University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Stack Frames

References:

1) Textbook
2) Mark Redekopp's slide series

Shahin Nazarian

Spring 2010

Arguments and Return Values

- MIPS convention is to use certain registers for this task
  - $a0 - $a3 used to pass up to 4 arguments. If more arguments, use the stack
  - $v0 used for return value
    - Only 1 return value but it may be a double-word (64-bits) in which case $v1 will also be used

Number	Name	Purpose
$0	$zero	Constant 0
$1	$at	Assembler temporary (psuedo-instrucs.)
$2-$3	$v0-$v1	Return value ($v1 only used for dword)
$4-$7	$a0-$a3	Arguments (first 4 of a subroutine)
$8-$15, $24,$25	$t0-$t9	Temporary registers
$16-$23	$s0-$s7	Saved registers
$26-$27	$k0-$k1	Kernel reserved
$28	$gp	Global pointer (static global data var's.)
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

Arguments and Return Values

```
void main() {
    int ans,arg1,arg2;
    ans = avg(arg1, arg2);

}


int avg(int a, int b) {
    int temp=1;  // local var's
    return a+b >> temp;

}
```

C Code

```
        ...

MAIN:   la   $s1, arg1
        la   $s2, arg2
        lw   $a0, 0($s0)
        lw   $a1, 0($s1)
        jal  AVG
        sw   $v0, ($s2)
        ...

AVG:    li   $t0, 1
        add  $v0,$a0,$a1
        srav $v0,$v0,$t0
        jr   $ra
```

Equivalent Assembly

Assembly & HLL's

- When coding in assembly, a programmer can optimize usage of registers and store only what is needed to memory/stack
  - Can pass additional arguments in registers (beyond $a0-$a3)
  - Can allocate variables to registers (not use memory)
  - Can handle spilling registers to memory only when necessary
- When coding in an HLL & using a compiler, certain conventions are followed that may lead to heavier usage of the stack and memory

Compiler Handling of Subroutines

- High level languages (HLL) use the stack:
  - to save register values including the return address
  - to pass additional arguments to a subroutine
  - for storage of local variables declared in the subroutine
- Compilers usually create a data structure called a "frame" on the stack to store this information each time a subroutine is called
- To access this data structure on the stack a pointer called the "frame pointer" ($fp) is often used in addition to the normal stack pointer ($sp)

Stack Frame Motivation

- The caller needs to ensure the callee routine does not overwrite a needed register
  1. Caller may have his own arguments in $a0-$a3 and then need to call a subroutine and use $a0-$a3
  2. Return address ($ra)
  3. Register values calculated before the call but used after the call (e.g. $s0)

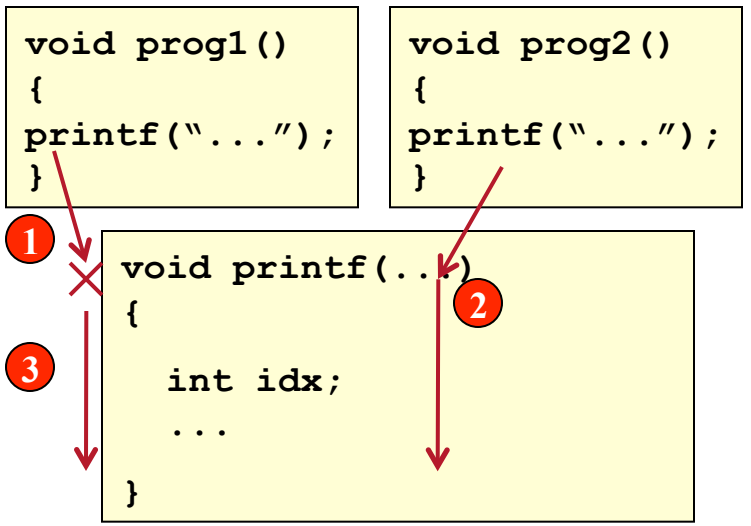
```
          .text
MAIN:     lw    $a0,0($t0)
          lw    $a1,0($t1)
          jal   CALLER
          ...
CALLER:   lw    $s0, 0($a0)     1
          addi  $s0,$s0,2       3
          li    $a0, 5          1
          jal   CALLEE          2
          add   $s0,$a0,$s0     1   3
          jr    $ra             2
CALLEE:   addi  $s0,$a0,5       3
          sra   $v0,$s0,1
          jr    $ra             2
```

More Stack Frame Motivation

- Recursive and re-entrant routines need multiple copies of their local variables
  - Recursive routines have multiple copies of arguments and local variables alive at the same time (i.e. fact(3), fact(2), fact(1) will all be live at the same time)
  - In multitasking system, multiple processes may be calling the same routine (i.e. prog1 calls printf and then blocks while prog2 executes and calls printf as well)
- The stack can be used to allow each instance of a routine having its own storage

```
int fact(int n){
  if(n == 1) return 1;
  else
    return n*fact(n-1);
}
```

Recursive Routine
(multiple copies of n)



Re-entrant Routine
(multiple copies of idx)

MIPS Register Conventions

- Highlighted registers should be "preserved" across subroutine calls (i.e. the callee must save/restore that register if it needs to use it)
- Non-highlighted registers may be overwritten freely by a subroutine
  - Thus the caller must save/restore any needed registers before/after calling the routine

Number	Name	Purpose
$0	$zero	Constant 0
$1	$at	Assembler temporary (psuedo-instrucs.)
$2-$3	$v0-$v1	Return value ($v1 only used for dwords)
$4-$7	$a0-$a3	Arguments (first 4 of a subroutine)
$8-$15, $24,$25	$t0-$t9	Temporary registers
$16-$23	$s0-$s7	Saved registers
$26-$27	$k0-$k1	Kernel reserved
$28	$gp	Global pointer (static global data var's.)
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

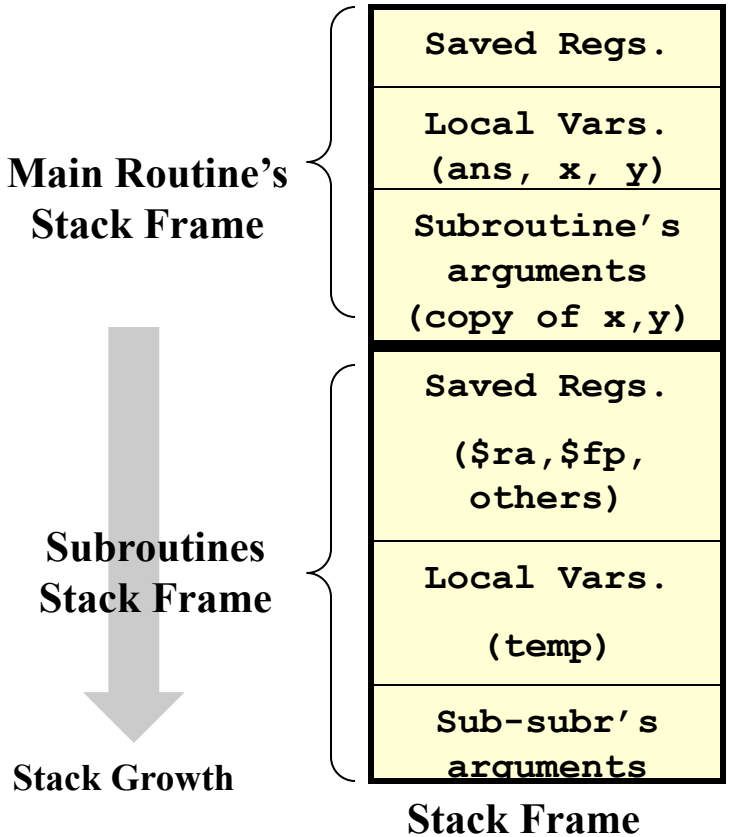
MIPS Register Storage Convention

- MIPS suggests the following convention for who is responsible for saving register values onto the stack
  - Caller is responsible for
    - Saving any $a0-$a3, $t0-$t9, $v0-$v1 it will need after the call to the subroutine
    - Stuffing $a0-$a3 with arguments for the "callee"
    - Pushing additional args. for the "callee" on the stack
  - Callee is responsible for
    - Saving $ra, $fp, & any $s0-$s7 it uses
    - Allocating space on the stack for its own local vars.

Stack Frames

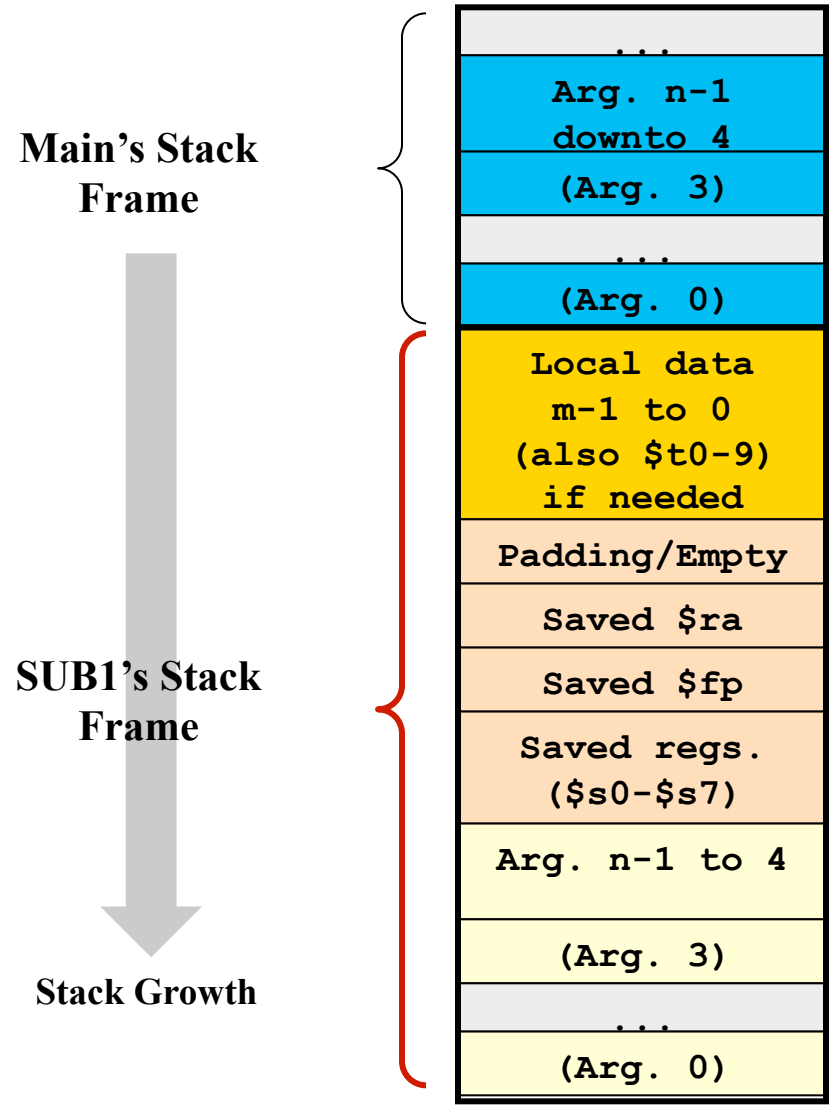
- Frame = All data on stack belonging to a routine
  - Location for arguments (in addition to $a0-$a3)
  - Location for saved registers ($fp, $s0-$s7, $ra)
  - Location for local variables (those declared in a function)
- See section A.6 in textbook and PDF handout

```
void main() {
  int ans, x, y;
  ...
  ans = avg(x, y);
}

int avg(int a, int b) {
  int temp=1;  // local var's
  ...
}
```



