

# **CS-200**

## **Computer Architecture**

---

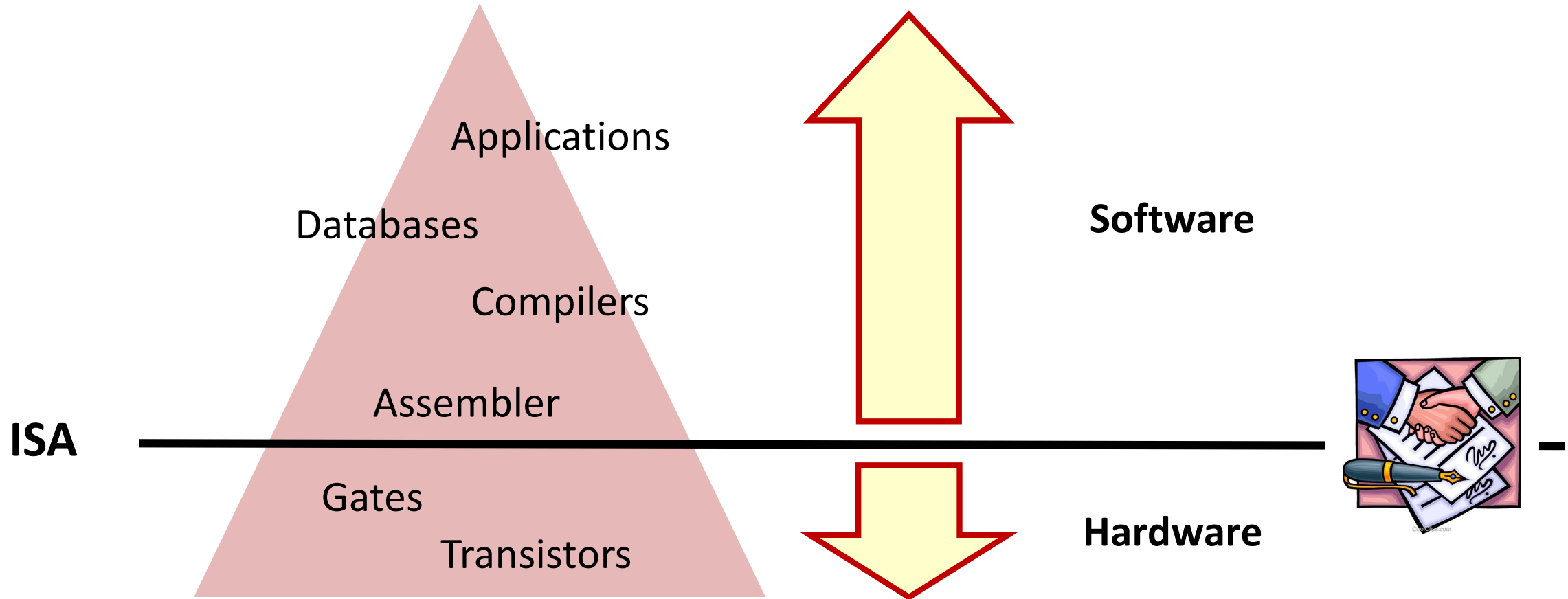
### **Part 1b. Instruction Set Architecture**

#### **Branches, Functions, and Stack**

Paolo Ienne

`<paolo.ienne@epfl.ch>`

# The Contract between HW and SW



# Arithmetic and Logic Instructions Are Easy

- Two operands

<code>sll</code>	<code>x5, x5, x9</code>	Shift <code>x5</code> left of <code>x9</code> positions $\rightarrow$ <code>x5</code>
<code>add</code>	<code>x6, x5, x7</code>	Add <code>x5</code> and <code>x7</code> $\rightarrow$ <code>x6</code>
<code>xor</code>	<code>x6, x6, x8</code>	Logic XOR bitwise <code>x6</code> and <code>x8</code> $\rightarrow$ <code>x6</code>
<code>slt</code>	<code>x8, x6, x7</code>	Set <code>x8</code> to <b>1</b> if <code>x6</code> is lower than <code>x7</code> , to <b>0</b> otherwise

