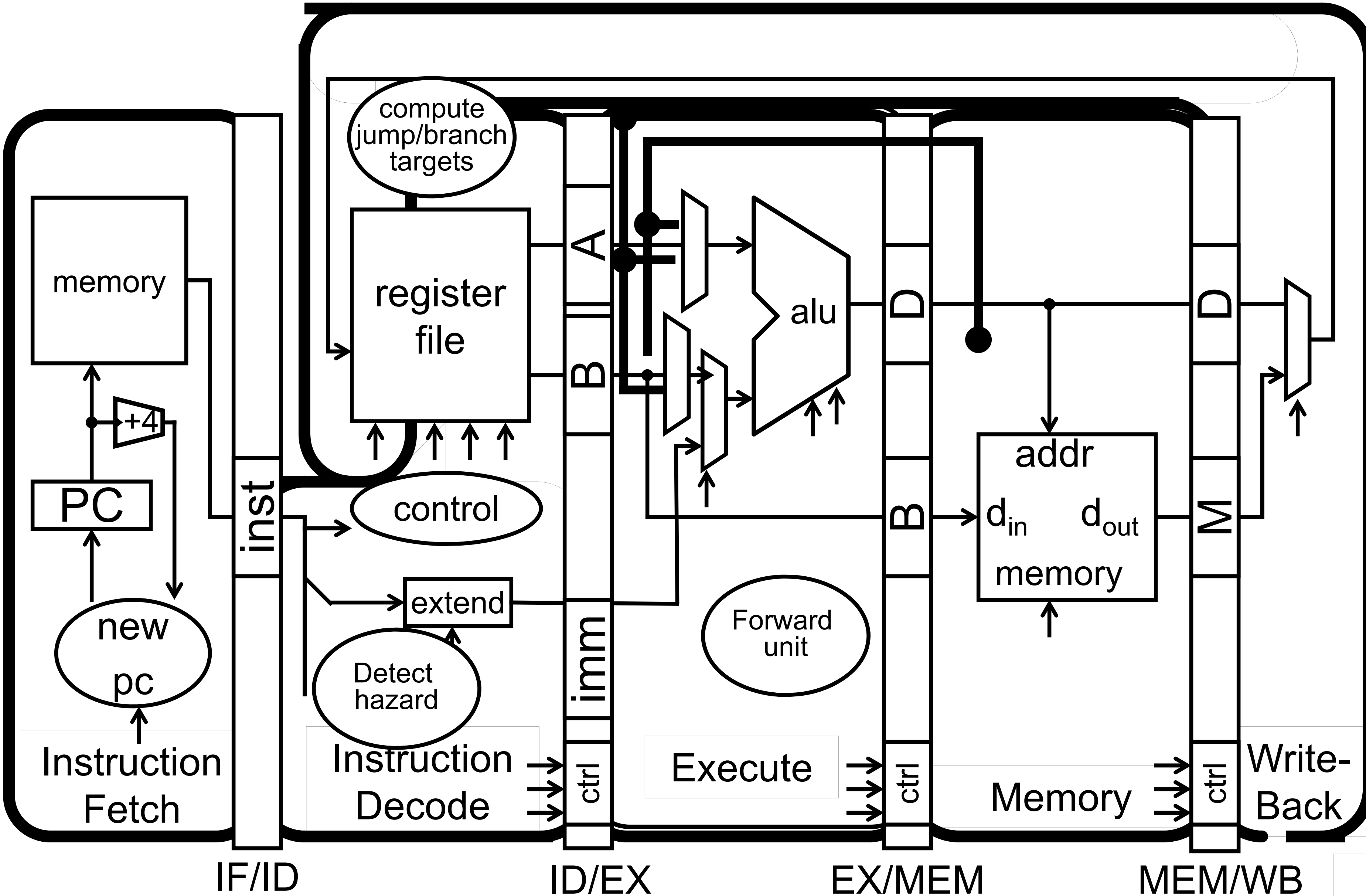
Calling Conventions

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, McKee and Sirer]

Big Picture: Where are we going?



Big Picture: Where are we going?

C

compiler

RISC-V assembly

assembler

machine code

CPU

Circuits

Gates

Transistors

Silicon

```
int x = 10;
x = 2 * x + 15;
```

x0 = 0
x5 = x0 + 10
x5 = x5<<1 #x5 = x5 * 2
x5 = x15 + 15

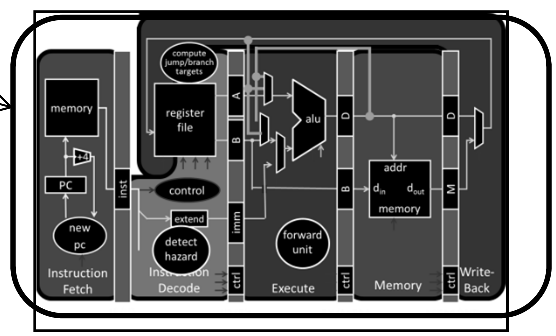
```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```

10	r0		r5	op = addi
000000001010	00000	000	00101	0010011

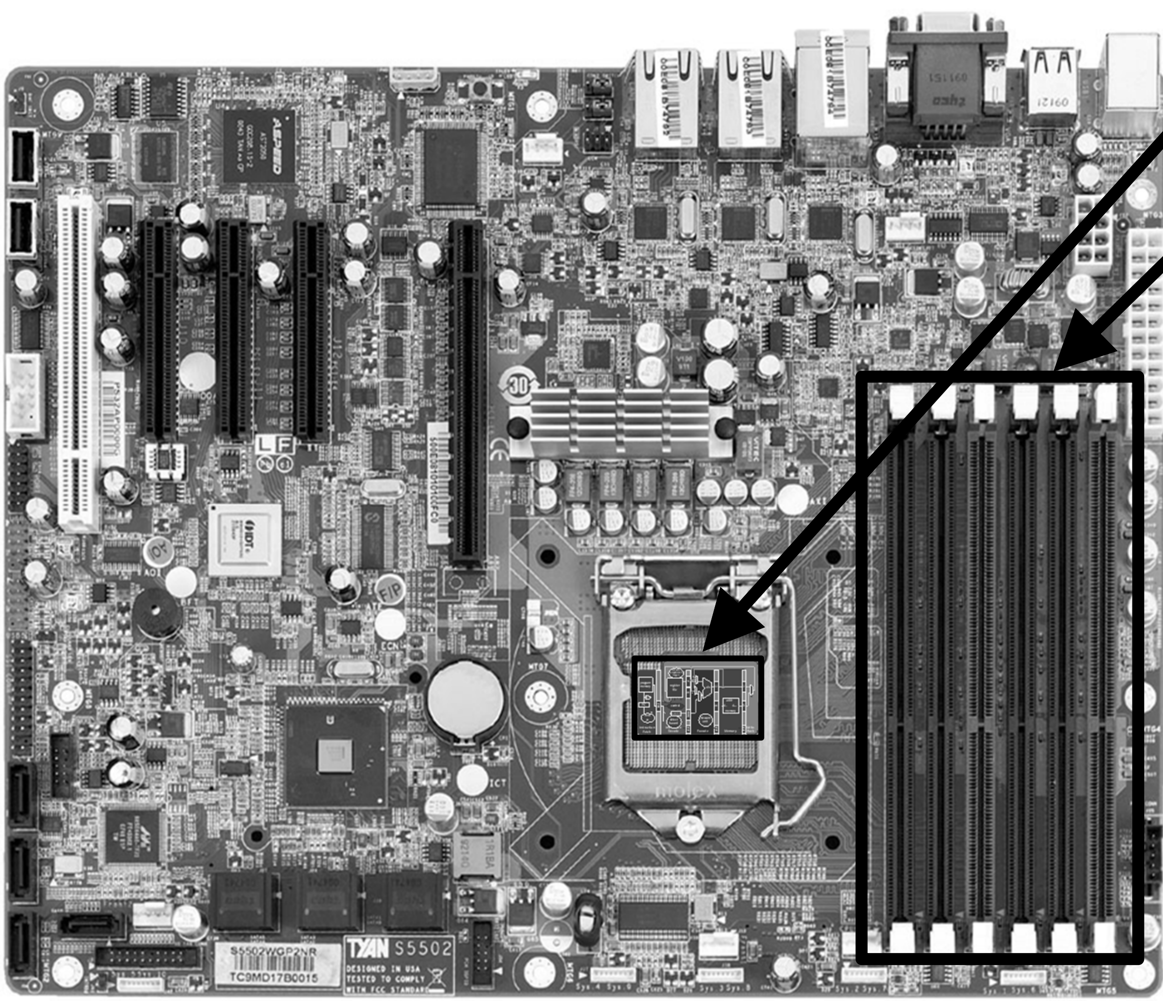
00000000001000101000001010000000

000000001111	00101	000	00101	0010011
15	r5		r5	op = addi

op = r-type x5 shamt=1 x5 func=sll



Big Picture: Where are we going?



CPU

Main Memory (DRAM)

SandyBridge Motherboard, 2011
http://news.softpedia.com

Goals for this week

Calling Convention for Procedure Calls
Enable code to be reused by allowing code snippets to be invoked

Will need a way to

- call the routine (i.e. transfer control to procedure)
- pass arguments
  - fixed length, variable length, recursively
- return to the caller
  - Putting results in a place where caller can find them
- Manage register

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

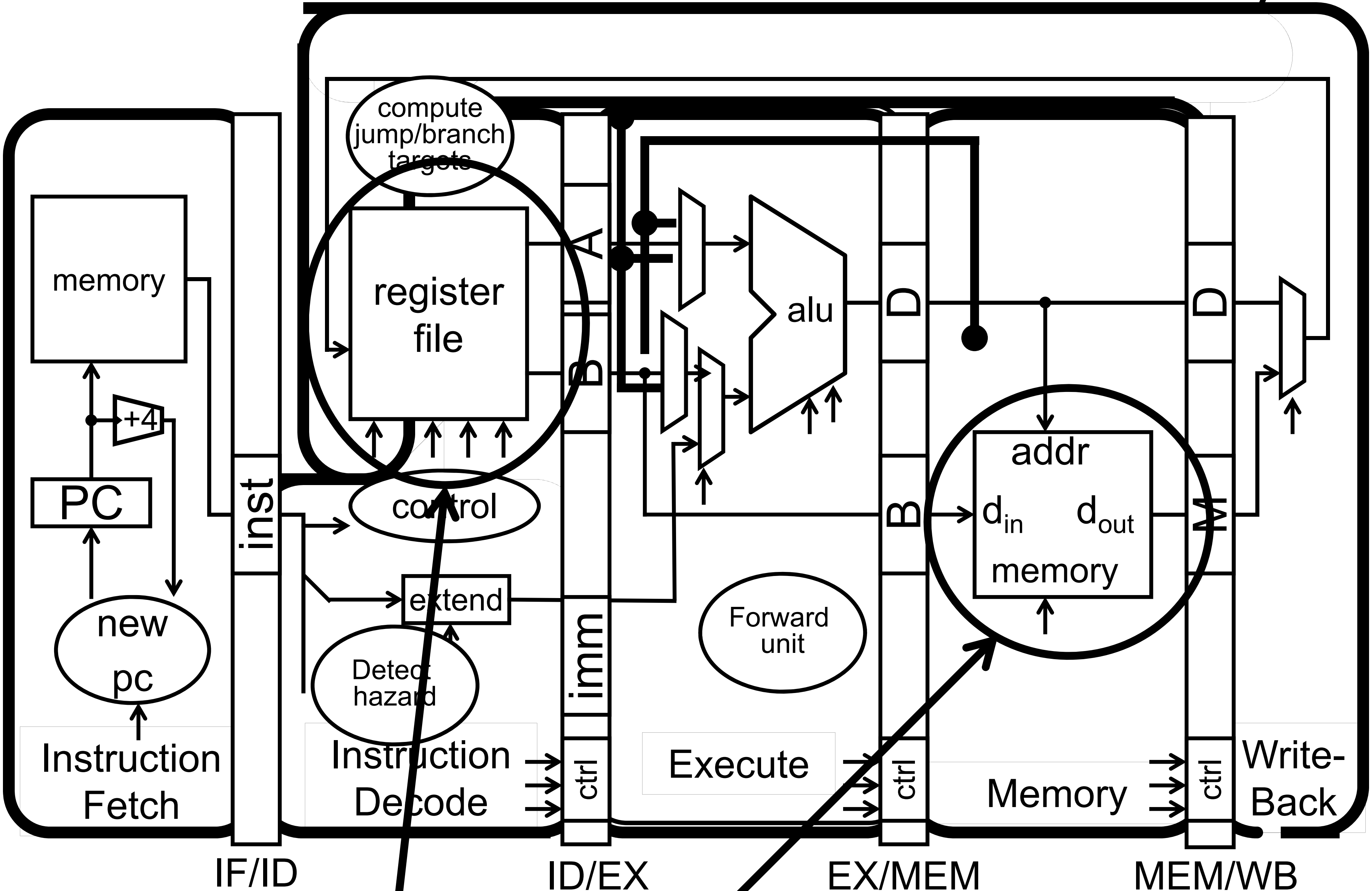
- Allow each routine to use registers
- Prevent routines from clobbering each others' data

What is a Convention?

Warning: There is no one true RISC-V calling convention.

lecture != book != gcc != spim != web

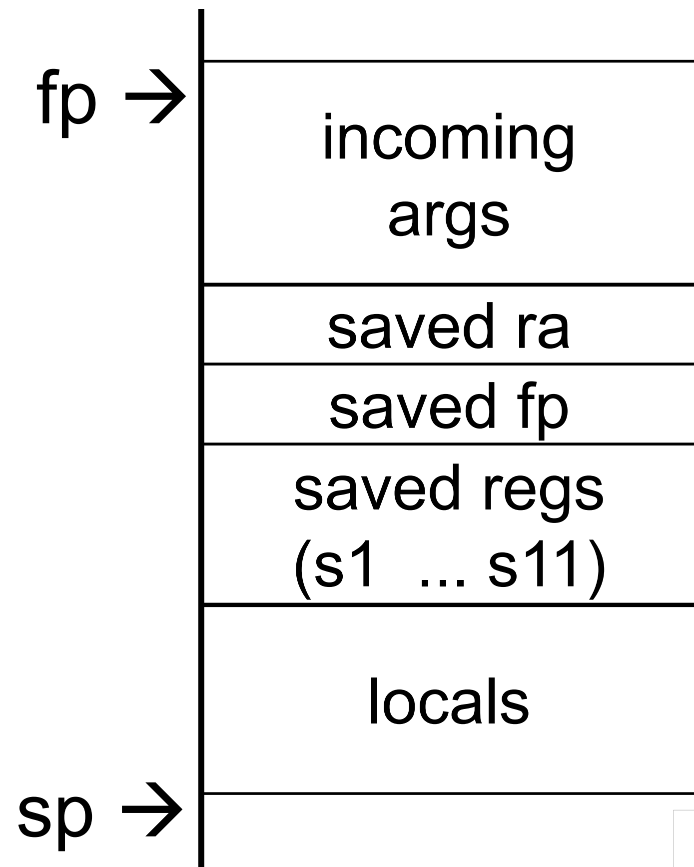
Cheat Sheet and Mental Model for Today



How do we share registers and use memory when making procedure calls?

Cheat Sheet and Mental Model for Today

- first eight arg words passed in a0, a1, … , a7
- remaining arg words passed in parent's stack frame
- return value (if any) in a0, a1
- stack frame at sp
  - contains ra (clobbered on JAL to sub-functions)
  - contains local vars (possibly clobbered by sub-functions)
  - contains space for incoming args
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp



RISC-V Register

- Return address: x1 (ra)
- Stack pointer: x2 (sp)
- Frame pointer: x8 (fp/s0)
- First eight arguments: x10-x17 (a0-a7)
- Return result: x10-x11 (a0-a1)
- Callee-save free regs: x18-x27 (s2-s11)
- Caller-save free regs: x5-x7,x28-x31 (t0-t6)
- Global pointer: x3 (gp)
- Thread pointer: x4 (tp)

RISC-V Register Conventions

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	

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

What is a Convention?

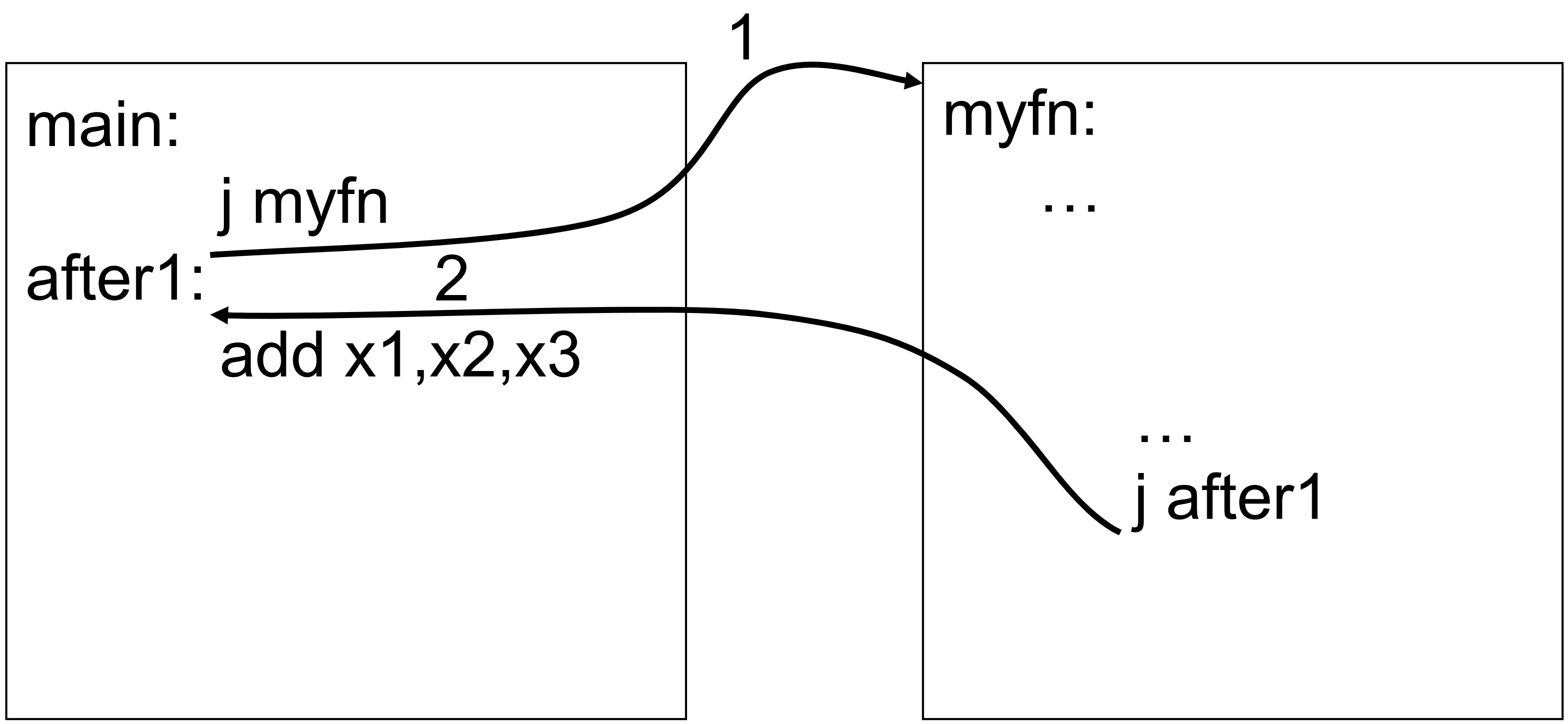
Warning: There is no one true RISC-V calling convention.

lecture != book != gcc != spim != web

How does a function call work?

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

int myfn(int n) {
   int f = 1;
   int i = 1;
   int j = n – 1;
   while(j >= 0) {
      f *= i;
      i++;
      j = n - i;
   }
   return f;
}
```

