal? Oh no!

```
f1:
  PC:   f2’s argument in $a0 to $a3
  PC+4: jal f2                    // $ra = PC+8
…
f2:
  PC+100: f3’s argument in $a0 to $a3
  PC+104: jal f3             // $ra = PC+108
                                                  f3: …
…                                                     jr $ra
```

What is the big deal? Oh no!

```
f1:
  PC:   f2’s argument in $a0 to $a3
  PC+4: jal f2                    // $ra = PC+8
…
f2:
  PC+100: f3’s argument in $a0 to $a3
  PC+104: jal f3              // $ra = PC+108
  jr $ra ☹     Oh no!!                          f3: …
…                                                   jr $ra
```

Saving and Restoring Registers (limited)

caller registers

callee registers

Why?

Callee does not know, registers used by callers, can be many callers too

Caller does not know the callee’s plan ☺

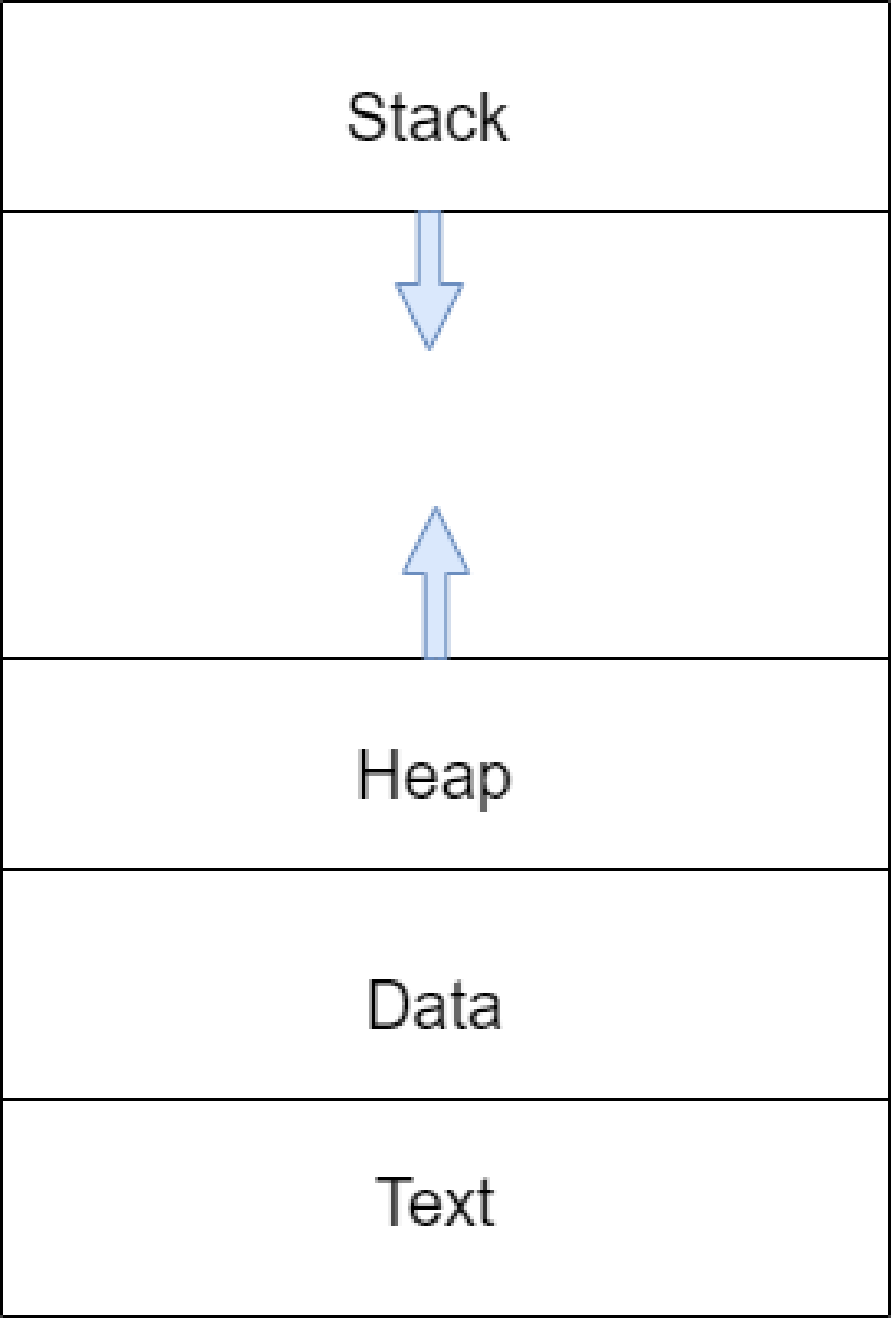
Do not forget 32 MIPS registers only Register spilling ☹

