UNIVERSITY of WASHINGTON

Agenda

- Function calls and Jumps
- Call Stack
- Register Convention
- Program memory layout

Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

- fixed length, variable length, recursively
- Get return value back to the caller

Manage Registers

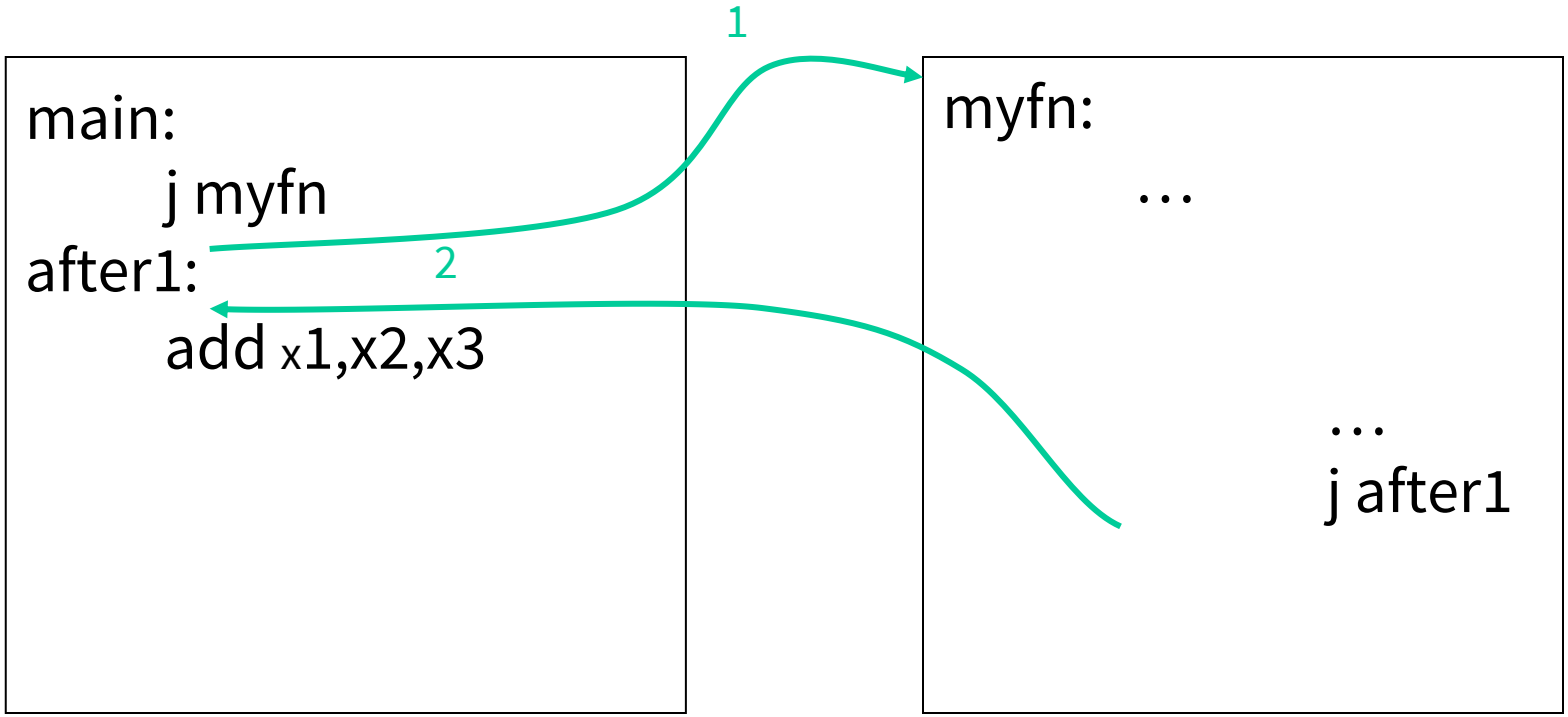
- Allow each routine to use registers
- Prevent routines from clobbering each others' data

UNIVERSITY of WASHINGTON

Six Steps of Calling a Function

1. Put *arguments* in a place where the function can access them
2. Transfer control to the function
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and "cleans up"
6. Control is returned to you

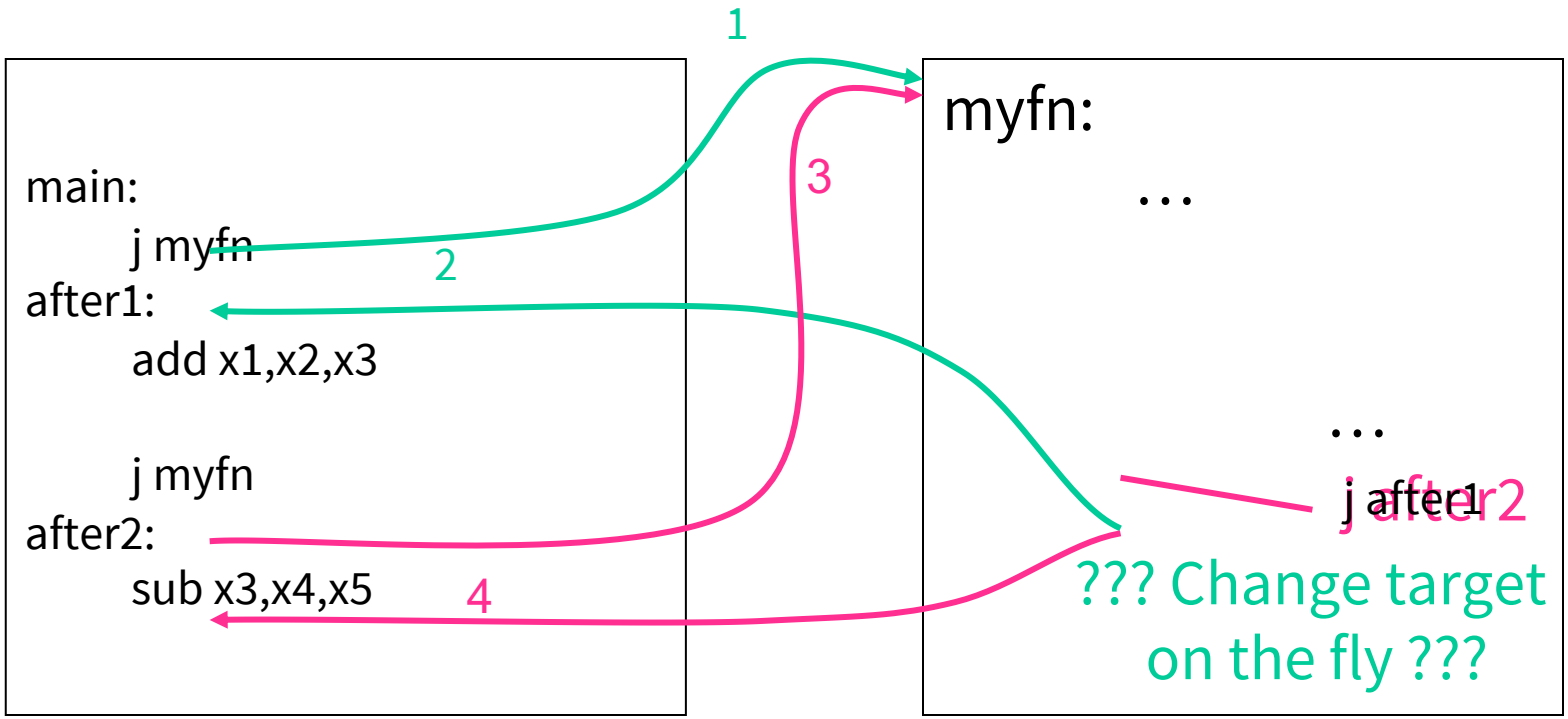
Jumps are not enough



Jumps to the callee

Jumps back

Jumps are not enough



Jumps to the callee

Jumps back

What about multiple sites?

Takeaway 1: Need Jump And Link

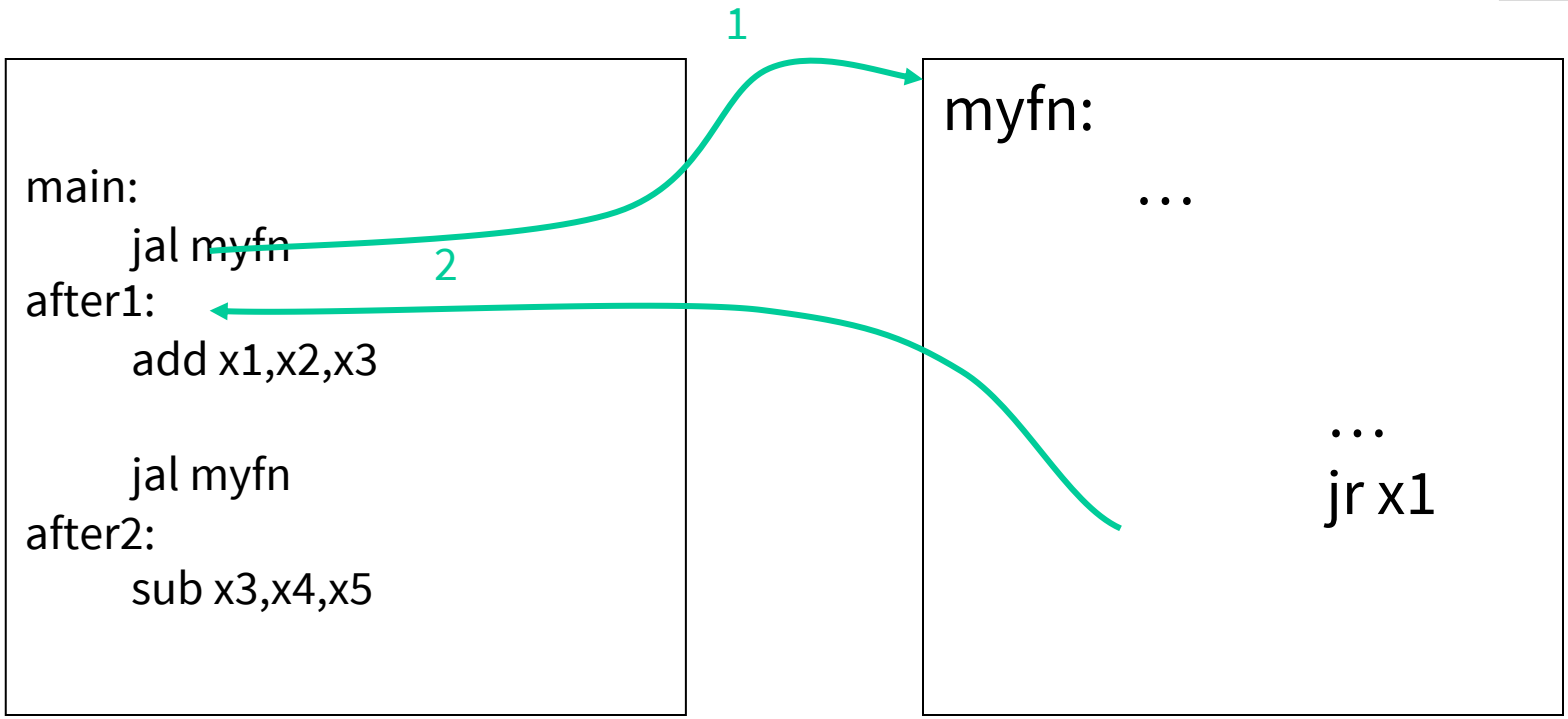
JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register x1 (aka $ra or return address)

Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register x1

Jump-and-Link / Jump Register

First call

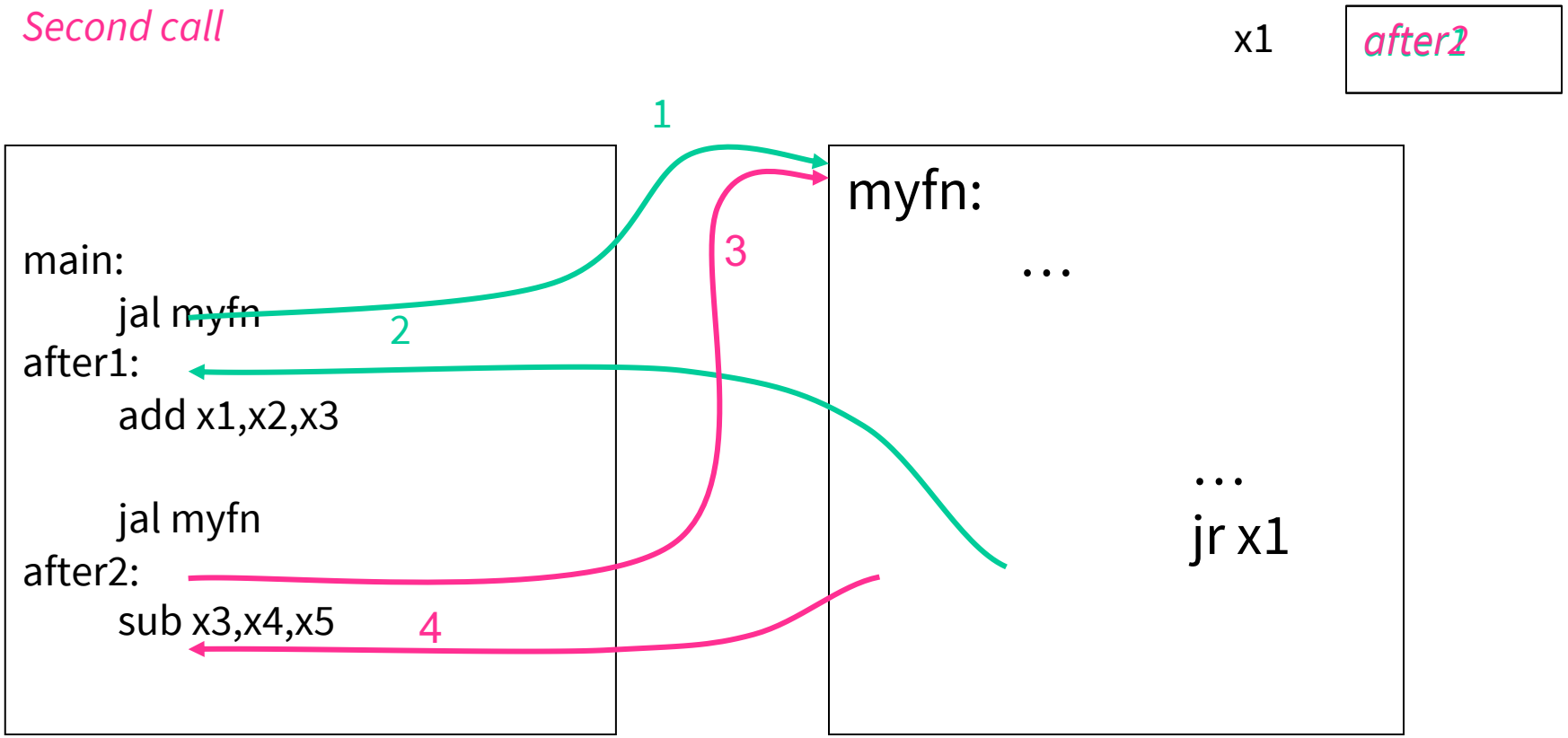
x1 | *after1*



JAL saves the PC in register $31

Subroutine returns by jumping to $31

Jump-and-Link / Jump Register



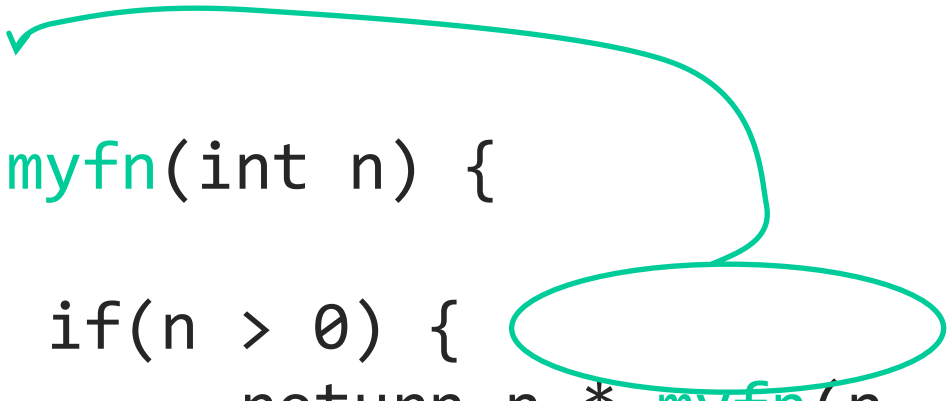
JAL saves the PC in register x1

Subroutine returns by jumping to x1

What happens for recursive invocations?

JAL / JR for Recursion?

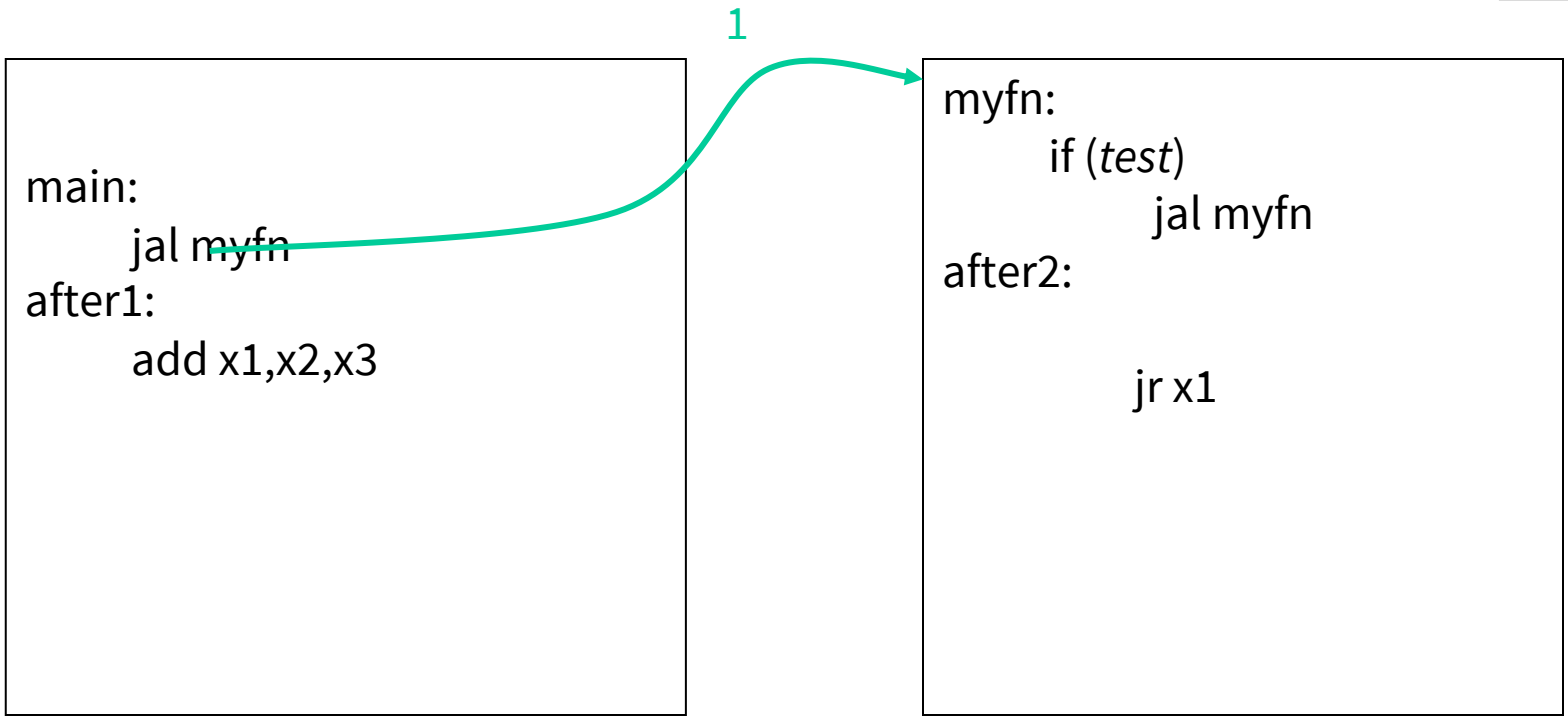
```
int main (int argc, char* argv[ ]) {
     int n = 9;
     int result = myfn(n);
}

int myfn(int n) {

     if(n > 0) {
          return n * myfn(n - 1);
     } else {
          return 1;
     }
}
```



JAL / JR for Recursion?

First call

x1 | *after1*



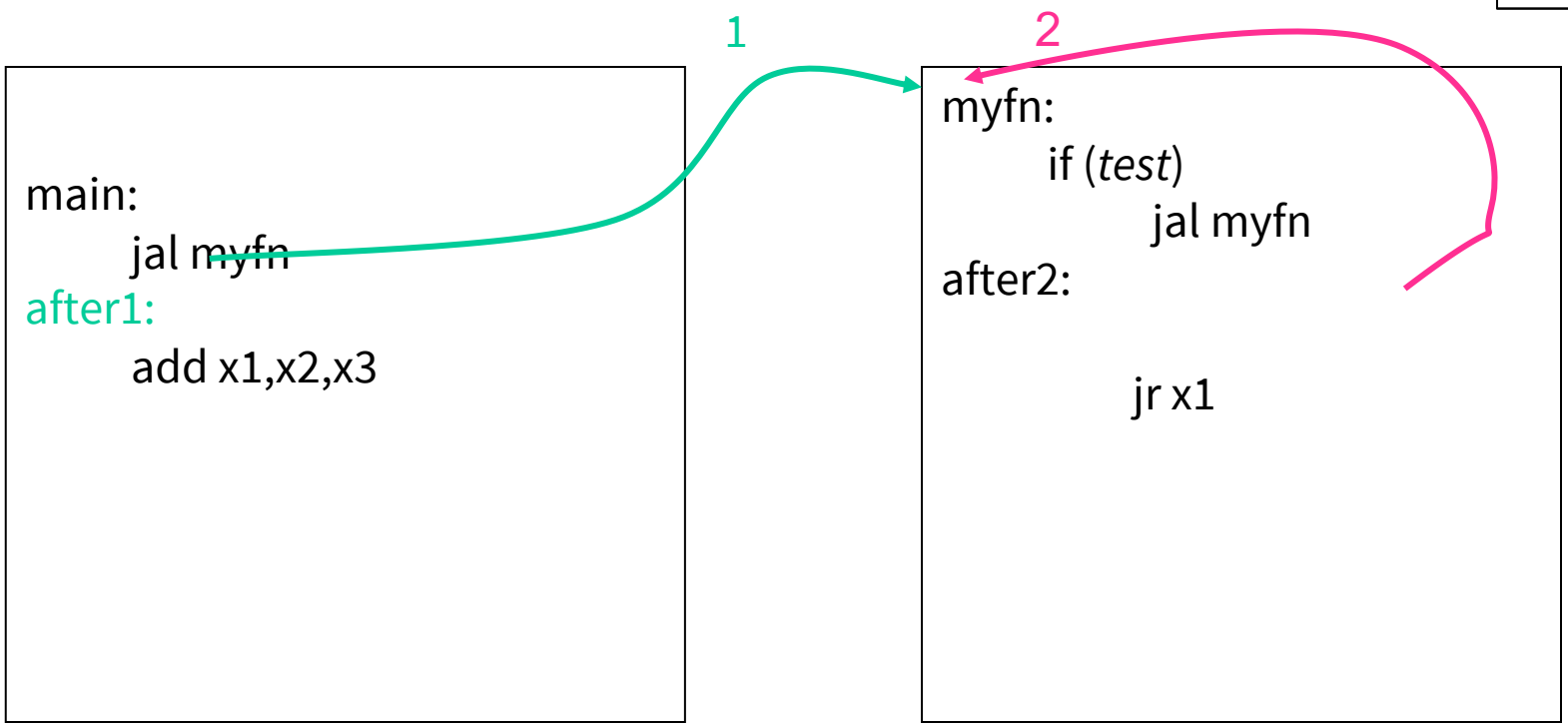
Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Recursive Call

x1 | *after2*



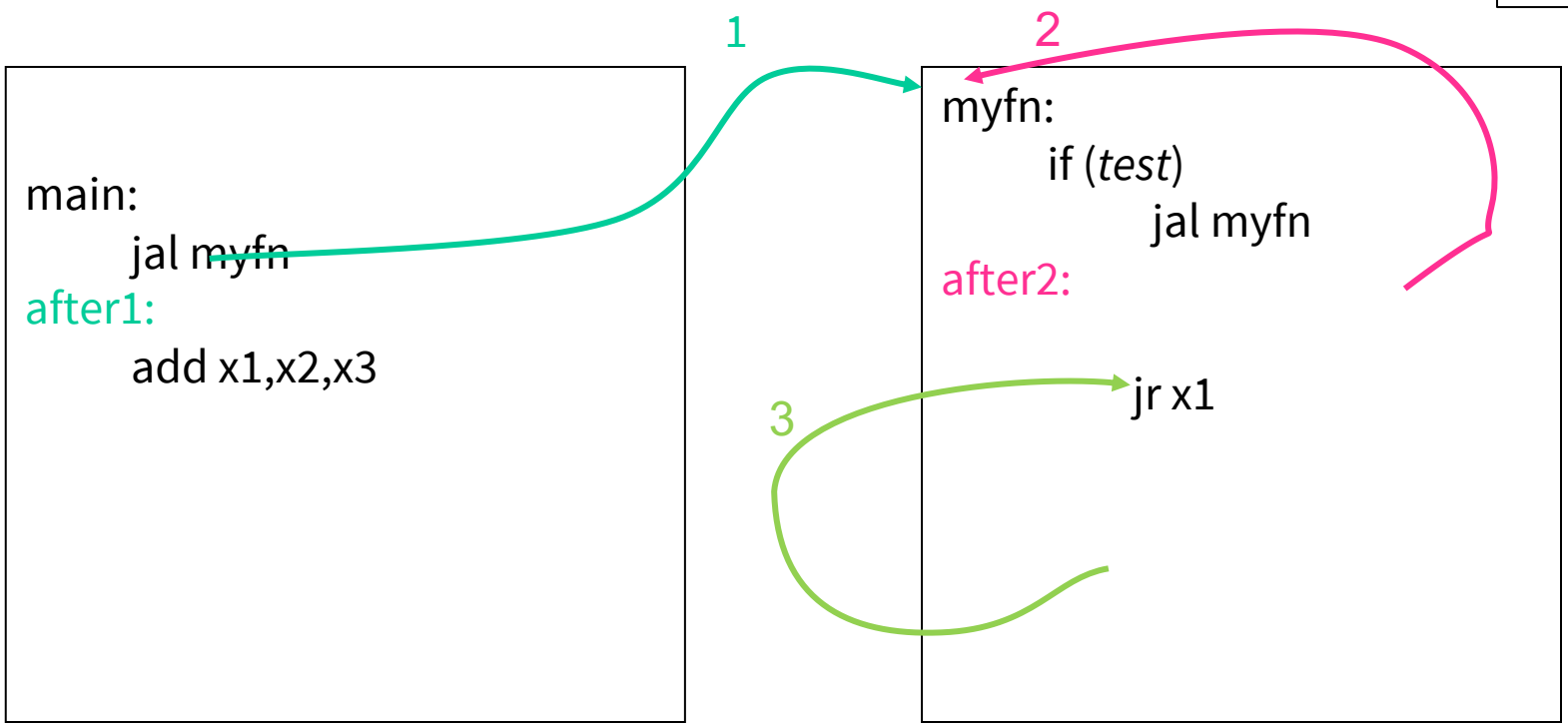
Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Recursive Call

