Compiling (especially C)

CSE 410 22wi

Lecture 05

What is a compiler?

- A compiler is a translator
- It translates the meaning of a program written in one language into an equivalent program written in (usually) another language
  - E.g., C to Assembler (or machine code)
  - E.g., Java to Java bytecode
  - E.g., Javascript to “native code”

Program

- A program is the specification of a computation
- It is not (necessarily) a description of what steps **must** be taken to carry out the computation
  - Even though we usually think of it that way

- A straightforward compilation will do a fairly direct translation
  - A multiply in the source program will cause there to be a multiple in the target program, say

- The compiler is free to create an target program that is equivalent to the source program, though

"Equivalent Program"

```
int main(int argc, char *argv[])
{
   int x = 20;
   int y = 21;
   int z;
   z = (x-y)*(x+y) / (3*x + 4*y);
   return 0;
}
```

```
int main(int argc, char *argv[])
{
   int x = 20;
   int y = 21;
   int z;
   z = 0;
   return 0;
}
```

```
int main(int argc, char *argv[])
{
    return 0;
}
```

C Overview

- C is a Higher Level Language (HLL)
- C is **much more convenient** to write than assembler, but C's semantics "expose" some aspects of the underlying hardware
  - In particular, main memory
- It looks pretty familiar to a Java programmer, but there are many details that are really different
  - We won't be attempting to master the language...

The C Compiler

- Decides where to store variables

```
int h = 8;
h:   .word    8
```

- Generates instructions / Manages Use of Registers

```
k#@#5-k#%4>

  lw     x10, h
  sll    x10, x10, 1
  addi   x10, x10, -1
  sw     x10, h
```

```
k#@#5-k#%4>

k#@#7-k#%4>

    lw     x10, h
    slli   x10, x10, 1
    addi   x10, x10, -1
    slli   x10, x10, 2
    addi   x10, x10, -1
    sw     x10, h
```

- Tries to detect errors and/or make it hard(er) to write programs that have errors

Anatomy of a C Program

“Global variable”

Command line arguments

Where execution starts

“local variable”

```
int  val = 10238;
int main(int argc, char *argv[])
{
    int i;
    for (i = 2; i<=val/2; i++) {
        if ((val/i)*i == val){
            printf(“%d\n”, i);
        }
    }
    return 0;
}
```

The language doesn’t have a print statement. Instead, call a subroutine.

Curly braces for compound statements

End of execution

Output:
2
5119

Compiling a C Program

```
        .data
val:    .word  10238
```

"Global variables"

```c
int  val = 10238;
int main(int argc, char *argv[])
{
    int i;
    for (i = 2; i<=val/2; i++) {
        if ((val/i)*i == val){
            printf("%d\n", i);
        }
    }
    return 0;
}
```

Compiling a C Program

```
int  val = 10238;
int i;
int main(int argc, char *argv[])
{
    int i;
    for (i = 2; i<=val/2; i++) {
        if ((val/i)*i == val){
            printf("%d\n", i);
        }
    }
    return 0;
}
```

```
        addi  x8, x0, 2
        sw    x8, i
        beq   x0, x0, test
body:
        <body code>
        addi  x8, x8, 1
        sw    x8, i
test:   lw    x9, val
        srai  x9, x9, 1
        blt   x8, x9, body
        beq   x8, x9, body
```

Compiling a C Program

```c
int  val = 10238;
int main(int argc, char *argv[])
{
   int i;
   for (i = 2; i<=val/2; i++) {
      if ((val/i)*i == val){
         printf("%d\n", i);
      }
   }
   return 0;
}
```

```
   lw    x7, val
   lw    x9, i
   div   x10, x9, x7
   mul   x10, x10, x9
   bne   x10, x7, else
    <body code>
else:
```

Part 1: Summary

```
int  val = 10238;
int main(int argc, char *argv[])
{
   int i;
   for (i = 2; i<=val/2; i++) {
        if ((val/i)*i == val){
           printf("%d\n", i);
      }
   }
   return 0;
}
```

- Translating the **bolded** lines to what we know of the ISA is pretty straightforward
- Notice that the language defines the rules for things like operator precedence, as well as basic syntax, like the use of curly braces for compound statements
- The language may provide higher level abstractions than can be implemented in a single machine instruction, like classes
  - Single statements in the language may generate many many assembler instructions, or may even require calling some language provided methodat runtime (e.g., object creation)

Part II: Subroutines

```
int  val = 10238;
int main(int argc, char *argv[])
{
    int i;
    for (i = 2; i<=val/2; i++) {
        if ((val/i)*i == val){
            printf("%d\n", i);
        }
    }
    return 0;
}
```

Subroutines?

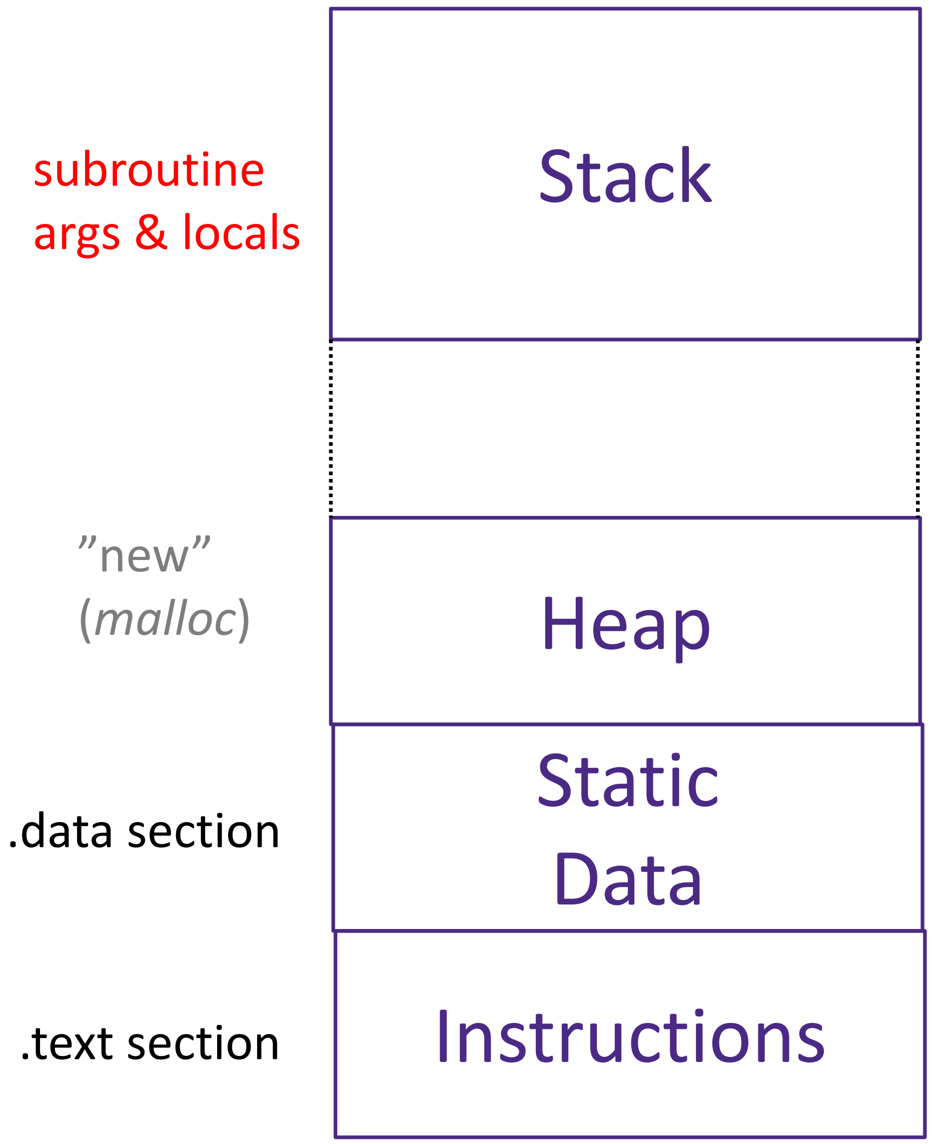
- Where do argument values go?
- Storage for local variables is allocated dynamically, on entry to subroutine.
- Control flow on return is "go back to wherever you came from"!