Stack Frame Organization

Stack Frame Organization



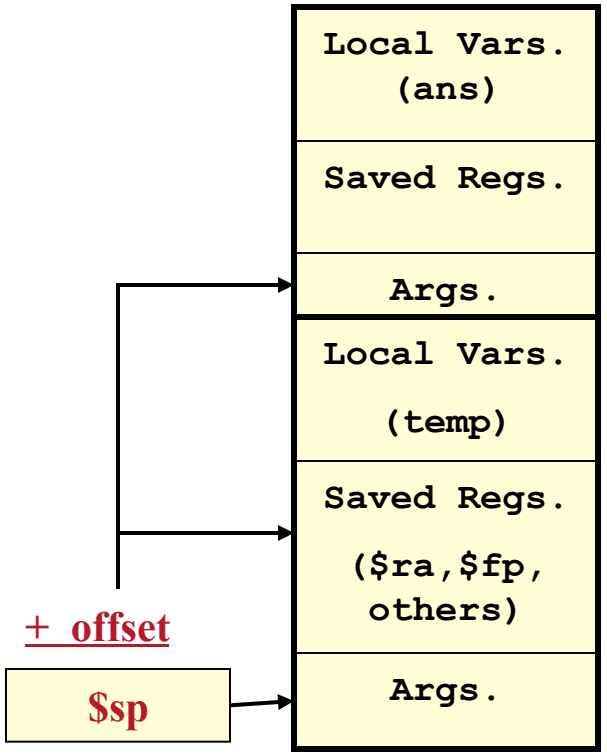
- Args. for SUB1 (or any other routine called by main)
- Arg. 0 – 3 are empty since they are passed via $a0 - $a3. Space is left for them however if they need to be saved when SUB1 calls another routine that needs arguments
- Space for any local/automatic declared variables in a routine
- Space for any "non-preserved" registers that need to be saved when SUB1 calls another routine
- Empty slot if needed because local data space must be double-word aligned (multiple of 8)
- Saved $ra / $fp. $ra not needed if leaf procedure
- Space for "preserved registers" that SUB1 will use
- Args. for any other routine called by SUB1

Building a Stack Frame

- Caller routine…
  - Save caller's "unpreserved" registers ($a0-$a3 in previous stack frame, $t0-$t9 in caller's frame)
  - Fill in arguments (using $a0-$a3 & stack)
  - Execute 'jal'
- Callee routine…
  - Allocate space for new frame by moving $sp down
  - Save "preserved registers" (i.e. $fp,$ra,etc.)
  - Setup new $fp to point to base of new stack frame
  - Execute Code
  - Place return value in $v0
  - Restore "preserved" registers
  - Deallocate stack frame by moving $sp up
  - Return using 'jr'
- Caller routine…
  - Restore "unpreserved" registers

Accessing Values on the Stack

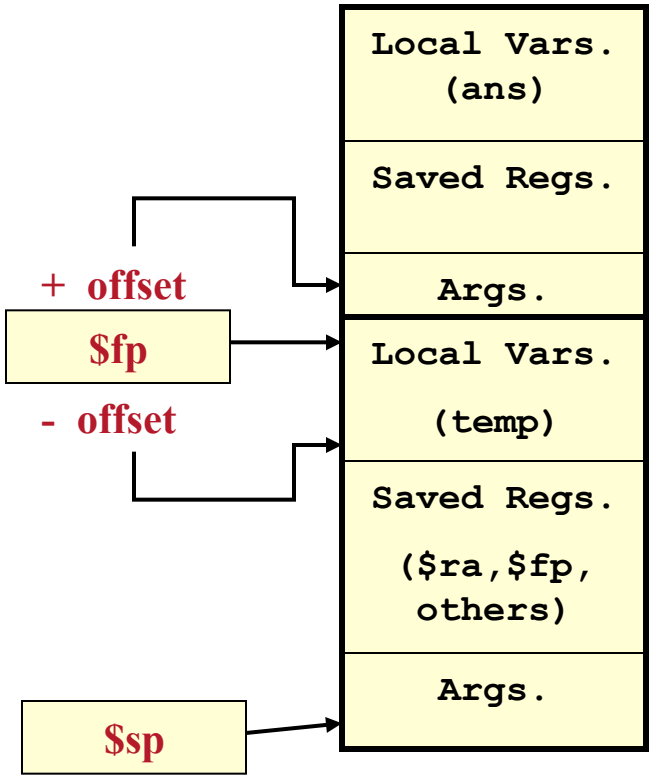
- Stack pointer ($sp) is usually used to access only the top value on the stack
- To access arguments and local variables, we need to access values buried in the stack
- We could try to use an offset from $sp, but other push operations by the callee may change the $sp requiring different displacements at different times for the same variable
- This can work, but can be confusing
- Solution: Use another pointer that doesn't change throughout execution of a subroutine



To access parameters we could try to use some displacement [i.e. d($sp)]

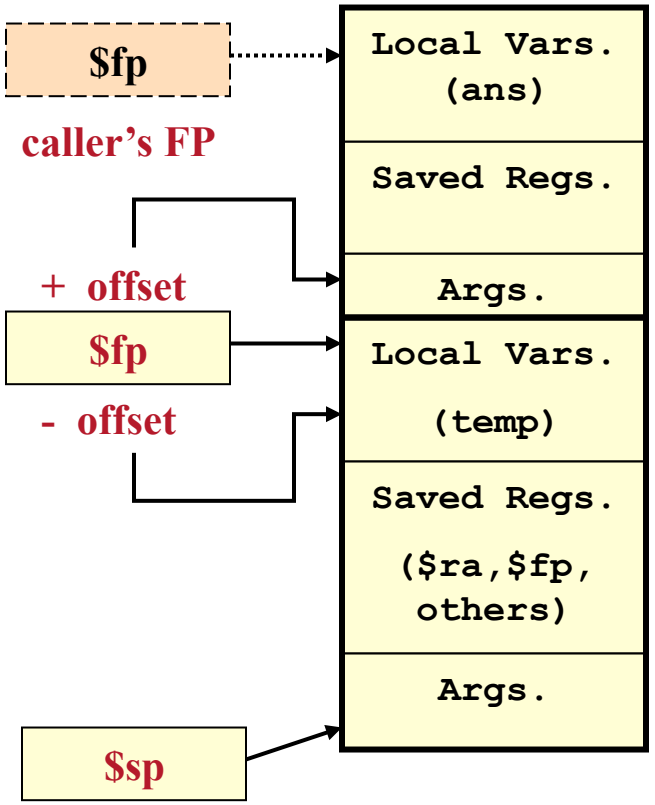
Frame Pointer

- Use a new pointer called Frame Pointer ($fp) to point to the base of the current routines frame (i.e. the *first* word of the stack frame)
- $fp will not change during the course of subroutine execution
- Can use constant offsets from $fp to access parameters or local variables
  - Key 1: $fp doesn't change during subroutine execution
  - Key 2: Number of arguments, local variables, and saved registers is a known value



Frame Pointer and Subroutines

- Problem is that each executing subroutine needs its own value of $fp
- The called subroutine must save the caller's $fp and setup its own $fp
  - Usually performed immediately after allocating frame space and saving $ra
- The called subroutine must restore the caller's $fp before it returns



Example I

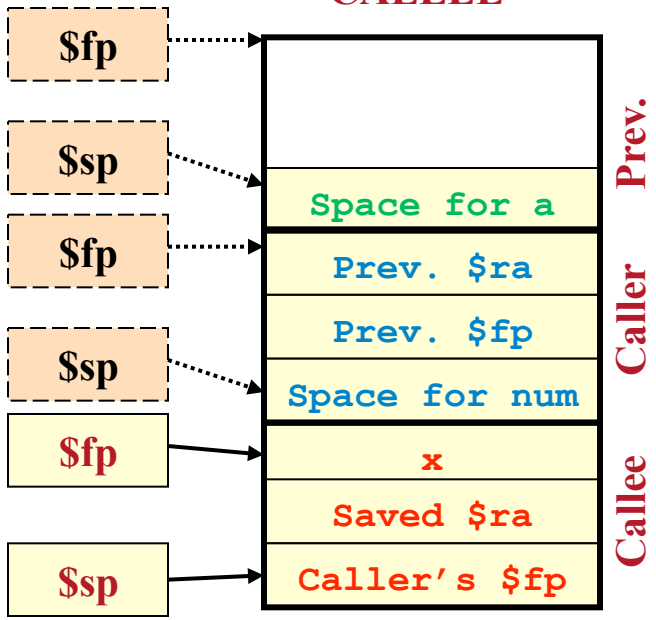
C Code

```
void caller(int a){
  callee(1);
}

int callee(int num){
  int x = 6;
  return x + num;

}
```