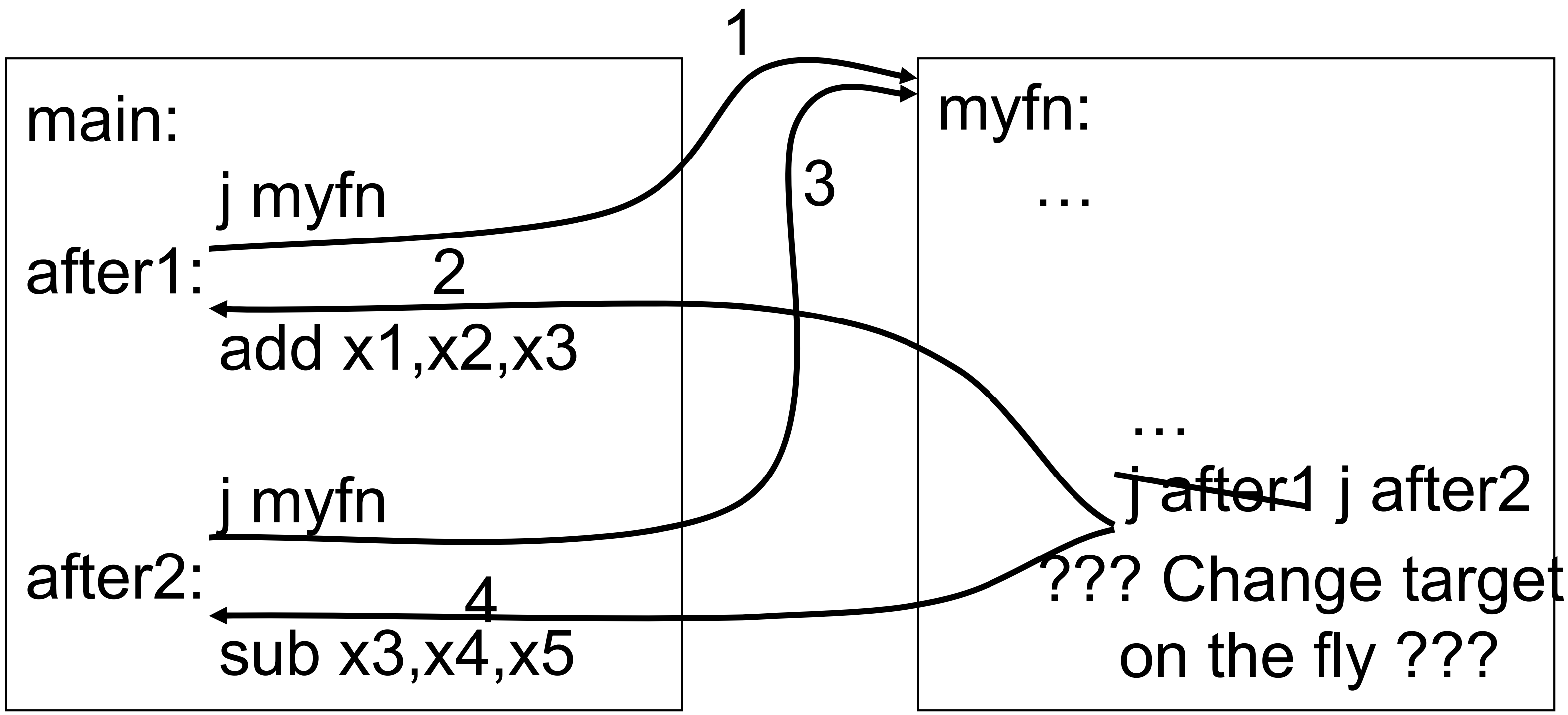
Jumps are not enough



Jumps to the callee

Jumps back

Jumps are not enough



Jumps to the callee

Jumps back

What about multiple sites?

Takeaway1: 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*

1

```
main:
        jal myfn
after1:
        add x1,x2,x3

        jal myfn
after2:
        sub x3,x4,x5
```

2

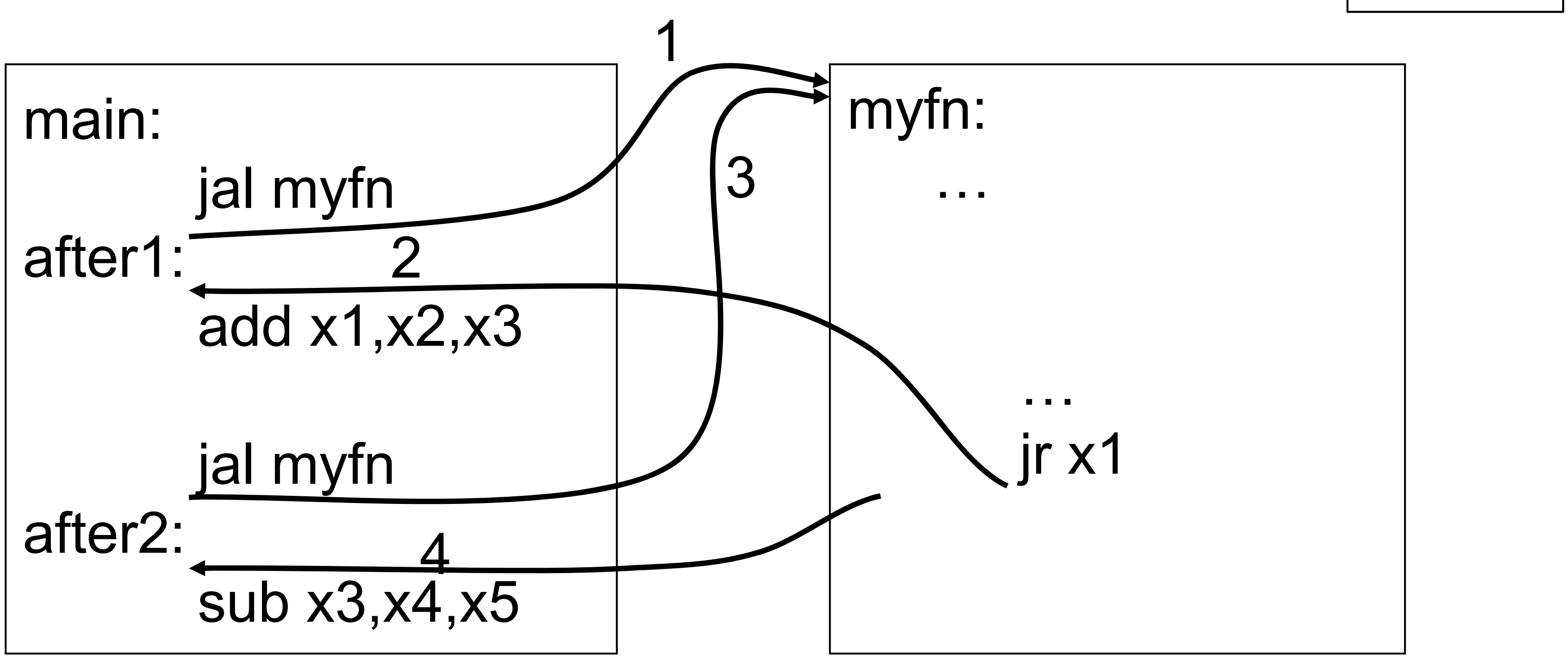
```
myfn:
    …

        …
        jr x1
```

JAL saves the PC in register $31

Subroutine returns by jumping to $31

Jump-and-Link / Jump Register

Second call



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) {
   int f = 1;
   int i = 1;
   int j = n – 1;
   while(j >= 0) {
       f *= i;
       i++;
       j = n - i;
   }
   return f;
}
```

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*

1

```
main:
        jal myfn
after1:
        add x1,x2,x3
```

```
myfn:
        if (test)
      jal myfn
after2:

     jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Recursive Call

x1 *after2*

1

```
main:
        jal myfn
after1:
        add x1,x2,x3
```

2

```
myfn:
        if (test)
      jal myfn
after2:

    jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Recursive Call

x1 *after2*

1 2 3

```
main:
        jal myfn
after1:
        add x1,x2,x3
```

```
myfn:
        if (test)
      jal myfn
after2:

      jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???

x1 *after2*

1 2

```
main:
        jal myfn
after1:
        add x1,x2,x3
```

```
myfn:
        if (test)
      jal myfn
after2:
             Stuck!
      jr x1
```

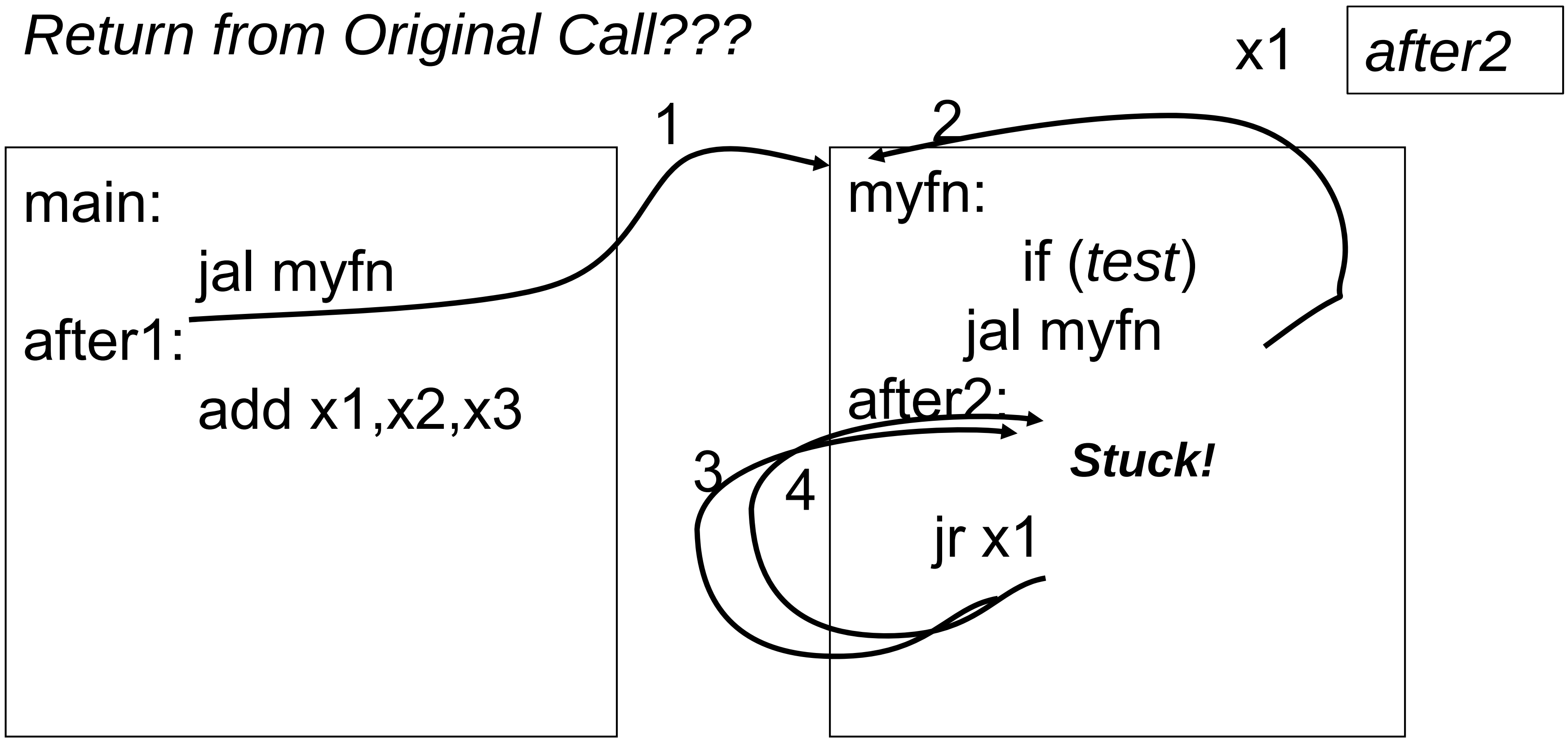
3 4

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???