x1 | *after2*



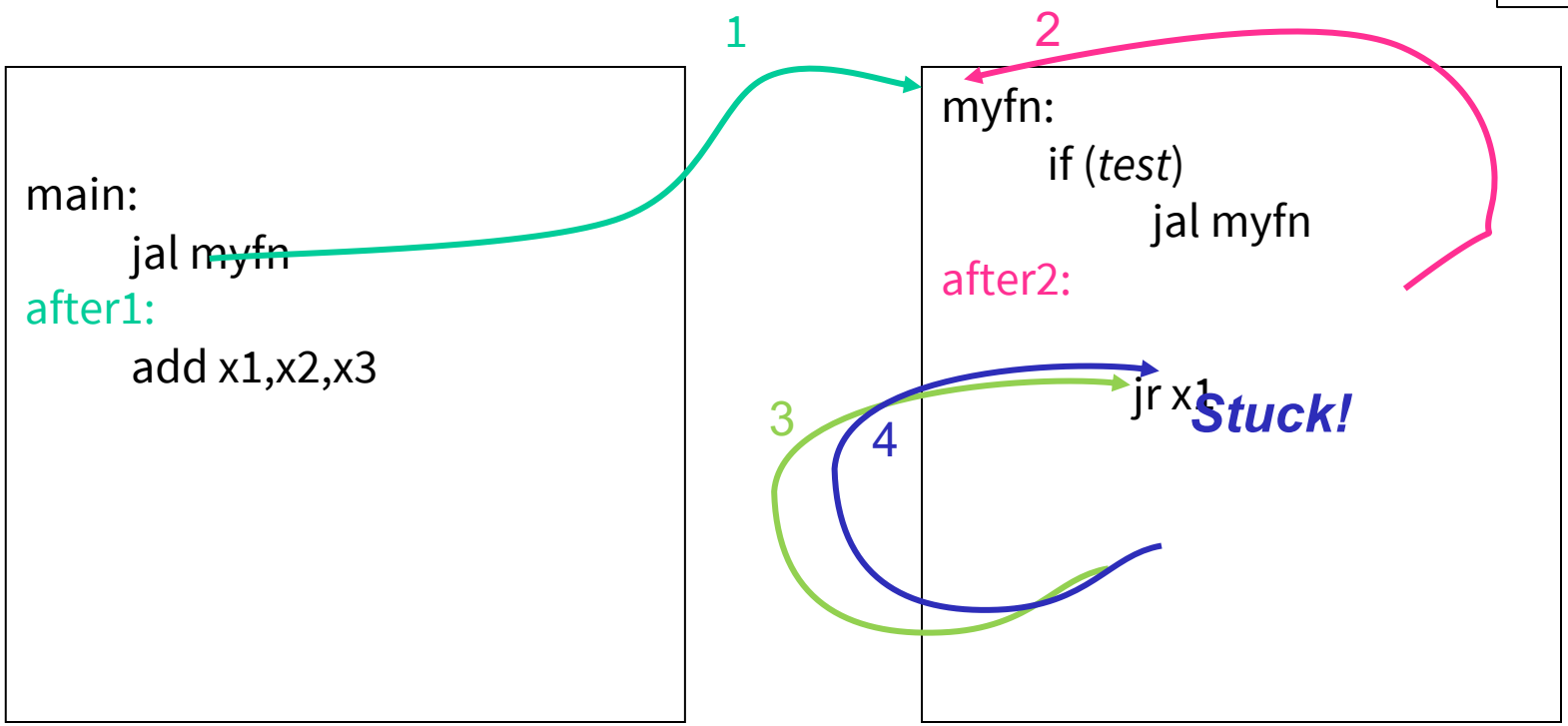
Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???

x1 | *after2*



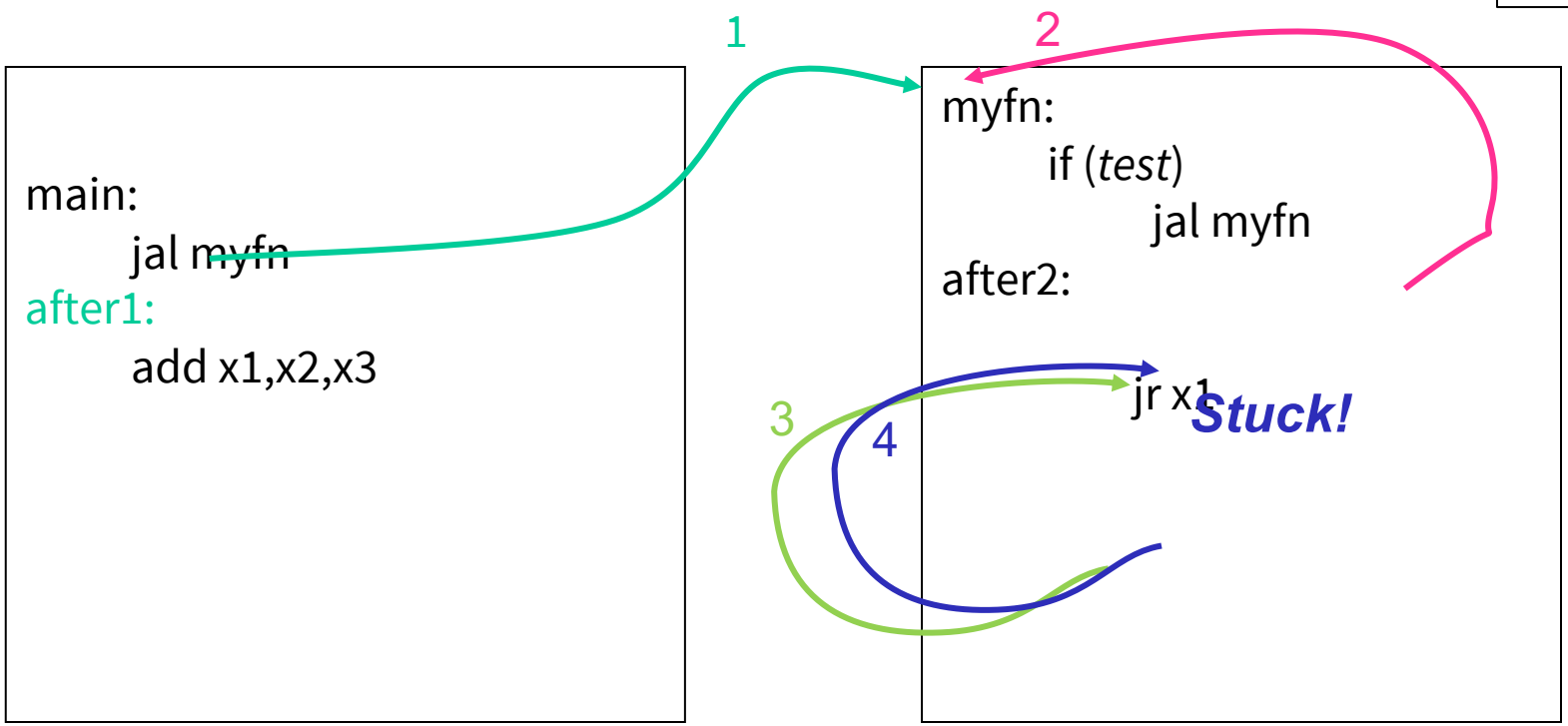
Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???

x1 | *after2*



Problems with recursion:

- overwrites contents of x1
- Need a way to save and restore register contents

UNIVERSITY of WASHINGTON

Agenda

- Function calls and Jumps
- Call Stack
- Register Convention
- Program memory layout

UNIVERSITY of WASHINGTON

Takeaway2: Need a Call Stack

JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register x1 (aka ra or return address) Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register x1

Need a Call Stack to return to correct calling procedure. To maintain a stack, need to store an ***activation record*** (aka a "stack frame") in memory. Stacks keep track of the correct return address by storing the contents of x1 in memory (the stack).

Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

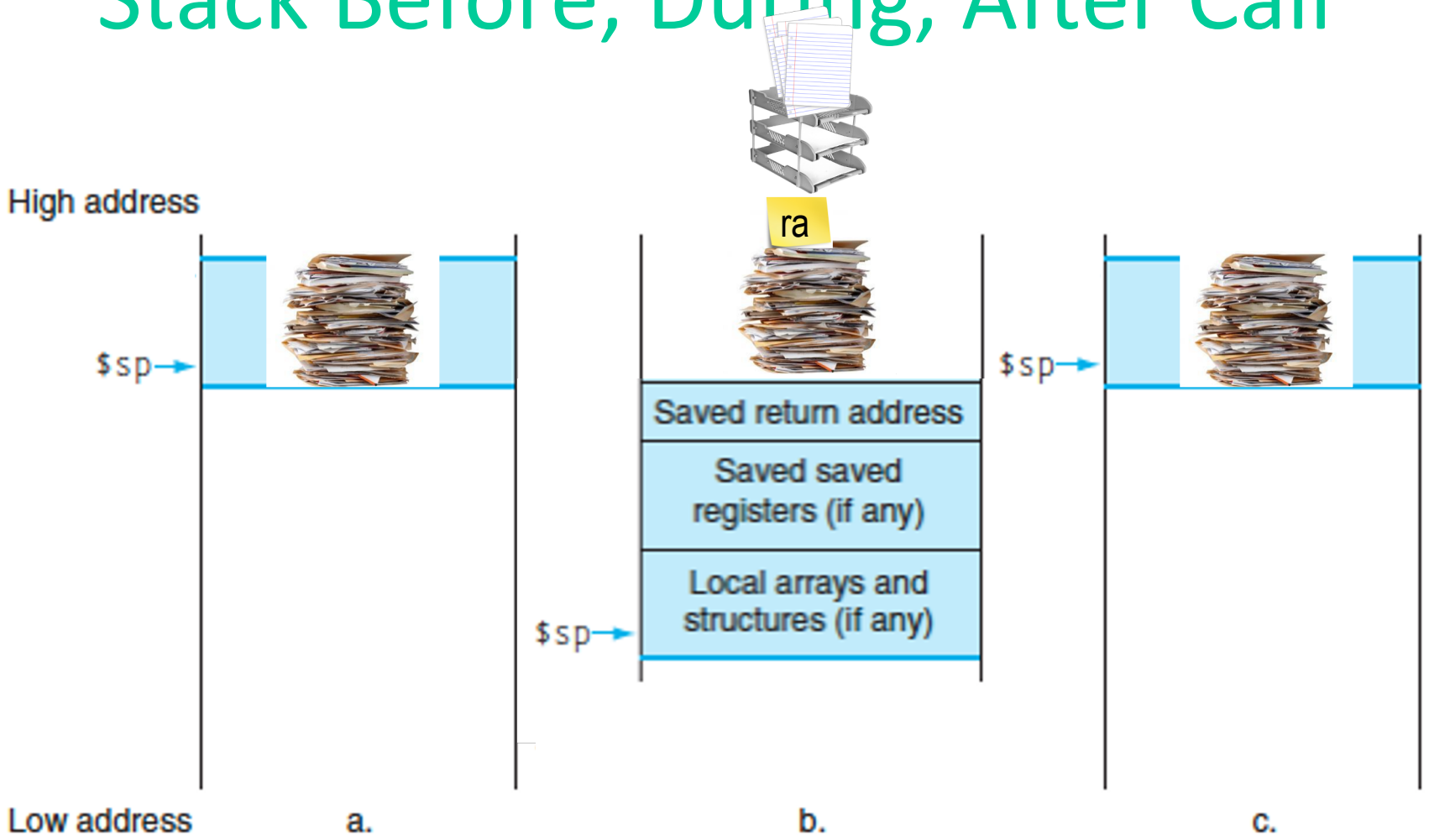
- dedicated register (x2) on the RISC-V

Manipulated by push/pop operations

- push: move sp down, store
- pop: load, move sp up

UNIVERSITY of WASHINGTON

Stack Before, During, After Call



UNIVERSITY of WASHINGTON

Local Variables and Arrays

- Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame (**Recall:** spilling to memory)

- Locally declared arrays and structs are also allocated as part of the stack frame

- Stack manipulation is same as before
  - Move `sp` down an extra amount and use the space it created as storage

UNIVERSITY of WASHINGTON

Function Call Example

```
int Leaf(int g, int h, int i, int j) {
      int f;
      f = (g + h) - (i + j);
      return f;
}
```

- Parameter variables **`g`**, **`h`**, **`i`**, and **`j`** in argument registers **`a0`**, **`a1`**, **`a2`**, and **`a3`**, and **`f`** in **`s0`**
- Assume need one temporary register `s1`

