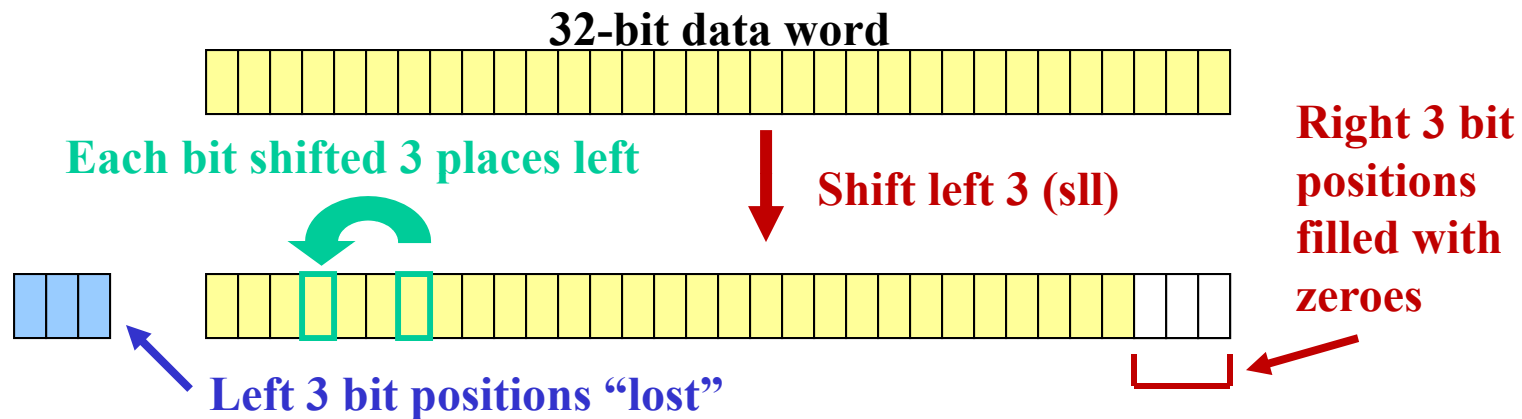




## Shift and Rotate Instructions

- **Shift and rotate instructions facilitate manipulations of data (that is, modifying part of a 32-bit data word).**
- **Such operations might include:**
  - Re-arrangement of bytes in a word
  - “Quick” divide or multiply by 2, 4, or any number =  $2^{\pm n}$
  - “Masking” – Adding or deleting certain fields of a word
- **Assume that we wish to multiply by a power of 2:**
  - **Multiplying by  $2^n$  in binary is similar to multiplying by  $10^n$  in decimal; add n zeroes on the right end of the number.**
  - **We do this by shifting the number in the register n places left.**
  - This “ $\times 2^n$ ” function is sll \$rd, \$rs, n. (Here, sll = “shift left logical,” \$rd is the destination register, \$rs the source, and n the shift amount.)

