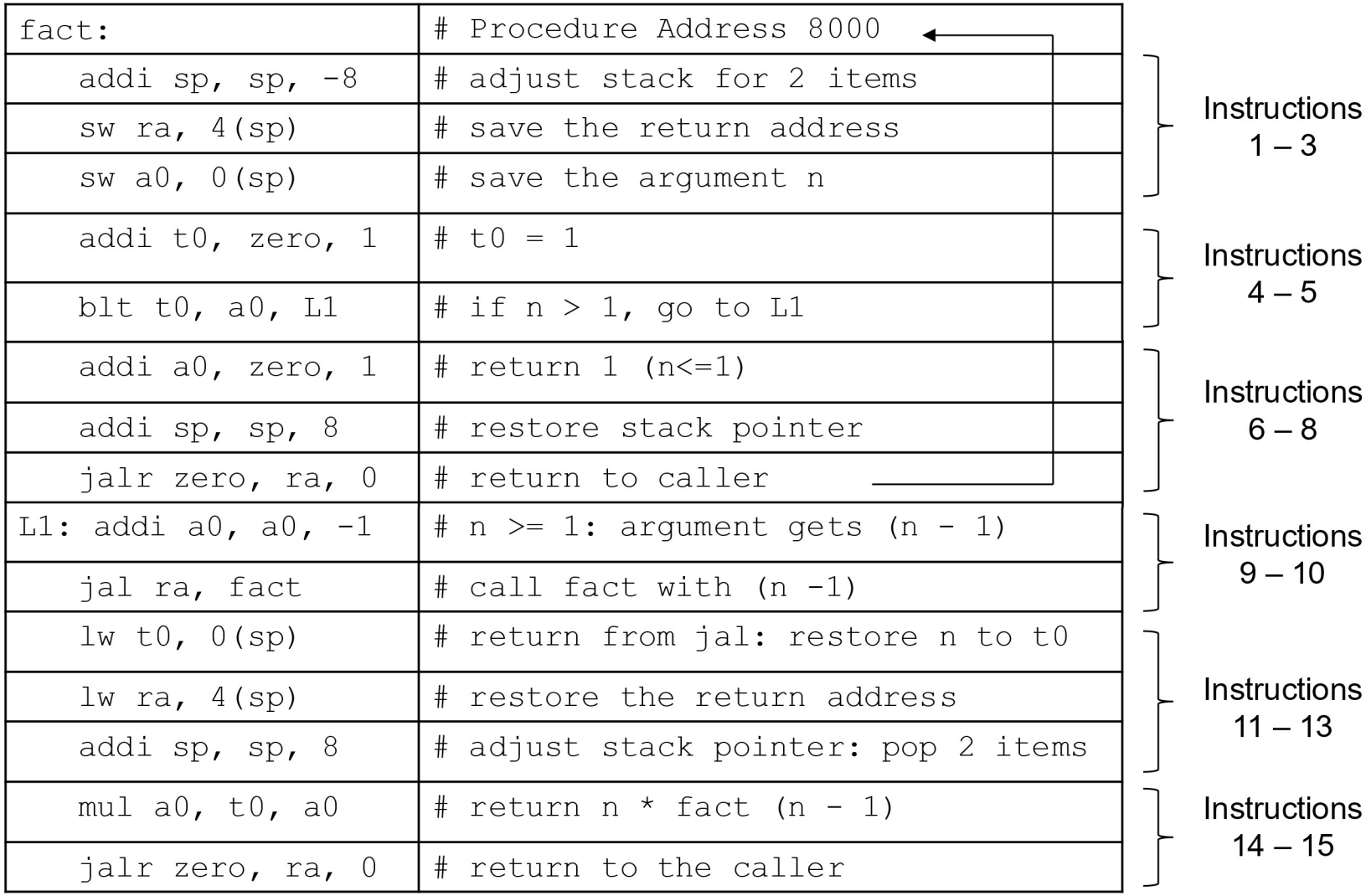
Recursion: Caller IS Callee

- **Each call creates a new stack frame**
- **special case of nested procedure call**

