Lesson on Lab 1

MIPS Calling Convention

- It is all about supporting *procedures* in hardware
- A procedure is a set of instructions that performs a specific task and then comes back to the point of origin
- When a procedure is called by a *caller*, the program must follow some steps
  1. Put parameters in a place where the called procedure (*callee*) can access them
  2. Transfer control to the procedure
  3. Acquire enough space on the memory needed for the procedure. This memory is called *stack*
  4. Perform the task
  5. Give back the result to the caller
  6. Return control to the point of origin

What is stack

- Every procedure can access a few number of registers
- What if it needs more?
  - It spills into some place in memory called *stack*
  - Every procedure has its own share of memory which is called *stack frame*
- What if callee needs to access the same register as caller does?
  - Callee must cover all its tracks on its return, i.e.,
  - On the return of the callee, any register needed by the caller must be restored to its value which it contained *before* the procedure call (preserved registers)

Prologue and Epilogue

- Somehow callee **must** cover its track.

Prologue

  - At the beginning of the procedure, some preparation must be performed, i.e.,
  - (i) allocate required amount of space on the stack for the callee and (ii) save preserved registers on the stack
  - Likewise, at the end of the procedure, we want to restore the stack and registers to their previous state before the procedure was called

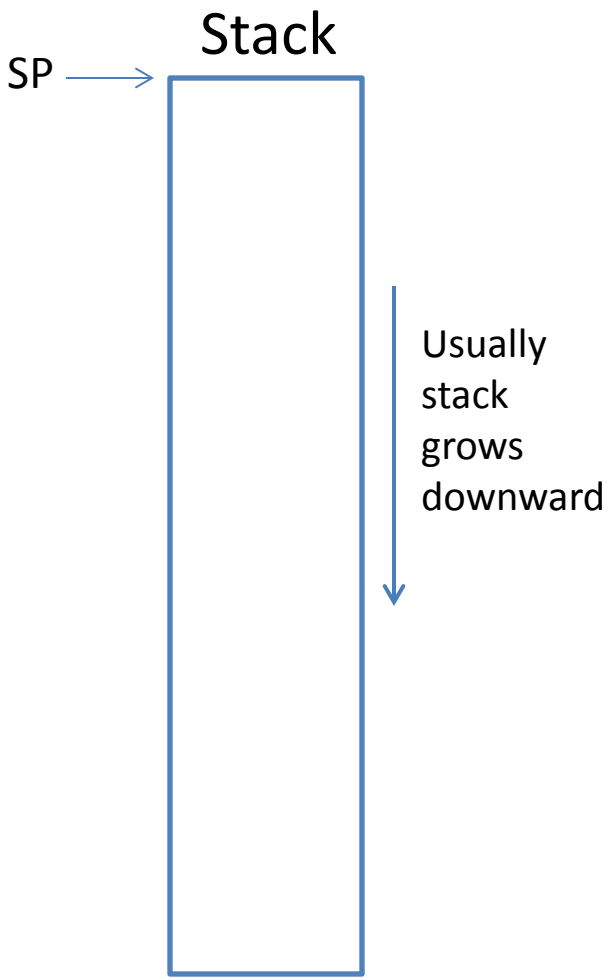
Epilogue

Calling Convention

- The set of rules that defines how registers should be used and what is the organization of the stack is called *calling convention*
- There is no one single MIPS calling convention
- As such, what you will find in the book or other resources might be slightly different from how it is handled in SimpleScalar
- But, once you get the idea they all look very similar