UNIVERSITY of WASHINGTON

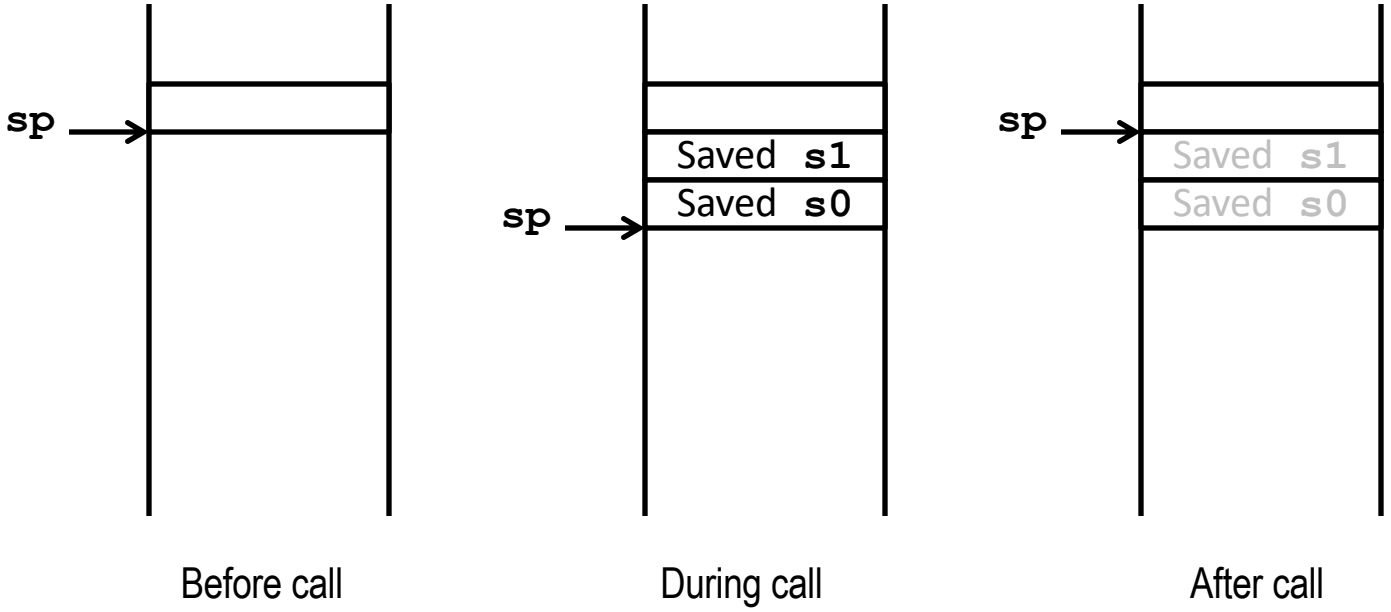
RISC-V Code for Leaf()

```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      add s0,a0,a1      # f = g + h
      add s1,a2,a3      # s1 = i + j
      sub a0,s0,s1      # return value (g + h) – (i + j)
      lw s0, 0(sp)      # restore register s0 for caller
      lw s1, 4(sp)      # restore register s1 for caller
      addi sp,sp,8      # adjust stack to delete 2 items
      jr ra             # jump back to calling routine
```

UNIVERSITY of WASHINGTON

Stack Before, During, After Function

- Need to save old values of **s0** and s1



UNIVERSITY of WASHINGTON

Nested Procedures

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`
- So there's a value in `ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`

Need to save `sumSquare` return address before call to `mult` – again, use stack

UNIVERSITY of WASHINGTON

Agenda

- Function calls and Jumps
- Call Stack
- Register Convention
- Program memory layout

UNIVERSITY of WASHINGTON

Register Conventions

- CalleR: the calling function
- CalleE: the function being called

- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.

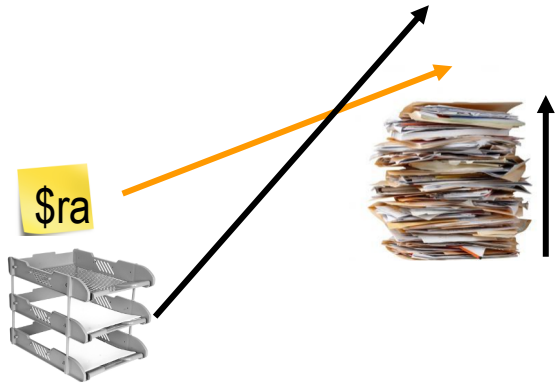
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**`jal`**) and which may be changed.

UNIVERSITY of WASHINGTON

Basic Structure of a Function

Prologue

```
func_label:
addi sp,sp, -framesize
sw ra, <framesize-4>(sp)
save other regs if need be
```

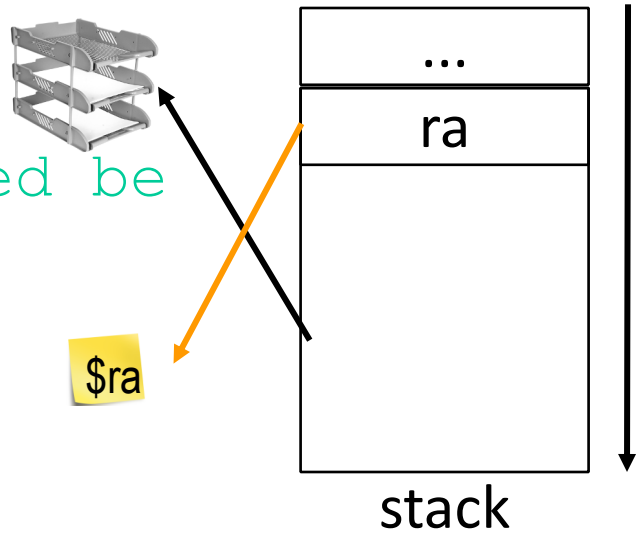


Body **(call other functions…)**

```
...
```

Epilogue

```
restore other regs if need be
lw ra, <framesize-4>(sp)
addi sp,sp, framesize
jr ra
```



UNIVERSITY of WASHINGTON

Using Stack to Backup Registers

- Limited number of registers for everyone to use (limited desk space)
- All functions use the same conventions -- look for arguments/return addresses in the same places
  - What happens if a function calls another function?

    (`ra` would get overwritten!)

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

UNIVERSITY of WASHINGTON

x0	zero	zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global data pointer
x4	tp	thread pointer
x5	t0	**temps (caller save)**
x6	t1	
x7	t2	
x8	s0/fp	frame pointer
x9	s1	**saved (callee save)**
x10	a0	function args or return values
x11	a1	
x12	a2	function arguments
x13	a3	
x14	a4	
x15	a5	function arguments
x16	a6	
x17	a7	
x18	s2	**saved (callee save)**
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s7	
x25	s9	
x26	s10	
x27	s11	
x28	t3	**temps (caller save)**
x29	t4	
x30	t5	
x31	t6	

UNIVERSITY of WASHINGTON

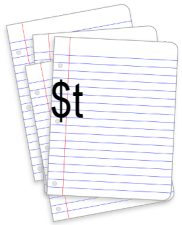
Saved Registers

- These registers are expected to be the same before and after a function call
  - If calleE uses them, it must restore values before returning
  - This means save the old values, use the registers, then reload the old values back into the registers
- `s0-s11` (*saved* registers)
- `sp` (stack pointer)
  - If not in same place, the caller won't be able to properly restore values from the stack
- `ra` (return address)

UNIVERSITY of WASHINGTON

Volatile Registers

- These registers can be freely changed by the calleE
  - If calleR needs them, it must save those values before making a procedure call
- `t0-t6` (*temporary* registers)
- `a0-a7` (return address and arguments)
  - These will change if calleE invokes another function (nested function means calleE is also a calleR)



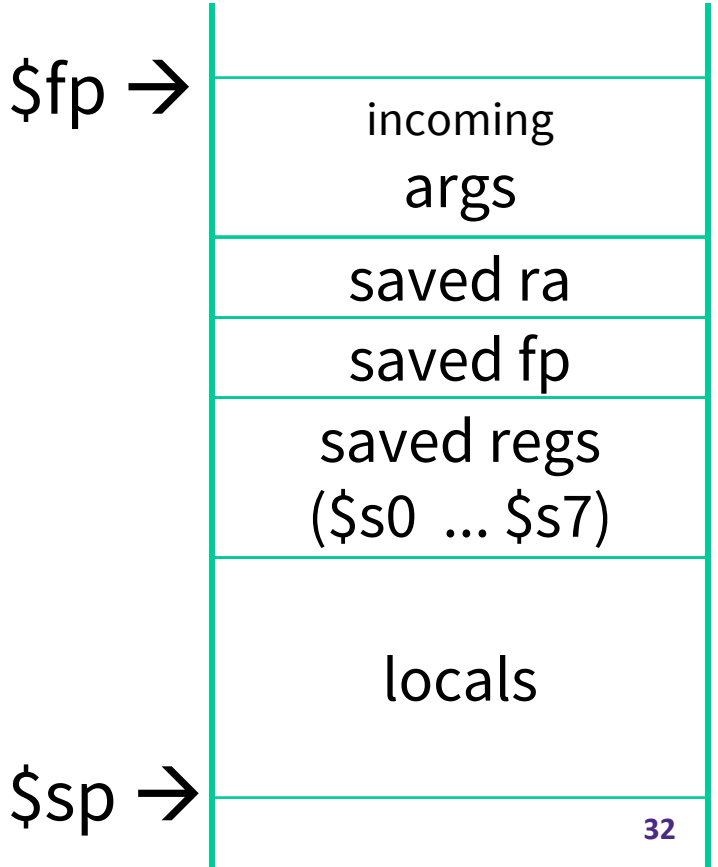
UNIVERSITY of WASHINGTON

RISCV Register Summary

Return address: x1 (ra)
Stack pointer: x2 (sp)
Frame pointer: x8 (fp/s0)
First four arguments: x10-x17 (a0-a7)
Return result: x10-x11 (a0-a1)
Callee-save free regs: x9,x18-x27 (s1-s11)
Caller-save (temp) free regs: x5-x7, x28-x31 (t0-t6)
Global pointer: x3 (gp)

Convention Summary

- first eight arg words passed in $a0-$a7
- Space for args in child's stack frame
- return value (if any) in $a0, $a1
- stack frame ($fp to $sp) contains:
  - $ra (clobbered on JALs)
  - local variables
  - space for 8 arguments to Callees
  - arguments 9+ to Callees
- callee save regs: preserved
- caller save regs: not preserved
- global data accessed via $gp



UNIVERSITY of WASHINGTON

Activity #1: Calling Convention Example

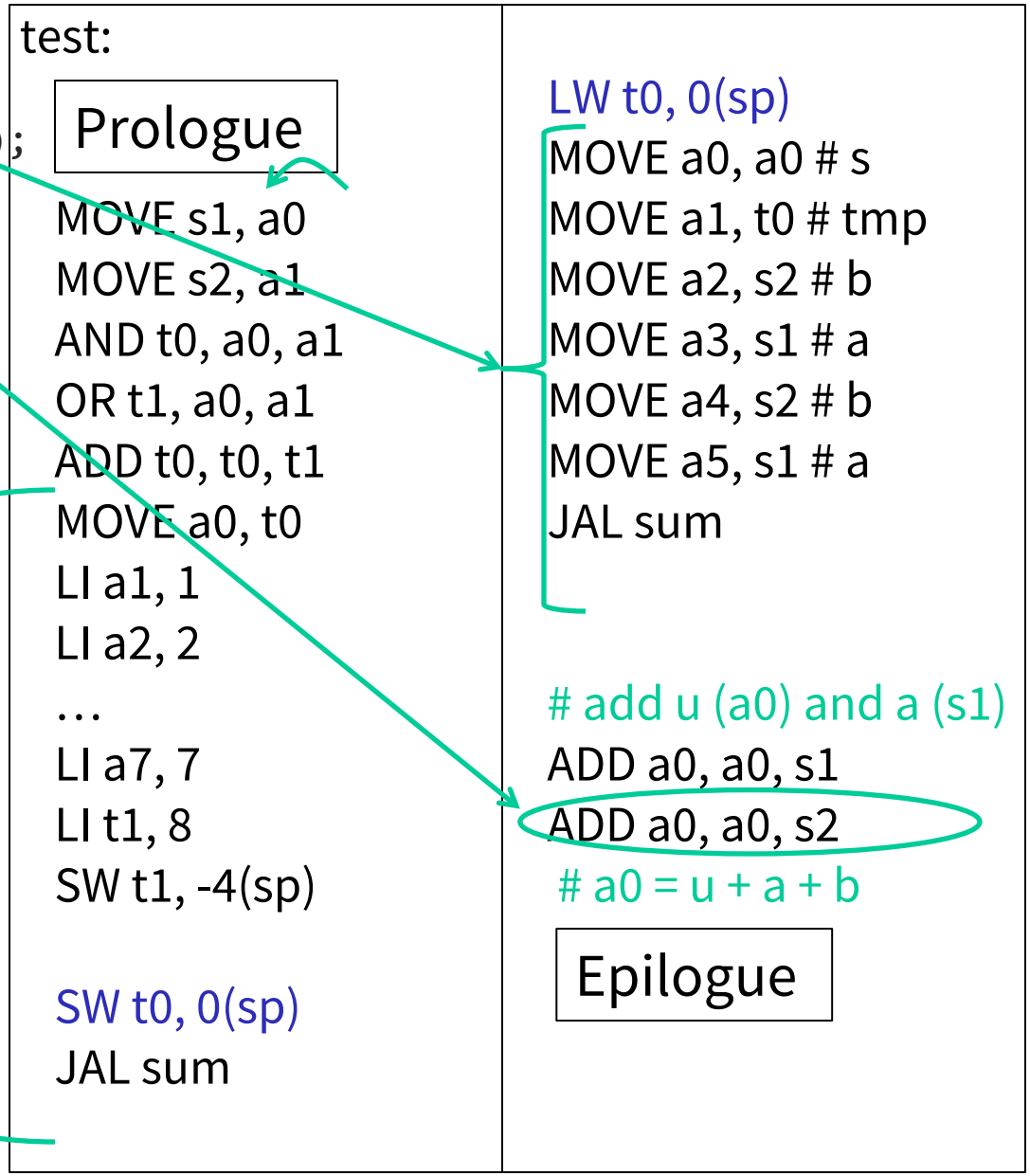
```
int test(int a, int b) {
 int tmp = (a&b)+(a|b);
 int s = sum(tmp,1,2,3,4,5,6,7,8);
 int u = sum(s,tmp,b,a,b,a);
 return u + a + b;
}
```

Correct Order:

1. Body First
2. Determine stack frame size
3. Complete Prologue/Epilogue

UNIVERSITY of WASHINGTON

```
int test(int a, int b) {
 int tmp = (a&b)+(a|b);
 int s =sum(tmp,1,2,3,4,5,6,7,8);
 int u = sum(s,tmp,b,a,b,a);
 return u + a + b;
}
```



UNIVERSITY of WASHINGTON

```
int test(int a, int b) {
 int tmp = (a&b)+(a|b);
 int s =sum(tmp,1,2,3,4,5,6,7,8);
 int u = sum(s,tmp,b,a,b,a);
 return u + a + b;
}
```

How many bytes do we need to allocate for the stack frame?