Classic Factorial Problem in RISC-V

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

- Argument n in a0
- Result in a0

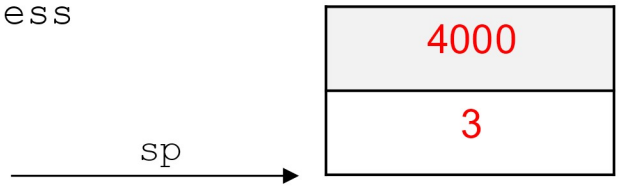

fact: # Procedure Address 8000
addi sp, sp, -8 # adjust stack for 2 items
sw ra, 4(sp) # save the return address
sw a0, 0(sp) # save the argument n
addi t0, zero, 1 # t0 = 1
blt t0, a0, L1 # if n > 1, go to L1
addi a0, zero, 1 # return 1 (n<=1)
addi sp, sp, 8 # restore stack pointer
jalr zero, ra, 0 # return to caller
L1: addi a0, a0, -1 # n >= 1: argument gets (n - 1)
jal ra, fact # call fact with (n -1)
lw t0, 0(sp) # return from jal: restore n to t0
lw ra, 4(sp) # restore the return address
addi sp, sp, 8 # adjust stack pointer: pop 2 items
mul a0, t0, a0 # return n * fact (n - 1)
jalr zero, ra, 0 # return to the caller
Instructions 1 – 3
Instructions 4 – 5
Instructions 6 – 8
Instructions 9 – 10
Instructions 11 – 13
Instructions 14 – 15

Main() calls fact(): First Call (n = 3)

- **Assume `jal ra, fact` is located at location 3996 in `main()`**
- **Argument n = 3 in `a0`**
- **Return address for `main()` = _____ so `ra` = ____**

- Instructions 1 – 3.
- Fill out the stack after the execution of the following instructions
- `sp` already adjusted in picture

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```



Instructions 4 – 5

- **Test for** $a0 > 1$
- **a0 = _____**

  will blt jump or continue?

```
addi t0, zero, 1                # test for n > 1
blt  t0, a0, L1
```

Instructions 9 – 10

- **Jump to L1**
- **After executing instructions 9 and 10**

```
L1: addi a0, a0, -1      # n > 1: argument gets (n - 1)
    jal  ra, fact                # call fact with (n -1)
```

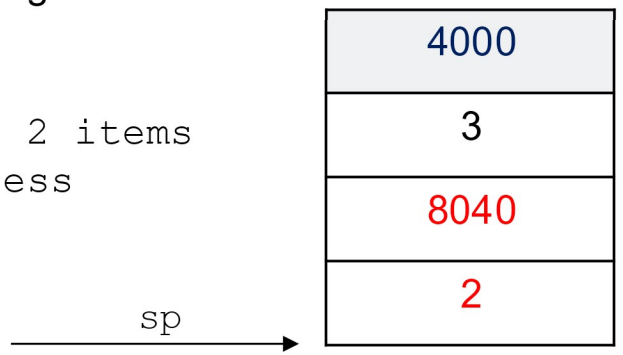
- a0 updated to 2
- address of fact = 8000
- `ra` = 8040

`fact()` calls `fact()`: Second Call (n = 2)

- **Argument n = 2 in a0**
- **Return address for `fact()` = _____ , so `ra` = _____**

  - Instructions 1 – 3.
  - Fill out the stack after the execution of the following instructions
  - `sp` already adjusted

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```



Instructions 4 – 5

- **Test for** $a0 > 1$
- **a0 = ____**

  will blt jump or continue?

```
addi t0, zero, 1                # test for n > 1
blt  t0, a0, L1
```

Instructions 9 – 10

- **Jump to L1**
- **After executing instructions 9 and 10**

```
L1: addi a0,a0,-1       # n > 1: argument gets (n - 1)
    jal ra, fact        # call fact with (n -1)
```

