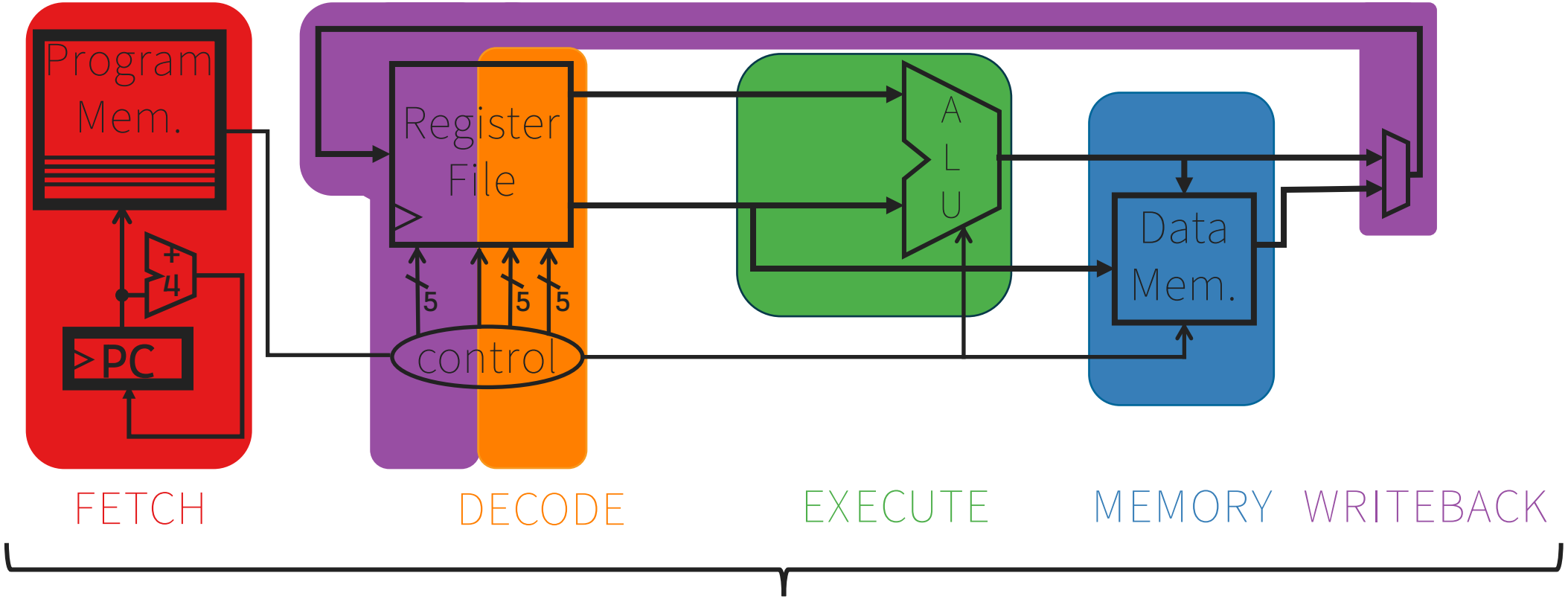
Calling Conventions

CS 3410: Computer System Organization and Programming

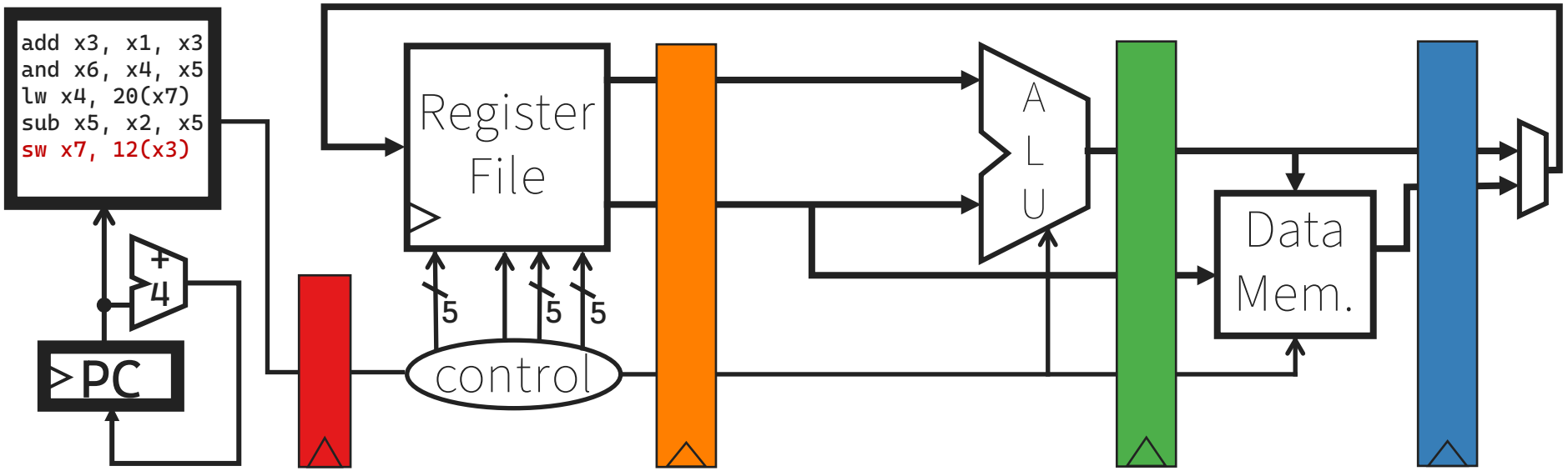
[K. Bala, A. Bracy, G. Guidi, E. Sirer, A. Sampson, Z. Susag, and H. Weatherspoon]

Recall CPU Performance: Single-Cycle



Clock frequency must be slow enough for the very slowest instruction to complete in 1 cycle

Recall CPU Performance: Pipelined



`sw x7, 12(x3)` `sub x5, x2, x5` `lw x4, 20(x7)` `and x6, x4, x5` `add x3,x1,x3`

Recall CPU Performance [PollEV Question]

Metric	Single Cycle	Multi Cycle	Pipelined
Clock Period (time / cycle)	F + D + X + M + W	MAX (F, D, X, M, W) + ε	MAX (F, D, X, M, W) + ε
Cycles Per Instruction (CPI)	1	(It depends!)	1
Performance (time / instruction)	Multiply down to see who wins!		

- How much faster is a Pipelined processor compared to a Single Cycle
- Some concrete numbers:
  - Stage latency: F = 170ns, D = 180ns, X = 200ns, M = 200ns, W = 150ns , Register = 5ns
  - **Branch**: 20% (3 cycles), **Load**: 20% (5 cycles), **ALU**: 60% (4 cycles)

Big Picture: How to ~~Design~~ Program a Processor

C

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

compiler

How do functions work?

RISC-V assembly language

```
addi x5, x0, 10      x5 = x0 + 10
muli x5, x5, 2       x5 = x5 * 2
addi x5, x5, 15      x5 = x5 + 15
```

x0 = 0

assembler

RISC-V machine language

```
000000001010 00000 000 00101 0010011
00000000001000101001001010011111
000000001111 00101 000 00101 0010011
```

10 x0 x5 op = addi

15 x5 x5 op = addi

EVERYTHING IS A NUMBER!

Big Picture: How to ~~Design~~ Program a Processor

C

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

compiler

How do functions work?

RISC-V assembly language

```
addi x5, x0, 10     <-- x5 = x0 + 10
slli x5, x5, 1      <-- x5 = x5 * 2
addi x5, x5, 15     <-- x5 = x5 + 15
```

x0 = 0

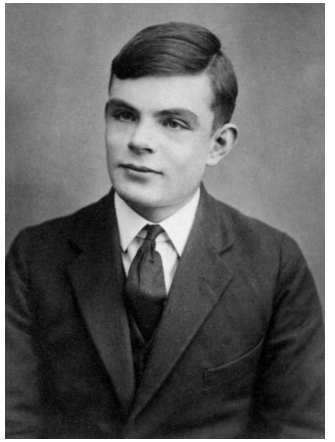
assembler

RISC-V machine language

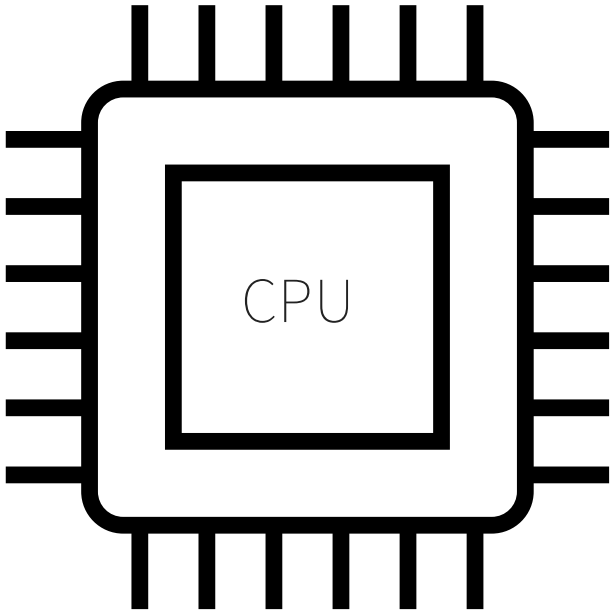
10	x0	x5	op = addi
000000001010	00000	000	00101	0010011
00000000000100101001001010011111			
000000001111	00101	000	00101	0010011
15	x5	x5	op = addi

EVERYTHING IS A NUMBER!

Big Picture: How to ~~Design~~ Program a Processor

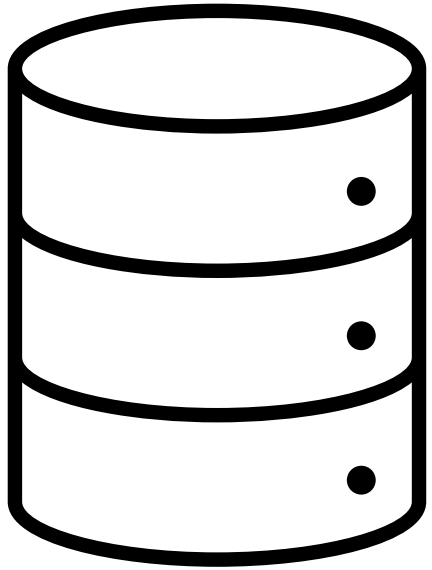


Processor



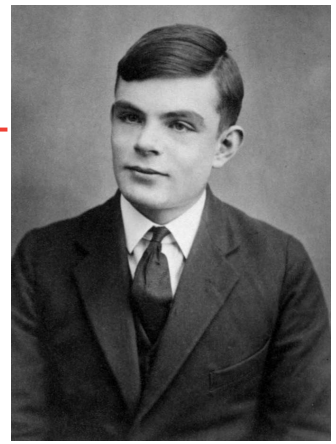
- ✓ Runs code; does computations
- ✗ Doesn't remember anything