Assembly Code

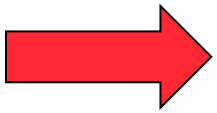
```
          .text
CALLER:   addi   $sp,$sp,-12
          sw     $ra,8($sp)
          sw     $fp,4($sp)
          addi   $fp,$sp,8
          sw     $a0,4($fp)
          li     $a0, 1
          jal    CALLEE
          lw     $a0,4($fp)
          lw     $ra,8($sp)
          lw     $fp,4($sp)
          addi   $sp,$sp,12
          jr     $ra

CALLEE:   addi   $sp,$sp,-12
          sw     $ra,4($sp)
          sw     $fp,0($sp)
          addi   $fp,$sp,8
          li     $t0,6
          sw     $t0,0($fp)
          add    $v0,$t0,$a0
          lw     $fp,0($sp)
          lw     $ra,4($sp)
          addi   $sp,$sp,12
          jr     $ra
```

Stack during execution of CALLEE



Example II

```
int ans;
void main() {
  int x = 3;
  ans = avg(1,5);
  x = x + 1;
}

int avg(int a, int b) {
  int temp = 1;
  return a + b >> temp;
}
```

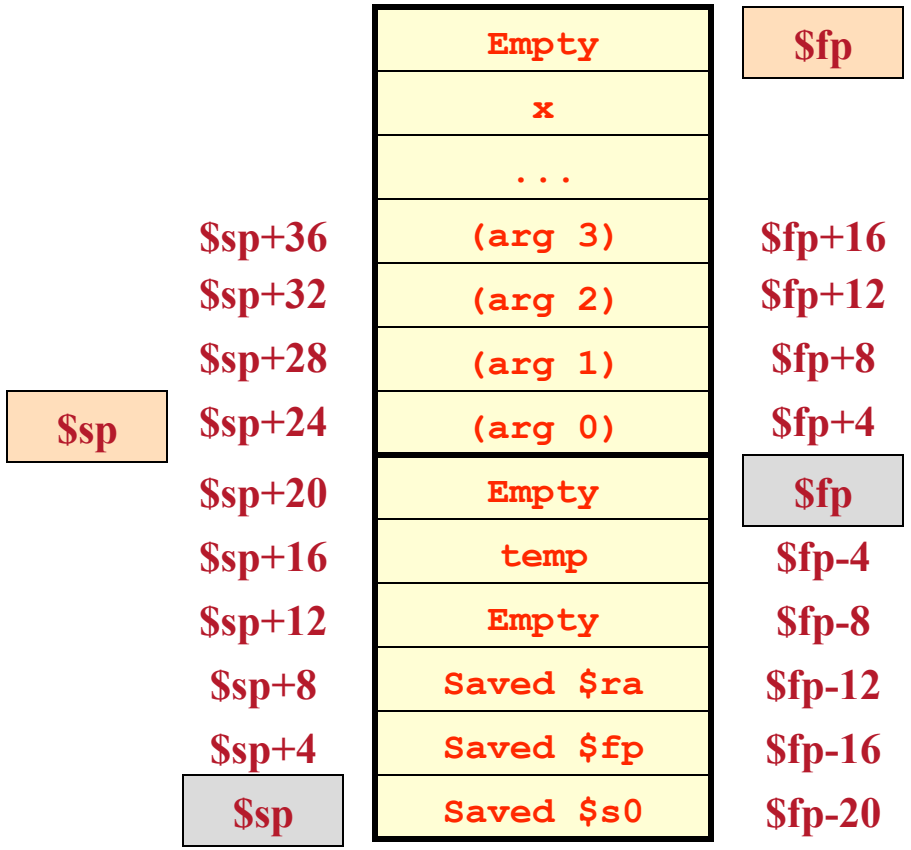


```
          .text
MAIN:     ...
          li    $s0, 3
          li    $a0, 1
          li    $a1, 5
          jal   AVG
          sw    $v0,0($gp)
          addi  $s0,$s0,1
          sw    $s0,-4($fp)
          ...

AVG:      addi  $sp,$sp,-24
          sw    $ra,8($sp)
          sw    $fp,4($sp)
          addi  $fp,$sp,20
          sw    $s0,-20($fp)
          li    $s0,1
          sw    $s0,-4($fp)
          add   $v0,$a0,$a1
          srav  $v0,$v0,$s0
          lw    $s0,-20(fp)
          lw    $fp,4($sp)
          lw    $ra,8($sp)
          addi  $sp,$sp,24
          jr    $ra
```

Example II (Cont.)

```
          .text
MAIN:     ...
          li    $s0, 3
          li    $a0, 1
          li    $a1, 5
          jal   AVG
          sw    $v0,0($gp)
          addi  $s0,$s0,1
          sw    $s0,-4($fp)
          ...

AVG:      addi  $sp,$sp,-24
          sw    $ra,8($sp)
          sw    $fp,4($sp)
          addi  $fp,$sp,20
          sw    $s0,-20($fp)
          li    $s0,1
          sw    $s0,-4($fp)
          add   $v0,$a0,$a1
          srav  $v0,$v0,$s0
          lw    $s0,-20(fp)
          lw    $fp,4($sp)
          lw    $ra,8($sp)
          addi  $sp,$sp,24
          jr    $ra
```

Convention: Local variable section must start and end on an 8-byte boundary



Stack during execution of AVG

Example III

- Subroutine to sum an array of integers

```
int sumit(int data[], int length)
{
  int sum, int i;
  sum = 0;
  for(i=0; i < length; i++)
    sum = sum + data[i];
  return sum;
}
```

Example III (Cont.)

```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp) # sum = 0;
        sw     $zero,0($fp)  # i = 0;
LOOP:   lw     $t0,0($fp)    # $t0 = i
        bge    $t0,$a0,FIN   # is i < length
        lw     $v0,-4($fp)   # $v0 = sum
        sll    $t3,$t0,2     # i = i * 4
        add    $t3,$a0,$t3   # $t3 = &(data[i])
        lw     $t4,0($t3)    # $t4 = data[i]
        add    $v0,$v0,$t4   # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1     # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```

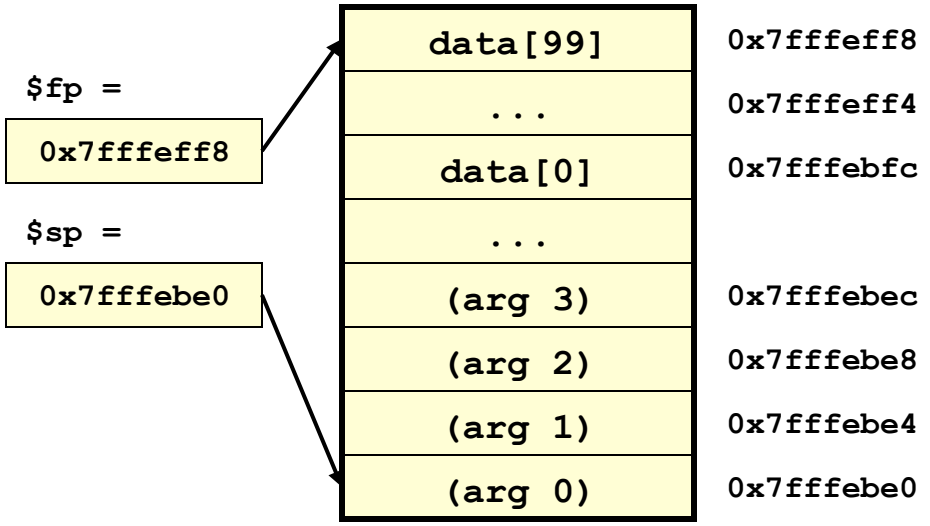
Naïve, unoptimized implementation

```
        .text
SUMIT:  move   $v0,$zero     # $v0 = sum = 0
LOOP:   blez   $a1,FIN       # check length > 0
        lw     $t1,0($a0)    # $t4 = data[i]
        add    $v0,$v0,$t1   # sum += data[i]
        addi   $a0,$a0,4     # increment ptr.
        addi   $a1,$a1,-1    # length--
        b      LOOP
FIN:    jr     $ra
```

Hand-coded assembly implementation or optimized compiler output

Example III (Cont.)

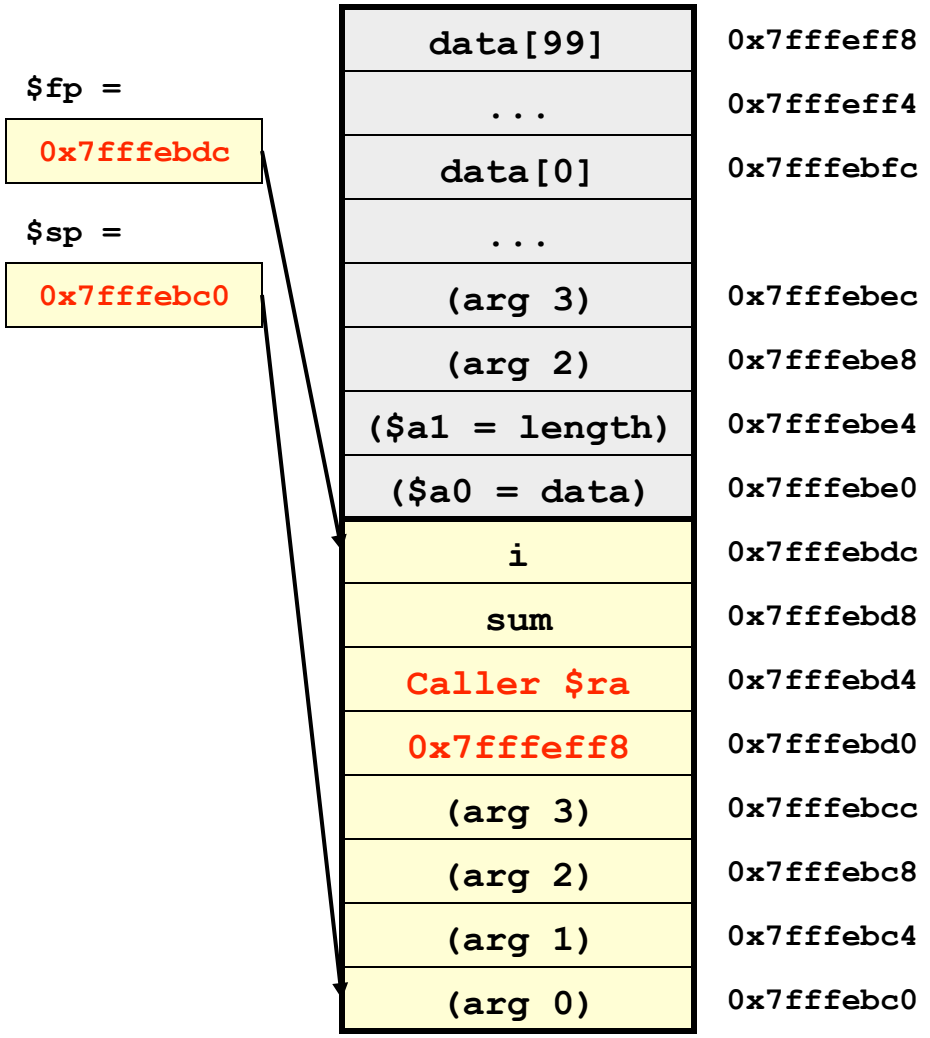
```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp)  # sum = 0;
        sw     $zero,0($fp)   # i = 0;
LOOP:   lw     $t0,0($fp)     # $t0 = i
        bge    $t0,$a0,FIN    # is i < length
        lw     $v0,-4($fp)    # $v0 = sum
        sll    $t3,$t0,2      # i = i * 4
        add    $t3,$a0,$t3    # $t3 = &(data[i])
        lw     $t4,0($t3)     # $t4 = data[i]
        add    $v0,$v0,$t4    # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1      # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```



