Roadmap

C:

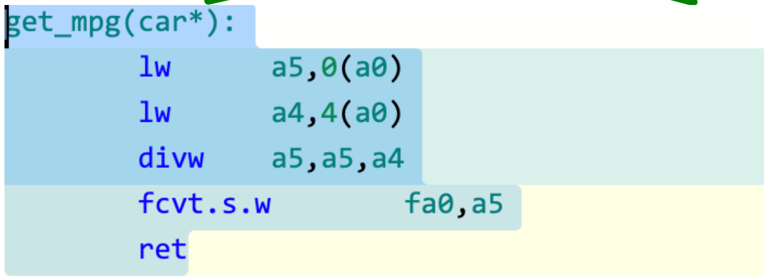
```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Memory & data
Arrays & structs
Integers & floats
RISC V assembly
Procedures & stacks
Executables
Memory & caches
Processor Pipeline
Performance
Parallelism

Assembly language:



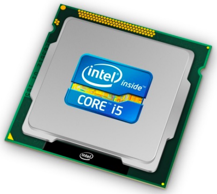
```
get_mpg(car*):
        lw      a5,0(a0)
        lw      a4,4(a0)
        divw    a5,a5,a4
        fcvt.s.w        fa0,a5
        ret
```

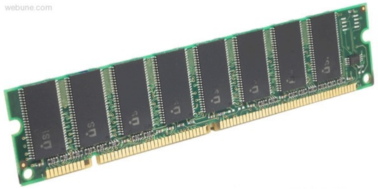
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Computer system:





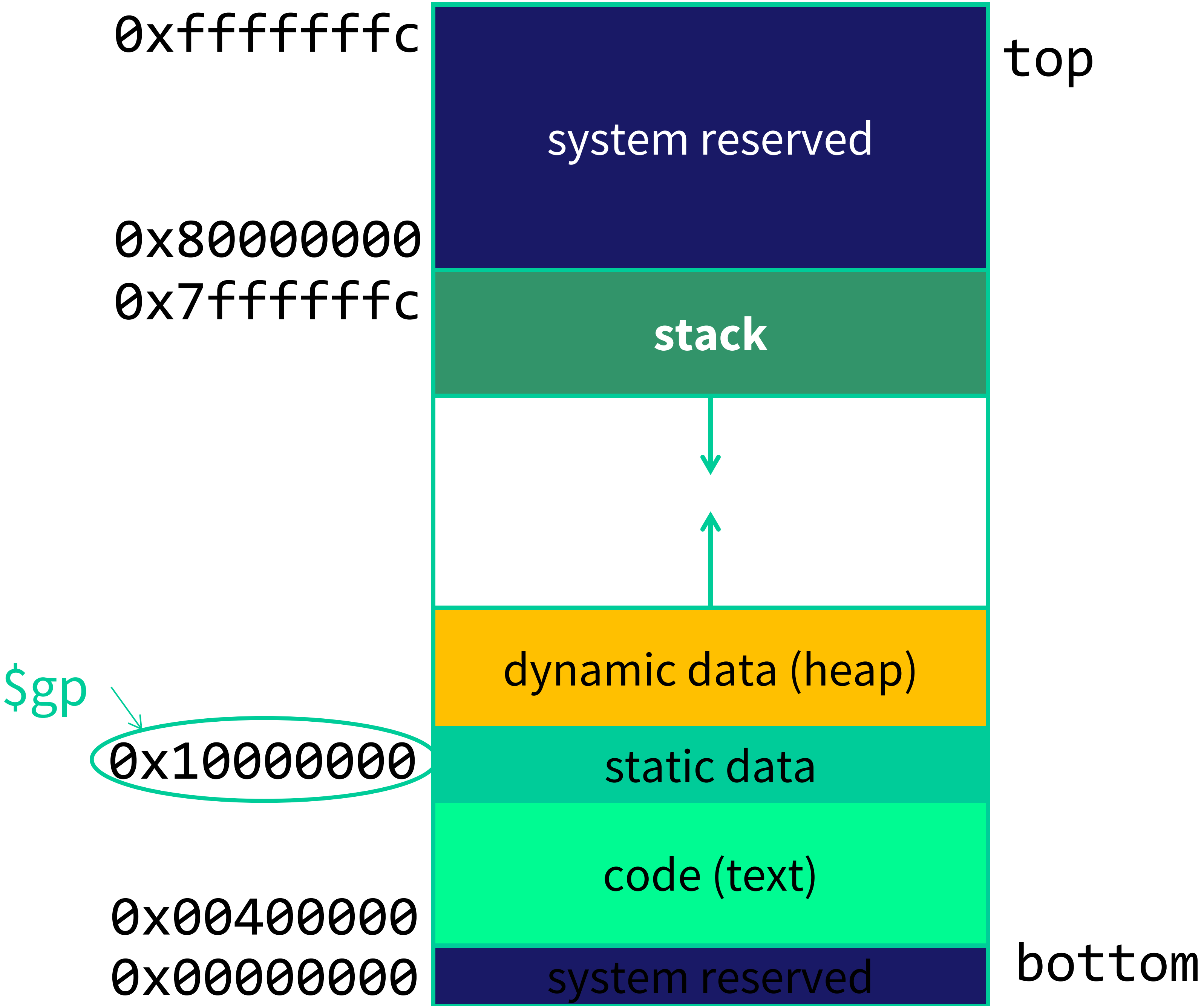
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

So how do we represent instructions?

- **Remember:** Computer only understands 1s and 0s, so assembler string "`add x10,x11,x0`" is meaningless to hardware

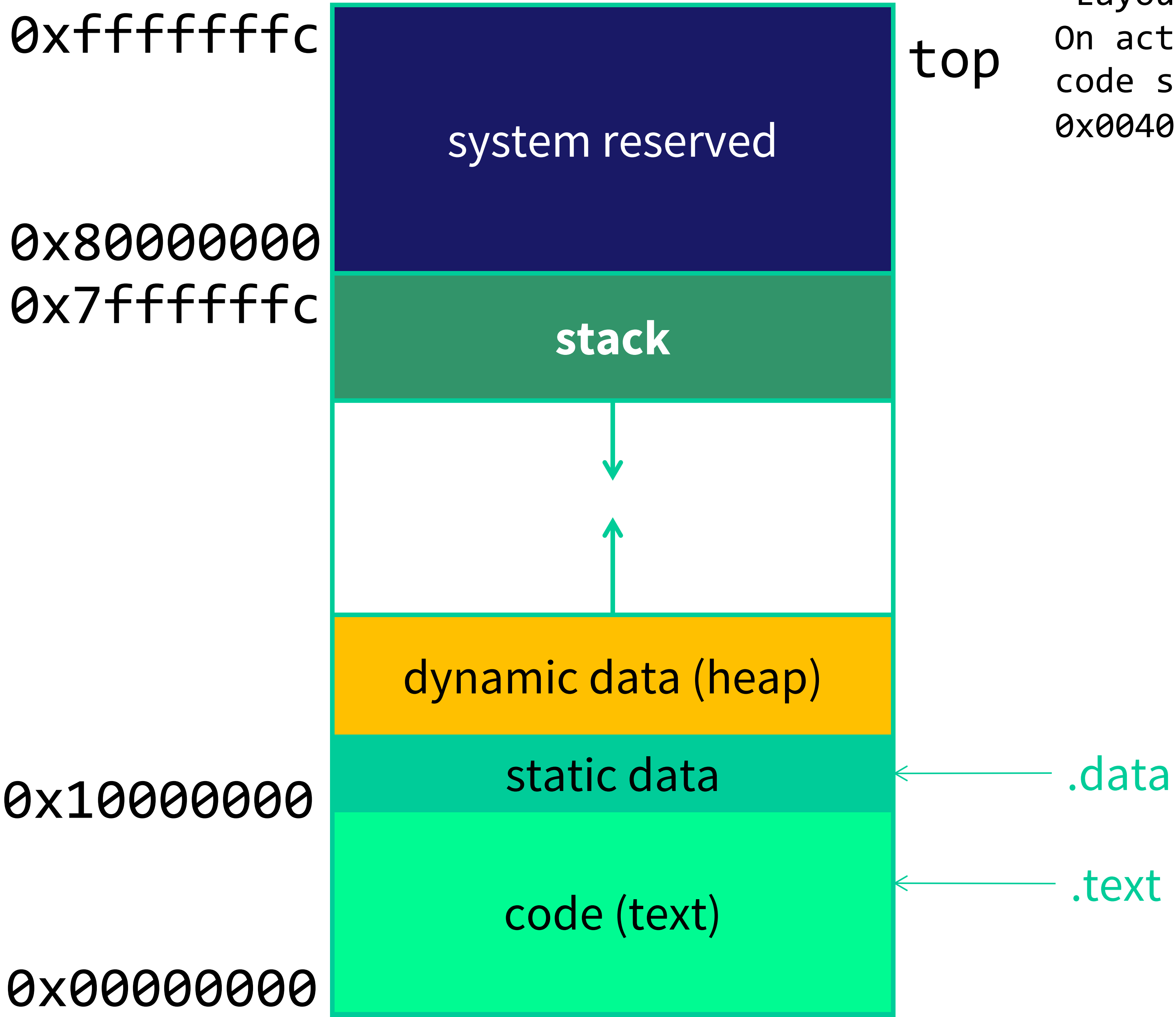
Anatomy of an executing program



Big Idea: Stored-Program Concept

INSTRUCTIONS ARE DATA

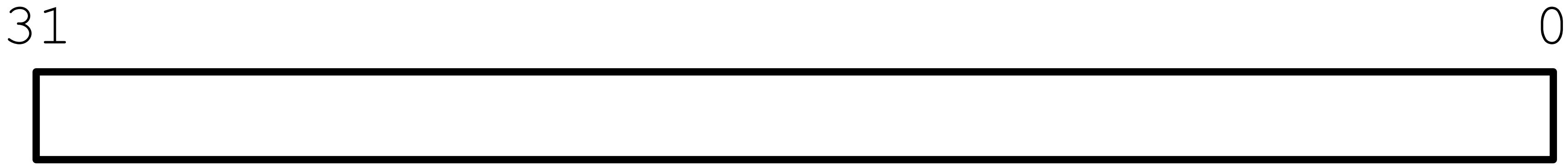
- programs can be stored in memory as numbers
- Before: a number can mean anything
- **Now: make convention for interpreting numbers as instructions**



*Layout in 295
On actual hw
code starts at
0x00400000

Instructions as Numbers

- By convention, RISCV instructions are each 1 word = 4 bytes = 32 bits



- Divide the 32 bits of an instruction into "fields"
  - regular field sizes → simpler hardware
  - will need some variation….

Assembler demo

Jump Table Demo