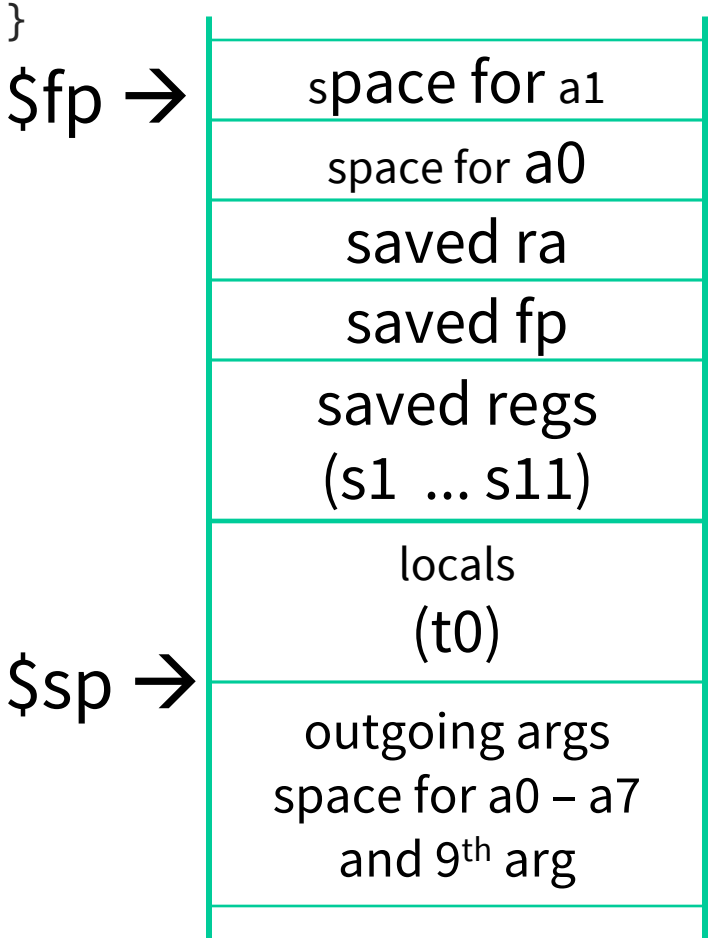
a) 24
b) 28
c) 36
d) 40
e) 48

```
test:
  Prologue
  MOVE s1, a0
  MOVE s2, a1
  AND t0, a0, a1
  OR t1, a0, a1
  ADD t0, t0, t1
  MOVE a0, t0
  LI a1, 1
  LI a2, 2
  …
  LI a7, 7
  LI t1, 8
  SW t1, -4(sp)

  SW t0, 0(sp)
  JAL sum
```

```
  LW t0, 0(sp)
  MOVE a0, a0 # s
  MOVE a1, t0 # tmp
  MOVE a2, s2 # b
  MOVE a3, s1 # a
  MOVE a4, s2 # b
  MOVE a5, s1 # a
  JAL sum


  # add u (v0) and a (s1)
  ADD a0, a0, s1
  ADD a0, a0, s2
   # a0 = u + a + b
  Epilogue
```

UNIVERSITY of WASHINGTON

```
int test(int a, int b) {
 int tmp = (a&b)+(a|b);
 int s =sum(tmp,1,2,3,4,5,6,7,8);
 int u = sum(s,tmp,b,a,b,a);
 return u + a + b;
}
```



test:

Prologue

MOVE s1, a0
MOVE s2, a1
AND t0, a0, a1
OR t1, a0, a1
ADD t0, t0, t1
MOVE a0, t0
LI a1, 1
LI a2, 2
…
LI a7, 7
LI t1, 8
SW t1, -4(sp)

SW t0, 0(sp)
JAL sum

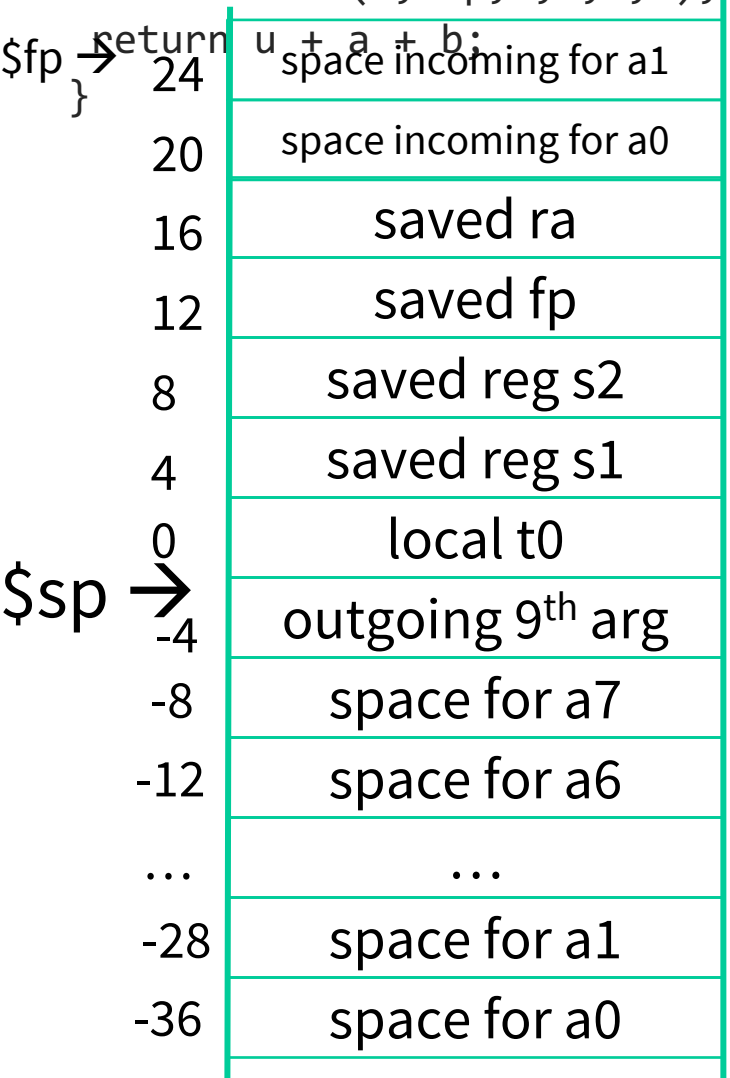
LW t0, 0(sp)
MOVE a0, v0 # s
MOVE a1, t0 # tmp
MOVE a2, s2 # b
MOVE a3, s1 # a
MOVE a4, s2 # b
MOVE a5, s1 # a
JAL sum

# add u (a0) and a (s1)
ADD a0, a0, s1
ADD a0, a0, s2
# a0 = u + a + b

Epilogue

UNIVERSITY of WASHINGTON

```
int test(int a, int b) {
 int tmp = (a&b)+(a|b);
 int s =sum(tmp,1,2,3,4,5,6,7,8);
 int u = sum(s,tmp,b,a,b,a);
 return u + a + b;
}
```



test:

Prologue

```
MOVE s1, a0
MOVE s2, a1
AND t0, a0, a1
OR t1, a0, a1
ADD t0, t0, t1
MOVE a0, t0
LI a1, 1
LI a2, 2
…
LI a7, 7
LI t1, 8
SW t1, -4(sp)

SW t0, 0(sp)
JAL sum
```

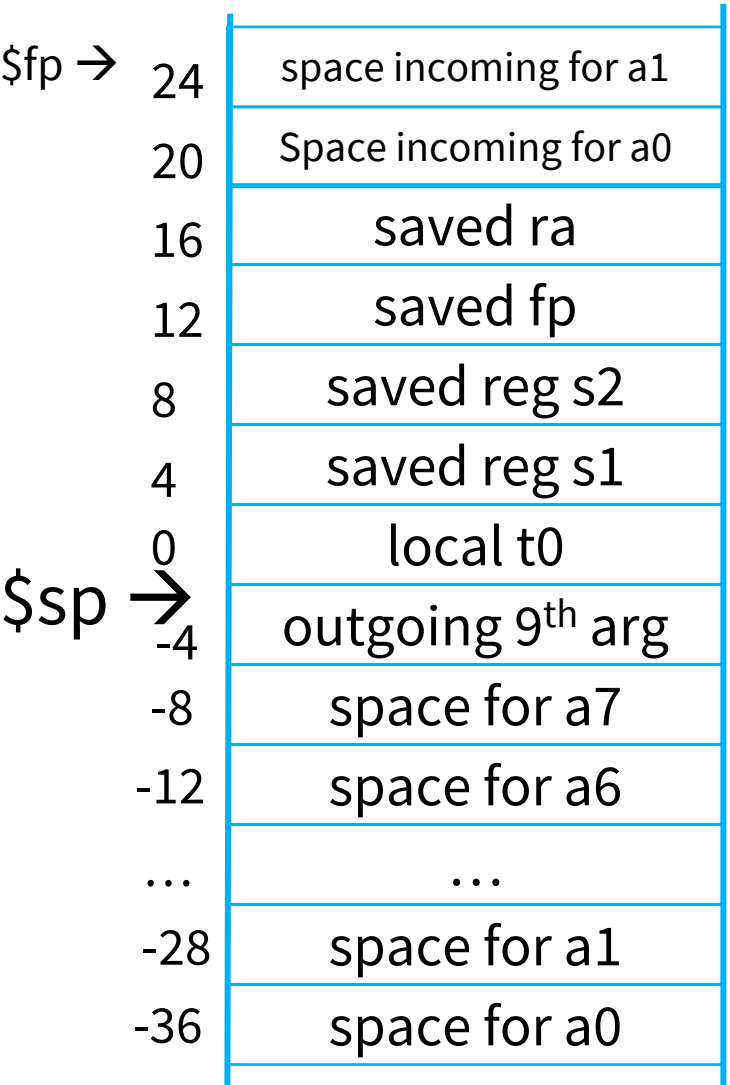
```
LW t0, 0(sp)
MOVE a0, a0 # s
MOVE a1, t0 # tmp
MOVE a2, s2 # b
MOVE a3, s1 # a
MOVE a4, s2 # b
MOVE a5, s1 # a
JAL sum

# add u (a0) and a (s1)
ADD a0, a0, s1
ADD a0, a0, s2
 # a0 = u + a + b
```

Epilogue

UNIVERSITY of WASHINGTON

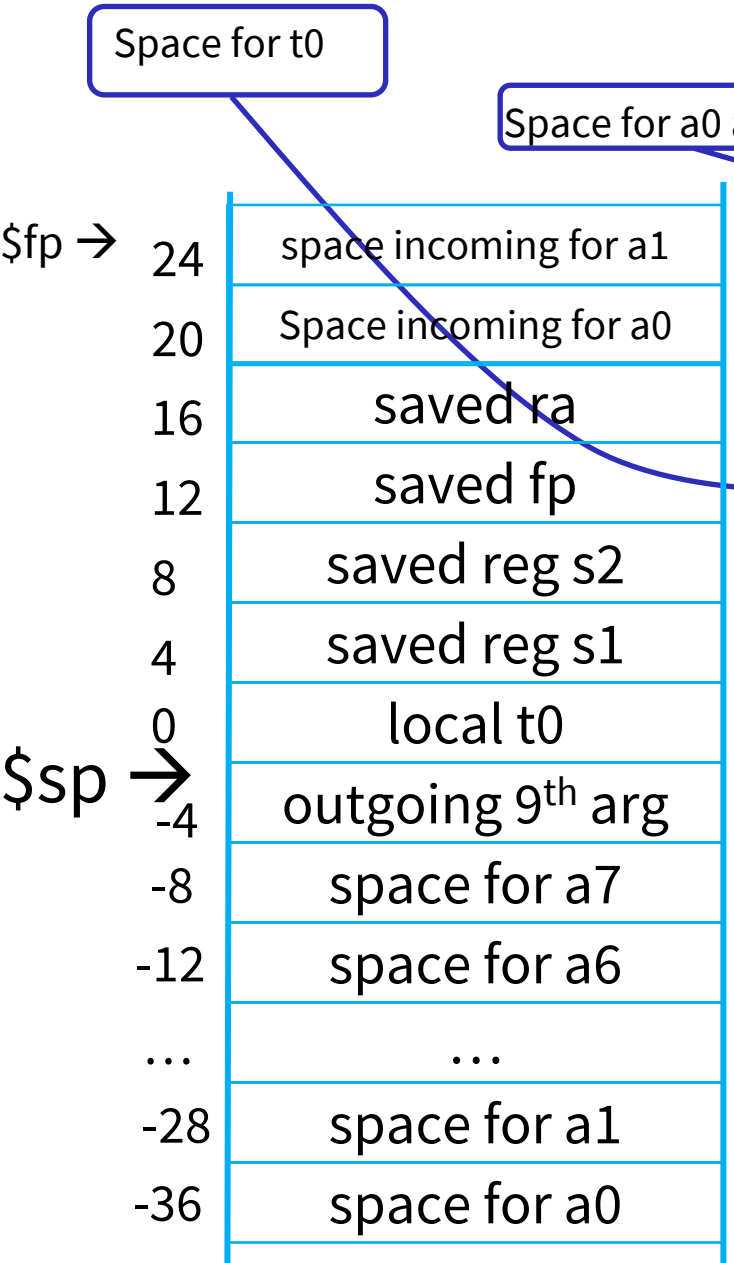
Activity #2: Calling Convention Example:

test:




```
# allocate frame
# save $ra
# save old $fp
# callee save ...
# callee save ...
# set new frame ptr
❖          ...
❖          ...
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
```