At start of SUMIT, FP and SP are pointing to stack frame of routine that is calling SUMIT. Argument $a0 is a pointer to a locally declared data array and $a1 is the length (e.g. 100)

Example III (Cont.)

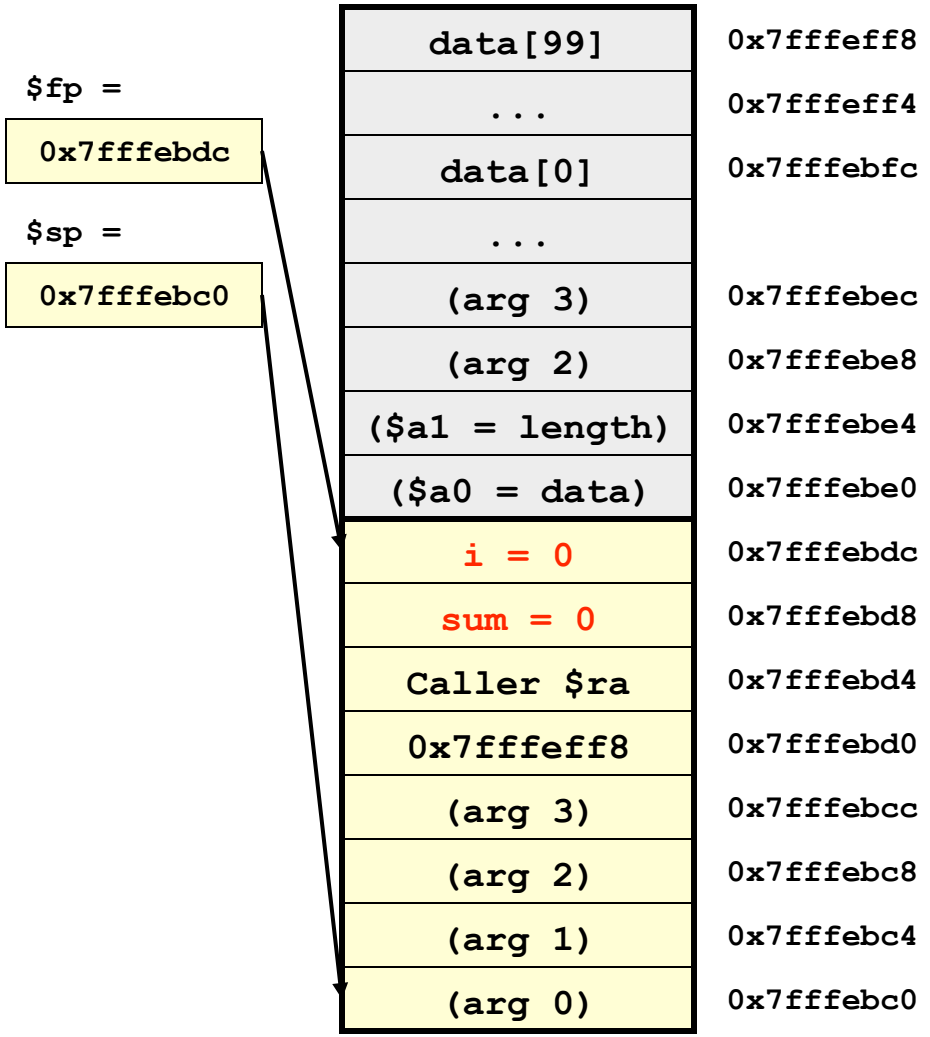
```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp)  # sum = 0;
        sw     $zero,0($fp)   # i = 0;
LOOP:   lw     $t0,0($fp)     # $t0 = i
        bge    $t0,$a0,FIN    # is i < length
        lw     $v0,-4($fp)    # $v0 = sum
        sll    $t3,$t0,2      # i = i * 4
        add    $t3,$a0,$t3    # $t3 = &(data[i])
        lw     $t4,0($t3)     # $t4 = data[i]
        add    $v0,$v0,$t4    # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1      # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```



We allocate the stack frame and save the $fp and $ra before adjusting the new $fp

Example III (Cont.)

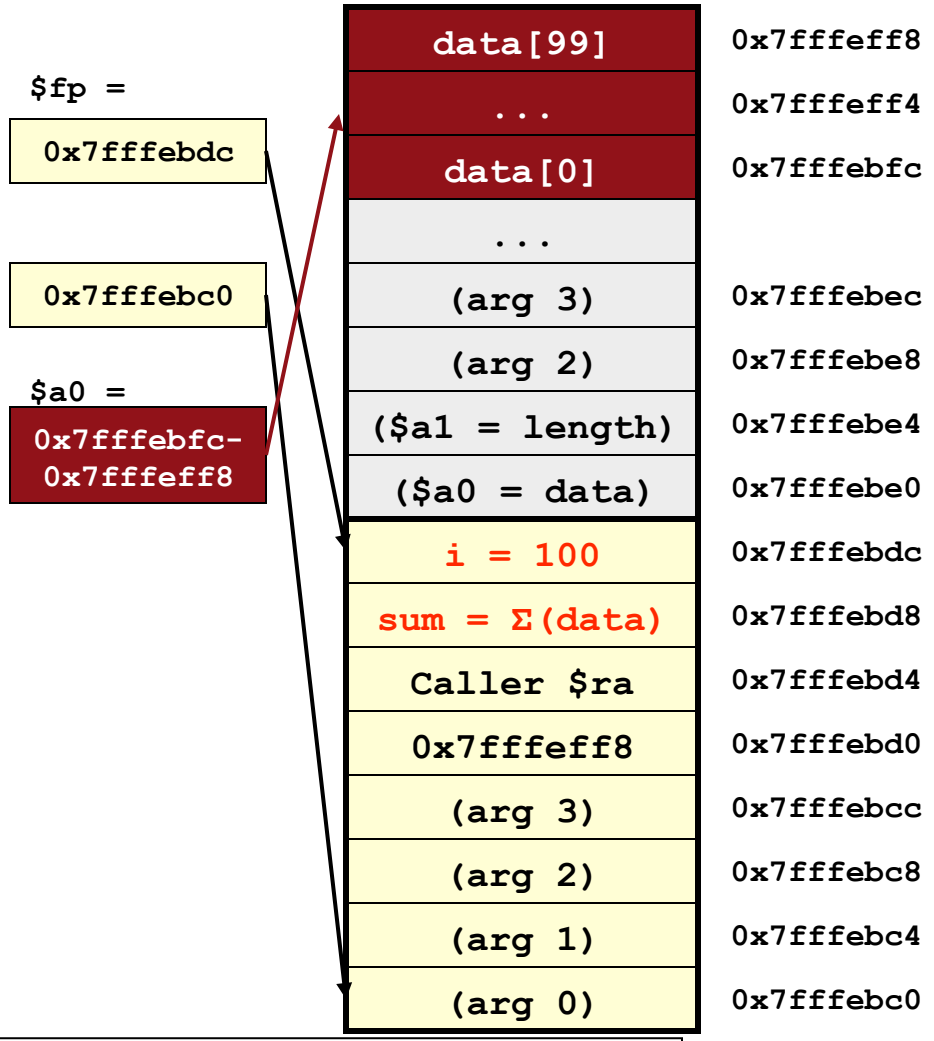
```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp)  # sum = 0;
        sw     $zero,0($fp)   # i = 0;
LOOP:   lw     $t0,0($fp)     # $t0 = i
        bge    $t0,$a0,FIN    # is i < length
        lw     $v0,-4($fp)    # $v0 = sum
        sll    $t3,$t0,2      # i = i * 4
        add    $t3,$a0,$t3    # $t3 = &(data[i])
        lw     $t4,0($t3)     # $t4 = data[i]
        add    $v0,$v0,$t4    # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1      # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```



We initialize the local variables i and sum using appropriate displacements from the $fp

Example III (Cont.)