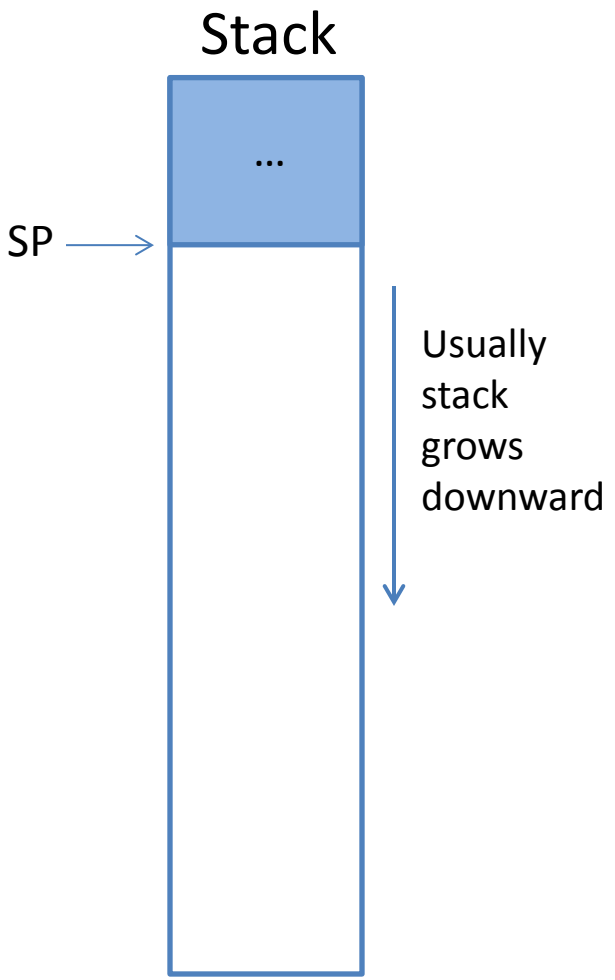
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