## Shift Left Logical

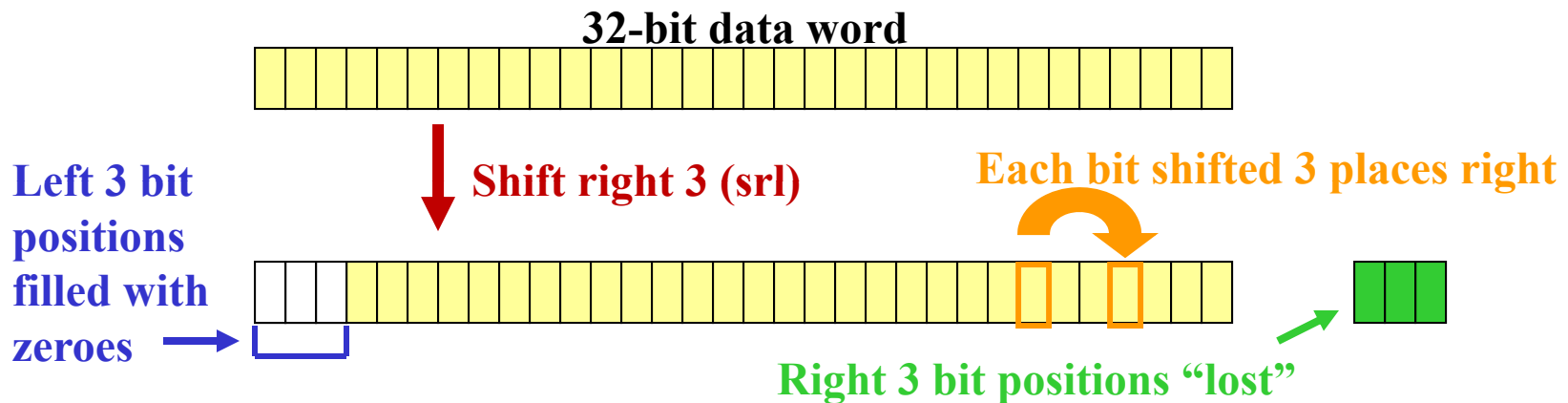


- The instruction **sll** shifts all bits in the 32-bit data word to the left the specified number of places, from 1 to 31.
- Vacated positions are automatically filled with **zeroes**.  
After an  $n$ -bit left shift, the  $n$  right positions will be 0.
- The  **$n$**  bits shifted out of the word are lost.

## The “Logical” Right Shift (srl)

- Similarly, right shift can be used to divide. This is like dividing by  $10^n$ , moving the decimal point n places left.
- We divide by  $2^n$  using srl: srl \$rd, \$rs, n, **n** the number of places to shift, **\$rs** the source, **\$rd** the destination.
- We are dealing with integer manipulation only (in EE 2310, we do not study floating-point instructions).
  - An srl will have an integer result, not true when dividing by  $2^n$ .
  - Thus we say that for a number M shifted right n places, we get  $\lfloor M/2^n \rfloor$  (where the  $\lfloor \ \rfloor$  denote the so-called “floor function,” the nearest integral value to the desired quotient).
- The n places vacated on the left in **srl** are filled with zeros, and the n bits on the right are lost.

## Shift Right Logical



- The MIPS instruction **srl** shifts all the bits in the 32-bit data word to the right from 1 to 31 places.
- Vacated positions are filled with **zeroes**. At the end of an n-bit right shift, the n left positions will be 0.
- Bits shifted out are eliminated. After an n-bit right shift, the original n bits at the right are lost.

## Arithmetic Right Shift (sra)

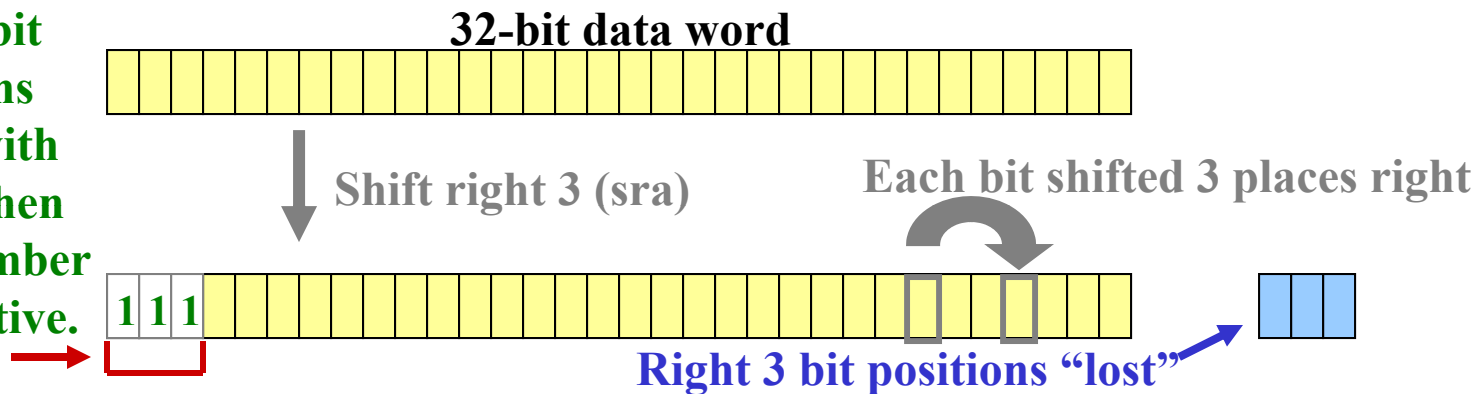
- Suppose we wish to perform the “/2” shifting function, except that our operand is a negative number.
- Suppose that we also wish to preserve the sign of the number after the shift. How would we do that?
- Consider **1111 1111 1111 1111 1111 1111 1000 0001**, a 32-bit 2’s complement number which equals  $-127$ . We still do a three-bit right-shift (i.e.,  $-127/2^3$ ), with one exception; **we will fill the empty positions with 1’s**.
  - The number  $\rightarrow$  **(111)1 1111 1111 1111 1111 1111 1111 0000**.
  - Taking the 2’s complement, the number is **[27-0’s] 1 0000**. Thus the number is  **$-16$** . But this is just the floor function of  $-127/2^3$  ( $-127/8 = 15.875, \approx -16$ ).