Problems with recursion:

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

Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

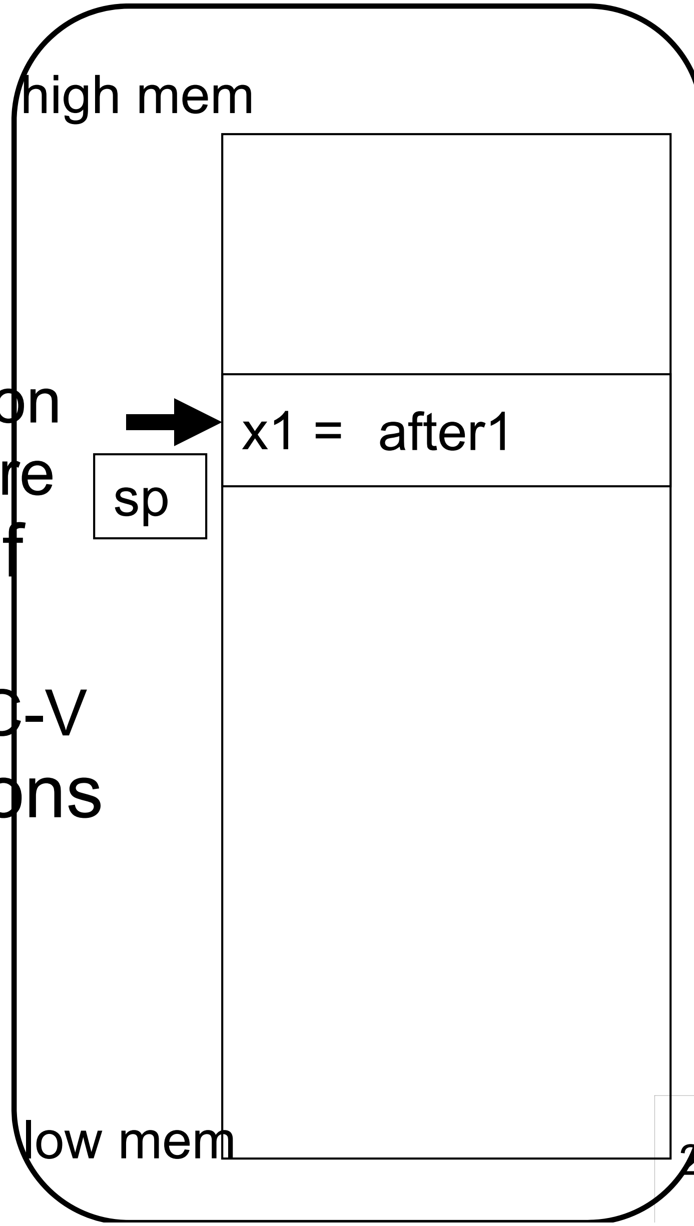
- the return address for that invocation
- the local variables for that procedure

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

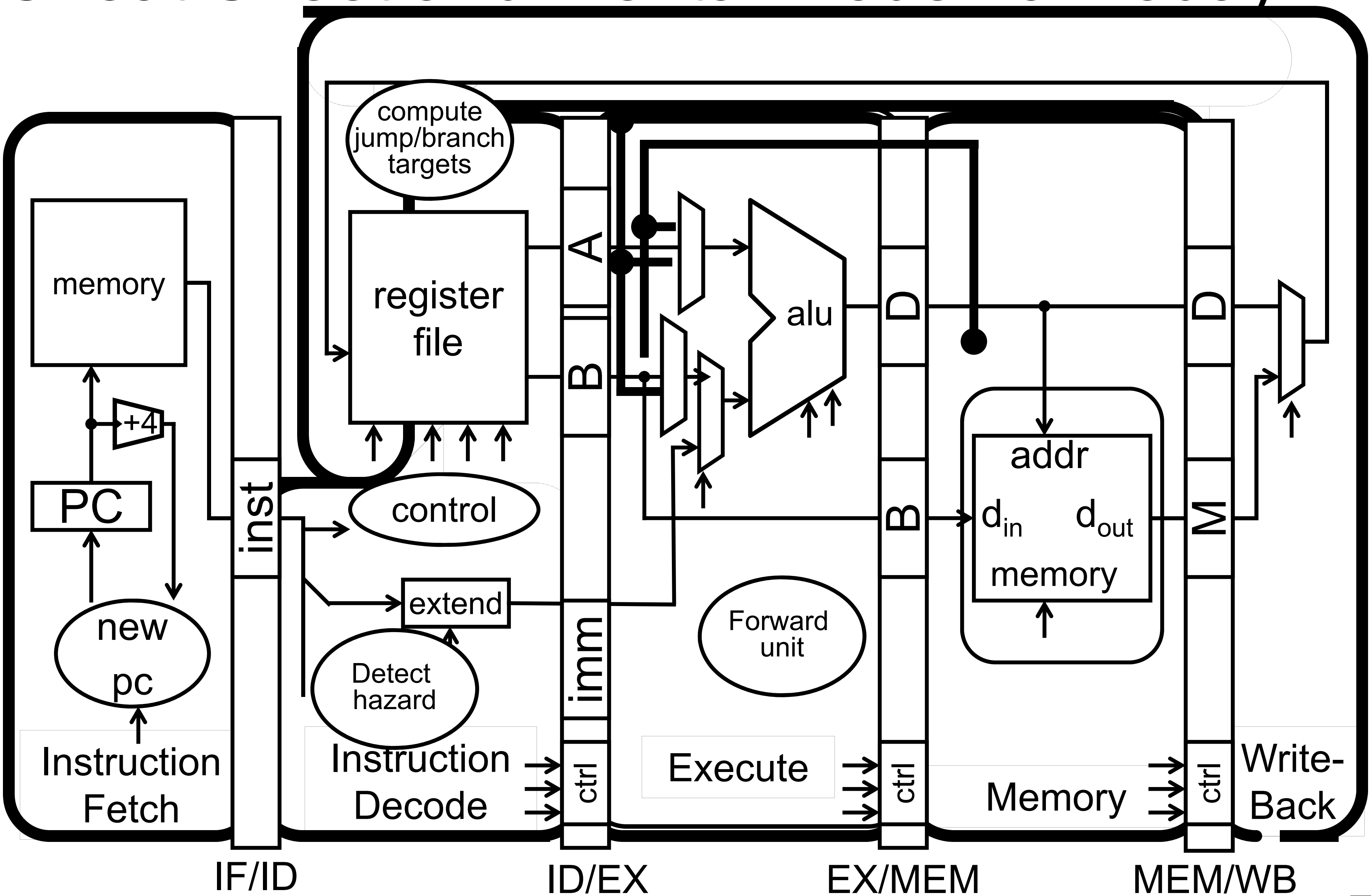
- dedicated register (x2) on the RISC-V

Manipulated by push/pop operations

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



Cheat Sheet and Mental Model for Today



Need a “Call Stack”

Call stack

- contains activation records (aka stack frames)

Each activation record contains

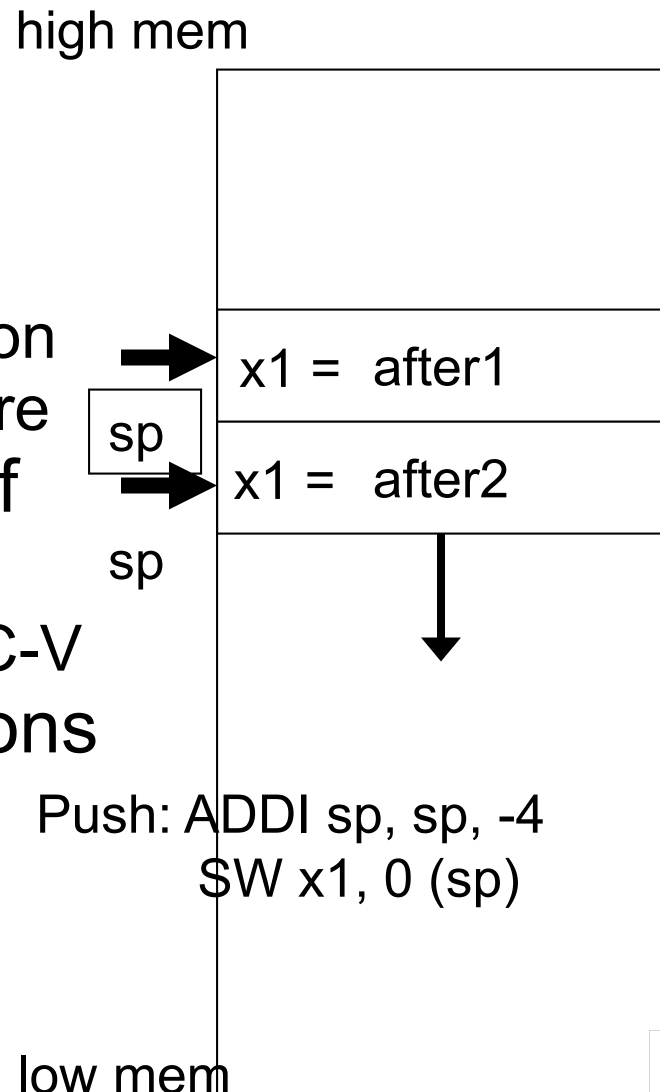
- the return address for that invocation
- the local variables for that procedure

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



Need a “Call Stack”

Call stack

- contains activation records (aka stack frames)

Each activation record contains

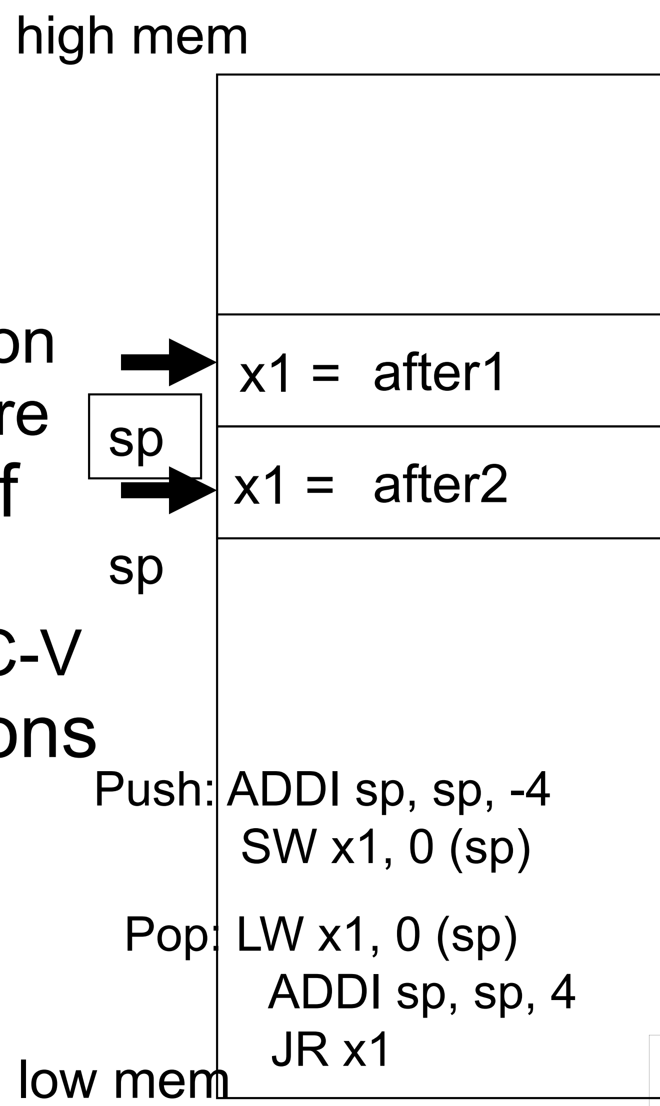
- the return address for that invocation
- the local variables for that procedure

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

- dedicated register (x2) on the RISC-V

Manipulated by push/pop operations

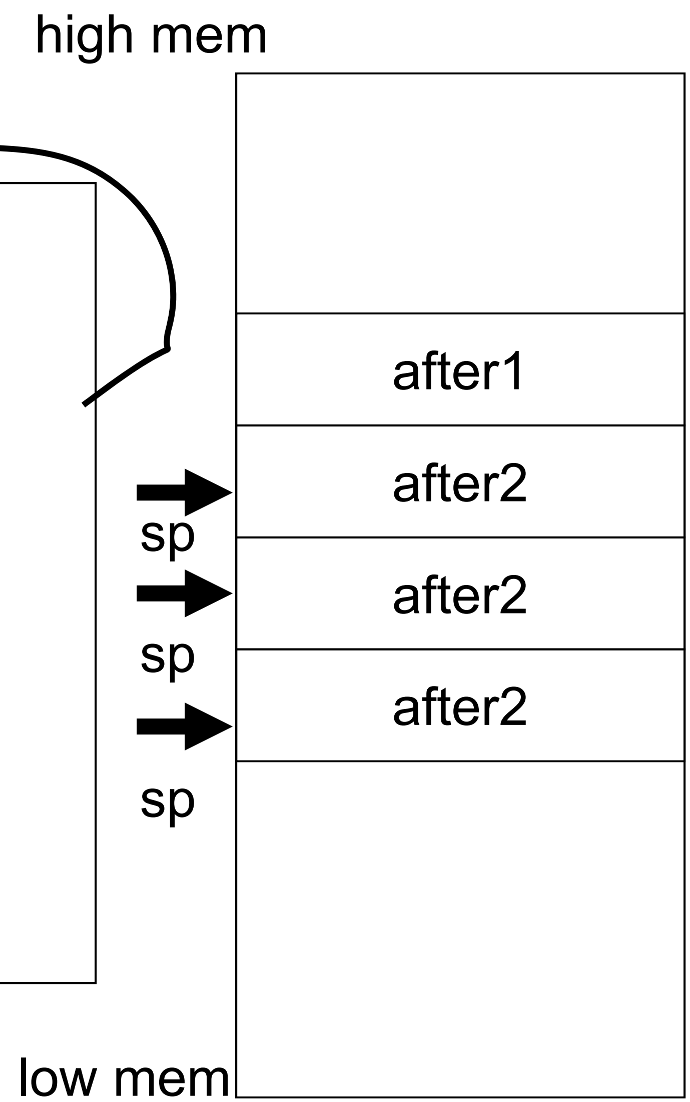
- push: move sp down, store
- pop: load, move sp up



Need a "Call Stack"

```
main:
    jal myfn
after1:
    add x1,x2,x3
```

```
myfn:
        addi sp,sp,-4
        sw x1, 0(sp)
        if (test)
        jal myfn
after2:
        lw x1, 0(sp)
        addi sp,sp,4
        jr x1
```

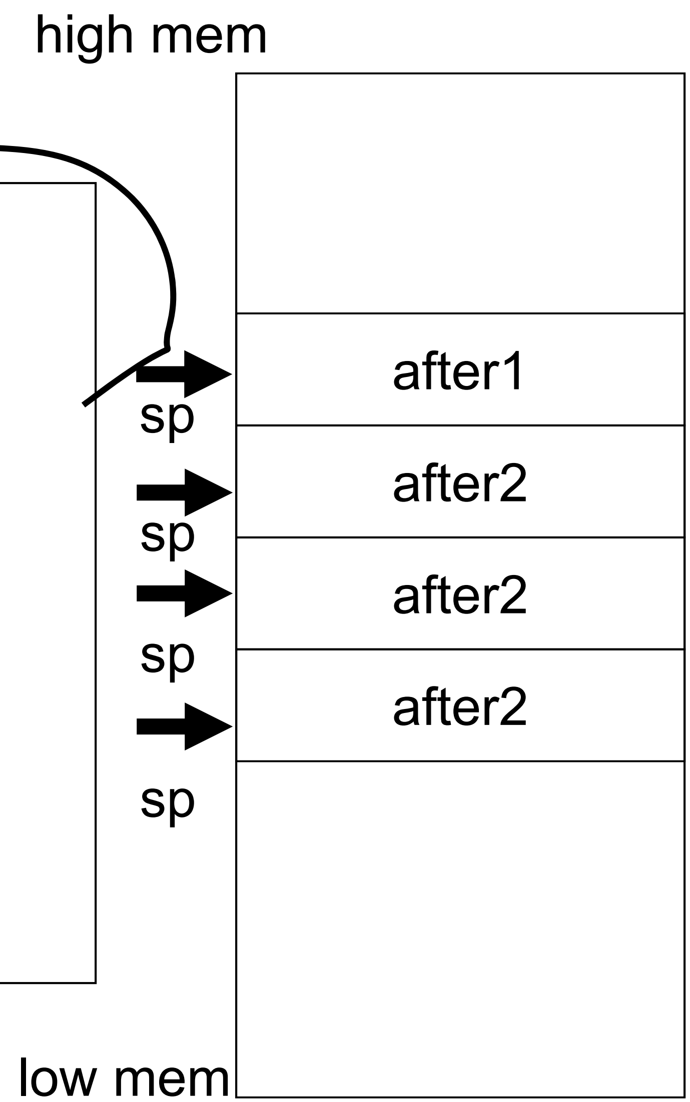


Stack used to save and restore contents of x1

Need a “Call Stack”

```
main:
    jal myfn
after1:
    add x1,x2,x3
```

```
myfn:
        addi sp,sp,-4
        sw x1, 0(sp)
        if (test)
        jal myfn
after2:
        lw x1, 0(sp)
        addi sp,sp,4
        jr x1
```