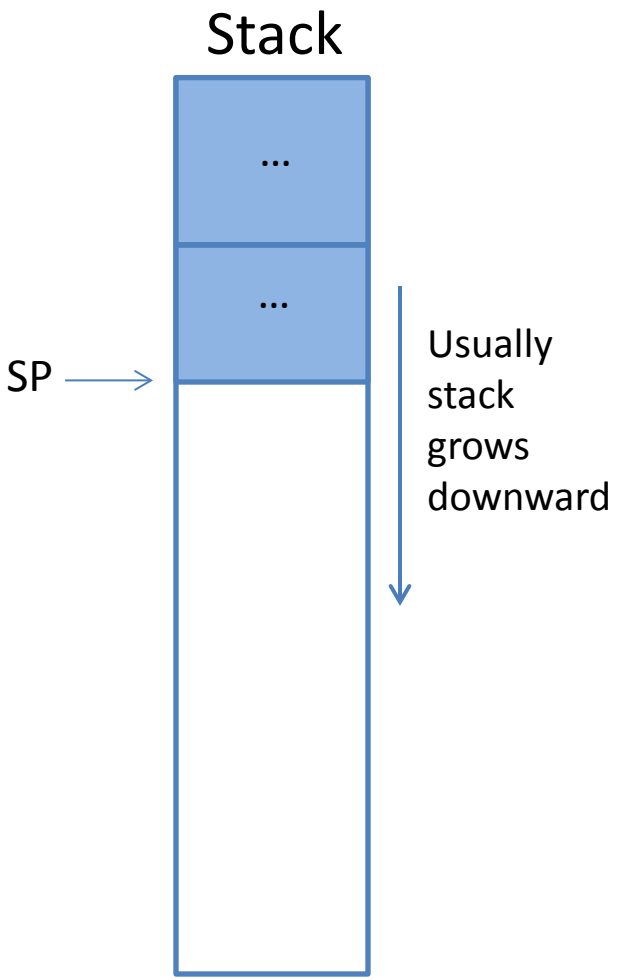
High level example

```
int main
{
        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



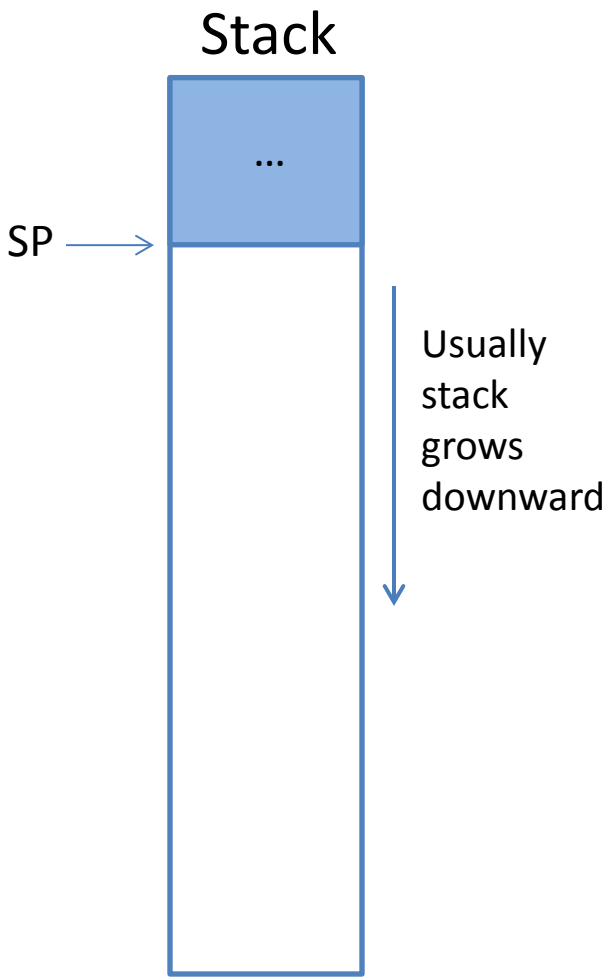
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



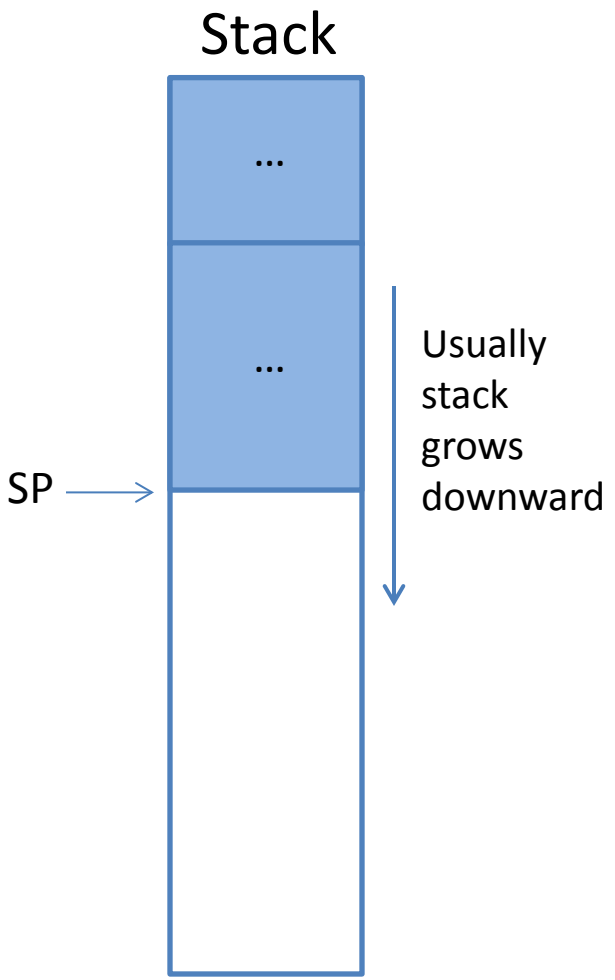