```
long switch_ex
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

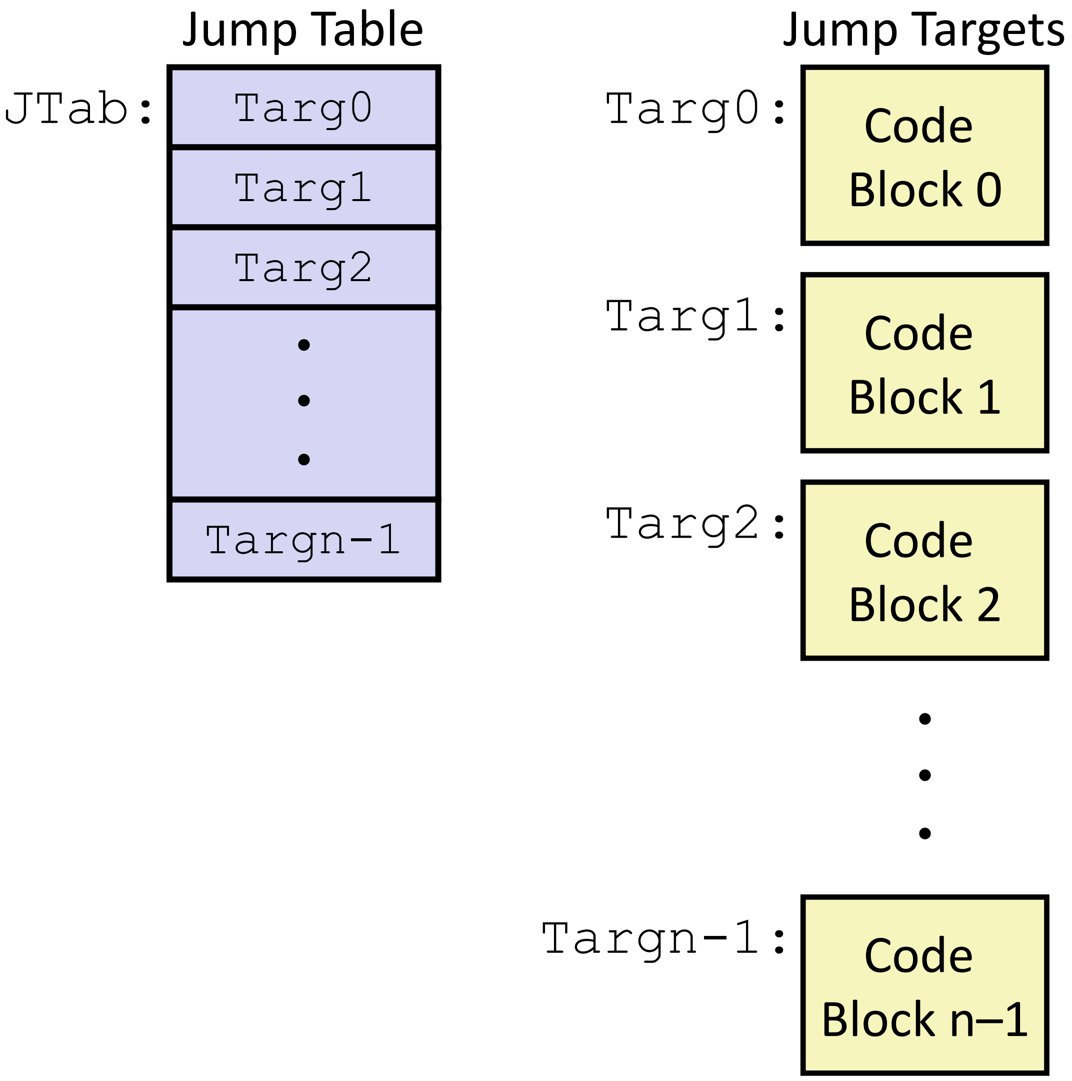
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - *Jump table*
  - *Indirect jump instruction*

Jump Table Structure

Switch Form

```
switch (x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



Jump Table

JTab:

- Targ0
- Targ1
- Targ2
- •
- •
- •
- Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block n–1

Approximate Translation

```
target = JTab[x];
goto target;
```

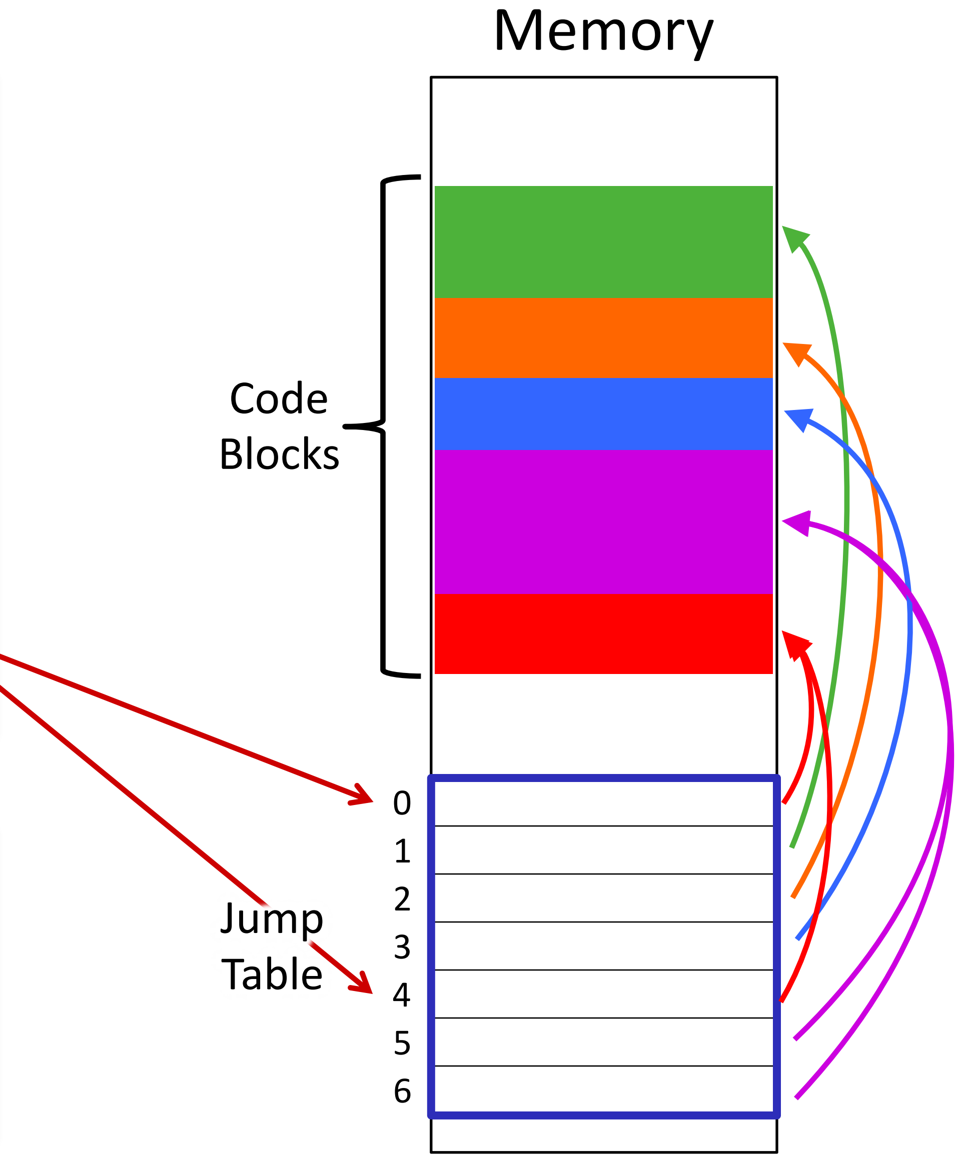
Jump Table Structure

C code:

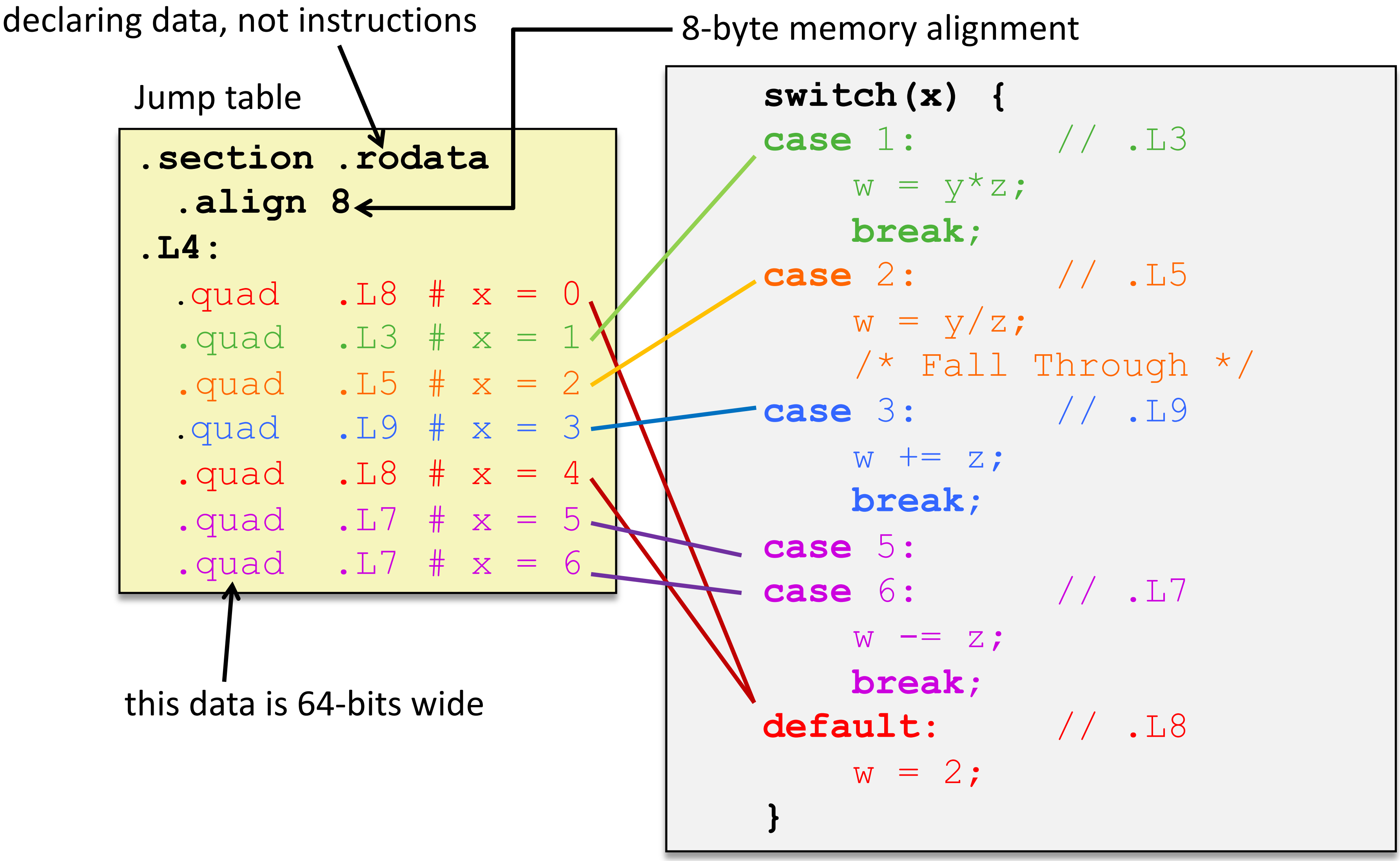
```
switch (x) {
  case 1: <some code>
          break;
  case 2: <some code>
  case 3: <some code>
          break;
  case 5:
  case 6: <some code>
          break;
  default: <some code>
}
```

Use the jump table when x ≤ 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Jump Table



declaring data, not instructions

8-byte memory alignment

Jump table

```
.section .rodata
  .align 8
.L4:
  .quad   .L8 # x = 0
  .quad   .L3 # x = 1
  .quad   .L5 # x = 2
  .quad   .L9 # x = 3
  .quad   .L8 # x = 4
  .quad   .L7 # x = 5
  .quad   .L7 # x = 6
```

this data is 64-bits wide

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

SMC Demo

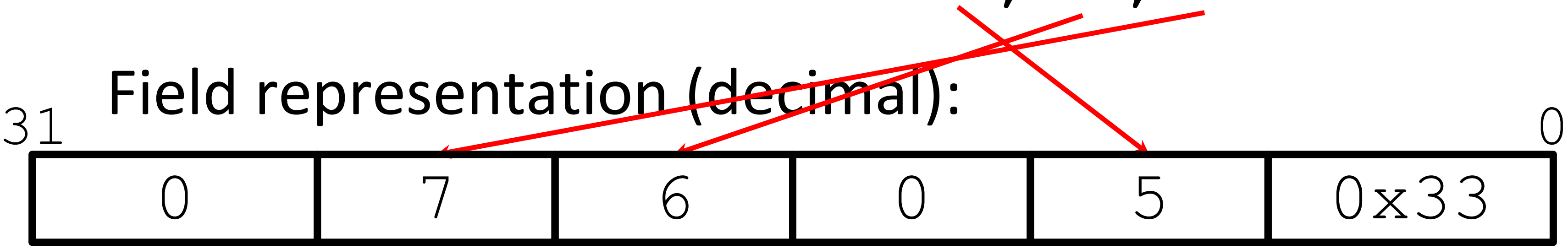
The 6 Instruction Formats

- R-Format: instructions using 3 register inputs
  - `add`, `xor`, `mul` —arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - `addi`, `lw`, `jalr`, `slli`
- S-Format: store instructions: `sw`, `sb`
- SB-Format: branch instructions: `beq`, `bge`
- U-Format: instructions with upper immediates
  - `lui`, `auipc` —upper immediate is 20-bits
- UJ-Format: the jump instruction: `jal`

R-Format Example

- RISCV Instruction: `add x5,x6,x7`

Field representation (decimal):



31					0
0	7	6	0	5	0x33

Field representation (binary):

31					0
0000000	00111	00110	000	00101	0110011

two

hex representation: 0x 0073 02B3

decimal representation: 7,537,331

Called a Machine Language Instruction

I-Format Instructions (3/4)

31 0

imm[11:0]	rs1	func3	rd	opcode

- opcode (7): uniquely specifies the instruction
- rs1 (5): specifies a register operand
- rd (5): specifies destination register that receives result of computation

I-Format Instructions (4/4)

31 0

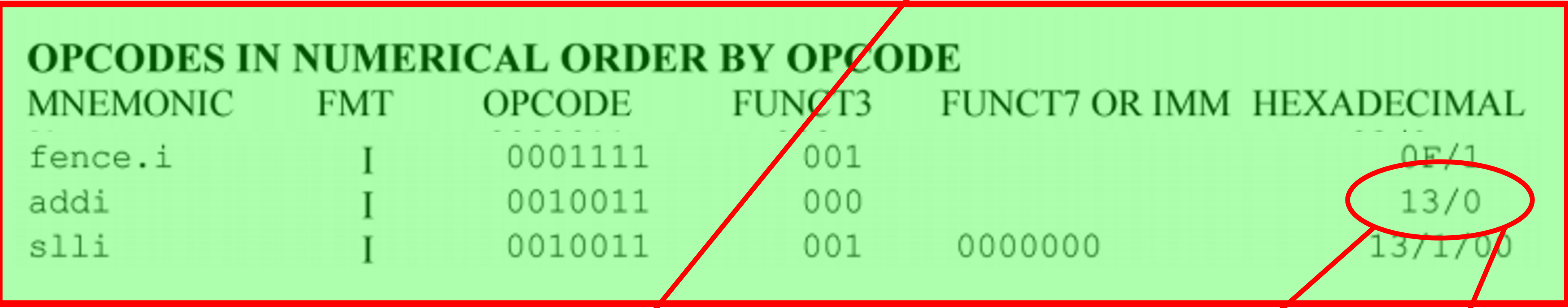
imm[11:0]	rs1	func3	rd	opcode

- immediate (12): 12 bit number
  - All computations done in words, so 12-bit immediate must be *extended* to 32 bits
  - always **sign-extended** to 32-bits before use in an arithmetic operation

- Can represent 2^{12} different immediates
  - imm[11:0] can hold values in range $[-2^{11}, +2^{11})$

I-Format Example (1/2)