Subroutines

- When compiling a call of a subroutine, the compiler has to generate instructions that put the arguments somewhere the called routine can find them
  - We may already know about instructions that will let us do that
- When compiling a subroutine, the compiler needs to generate code that will allocate memory for local variables at run time, and code that accesses the arguments supplied on this call
  - This sounds more like a question of memory management than what instructions are in the ISA...
- In the general case, the subroutine might itself call another subroutine (or even itself) before it returns
  - The only control flow instructions we have seen are branches. Can call/return be done with those? *(Hint: no)*

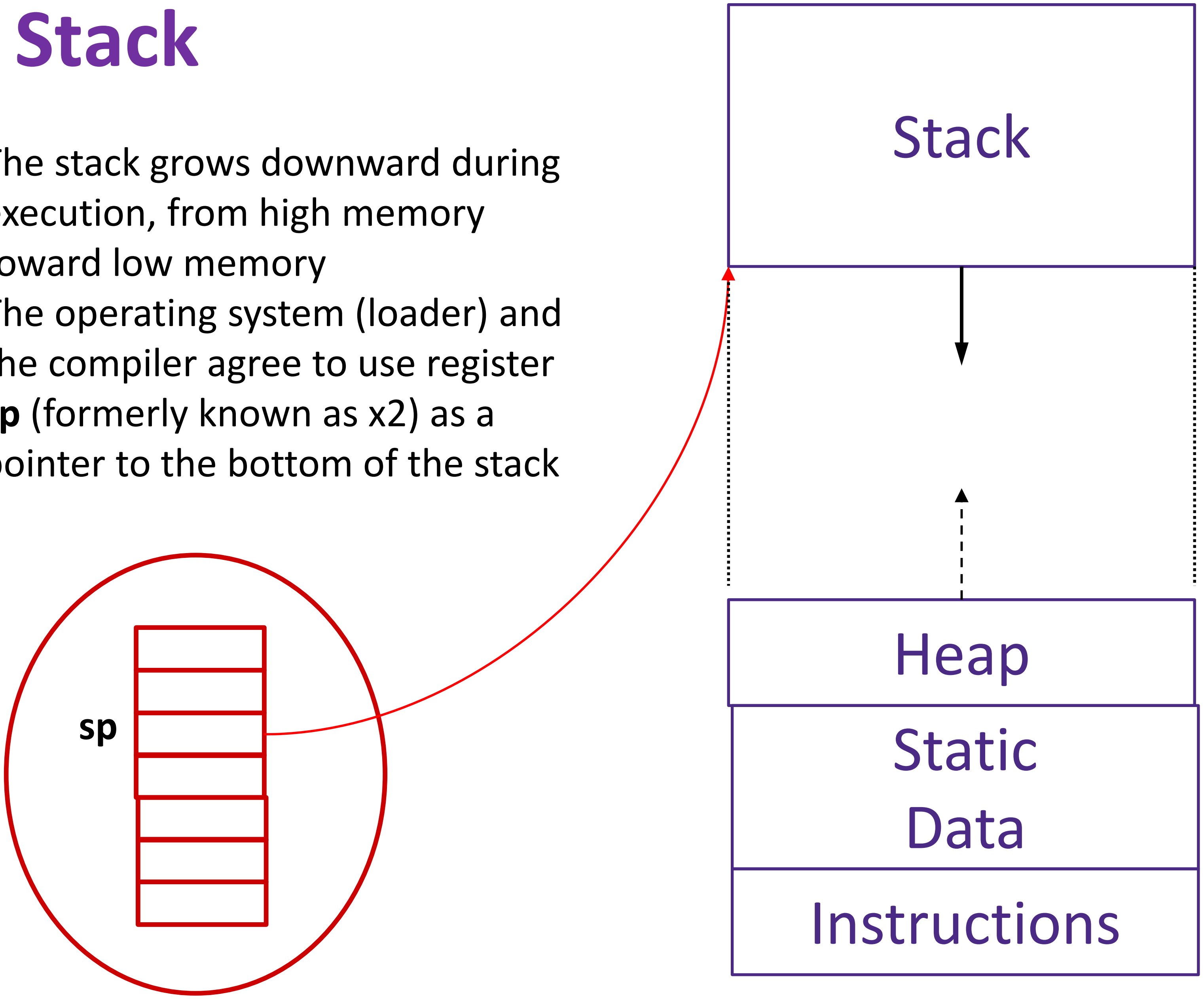
The Memory Model

- While compiling the code, the compiler "knows" what memory will look like at run time
- The OS (program loader) knows the same thing