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



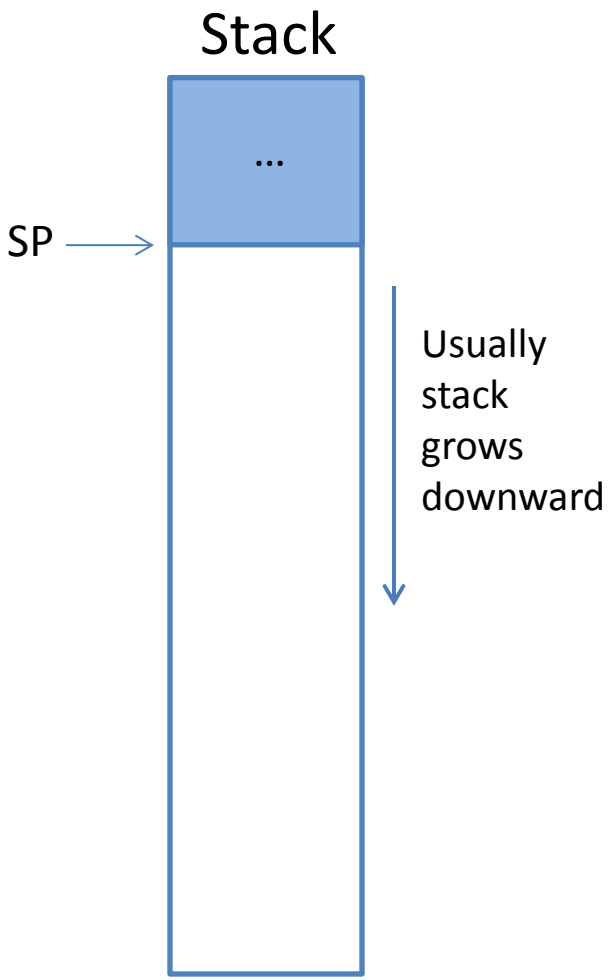
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



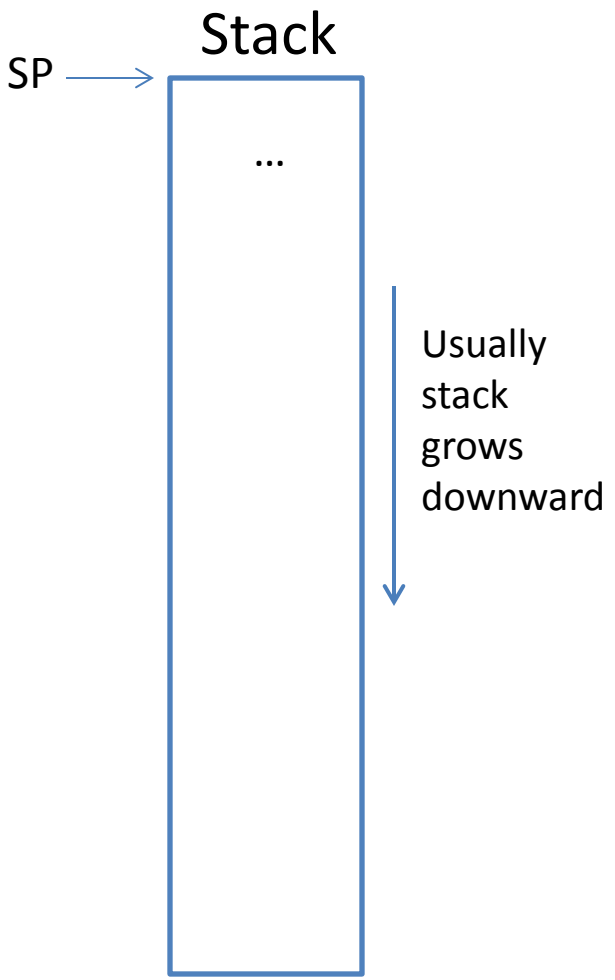
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



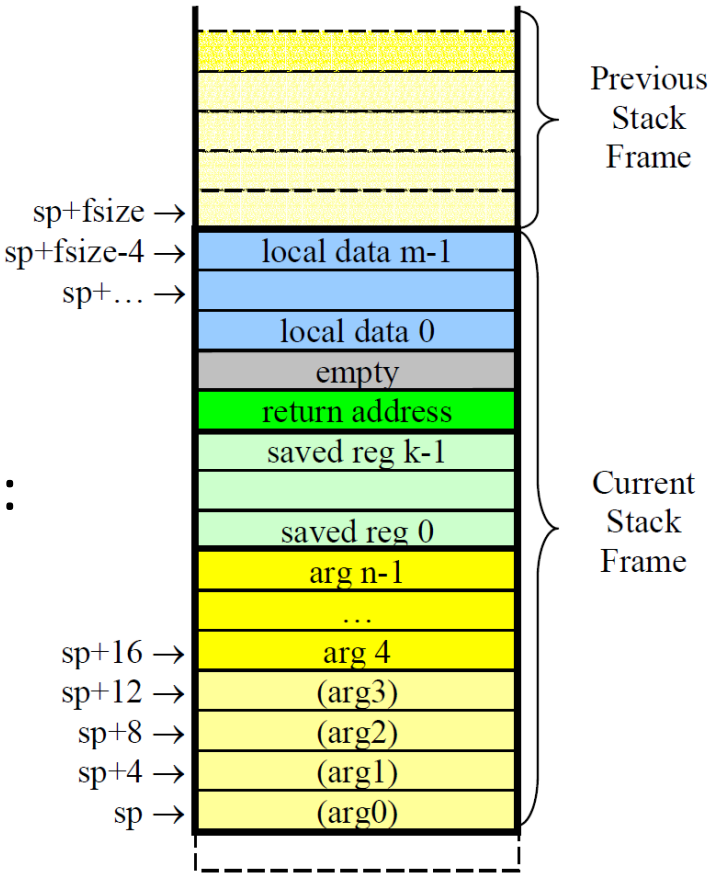
High level example