Quick recap

Register spilling, 32 MIPS registers, nested functions,

oh no!

Spilled registers: Where else can we store?

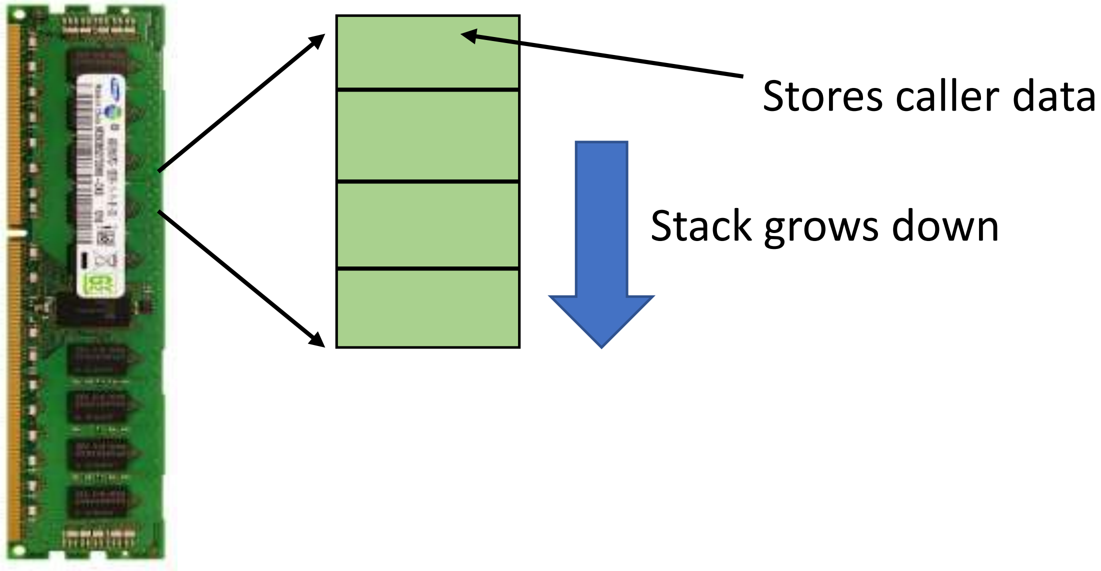


The loaded program

System program that loads the executable into the memory.