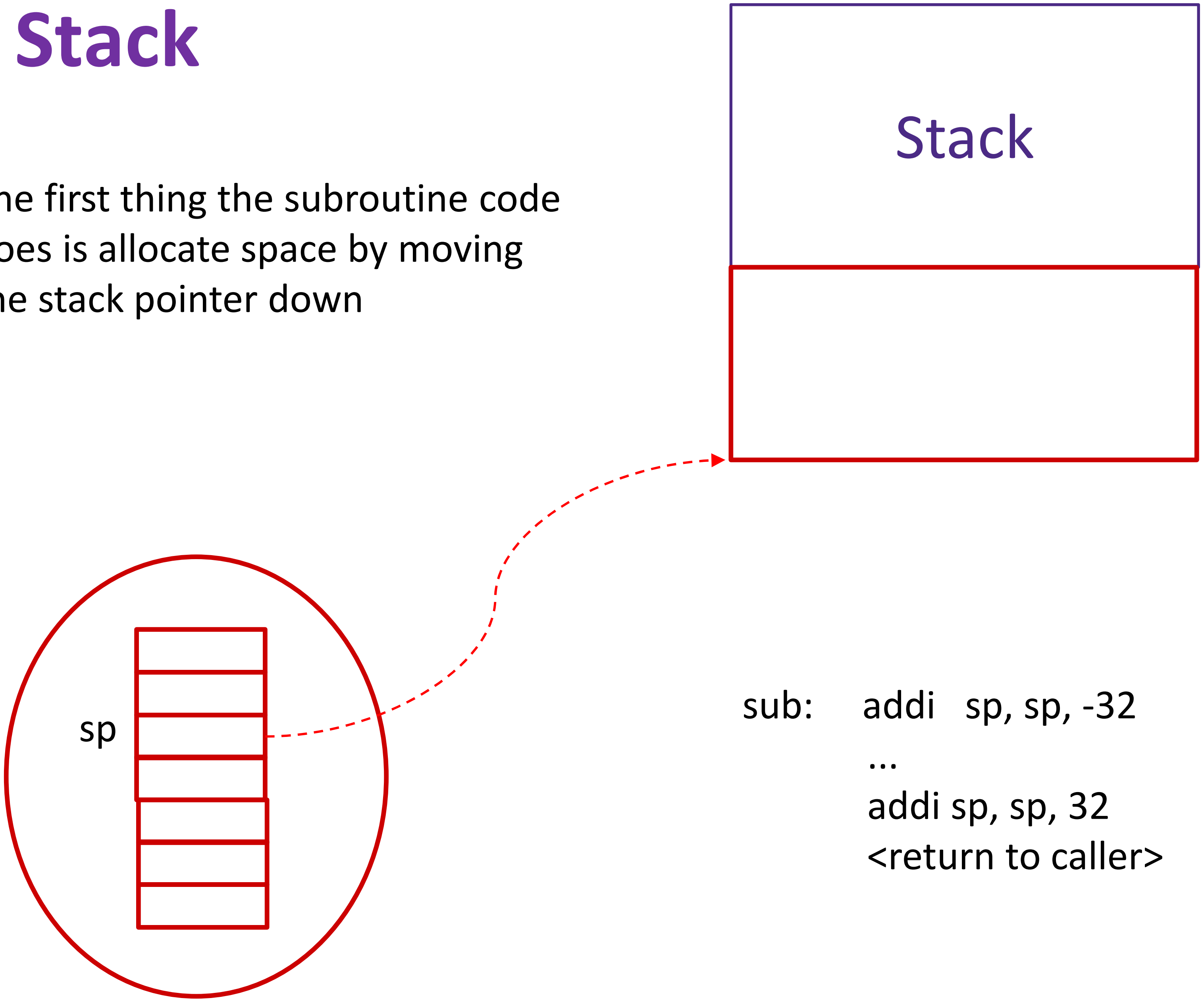
The Stack

- The stack grows downward during execution, from high memory toward low memory
- The operating system (loader) and the compiler agree to use register **sp** (formerly known as x2) as a pointer to the bottom of the stack



The Stack

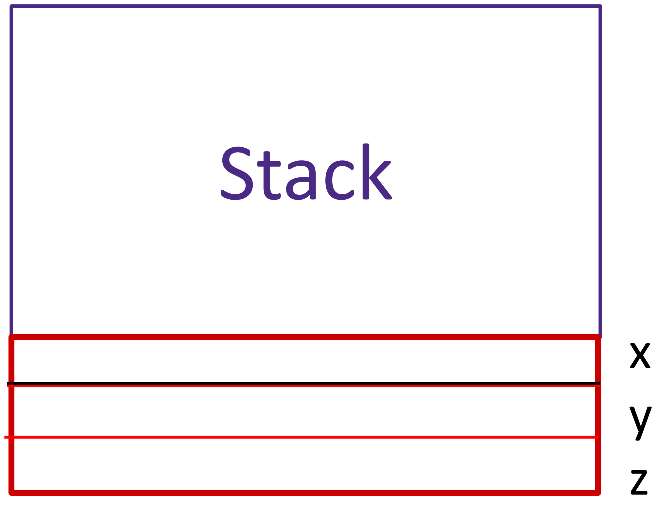
- The first thing the subroutine code does is allocate space by moving the stack pointer down

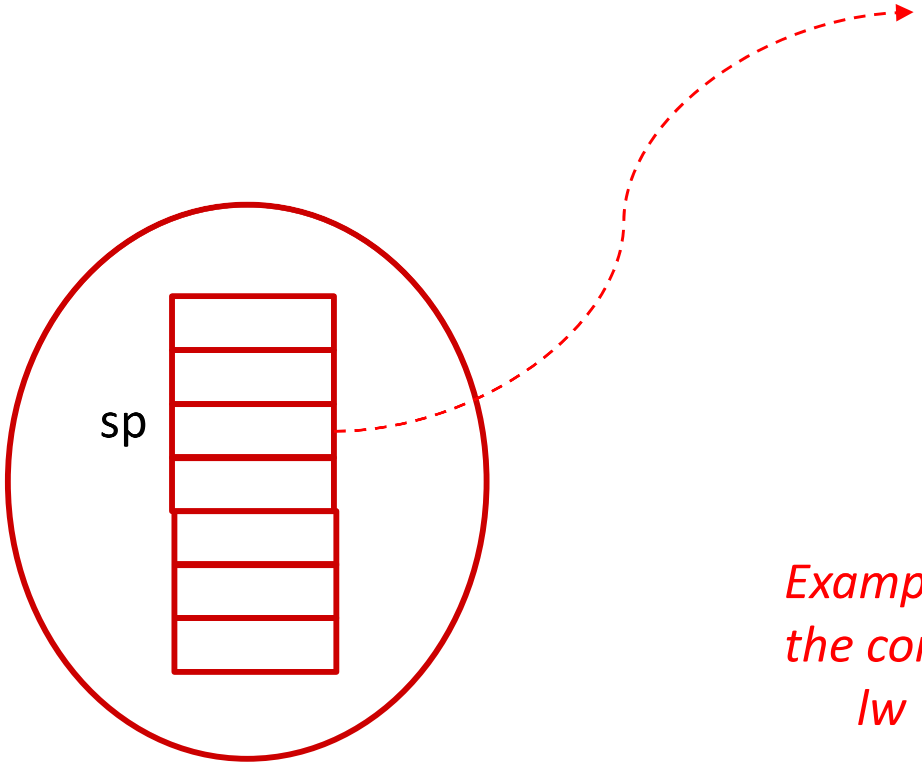


```
sub:    addi   sp, sp, -32
        ...
        addi sp, sp, 32
        <return to caller>
```