```
int main
{

        proc1
        {
        …
        }

        proc2
        {
        …
        }

}
```



What is in the stack?

- The organization of the stack might be **different** from one assembler to another
  - like the place in the stack frame where $ra is saved, or the place where $fp points to
- However, all different structures have to reserve space to store the following:
  - Arguments
  - Preserved registers
  - The return address ($31 or $ra)
  - Local data (like local variables)
  - Padding (empty word in this picture)



* Structure might look different for other assemblers

MIPS register conventions

Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15,$24,$25	$t0-$t9	The *Temporary Registers*.
$16-$23	$s0-$s7	The *Saved Registers*.
$26-$27	$k0-$k1	The *Kernel Reserved registers*. DO NOT USE.
$28	$gp	The *Globals Pointer* used for addressing static global variables. For now, ignore this.
$29	$sp	The *Stack Pointer*.
$30	$fp (or $s8)	The *Frame Pointer*, if needed . Programs that do not use an explicit frame pointer can use register $30 as another saved register. Not recommended however.
$31	$ra	The *Return Address* in a subroutine call.

The shaded rows indicate registers whose value must be preserved across subroutine calls.

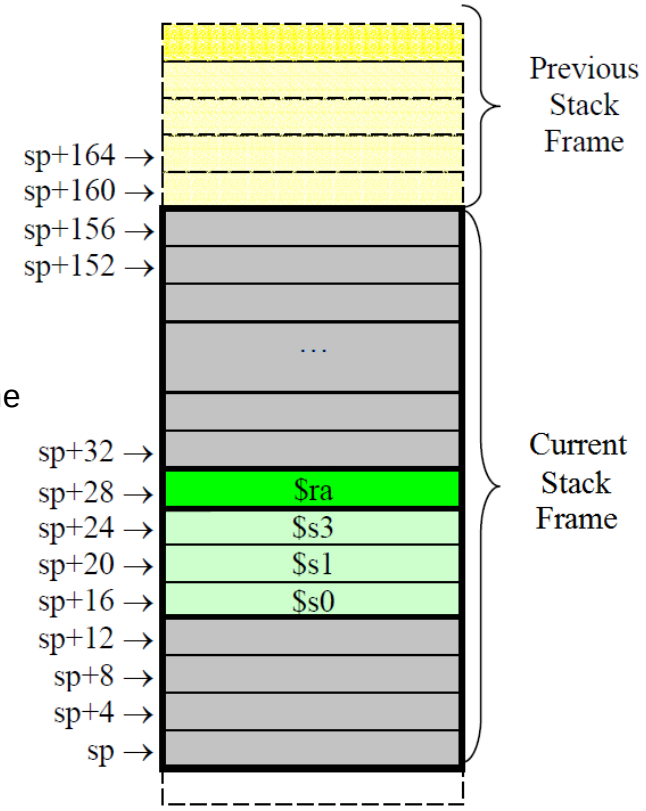
Let's do an example

