

University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Subroutines Stacks

References:

- 1) Textbook
- 2) Mark Redekopp's slide series

Shahin Nazarian

Spring 2010


Subroutines (Functions)

- Subroutines or functions are portions of code that we can call from anywhere in our code, execute that subroutine, and then return to where we left off

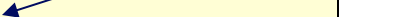
C code:

```
void main() {  
    ...  
    x = 8;  
    res = avg(x, 4);  
    ...  
}  
  
int avg(int a, int b){  
    return (a+b)/2;  
}
```

A subroutine to
calculate the average
of 2 numbers



We call the
subroutine to
calculate the
average



Subroutines (Cont.)

- Subroutines are similar to branches where we jump to a new location in the code

C code:

```
void main() {  
    ...  
    x = 8;  
    res = avg(x, 4);  
    ...  
}  
int avg(int a, int b) {  
    return (a+b)/2;  
}
```

**Call “avg” sub-routine
will require us to
branch to that code**

Normal Branches vs. Subroutines

- The main difference between normal branches and subroutines is that subroutines automatically return to location after the subroutine call

C code:

```
void main() {  
    ...  
    x = 8;  
    res = avg(x, 4);  
    ...  
}  
  
int avg(int a, int b) {  
    return (a+b)/2;  
}
```

After subroutine completes, return to the statement in the main code where we left off

2

1

Call "avg" sub-routine to calculate the average

Implementing Subroutines

- To implement subroutines in assembly we need to be able to:
 - Branch to the subroutine code
 - Know where to return to when we finish the subroutine

C code:

```
...  
res = avg(x, 4);  
...  
  
int avg(int a, int b)  
{ ... }
```



Assembly:

```
...  
jal  AVG  
...  
.org  0x0800  
AVG:  ...  
      jr  $ra
```

Jumping to a Subroutine

- JAL instruction (Jump And Link)
 - Format: **jal Address/Label** [i.e., \$ra=PC, PC=label]
 - Similar to jump: we load an address into the PC
 - Same limitations (26-bit address) as jump instruction
 - Addr is usually specified by a label
- JALR instruction (Jump And Link Register)
 - Format: **jalr \$rs** [i.e., \$ra=PC, PC=\$rs]
 - Jumps to address specified by \$rs
- In addition to jumping, JAL/JALR stores the PC into R[31] (i.e., \$ra = return address) to be used as a link to return to after the subroutine completes
- JR instruction (Jump Register)
 - Format: **jr \$rs** [i.e., PC=\$ra]

Jumping to a Subroutine

- Use the JAL instruction to jump execution to the subroutine and leave a link to the following instruction

PC before exec. of jal:

0040 0000

\$ra before exec. of jal:

0000 0000

PC after exec. of jal:

0040 0810

\$ra after exec. of jal:

0040 0004

Assembly:

```
0x400000  jal  AVG
0x400004  add
          ...

AVG: = 0x400810
          add
          ...
          jr  $ra
```

jal will cause the program to jump to the label AVG and store the return address in \$ra/\$31

Returning from a Subroutine

- Use a JR with the \$ra register to return to the instruction after the JAL that called this subroutine

PC before exec. of jr:

0040 08ec

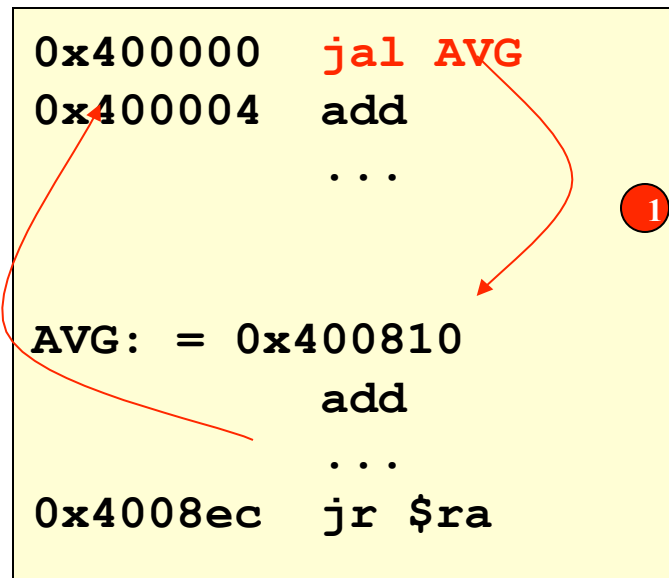
\$ra before exec. of jr:

0040 0004

PC after exec. of jr:

0040 0004

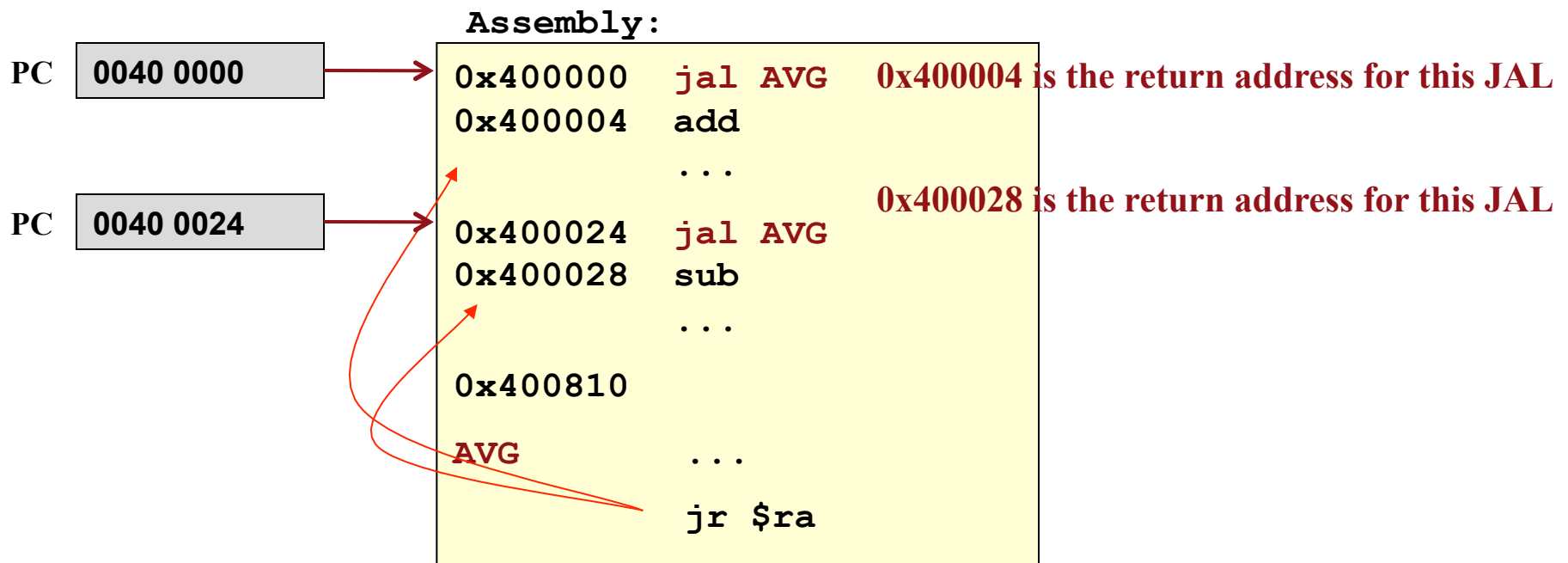
Go back to where we left off using the return address stored by JAL



jal will cause the program to jump to the label AVG and store the return address in \$ra/\$31.