The Stack / Locals

```
int sub(int w) {
      int x, y, z;
       ...
       return x;
}
```





```
sub:    addi   sp, sp, -12
        ...
        addi sp, sp, 12
        <return to caller>
```

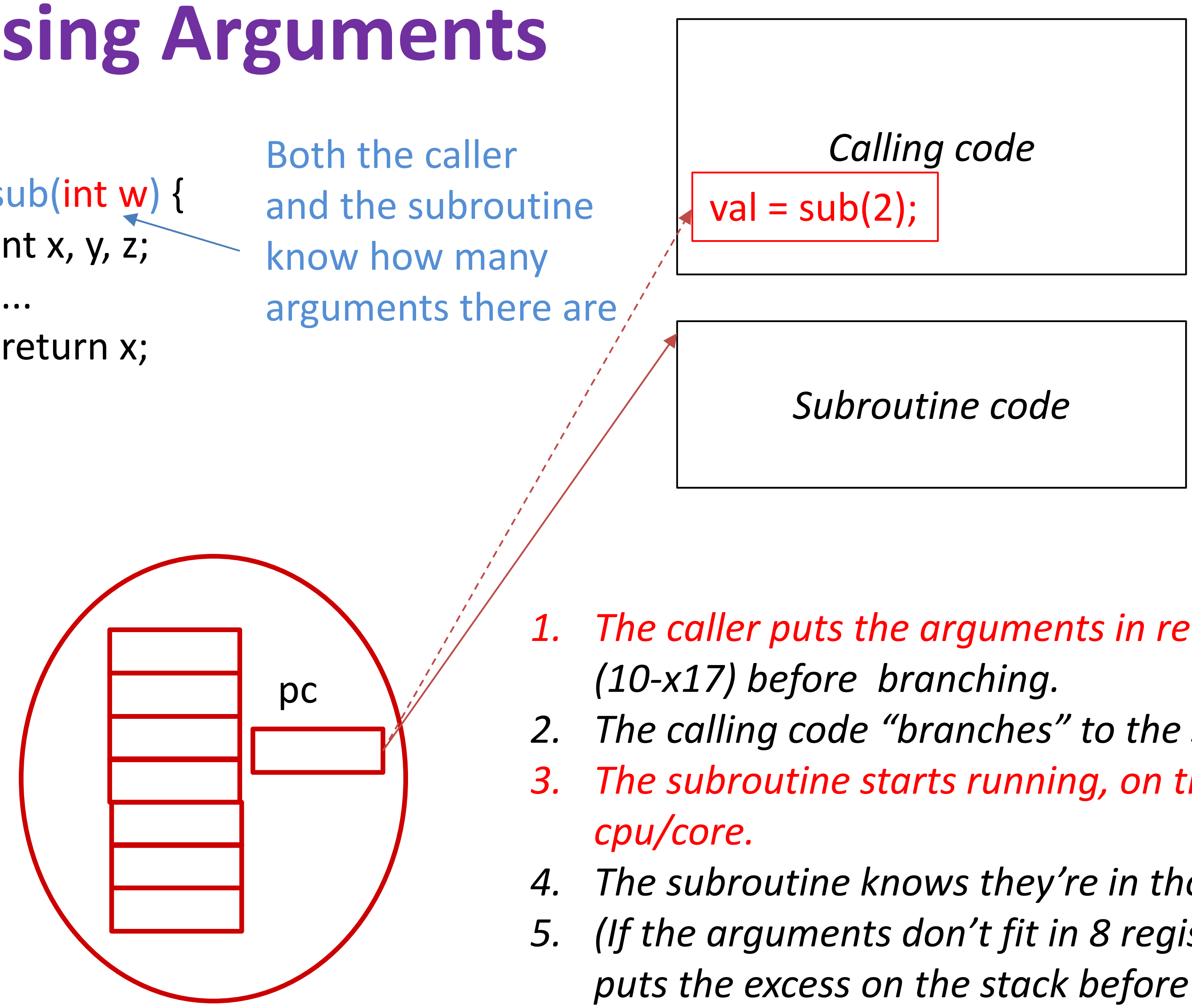
Example: to move variable x into a register, the compiler could generate

lw   x10, 8(sp)

Passing Arguments

```
int sub(int w) {
      int x, y, z;
       …
       return x;
}
```