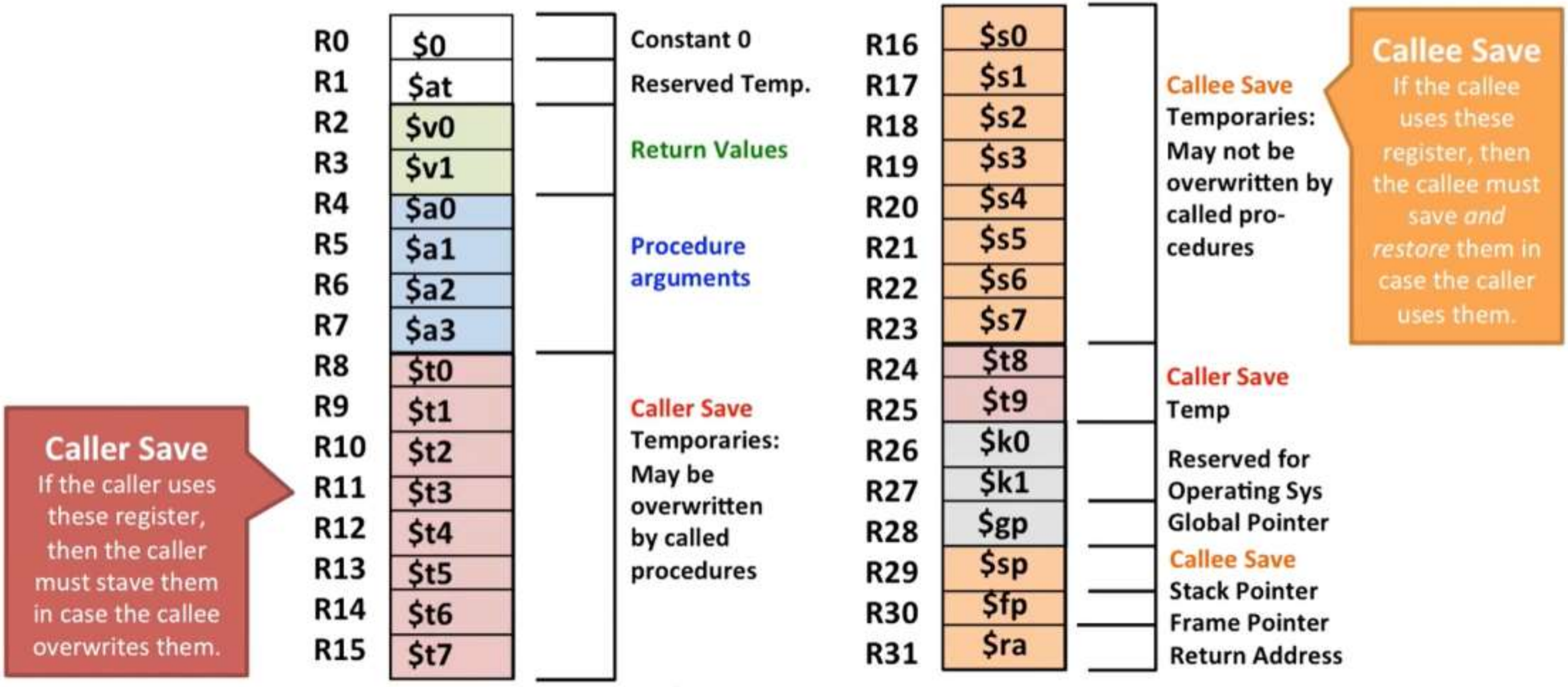
Every executable has a text, heap/stack data segments

MIPS way of handling it:
The Stack (part of DRAM, for each function call)



$sp (stack pointer) points to the address where stack ends

One per function, private memory area, else the same problem ☹



R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	

R16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Callee Save Stack Pointer
R30	$fp	Frame Pointer
R31	$ra	Return Address

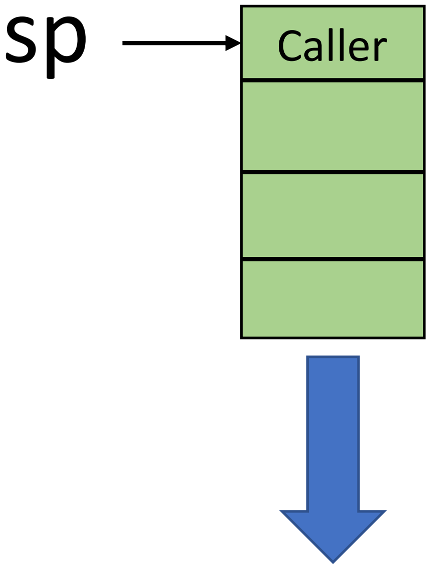
Caller Save

If the caller uses these register, then the caller must stave them in case the callee overwrites them.