## Arithmetic Right Shift (2)

- For a 32-bit positive number  $M$ , when doing an `srl n` places, replacing empty bit positions with 0's, using integer MIPS instructions, always results in the floor function  $\lfloor M/2^n \rfloor$ .
- When a negative number is right shifted and the empty left bit positions are replaced with 1's, the correct floor function result for a negative number is obtained.
- This is the reason for `sra`: `sra $rd, $rs, n`.
  - `$rd` is the destination, `$rs` the source, `n` the number of places to shift.
- In `sra`, shifted-out bits on the left are replaced by 0's for a positive number, and 1's for a negative number.
- Note that there is NO arithmetic shift left.

## Arithmetic Right Shift

Left 3 bit  
positions  
filled with  
ones when  
the number  
is negative.



- **Sra** takes into account the **sign** of the number.
- If the number is **positive** (MSB=0), the shift is like srl; if **negative** (MSB=1), vacated spaces are filled with 1's.
- This preserves the sign of the number. An n-bit **sra** of a negative number is **like dividing the number by  $2^n$** , except that the "floor function" results, not the actual number, if there is a fractional remainder.



## Rotate Instructions

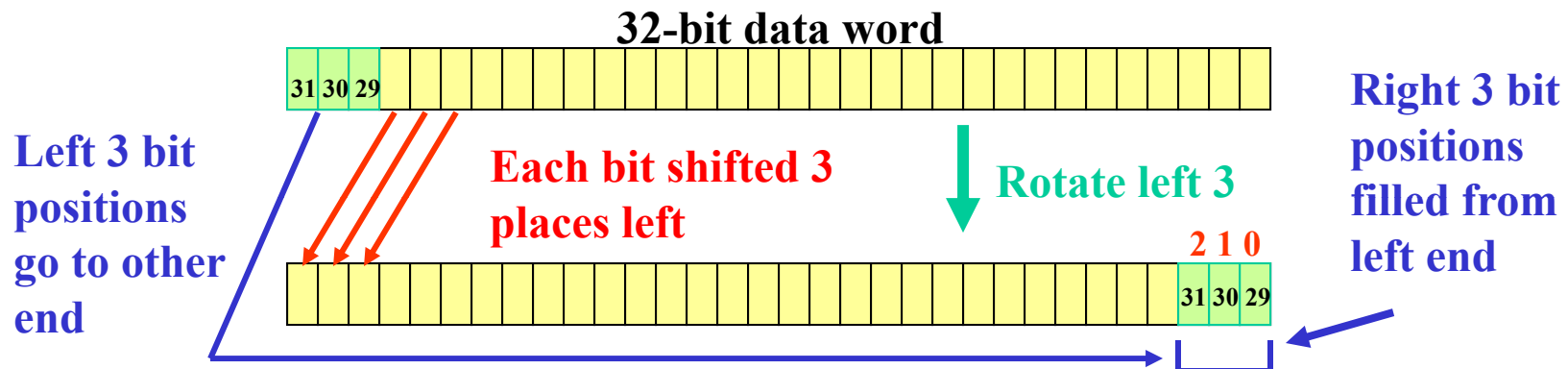
- Rotate instructions are similar to shift instructions.
- In a shift instruction, an n-bit shift to left or right results in n bits being discarded.
- **Further, the bit positions on the opposite end are vacated or filled with 0's (srl, sll) or 1's (sra only).**
- **Rotate instructions are shifts that do not eliminate bits.**
- For a left rotate (rol), bits shifted off the left end of a data word fill the vacated positions on the right.
- Likewise, for a right rotate (ror), bits "falling off" the right end appear in the vacated positions at left.
- **Note that there are NO arithmetic rotates.**