Stack used to save and restore contents of x1

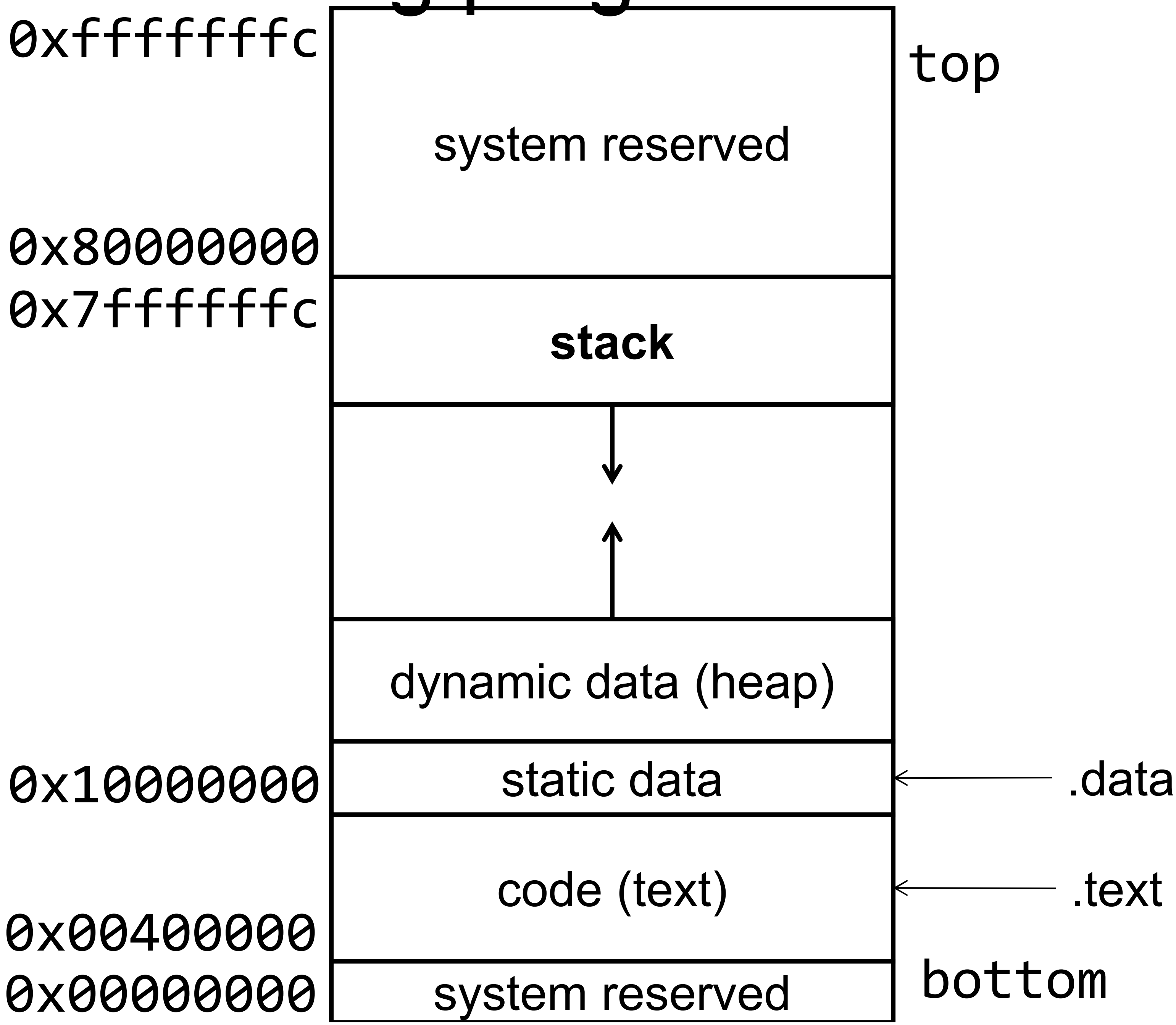
Stack Growth

(Call) Stacks start at a high address in memory

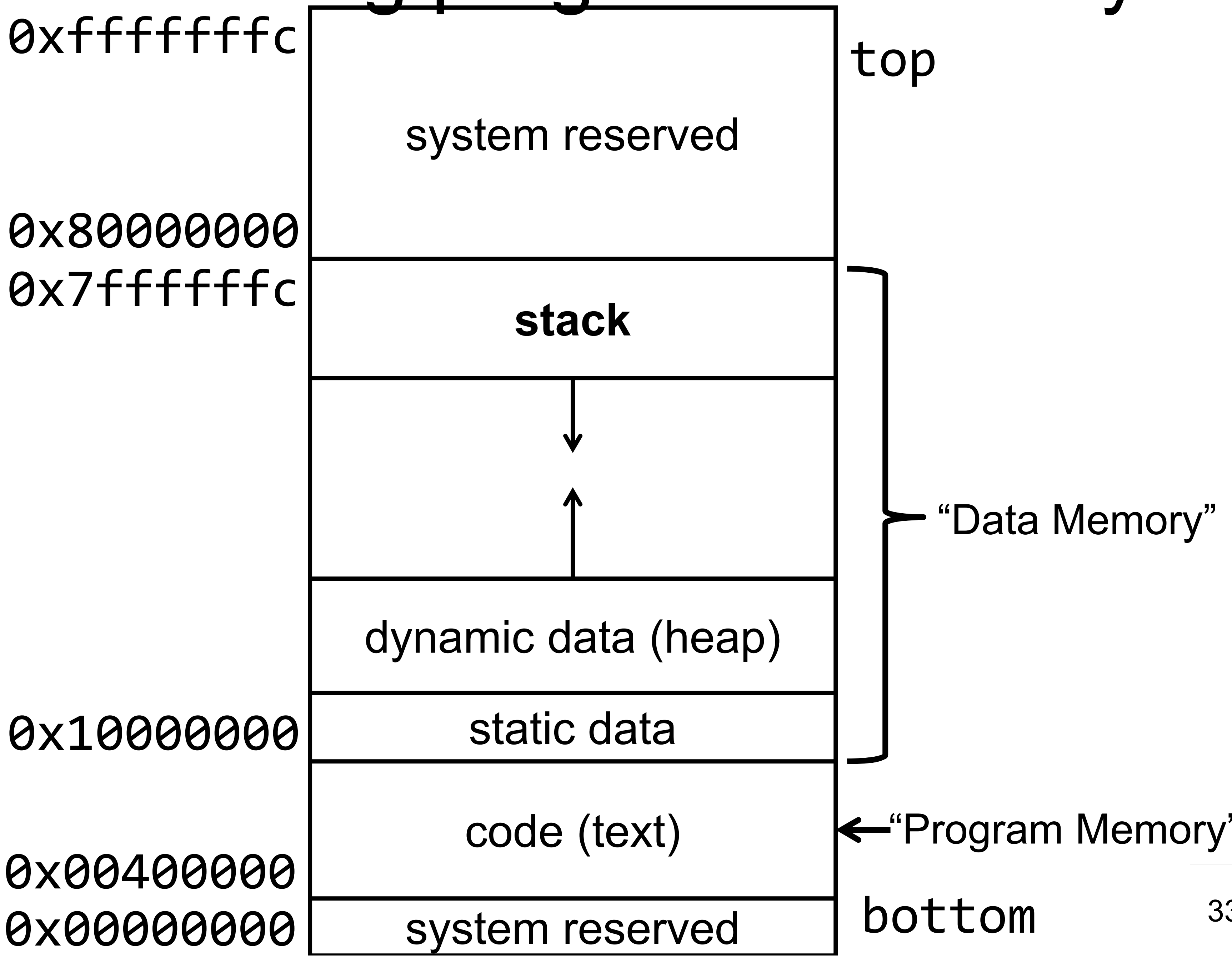
Stacks grow down as frames are pushed on

- Note: data region starts at a low address and grows up
- The growth potential of stacks and data region are not artificially limited

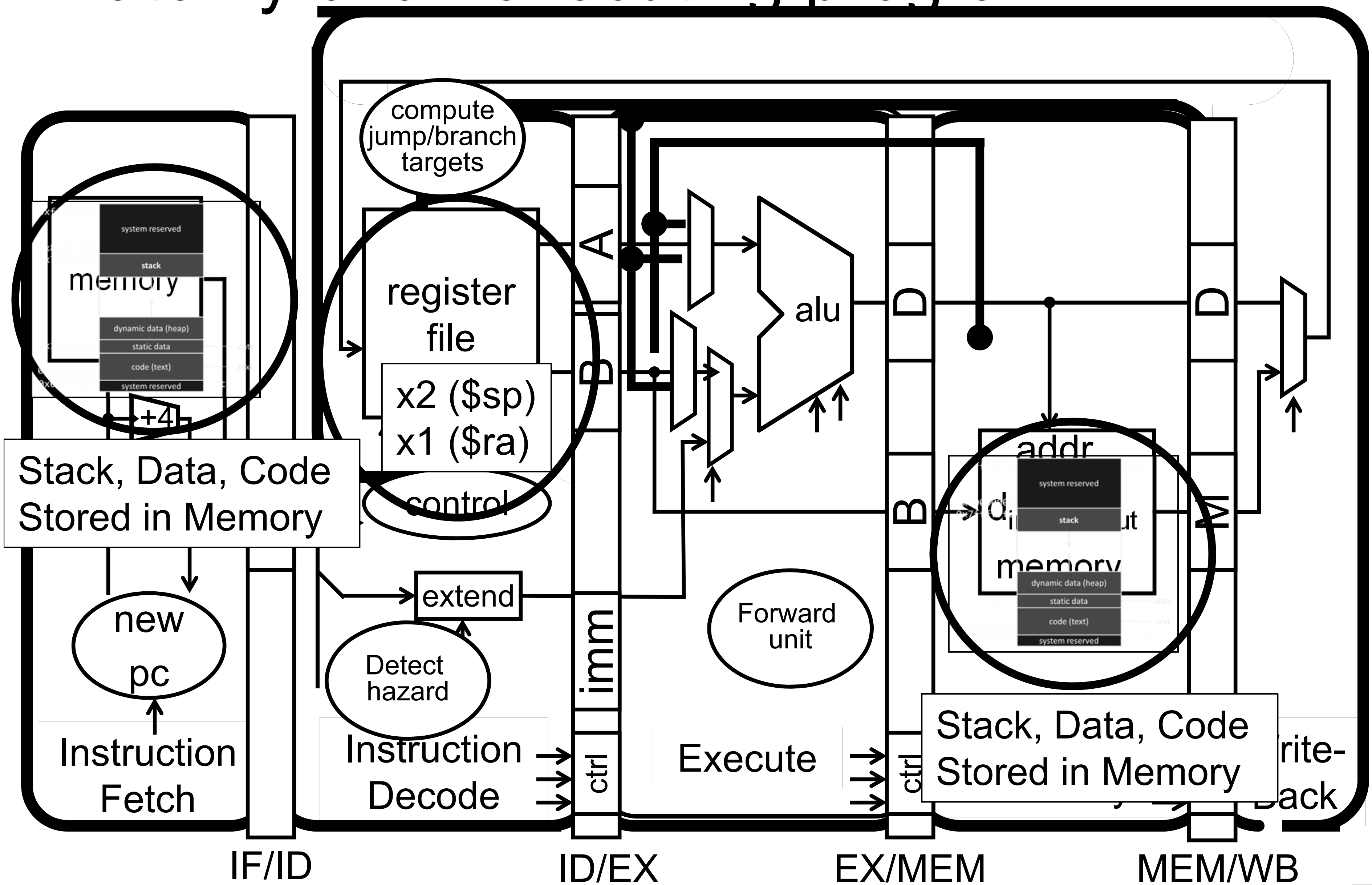
An executing program in memory



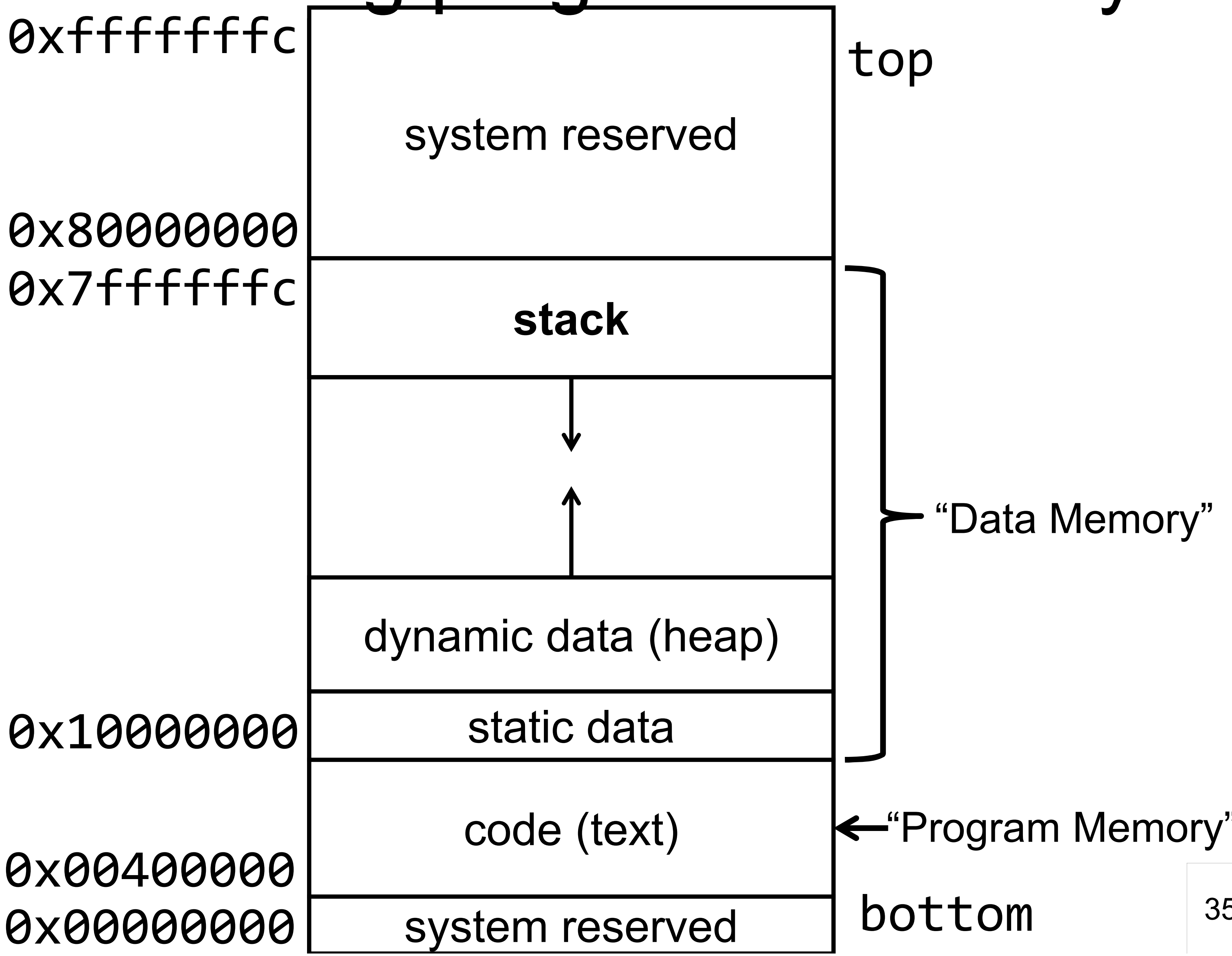
An executing program in memory



Anatomy of an executing program



An executing program in memory



Return Address lives in Stack Frame

Stack Manipulated by push/pop operations

Context: after 2nd JAL to myfn (from myfn)

PUSH: ADDI sp, sp, -20 // move sp down

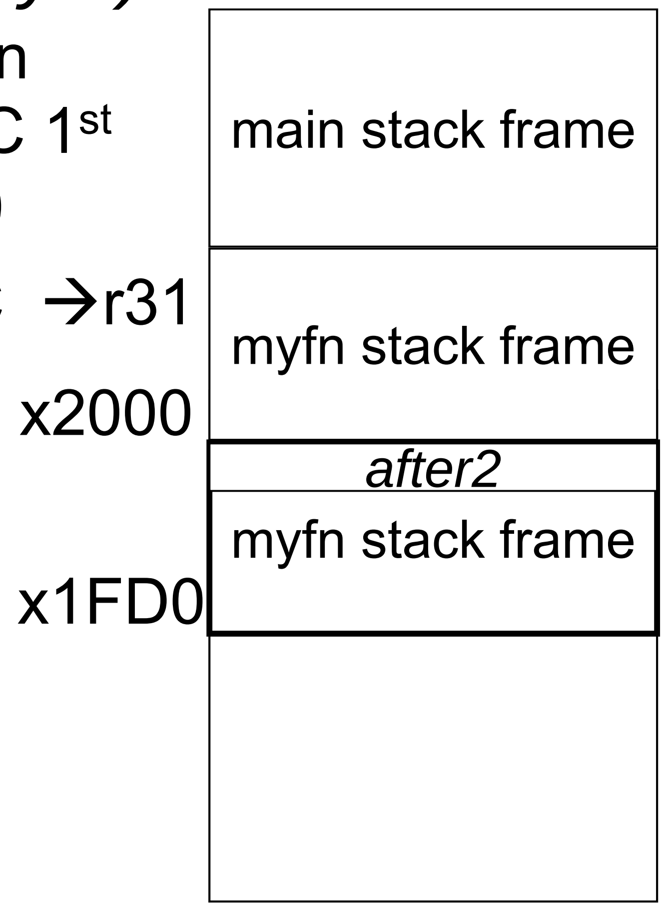
SW x1, 16(sp) // store retn PC 1st

Context: 2nd myfn is done (x1 == ???)