UNIVERSITY of WASHINGTON




```
test:
    ADDI sp, sp, -28        # allocate frame
    SW ra, sp, 16           # save $ra
    SW fp, sp, 12           # save old $fp
    SW s2, sp, 8            # callee save ...
    SW s1, sp, 4            # callee save ...
    ADDI fp, sp, 24         # set new frame ptr
    Body                    ...
    (previous slide, Activity #1)
                            ...
    LW s1, sp, 4            # restore ...
    LW s2, sp, 8            # restore ...
    LW fp, sp, 12           # restore old $fp
    LW ra, sp, 16           # restore $ra
    ADDI sp, sp, 28         # dealloc frame
    JR ra
```

UNIVERSITY of WASHINGTON

Function Call Example

```
... sum(a,b); ...          /* a→s0,b→s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

RISC-V

address (decimal)

```
1000  addi a0,s0,0          # x = a
1004  addi a1,s1,0          # y = b
1008  addi ra,x0,1016   # ra=1016
1012  j    sum                # jump to sum
1016
...
2000  sum: add a0,a0,a1
2004  jr   ra               # return
```

Would we know this before compiling?

Otherwise we don't know where we came from

UNIVERSITY of WASHINGTON

Function Call Example

```
... sum(a,b); ...          /* a→s0,b→s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

address (decimal)

RISCV

```
1000  addi a0,s0,0        # x = a
1004  addi a1,s1,0        # y = b
1008  jal  sum            # ra=1012, goto sum
1012
...

2000  sum: add v0,a0,a1
2004  jr   ra             # return
```

UNIVERSITY of WASHINGTON

Example: sumSquare

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

- What do we need to save?
  - Call to `mult` will overwrite `ra`, so save it
  - Reusing `a1` to pass 2nd argument to `mult`, but need current value (`y`) later, so save `a1`
- To save something to the Stack, move `sp` *down* the required amount and fill the "created" space

UNIVERSITY of WASHINGTON

Example: sumSquare

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
        addi sp,sp,-8       # make space on stack
"push"  sw ra, 4(sp)        # save ret addr
        sw a1, 0(sp)        # save y
        add a1,a0,x0        # set 2nd mult arg
        jal mult            # call mult
        lw a1, 0(sp)        # restore y
"pop"   add a0,a0,a1        # ret val = mult(x,x)+y
        lw ra, 4(sp)        # get ret addr
        addi sp,sp,8        # restore stack
        jr ra
mult:   ...
```

UNIVERSITY of WASHINGTON

Example: Using Volatile Registers

```
myFunc: # Uses t0
   addiu       sp,sp,-4     # This is the Prologue
   sw          ra,0(sp)     # Save saved registers
   ...                      # Do stuff with t0
   addiu       sp,sp,-4     # Save volatile registers
   sw          t0,0(sp)     #   before calling a function
   jal         func1        # Function may change t0
   lw          t0,0(sp)     # Restore volatile registers
   addiu       sp,sp,4      #   before you use them again
   ...                      # Do stuff with t0
   lw          ra,0(sp)     # This is the Epilogue
   addiu       sp,sp,4    # Restore saved registers
   jr          ra          # return
```

UNIVERSITY of WASHINGTON

Example: Using Saved Registers

```
myFunc: # Uses s0 and s1
   addiu       sp,sp,-12    # This is the Prologue
   sw          ra,8(sp)     # Save saved registers
   sw          s0,4(sp)
   sw          s1,0(sp)
   ...                      # Do stuff with s0 and s1
   jal         func1        # s0 and s1 unchanged by
   ...                      #   function calls, so can keep
   jal         func2        #   using them normally
   ...                      # Do stuff with s0 and s1
   lw          s1,0(sp)     # This is the Epilogue
   lw          s0,4(sp)     # Restore saved registers
   lw          ra,8(sp)
   addiu       sp,sp,12
   jr          ra           # return
```

UNIVERSITY of WASHINGTON

Choosing Your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save when you absolutely have to
- Function does NOT call another function
  - Use only `t0-t6` and there is nothing to save!
- Function calls other function(s)
  - Values you need throughout go in `s0-s11`, others go in `t0-t6`
  - At each function call, check number arguments and return values for whether you or not you need to save

UNIVERSITY of WASHINGTON

Agenda

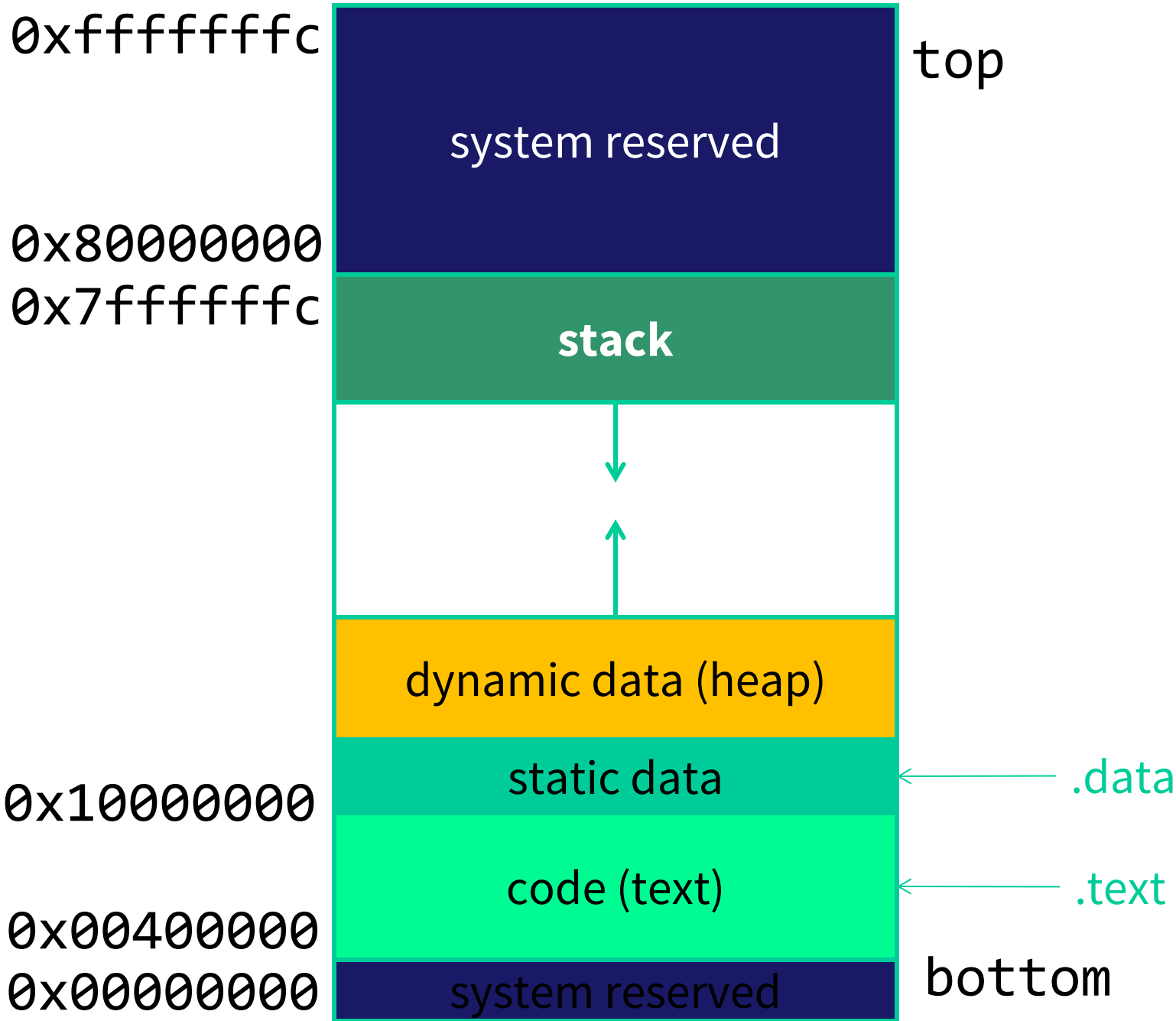
- Function calls and Jumps
- Call Stack
- Register Convention
- Program memory layout

UNIVERSITY of WASHINGTON

Where is the Stack in Memory?

- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal: **`bfff_fff0`**$_{hex}$
  - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV32 programs (*text segment*) in low end
  - `0001_0000`$_{hex}$
- *static data segment* (constants and other static variables) above text for static variables
  - RISC-V convention *global pointer* (**`gp`**) points to static
  - RV32 **`gp`** = **`1000_0000`**$_{hex}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

UNIVERSITY of WASHINGTON



UNIVERSITY of WASHINGTON

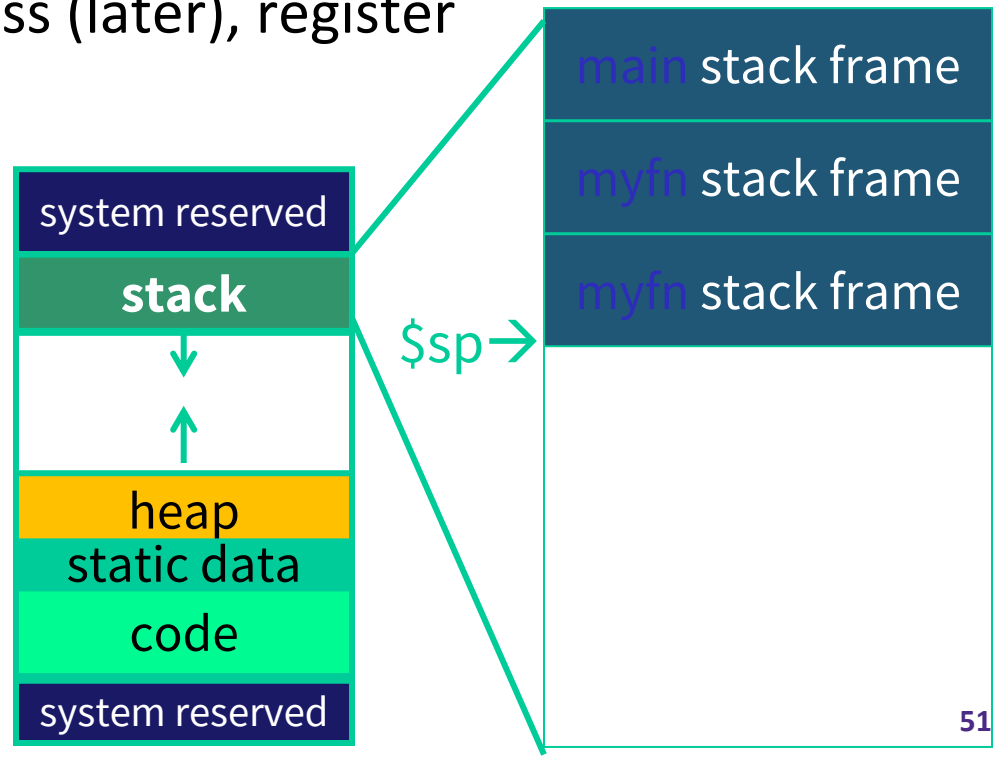
Stack contains stack frames (aka "activation records")

- 1 stack frame per dynamic function
- Exists only for the duration of function
- Grows down, "top" of stack is sp, x2
- Example: lw x5, 0(sp) puts word at top of stack into x5

Each stack frame contains:

- Local variables, return address (later), register backups (later)

```
int main(…) {
      …
      myfn(x);
}
int myfn(int n) {
      …
      myfn();
}
```



Frame Pointer

It is often cumbersome to keep track of location of data on the stack

- The offsets change as new values are pushed onto and popped off of the stack

Keep a pointer to the bottom of the top stack frame

- Simplifies the task of referring to items on the stack

A frame pointer, x8, aka fp/s0

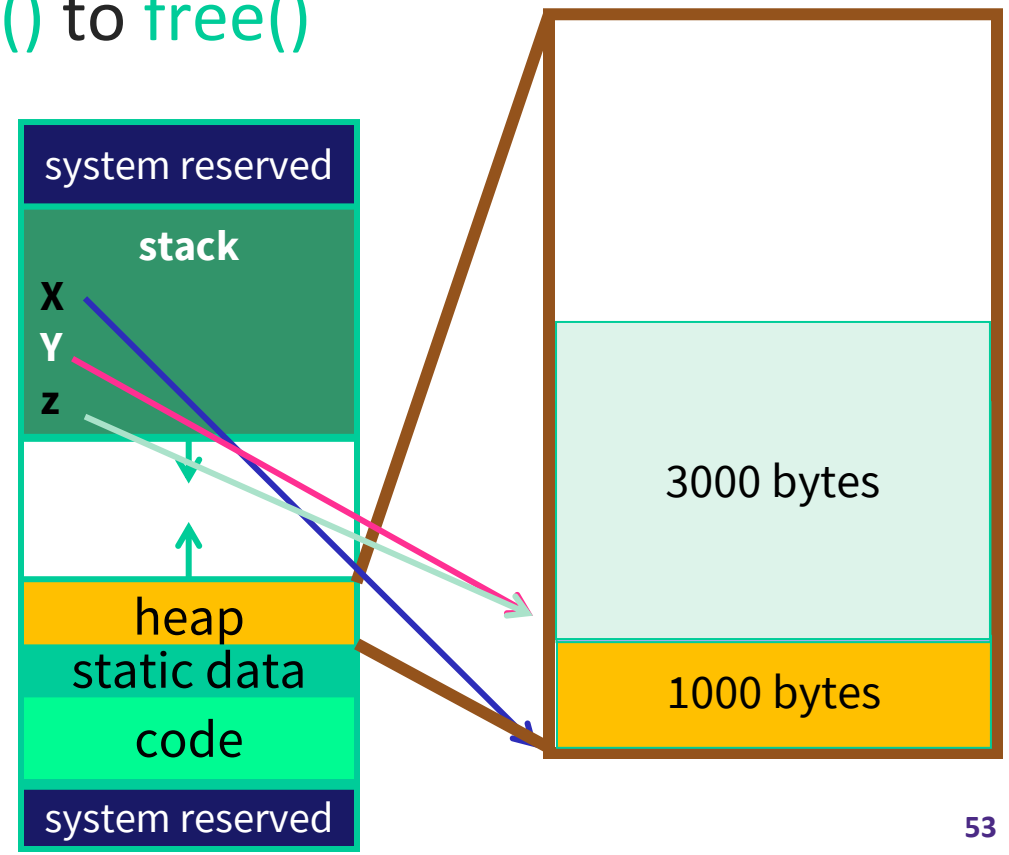
- Value of sp upon procedure entry
- Can be used to restore sp on exit

UNIVERSITY of WASHINGTON

The Heap

- Heap holds dynamically allocated memory
- Program must maintain pointers to anything allocated
  - Example: if x5 holds x
  - lw x6, 0(x5) gets first word x points to
- Data exists from malloc() to free()