## Rotate Instructions (2)

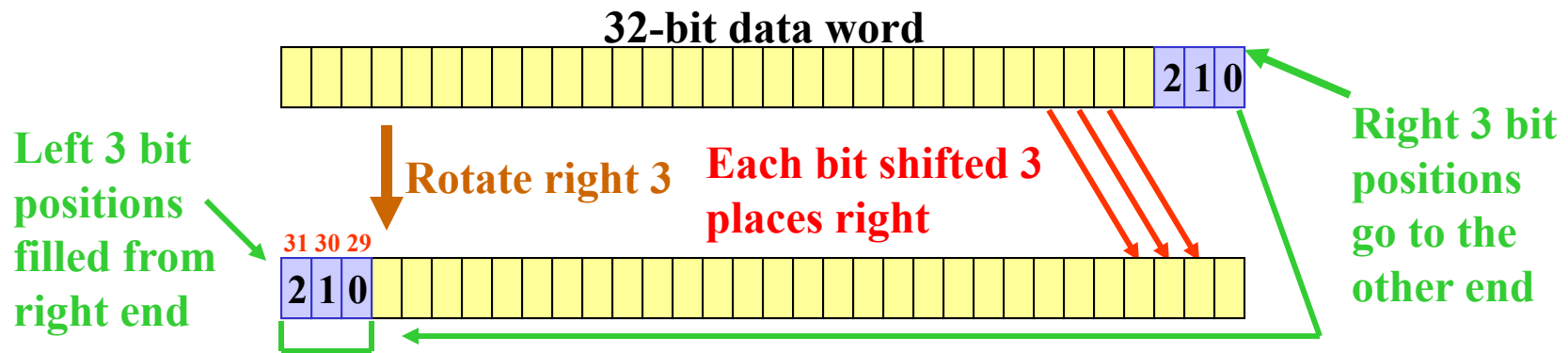
- The rotate instructions are:
  - **rol \$rd, \$rs, n** – Rotate left; rotate the contents of \$rs n bits left, and place the result in \$rd.
  - **ror \$rd, \$rs, n** – Rotate right; rotate the contents of \$rs n bits right and place the result in \$rd.
  - As usual, the contents of \$rs are not changed.
- Note that in the MIPS computer, **rol** and **ror** are pseudo instructions, which are actually performed using both left and right shifts and an OR-ing of the two resulting shifts.

## Rotate Left Diagram



- For the three-bit rotate left (rol) shown above, the three left-most bits are shifted off the left side of the data word, but immediately appear as the three right-most bits, still in the same sequence, left-to-right.
- All the other bits in the word are simply shifted left three places, just as in a shift left (sll) instruction.
- Note that no bits are lost.

## Rotate Right Diagram



- Similarly, for the three-bit ror, **the three right-most bits fall off the right side of the data word, but immediately appear as the left-most bits, still in the same sequence.**
- **All the other bits in the word are simply shifted right three places, just as in a shift right (srl) instruction.**
- Once again, we see that no bits are lost.

## Logical Instructions, Shift, Rotate, and “Masking”

- MIPS logical instructions, used with shift and rotate, **are useful in manipulating parts of a data word, and isolating specific groups of bits to examine.**
- There are five logical instructions in MIPS: **AND, OR, NOR, NOT, and XOR.**
  - **and \$rd, \$rs, \$rt\*** – the bitwise-AND of \$rs and \$rt  $\rightarrow$  \$rd.
  - **or \$rd, \$rs, \$rt** – the bitwise-OR of \$rs and \$rt  $\rightarrow$  \$rd.
  - **nor \$rd, \$rs, \$rt** – the bitwise-NOR of \$rs and \$rt  $\rightarrow$  \$rd.
  - **not \$rd, \$rs** – the bitwise-logical negation of \$rs  $\rightarrow$  \$rd.
  - **xor \$rd, \$rs, \$rt** – the bitwise-XOR of \$rs and \$rt  $\rightarrow$  \$rd.