Memory

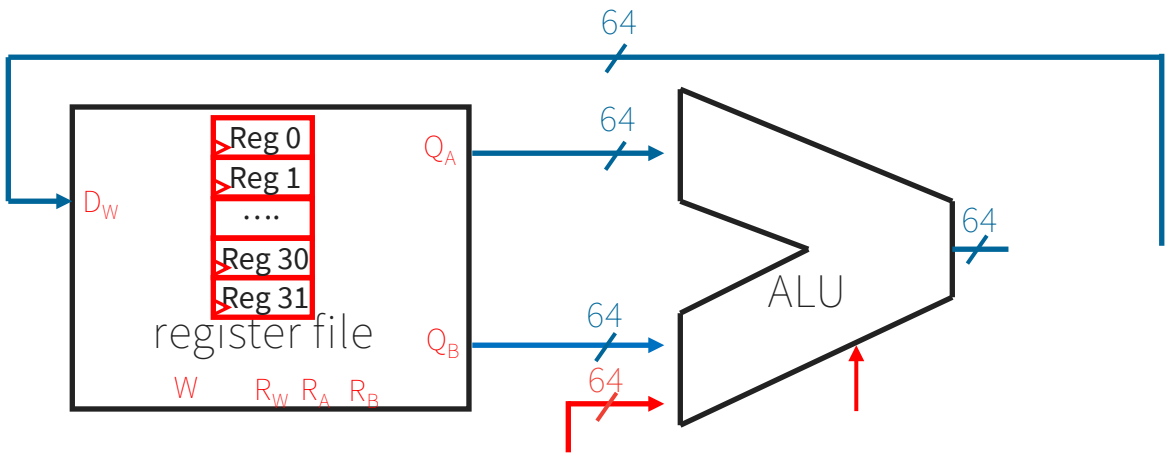


- ✗ Can't compute anything
- ✓ Stores data

Big Picture: How to ~~Design~~ Processor a Processo

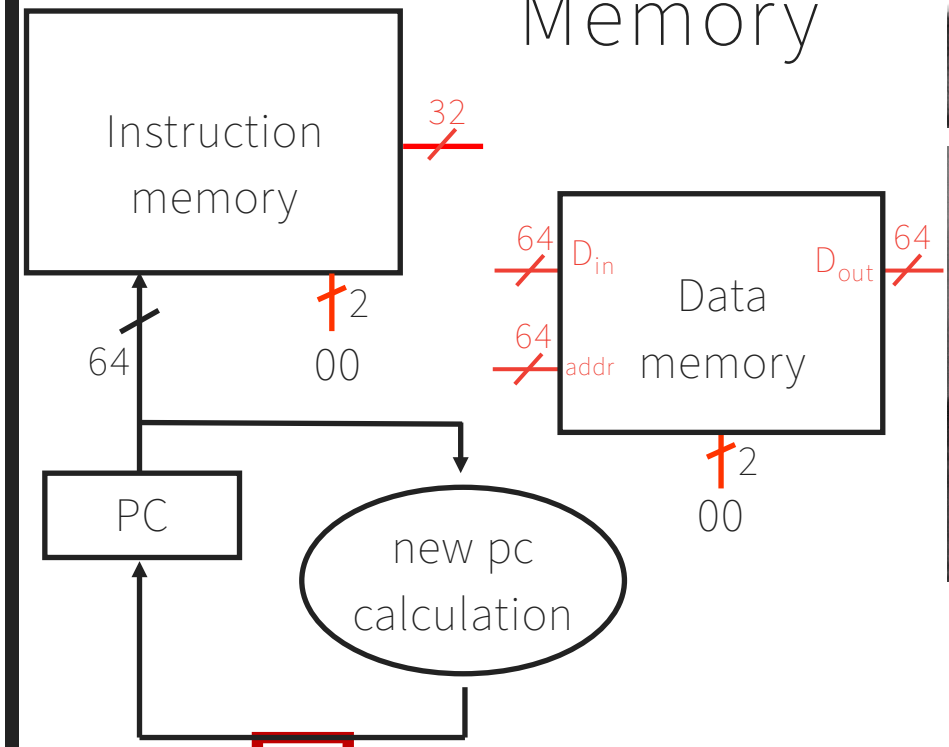


Processor



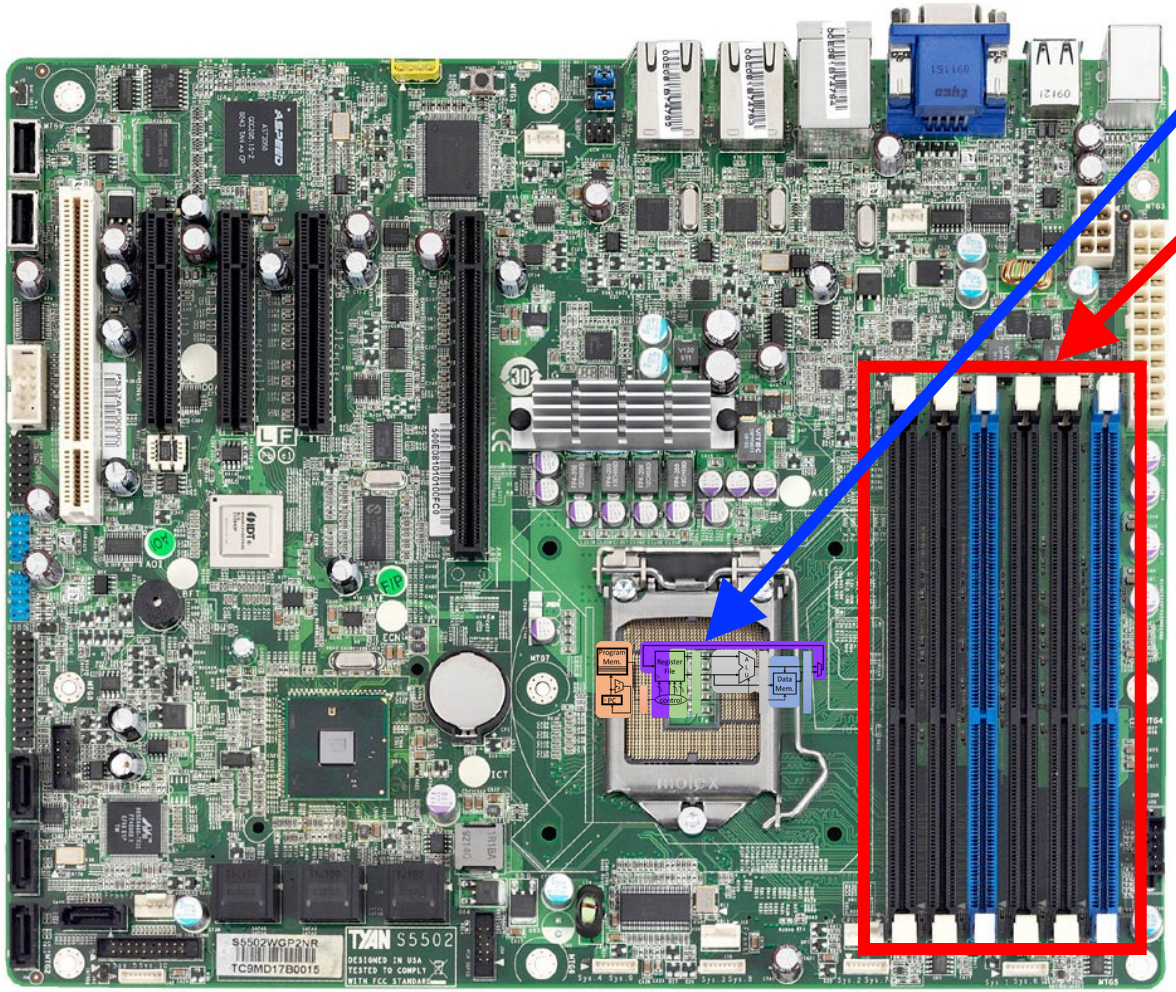
- ✓ Runs code; does computations
- ✗ Doesn't remember anything

Memory



- ✗ Can't compute anything
- ✓ Stores data

Actual picture of memory



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

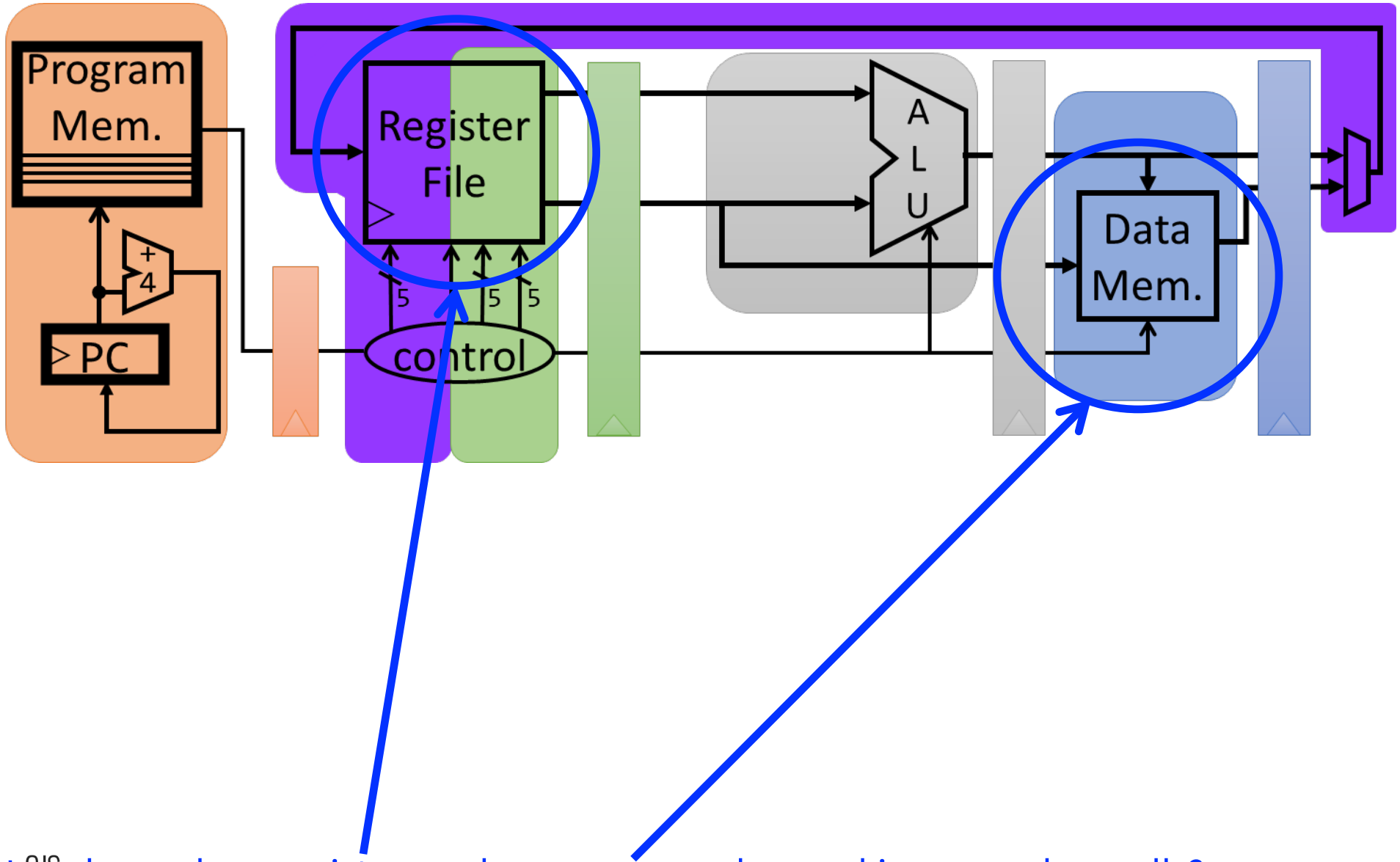
Manage Registers

- 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

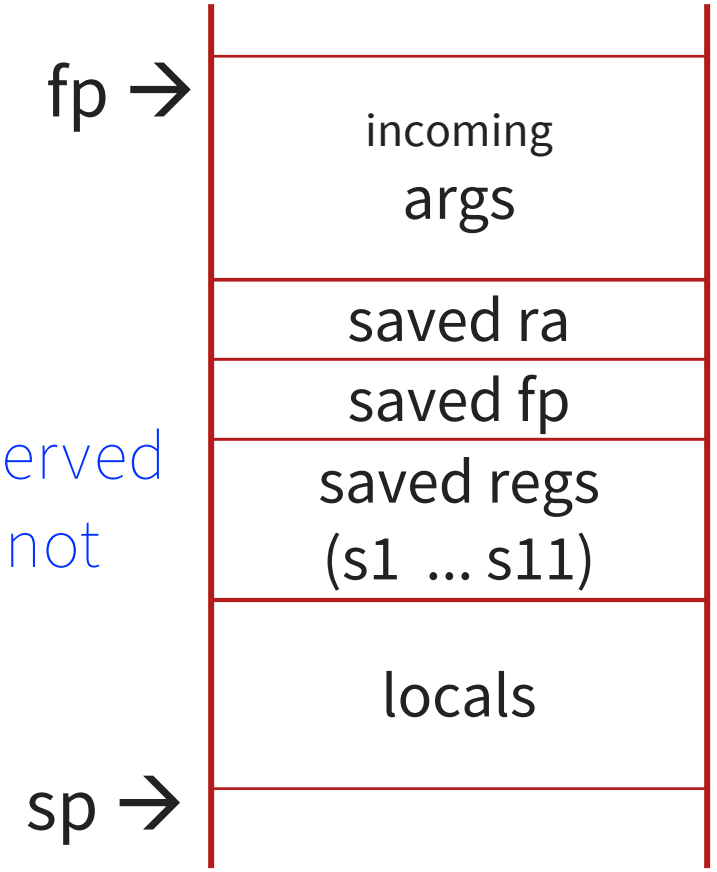
Cheat Sheet and Mental Model for Today



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 registers a0, a1, . . . , a7
- Space for args passed in childs's stack frame
- return value (if any) in a0, a1
- stack frame at sp
  - contains ra (clobbered on JAL to sub-functions)
  - contains fp
  - contains local vars
    (possibly clobbered by sub-functions)
  - contains space for incoming args
- Saved registers (callee save regs) are preserved
- Temporary registers (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

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global data pointer	- -
x4	tp	thread pointer	- -
x5-x7	t0-t2	temporaries	Caller
x8	s0/fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-x11	a0-a1	fn arguments / return values	Caller
x12-x17	a2-a7	function arguments	Caller
x18-x27	s2-s11	saved registers	Callee
x28-x31	t3-t6	temporaries	Caller

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

What goes where?

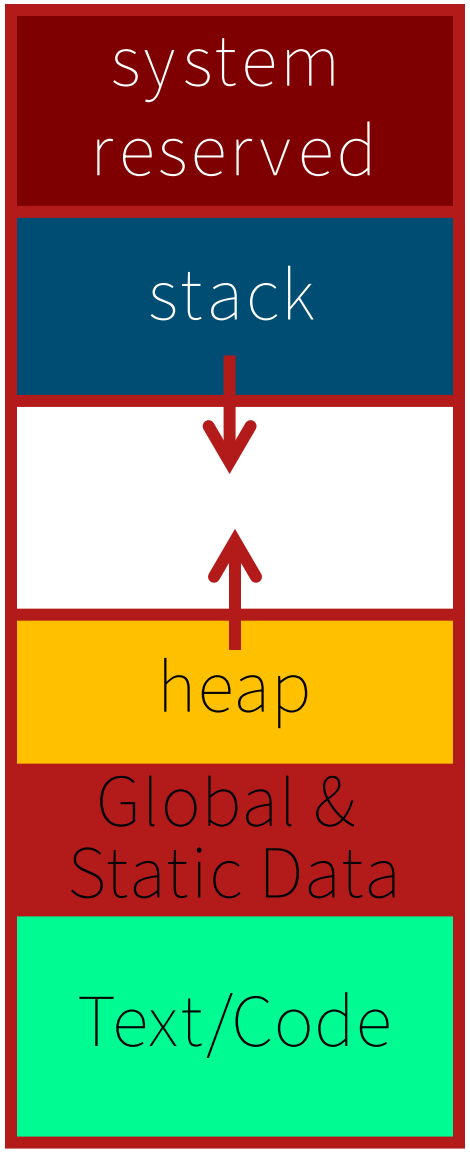
```
#include <stdio.h>

int padding = 3;

int addfn(int a) {
  return a + padding;
}

int main() {
  int sum, a = 1, b = 3;
  sum = addfn(a);
  printf("padding %d yields %d\n", a, sum);
  sum = addfn(b);
  printf("padding %d yields %d\n", b, sum);
}
```

visualize



How does a function call work?

```
int addfn(int a, int b) {
    return a + b;
}


int main() {
    int sum;
    sum = addfn(1,2);
    sum = addfn(3,4);
}
```

How does…

- main call addfn?
- addfn return back to main?
- addfn get its arguments?
- addfn return its result?

Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

Manage Registers

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

Optimizations & Manipulations?

What is a Convention?

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

lecture != book != gcc != web

JAL/JALR in action!

```
int addfn(int a, int b) {
      return a + b;
}

int main() {
   int sum;
   sum = addfn(v1,v2);
   sum = addfn(v1,v2);
}
```

```
addfn:
x040      ADD
x044      JALR x0,0(x1)

main:
x100      JAL x1, -x060
x104      JAL x1, -x064
x108
```

```
JAL    R[rd] = PC+4; PC = PC+IMM
JALR   R[rd] = PC+4; PC = R[rs1]+IMM
```

x0 cannot be overwritten
(not using this functionality)

PC	x100
x1	?

JAL/JALR in action!