```
addi x15,x1,-50
```



OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00

```
rd  = x15
rs1 = x1
```

31				0
111111001110	00001	0	01111	0x13
imm[11:0]	rs1	func3	rd	opcode

Load Instructions are also I-Type

31				0
imm[11:0]	rs1	func3	rd	opcode
offset[11:0]	base	width	dst	LOAD

- The 12-bit signed immediate is added to the base address in register `rs1` to form the memory address
  - This is very similar to the add-immediate operation but used to create address, not to create final result
- Value loaded from memory is stored in `rd`

S-Format Used for Stores

- Store needs to read two registers, `rs1` for base memory address, and `rs2` for data to be stored, as well as need immediate offset!
- Can't have both `rs2` and immediate in same place as other instructions!
- Note: stores don't write a value to the register file, no `rd`!
- RISC-V design decision is **move low 5 bits of immediate** to where **`rd`** field was in other instructions – keep `rs1`/`rs2` fields in same place
- register names more critical than immediate bits in hardware design

31 0

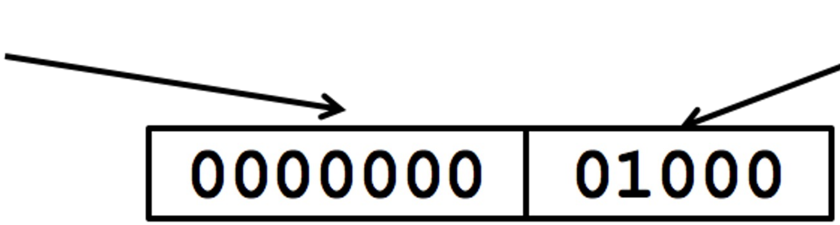
imm[11:5]	rs2	rs1	**func3**	**imm[4:0]**	opcode

S-Format Example

```
sw x14, 8(x2)
```

31 0

imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
00000000	01110	00010	010	**01000**	0100011
off[11:5] = 0	rs2=14	rs1=2	SW	**off[4:0] = 8**	STORE



0000000	**01000**

combined 12-bit offset = 8

RISC-V B-Format for Branches

- B-format is mostly same as S-Format, with two register sources (`rs1`/`rs2`) and a 12-bit immediate
- But now immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

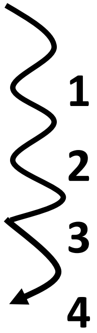
31					0
imm[12|10:5]	rs2	rs1	func3	**imm[4:1|11]**	opcode
7	5	5	3	**5**	7

Branch Example (1/2)

- RISCV Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  <target instr>
```

Start counting from instruction AFTER the branch



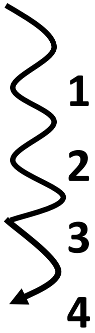
- Branch offset = 4×32-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

Branch Example (1/2)

- RISCV Code:

```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End:  <target instr>
```

Start counting from branch

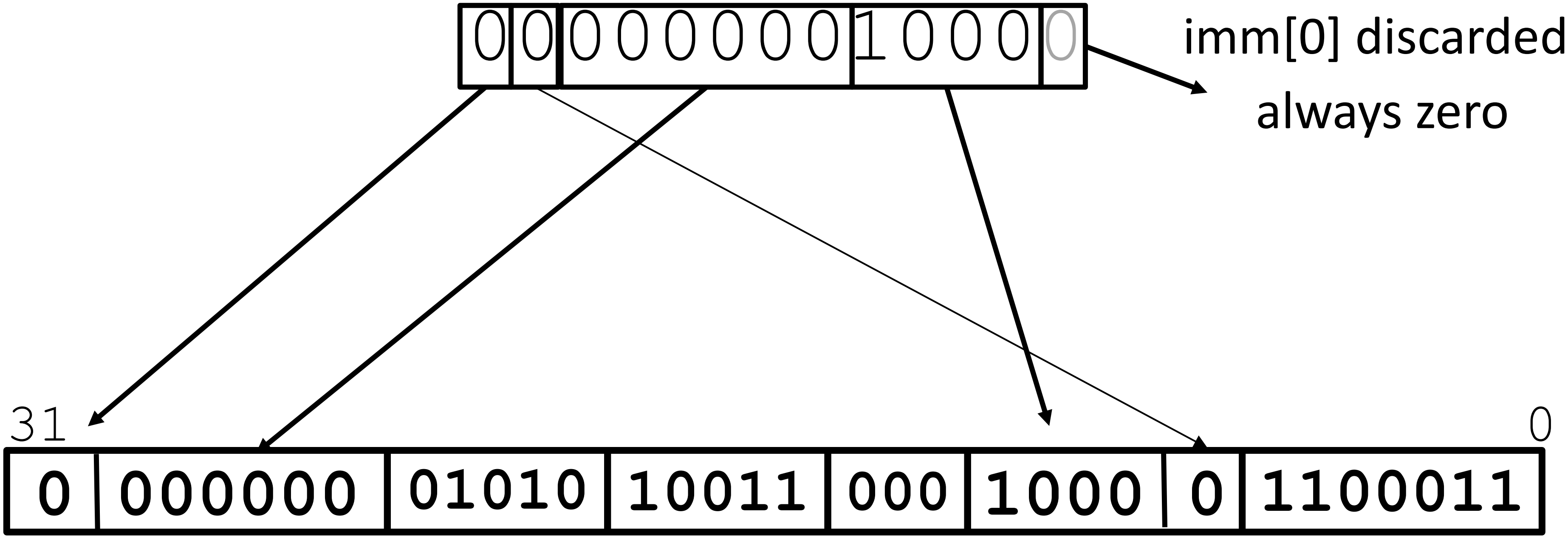


31 7	5	5	3	5	7 0
???????	01010	10011	000	?????	1100011
	rs2=10	rs1=19	BEQ		BRANCH

Branch Example (1/2)

beq x19,x10,offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



0	0	0 0 0 0 0 0	1 0 0 0	0

imm[0] discarded, always zero

31 0

0	000000	01010	10011	000	1000	0	1100011

imm[12|10:5]

imm[4:1|11]

U-Format for “Upper Immediate” instructions

31		0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI/AUIPC

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, `rd`
- Used for two instructions
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- `lui` writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits
- Together with an `addi` to set low 12 bits, can create any 32-bit value in a register using two instructions (`lui`/`addi`).

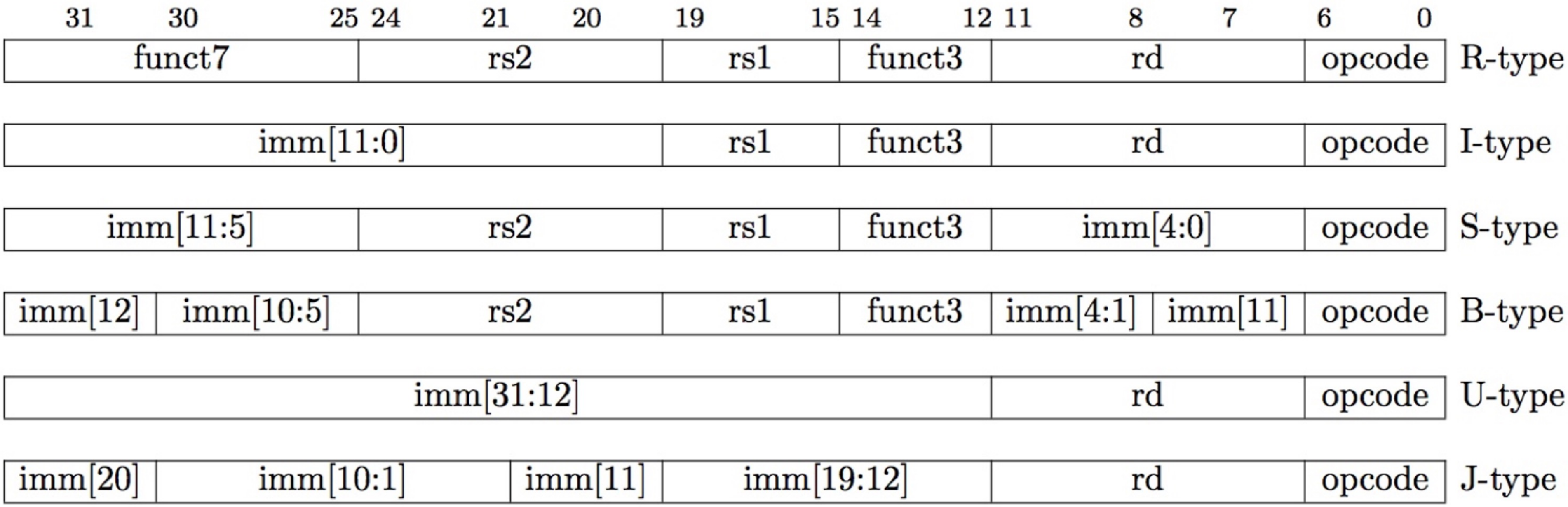
```
lui x10, 0x87654                    # x10 =
0x87654000
addi x10, x10, 0x321   # x10 = 0x87654321
```