POP: LW x1, 16(sp) // restore retn PC →r31

ADDI sp, sp, 20 // move sp up

JR x1 // return



r29 x1FD0

r31 *XXXX*

For now: Assume each frame = x20 bytes
(just to make this example concrete)

The Stack

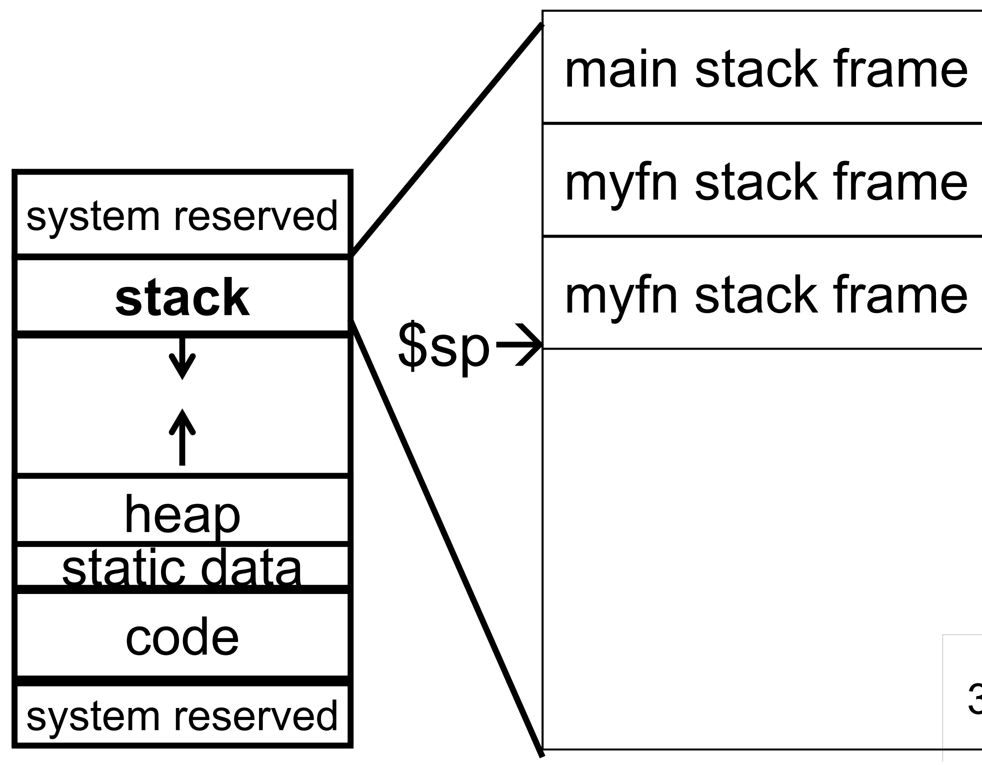
Stack contains stack frames (aka "activation records")

- 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)

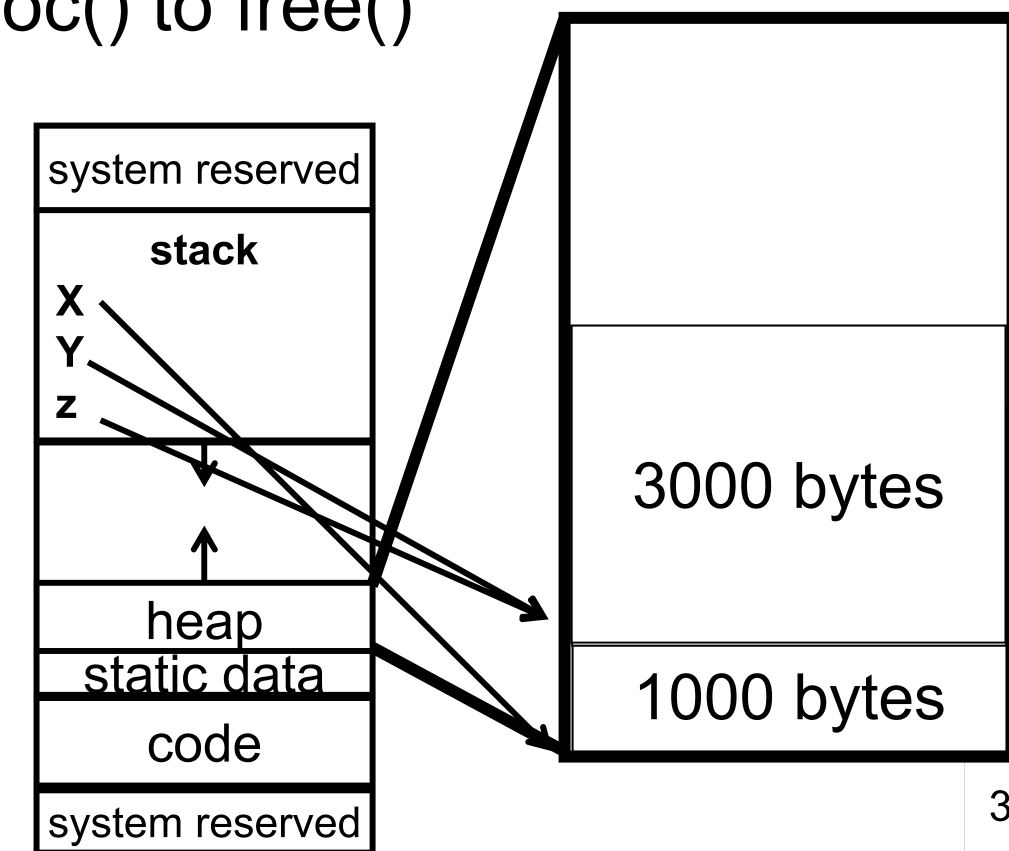
```
int main(…) {
     …
     myfn(x);
}
int myfn(int n) {
     …
     myfn();
}
```



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);
}
```



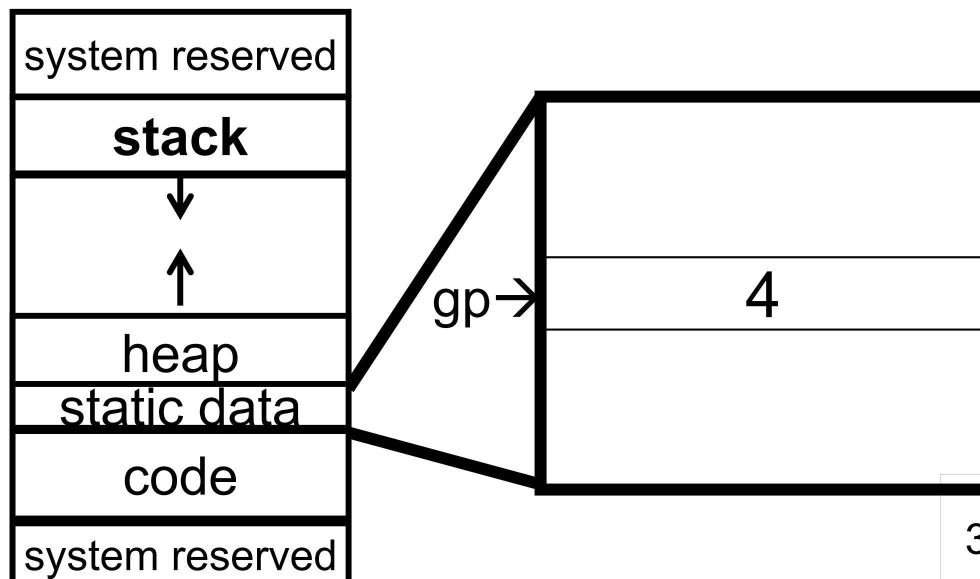
Data Segment

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(...) {
      ...
}
```



Globals and Locals

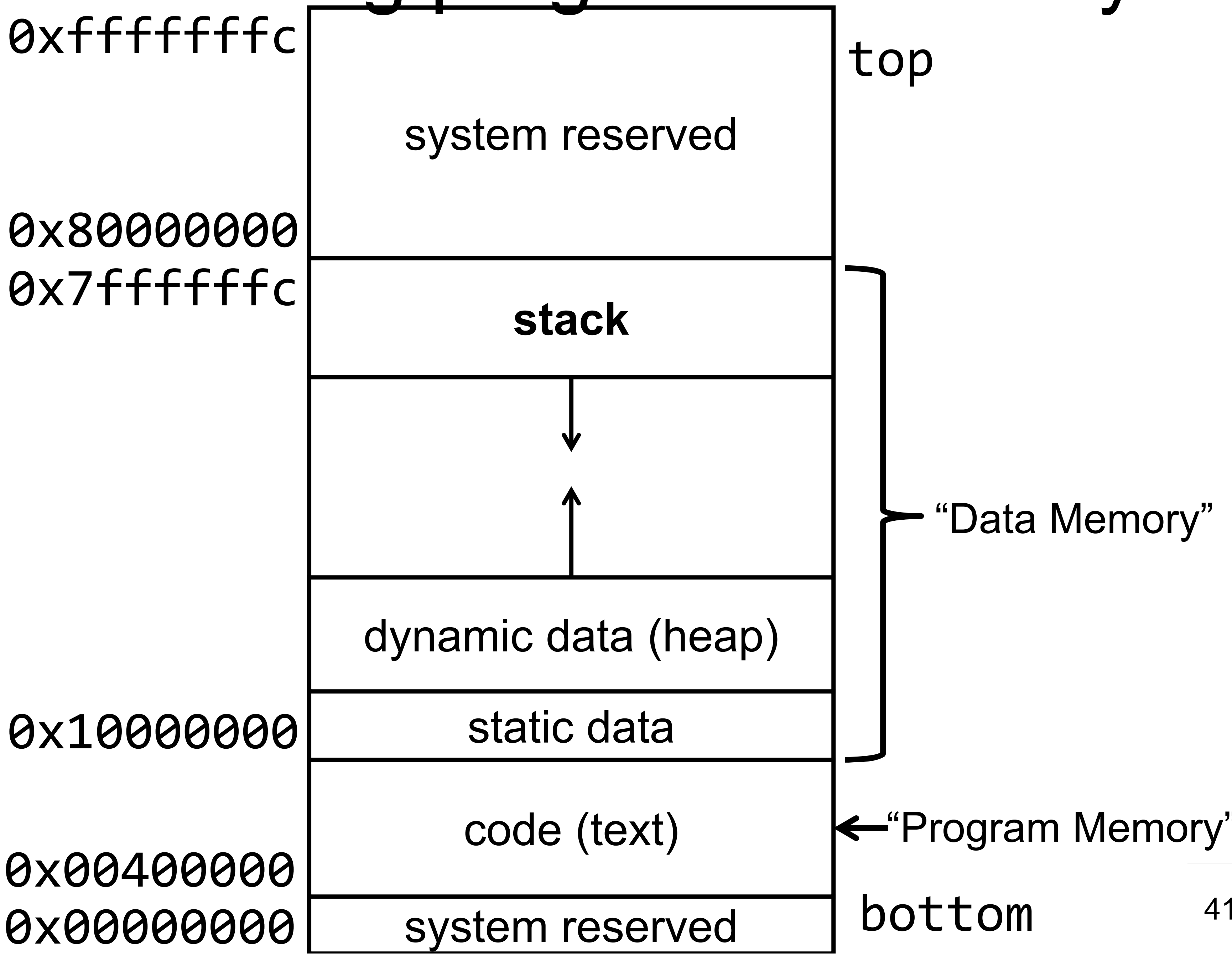
Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

```
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

An executing program in memory



Globals and Locals

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);
}
```

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).

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

Next Goal

Need consistent way of passing arguments and getting the result of a subroutine invocation

Arguments & Return Values

Need consistent way of passing arguments and getting the result of a subroutine invocation

Given a procedure signature, need to know where arguments should be placed

$a0, $a1

- `int min(int a, int b);`
- `int subf(int a, int b, int c, int d, int e, int f, int g, int h, int i);`

  stack?
- `int isalpha(char c);`
- `int treesort(struct Tree *root);`

  $a0
- `struct Node *createNode();`
- `struct Node mynode();`

  $a0, $a1

  $a0

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:

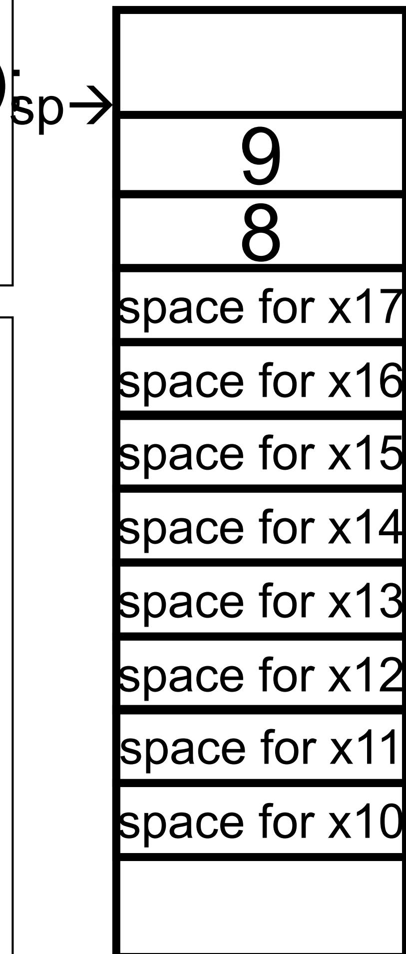
- args passed in \$a0, \$a1, …, \$a7
- return value (if any) in \$a0, \$a1
- stack frame at \$sp
  - contains \$ra (clobbered on JAL to sub-functions

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

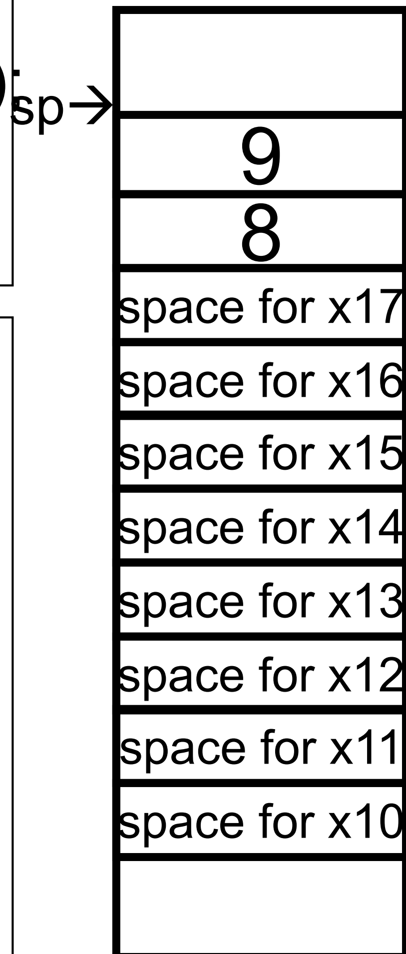
Args passed in child's stack frame

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

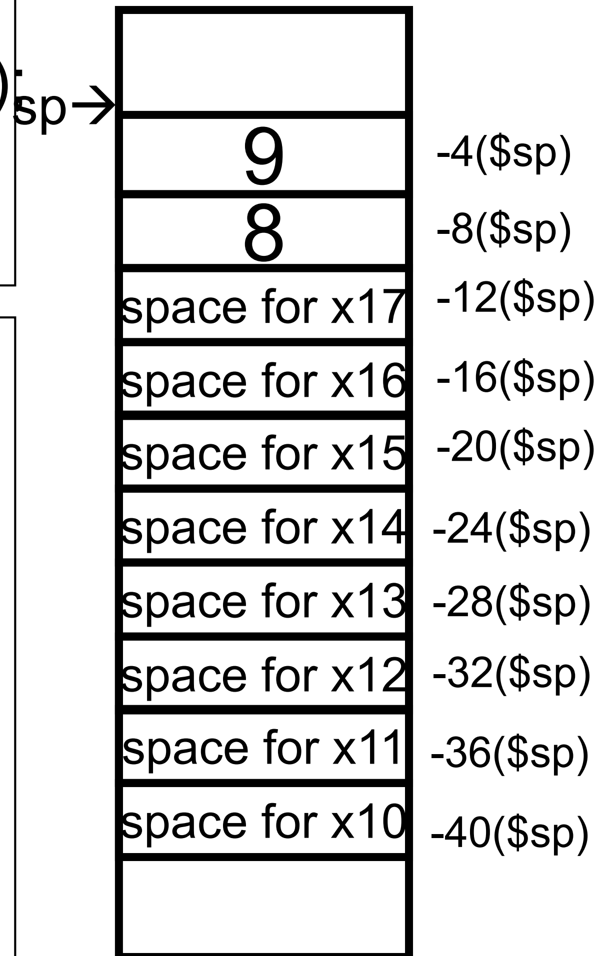
Args passed in child’s stack frame

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!)

iClicker Question

Which is a true statement about the arguments to the function

```
void sub(int a, int b, int c, int d, int e, int f,
int g, int h, int i);
```

A. Arguments `a-i` are all passed in registers.
B. Arguments `a-i` are all stored on the stack.
C. Only i is stored on the stack, but space is allocated for all 9 arguments.
D. Only a-h are stored on the stack, but space is allocated for all 9 arguments.