Both the caller and the subroutine know how many arguments there are

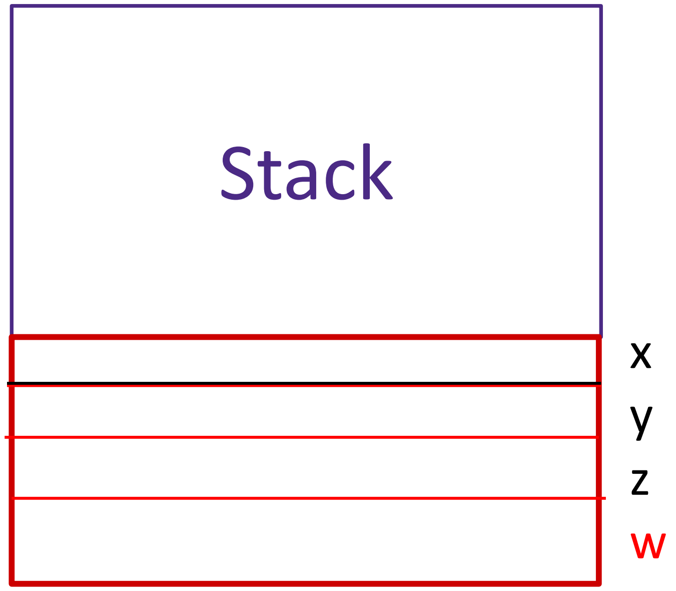


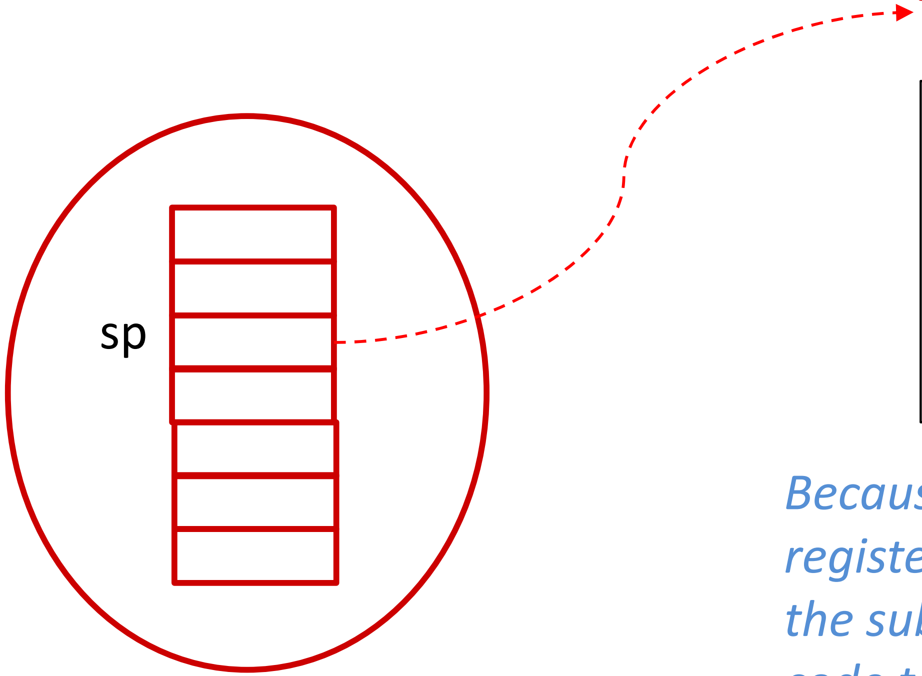
1. *The caller puts the arguments in registers a0-a7 (10-x17) before branching.*
2. *The calling code "branches" to the subroutine.*
3. *The subroutine starts running, on the same cpu/core.*
4. *The subroutine knows they're in those registers.*
5. *(If the arguments don't fit in 8 registers, the caller puts the excess on the stack before branching, and the subroutine gets them from there.)*

The Stack / Arguments

```
val = sub(2);
```

```
int sub(int w) {
      int x, y, z;
       ...
       return x;
}
```



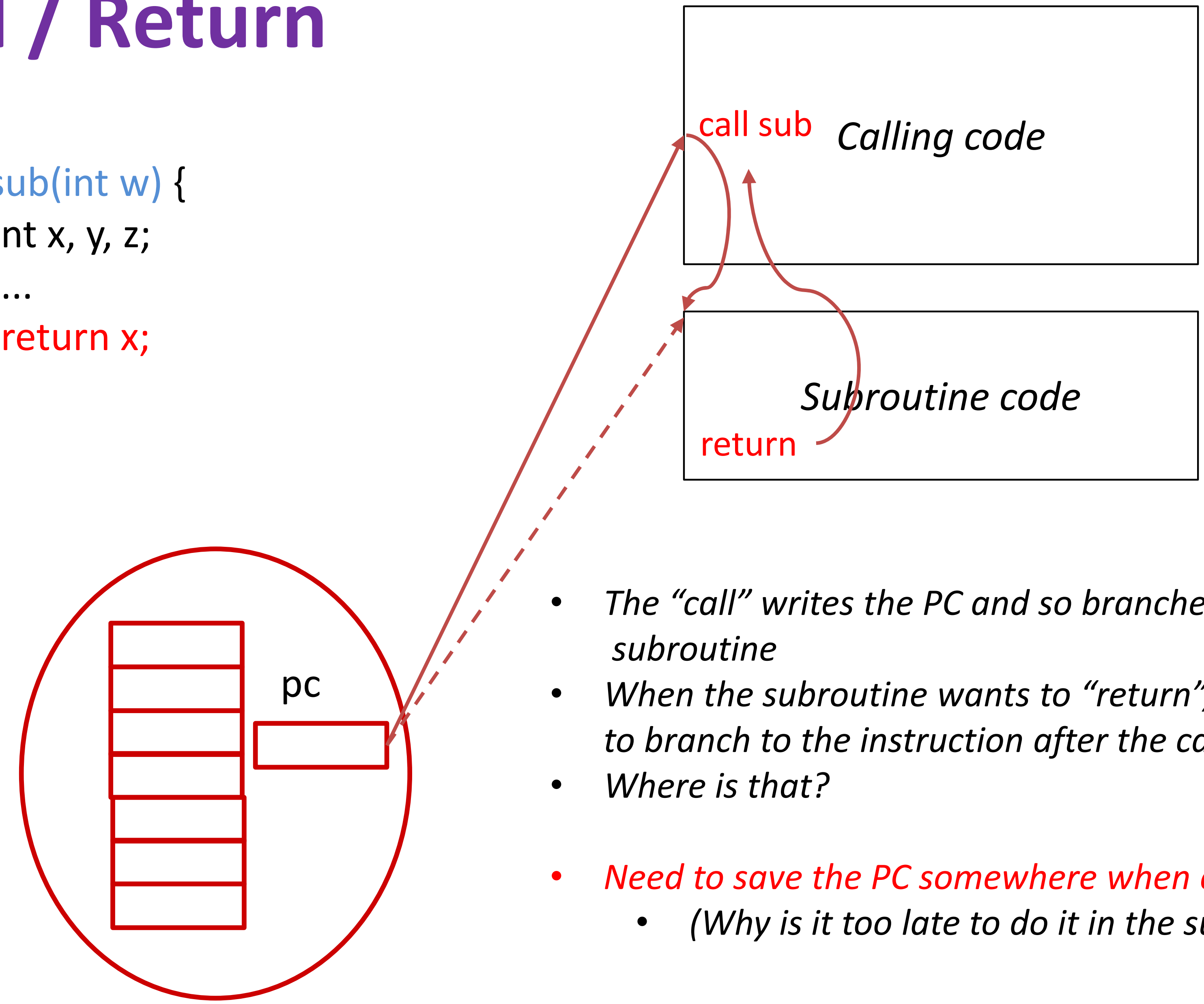


```
sub:    addi   sp, sp, -16
        sw     a0, 0(sp)
        ...
        addi sp, sp, 16
        <return to caller>
```

Because the compiler might want to use register a0 for the subroutine's code (e.g., the subroutine calls a subroutine), it generates code to make space to save it on the stack and to copy a0... to that space on entry.