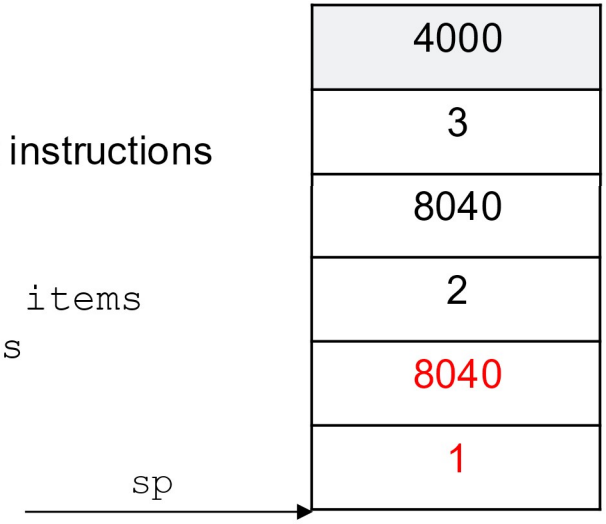
  - a0 updated to 1
  - address of fact = 8000
  - `ra` = 8040

`fact()` calls `fact()`: Third Call (n = 1)

- **Argument n = 1 in a0**
- **Return address for `fact()` = _____ so (`ra` = _____)**

- Instructions 1 – 3.
- Fill out the stack after the execution of the following instructions
- `sp` already adjusted

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```



Instructions 4 – 5

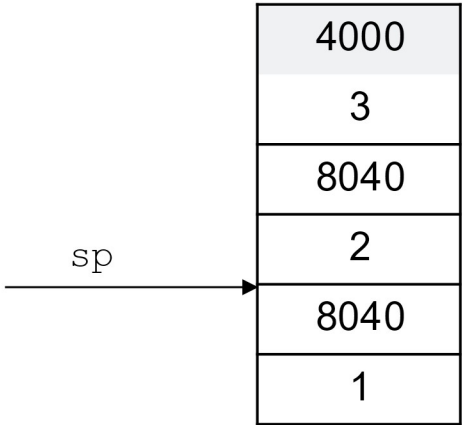
- **Test for** $a0 > 1$
- **a0 = ____**

  will blt jump or continue?

```
addi t0, zero, 1                # test for n > 1
blt  t0, a0, L1
```

Instructions 6 – 8

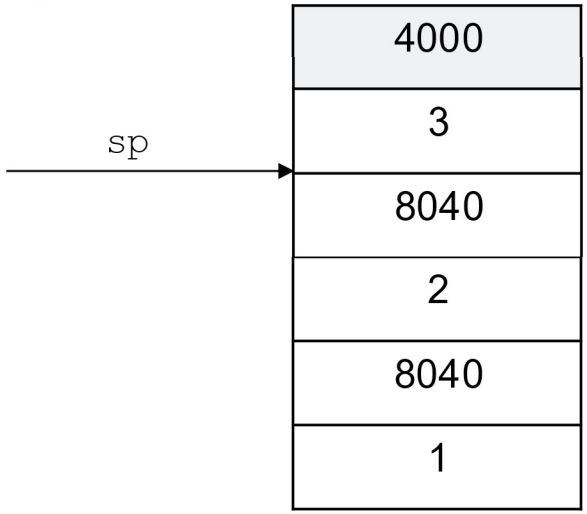
```
addi a0, zero, 1 # return 1
addi sp, sp, 8   # restore stack pointer
jalr zero, ra, 0 # return to caller
```



- **Return to** 8040 **with a0 =** 1
- **New PC =** 8040
- **How many times are these 3 instructions executed over the duration of the program (until the completion of the calculation of the factorial)?**
- **We adjusted `sp` but never popped anything off. Why?**

Instructions 11 – 13

```
lw t0, 0(sp)      # return from jal: recover argument n
lw ra, 4(sp)      # restore the return address
addi sp, sp, 8    # adjust stack pointer to pop 2 items
```



- **a0 =1 (from returned call)**
- **t0 = 2**
- **ra = 8040**
- **What is left in the stack after the execution of the instructions above?**