```
int g( int x, int y )
{
     int a[32];
          ... (calculate using x, y, a);
     a[1] = f(y,x,a[2]);
     a[0] = f(x,y,a[1]);
     return a[0];
}
```

Generate the assembly file

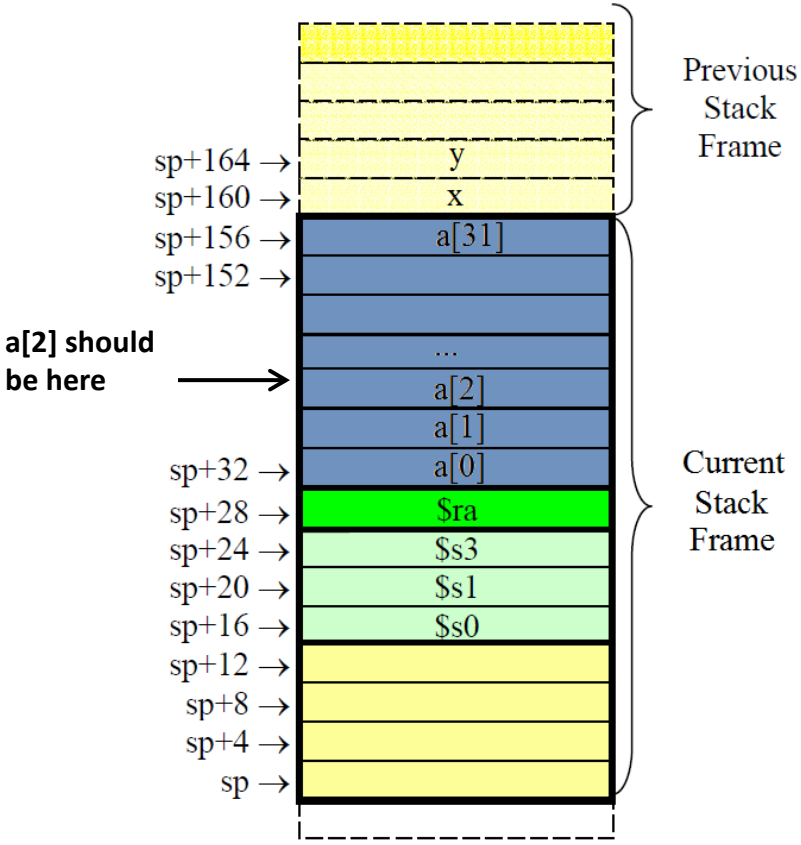
```
# start of prologue
addiu $sp,$sp,(-160) # allocate the required space
                     # on the stack for this stack frame
sw $ra,28($sp) # save the return address
sw $s0,16($sp) # save value of $s0
sw $s1,20($sp) # save value of $s1
sw $s3,24($sp) # save value of $s3
# end of prologue
```

Let's assume that s0, s1, s3 are used by the caller. So they must be saved (look at the shaded rows in the previous slide)



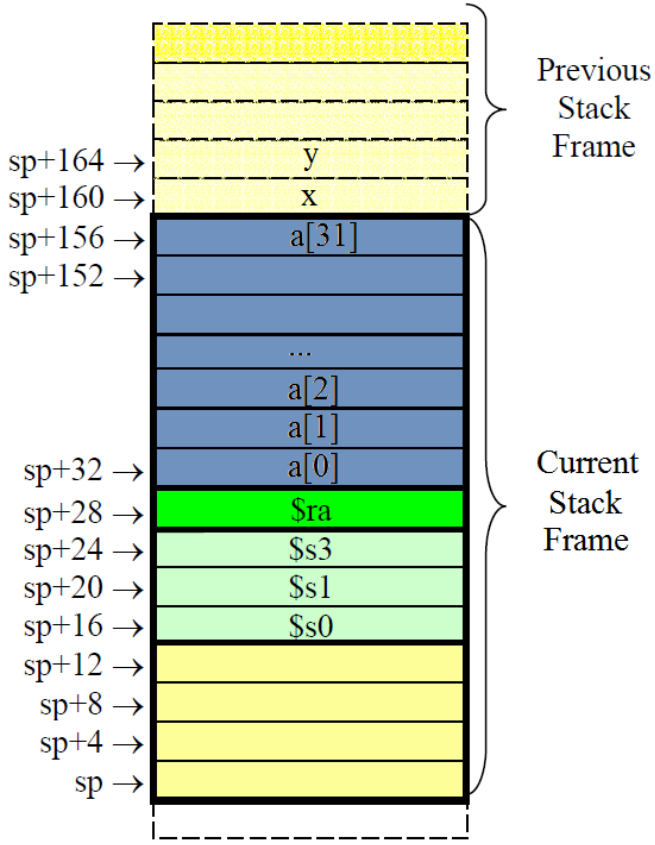
Example (*cont'd.*)