jalr

31				0
imm[11:0]	rs1	func3	rd	opcode
offset	base	0	dest	JALR

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute address
lui x1, <hi 20 bits>
jalr ra, x1, <lo 12 bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi 20 bits>
jalr x0, x1, <lo 12 bits>
```

Summary of RISC-V Instruction Formats



<table>
<tr><th>31</th><th>30 25</th><th>24 21</th><th>20</th><th>19 15</th><th>14 12</th><th>11 8</th><th>7</th><th>6 0</th><th></th></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-type</td></tr>
</table>

Agenda

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

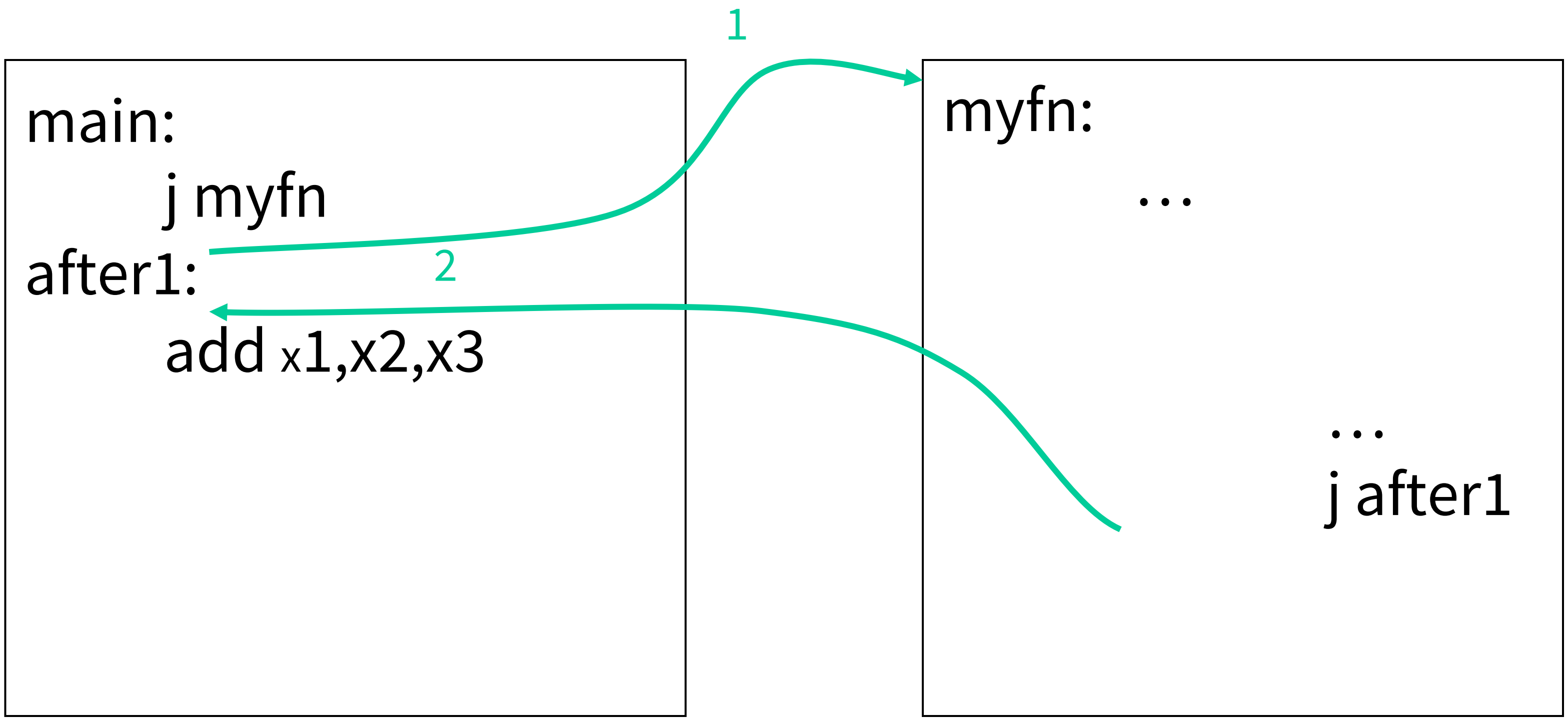
Calling Convention for Procedure Calls

- 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

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



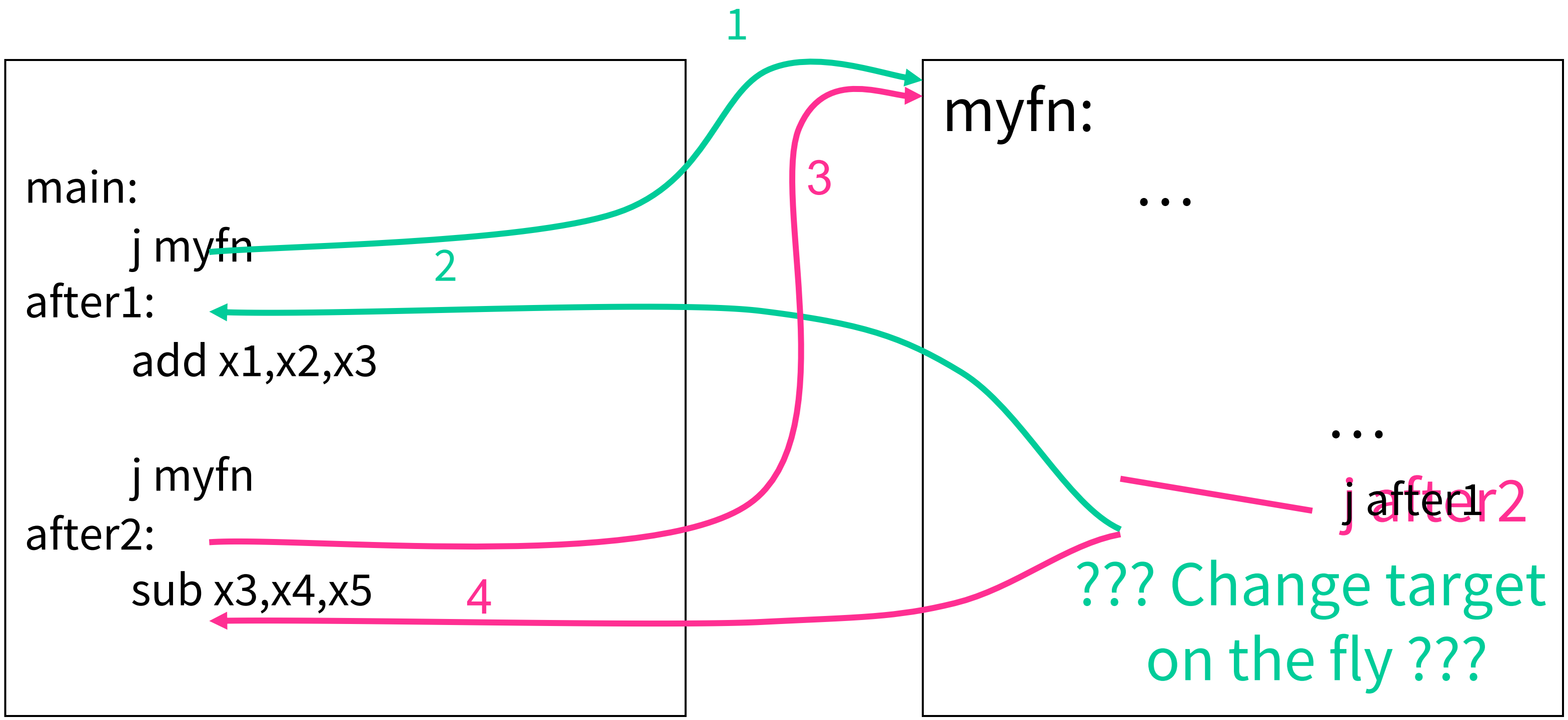
```
main:
        j myfn
after1:
        add x1,x2,x3
```

```
myfn:
        …

                …
                j after1
```

Jumps to the callee

Jumps back

Jumps are not enough



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

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

```
myfn:
        …

        …
        j after1
        j after2
```

??? Change target on the fly ???

Jumps to the callee

Jumps back

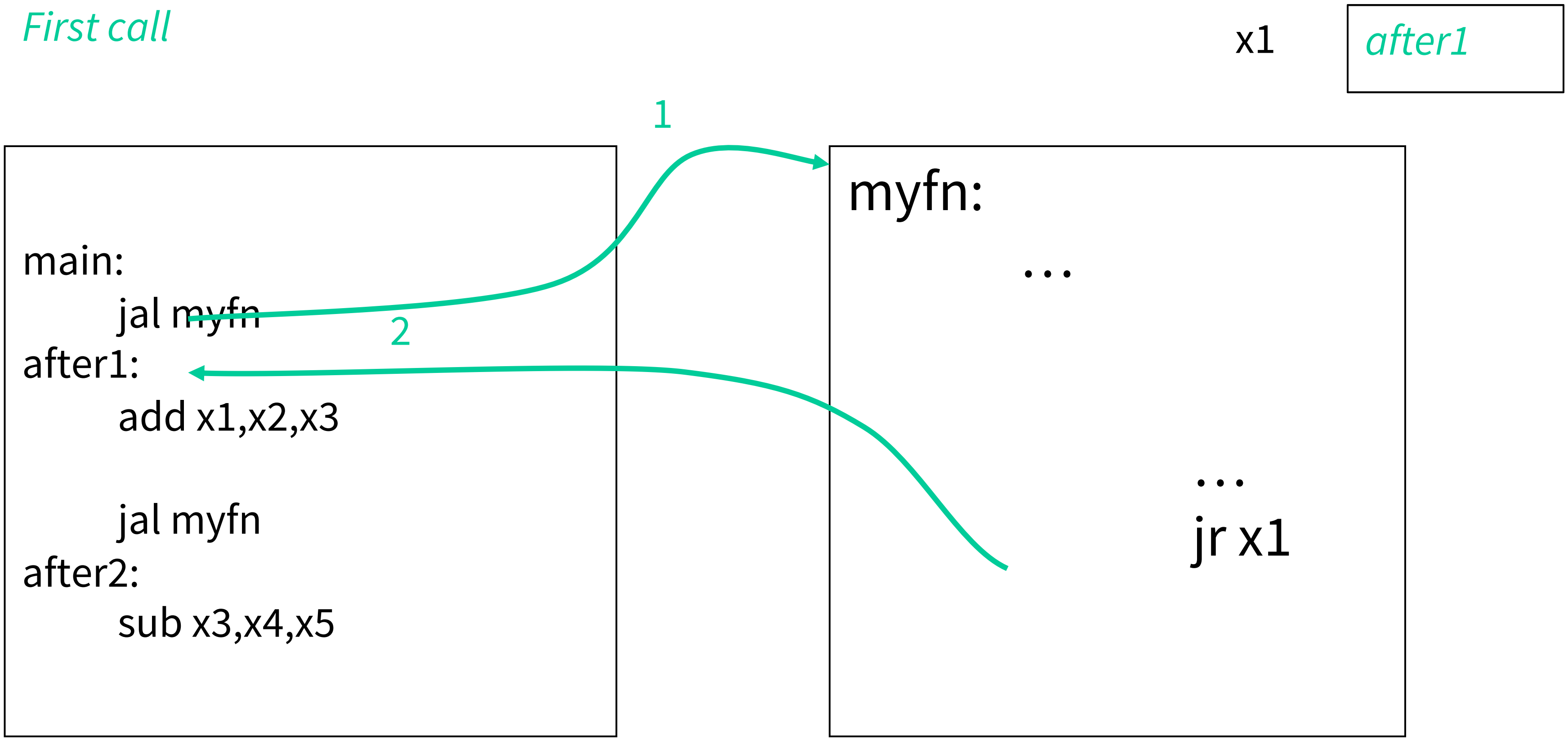
What about multiple sites?

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*

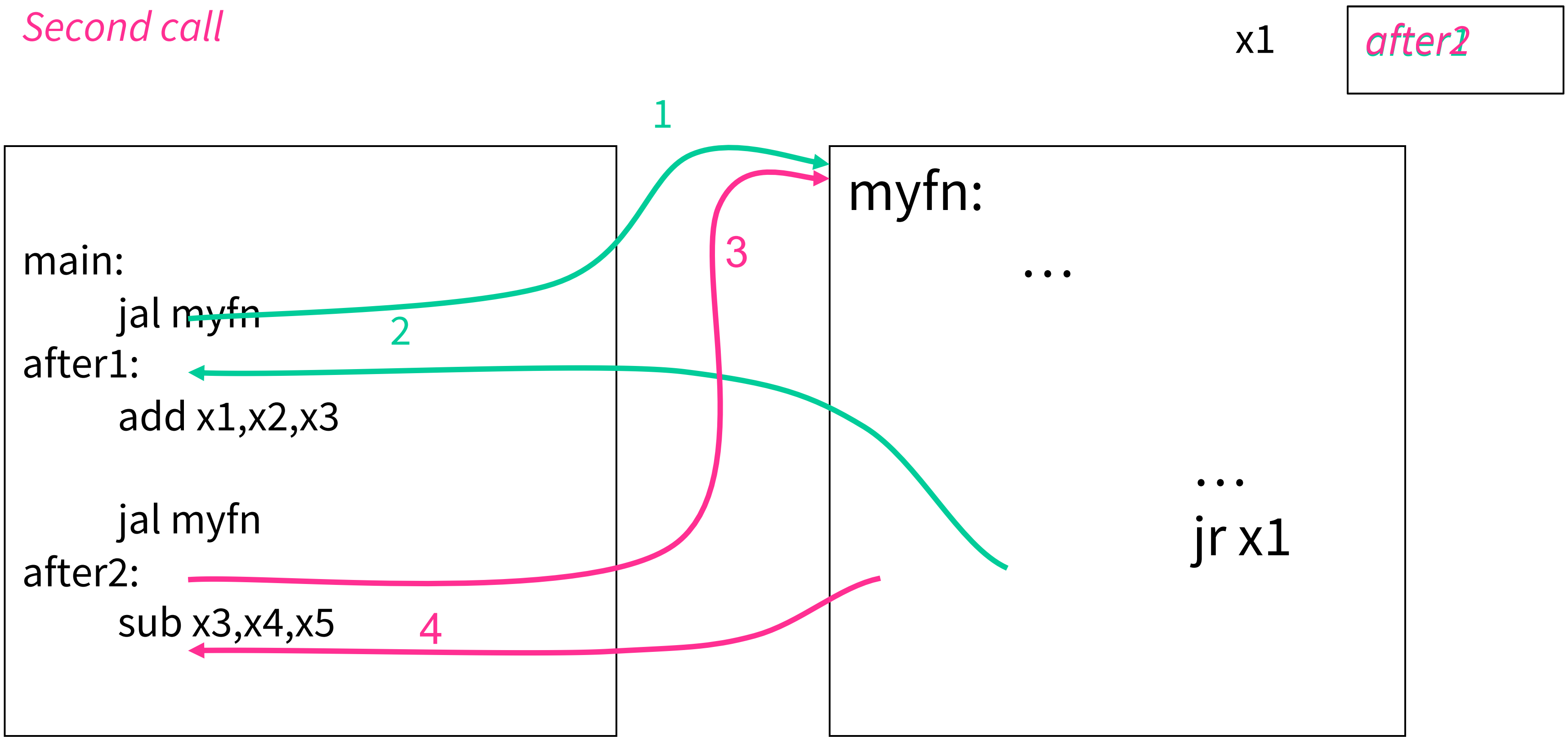
```
main:
        jal myfn
after1:
        add x1,x2,x3