**\* \$rt is replaced by a number in the immediate versions of all the above except NOT.**

## Masking Examples

- Consider this instruction: **andi \$t1, \$t2, 0xf.**
  - Assume \$t2 contains 0x f38d b937, or in binary:  
**1111 0011 1000 1101 1011 1001 0011 0111.**
  - The instruction is the “immediate” version of AND, so we want to AND the contents of \$t2 with 0xf or  
**0000 0000 0000 0000 0000 0000 0000 1111.**
  - Since this is a bitwise-AND of the two words, only the final four bits will produce any results other than 0 (**0·x=0**).
  - **When we AND the rightmost-four bits in \$t2 with the four bits 1111, we get (of course) 0111.**
  - The 0xf **acts as an “erase mask”** – removing all but the right four bits and giving a result of 0x 0000 0007 stored in \$t1.

## Masking Examples (2)

- A mask can also be used to add bit patterns to the contents of a 32-bit MIPS word.
- For instance, recall the pseudoinstruction **la \$a0, address**: This becomes, after assembled by SPIM:  
**lui \$at, 4097**      (0x1001 → upper 16 bits of \$at).  
**or \$a0,\$at,disp**  
where the immediate (“disp”) is the number of bytes between the first data location (always 0x 1001 0000) and the address of the first byte in the string.
- The OR instruction is used to combine the 16 bits in \$at with the lower 16 bits (“disp”) into \$a0. Here, the masking function adds bits rather than removing them.



## Example: Examining Word Segments

- Suppose we want to examine a byte in memory. We want to print the two hex digits that make up the byte.
- To do so we might do as follows:
  - `lb $t0, add1` \*      \* `add1` is the address of the given byte in memory
  - `and $a0, $t0, 0xf`
  - `li $v0, 1`
  - `syscall`
  - `ror $t1, $t0, 4`
  - `and $a0, $t1, 0xf`
  - `li $v0, 1`
  - `syscall`



## Program 1

- **Write the following short program:**
  - A single data declaration – **chars: .word 21445455**
  - Write a very short program to use a rotate right instruction to output the four bytes of the word above as four ASCII characters.
  - Don't bother to make this a loop; simply write the linear instructions to output the four bytes. The program will be less than 15 instructions (including directives). Hints:
    - Use **syscall 11** for the outputs; it outputs the bottom 8 bits of \$a0 as an ASCII character.
    - Output the first character before rotating.
    - Use rotate right and output the other three characters in the correct order to get the desired output.
    - What is output?





## Subroutines or Procedures

- A subroutine or procedure (sometimes referred to as a subprogram) is a segment of a larger program that is typically relatively independent of that program.
  1. Common functions or capabilities are often required by multiple software subsystems or modules in a larger program.
  2. Rather than have each module duplicate redundant functions, common modules are created that can be called when needed by any module or subsystem of the overall program.
  3. Such reusable modules are quite common in large programs.
  4. The reuse requirement means that these modules must be carefully written.

## Subroutines (2)

- **Subroutine requirements:**
  - Defined by its inputs and outputs only (very similar to computer hardware or logic design).
  - Can be debugged using simulated inputs.
  - As long as the subroutine “meets the “spec,” it should “plug into” and operate well with the larger program.
- **Modern programming involves “hierarchical design:**
  - Large programs are structured in layers.
  - Executive layers supervise overall operation, while middle layers (“middle managers”) summon “worker” modules.
  - These lowest-layer modules perform actual functions.
  - Many of these are procedures.

## Subroutine Support in SPIM

- Many programs use procedures or subroutines to provide the desired functionality required, which are used, or “called,” as needed.
- **Writing these modules is a key part of proper design and development of many programs.**
- Compilers provide sophisticated support for procedure development.
- **SPIM, as an assembler, provides some support for procedures.**
- Two special MIPS instructions are provided to call a procedure, and return to the calling program (as we have discussed previously):
  - **jal – Operates just like jump (next instruction executed is at “label”), except that  $PC + 4 \rightarrow \$ra$ . This instruction calls the subroutine.**
  - **jr  $\$ra - [\$ra] \rightarrow PC$ ; the next instruction executed is the one after the jal instruction. This instruction returns the program to the point of the call.**



## The Stack

- There are several bookkeeping activities when calling subroutines:
  - The calling program must pass arguments (that is, data to be processed) to the procedure, and get the result(s) returned.
  - The procedure must protect existing register data, which will be required by the calling program, when it resumes.
  - Many procedures are recursive, that is, capable of calling themselves. A recursive procedure must be written very carefully.
- Possible multi-level procedure calls means that data must be preserved across procedure calls.
- The stack is ideal for this use.