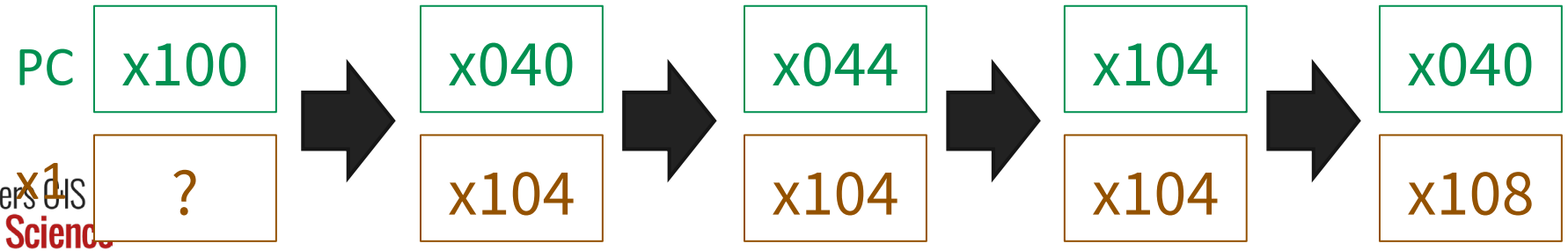
```
int addfn(int a, int b) {
      return a + b;
}

int main() {
   int sum;
   sum = addfn(v1,v2);
   sum = addfn(v1,v2);
}
```

```
addfn:
x040      ADD
x044      JALR x0,0(x1)

main:
x100      JAL x1, -x060
x104      JAL x1, -x064
x108
```

```
JAL    R[rd] = PC+4; PC = PC+IMM
JALR   R[rd] = PC+4; PC = R[rs1]+IMM
```



RISC-V Register Conventions

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller

Anybody worried about anything?

```
int addfn(int a, int b) {
      return a + b;
}

int main() {
   int sum;
   sum = addfn(v1,v2);
   sum = addfn(v1,v2);
}
```

```
        addfn:
x040        ADD
x044        JALR x0,0(x1)

        main:
x100        JAL x1, -x060
x104        JAL x1, -x064
x108
```

What about now?

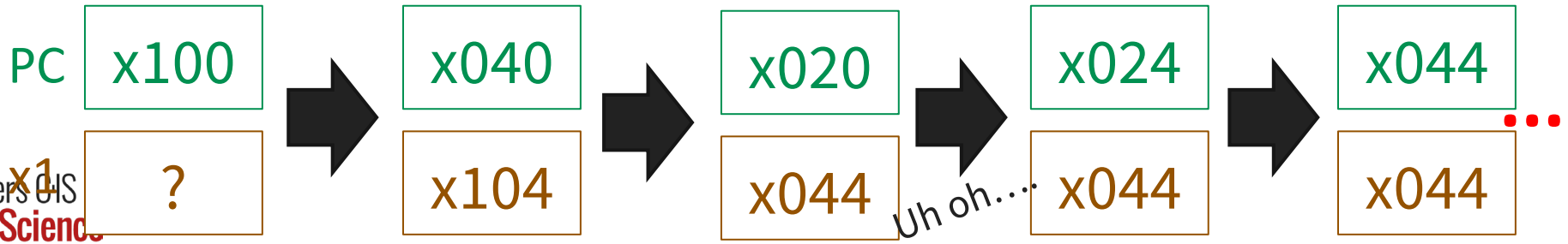
```
int addfn(int a, int b) {
      printf("hi");
      return a + b;
}

int main() {
   int sum;
   sum = addfn(v1,v2);
   sum = addfn(v1,v2);
}
```

```
        printf:
x020        BLAHBLAHBLAH
x024        JALR x0,0(x1)

        addfn:
x040        JAL x1, -x020
x044        ADD
x048        JALR x0,0(x1)

        main:
x100        JAL x1, -x060
x104        JAL x1, -x064
x108
```

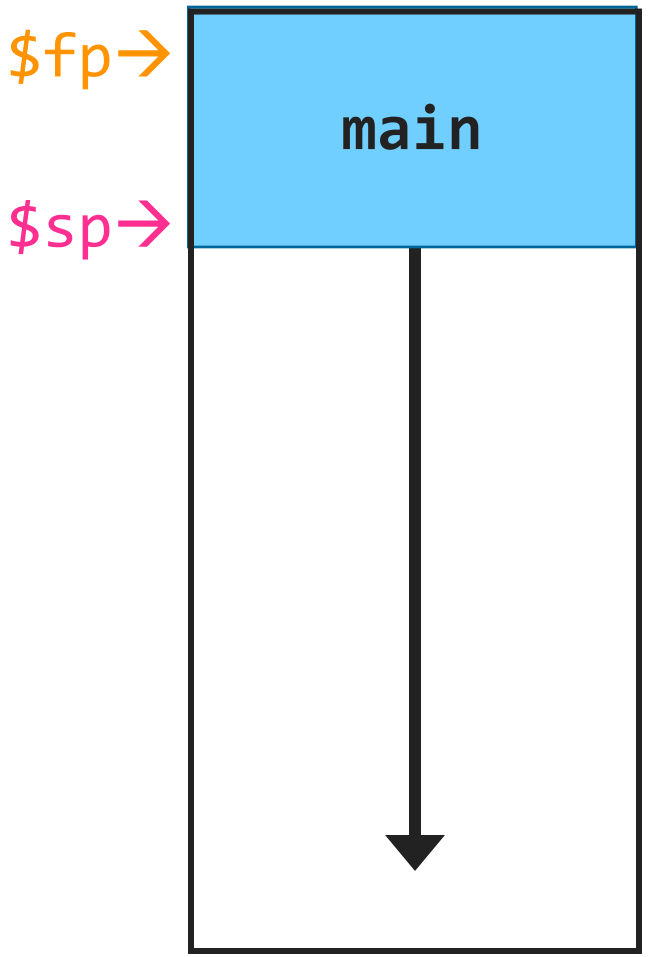


Enter: the Call Stack

Part of memory that holds function data

- 1 stack frame per *dynamic* function
- exists only for the life of the function
- grows **down**
  - $fp points to "bottom"
  - $sp points to "top"

```
int addfn(int a, int b) {
        printf("hi");
        return a + b;
}
int main() {
    int sum;
    sum = addfn(v1,v2);
    sum = addfn(v1,v2);
}
```

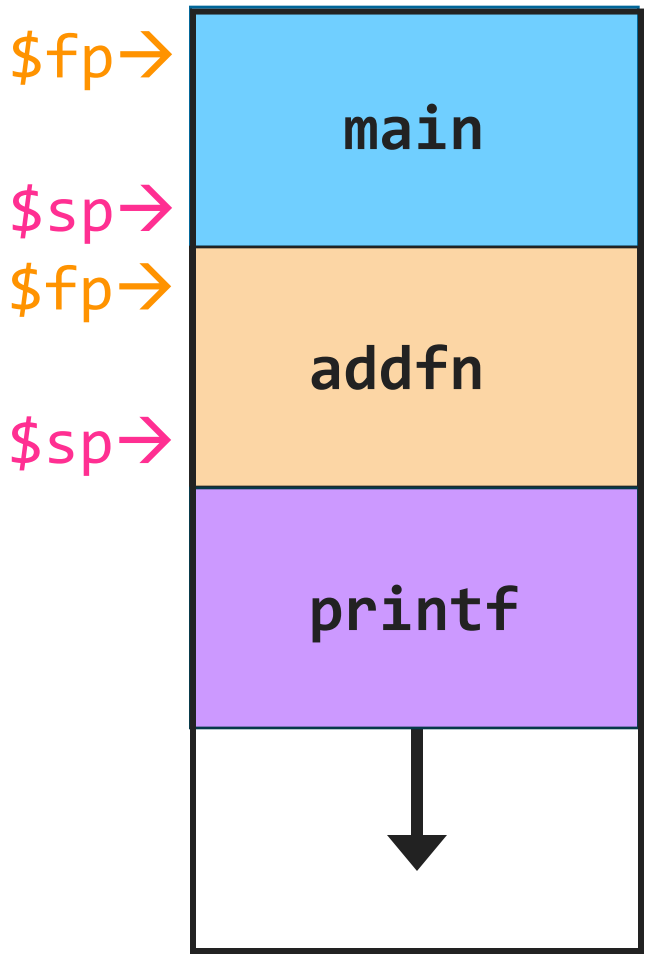


Enter: the Call Stack

Part of memory that holds function data

- 1 stack frame per *dynamic* function
- exists only for the life of the function
- grows **down**
  - $fp points to "bottom"
  - $sp points to "top"

```
int addfn(int a, int b) {
    printf("hi");
    return a + b;
}
int main() {
  int sum;
  sum = addfn(v1,v2);
  sum = addfn(v1,v2);
}
```



RISC-V Register Conventions

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x8	s0/fp	frame pointer	Callee

Stack Manipulations

PUSH something onto the stack:

Example: store your $ra on the stack

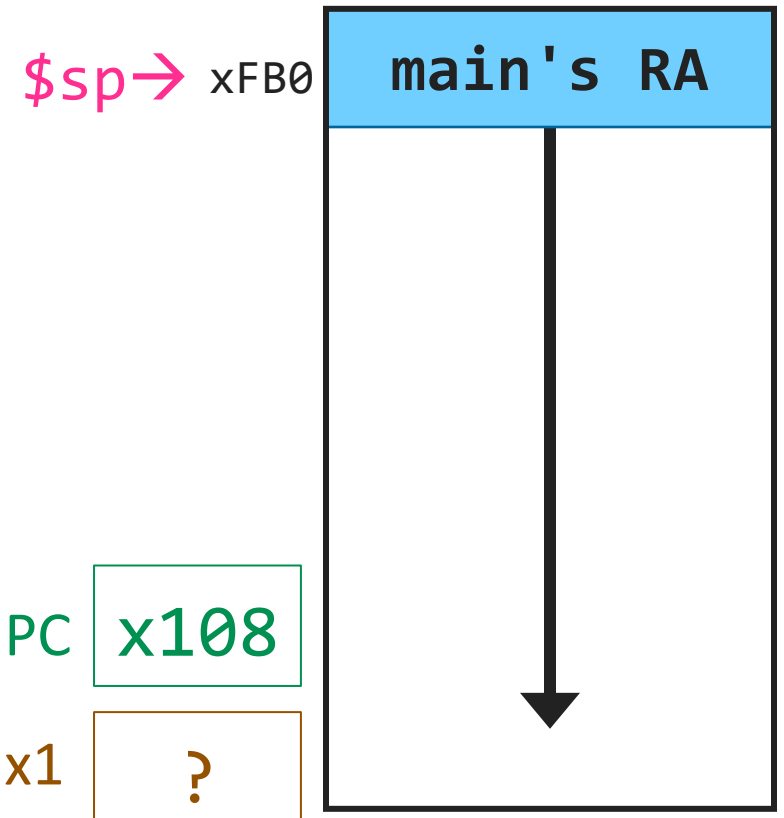
- Grow the stack: `ADDI sp, sp, -8`
- Put $ra at new "top" `SD x1, 0(sp)`

POP something off of the stack:

Example: pop your $ra off the stack

- Put ret addr back in $ra `LD x1, 0(sp)`
- Shrink the stack: `ADDI sp, sp, 8`

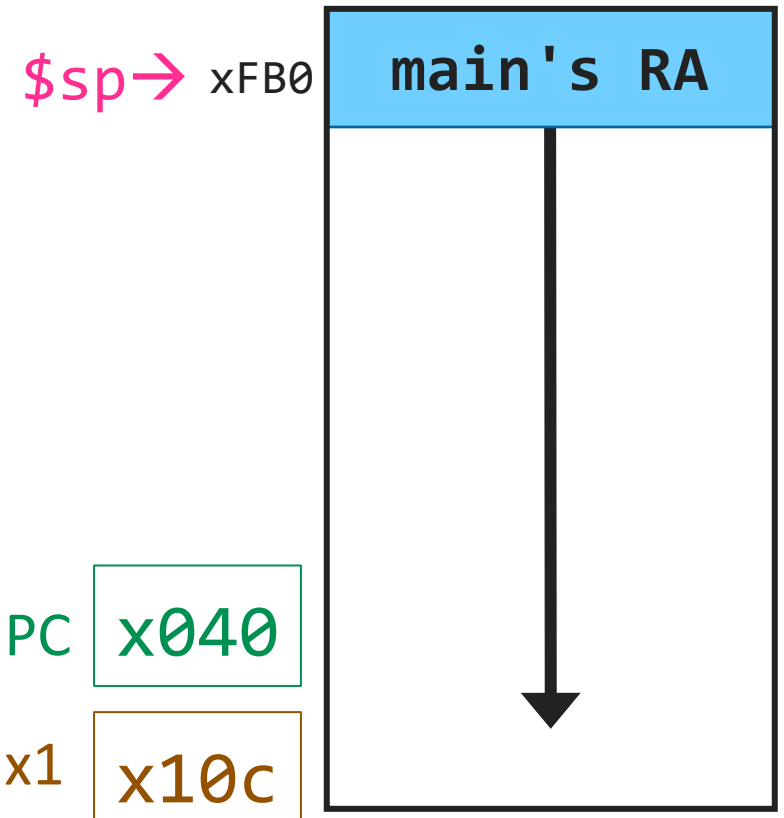
RA Example (1)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

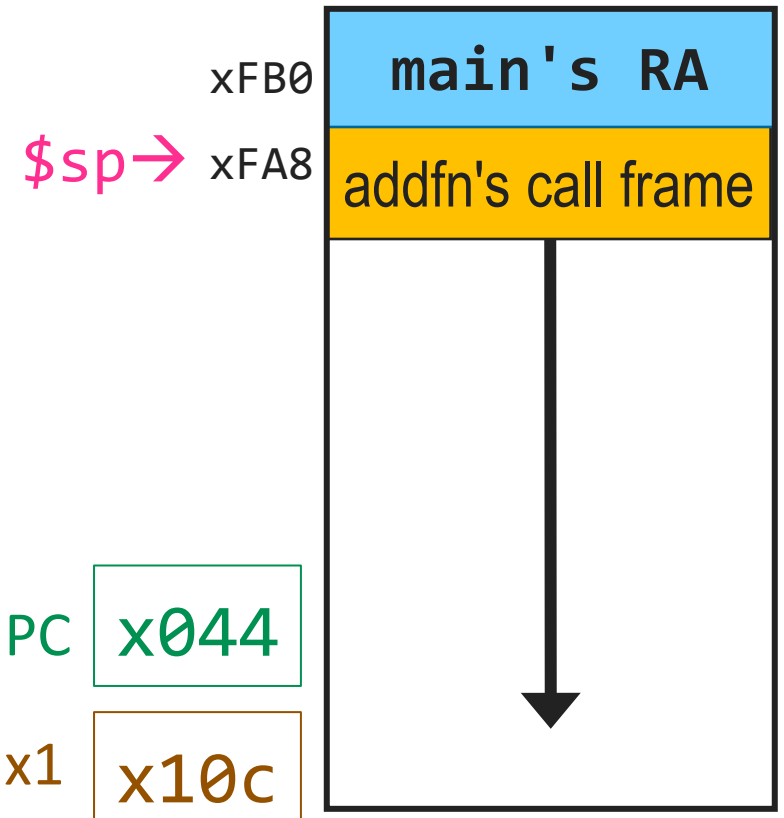
RA Example (2)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 4
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