Call / Return

```
int sub(int w) {
      int x, y, z;
       …
       return x;
}
```



- *The "call" writes the PC and so branches to the subroutine*
- *When the subroutine wants to "return", it needs to branch to the instruction after the call.*
- *Where is that?*
- *Need to save the PC somewhere when calling*
  - *(Why is it too late to do it in the subroutine?)*

Jump-and-Link

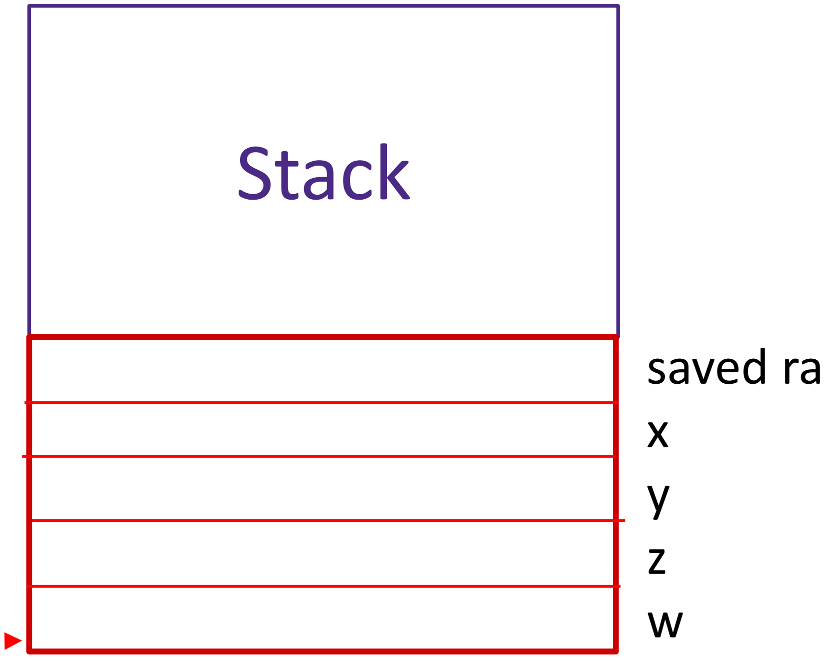
- jal is an instruction that
  1. saves the already updated PC to a register, and
  2. branches
- So, we transfer control to the subroutine and when it starts running the return address is in a register

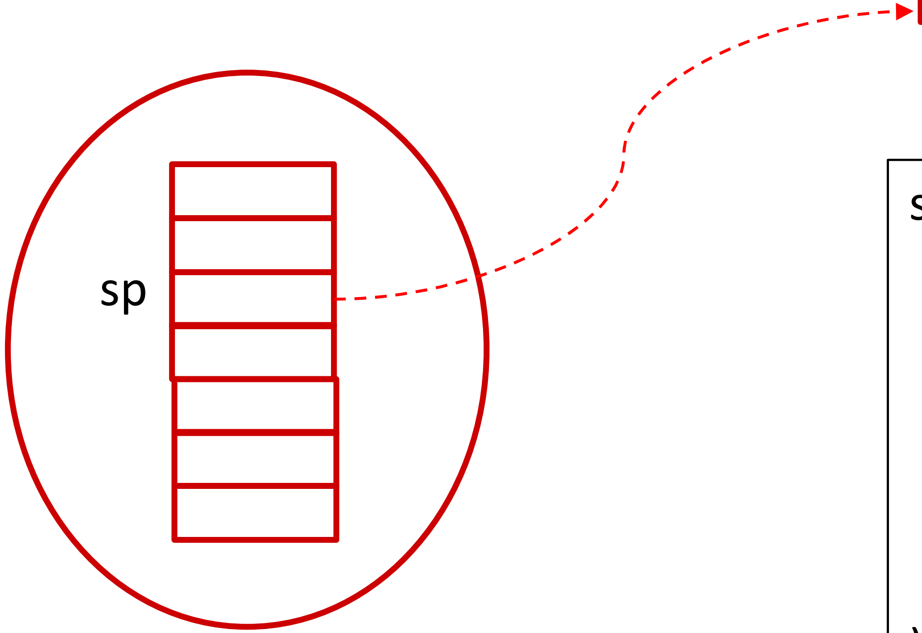
- By convention, register ra (x1) is used to save the PC.
- The caller knows to set ra (with a jal instruction), and the subroutine knows to save ra on the stack until it needs it to do a return

Call / Return

```
int sub(int w) {
      int x, y, z;
      ...
      return x;
}
```

```
val = sub(2);
```





```
sub:    addi   sp, sp, -20
         sw     a0, 0(sp)
         sw     ra, -16(sp)
         ...
         lw     ra, -16(sp)
         addi sp, sp, 20
         jr      ra
)
```

One Last Detail: The Return Value

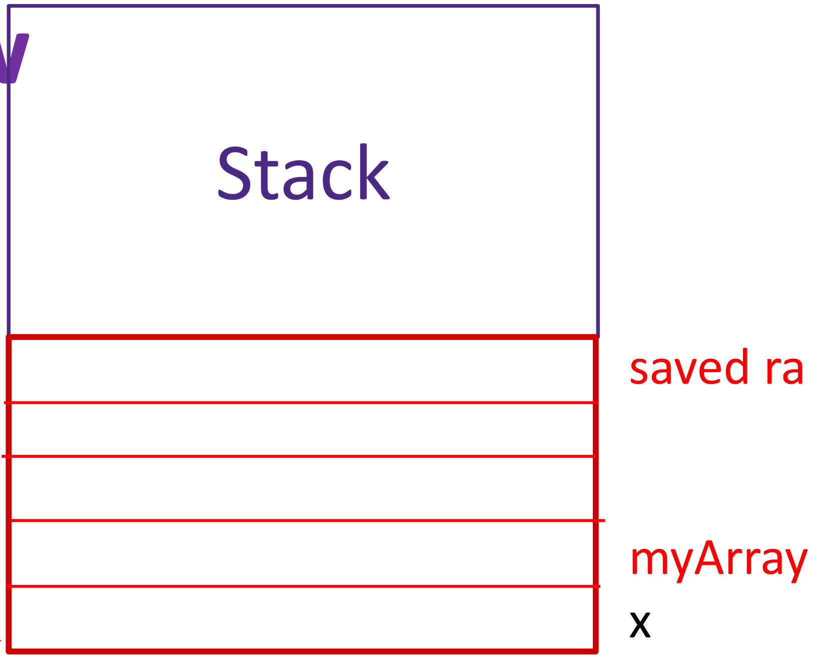
```
int sub(int w) {
    int x, y, z;
     ...
     return x;
}
```