True  
False

- One operand and a constant (**12-bit immediate**)

<code>slli</code>	<code>x5, x5, 3</code>	Shift <code>x5</code> left of 3 positions $\rightarrow$ <code>x5</code>
<code>addi</code>	<code>x6, x5, 72</code>	Add 72 to <code>x5</code> $\rightarrow$ <code>x6</code>
<code>xori</code>	<code>x6, x6, -1</code>	Logic XOR bitwise <code>x6</code> and <code>0xFFFFFFFF</code> $\rightarrow$ <code>x6</code>
<code>slti</code>	<code>x8, x6, 321</code>	Set <code>x8</code> to <b>1</b> if <code>x6</code> is lower than 321, to <b>0</b> otherwise

`xori rd,rs1,imm`     $rd \leftarrow rs1 \wedge \text{sext}(imm)$

0xFFF sign extended  
to 0xFFFFFFFF

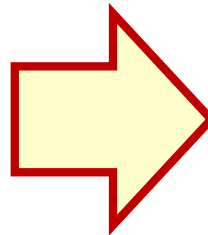
# Constants Must Take $\leq 12$ Bits

- The constant (**immediate**) is part of the instruction!

	31	25	24	20	19	15	14	12	11	7	6	0	
R	funct7				rs2		rs1	funct3	rd		opcode		Register-Register
I	imm[11:0]				rs1		funct3	rd		opcode			Register-Immediate
I	funct7				imm[4:0]		rs1	funct3	rd		opcode		Register-Immediate Shift
S	imm[11:5]				rs2		rs1	funct3	imm[4:0]		opcode		Store
B	imm[12-10:5]				rs2		rs1	funct3	imm[4:1-11]		opcode		Branch
U	imm[31:12]				rd		opcode						Upper Immediate
J	imm[20-10:1-11-19:12]				rd		opcode						Jump

- To use larger constants, one needs to go through a register

~~xori x6, x6, 0x12345678~~

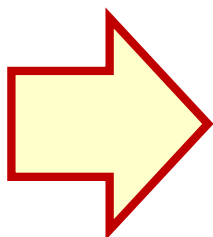


```
lui    x5, 0x12345
addi   x5, x5, 0x678
xor    x6, x6, x5
```

# Assembler Directives

- The assembler can help to have more readable code

```
lui    x5, 0x12345
addi   x5, x5, 0x678
xor    x6, x6, x5
```



```
.equ    something, 0x12345678
```

```
lui    x5, %hi(something)
addi   x5, x5, %lo(something)
xor    x6, x6, x5
```

Directive	Effect
<code>.text</code>	Store subsequent instructions at next available address in <i>text</i> segment
<code>.data</code>	Store subsequent items at next available address in <i>data</i> segment
<code>.ascii</code>	Store string followed by null-terminator in <code>.data</code> segment
<code>.byte</code>	Store listed values as 8-bit bytes
<code>.word</code>	Store listed values as 32-bit words
<code>.equ</code>	Define constants

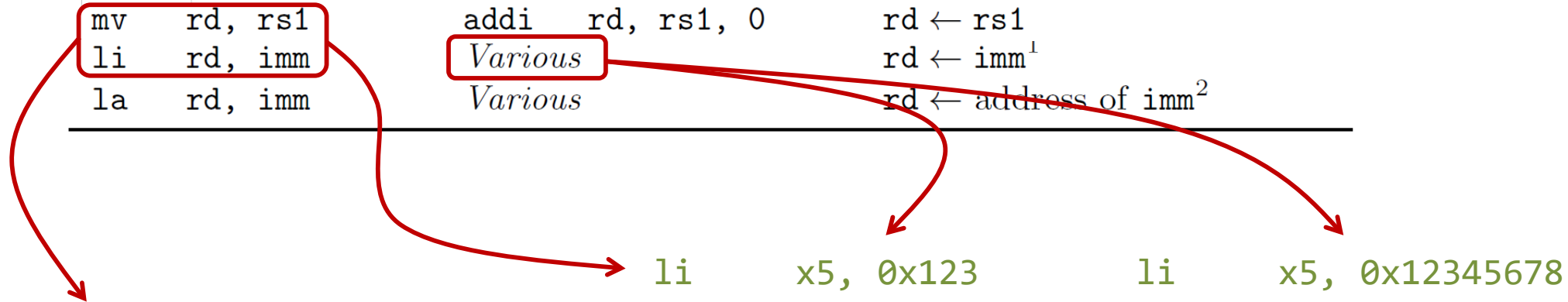
This is actually subtly wrong for some values of `something` (e.g., `0x12345fff`)  
Can you tell why?