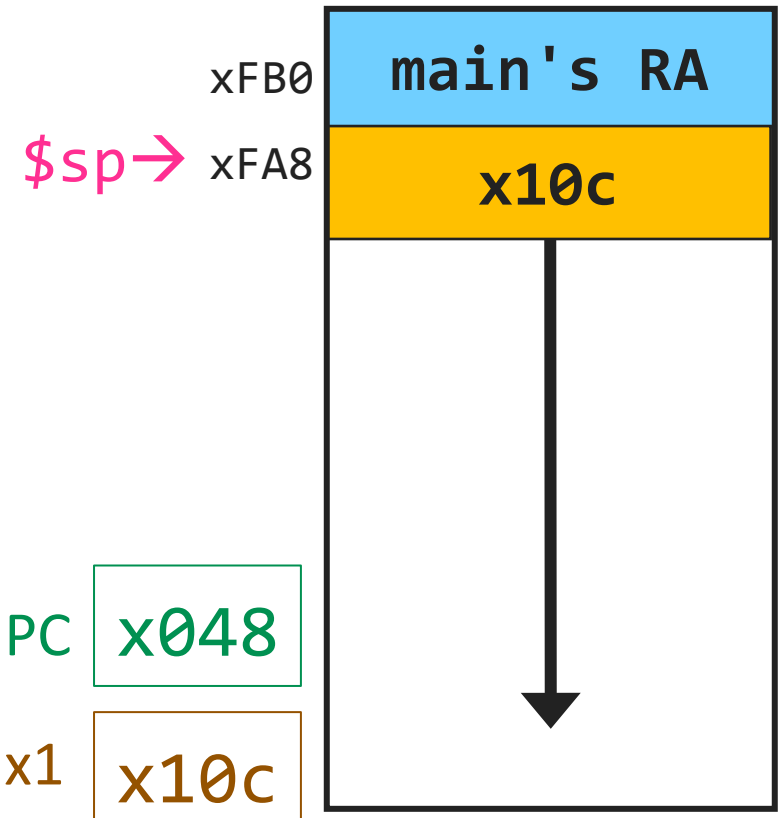
RA Example (3)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

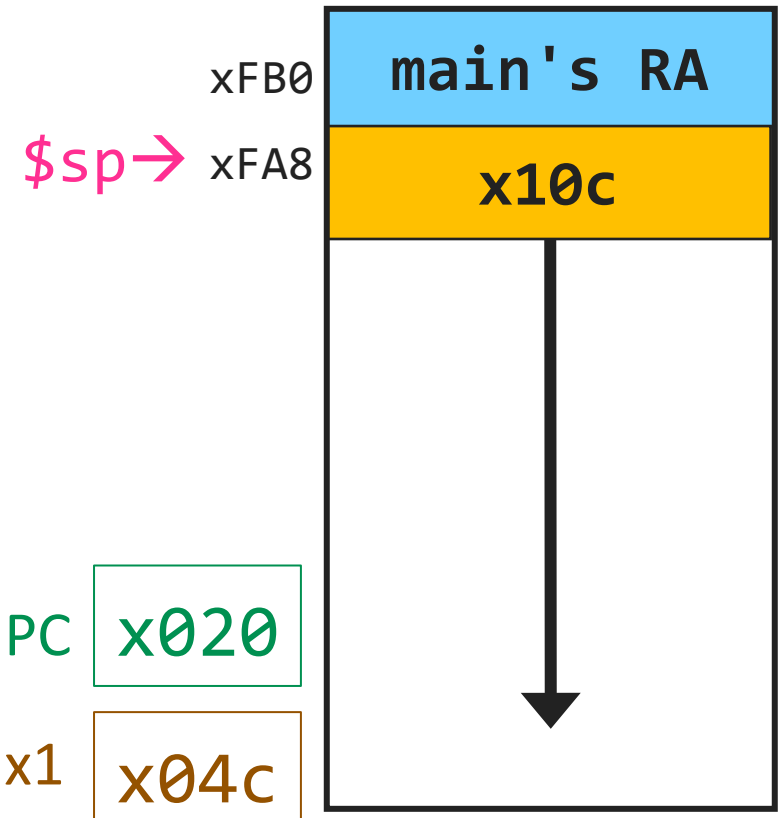
RA Example (4)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

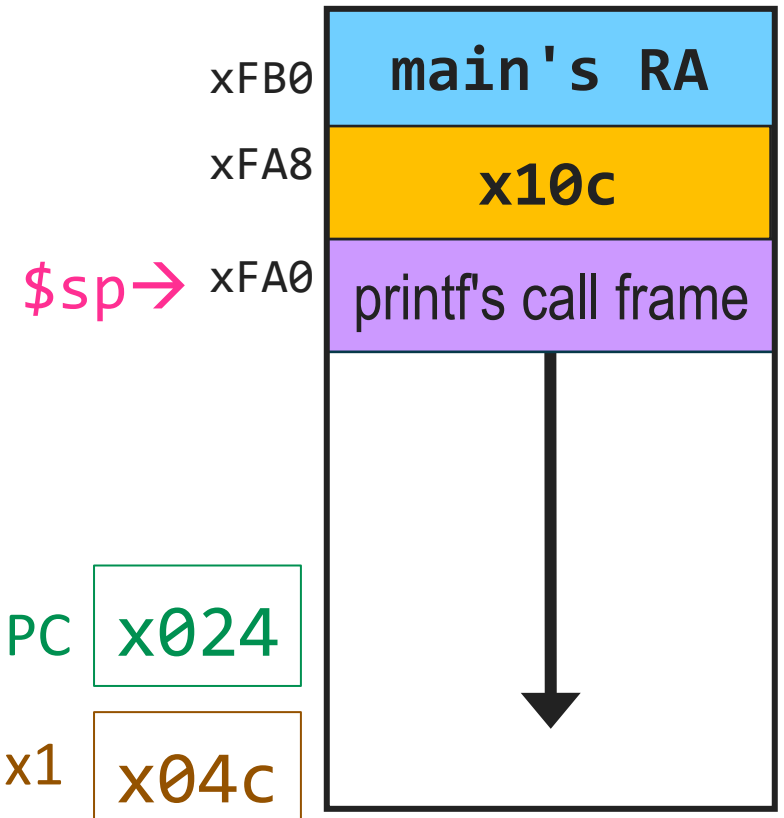
RA Example (5)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

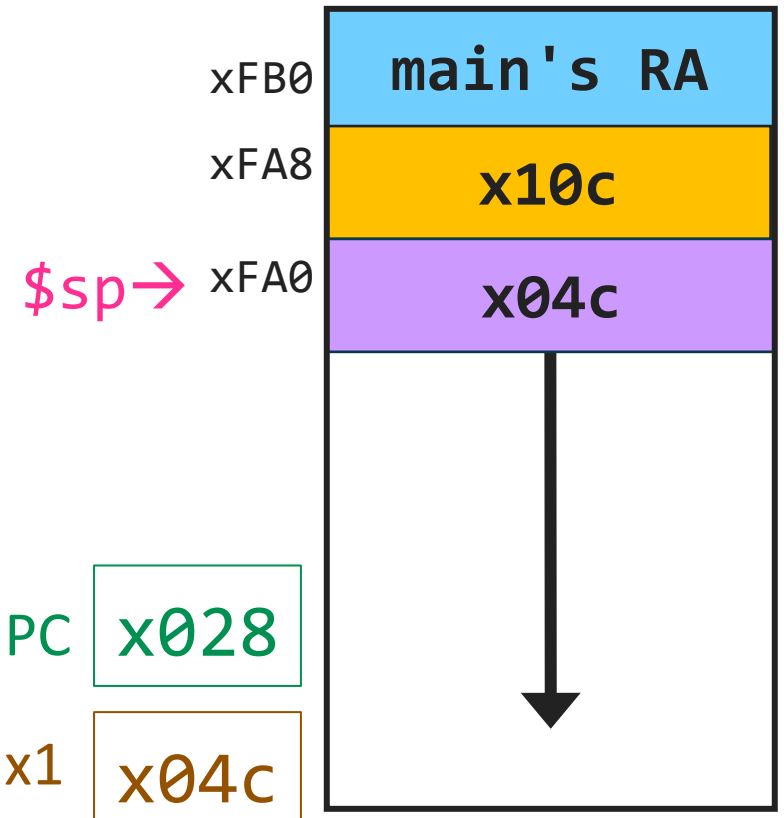
RA Example (6)




```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

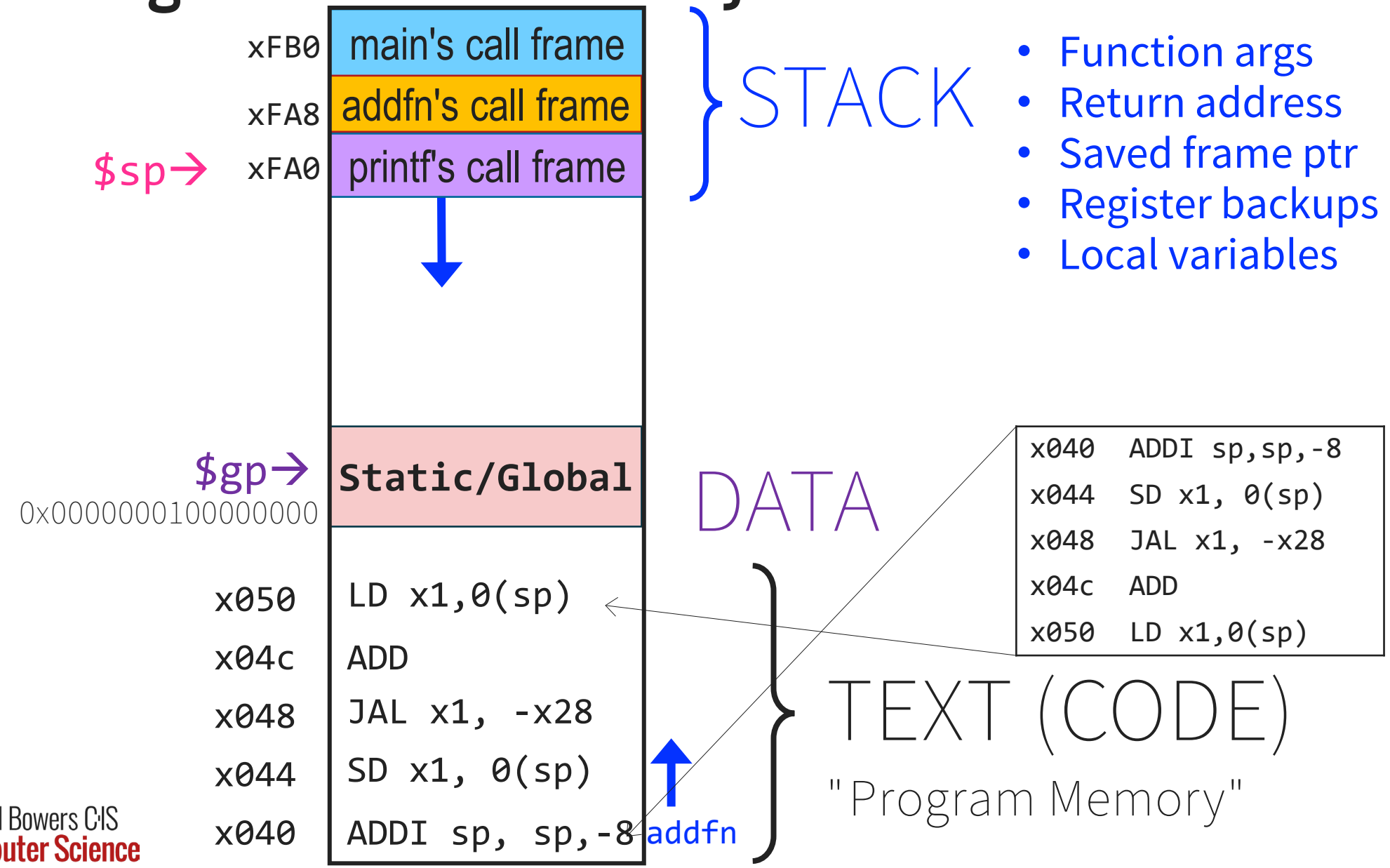
RA Example (7)



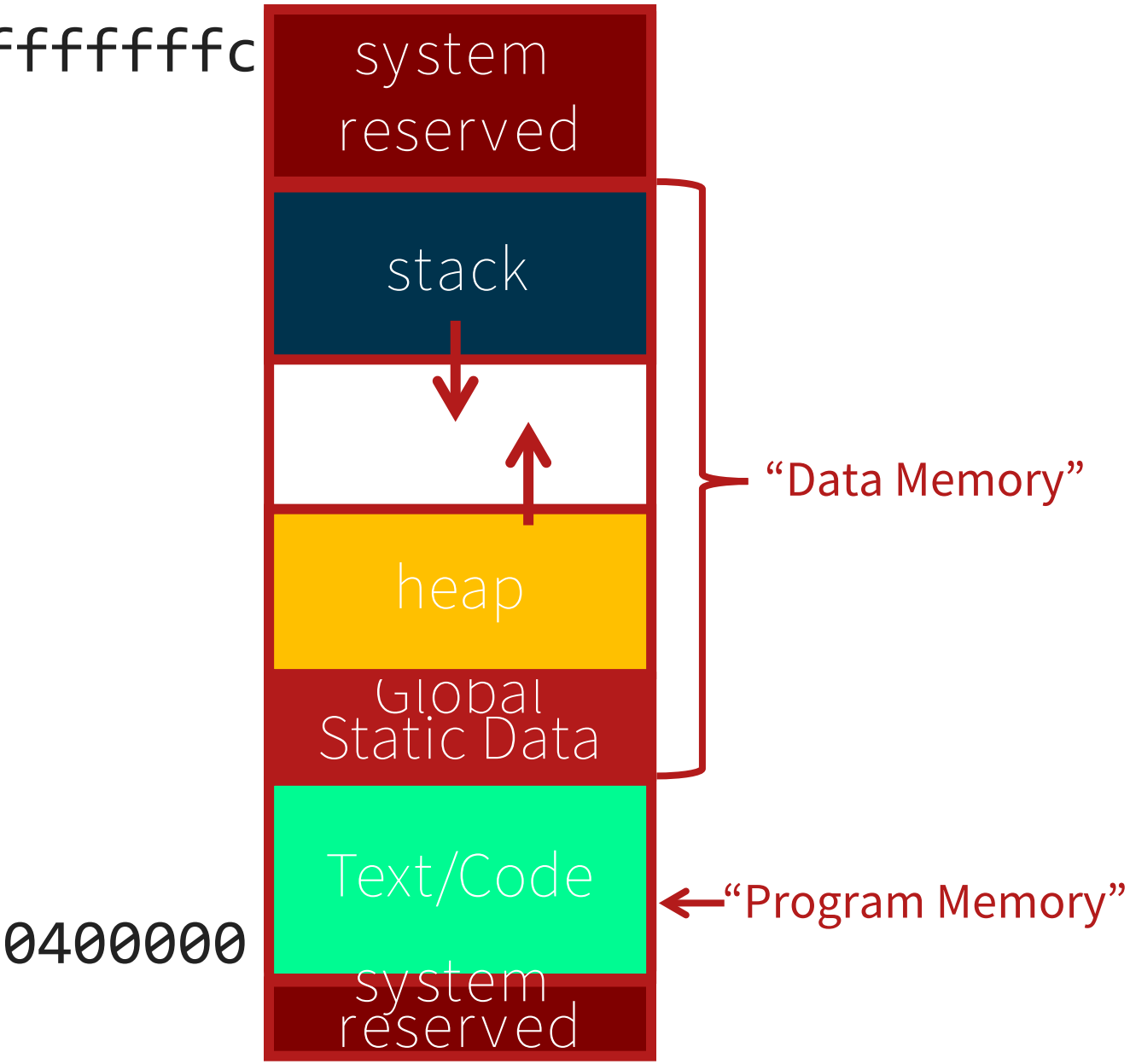