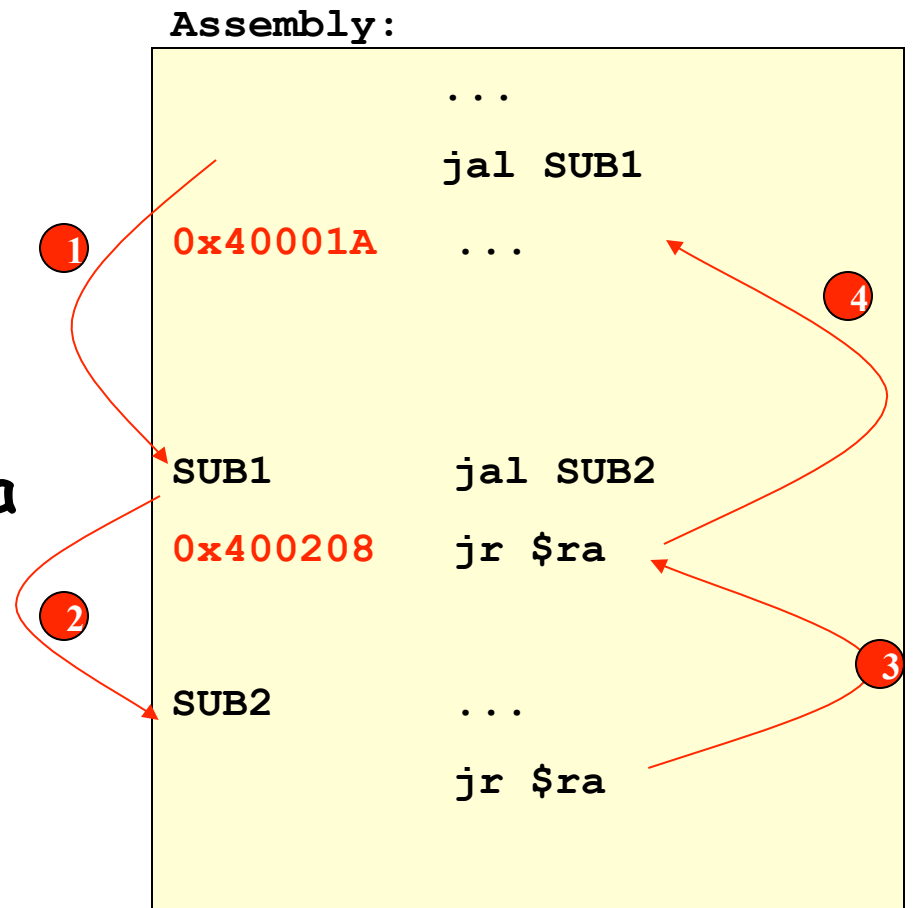
Return Addresses

- No single return address for a subroutine since **AVG** may be called many times from many places in the code
- **JAL** always stores the address of the instruction after it (i.e. PC of 'jal' location + 4)



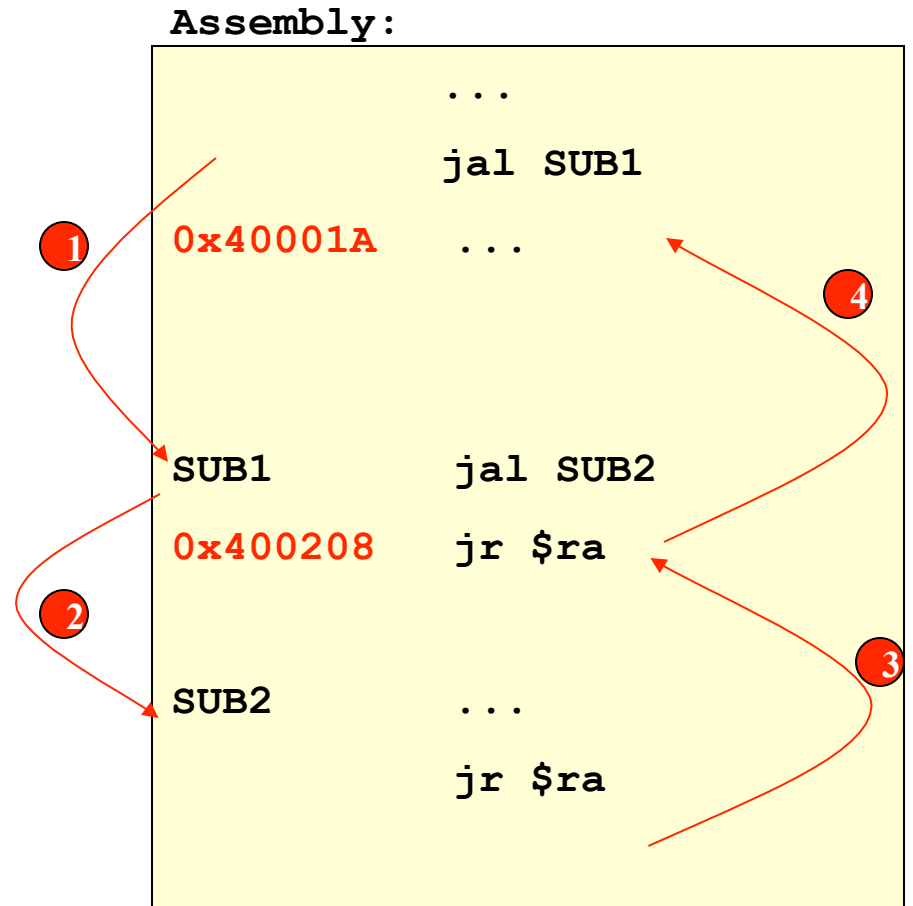
Return Addresses

- A further complication is nested subroutines (a subroutine calling another subroutine)
- Main routine calls SUB1 which calls SUB2
- Must store both return addresses but only one \$ra register



Dealing with Return Addresses

- Multiple return addresses can be spilled to memory
 - “Always” have enough memory
- Note: Return addresses will be accessed in reverse order as they are stored
 - 0x400208 is the second RA to be stored but should be the first one used to return
 - A stack is appropriate!