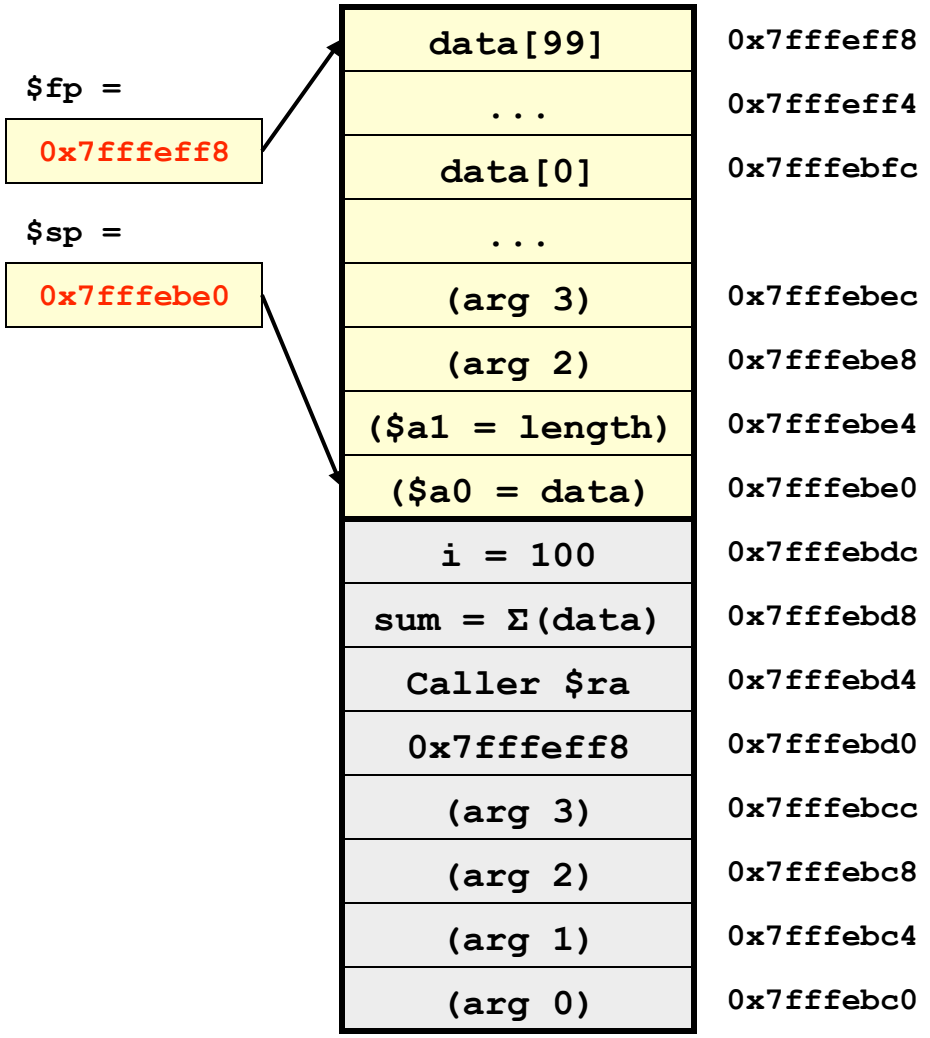
```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp)  # sum = 0;
        sw     $zero,0($fp)   # i = 0;
LOOP:   lw     $t0,0($fp)     # $t0 = i
        bge    $t0,$a0,FIN    # is i < length
        lw     $v0,-4($fp)    # $v0 = sum
        sll    $t3,$t0,2      # i = i * 4
        add    $t3,$a0,$t3    # $t3 = &(data[i])
        lw     $t4,0($t3)     # $t4 = data[i]
        add    $v0,$v0,$t4    # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1      # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```



We run through the loop, length number of times (e.g. 100) updating i and sum each iteration. $t3 acts as a ptr. to the data array allocated in the previous stack frame

Example III (Cont.)

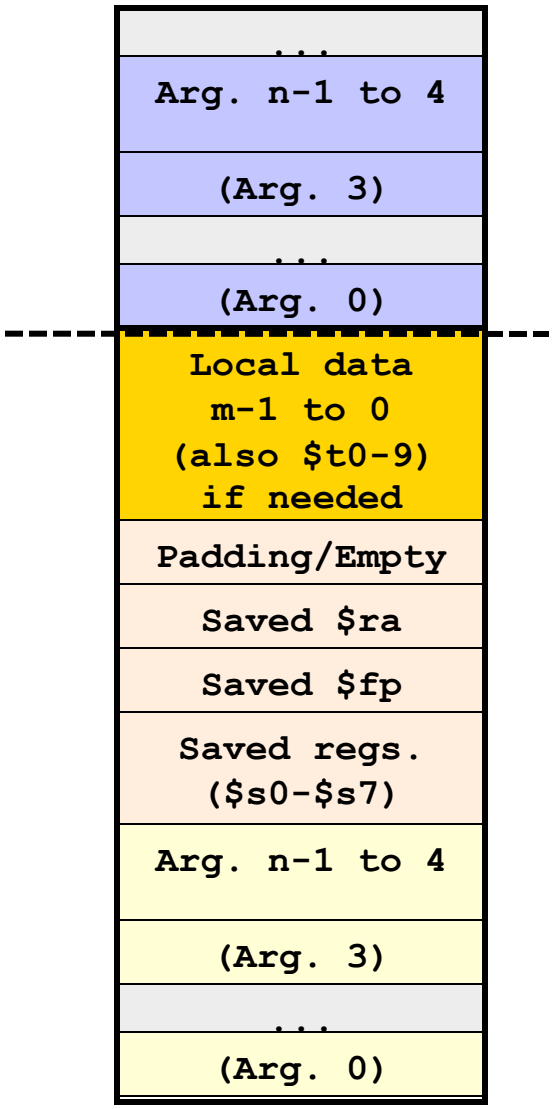
```
        .text
SUMIT:  addi   $sp,$sp,-32
        sw     $ra,20($sp)
        sw     $fp,16($sp)
        addi   $fp,$sp,28
        sw     $zero,-4($fp)  # sum = 0;
        sw     $zero,0($fp)   # i = 0;
LOOP:   lw     $t0,0($fp)     # $t0 = i
        bge    $t0,$a0,FIN    # is i < length
        lw     $v0,-4($fp)    # $v0 = sum
        sll    $t3,$t0,2      # i = i * 4
        add    $t3,$a0,$t3    # $t3 = &(data[i])
        lw     $t4,0($t3)     # $t4 = data[i]
        add    $v0,$v0,$t4    # sum += data[i]
        sw     $v0,-4($fp)
        addi   $t0,$t0,1      # i++
        sw     $t0,0($fp)
        b      LOOP
FIN:    lw     $fp,16($sp)
        lw     $ra,20($sp)
        addi   $sp,$sp,32
        jr     $ra
```



We restore the $fp and $ra, then deallocate the stack frame by resetting the $sp back to its original value

Stack Summary

- Data associated with a subroutine is a many-to-one relationship (i.e. many instances may be running at the same time). A stack allows for any number of concurrent instances to all have their own storage.
- Stack always grows towards lower addresses
- Stack frames defines organization of data related to a subroutine
- A subroutine should leave the stack & $sp in the same condition it found it
- $sp and $fp are dedicated registers to maintaining the system stack



Recursive Routines

- Routines that call themselves
- Good example of why stack frames are needed since each instance of the routine needs its own data
- In the following factorial example a compiler would use the standard stack frame organization, however, we'll hand-code our own

Recursive Factorial Routine

C Code:

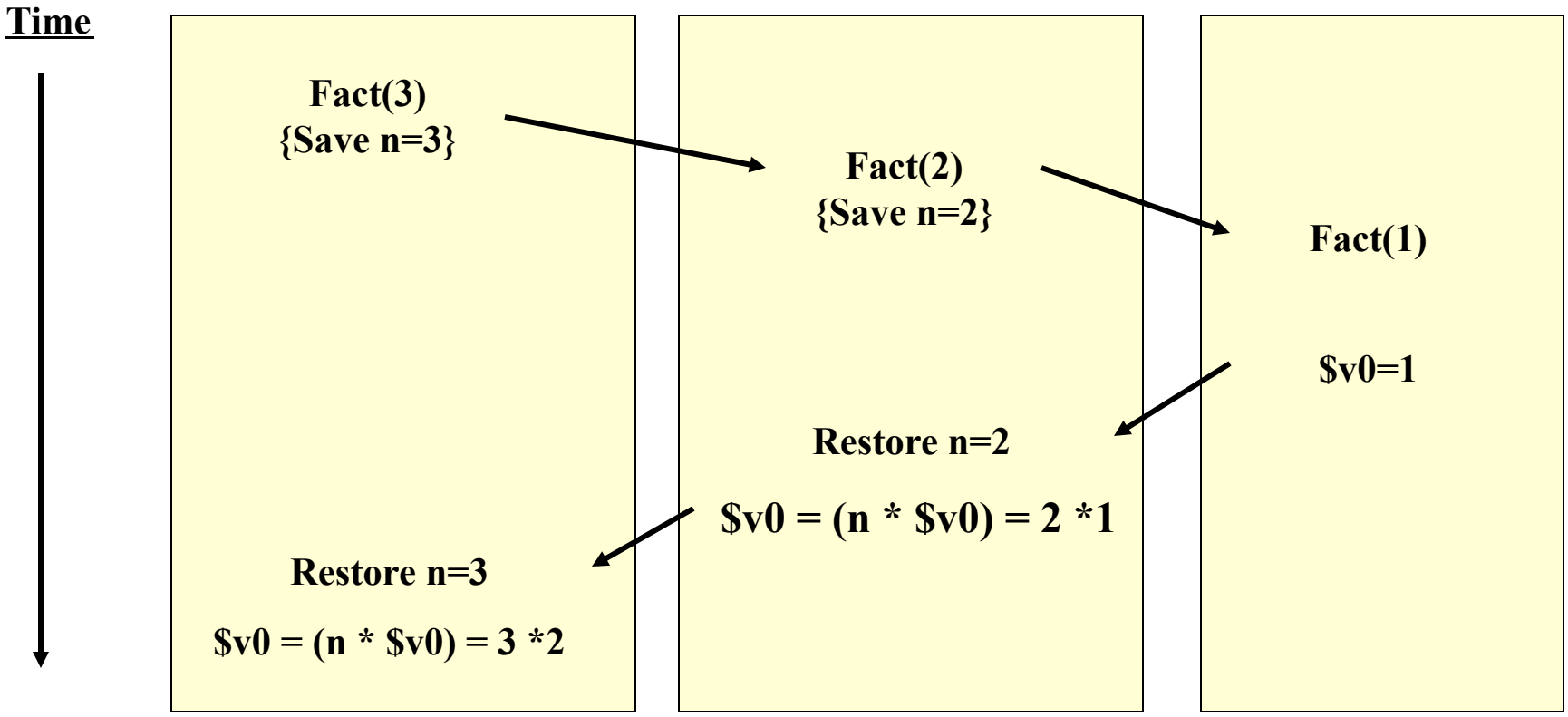
```
void main(){
    int x;
    x = fact(3);
}

int fact(int n)
{
  if(n == 1)
     // 1! = 1
     return 1;
  else {
     // calculate (n-1)!
     return n*fact(n-1);