```
x020  printf:  ADDI sp,sp,-8
x024           SD x1, 0(sp)
x028           ...
x0..           JARL x0,0(x1)
               ...
x040  addfn:   ADDI sp,sp,-8
x044           SD x1, 0(sp)
x048           JAL x1, -x28
x04c           ADD
x050           LD x1,0(sp)
x054           ADDI sp, sp, 8
x058           JALR x0,0(x1)

x100  main:    ADDI sp,sp,-8
x104           SD x1, 0(sp)
x108           JAL x1, addfn
x10c           JAL x1, addfn
```

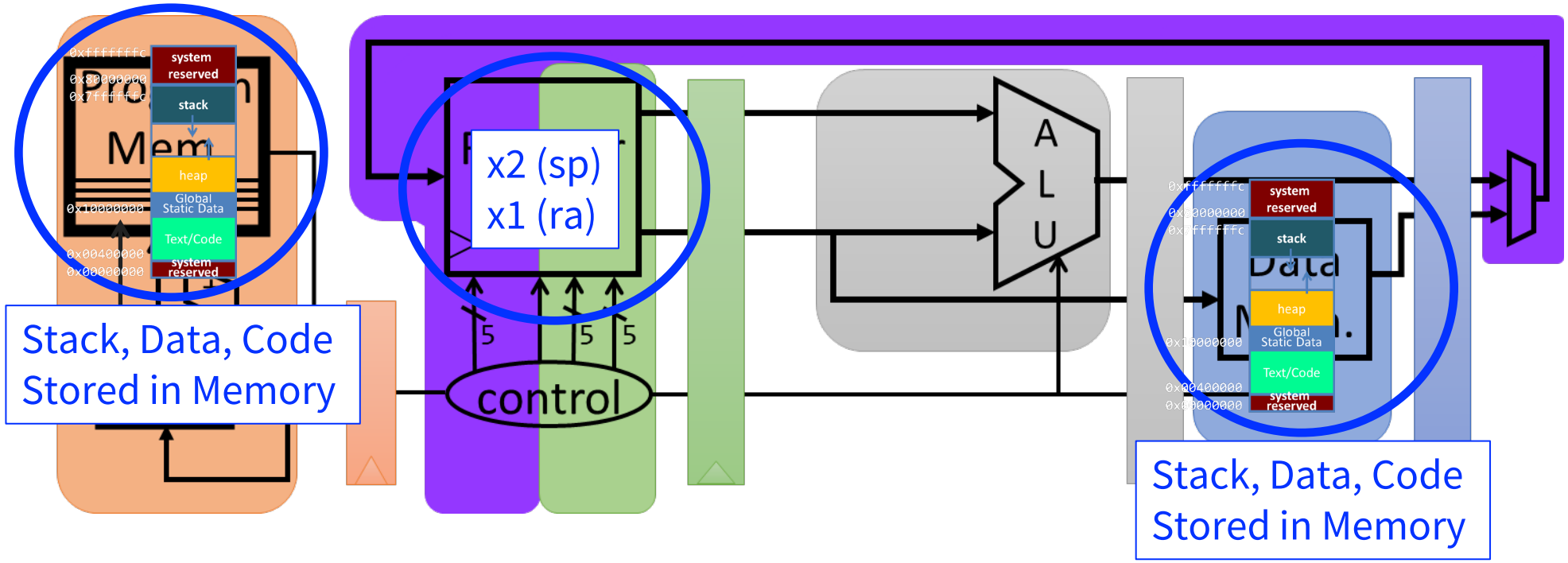
The Big Picture of Memory



An executing program in memory



Anatomy of an executing program



Global Data

How does a function load global data?

- global variables are just above 0x0000000100000000

Convention: *global pointer*

- x3 is gp (pointer into *middle* of global data section)
  gp = 0x0000000100000800
- Access most global data using LW at gp +/- offset

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

Cornell Bowers CIS
Computer Science

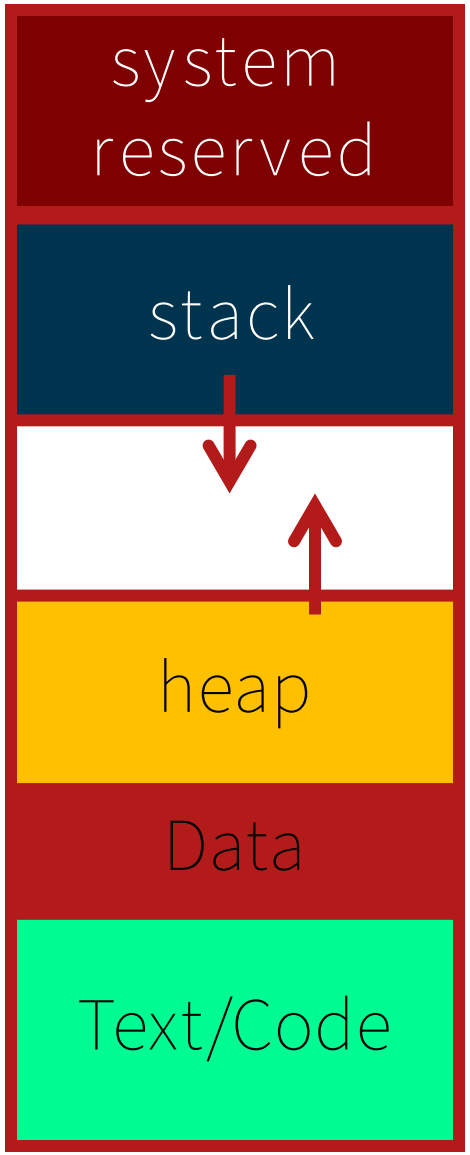
RISC-V Register Conventions

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global data pointer	--
x8	s0/fp	frame pointer	Callee

Globals and Locals

Where is…

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



Globals and Locals [PollEV Question #2]

Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

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

How does a function call work?

```
int addfn(int a, int b) {
    return a + b;
}


int main() {
    int sum;
    sum = addfn(1,2);
    sum = addfn(3,4);
}
```

How does…

- main call addfn? ✓
- addfn return back to main? ✓
- addfn get its arguments?
- addfn return its result?

Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

- Arguments passed to a routine via x10-x17
- Return values passed back to the caller via x10, x11

Manage Registers

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

Optimizations & Manipulations?

Second Lecture Started Here

Next Goal

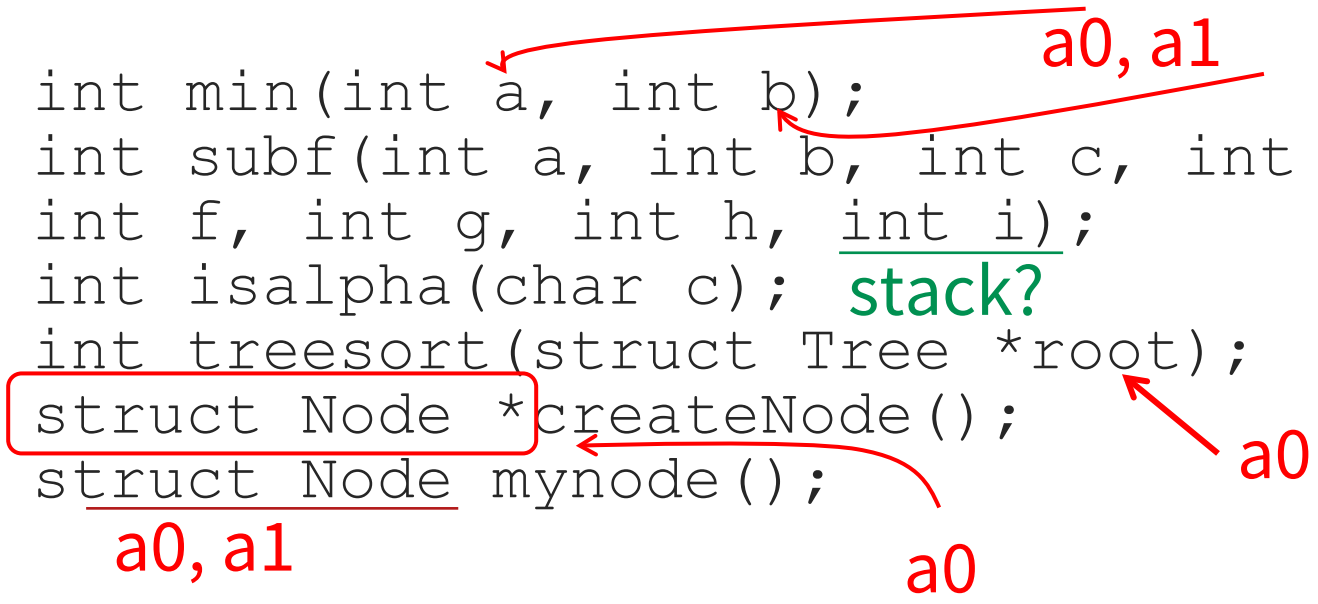
- Need consistent way of passing arguments and getting the result of a subroutine invocation

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

- `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);`
- `int isalpha(char c);`
- `int treesort(struct Tree *root);`
- `struct Node *createNode();`
- `struct Node mynode();`



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 = addfn(6, 7);
  x = x + 2;
}
```

```
main:
  li x10, 6
  li x11, 7
  jal addfn
  addi x10, 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:

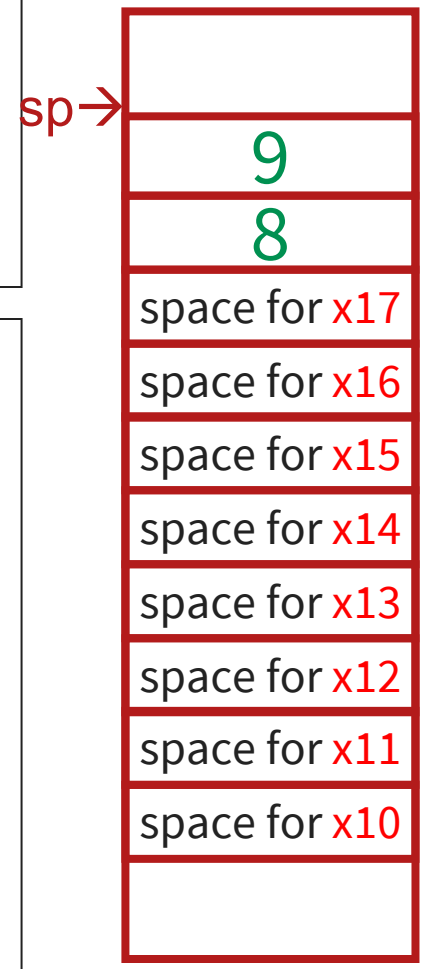
- args passed in a0, a1, …, a7
- return value (if any) in a0, a1
- stack frame at sp
  - contains ra (clobbered on JAL to sub-functions)

Q: What about argument lists?

Many Arguments (8+ args)

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

```
main:
  li x10, 0
  li x11, 1
  …
  li x17, 7
  li x5, 8
  sw x5, -16(x2)
  li x5, 9
  sw x5, -5(x2)
  jal addfn
```



First eight arguments:

passed in registers x10-x17

- aka a0, a1, …, a7

Subsequent arguments:

"spill" onto the stack

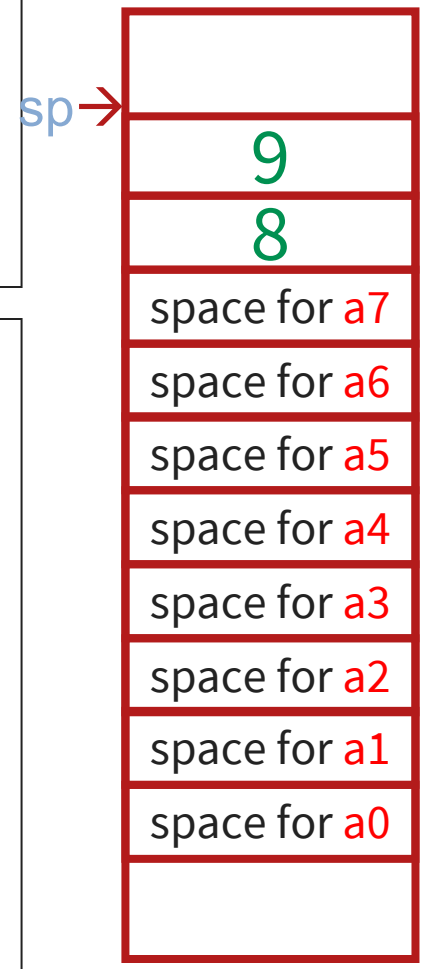
Args passed in child's stack frame

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

Many Arguments (8+ args)

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