```
int g( int x, int y )
{
     int a[32];
          ... (calculate using x, y, a);
     a[1] = f(y,x,a[2]);
     a[0] = f(x,y,a[1]);
     return a[0];
}

# start of body
...       # calculate using $a0, $a1 and a
          # array a is stored at addresses
          # 32($sp) to 156($sp)
# save $a0 and $a1 in caller's stack frame
sw $a0,160(sp) # save $a0 (variable x)
sw $a1,164(sp) # save $a1 (variable y)
# first call to function f
lw $a0,164(sp) # arg0 is variable y
lw $a1,160(sp) # arg1 is variable x
lw $a2,40(sp) # arg2 is a[2]
jal f     # call f
sw $v0,36(sp) # store value of f into a[1]
```



Example (*cont'd.*)

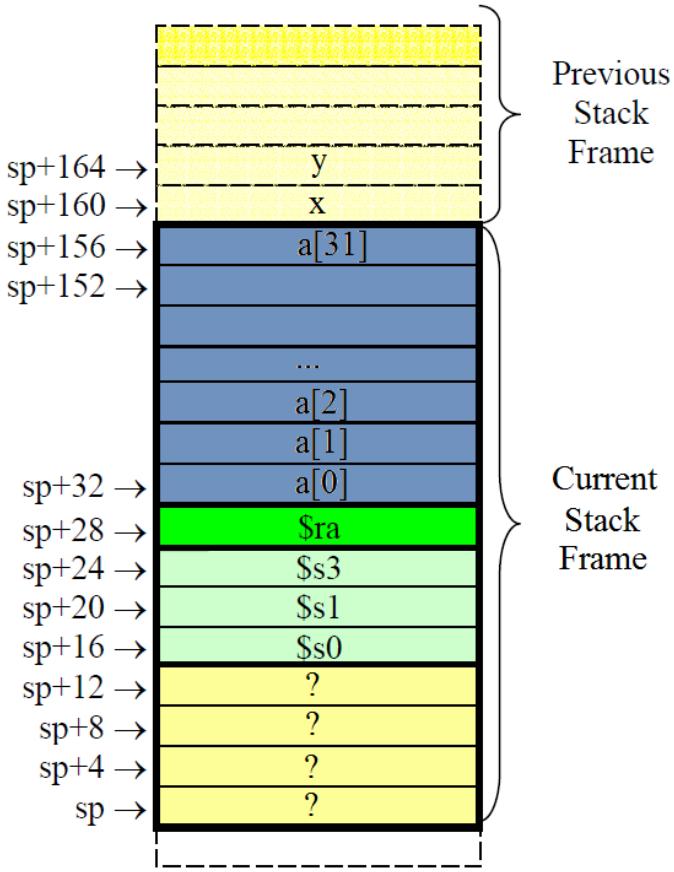
```
int g( int x, int y )
{
     int a[32];
          ... (calculate using x, y, a);
     a[1] = f(y,x,a[2]);
     a[0] = f(x,y,a[1]);
     return a[0];
}