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

```
myfn:
        …

                …
                jr x1
```

JAL saves the PC in register $31

Subroutine returns by jumping to $31

Jump-and-Link / Jump Register



Second call

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

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

```
myfn:
        …

                …
                jr x1
```

JAL saves the PC in register x1

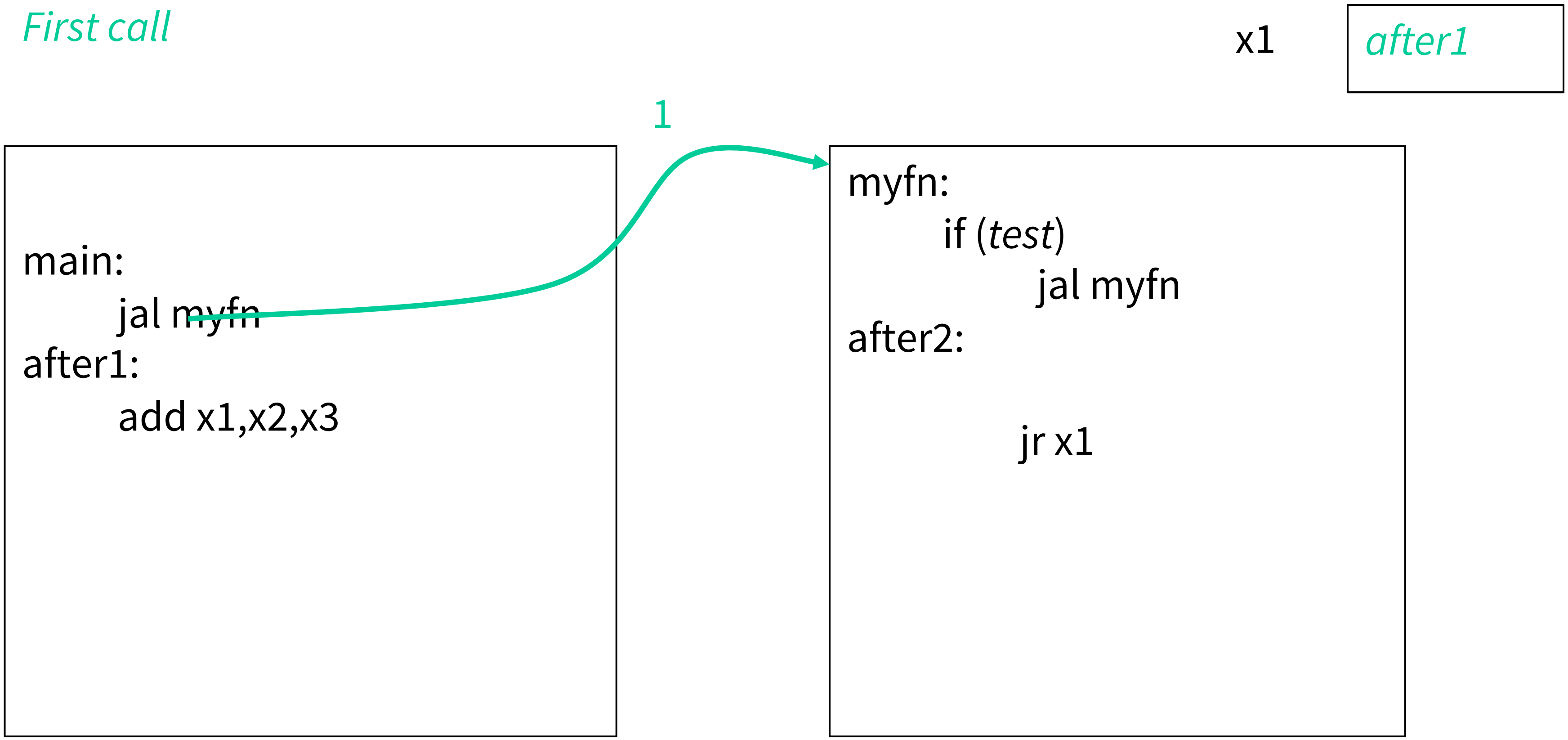
Subroutine returns by jumping to x1

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*

1

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

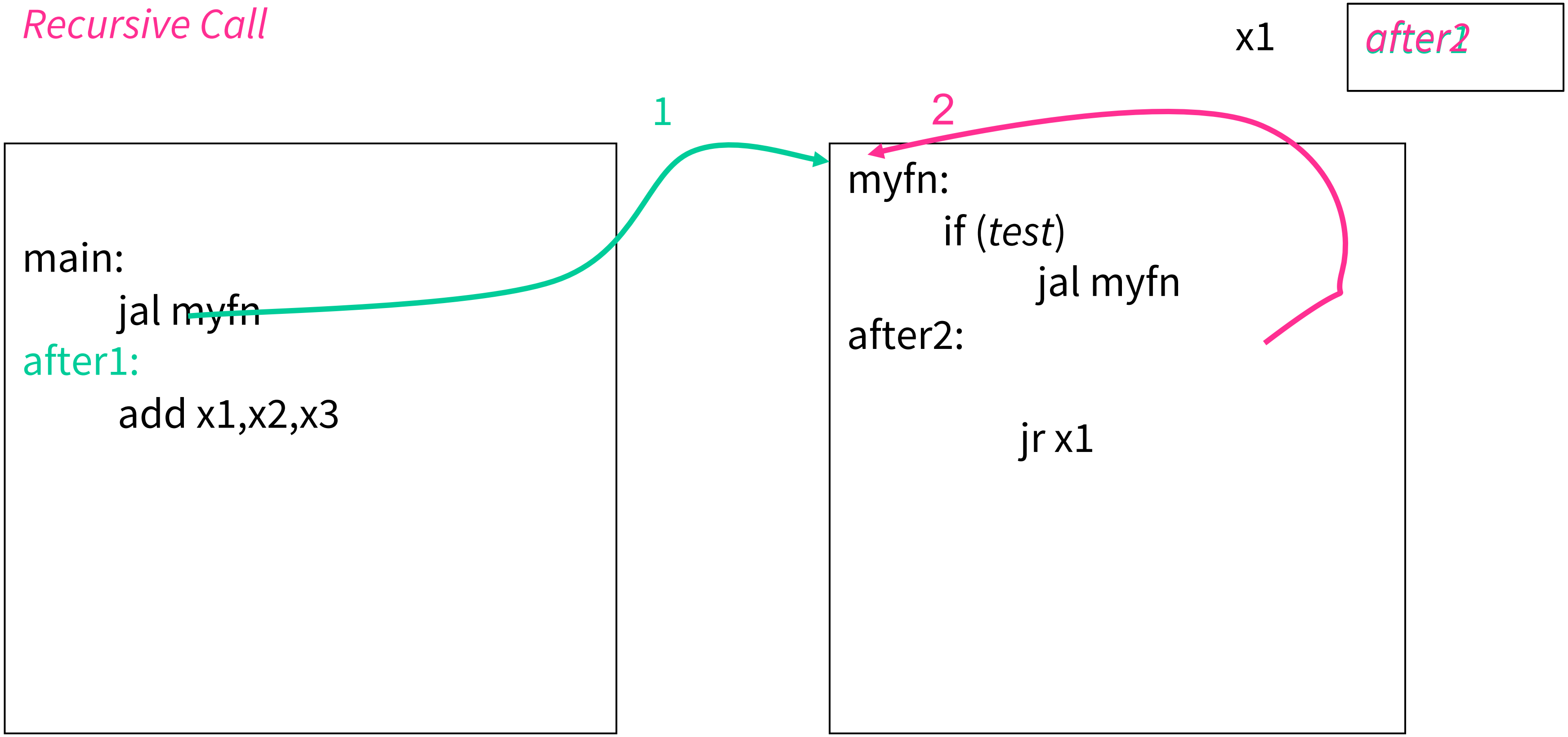
```
myfn:
        if (test)
                jal myfn
after2:

               jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Recursive Call



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

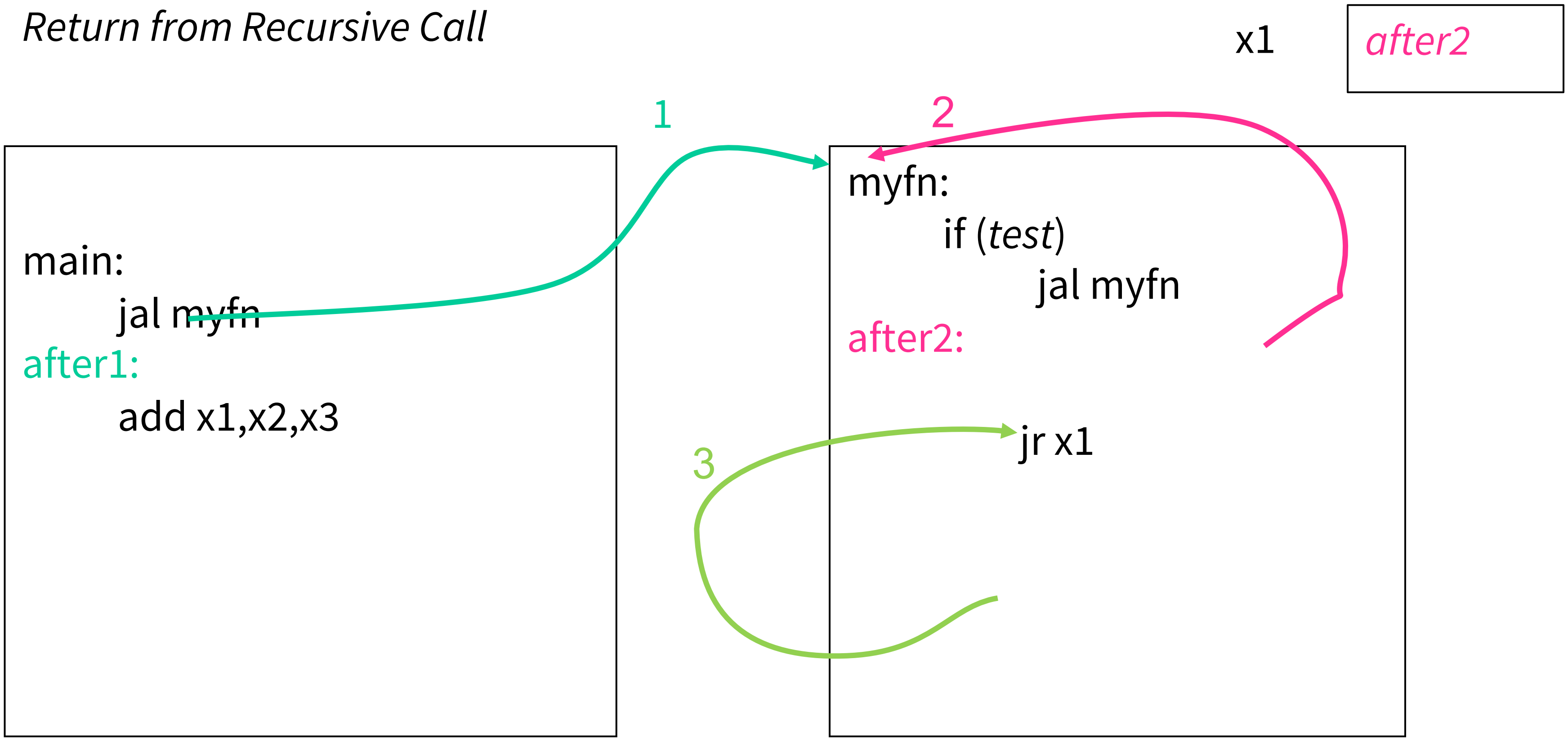
```
myfn:
        if (test)
                jal myfn
after2:

                jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Recursive Call



x1 *after2*

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

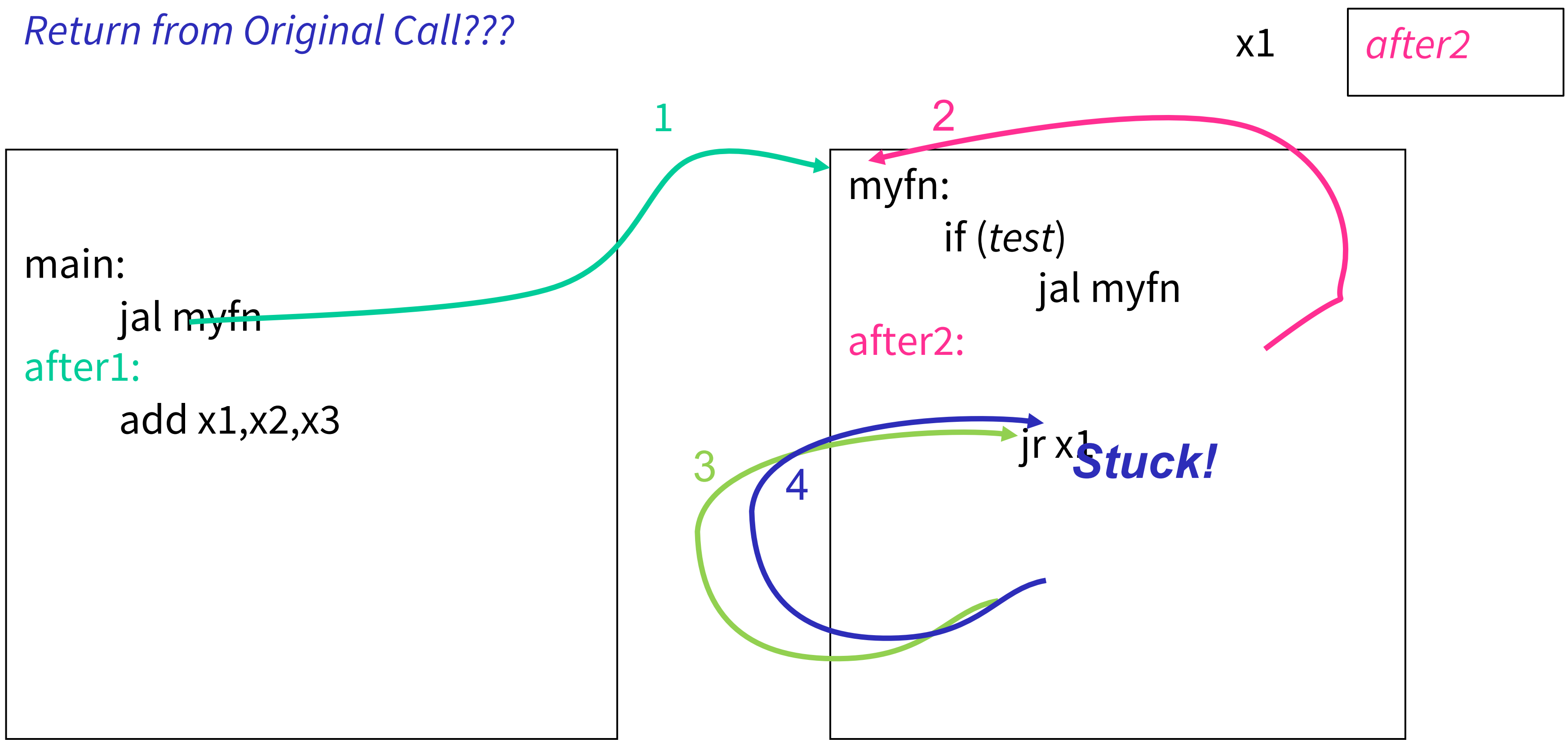
```
myfn:
        if (test)
                jal myfn
after2:

        jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???



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

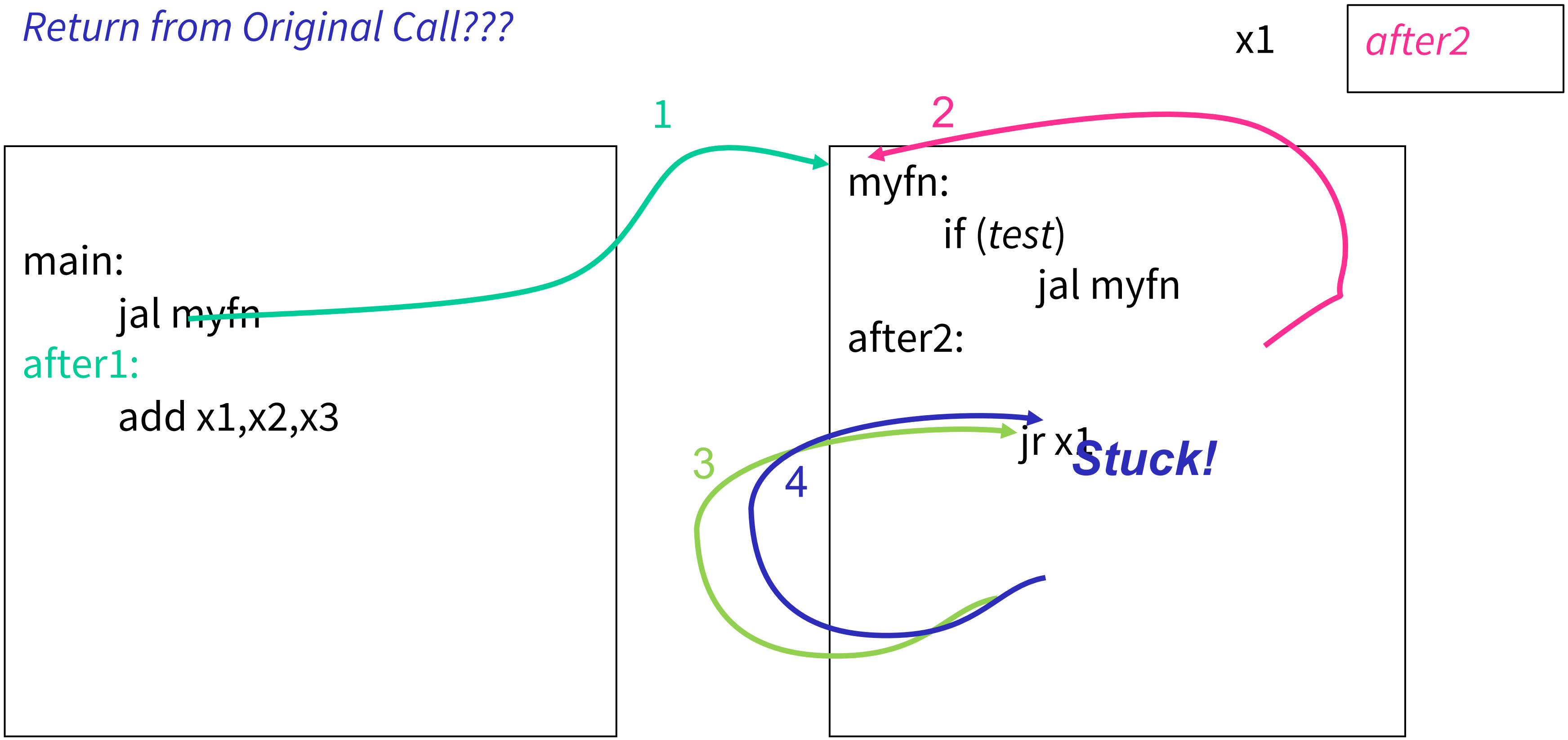
```
myfn:
        if (test)
                jal myfn
after2:

        jr x1
```

Problems with recursion:

- overwrites contents of x1

JAL / JR for Recursion?

Return from Original Call???



Problems with recursion:

overwrites contents of x1

Need a way to save and restore register contents

Agenda

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

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

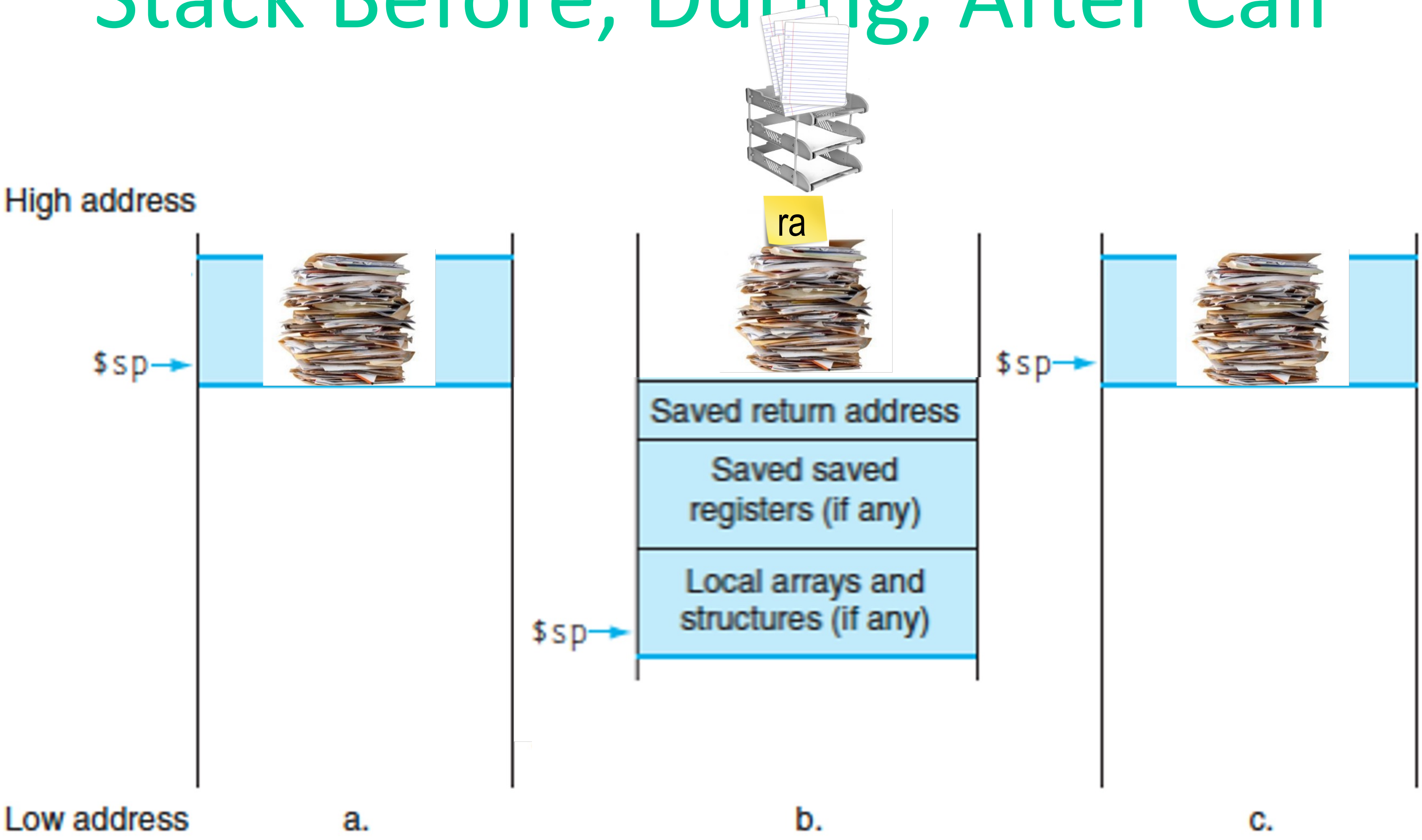
A stack pointer (sp) keeps track of the top of the stack

- dedicated register (x2) on the RISC-V

Manipulated by push/pop operations

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

Stack Before, During, After Call


ra
High address
$sp
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$sp
Low address
a.
b.
c.

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

Function Call Example

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

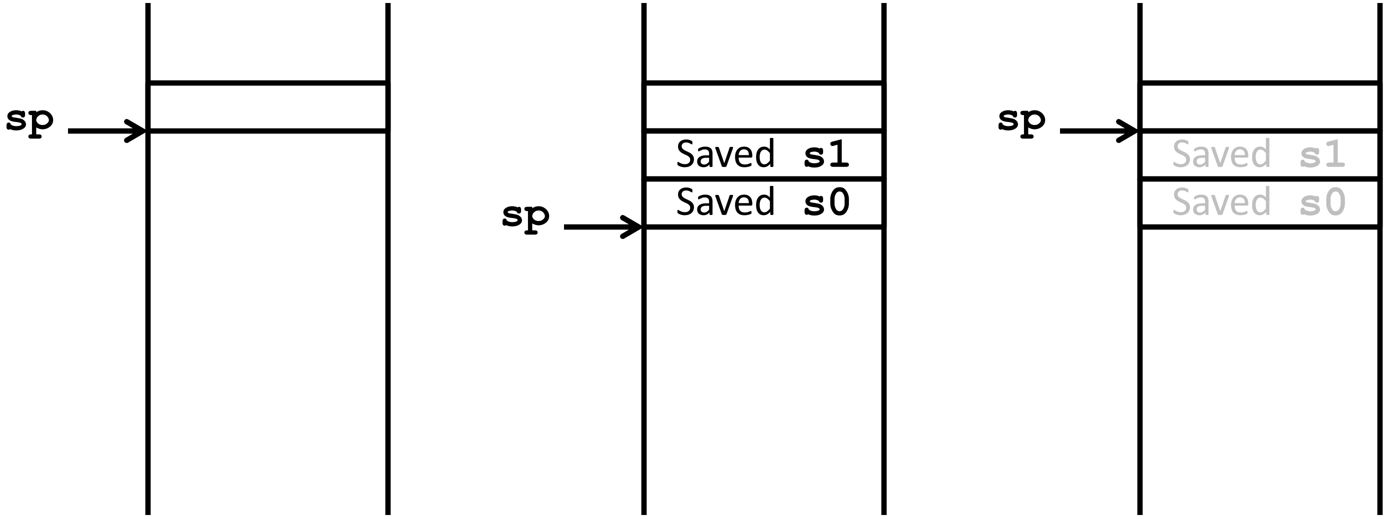
- Parameter variables **g**, **h**, **i**, and **j** in argument registers **a0**, **a1**, **a2**, and **a3**, and **f** in **s0**
- Assume need one temporary register s1

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
```