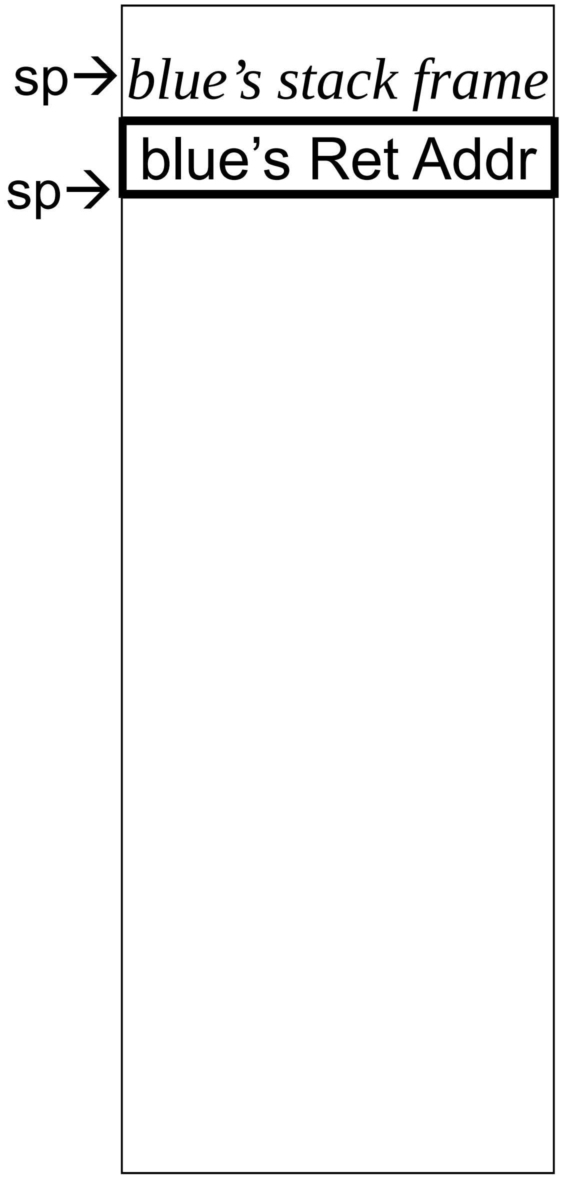
iClicker Question

Which is a true statement about the arguments to the function

```
void sub(int a, int b, int c, int d, int e, int f,
int g, int h, int i);
```

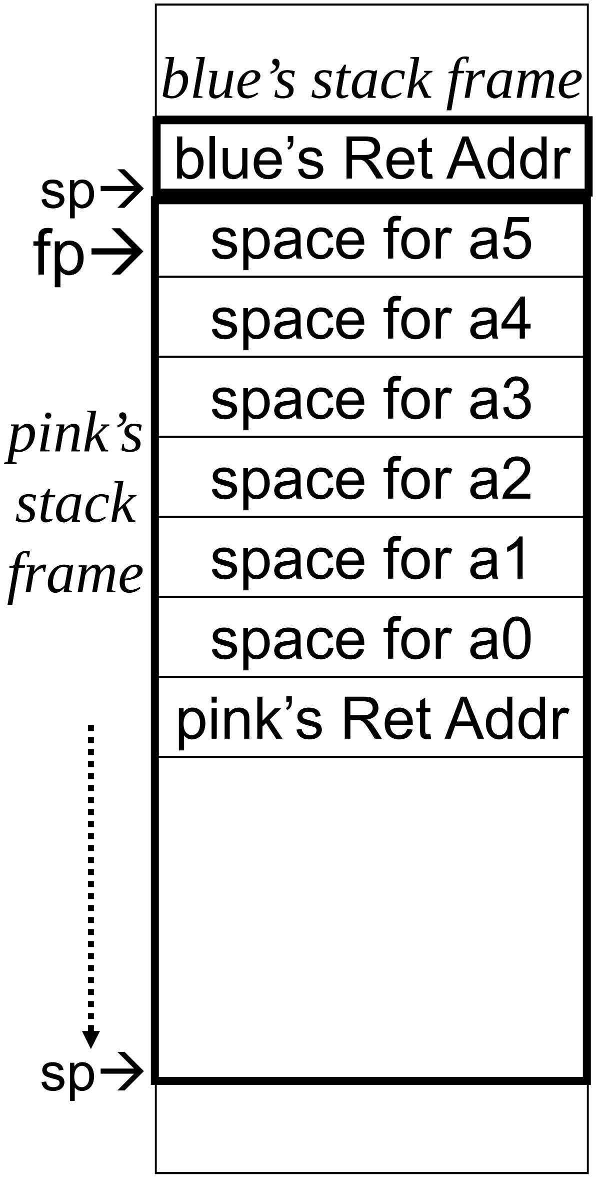
A. Arguments `a-i` are all passed in registers.
B. Arguments `a-i` are all stored on the stack.
C. Only i is stored on the stack, but space is allocated for all 9 arguments.
D. Only a-h are stored on the stack, but space is allocated for all 9 arguments.

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) {
    ...
}
```

Conventions so far

- first eight arg words passed in \$a0, \$a1, …, \$a7
- Space for args in child's stack frame
- return value (if any) in \$a0, \$a1
- stack frame (\$fp/\$s0 to \$sp) contains:
  - \$ra (clobbered on JAL to sub-functions)
  - space for 8 arguments to Callees
  - arguments 9+ to Callees

RISCV Register Conventions so far:

x0	zero	zero
x1	ra	Return address
x2	sp	Stack pointer
x3		
x4		
x5	t0	Temporary registers
x6	t1	
x7	t2	
x8	s0/fp	Saved register or framepointer
x9	s1	Saved register
x10	a0	Function args or return values
x11	a1	
x12	a2	Function args
x13	a3	
x14	a4	

x16		
x17		
x18	s2	Saved registers
x19		
x20		
x21		
x22		
x23		
x24		
x25		
x26		
x27		
x28	t3	Temporary registers
x29		
x30		
x31		

C & RISCV: the fine print

C allows passing whole structs

a0, a1 a2, a3

- `int dist(struct Point p1, struct Point p2);`
- Treated as collection of consecutive 32-bit arguments
  - Registers for first 4 words, stack for rest
- Better: `int dist(struct Point *p1, struct Point *p2);`

a0 a1

Where are the arguments to:

a0, a1

```
void sub(int a, int b, int c, int d, int e,
int f, int g, int h, int i);
```

stack

```
void isalpha(char c);
void treesort(struct Tree *root);
```

a0

Where are the return values from:

```
struct Node *createNode();
struct Node mynode();
```

a0

a0, a1

Many combinations of char, short, int, void *, struct, *etc.*

- RISCV treats char, short, int and void * identically

Globals and Locals

Global variables are allocated in the "data" region of the program

- Exist for all time, accessible to all routines

Local variables are allocated within the stack frame

- Exist solely for the duration of the stack frame

Dangling pointers are pointers into a destroyed stack frame

- C lets you create these, Java does not
- int *foo() { int a; return &a; }

Return the address of a,
But a is stored on stack, so will be removed
when call returns and point will be invalid

Global and Locals

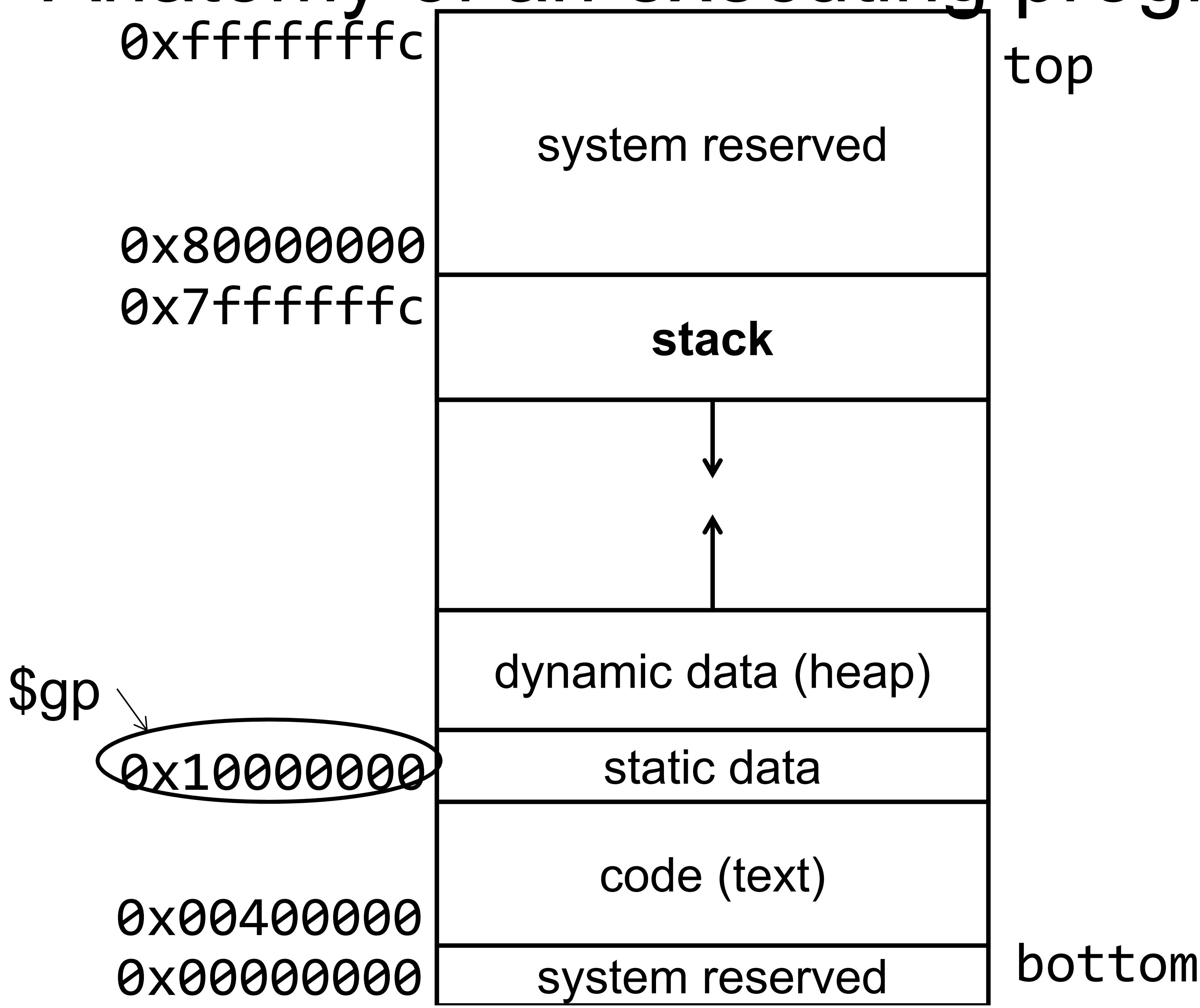
How does a function load global data?

- 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



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

Conventions so far

- 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/s0 to sp) contains:
  - ra (clobbered on JALs)
  - space for 8 arguments
  - arguments 9+
- global data accessed via gp

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

Next Goal

What convention should we use to share use of registers across procedure calls?

Register Management

Functions:

- Are compiled in isolation
- Make use of general purpose registers
- Call other functions in the middle of their execution
  - These functions also use general purpose registers!
  - No way to coordinate between caller & callee

→ Need a convention for register management

Register Usage

Suppose a routine would like to store a value in a register

Two options: *callee-save* and *caller-save*

Callee-save:

- Assume that one of the callers is already using that register to hold a value of interest
- Save the previous contents of the register on procedure entry, restore just before procedure return
- E.g. $ra, $fp/$s0, $s1-$s11, $gp, $tp
- Also, $sp

Caller-save:

- Assume that a caller can clobber any one of the registers
- Save the previous contents of the register before proc call
- Restore after the call
- E.g. $a0-a7, $t0-$t6

RISCV calling convention supports both

Caller-saved

Registers that the caller cares about: t0… t9

About to call a function?

- Need value in a t-register *after* function returns?
  - → save it to the stack before fn call
  - → restore it from the stack after fn returns
- Don't need value? → do nothing

<u>Suppose:</u>

t0 holds x

t1 holds y

t2 holds z

Where do we save and restore?

Functions

- Can freely use these registers
- Must assume that their contents are destroyed by other functions

```
void myfn(int a) {
  int x = 10;
  int y = max(x, a);
  int z = some_fn(y);
  return (z + y);
}
```

Callee-saved

Registers a function intends to use: s0… s9

About to use an s-register? You **MUST**:

- Save the current value on the stack ***before*** using
- Restore the old value from the stack before fn returns

Suppose:

s1 holds x

s2 holds y

s3 holds z

Where do we save and restore?

Functions

- Must save these registers before using them
- May assume that their contents are preserved even across fn calls

```
void myfn(int a) {
  int x = 10;
  int y = max(x, a);
  int z = some_fn(y);
  return (z + y);
}
```

Caller-Saved Registers in Practice

```
main:
  …
  [use x5 & x6]
  …
  addi x2, x2, -8
  sw x6, 4(x2)
  sw x5, 0(x2)
  jal myfn
  lw x6, 4(x2)
  lw x5, 0(x2)
  addi x2, x2, 8
  …
  [use x5 & x6]
```

Assume the registers are free for the taking, use with no overhead

Since subroutines will do the same, must protect values needed later:

- Save before fn call
- Restore after fn call

Notice: Good registers to use if you don't call too many functions or if the values don't matter later on anyway.

Caller-Saved Registers in Practice

```
main:
  …
  [use $t0 & $t1]
  …
  addi $sp, $sp,-8
  sw $t1, 4($sp)
  sw $t0, 0($sp)
  jal myfn
  lw $t1, 4($sp)
  lw $t0, 0($sp)
  addi $sp, $sp, 8
  …
  [use $t0 & $t1]
```

Assume the registers are free for the taking, use with no overhead

Since subroutines will do the same, must protect values needed later:

- Save before fn call
- Restore after fn call

Notice: Good registers to use if you don't call too many functions or if the values don't matter later on anyway.

Callee-Saved Registers in Practice

```
main:
 addi x2, x2, -16
 sw x1, 12(x2)
 sw x8, 8(x2)
 sw x18, 4(x2)
 sw x9, 0(x2)
 addi  x8, x2, 12
    …
 [use x9 and x18]
    …
 lw x1, 12(x2)
 lw x8, 8(x2)($sp)
 lw x18, 4(x2)
 lw x9, 0(x2)
 addi x2, x2, 16
 jr x1
```

Assume caller is using the registers
- Save on entry
- Restore on exit

Notice: Good registers to use if you make a lot of function calls and need values that are preserved across all of them. Also, good if caller is actually using the registers, otherwise the save and restores are wasted. But hard to know this.

Callee-Saved Registers in Practice