## The Stack (2)

- A stack is a special area in memory that is used as a “data buffer” where data can be stored temporarily.
  - The stack usually starts at the highest memory location (which can be beneath the operating system, as is true in MIPS).
  - As items are inserted onto the stack, the list grows downward (i. e., towards lower addresses).
  - The insertion point on the stack is called the “top” (even though it is the lowest address of the items in the stack).
  - Placing data into the next available stack position is called a “**push**,” to retrieve data is called a “**pop**.”
- The stack is a “**LIFO**” (“last-in, first-out”) data buffer. The last item “**pushed**” is the first item “**popped**.”



## The Stack Pointer

- The MIPS Stack Pointer is register **\$29 (\$sp)**.
- **\$sp** always points to the top of the stack.
- In high-level languages, stack use is easy:
  - A “push” command stores data on the top of the stack.
  - A “pop” loads the last item pushed to the desired register (thus “emptying” the location).
  - As noted above, the “top of the stack” is really the bottom, a terminology problem going back to the early days of programming.
  - MIPS can address **0x ffff ffff** bytes. However, **0x 8000 0000** and up is reserved for the OS, so the stack technically starts at **0x 7fff ffff**.

## Stack Terminology and Custom

- Due to simulated OS stack storage, the stack pointer is a little lower when you start up PCSPIM : **0x 7fff effc**.
- There are two possible stack pointer conventions:
  - (1) **\$sp** points to the first empty location on the stack. This was the original convention.
  - (2) **\$sp** points to the last filled location on the stack. Dr. Pervin, author of your SPIM textbook, and some other MIPS experts prefer this convention.
  - In either case, the stack pointer points to the “top of the stack.”
- I will go with Dr. Pervin in this case. Thus, in EE 2310 we will always assume that **\$sp** points to the last filled location on the stack.