Stack Before, During, After Function

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



Before call During call After call

Agenda

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

Register Conventions

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

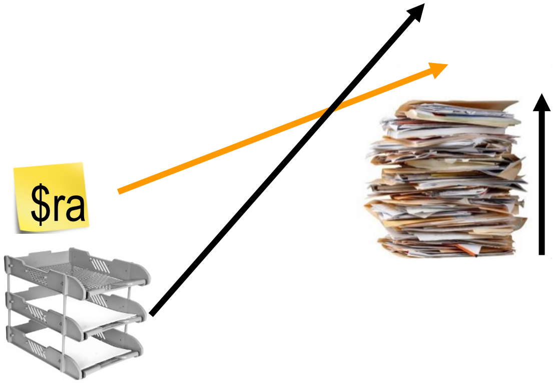
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.

- 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.

Basic Structure of a Function

Prologue

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

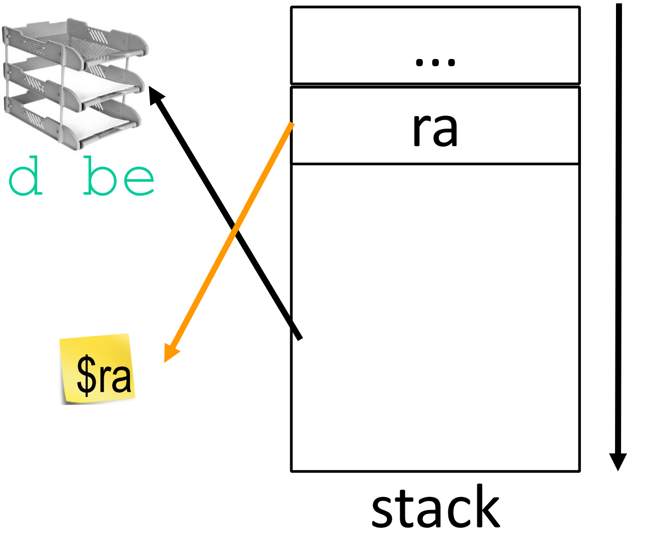


Body **(call other functions…)**

```
   ...
```

Epilogue

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

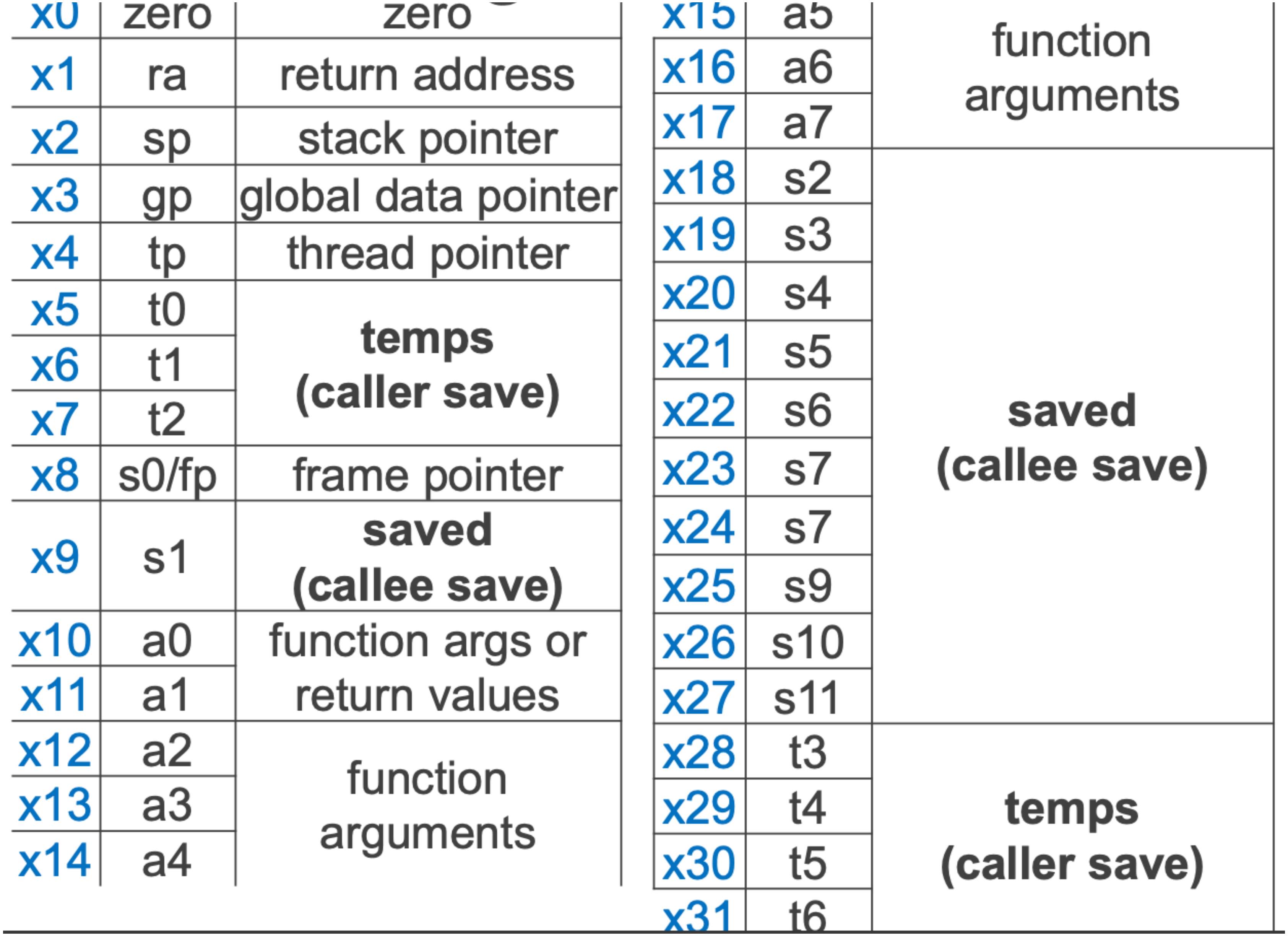


Using Stack to Backup Registers

- 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:



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	

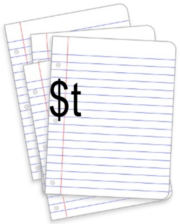
x15	a5	function arguments
x16	a6	
x17	a7	
x18	s2	**saved (callee save)**
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s7	
x25	s9	
x26	s10	
x27	s11	
x28	t3	**temps (caller save)**
x29	t4	
x30	t5	
x31	t6	

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)

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)

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

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:        ...
```

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

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!)

Agenda

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

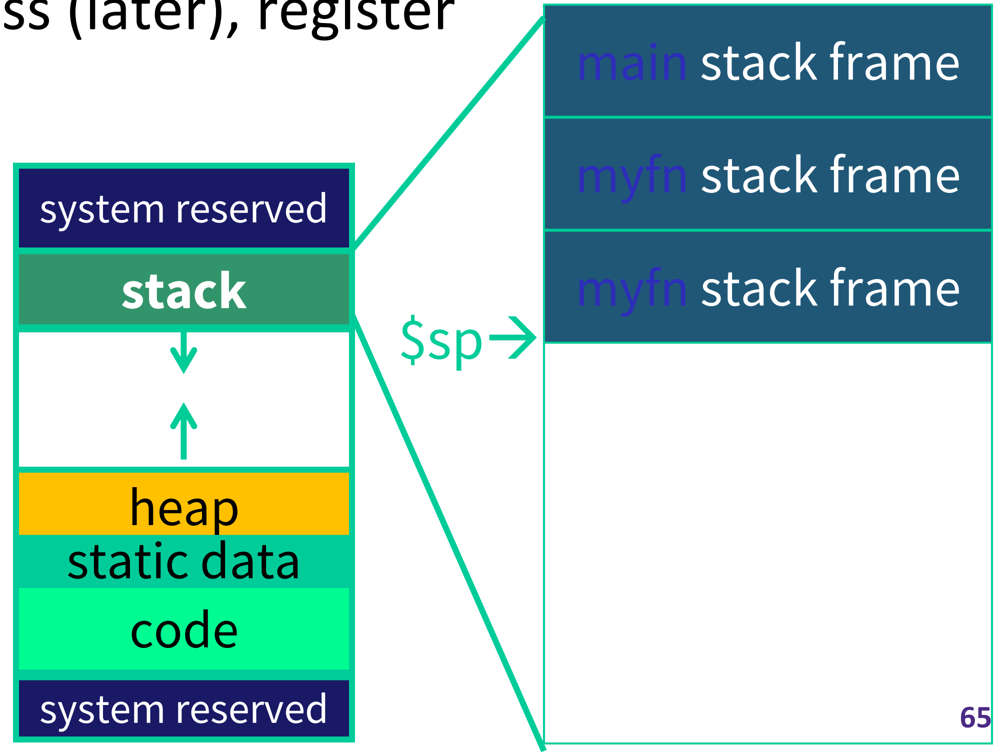
Stack contains stack frames (aka "activation records")

- 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

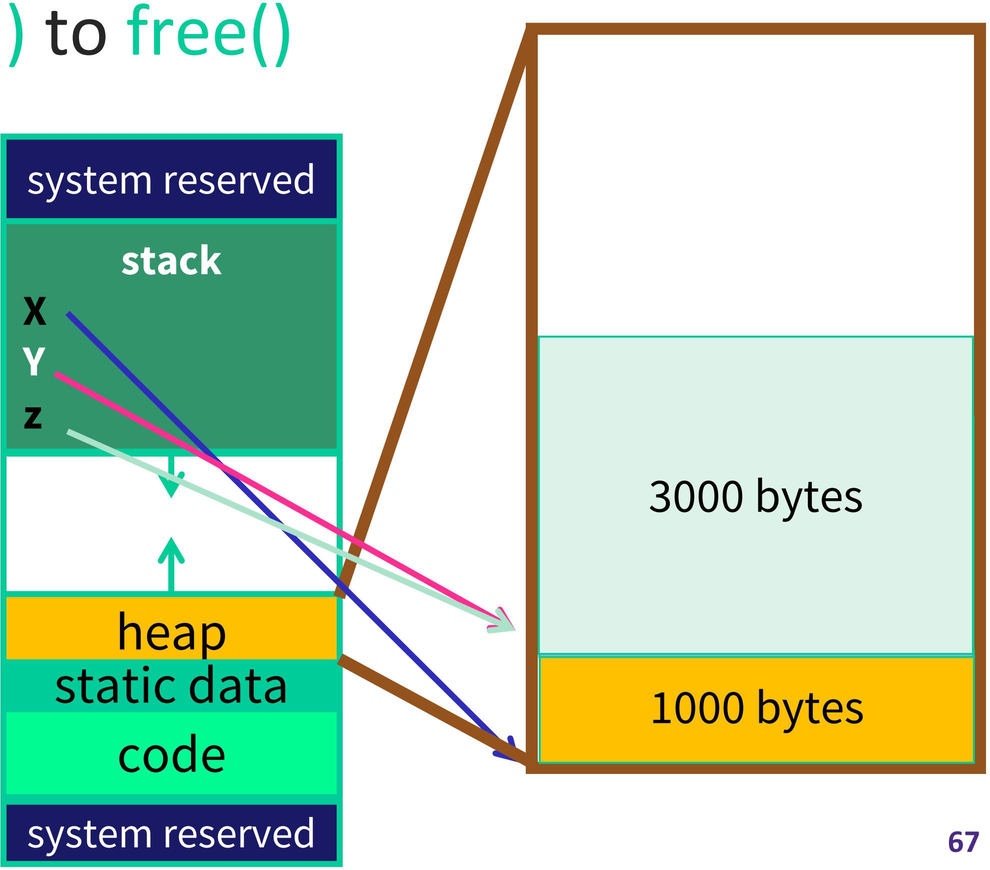
A frame pointer, x8, aka fp/s0

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

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);
}
```