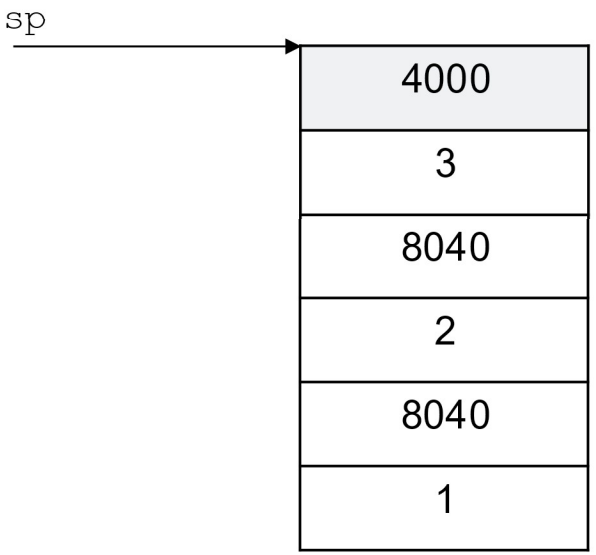
Instructions 14 - 15

```
mul a0, t0, a0              # return n * fact (n - 1)
jalr zero, ra, 0
```

- **Return to __fact()_____ with a0 = __2____**
- **New PC = _8040_____**

Instructions 11 – 13

```
lw t0, 0(sp)      # return from jal: restore argument n
lw ra, 4(sp)      # restore the return address
addi sp, sp, 8    # adjust stack pointer to pop 2 items
```



- **a0 = 2 (from returned call)**
- **t0 = 3**
- **ra = 4000**
- **What is left in the stack after the execution of the instructions above?**

58

Instructions 14 - 15

```
mul a0, t0, a0    # return n * fact (n - 1)
jalr zero, ra, 0 # return to caller
```

- **Return to _main()__ with a0 = __6___**
- **New PC = _4000_**

Recursive Procedure Call Example: Fibonacci Numbers

- **The Fibonacci numbers are defined as follows:**

  F(n) = F(n – 1) + F(n – 2),
  F(0) and F(1) are defined to be 1

- **Rewriting this in C :**

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Approaching the RISC-V Code

- **Decide what goes on the stack, write the prologue**
  - Allocating stack
  - Saving initial values
- **Write the epilogue**
  - Restore values from the stack
  - Release the stack
  - Return to the caller
- **Write the body– carefully**
  - This matches the C function body

Fibonacci Numbers (Prologue)

- **Space for three words on the stack**
  - return addr (ra)
  - n (a0)
  - temp sum (s0)
- **RISC-V code:**

```
fib:
    addi sp, sp, -12 # space for 3 words
    sw ra, 8(sp)     # save return address
    sw s0, 4(sp)     # save s0, used within
```

Fibonacci Numbers (Epilogue)

```
fib_done:
    lw   s0, 4(sp)    # restore s0
    lw   ra, 8(sp)    # restore ra

    addi sp, sp, 12   # pop the stack frame
    jalr zero, ra, 0 # return to caller
```

Fibonacci Numbers: Body Part 1

- **Start with the base cases**

```
if (n == 0) return 1;
if (n == 1) return 1;
```

- **RISC-V code:**

```
addi t1, a0, 0          # save a0 in t1
addi a0, x0, 1          # a0 = 1 (setup return value)
beq  t1, x0, fib_done # if n==0 goto fib_done
addi t0, x0, 1
beq  t1, t0, fib_done # if n==1 goto fib_done
```

Fibonacci Numbers (Body Part 2)

- **Next, the recursive case: careful, this is tricky**

```
return fib(n-1) + fib(n-2)
```

- **RISC-V code:**

```
addi a0, t1, -1     # a0 = n-1
sw   a0, 0(sp)      # need a0 after jal
jal  ra, fib        # fib(n-1)
add  s0, a0, 0      # save result
lw   a0, 0(sp)      # restore a0
addi a0, a0, -1     # a0 = n-2 (already n-1)
jal  ra, fib        # fib(n-2)
add  a0, a0, s0     # a0=fib(n-2)+fib(n-1)
                    # continues at fib_done...
```

fib.s

- **Breakpoint on jal's and jalr**
- **Watch a0, s0, sp, ra**
- **Single step the first jal to see ra change (on first call)**
  - Updated, but stays unchanged, on the recursive call
- **Single step the** jalr x0, ra, 0 **to see PC change (on last return)**
  - Updated, but stays unchanged, on the recursive returns

RISC-V Register Conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

After-Class Activities

- **Draw out the stack showing the execution of fib.s with an input of 4 (the first 4 Fibonacci numbers will be printed)**
- **Compare with what RARS stack shows**