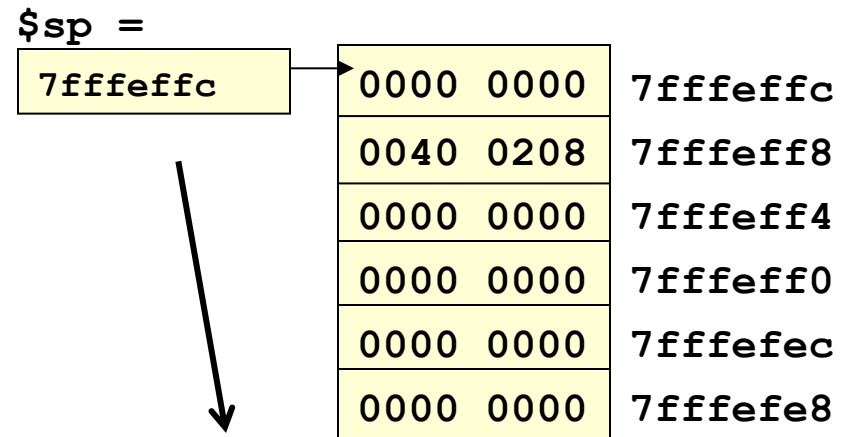


Stacks

- Stack is a data structure where data is accessed in reverse order as it is stored
- Use a stack to store the return addresses and other data
- System stack defined as growing towards smaller addresses
 - MARS starts stack at 0x7ffefffc**
 - Normal MIPS starts stack at 0x80000000**
- Top of stack is accessed and maintained using **$\$sp = R[29]$** (stack pointer)
 - $\$sp$ points at top occupied location of the stack**

Stack Pointer
Always points to top occupied element of the stack

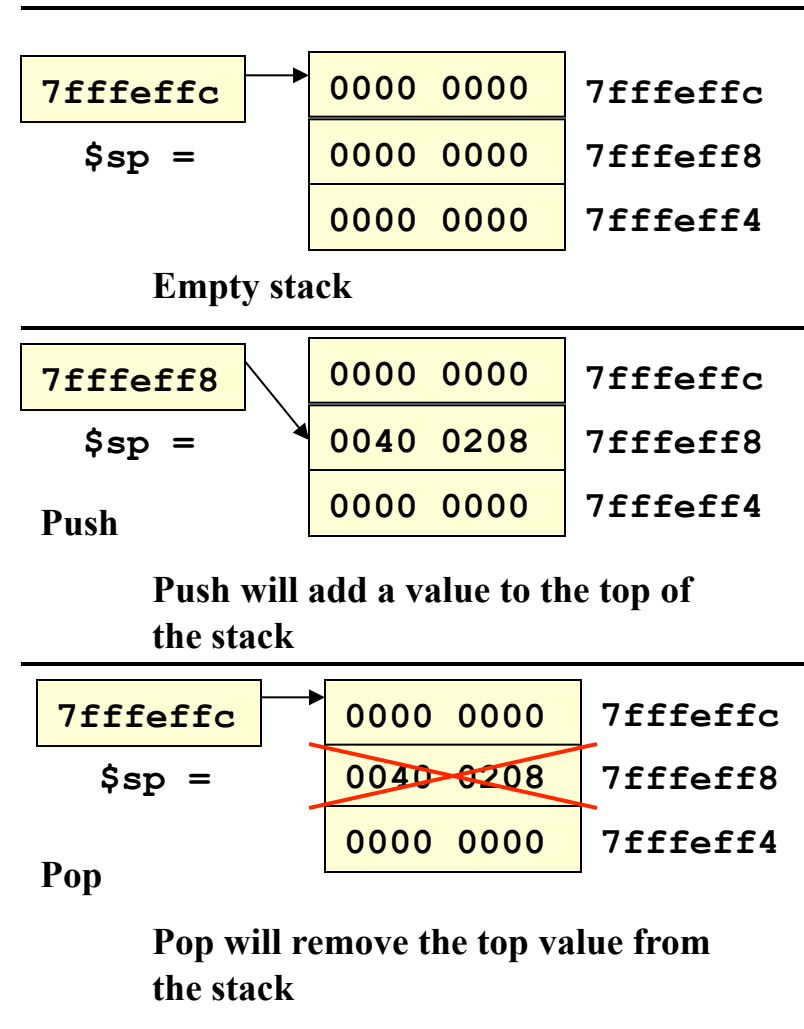
0x7ffefffc is the base of the system stack for the MARS simulator