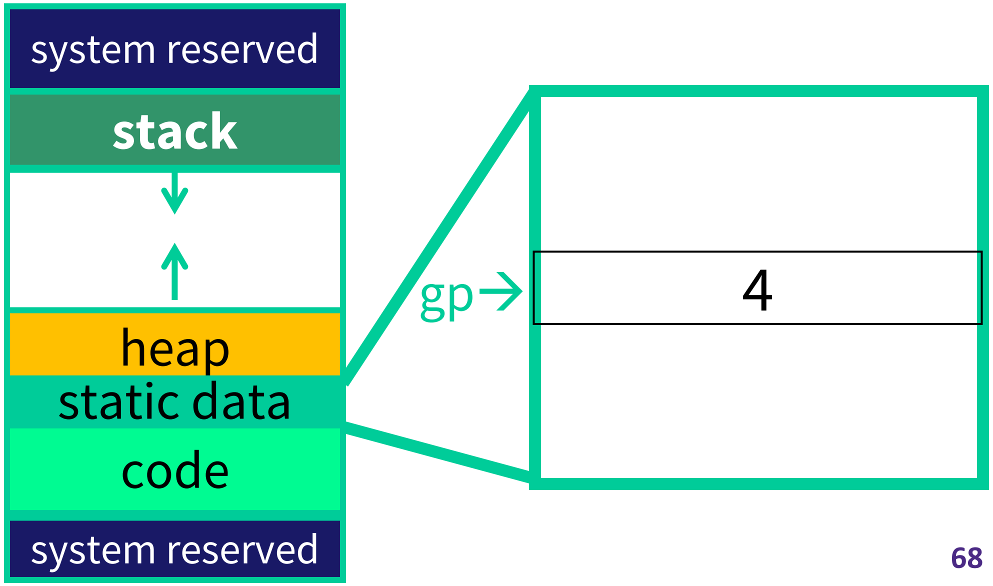


Data Segment

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

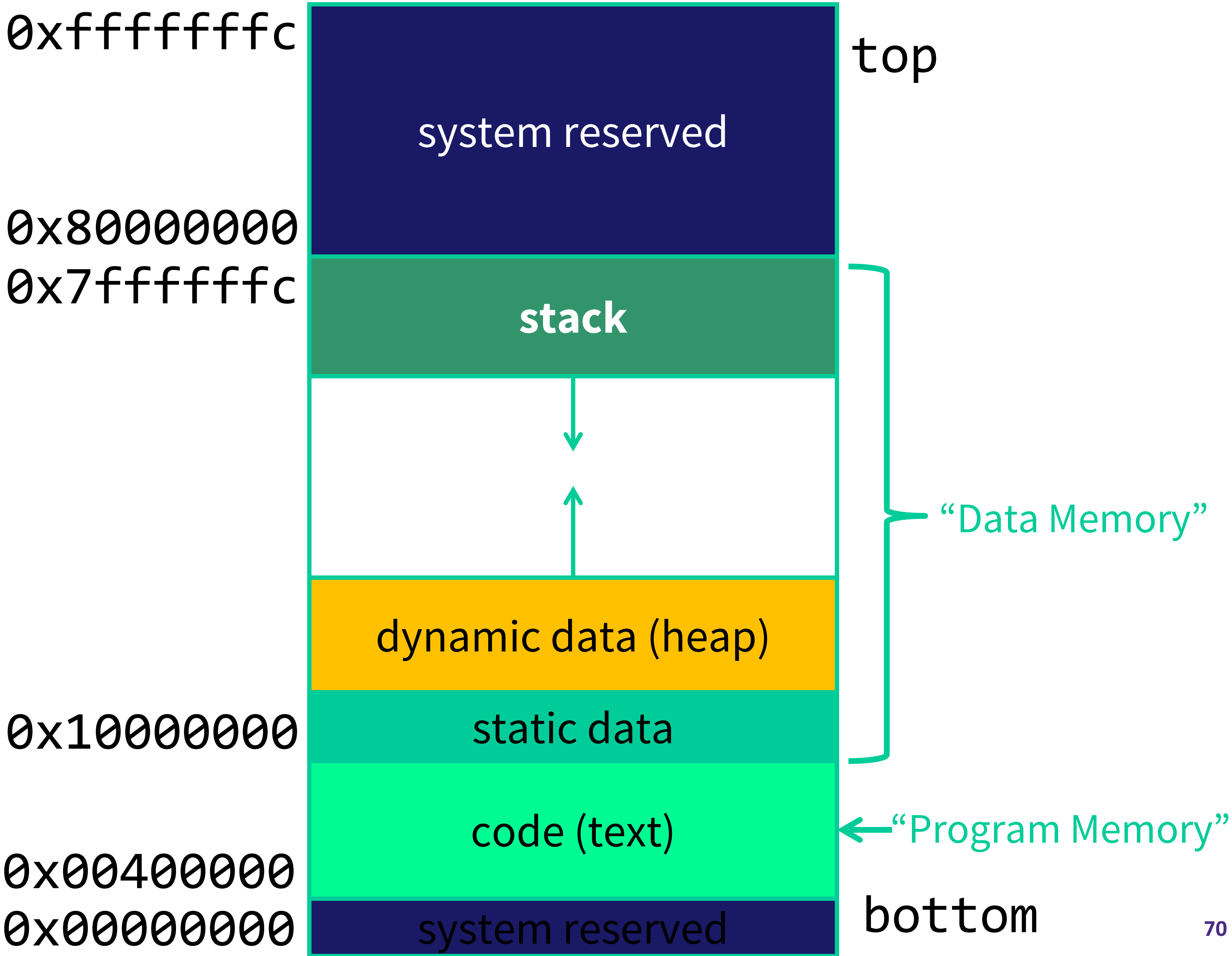


Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

```
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


0xfffffffc
0x80000000
0x7ffffffc
0x10000000
0x00400000
0x00000000
system reserved
stack
dynamic data (heap)
static data
code (text)
system reserved
top
“Data Memory”
“Program Memory”
bottom

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);
}
```

Global and Locals

How does a function load global data?

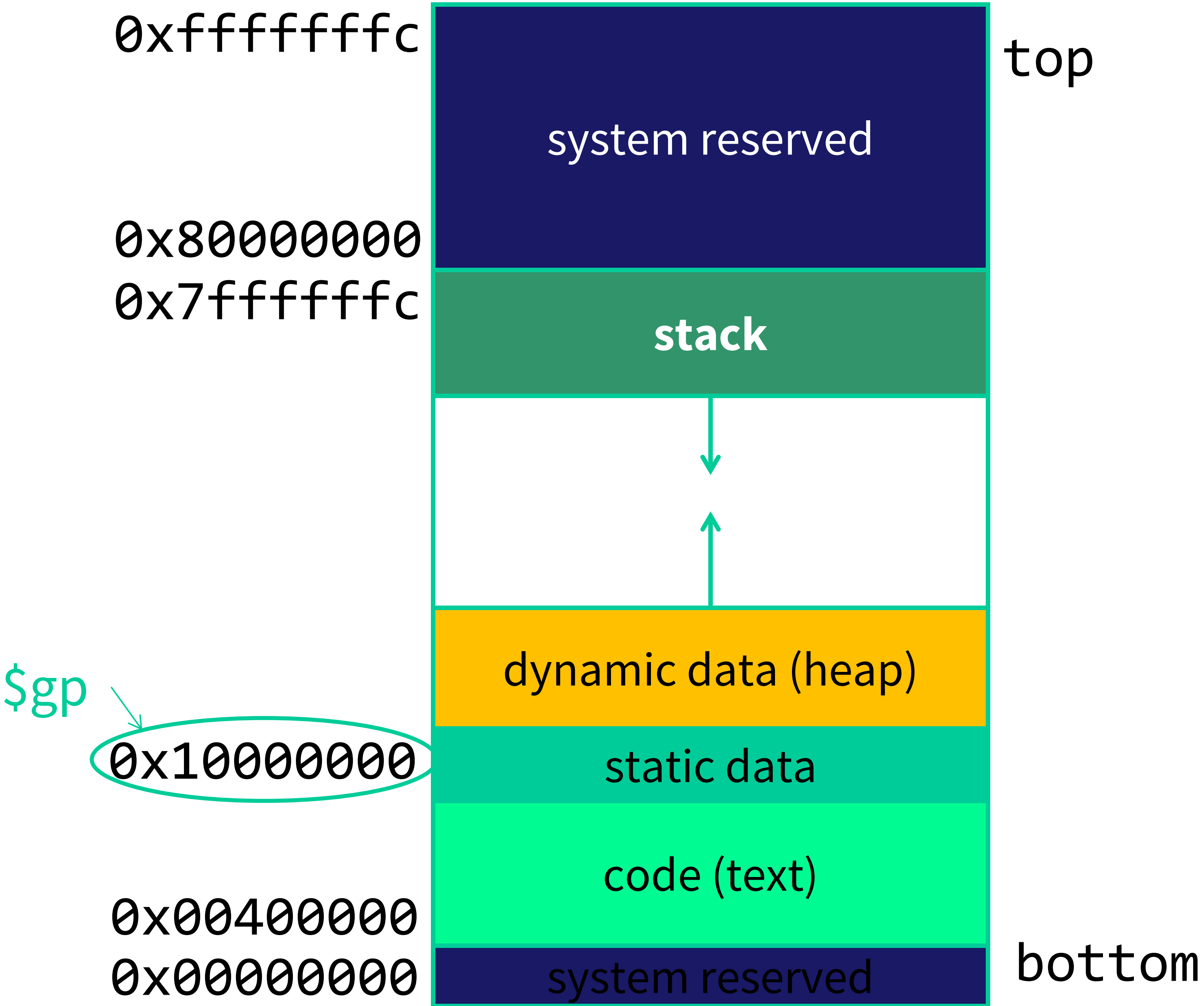
- global variables are just above 0x10000000

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


0xfffffffc
top
system reserved
0x80000000
0x7ffffffc
stack
dynamic data (heap)
$gp
0x10000000
static data
code (text)
0x00400000
0x00000000
system reserved
bottom