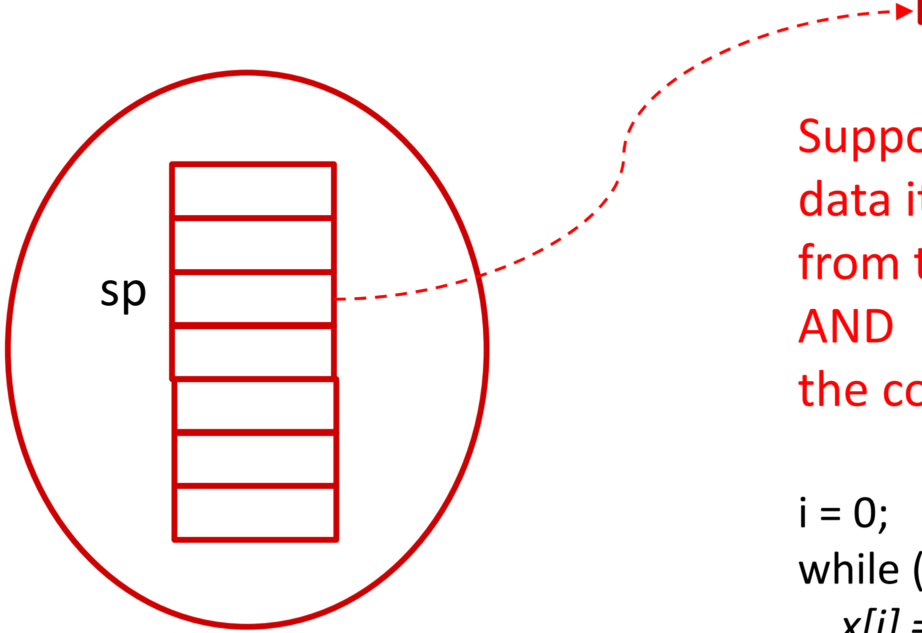
- The problem of returning a value back to the caller is just like the problem of passing arguments in from the caller
- The solution is the same
  - Leave the return value in a register
  - In RISC-V, register a0 is used

Security: Buffer Overflow

val = sub(2);

```
int sub(int x) {
      int myArray[3];
       ...
       return x;
}
```





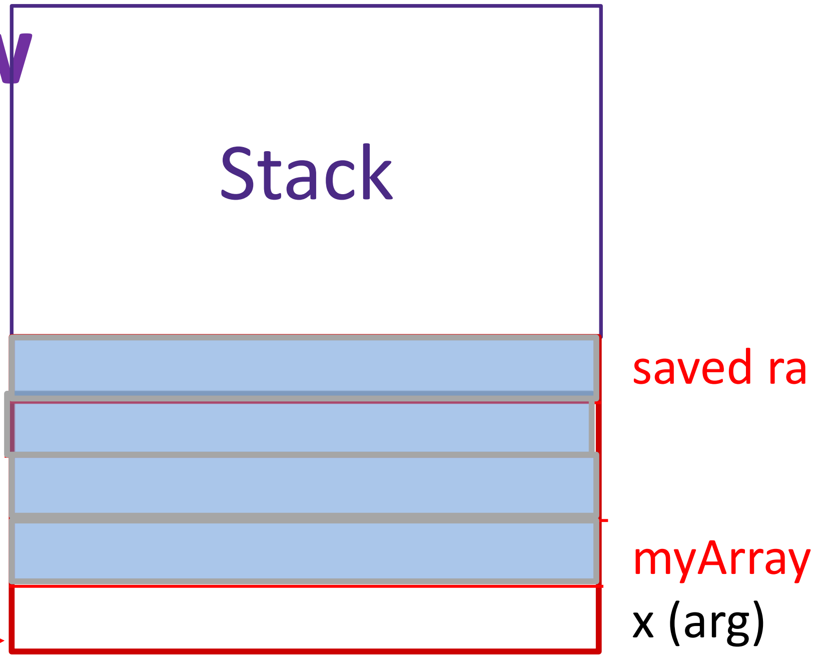
Suppose the code fills myArray with data it gets from the user (e.g., from a file or from the network)
AND
the code doesn't check how much data it gets.

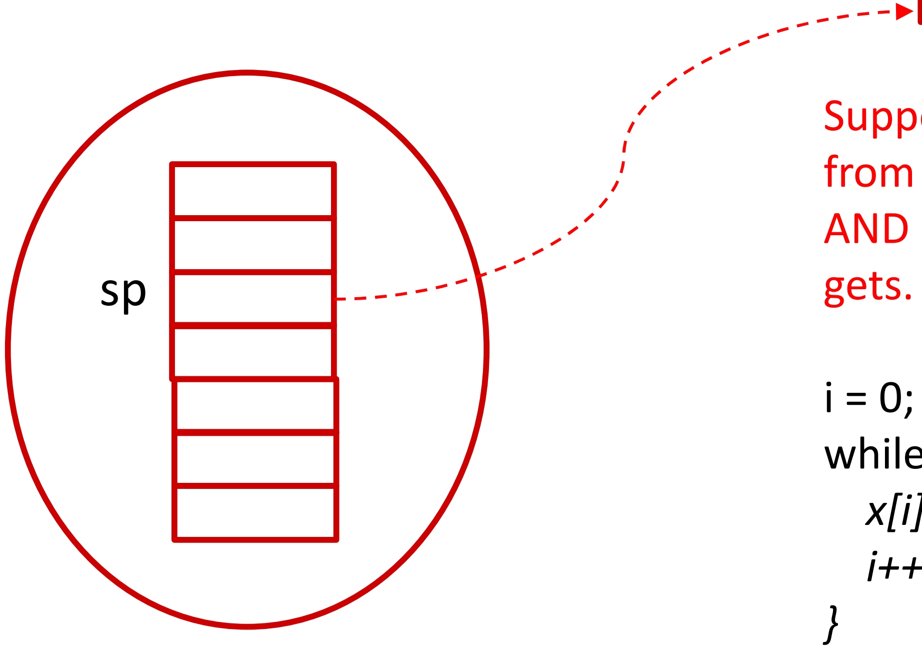
```
i = 0;
while ( more data ) {
   x[i] = <new data element>;
   i++;
}
```