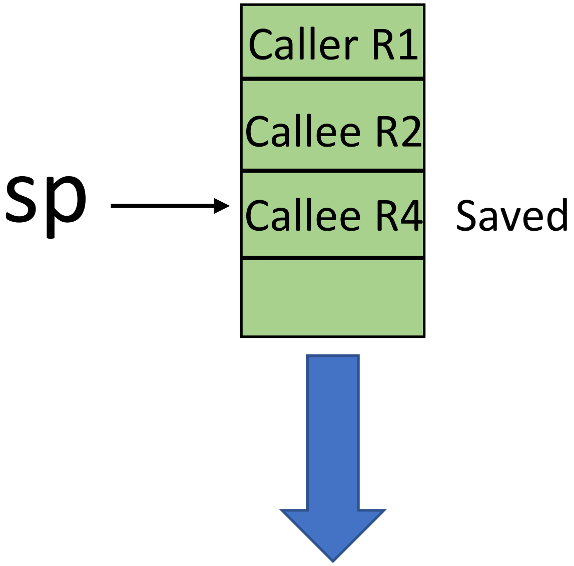
Callee Save

If the callee uses these register, then the callee must save *and restore* them in case the caller uses them.

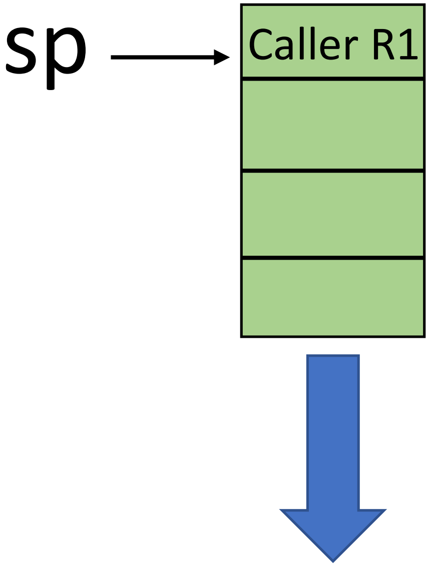
MIPS way of handling it: Before function call



MIPS way of handling it: Function call is ON



MIPS way of handling it: After the function call

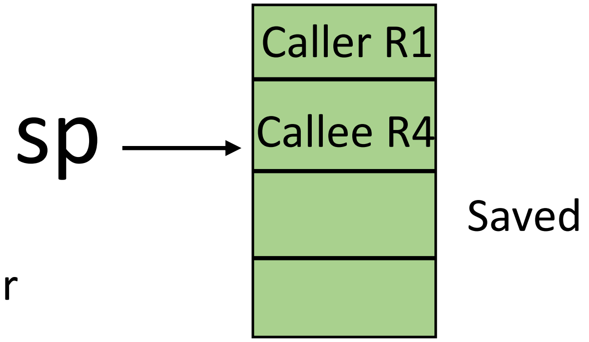


How to save and restore?

Save:

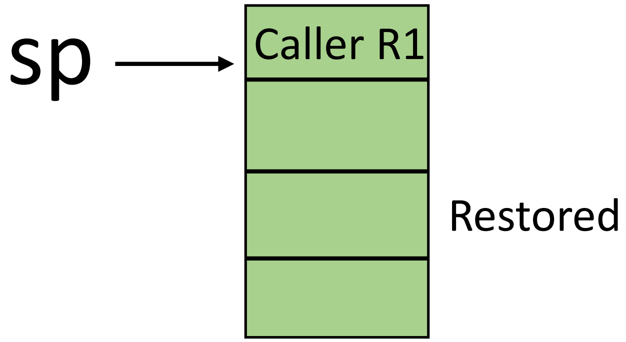
```
addi $sp, $sp, -4
sw R4, ($sp)
```

32 bit registers, 4 bytes, one word, remember



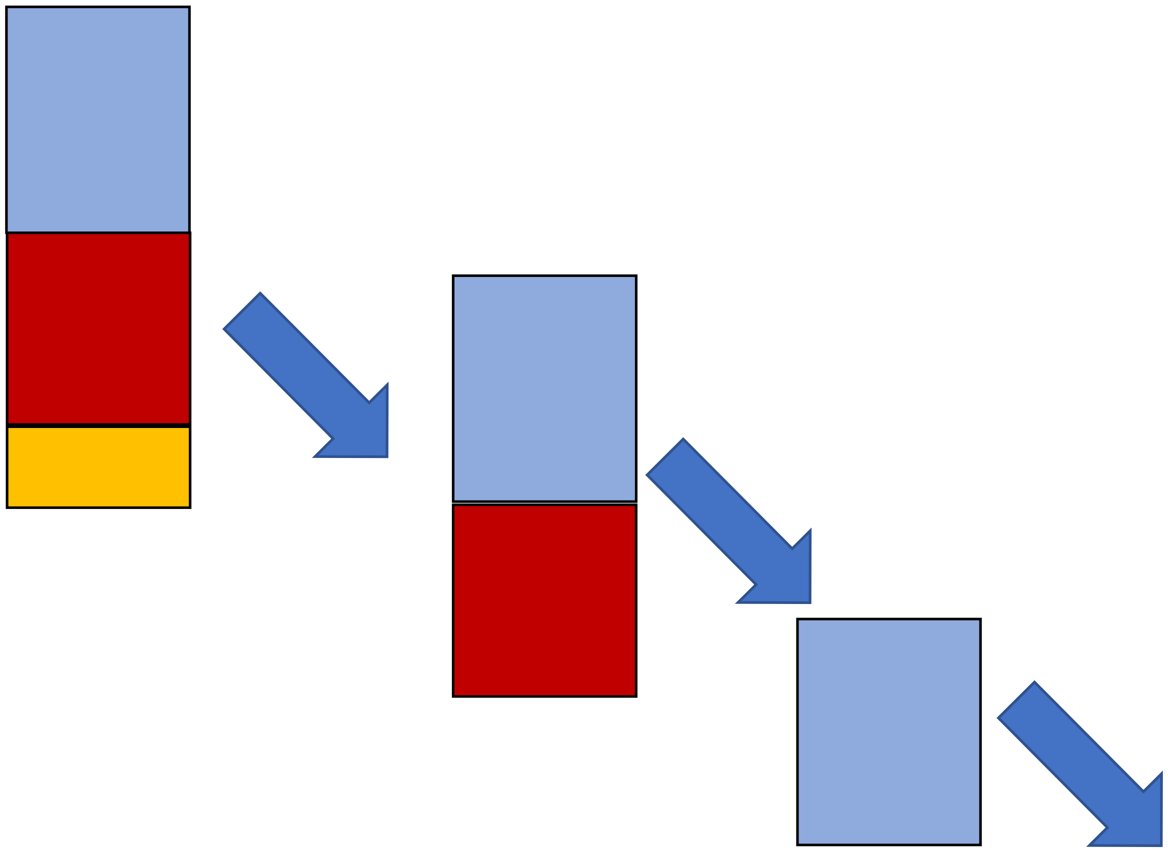
Restore:

```
lw R4, ($sp)
addi $sp, $sp, 4
```



Nested Functions (Remember main() is a function too ☺)

```
CS230 // jal cs230
{
    CS330 // jal cs330
     {
         CS430 // jal cs430
          {
          } //jr
      } //jr
} // jr
```