```
void some_function() {
  int *x = malloc(1000);
  int *y = malloc(2000);
  free(y);
  int *z = malloc(3000);
}
```

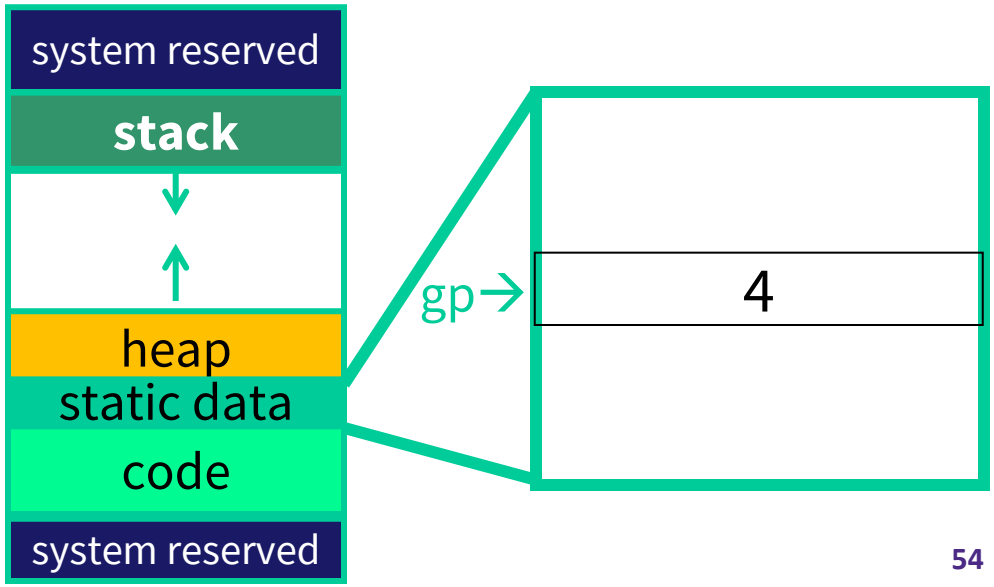


Data Segment

Data segment contains global variables

- Exist for all time, accessible to all routines
- Accessed w/global pointer
  - gp, x3, points to middle of segment
  - Example: lw x5, 0(gp) gets middle-most word (here, max_players)

```
int max_players = 4;

int main(...) {
      ...
}
```



UNIVERSITY of WASHINGTON

Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

```
int n = 100;
int main (int argc, char* argv[ ]) {
      int i, m = n, sum = 0;
      int* A = malloc(4*m + 4);
      for (i = 1; i <= m; i++) {
         sum += i; A[i] = sum; }
      printf ("Sum 1 to %d is %d\n", n, sum);
}
```

Where is `main` ?

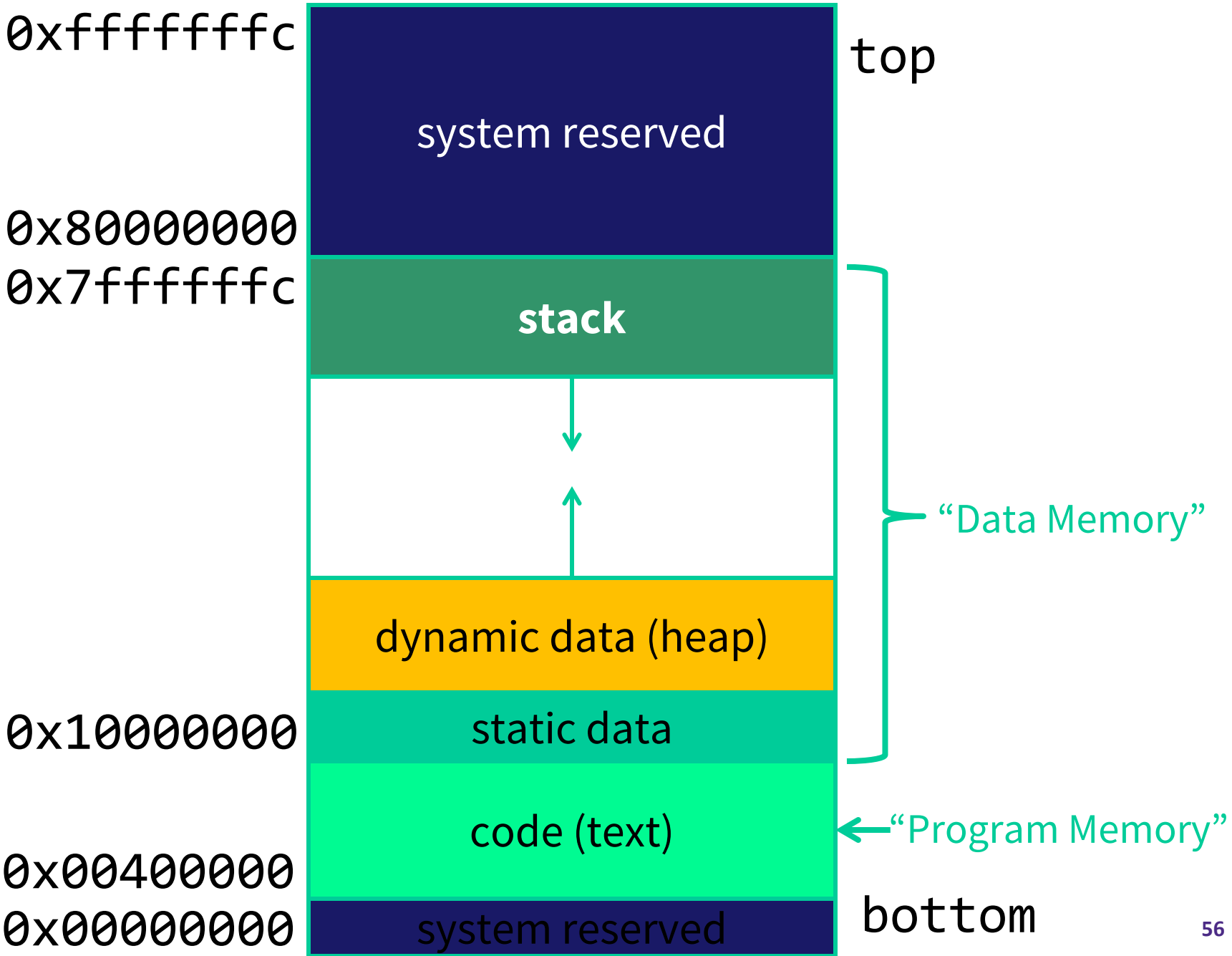
(A) Stack

(B) Heap

(C) Global Data

(D) Text

UNIVERSITY of WASHINGTON



UNIVERSITY of WASHINGTON

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum, A	w/in function	function invocation	stack
Global n, str	whole program	program execution	.data
Dynamic *A	Anywhere that has a pointer	b/w malloc and free	heap

```
int n = 100;
int main (int argc, char* argv[ ]) {
     int i, m = n, sum = 0;
     int* A = malloc(4*m + 4);
     for (i = 1; i <= m; i++) {
        sum += i; A[i] = sum; }
     printf ("Sum 1 to %d is %d\n", n, sum);
}
```

UNIVERSITY of WASHINGTON

Global and Locals

How does a function load global data?

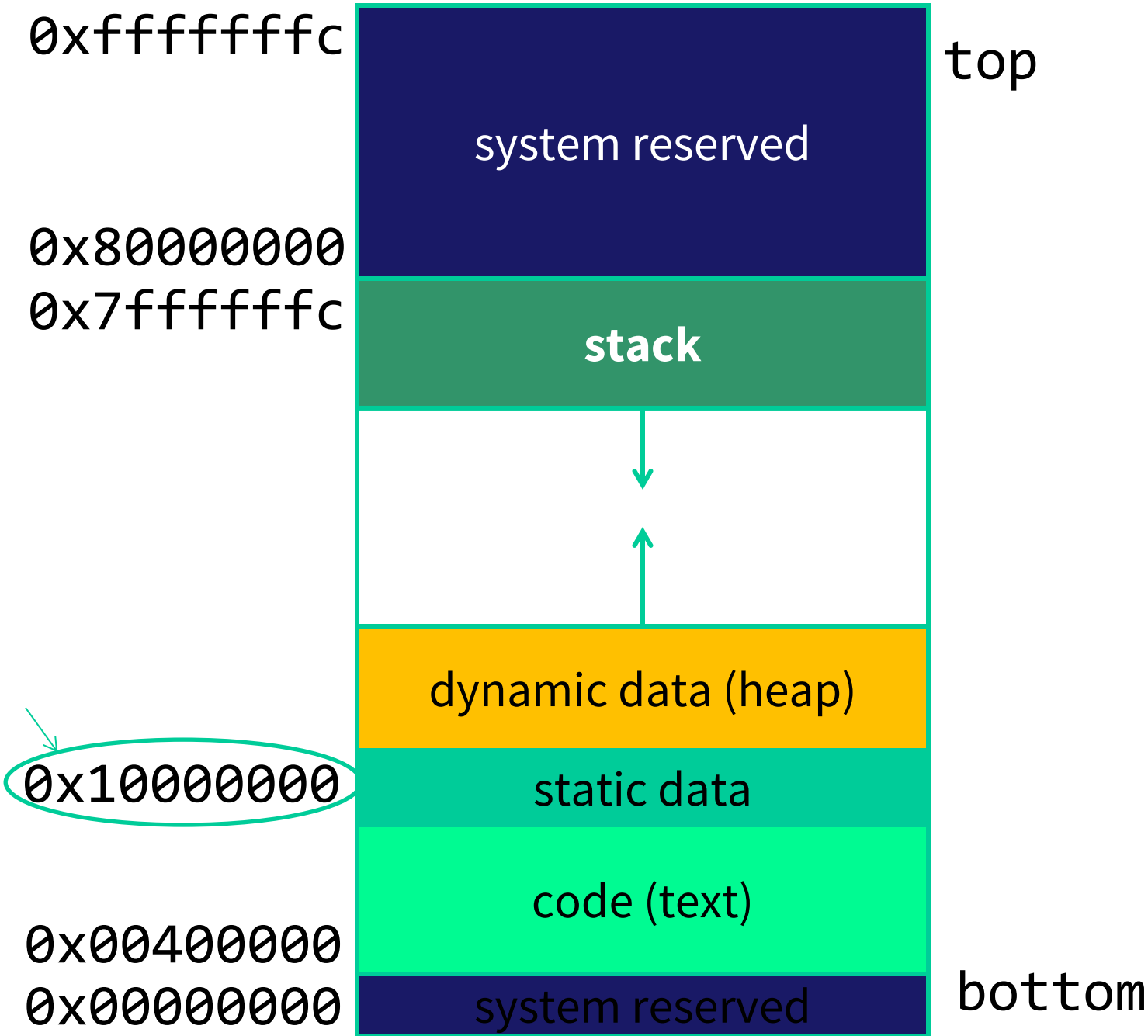
- global variables are just above 0x10000000

Convention: *global pointer*

- x3 is gp (pointer into *middle* of global data section)
  `gp = 0x10000800`
- Access most global data using LW at gp +/- offset