  }
}
```

Assembly:

```
          .text
MAIN:     addi    $sp,$sp,-8
          li      $a0, 3
          jal     FACT
          sw      $v0,4($sp)

FACT:     addi    $sp,$sp,-8
          sw      $ra,4($sp)
          addi    $v0,$zero,1
          beq     $a0,$v0,L1
          sw      $a0,8($sp)
          addi    $a0,$a0,-1
          jal     FACT
          lw      $a0,8($sp)
          mul     $v0,$a0,$v0
L1:       lw      $ra,4($sp)
          addi    $sp,$sp,8
          jr      $ra
```

- Key Detail: Make sure each call of Fact is working with its own value of n

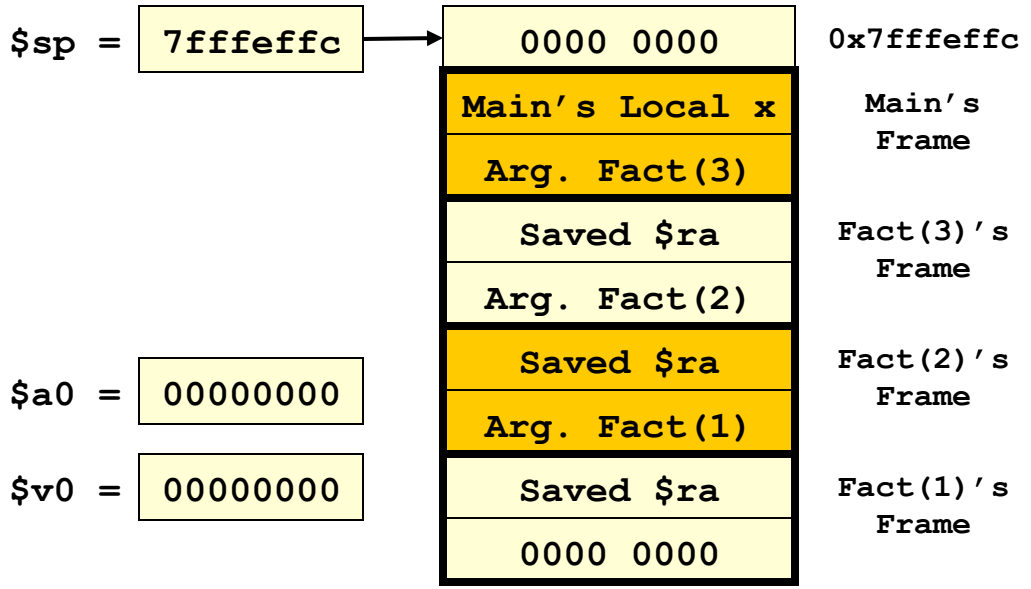
Recursive Call Timeline



- Before calling yourself, you need to save copies of all your locally declared variables/parameters (e.g. n)

Recursive Routine Example

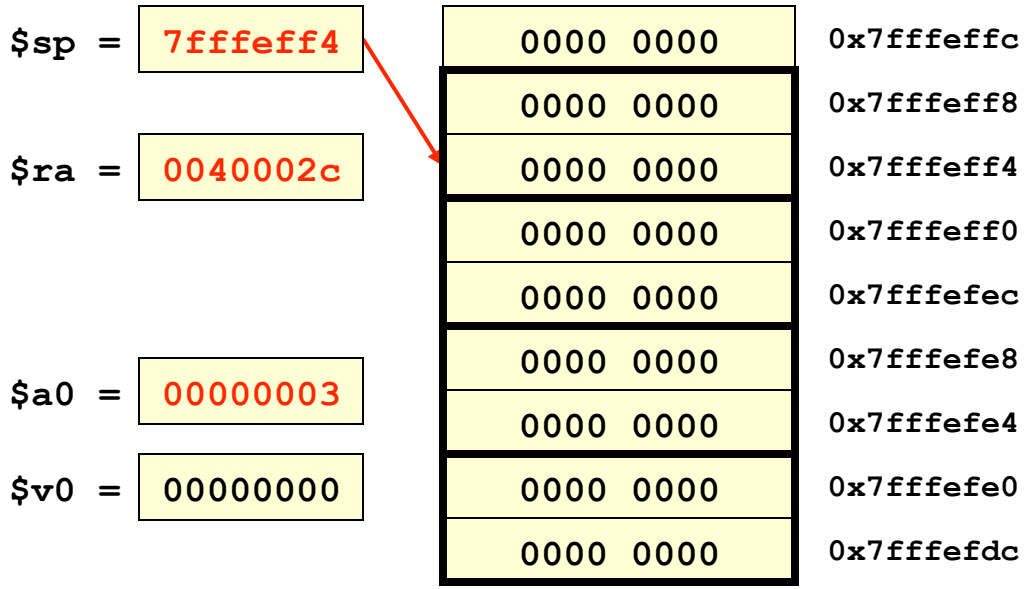
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
            jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
            lw      $a0,8($sp)
            mul     $v0,$a0,$v0
 L1:        lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- **Initial conditions**

Recursive Routine Example

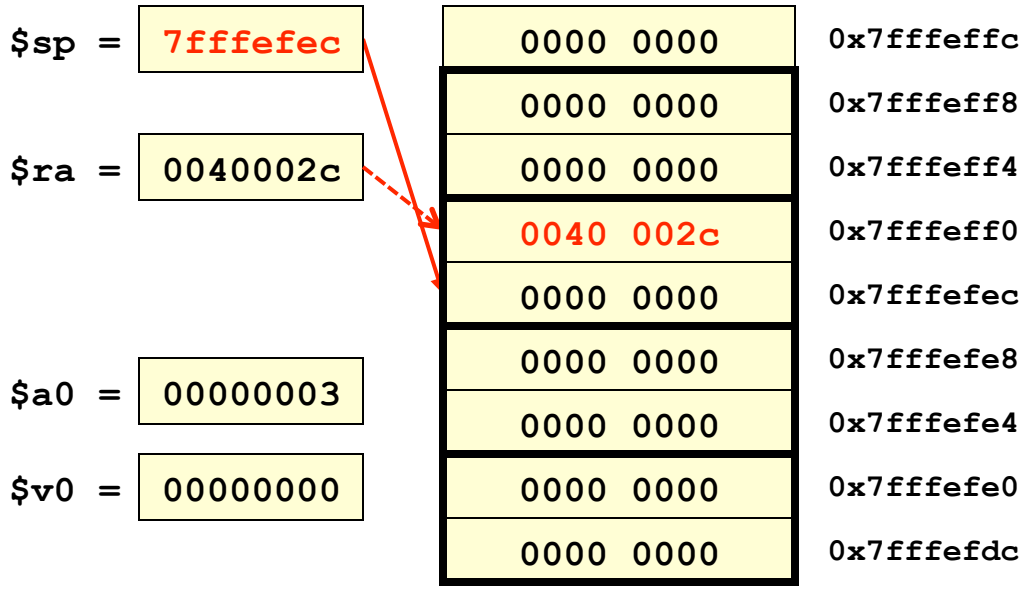
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Allocate space for local var. and arg. $a0 (n)
- Load the argument for fact(3) call
- Call the fact routine using 'jal'

Recursive Routine Example

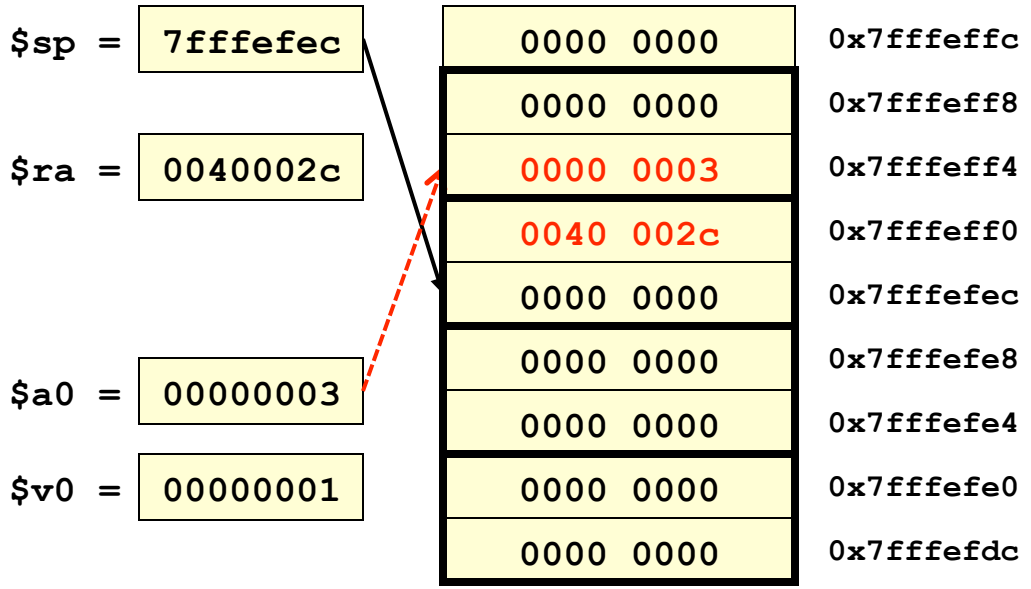
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Allocate room for $ra and for saving $a0 when we have to call fact(2)
- Store the $ra to the stack

Recursive Routine Example