Stack grows towards lower addresses

Stacks

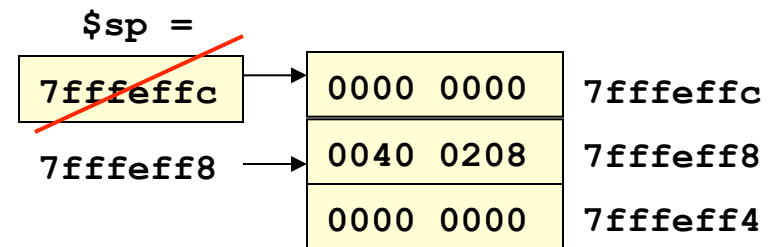
- 2 Operations on stack
 - **Push**: Put new data on top of stack
 - Decrement \$sp
 - Write value to where \$sp points
 - **Pop**: Retrieves and "removes" data from top of stack
 - Read value from where \$sp points
 - Increment \$sp to effectively "delete" top value



Push Operation

- Push: Put new data on top of stack
 - Decrement SP
 - `addi $sp,$sp,-4`
 - Always decrement by 4 since addresses are always stored as words (32-bits)
 - Write return address (\$ra) to where SP points
 - `sw $ra, 0($sp)`

Push return address (e.g.
0x00400208)

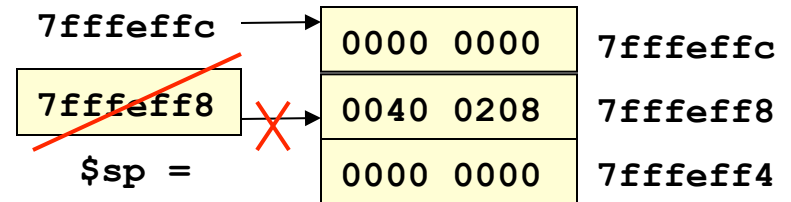


Decrement SP by 4 (since pushing a word), then write value to where \$sp is now pointing

Pop Operation

- Pop: Retrieves and “removes” data from top of stack
 - Read value from where SP points
 - lw \$ra, 0(\$sp)
 - Increment SP to effectively “delete” top value
 - addi \$sp,\$sp,4
 - Always increment by 4 when popping addresses

Pop return address



Read value that SP points at then increment SP (this effectively deletes the value because the next push will overwrite it)

Warning: Because the stack grows towards lower addresses, when you push something on the stack you subtract 4 from the SP and when you pop, you add 4 to the SP

Subroutines and the Stack

- When writing native assembly, programmer must add code to manage return addresses and the stack
- At the beginning of a routine (PREAMBLE)
 - Push \$ra (produced by 'jal') onto the stack

```
addi $sp,$sp,-4
sw   $ra,0($sp)
```
- Execute subroutine which can now freely call other routines
- At the end of a routine (POSTAMBLE)
 - Pop/restore \$ra from the stack

```
lw   $ra,0($sp)
addi $sp,$sp,4
jr   $ra
```