# second call to function f
lw $a0,160(sp) # arg0 is variable x
lw $a1,164(sp) # arg1 is variable y
lw $a2,36(sp) # arg2 is a[1]
jal f # call f
sw $v0,32(sp) # store value of f into a[0]
# load return value of g into $v0
lw $v0,32($sp) # result is a[0]
# end of body
```



Example (*cont'd.*)

```
int g( int x, int y )
{
     int a[32];
          ... (calculate using x, y, a);
     a[1] = f(y,x,a[2]);
     a[0] = f(x,y,a[1]);
     return a[0];
}
```

```
# start of epilogue
lw $s0,16($sp) # restore value of $s0
lw $s1,20($sp) # restore value of $s1
lw $s3,24($sp) # restore value of $s3
lw $ra,28($sp) # restore the return address
addiu $sp,$sp,160 # pop stack frame
# end of epilogue
jr $ra # return
```

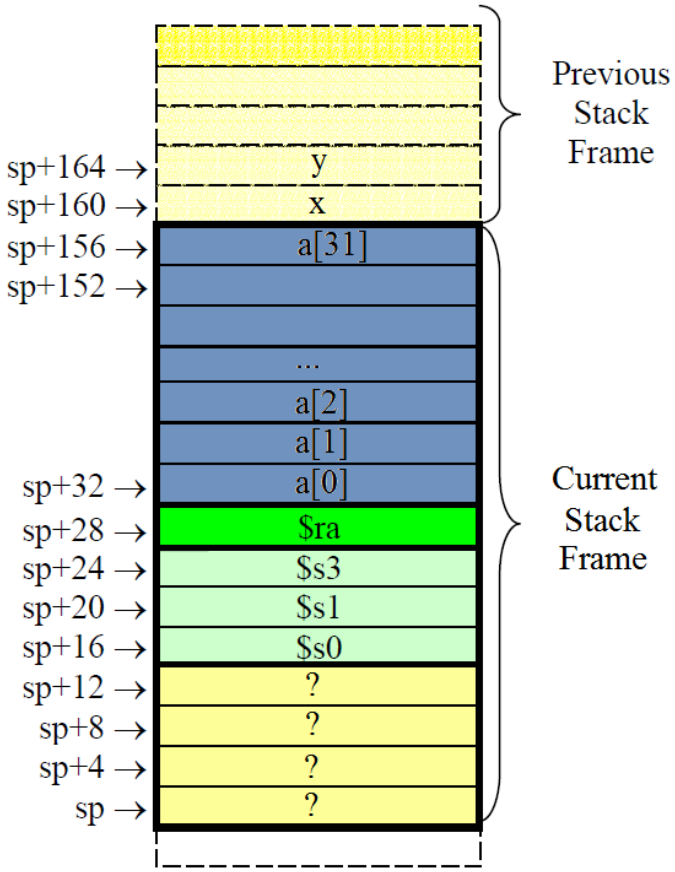


Example (*cont'd.*)

```
int g( int x, int y )
{
     int a[32];
          ... (calculate using x, y, a);
     a[1] = f(y,x,a[2]);
     a[0] = f(x,y,a[1]);
     return a[0];
}
```

```
# start of epilogue
lw $s0,16($sp) # restore value of $s0
lw $s1,20($sp) # restore value of $s1
lw $s3,24($sp) # restore value of $s3
lw $ra,28($sp) # restore the return address
addiu $sp,$sp,160 # pop stack frame
# end of epilogue
jr $ra # return
```

Is there something strange regarding the locations of *x* and *y*!?



Some Notes

- SimpleScalar uses $fp in addition to $sp
  - Why is $fp needed when $sp seems to be enough? (you can find the answer in the book)
  - In the book $fp points to the beginning of the stack frame, but in SimpleScalar …
- Pay attention to the way that arguments are passed
- .frame, .rdata, .aligned, … are called *directives*
  - They are not part of the instruction set
- You might get some empty words after filling up the stack
  - That is used for padding
  - To make the size of the stack “double-word aligned”