# A Strange Register: $x_0$

- The register  $x_0$  does not behave like all others:
  - Register  $x_0$  is always zero
  - One can write into  $x_0$  but this has no effect—because it is always zero
- Both aspects are handy in many situations
  - Zero is a very common, useful value—e.g., set  $x_5 \leftarrow -x_5$   
`sub x5, x0, x5`
  - If one does not care of the result of an instruction—e.g., no operation  
`add x0, x0, x0`

# Pseudoinstructions

Pseudoinstruction	Translation	Pseudocode
<b>Arithmetic</b>		
nop	addi zero, zero, 0	<i>Nothing</i>
neg rd, rs1	sub rd, zero, rs1	rd ← -rs1
<b>Logical</b>		
not rd, rs1	xori rd, rs1, -1	rd ← ~rs1
<b>Move</b>		
mv rd, rs1	addi rd, rs1, 0	rd ← rs1
li rd, imm	<i>Various</i>	rd ← imm <sup>1</sup>
la rd, imm	<i>Various</i>	rd ← address of imm <sup>2</sup>



li should probably be called mvi but is not for historical reasons

```

li    x5, 0x123
      ↓
addi  x5, x0, 0x123

li    x5, 0x12345678
      ↓
lui   x5, 0x12345
addi  x5, x5, 0x678
  
```

# Pseudoinstructions

Pseudoinstruction	Translation	Pseudocode
<b>Arithmetic</b>		
nop	addi zero, zero, 0	<i>Nothing</i>
neg rd, rs1	sub rd, zero, rs1	$rd \leftarrow -rs1$
<b>Logical</b>		
not rd, rs1	xori rd, rs1, -1	$rd \leftarrow \sim rs1$
<b>Move</b>		
mv rd, rs1	addi rd, rs1, 0	$rd \leftarrow rs1$
li rd, imm	<i>Various</i>	$rd \leftarrow imm^1$
la rd, imm	<i>Various</i>	$rd \leftarrow \text{address of } imm^2$

Use **li** when **imm** is a constant or a symbol defined with **.equ**

Use **la** when **imm** is a label

```
.equ my_constant, 0x12345678
```

```
my_variable: .word 0x12345678
```

```
li x5, my_constant
```