Subroutines and the Stack

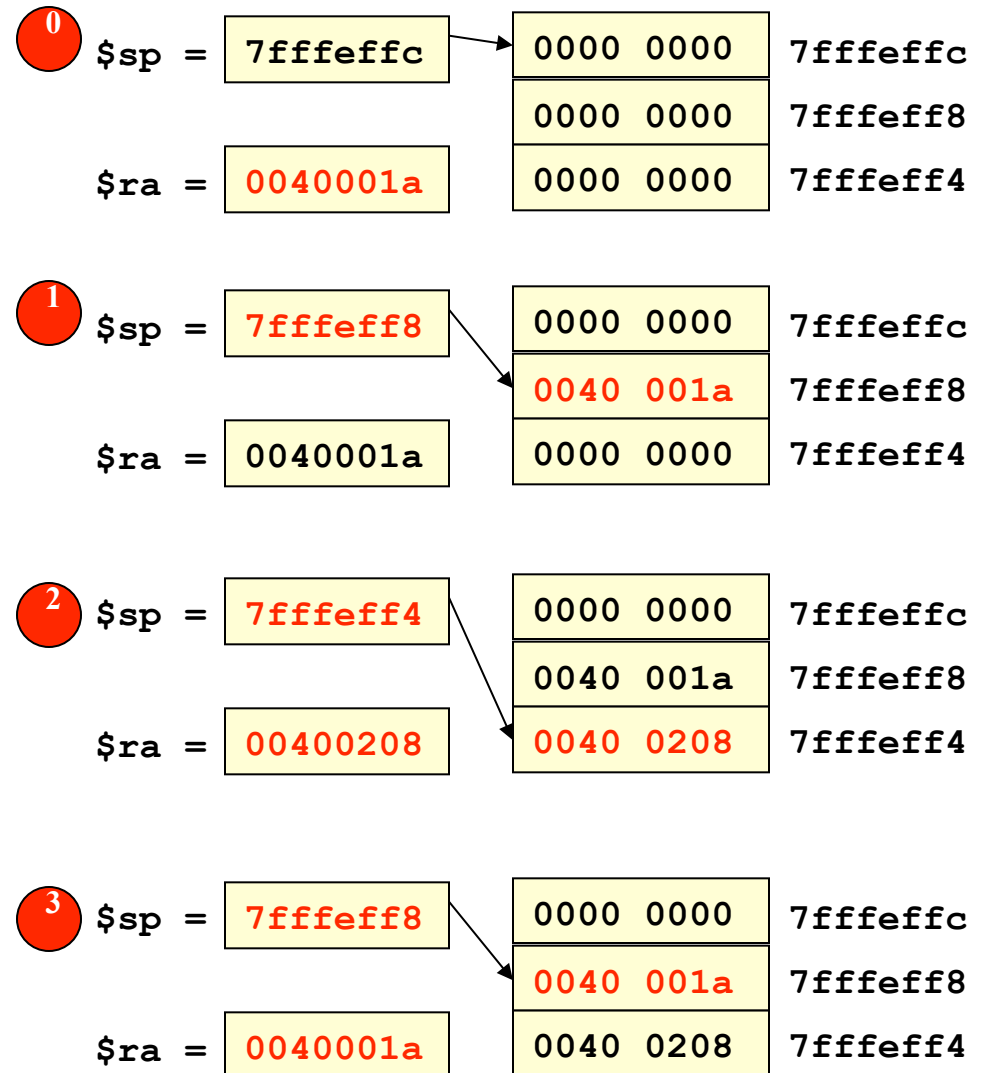
```

    ...
    jal  SUB1
0x40001A    ...

SUB1
    0 addi $sp,$sp,-4
    1 sw   $ra,0($sp)
    jal  SUB2
0x400208    lw   $ra,0($sp)
    3 addi $sp,$sp,4
    jr   $ra

SUB2
    addi $sp,$sp,-4
    2 sw   $ra,0($sp)
    ...
    lw   $ra,0($sp)
    addi $sp,$sp,4
    jr   $ra

```



Optimizations for Subroutines

- **Definition:**
 - Leaf procedure: A procedure that does not call another procedure
- **Optimization**
 - A leaf procedure need not save \$ra onto the stack since it will not call another routine (and thus not overwrite \$ra)