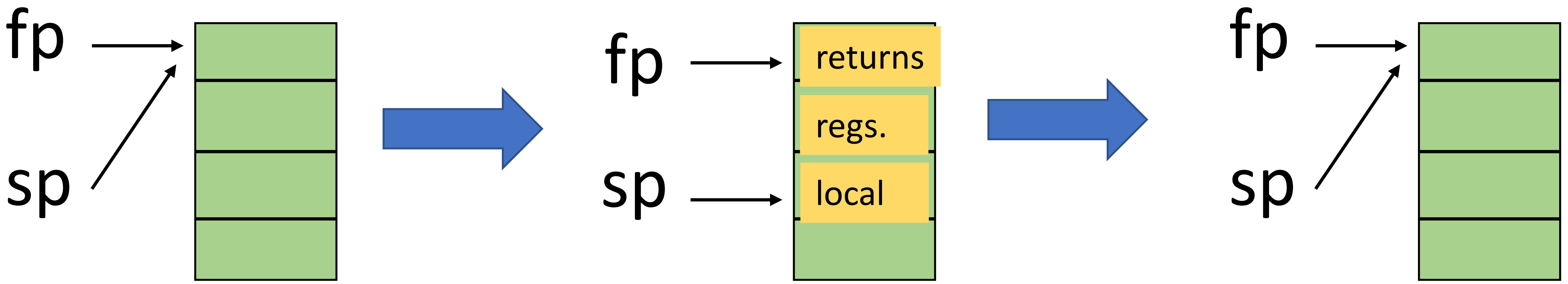
The final one: Frame pointer

Stack also stores local variables and data structures (local arrays and structures) for a function along with the return address(es).

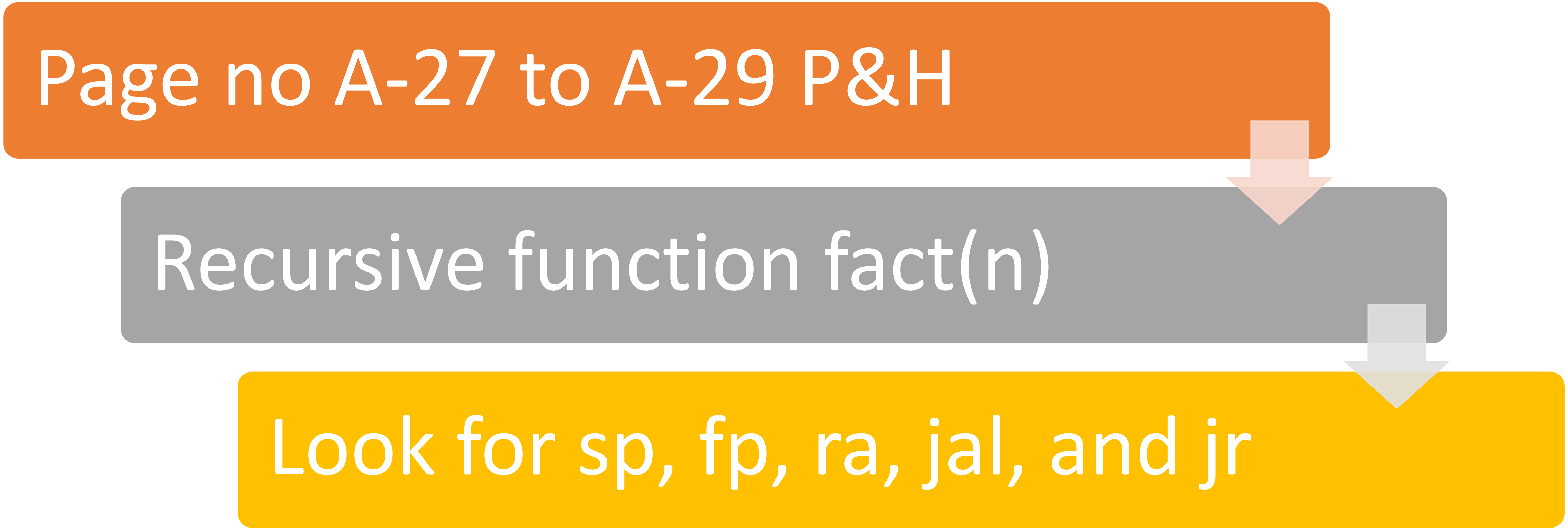
Frame pointer will get incremented and decremented based on the local arguments used.

The final one: Frame pointer

Frame pointer: Points to local variables and saved registers. Points to the highest address in the procedure frame. Stays there throughout the procedure. Stack pointer, moves around.



Awesomeness: You can access any using fp/sp and an offset



Page no A-27 to A-29 P&H

Recursive function fact(n)

Look for sp, fp, ra, jal, and jr

For the Curious Ones (Beyond CS230)

Stack buffer overflow - 101:
https://en.wikipedia.org/wiki/Stack_buffer_overflow

Coffee credits

Ankeet +5

ਸਤਿ ਸ੍ਰੀ ਅਕਾਲ