```
main:
  li a0, 0
  li a1, 1
  …
  li a7, 7
  li t0, 8
  sw t0, -16(sp)
  li t0, 9
  sw t0, -8(sp)
  jal addfn
```



First eight arguments:

passed in registers x10-x17

- aka a0, a1, …, a7

Subsequent arguments:

"spill" onto the stack

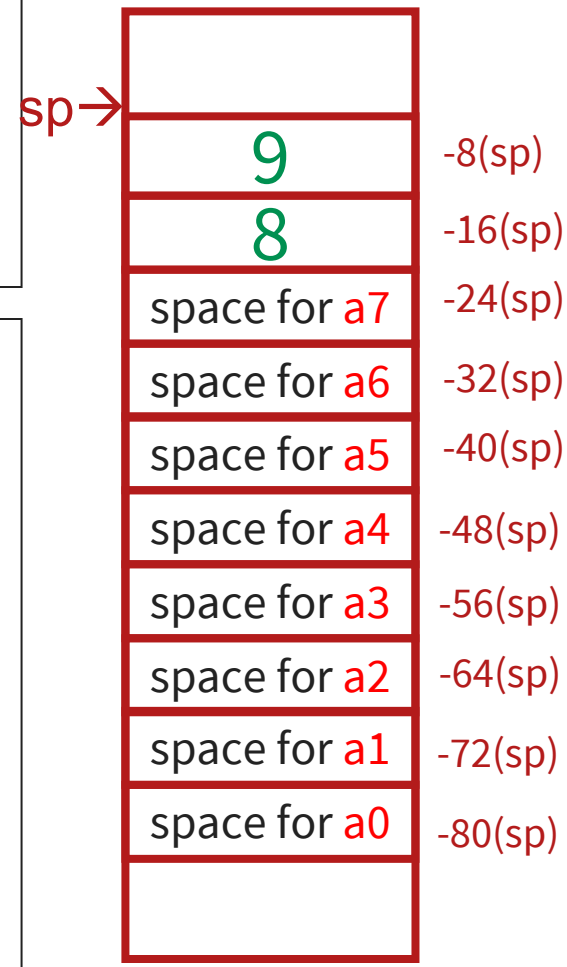
Args passed in child's stack frame

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

Argument Passing: *the Full Story*

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

```
main:
  li a0, 0
  li a1, 1
  …
  li a7, 7
  li t0, 8
  sw t0, -16(sp)
  li t0, 9
  sw t0, -8(sp)
  jal addfn
```



Arguments 1-8:

passed in x10-x17 *room on stack*

Arguments 9+:

placed on stack

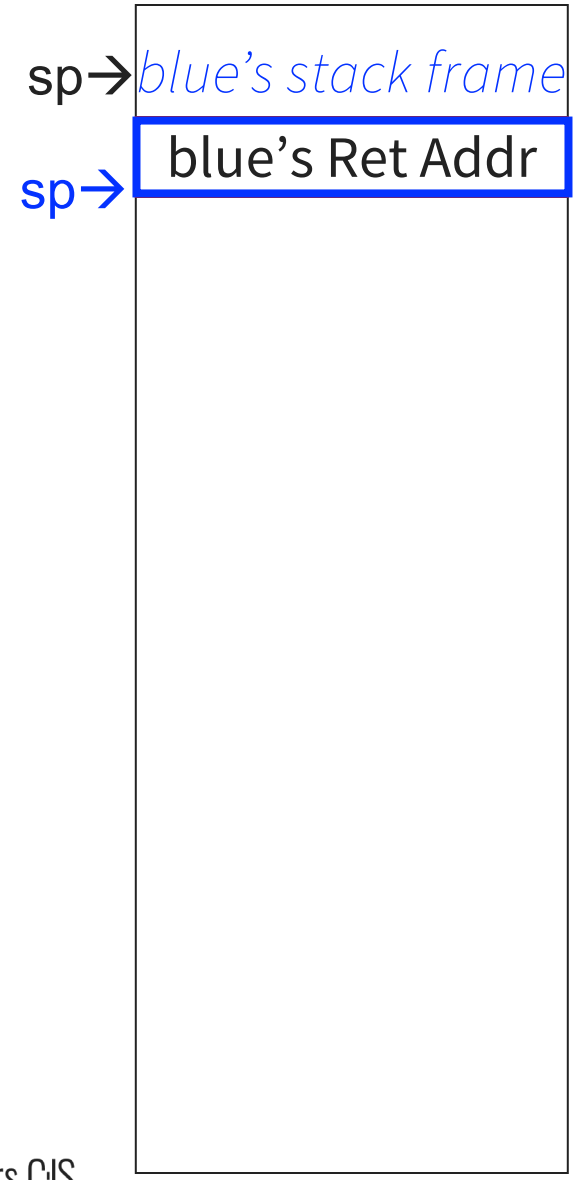
Args passed in child's stack frame

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

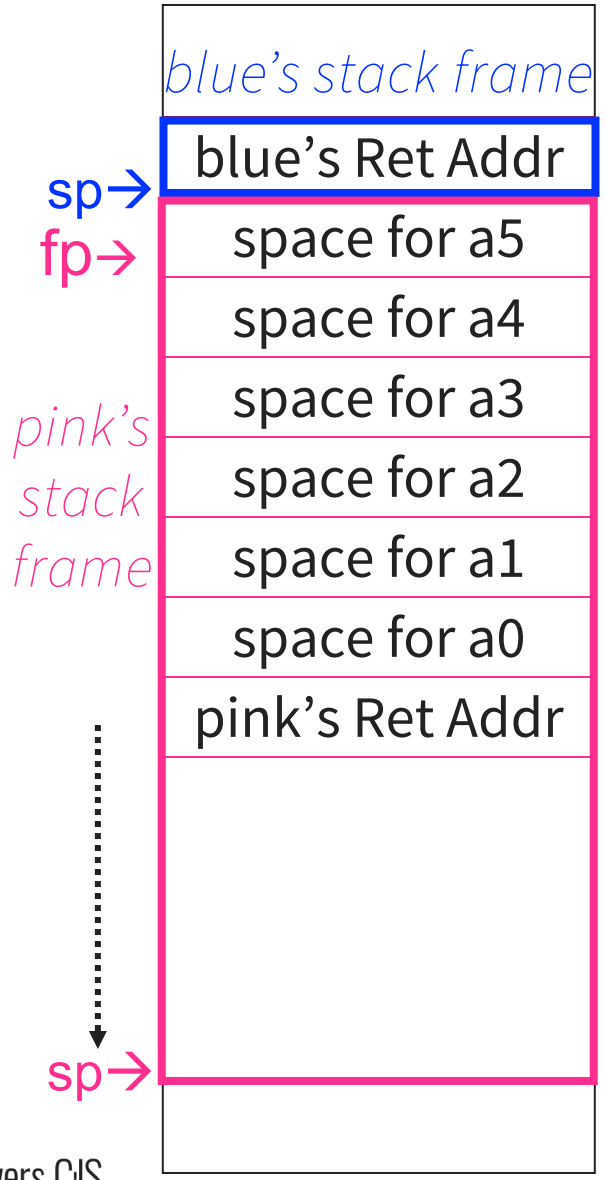
    → callee may treat all args as an array in memory
  - Particularly helpful for functions w/ variable length inputs: `printf("Scores: %d %d %d\n", 1, 2, 3);`
- Aside: not a bad place to store inputs if callee needs to call a function (your input cannot stay in a0 if you need to call another function!)

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 frame
- **sp** changes as functions call other functions, complicates accesses
- Convenient to keep pointer to bottom of stack == **frame pointer x8**
  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) {
    ...
}
```

Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

- Arguments passed to a routine via x10-x17
- Return values passed back to the caller via x10, x11

Manage Registers

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

Optimizations & Manipulations?

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: *Saved Registers* and *Temporary Registers*

Saved Registers (Callee-saved) `$s1-$s11, $gp, $tp, $sp`:

- Assume that another live function is currently using the register
  - On procedure entry: save the contents of the register
  - Just before procedure return: restore contents to the register
- Contents are safe even if you call a function

Temporary Registers (Caller-saved) `$a0-a7, $t0-$t6`:

- You don't need to protect the contents before you use the register (neither will any function you call)
- If you call a function:
  - Before procedure call: save the previous contents to stack
  - After procedure call: restore the value back to the register
  - (Otherwise, assume value was destroyed during the call)

Temporary (Caller-Saved) Registers in Practice

```
main:
 …
 [use x5 & x6]
 …
 addi x2, x2, -16
 sd x6, 8(x2)
 sd x5, 0(x2)
 jal addfn
 ld x6, 8(x2)
 ld x5, 0(x2)
 addi x2, x2, 16
 …
 [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.

Temporary (Caller-Saved) Registers in Practice

```
main:
 …
 [use t0 & t1]
 …
 addi sp, sp, -16
 sd t1, 8(sp)
 sd t0, 0(sp)
 jal addfn
 ld t1, 8(sp)
 ld t0, 0(sp)
 addi sp, sp, 16
 …
 [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.

Saved (Callee-Saved) Registers in Practice

```
main:
  addi x2, x2, -32
  sd x1, 24(x2)
  sd x8, 16(x2)
  sd x18, 8(x2)
  sd x9, 0(x2)
  addi  x8, x2, 24
            …
  [use x9 and x18]
     …
  ld x1, 24(x2)
  ld x8, 16(x2)
  ld x18, 8(x2)
  ld x9, 0(x2)
  addi x2, x2, 32
  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.

Saved (Callee-Saved) Registers in Practice

```
main:
  addi sp, sp, -32
  sd ra, 24(sp)
  sd fp, 16(sp)
  sd s2, 8(sp)
  sd s1, 0(sp)
  addi  fp, sp, 24
          …
  [use s1 and s2]
     …
  ld ra, 24(sp)
  ld fp, 16(sp)
  ld s2, 8(sp)
  ld s1, 0(sp)
  addi sp, sp, 32
  jr ra
```

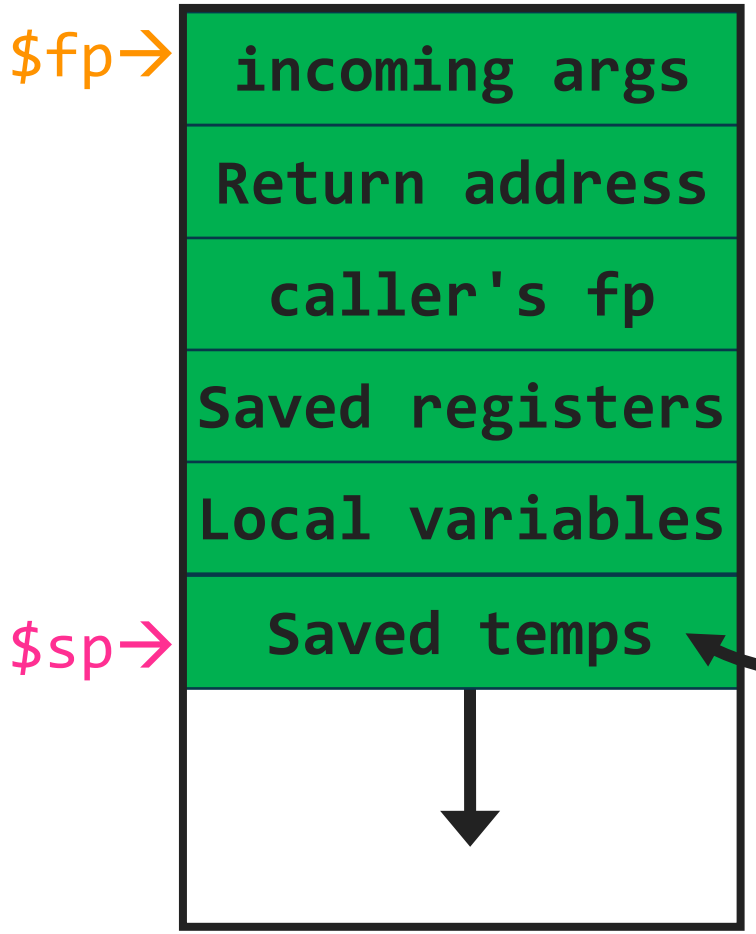
Assume caller is using the registers

- 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.

So... What's in a Stack Frame?



←Incoming args to purple fn

←Where purple fn returns to

←Bottom of caller's call frame

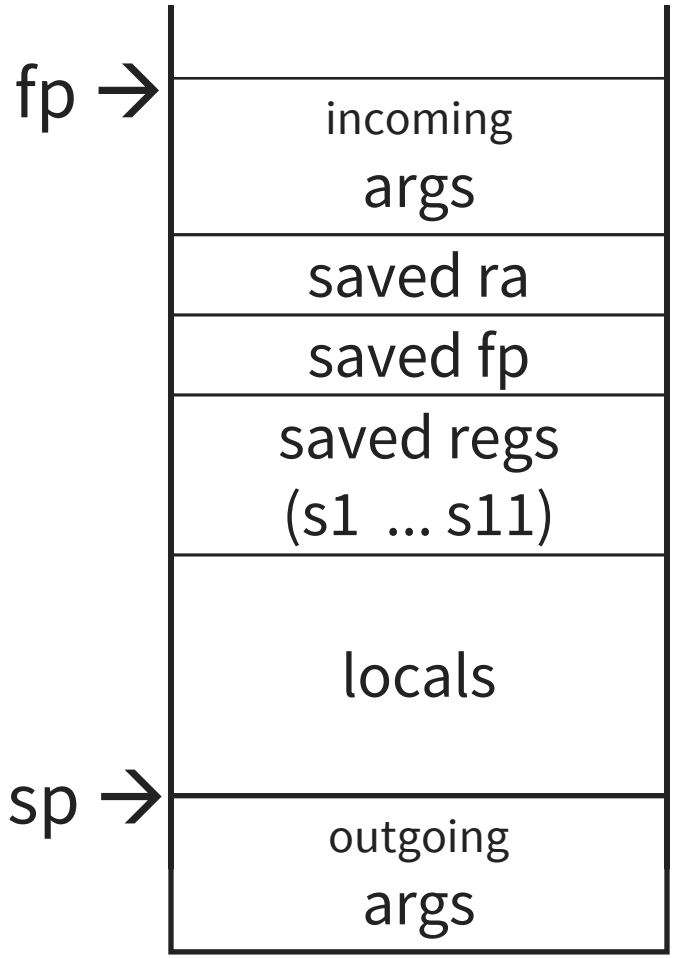
← x18-27 (s2-s11) that purple fn used

← local variables

- Arrays
- Anything that needs an address
- Anything that doesn't fit in a register

Temporary register contents that need to be saved across fn calls

Frame Layout on Stack



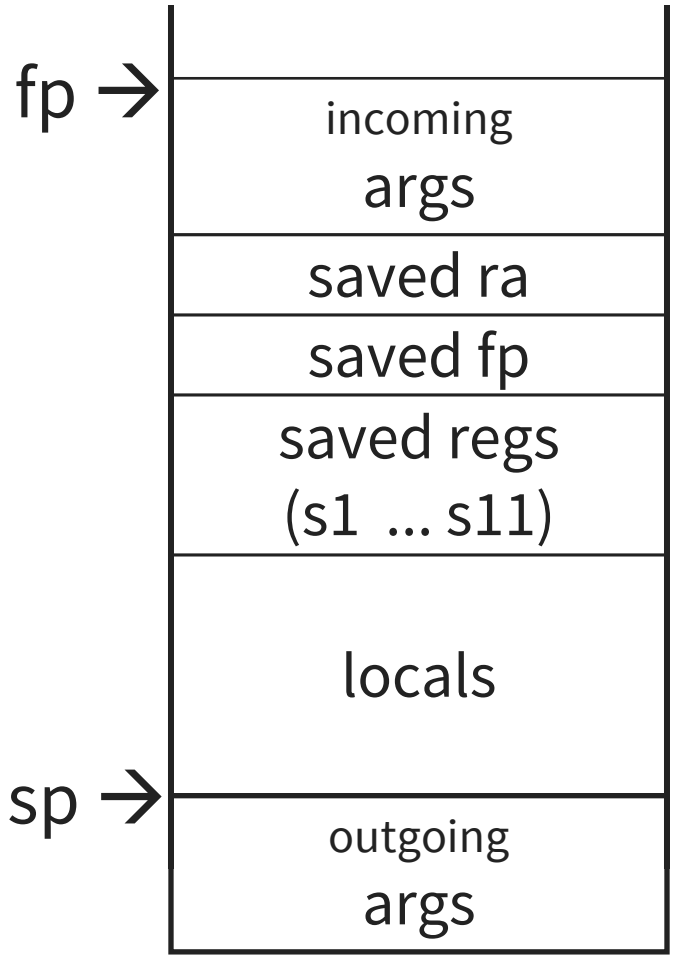
Assume a function uses two callee-save registers.

How do we allocate a stack frame? How large is the stack frame?

What should be stored in the stack frame?

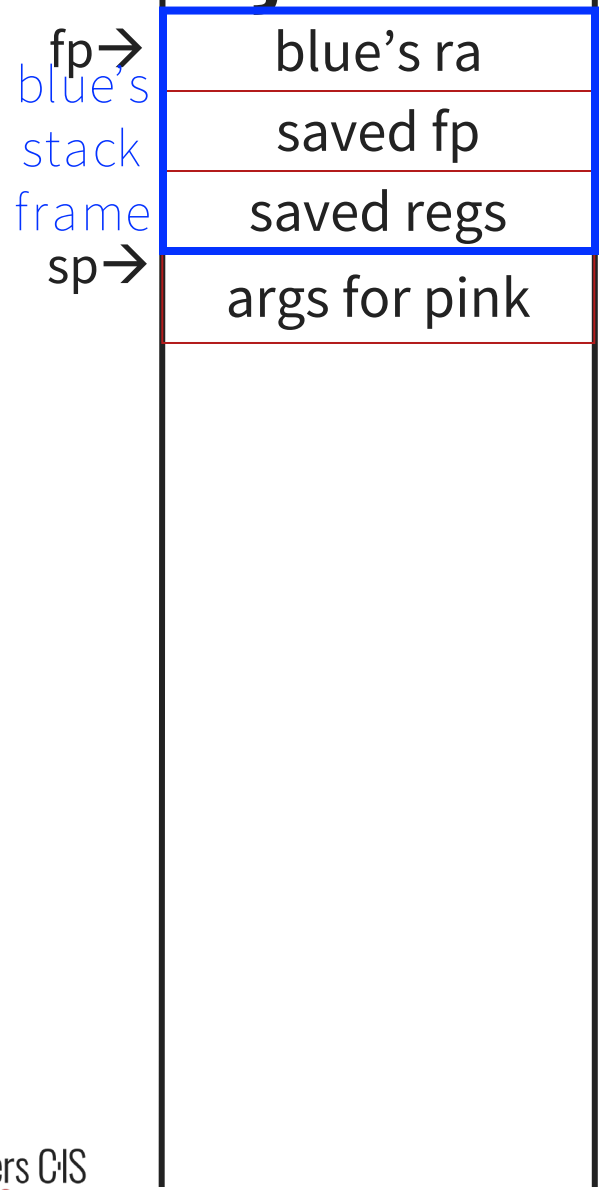
Where should everything be stored?

Frame Layout on Stack



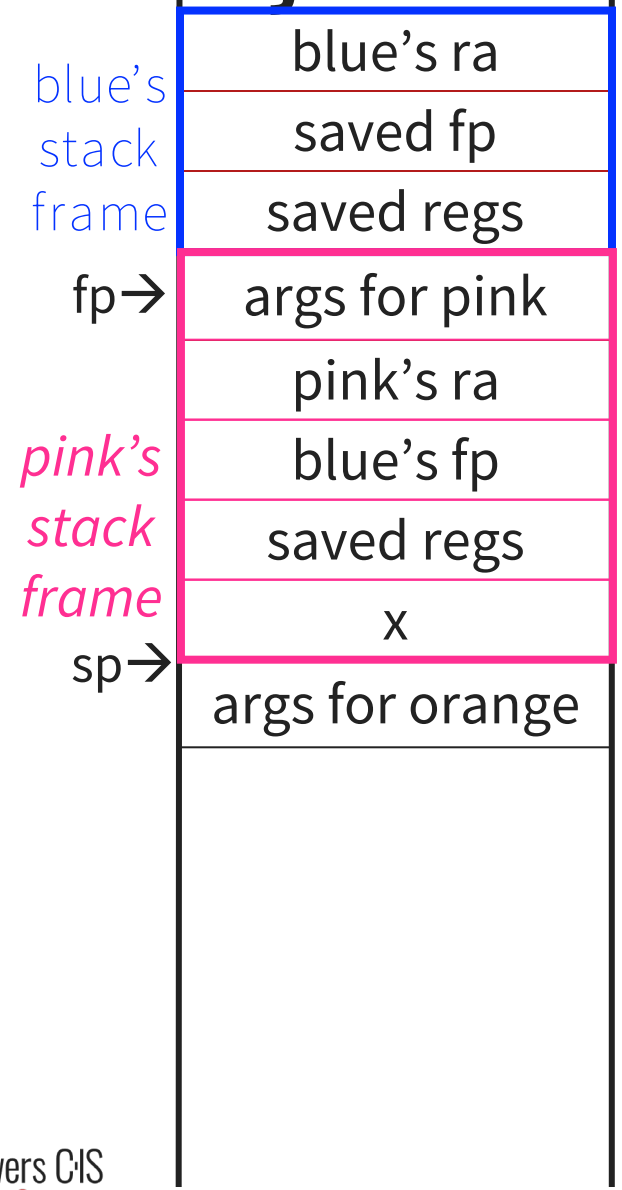