```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Perform the comparison of n == 1 ($a0 == $v0)
- Since n=3, this branch fails and execution continues by storing the current value of n on the stack before calling fact(2)

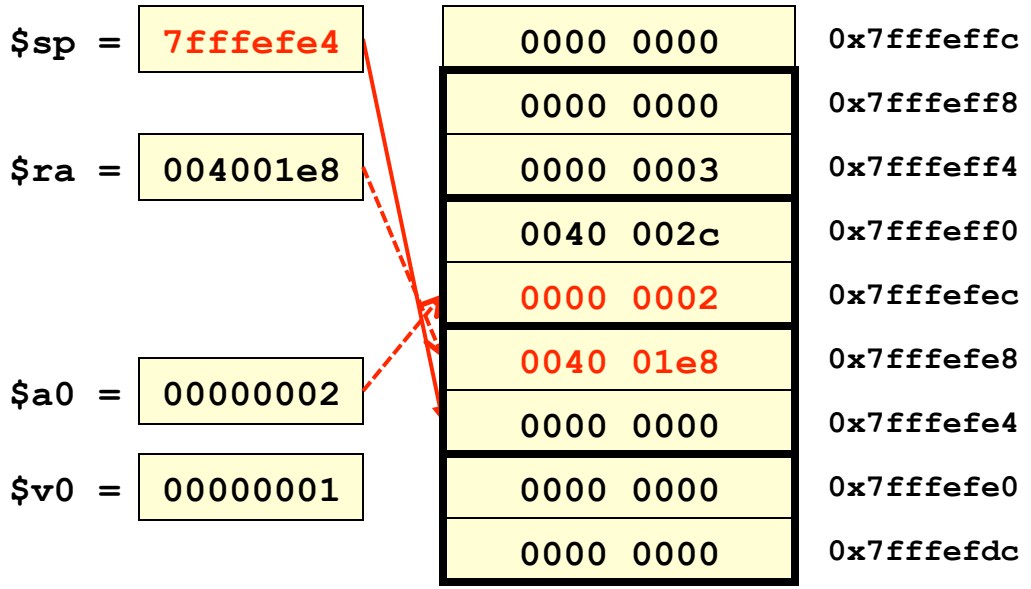
Recursive Routine Example

```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```

- Set n=2 and call fact(2)
- $ra is updated with the appropriate return address

Recursive Routine Example

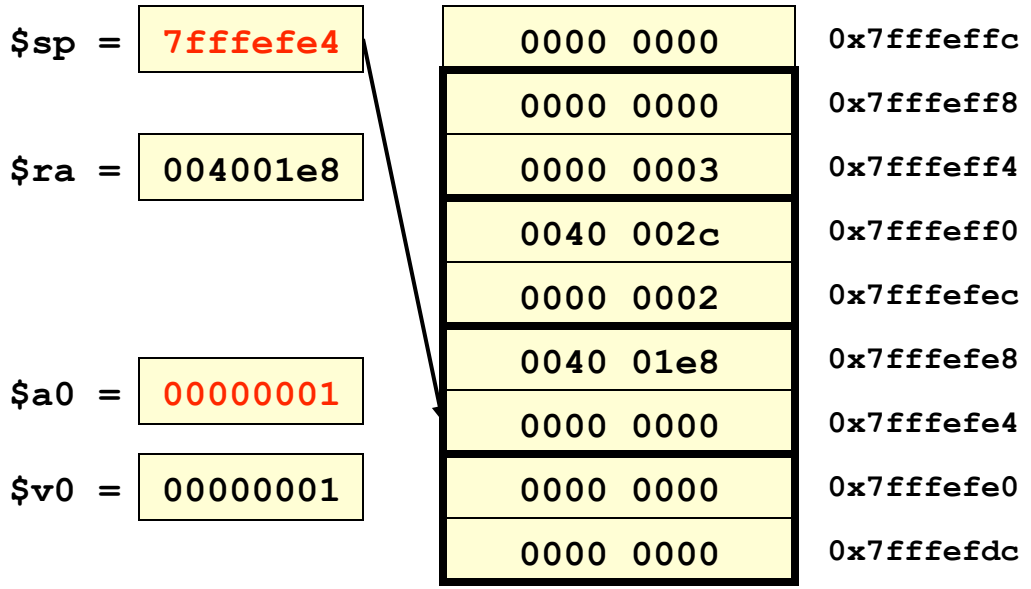
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Allocate space on stack for fact(2) call
- Save $ra, perform n==1 comparison which fails
- Store n=2 on the stack before calling fact(1)

Recursive Routine Example

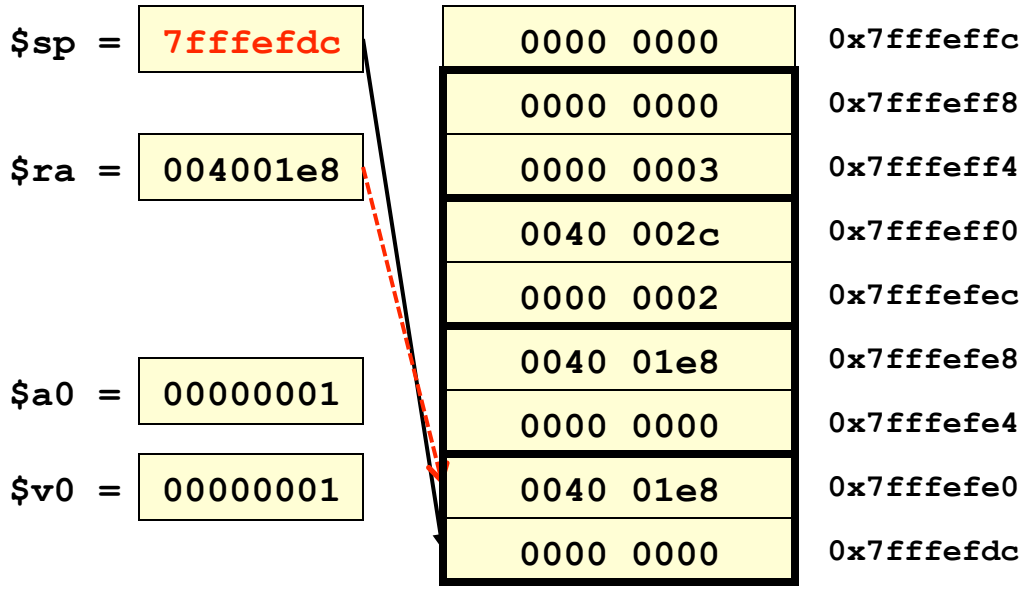
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Set n=1 and call fact(1)
- $ra is updated with the appropriate return address

Recursive Routine Example

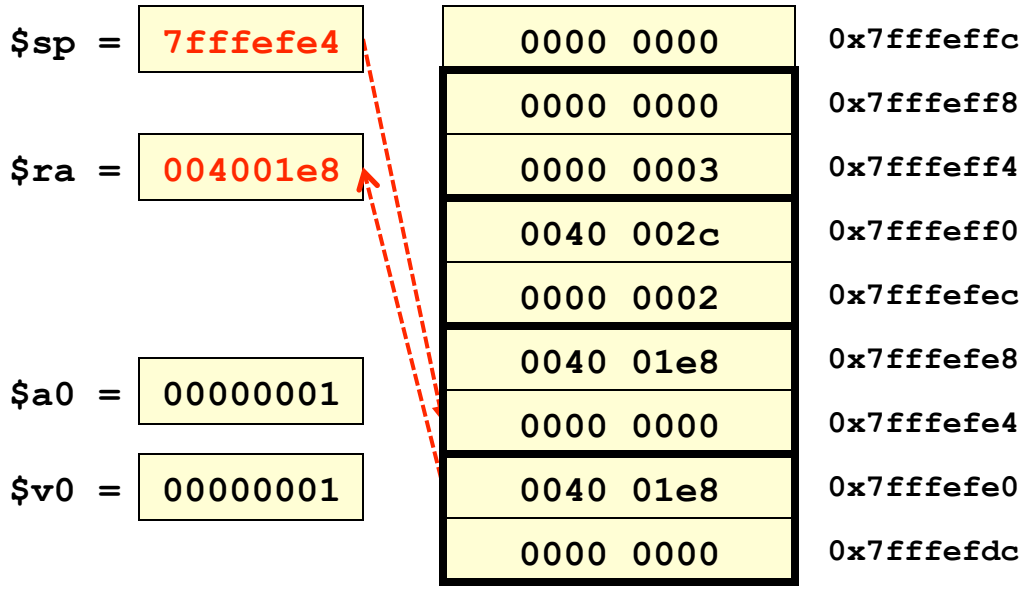
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Allocate space on stack for fact(1) call
- Save $ra, perform n==1 comparison which succeeds
- Branch to L1

Recursive Routine Example

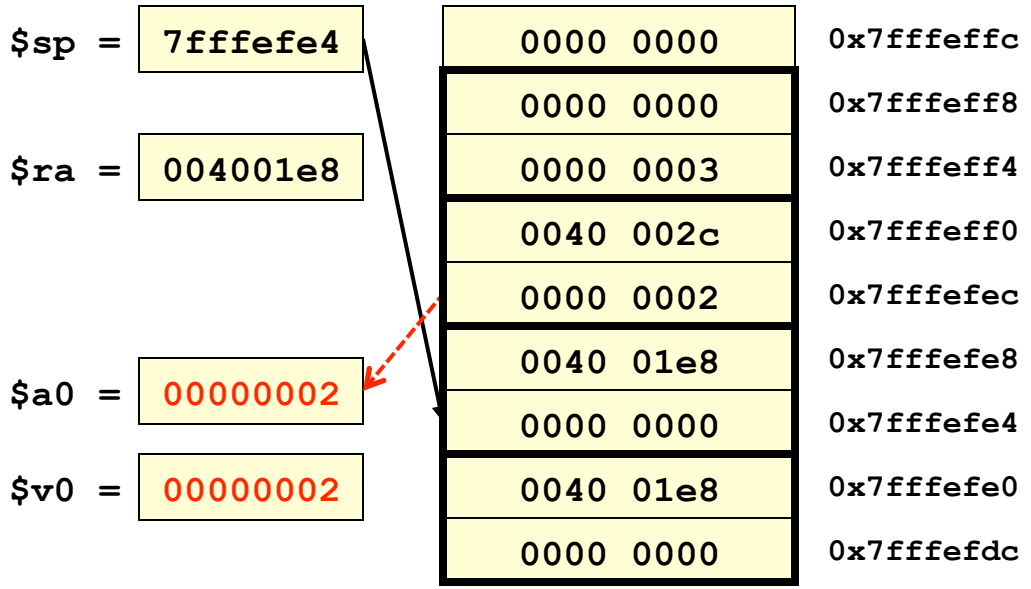
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Pop the $ra off the stack and return to that address