```
main:
  addi $sp, $sp, -16
  sw $ra, 12($sp)
  sw $fp, 8($sp)
  sw $s2, 4($sp)
  sw $s1, 0($sp)
  addi  $fp, $sp, 12
     …
  [use $s1 and $s2]
     …
  lw $ra, 12($sp)
  lw $fp, 8($sp)($sp)
  lw $s2, 4($sp)
  lw $s1, 0($sp)
  addi $sp, $sp, 16
  jr $ra
```

Assume caller is using the registers
- Save on entry
- Restore on exit

Notice: Good registers to use if you make a lot of function calls and need values that are preserved across all of them. Also, good if caller is actually using the registers, otherwise the save and restores are wasted. But hard to know this.

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to put a in a:

(A) Caller-saved register (t)
(B) Callee-saved register (s)
(C) Depends on where we put the other variables in this fn
(D) Both are equally valid

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to put a in a:

(A) Caller-saved register (t)
(B) Callee-saved register (s)
(C) Depends on where we put the other variables in this fn
(D) Both are equally valid

Repeat but assume that foo is recursive (bar/baz → foo)

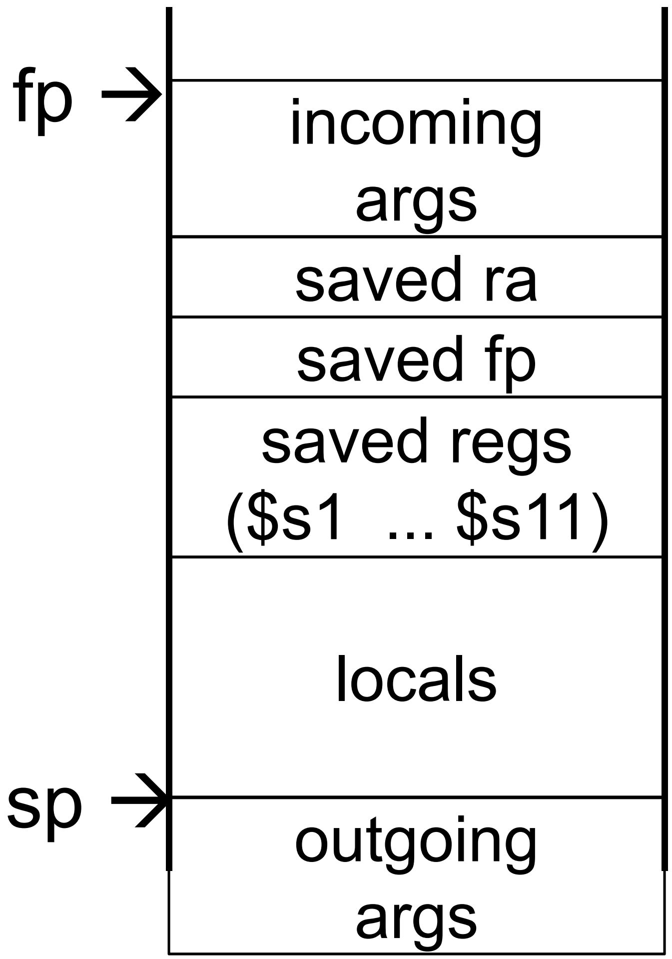
Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to put b in a:

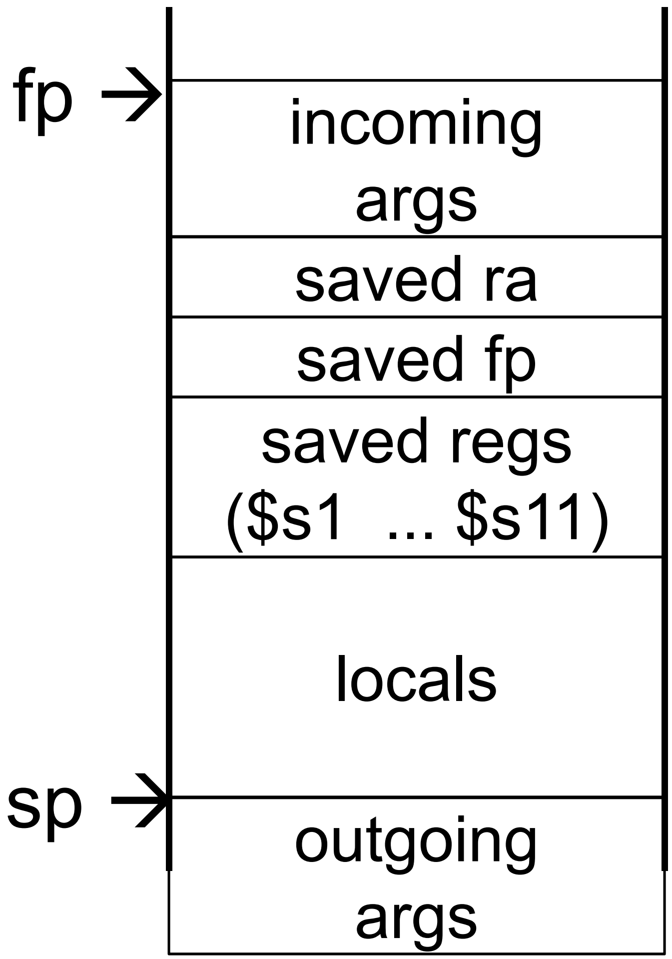
(A) Caller-saved register (t)
(B) Callee-saved register (s)
(C) Depends on where we put the other variables in this fn
(D) Both are equally valid

Frame Layout on Stack



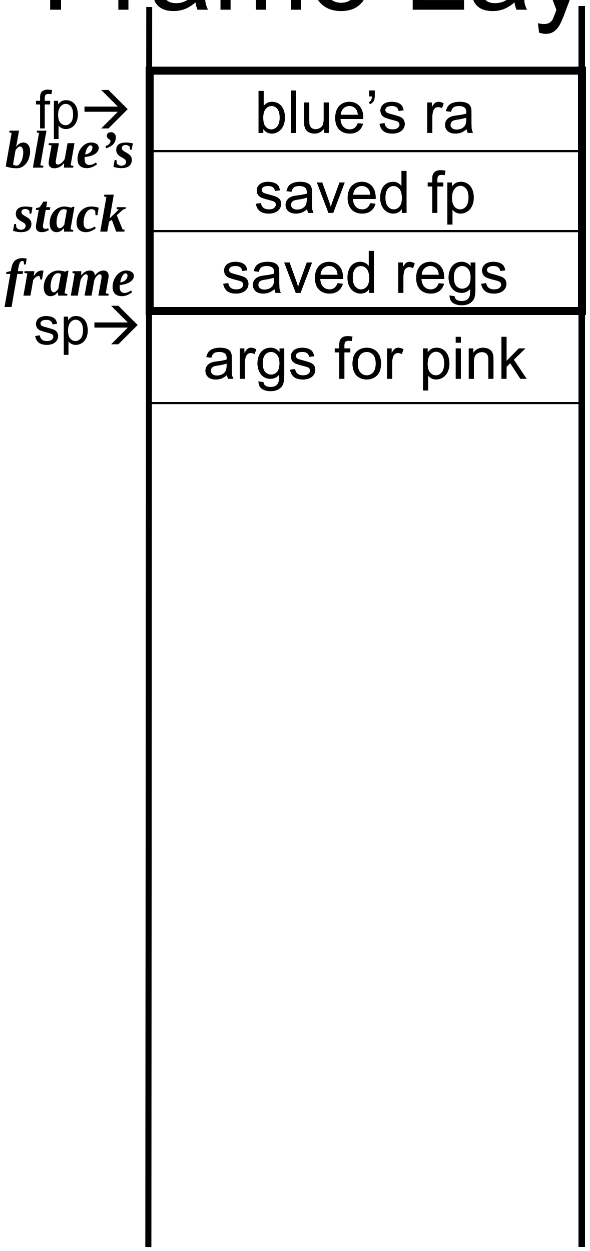
Assume a function uses two callee-save registers.
How do we allocate a stack frame?
How large is the stack frame?
What should be stored in the stack frame?
Where should everything be stored?

Frame Layout on Stack



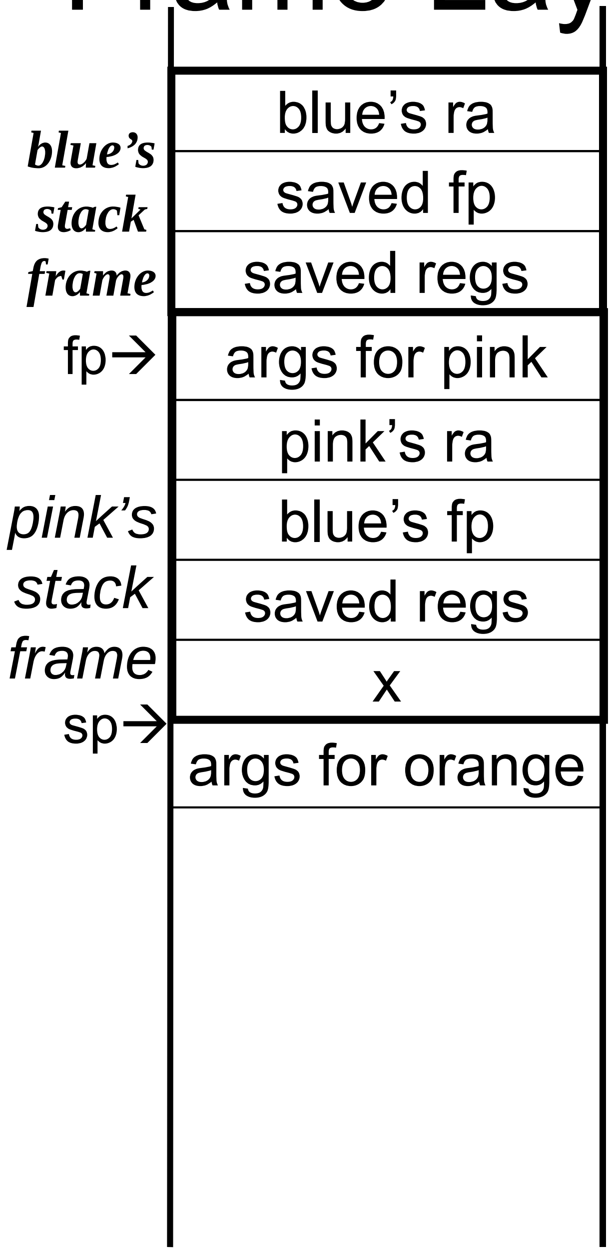

```
ADDI sp, sp, -16   # allocate frame
SW ra, 12(sp)      # save ra
SW fp, 8(sp)       # save old fp
SW s2, 4(sp)       # save ...
SW s1, 0(sp)       # save ...
ADDI fp, sp, 12    # set new frame ptr
…                  ...
BODY
…                  ...
LW s1, 0(sp)       # restore …
LW s2, 4(sp)       # restore …
LW fp, 8(sp)       # restore old fp
LW ra, 12(sp)      # restore ra
ADDI sp, sp, 16    # dealloc frame
JR ra
```

Frame Layout on Stack



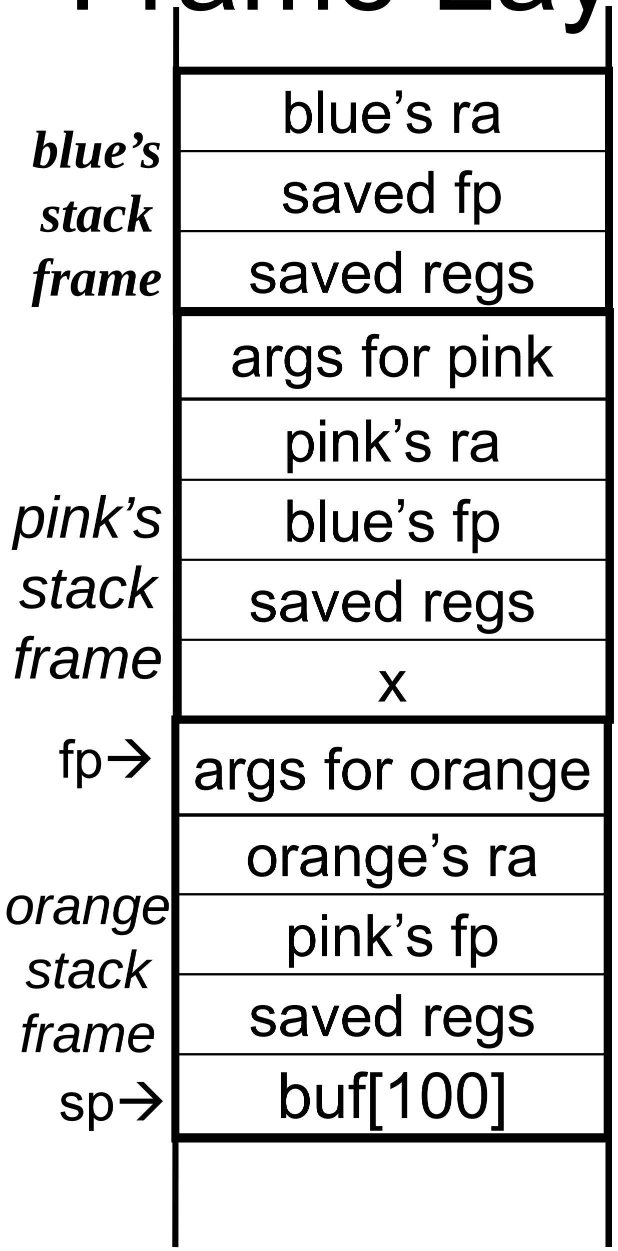

```
blue() {
   pink(0,1,2,3,4,5);
}
```

Frame Layout on Stack




```
blue() {
   pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    int x;
    orange(10,11,12,13,14);
}
```

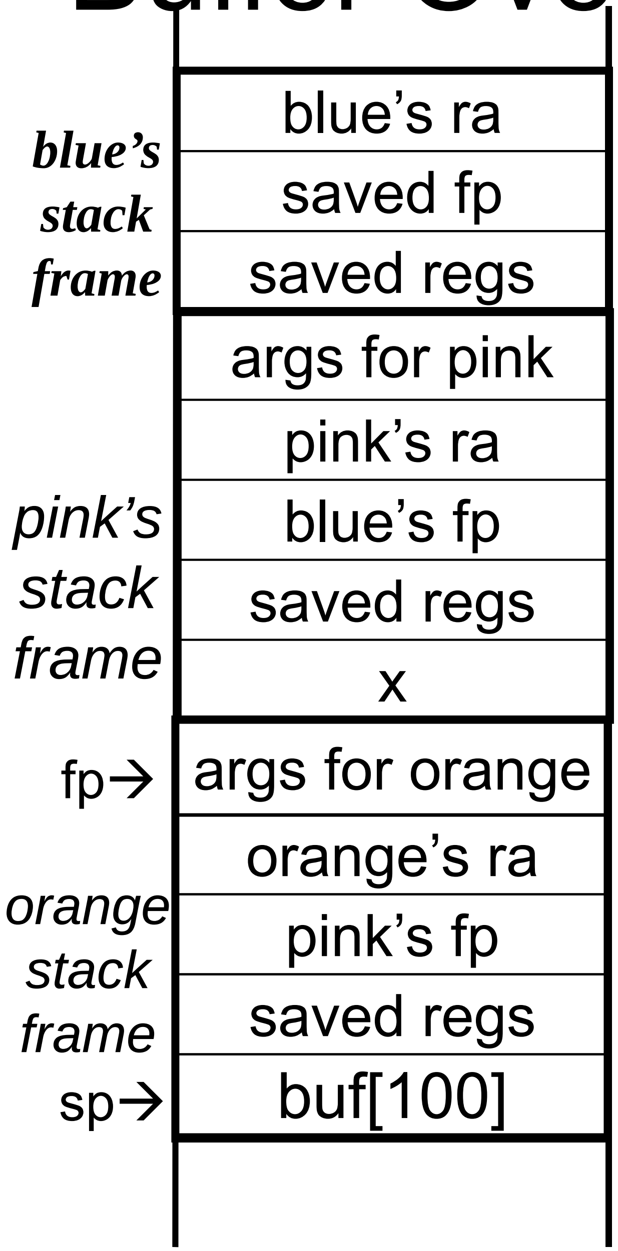
Frame Layout on Stack




```
blue() {
   pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
     int x;
     orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
     char buf[100];
     gets(buf);      // no bounds check!
}
```

What happens if more than 100 bytes is written to buf?

Buffer Overflow




```
blue() {
   pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
     int x;
     orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
      char buf[100];
      gets(buf);       // no bounds check!
}
```

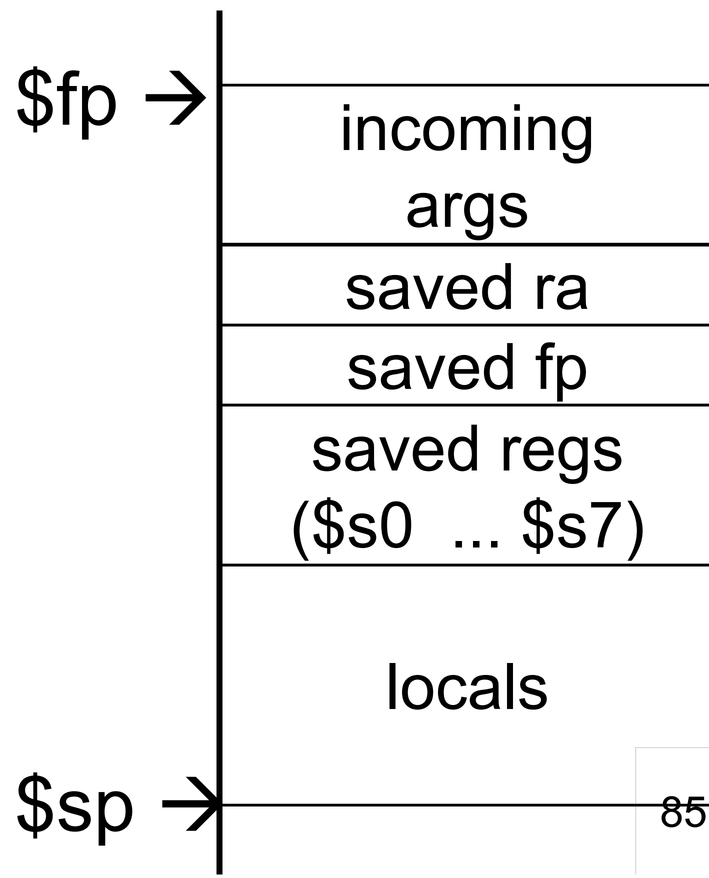
What happens if more than 100 bytes is written to buf?

RISCV Register Recap

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



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;
}
```

<u>Correct Order:</u>

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

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;
}
```

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

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;
}
```

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
```

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;
}
```

$fp →	space for a1
	space for a0
	saved ra
	saved fp
	saved regs (s1 ... s11)
	locals (t0)
$sp →	outgoing args space for a0 – a7 and 9th arg