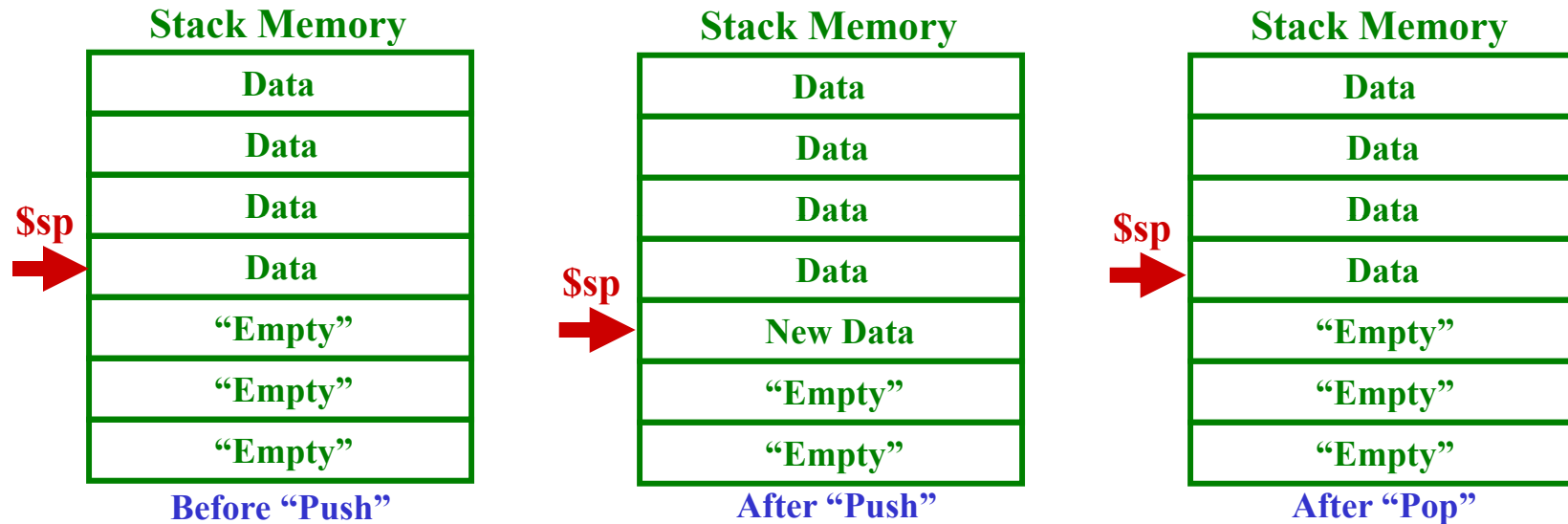
## Example of Stack Storage

- There are no “push” and “pop” commands in SPIM.
- To “push” in SPIM:\*
  - Decrement the stack pointer (e. g. : **sub \$sp, \$sp, 4**).
  - Store desired data on the stack [e. g. : **sw \$t0, (\$sp)**].
- “Pop” is simply the reverse:
  - Retrieve desired data from the stack [e. g. : **lw \$t0, (\$sp)**].
  - Increment the stack pointer (e. g. : **addi \$sp, \$sp, 4**).
- Note that words are customarily stored on the stack.

**\* Follows our “stack pointer points to the last filled location” convention.**



## View of the Stack in Action



- **"Push"\*** – **sub \$sp,\$sp,4** (pointing to empty location), **sw \$tx,(\$sp)**.
- **"Pop"** – **lw \$tx,(\$sp)** retrieves data, **addi \$sp,\$sp,4** **"empties"** memory.
- Although the "pop" location is defined as "empty," the actual data is still in that location until replaced by a subsequent push.

\* The above follows the "stack pointer points to the last filled location" convention.



## Handling Arguments When Calling a Procedure

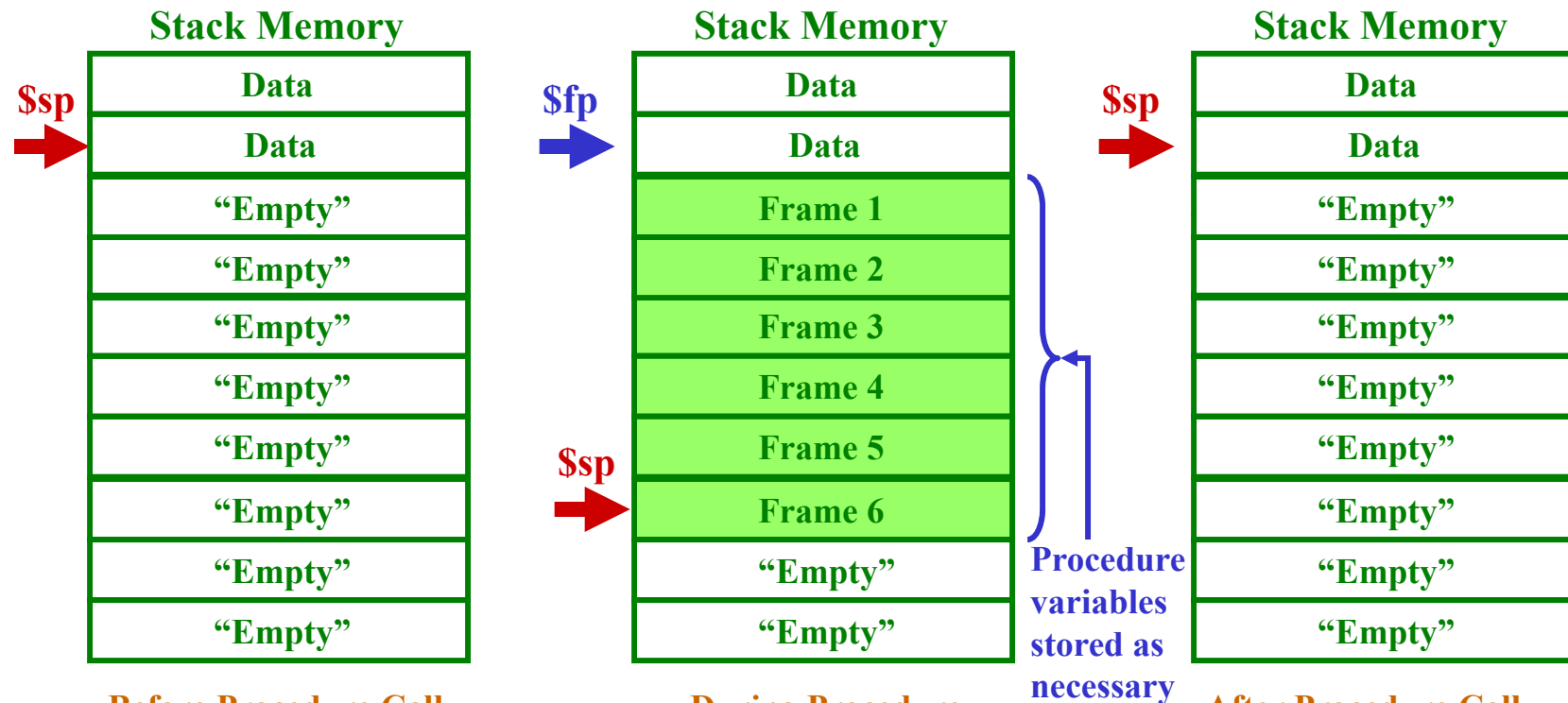
- When we studied registers, we noted that **\$a0-\$a3** were used for “passing arguments” (or data) to procedures.
- Your procedures may not be complicated enough to require parameter-passing, but you should understand the principle.
- The stack may also be used to pass data to a procedure.
- In fact, the stack is so important to procedure development that a special instruction and register have been provided to support the generation of procedure code.