```
ADDI sp, sp, -32        # allocate frame
SD ra, 24(sp)           # save ra
SD fp, 16(sp)           # save old fp
ADDI fp, sp, 24         # set new frame ptr
SD s2, 8(sp)            # save ...
SD s1, 0(sp)            # save ...
...                                    ...
BODY
...                                    ...
LD s1, 0(sp)            # restore ...
LD s2, 8(sp)            # restore ...
LD fp, 16(sp)           # restore old fp
LD ra, 24(sp)           # restore ra
ADDI sp, sp, 32         # dealloc frame
JR ra
```

Frame Layout on Stack



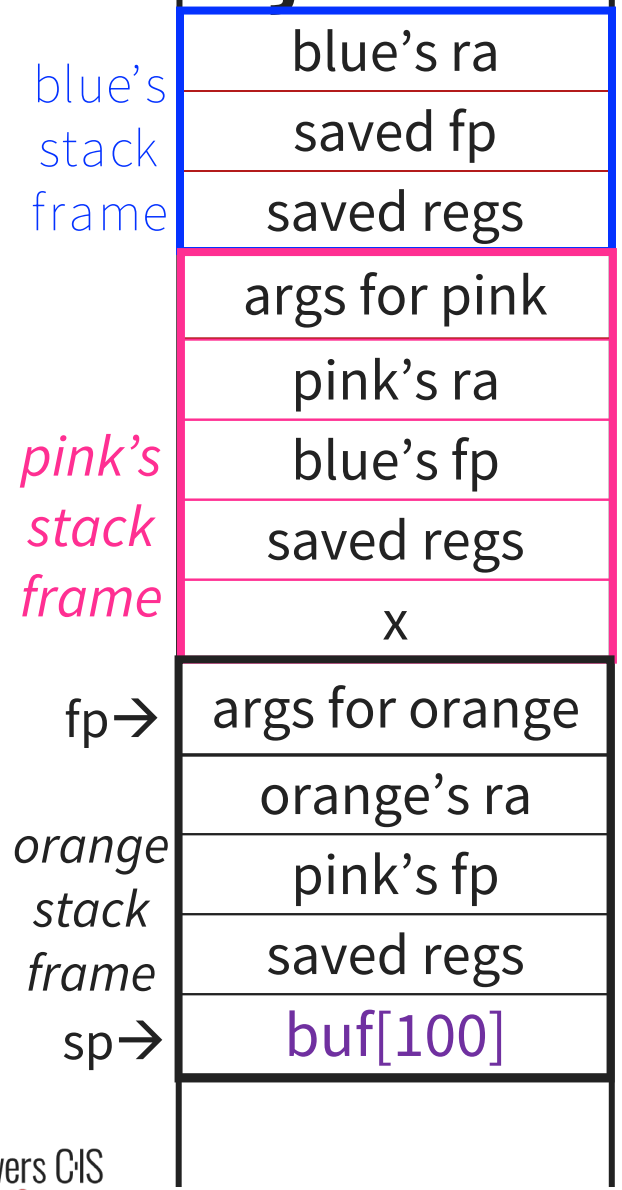

```
blue() {
    pink(0,1,2,3,4,5);
}
```

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);
}
```

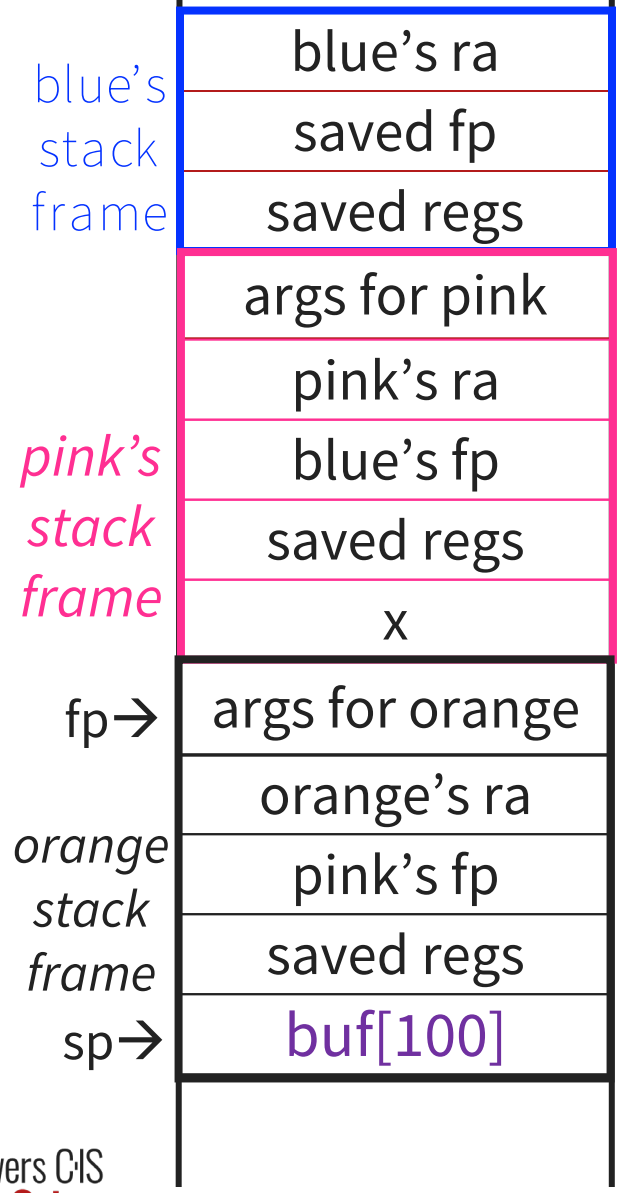
Frame Layout on Stack



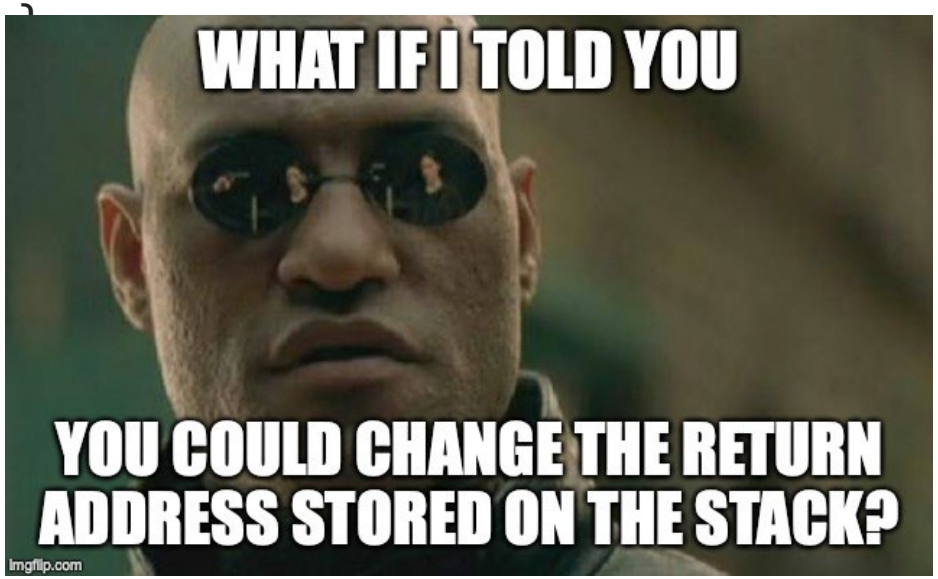

```
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    int x;
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
            char buf[100];
            gets(buf);      // no bounds check!
}
```