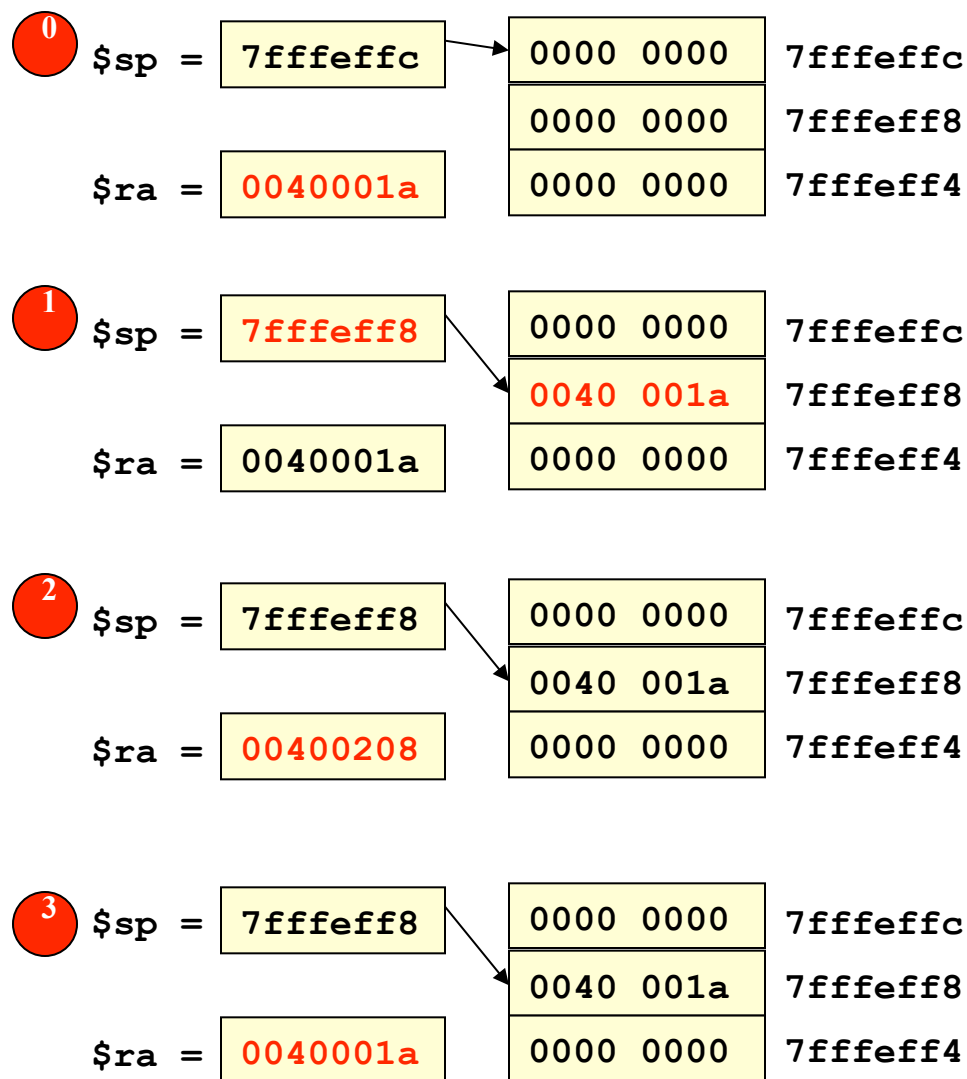
Leaf Subroutine

```

    ...
    jal    SUB1
0x40001A    ...

SUB1      0    addi $sp,$sp,-4
          1    sw    $ra,0($sp)
          1    jal    SUB2
0x400208    lw    $ra,0($sp)
          3    addi $sp,$sp,4
          3    jr     $ra

// Leaf Procedure
SUB2      2    ...
          jr     $ra
    
```



Arguments and Return Values

- Most subroutine calls pass arguments/parameters to the routine and the routine produces return values
- To implement this, there must be locations agreed upon by caller and callee for where this information will be found
- MIPS convention is to use certain registers for this task
 - \$a0 - \$a3 (R[4] - R[7]) used to pass up to 4 arguments
 - \$v0, \$v1 (R[2],R[3]) used to return up to a 64-bit value

```
void main() {  
    int arg1, arg2;  
    ans = avg(arg1, arg2);  
}  
  
int avg(int a, int b) {  
    int temp=1;  // local var's  
    return a+b >> temp;  
}
```

Arguments and Return Values

- Up to 4 arguments can be passed in \$a0-\$a3
 - If more arguments, use the stack
- Return value (usually HLL's) limit you to one return value in \$v0
 - For a 64-bit return value, use \$v1 as well

```
...  
MAIN:  li    $a0, 5  
       li    $a1, 9  
       jal   AVG  
       sw    $v0, ($s0)  
  
...  
       lw    $a0, 0($s0)  
       li    $a1, 0($s1)  
       jal   AVG  
       sw    $v0, ($s0)  
  
...  
AVG:   li    $t0, 1  
       add   $v0, $a0, $a1  
       srav  $v0, $v0, $t0  
       jr    $ra
```