## The Frame and Frame Pointer

- SPIM allows reservation of stack space.
- The frame pseudo-operation reserves space as follows:
  - .frame framereg, framesize, returnreg (example: .frame \$fp, 40, \$ra).
  - Frame size must always be the multiple of a word size (4 bytes).
  - The stack pointer is adjusted by subtracting the frame size from its current value: sub \$sp, \$sp, framesize.
  - Then the frame pointer is loaded: add \$fp, \$sp, framesize.
  - The frame is used within a procedure to pass parameters or to save register contents prior to executing procedure code, for example sw \$s1, (\$fp), where \$fp points to the first free or last filled location in the frame. \$fp is updated by sub \$fp, \$fp, 4, as for \$sp.

# Frame Construction



- The frame is an area of the stack that can be reserved, and has its own pointer. Data (such as callee-saved variables) can be stored as necessary in the stack frame. (The “point to the last filled location” convention is used above.)



## Caller- and Callee-Saved Registers

- We noted earlier that s-registers were “preserved across procedure calls.”
- That is, in SPIM there is a convention that called procedures must preserve contents of the s-registers.
- This means that the current contents of those s-registers must be saved before the registers are utilized in the procedure.
- Because of this convention, the calling program is entitled to assume that s-registers are not altered by the procedure.



## Caller- and Callee-Saved Registers (2)

- However, t-registers are fair game for a called procedure.
- Thus, after a procedure call, the calling program must assume that t-registers may have been altered.
- It is the responsibility of the calling program to preserve t-registers prior to the procedure call, or else face the possibility that t-register data may have been lost.
- Note that these are conventions which do not have to be followed. However, you ignore them at your own risk.
- Most of our procedures will be simple enough that we can ignore these rules, but you need to understand them.

## Program 2

- Declare the following four numbers in your data declaration (use “.word”).  
    num1: .word 34527  
    num2: .word 98564  
    num3: .word 12953  
    num4: .word 68577
- Then write a program to reverse their order using the stack, and print out the numbers in that reverse order.
- Remember to output a CR/LF between each number, so that each appears on a new line:

```
li $v0,11  
la $a0 0x0a  
syscall
```

} Outputs a CR/LF



## Program 3

- The preceding program was rather long, because we did not use loops to make it more compact. Using the same data:  

num1: .word 34527	num2: .word 98564
num3: .word 12953	num4: .word 68577
- Rewrite the program to reverse the order of the numbers and print out as before, but use two short loops to (a) store the words in the stack, and (b) print them out.
- You still need to output a CR/LF between numbers, so that each appears on a new line.
- Hints:
  - You will need to load the address of the first number (num1) into a register, and then address it (and the other numbers) by the indirect-register-plus-offset method.
  - You will need a counter for each loop to determine when you have stored (and later output) four numbers.