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!
```



n 100

Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

- Arguments passed to a routine via x10-x17
- Return values passed back to the caller via x10, x11

Manage Registers

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

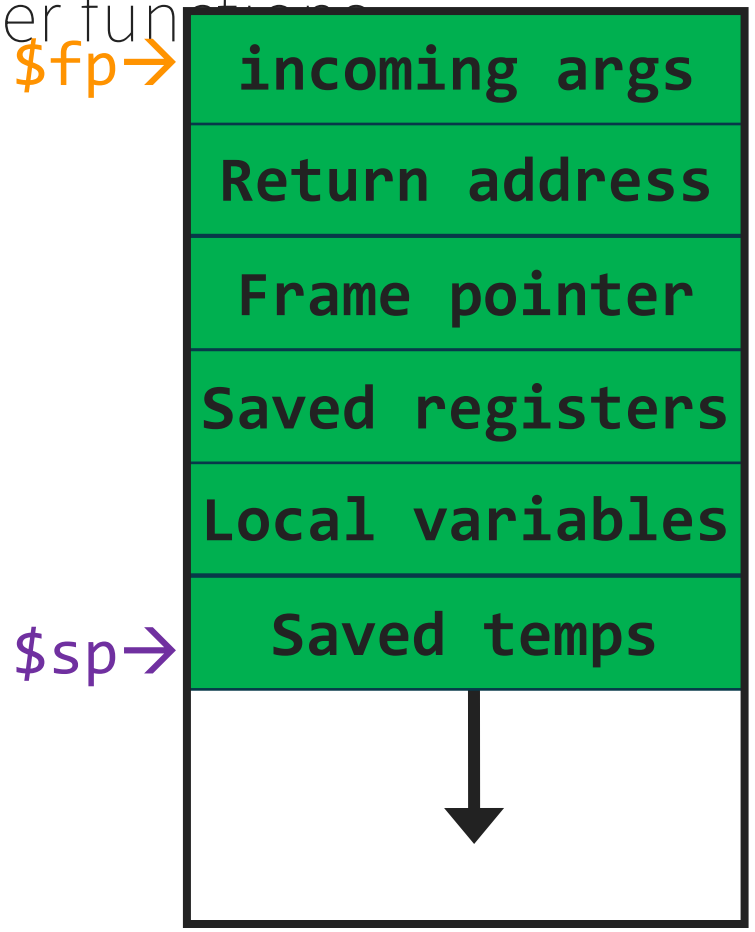
Optimizations & Manipulations?

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?



Calling Convention for Procedure Calls

Transfer Control

- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine

- Arguments passed to a routine via x10-x17
- Return values passed back to the caller via x10, x11

Manage Registers

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

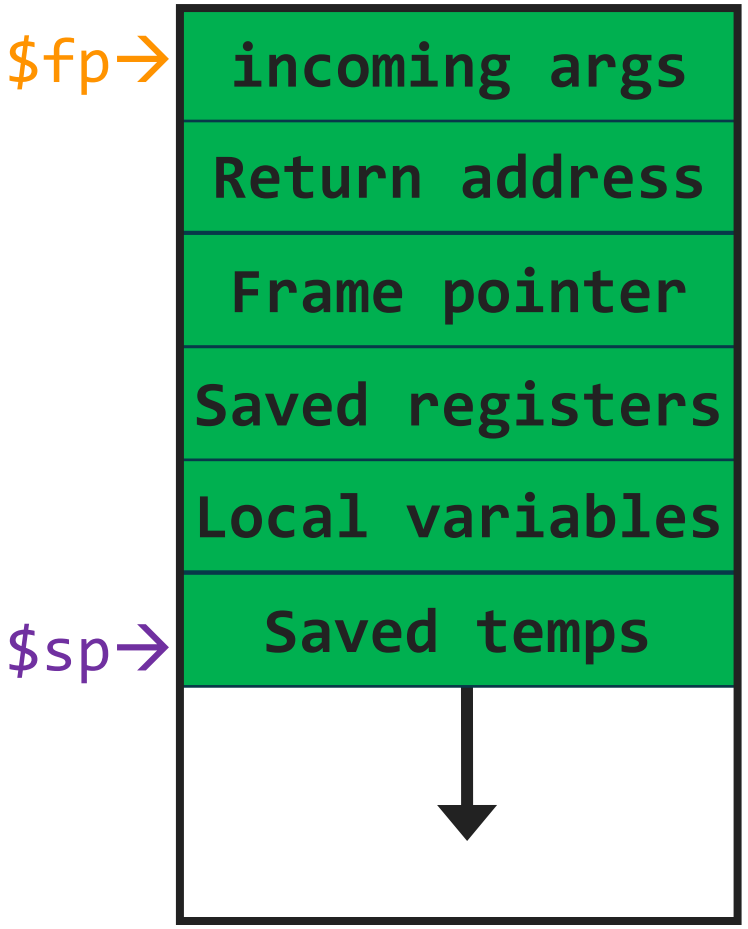
Optimizations & Manipulations?

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?



Putting it all together

1. Body First
2. Determine Stack Frame size
3. Prologue
4. Epilogue

See this in action

1. At the end of this slide deck
2. Textbook (2.13 "A C Sort Example to Put it All Together")
3. Lab Section!

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

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;
}
```

Activity #2: Calling Convention Example: Prologue, Epilogue

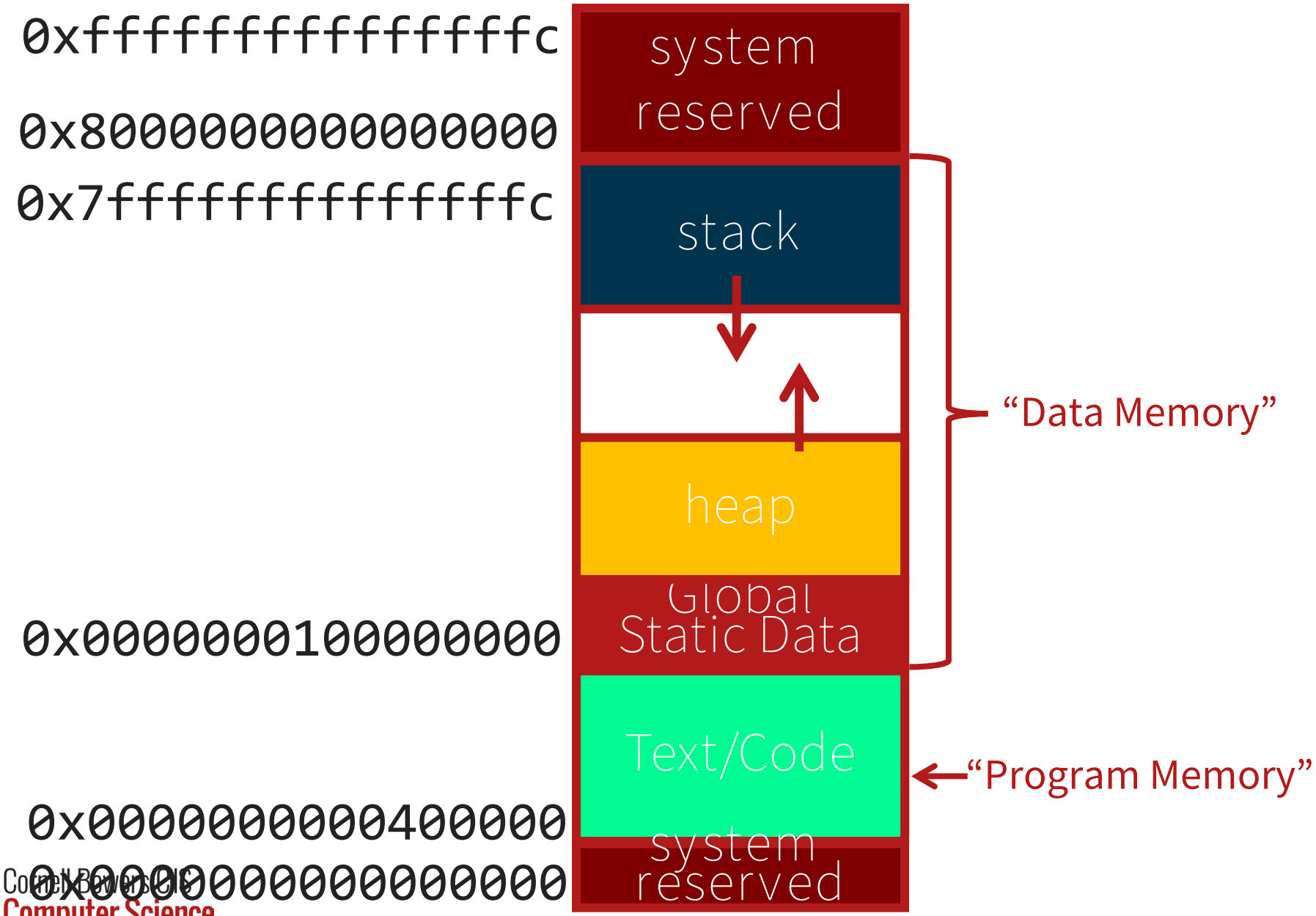
```
# allocate frame
# save ra
# save old fp
# callee save ...
# callee save ...
# set new frame ptr
        ...

        ...
# restore ...
# restore ...
# restore old fp
# restore ra
# dealloc frame
```

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)?

An executing program in memory



Activity #3: Debugging

```
init():     0x0000000000400000
printf(s, …): 0x00000000004002B4
vnorm(a,b):  0x000000000040107C
main(a,b): 0x00000000004010A0
pi:         0x0000000010000000
str1:       0x0000000010000004
```

What func is running?

Who called it?

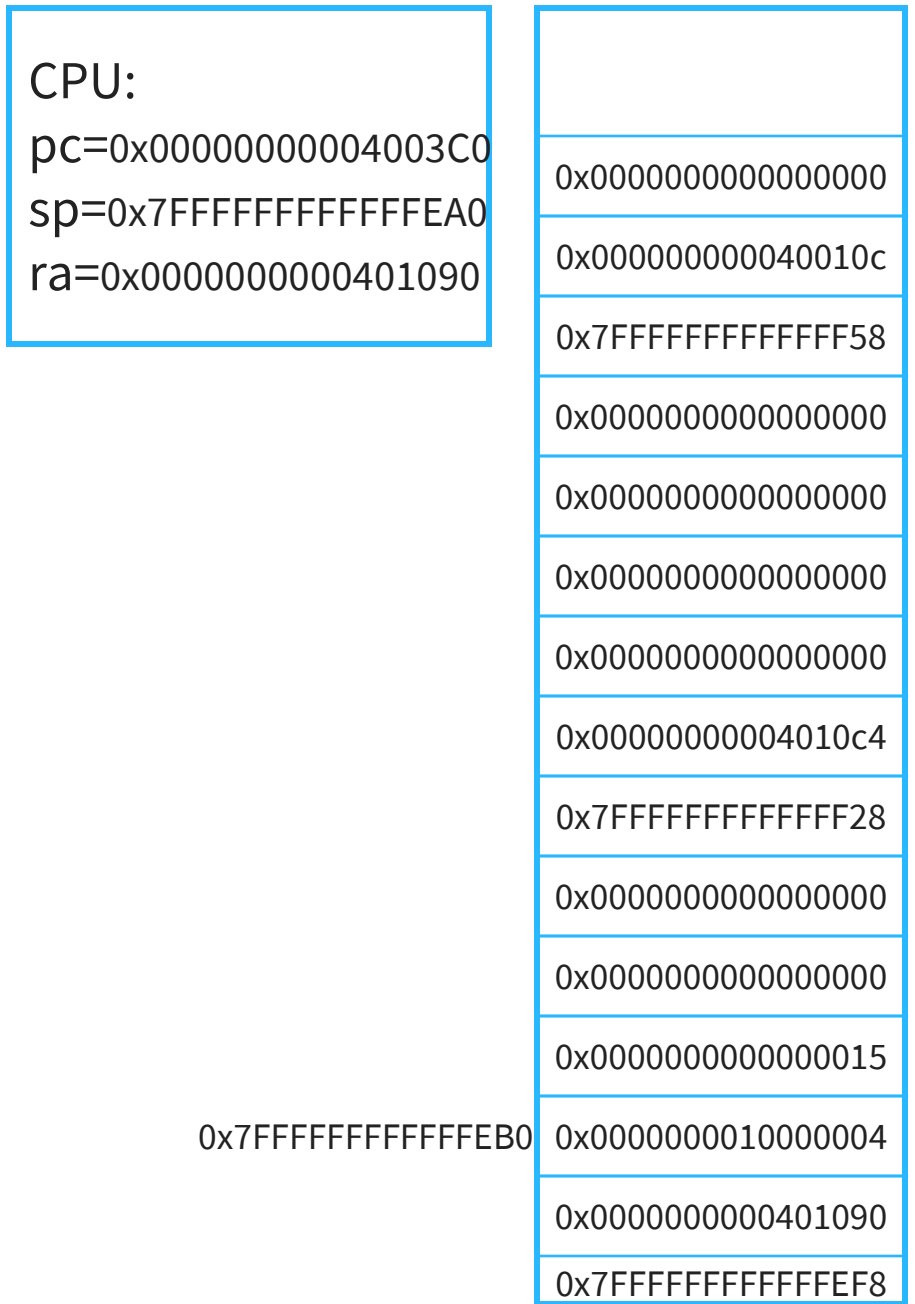
Has it called anything?

Will it?

Args?

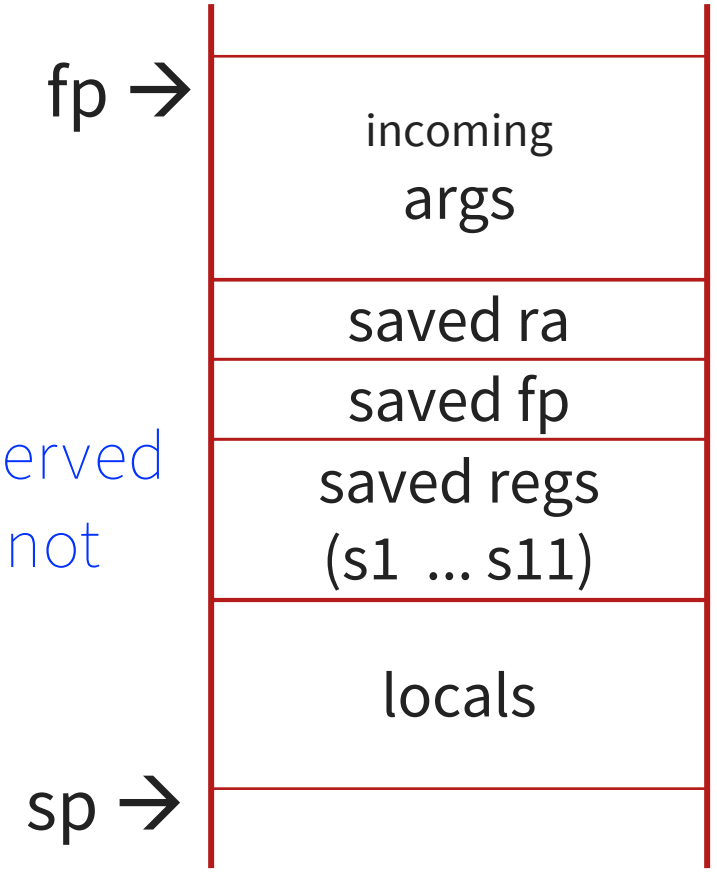
Stack depth?

Call trace?



Cheat Sheet and Mental Model Recap

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



RISC-V Register Conventions

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global data pointer	- -
x4	tp	thread pointer	- -
x5-x7	t0-t2	temporaries	Caller
x8	s0/fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-x11	a0-a1	fn arguments / return values	Caller
x12-x17	a2-a7	function arguments	Caller
x18-x27	s2-s11	saved registers	Callee
x28-x31	t3-t6	temporaries	Caller

Calling Conventions

CS 3410: Computer System Organization and Programming