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

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;
}
```

$fp →	24	space incoming for a1
	20	space incoming for a0
	16	saved ra
	12	saved fp
	8	saved reg s2
	4	saved reg s1
	0	local t0
$sp →	-4	outgoing 9^{th} arg
	-8	space for a7
	-12	space for a6
	…	…
	-28	space for a1
	-36	space for a0

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

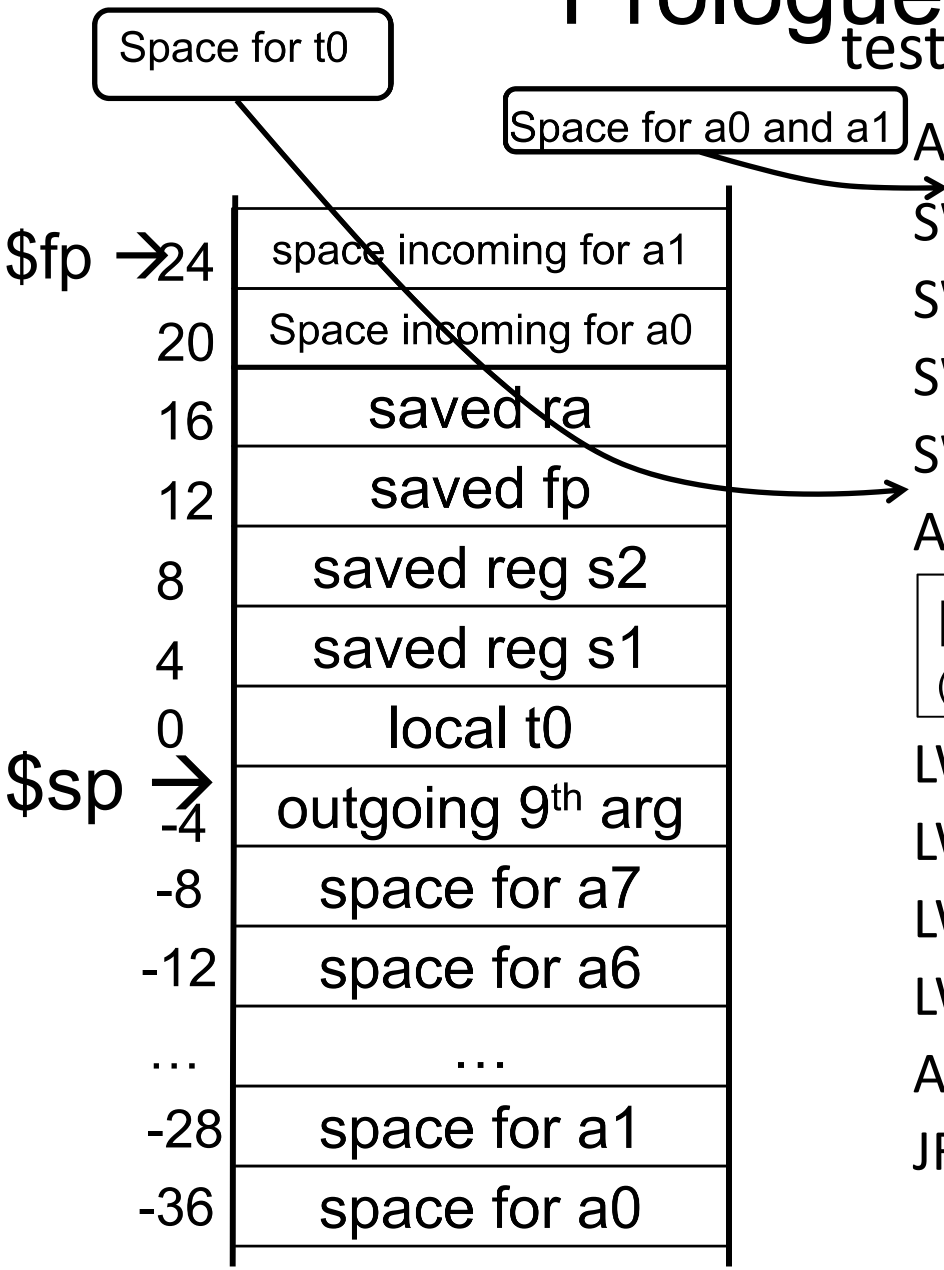
Activity #2: Calling Convention Example: Prologue, Epilogue

test:

$fp →	24	space incoming for a1
	20	Space incoming for a0
	16	saved ra
	12	saved fp
	8	saved reg s2
	4	saved reg s1
	0	local t0
$sp →	-4	outgoing 9th arg
	-8	space for a7
	-12	space for a6
	…	…
	-28	space for a1
	-36	space for a0

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

Activity #2: Calling Convention Example: Prologue, Epilogue



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)

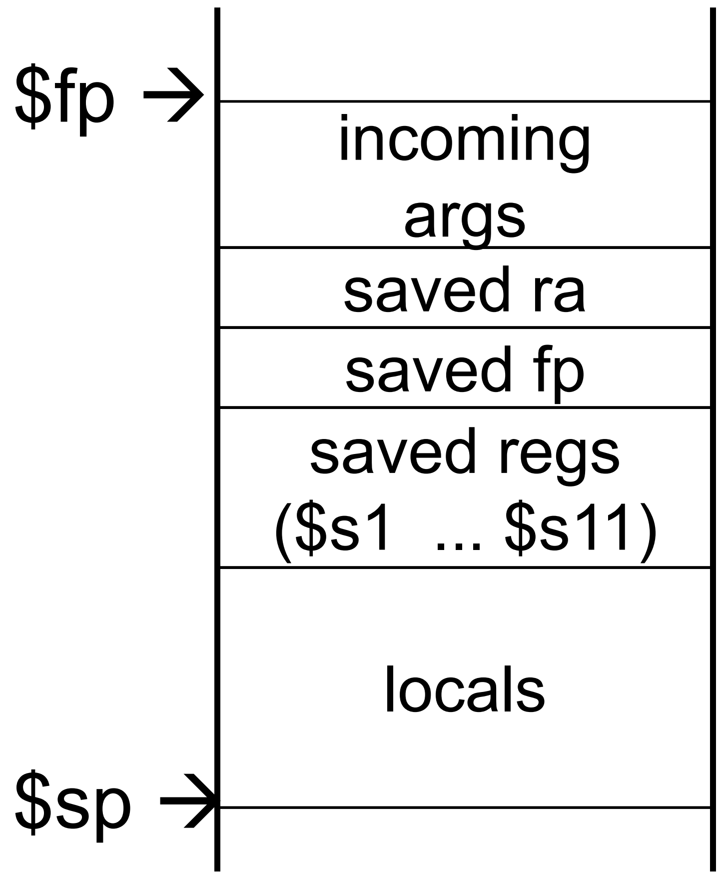
```
LW s1, sp, 4         # restore …
LW s2, sp, 8         # restore …
LW fp, sp, 12        # restore old $fp
LW ra, sp, 16        # restore $ra
ADDI sp, sp, 28      # dealloc frame
JR ra
```

Next Goal

Can we optimize the assembly code at all?

Minimum stack size for a standard function?

Minimum stack size for a standard function?



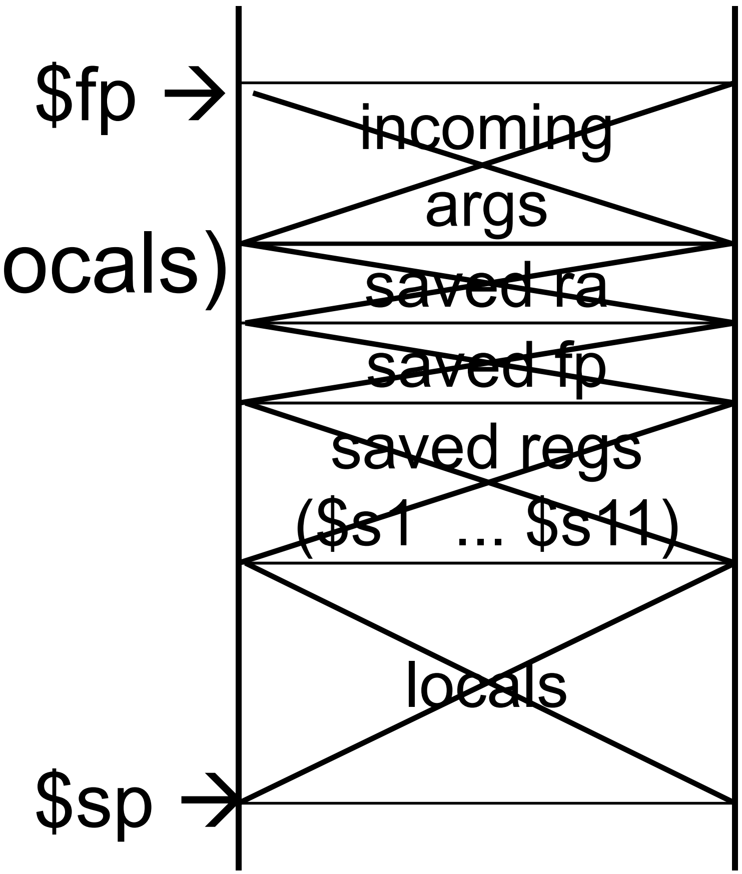
Minimum stack size for a standard function?

Leaf function does not invoke any other functions

int f(int x, int y) { return (x+y); }

Optimizations?

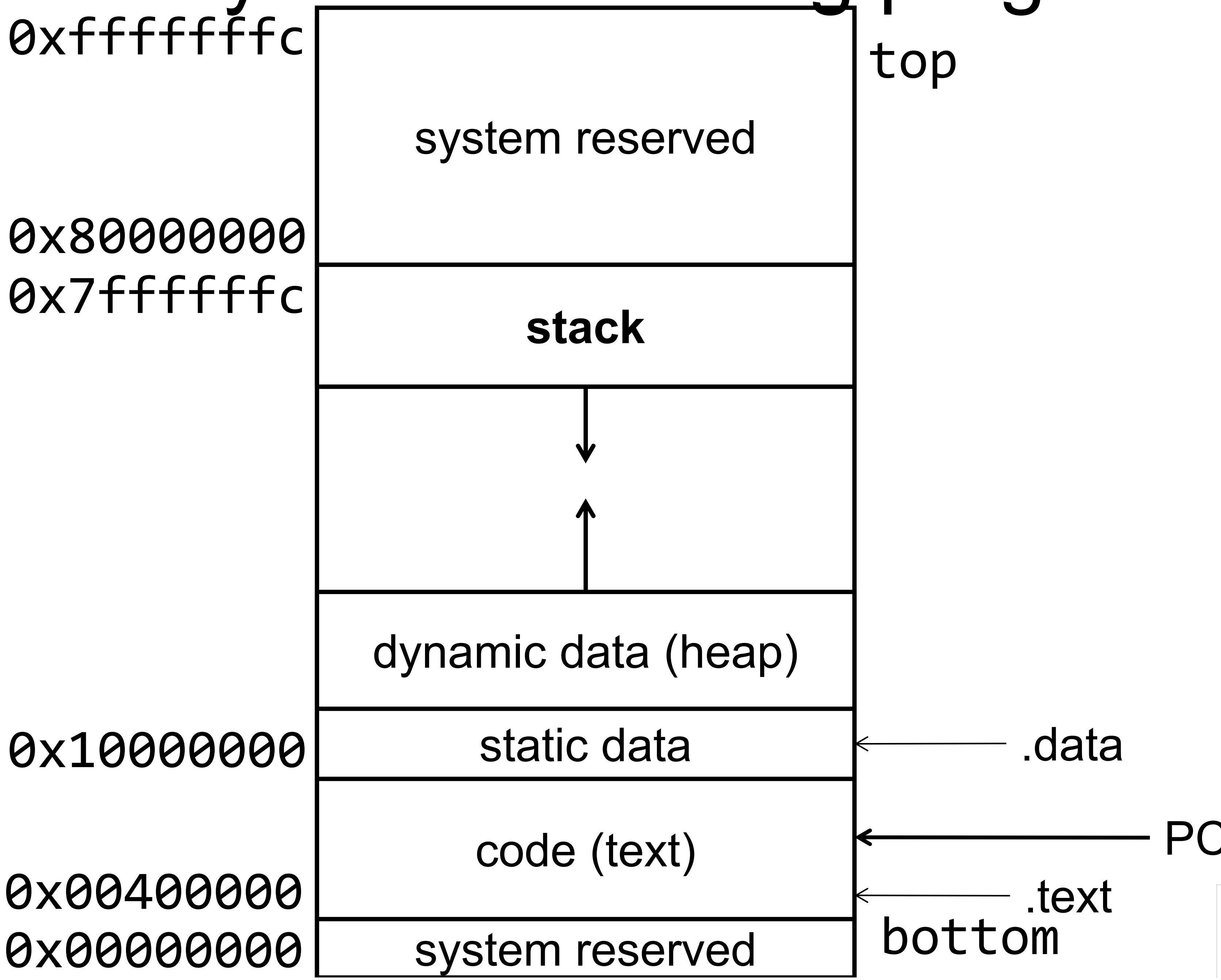
- No saved regs (or locals)
- No incoming args
- Don't push \$ra
- No frame at all?
    - Maybe.



Next Goal

Given a running program (a process), how do we know what is going on (what function is executing, what arguments were passed to where, where is the stack and current stack frame, where is the code and data, etc)?

Anatomy of an executing program



Activity #4: Debugging

init():	0x400000
printf(s, …):	0x4002B4
vnorm(a,b):	0x40107C
main(a,b):	0x4010A0
pi:	0x10000000
str1:	0x10000004

CPU:
$pc=0x004003C0
$sp=0x7FFFFFAC
$ra=0x00401090

	0x00000000
	0x0040010c
	0x7FFFFFF4
	0x00000000
	0x00000000
	0x00000000
	0x00000000
	0x004010c4
	0x7FFFFFDC
	0x00000000
	0x00000000
	0x00000015
0x7FFFFFB0	0x10000004
	0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

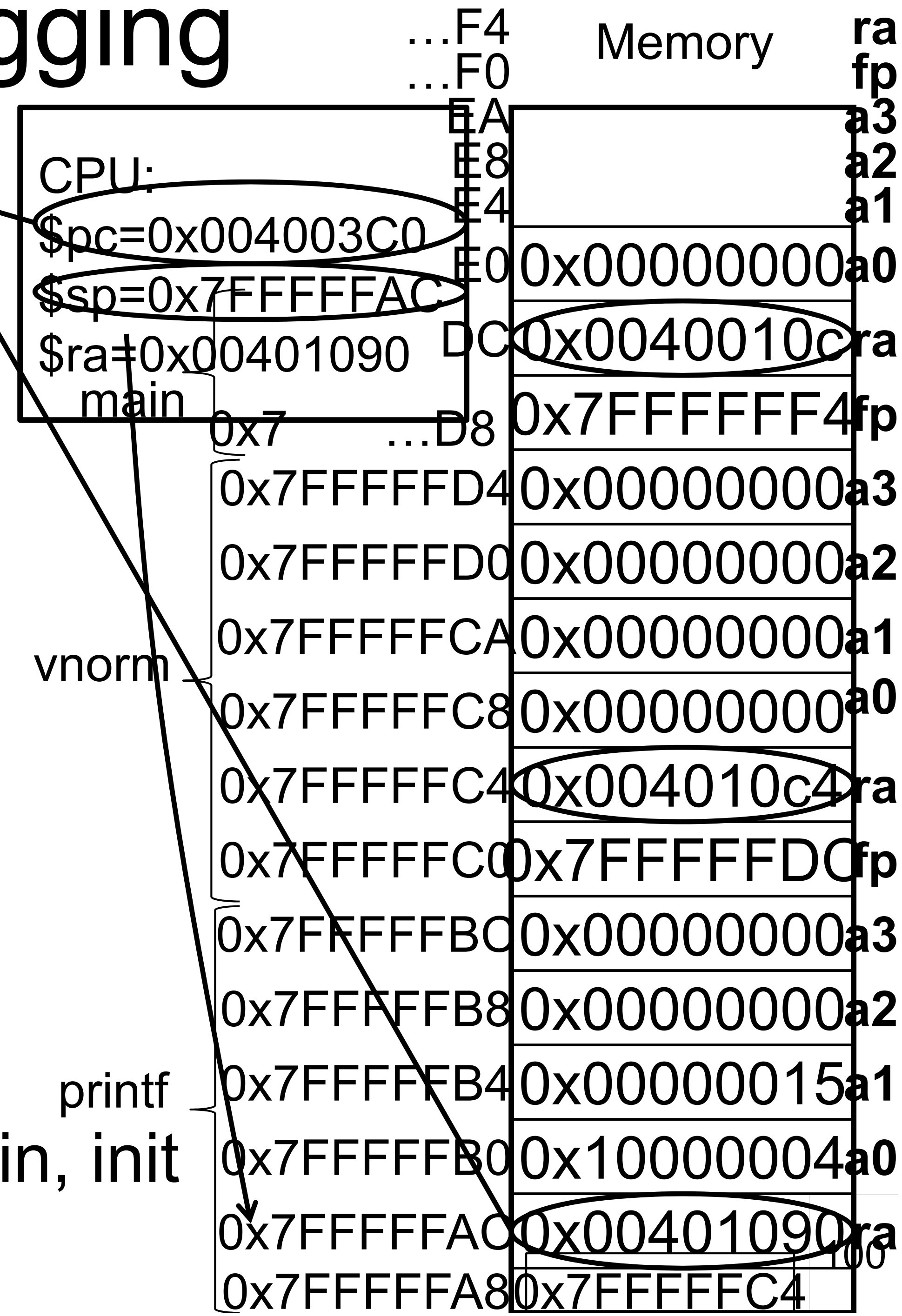
Args?

Stack depth?

Call trace?

Activity #4: Debugging

init():	0x400000
printf(s, …):	0x4002B4
vnorm(a,b):	0x40107C
main(a,b):	0x4010A0
pi:	0x10000000
str1:	0x10000004



What func is running? printf

Who called it? vnorm

Has it called anything? no

Will it? no b/c no space for outgoing args

Args? Str1 and 0x15

Stack depth? 4

Call trace? printf, vnorm, main, init

Recap

- How to write and Debug a RISCV program using calling convention
- First eight arg words passed in a0, a1, …, a7
- Space for args passed in child's stack frame
- return value (if any) in a0, a1
- stack frame (fp/s0 to sp) contains:
  - ra (clobbered on JAL to sub-functions)
  - fp
  - local vars (possibly clobbered by sub-functions)
  - Contains space for incoming args
- callee save regs are preserved
- caller save regs are not
- Global data accessed via gp