Recursive Routine Example

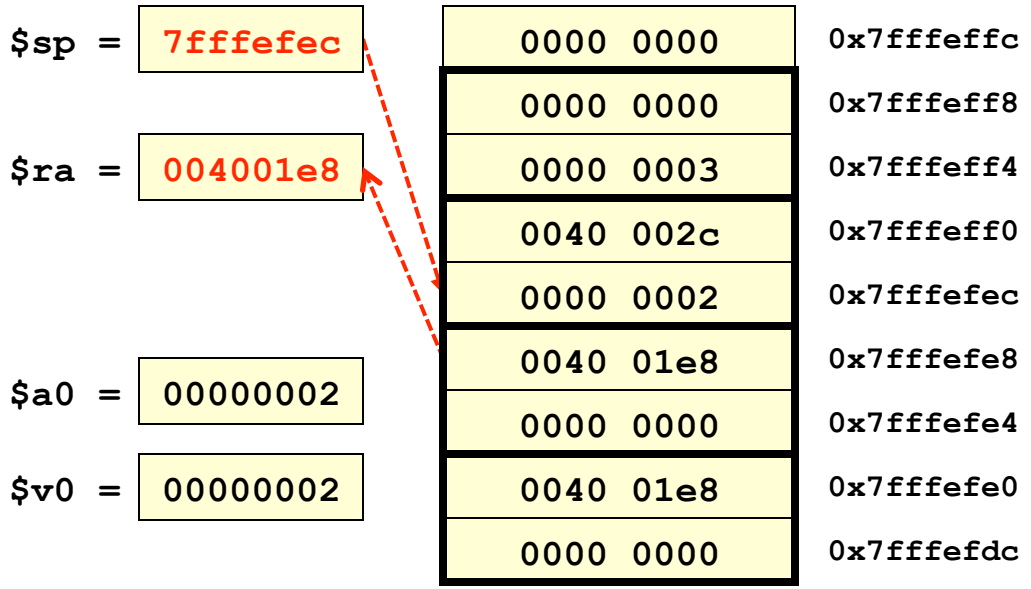
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Restore $a0 = 2
- Multiply $v0 = 1 * $a0 = 2 and store in $v0

Recursive Routine Example

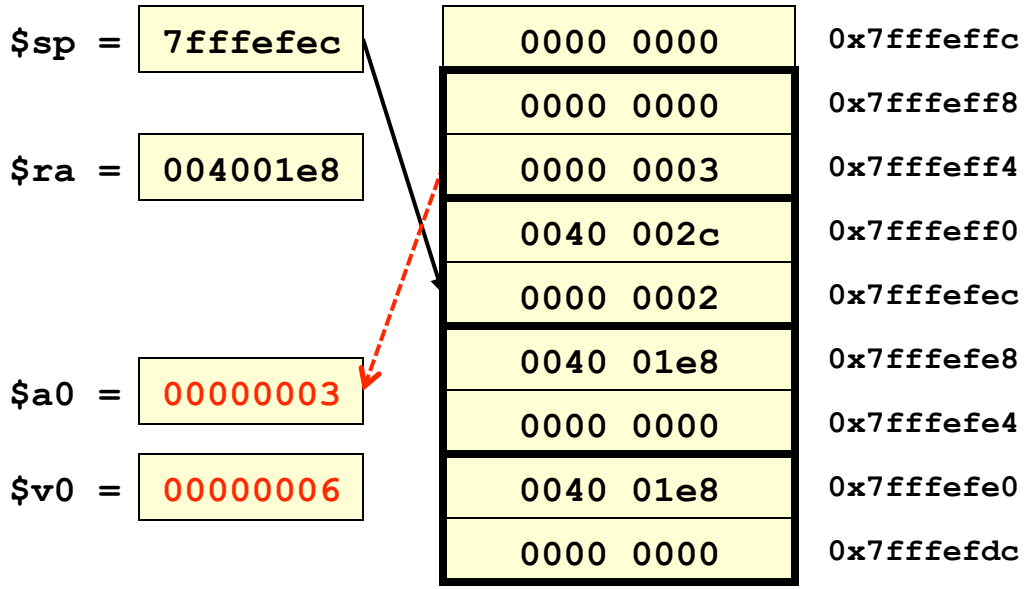
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Pop the $ra off the stack and return to that address

Recursive Routine Example

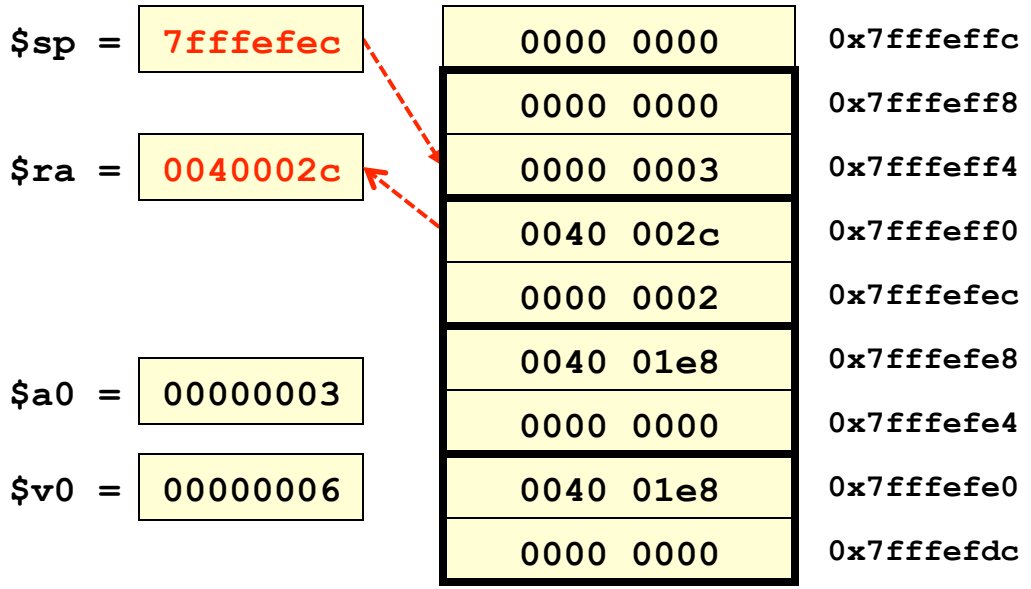
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Restore $a0 = 3
- Multiply $v0 = 2 * $a0 = 3 and store in $v0

Recursive Routine Example

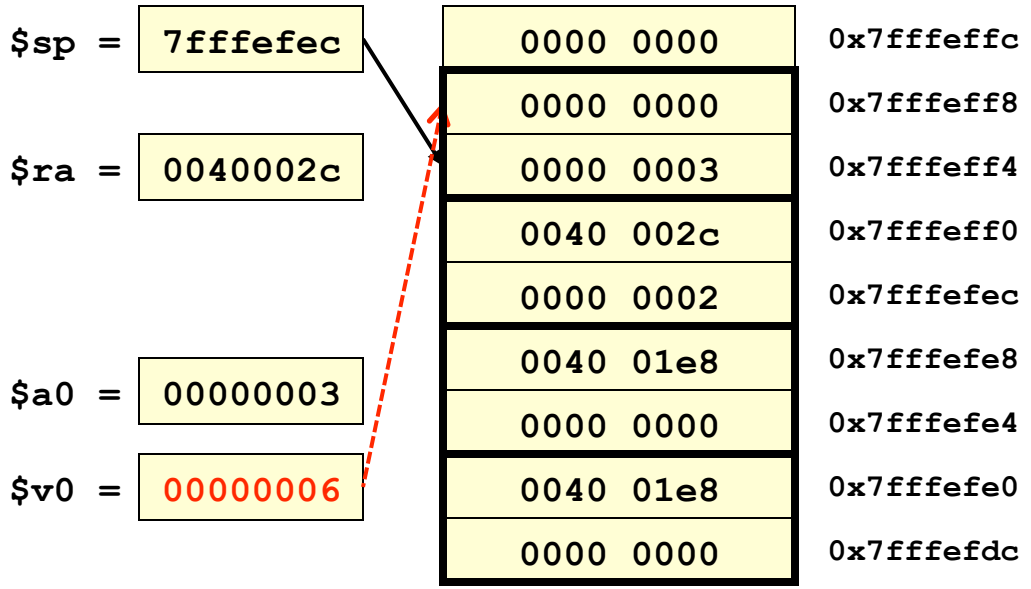
```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- Pop the $ra off the stack and return to that address

Recursive Routine Example

```
            .text
MAIN:       addi    $sp,$sp,-8
            li      $a0, 3
0x40002c    jal     FACT
            sw      $v0,4($sp)

FACT:       addi    $sp,$sp,-8
            sw      $ra,4($sp)
            addi    $v0,$zero,1
            beq     $a0,$v0,L1
            sw      $a0,8($sp)
            addi    $a0,$a0,-1
            jal     FACT
0x4001e8    lw      $a0,8($sp)
            mul     $v0,$a0,$v0
L1:         lw      $ra,4($sp)
            addi    $sp,$sp,8
            jr      $ra
```



- $v0 contains 3! = 6 and is stored in the local variable