```
LW t0, 0x800(gp)
LW t1, 0x7FF(gp)
```

Anatomy of an executing program



UNIVERSITY of WASHINGTON

Calling Convention for Procedure Calls

~~Transfer Control~~

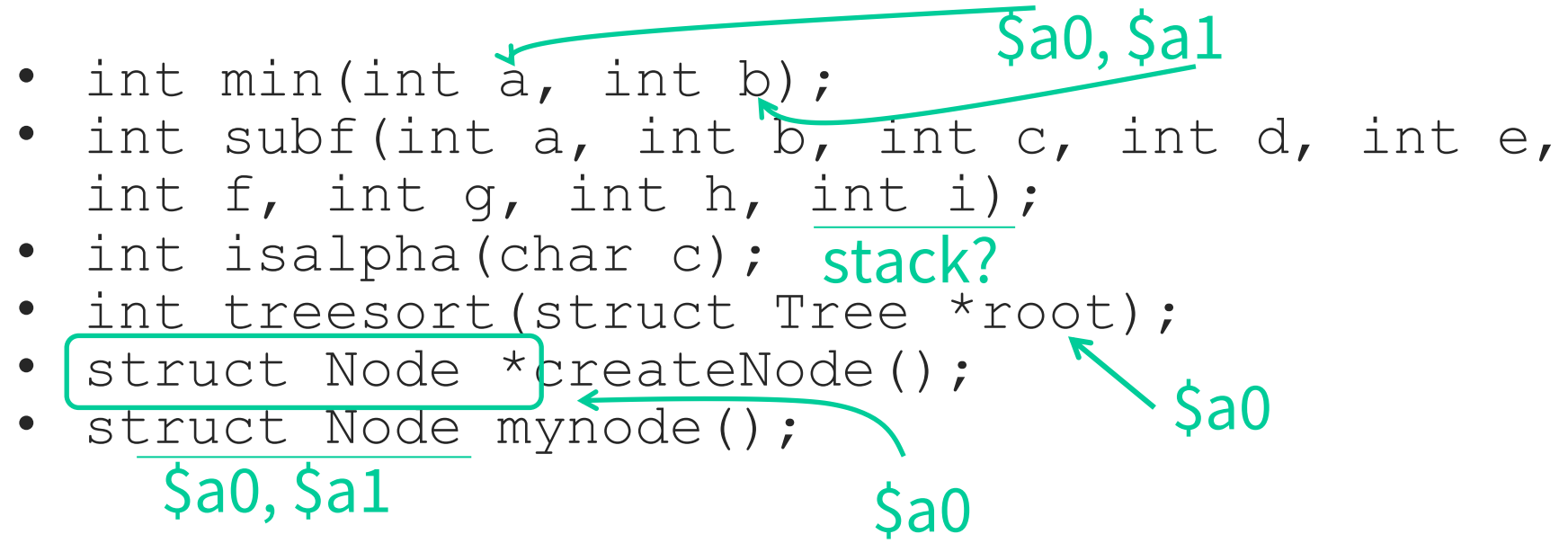
- ~~Caller → Routine~~
- ~~Routine → Caller~~

Pass Arguments to and from the routine

- fixed length, variable length, recursively
- Get return value back to the caller

Arguments & Return Values

Need consistent way of passing args and result
Given a procedure signature, need to know where arguments should be placed



Too many combinations of char, short, int, void *, struct, etc.

- RISC-V treats char, short, int and void * identically

Simple Argument Passing (1-8 args)

```
main() {
  int x = myfn(6, 7);
  x = x + 2;
}
```

```
main:
  li x10, 6
  li x11, 7
  jal myfn
  addi x5, x10, 2
```

First eight arguments:

passed in registers x10-x17

- aka $a0, $a1, …, $a7

Returned result:

passed back in a register

- Specifically, x10, aka a0
- And x11, aka a1

Note: This is *not* the entire story for 1-8 arguments. Please see *the Full Story* slides.

Conventions so far:

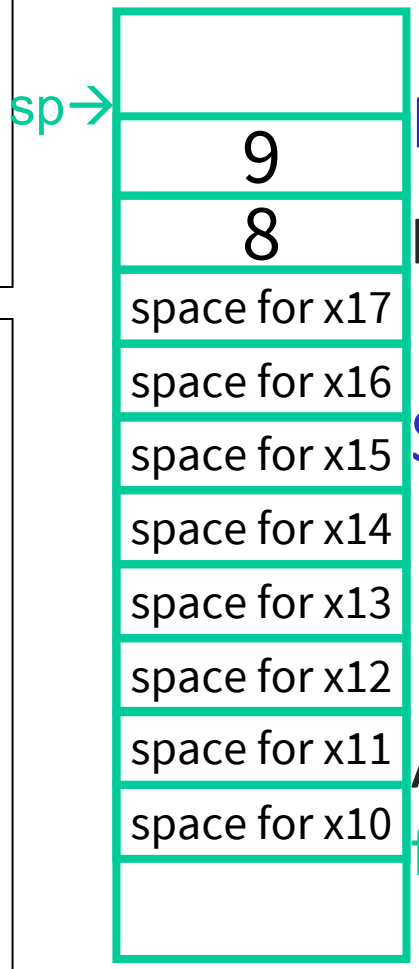
- args passed in $a0, $a1, …, $a7
- return value (if any) in $a0, $a1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)

Q: What about argument lists?

Many Arguments (8+ args)

```
main() {
  myfn(0,1,2,..,7,8,9);
  …
}
```

```
main:
  li x10, 0
  li x11, 1
  …
  li x17, 7
  li x5, 8
  sw x5, -8(x2)
  li x5, 9
  sw x5, -4(x2)
  jal myfn
```



First eight arguments:

passed in registers x10-x17

- aka a0, a1, …, a7

Subsequent arguments:

"spill" onto the stack

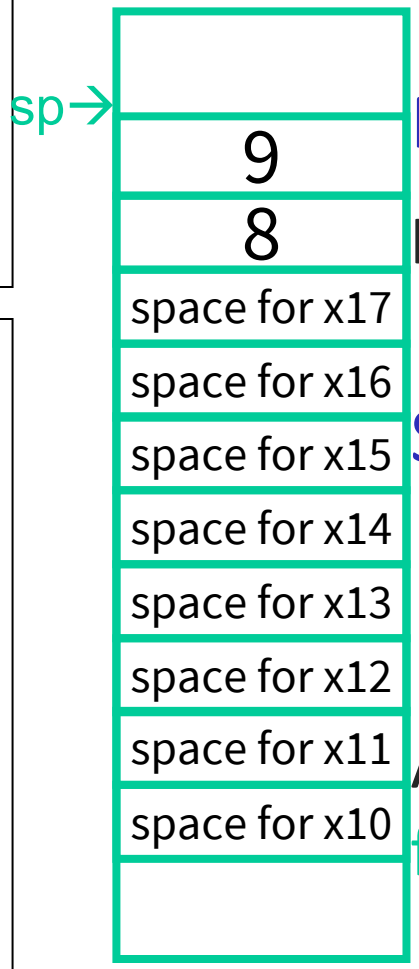
Args passed in child's stack frame

Note: This is *not* the entire story for 9+ args.
Please see *the Full Story* slides.

Many Arguments (8+ args)

```
main() {
  myfn(0,1,2,..,7,8,9);
  …
}
```

```
main:
  li a0, 0
  li a1, 1
  …
  li a7, 7
  li t0, 8
  sw t0, -8(sp)
  li t0, 9
  sw t0, -4(sp)
  jal myfn
```



First eight arguments:

passed in registers x10-x17

- aka a0, a1, …, a7

Subsequent arguments:

"spill" onto the stack

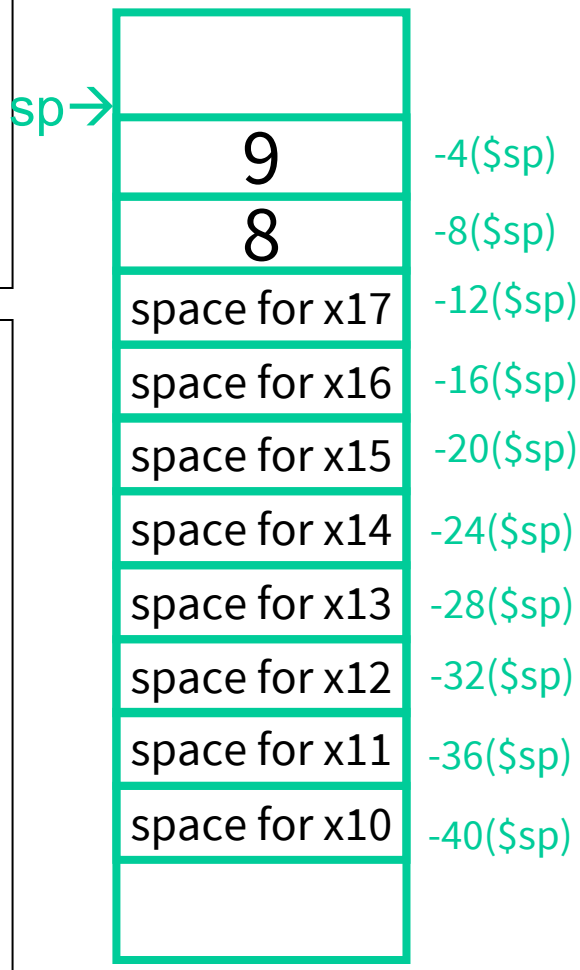
Args passed in child's stack frame

Note: This is *not* the entire story for 9+ args.
Please see *the Full Story* slides.

Argument Passing: *the Full Story*

```
main() {
  myfn(0,1,2,..,7,8,9);
  …
}
```

```
main:
  li a0, 0
  li a1, 1
  …
  li a7, 7
  li t0, 8
  sw t0, -8(x2)
  li t0, 9
  sw t0, -4(x2)
  jal myfn
```



Arguments 1-8:

passed in x10-x17 *room on stack*

Arguments 9+:

placed on stack

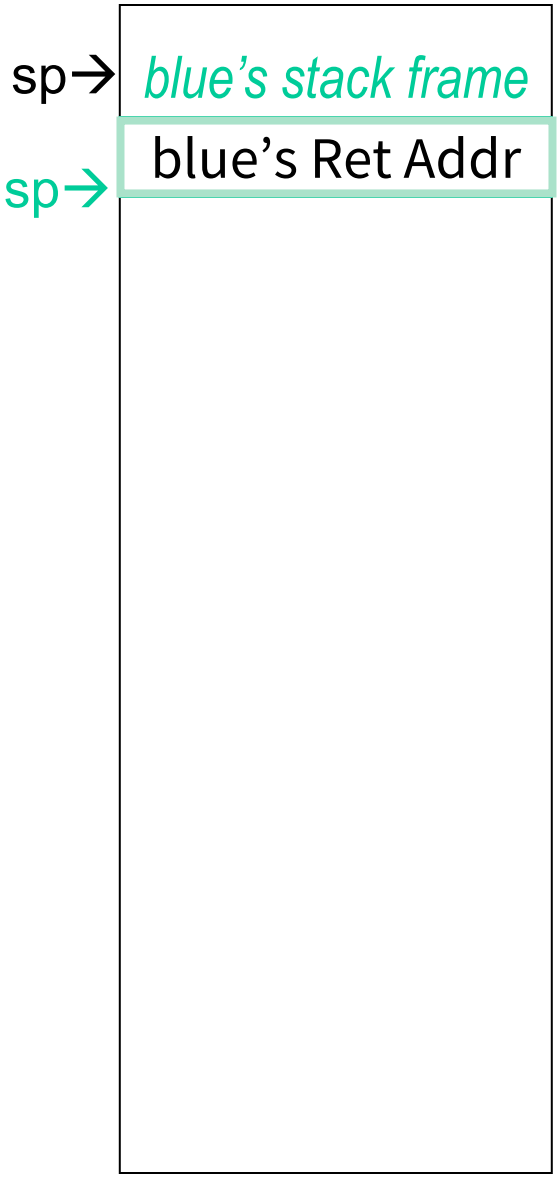
Args passed in child's stack frame

Pros of Argument Passing Convention

- Consistent way of passing arguments to and from subroutines
- Creates single location for all arguments
  - Caller makes room for a0-a7 on stack
  - Callee must copy values from a0-a7 to stack

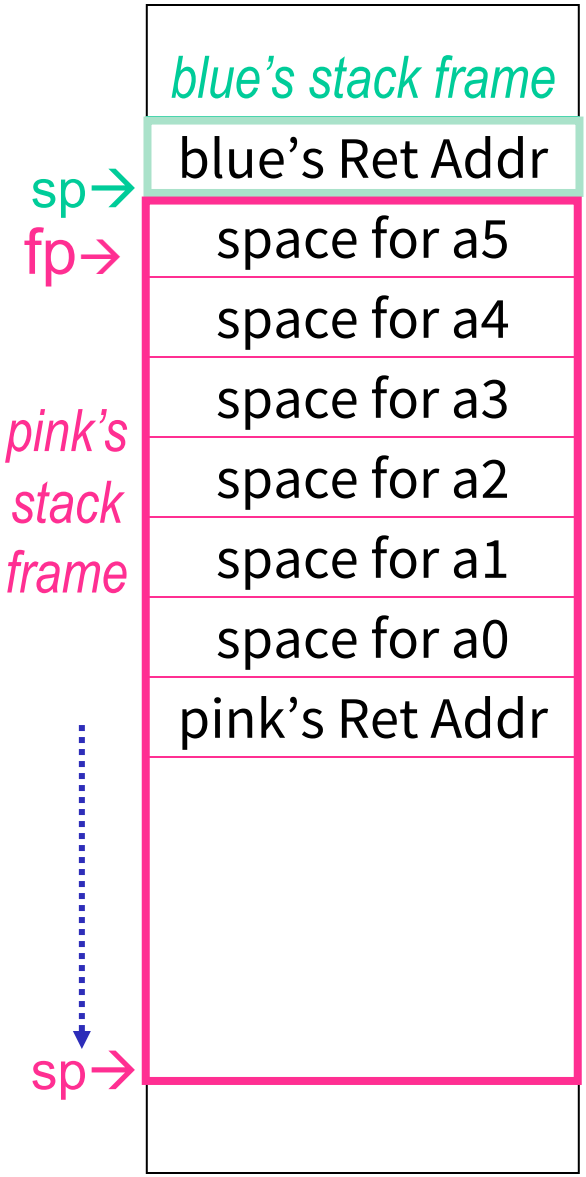
  → callee may treat all args as an array in memory
  - Particularly helpful for functions w/ variable length inputs: `printf("Scores: %d %d %d\n", 1, 2, 3);`
- Aside: not a bad place to store inputs if callee needs to call a function (your input cannot stay in $a0 if you need to call another function!)

Frame Layout & the Frame Pointer




```
blue() {
    pink(0,1,2,3,4,5);
}
```

Frame Layout & the Frame Pointer



Notice

- Pink's arguments are on **pink's** stack
- **sp** changes as functions call other functions, complicates accesses

→ Convenient to keep pointer to bottom of stack == **frame pointer**

x8, aka fp (also known as s0)

can be used to restore sp on exit

```
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    ...
}
```