Security: Buffer Overflow

val = sub(2);

```
int sub(int x) {
      int myArray[3];
       ...
       return x;
}
```





Suppose the code fills myArray withdata it gets from the user: from a file or from the network AND the code doesn't check how much data it gets.

```
i = 0;
while ( more data ) {
   x[i] = <new data element>;
   i++;
}
return;  // !!!
```

Procedures: Managing Register Usage

Caller

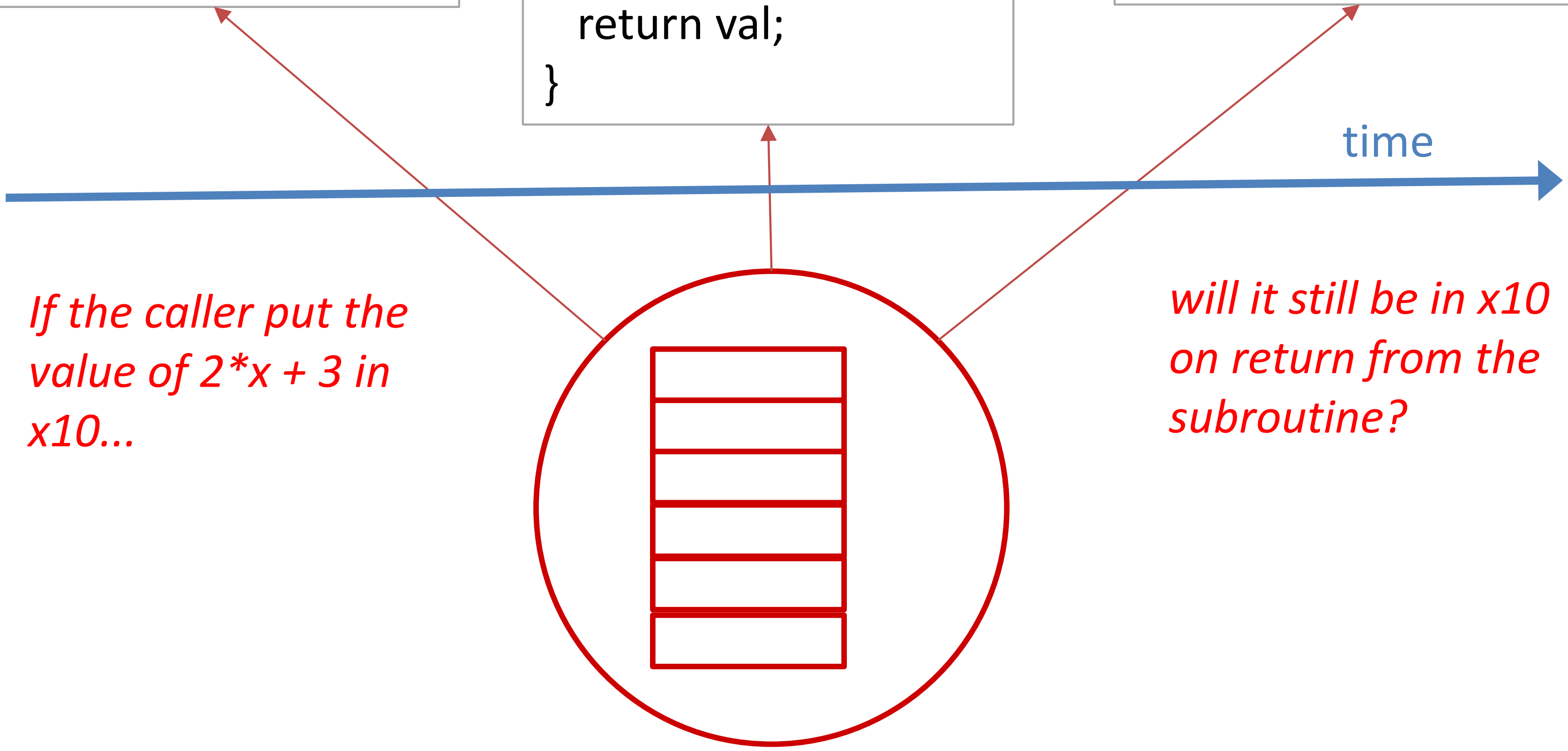
```
y = 2*x + 3;
y = sub(y);
if ( y < 0  ) y = 2*x + 3;
```

Callee

```
int sub(int x)
{
   ...
   return val;
}
```

Caller

```
y = 2*x + 3;
y = sub(y);
if ( y < 0  ) y = 2*x + 3;
```



*If the caller put the value of 2*x + 3 in x10...*

will it still be in x10 on return from the subroutine?

Procedures: Managing Register Usage

- Options:
  - Caller Saves All
    - Before making the call, the caller saves any values currently in registers that it might want to use again after the call
    - Callee doesn't have to save and restore any register
  - Callee Saves All
    - Callee must save the value in any register it wants to use, and must restore it just before returning
  - Problems
    - Caller Saves All might save registers the callee isn't going to use anyway
    - Callee Saves All might save registers the caller doesn't care about

Procedures: Managing Register Usage

- RISC-V solution
  - 12 registers are callee saved
    - Callee needs to save and restore their values if it wants to use those registers
    - So, callee should avoid using those registers
    - Called s0 through s11
  - 7 registers are caller saved
    - Caller needs to save them before call and restore them after if it wants to preserve their values across the call
    - Callee is free to use those registers without saving/restoring
    - So, callee wants to use those registers before any of s0-s11
    - Called t0 through t6

Procedure Call Summary (to this point)

- Procedure call works by agreement between the caller and the callee
- Both caller and callee know "the signature" of the procedure: e.g., int sub(int x, int y)
- Both know the caller will leave the two arguments in registers a0 through a7, so the callee should look for them there
- Both know the caller must save values in registers t0 through t6
- Both know the caller will use a jal that saves the address of its next instruction in register ra
- On entry, the callee allocates space by moving the stack pointer down to:
  - to (possibly) save the arguments in memory,
  - for its own local variables
  - to save any of s0-s11 that it wants to use, O
- On return, the callee loads the saved ra value back into ra, restores any saved s0-s11 registers, moves the stack pointer back up to where it was, and branches to the address in ra