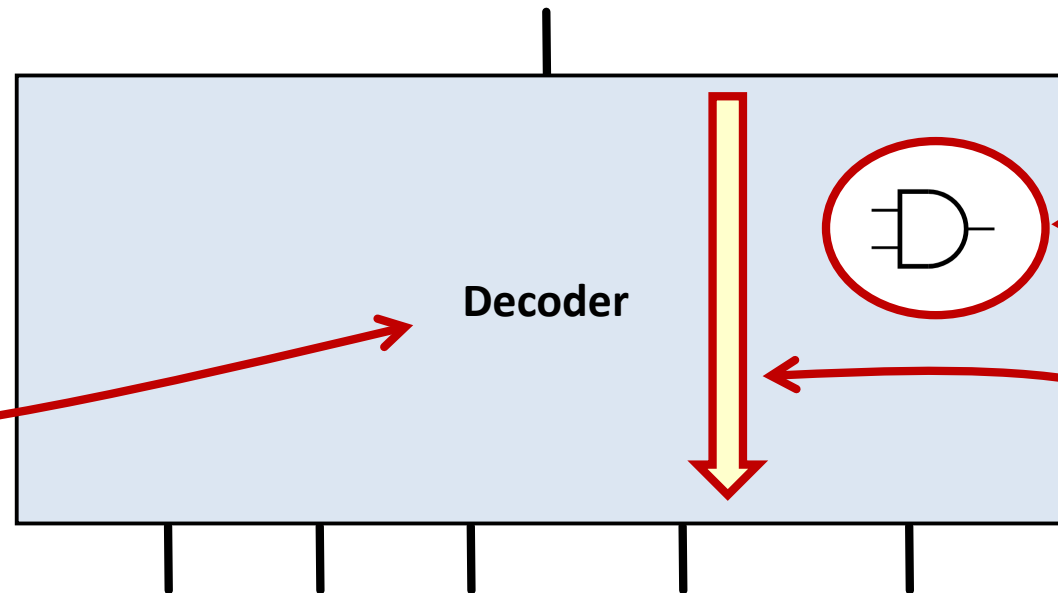
```
la x5, my_variable
```

# Why Pseudoinstructions?

“and x5, x1, x3”

0000 0000 0010 0000 1000 0011 0010 0011

Make this simple!



We want to minimize hardware resources...

...and we want to spend minimal time decoding!

01001 1 00001 00011 0110011  
“x5” 1 “x1” “x3” “and”

# Control Flow (or Control Transfer)

	value
x0	0
x1	0x00123456
x2	0
x3	...
...	
x30	...
x31	...

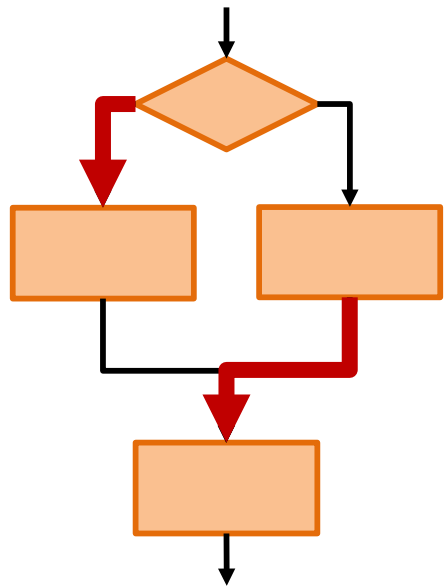
```
li    x1, 0x00123456
li    x2, 0
li    x3, 1
li    x4, 0
li    x5, 0
li    x6, 32
loop: and  x5, x1, x3
      add  x2, x2, x5
      srli x1, x1, 1
      addi x4, x4, 1
      bne  x4, x6, loop
```

Only one form of control flow:  
branch/jump to a particular instruction



# An IF-THEN-ELSE

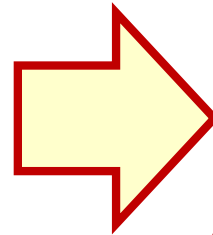
Branch if equal



```

if (x5 == 72) {
    x6 = x6 + 1;
} else {
    x6 = x6 - 1;
}
...

```



```

.text

    li x7, 72
    beq x5, x7, then_clause

else_clause:
    addi x6, x6, -1
    j end_if

then_clause:
    addi x6, x6, 1

end_if:
    ...

```

B	imm[12-10:5]	rs2	rs1	funct3	imm[4:1-11]	opcode	Branch
---	--------------	-----	-----	--------	-------------	--------	--------

**beqi** does not exist  
(no space in the encoding for an immediate)

Jump back to the rest of the code

# Jumps and Branches

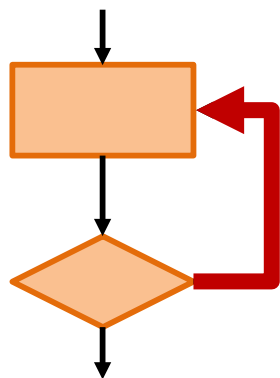
- A common but far from universal distinction
  - **Jumps** → **unconditional** control transfer instructions
  - **Branches** → **conditional** control transfer instructions
- This is the RISC-V convention (inherited from MIPS and used by SPARC, Alpha, etc.)
- Other processors do not. In **x86**, for instance, everything is a jump: JMP, JZ, JC, JNO

# Pseudoinstructions

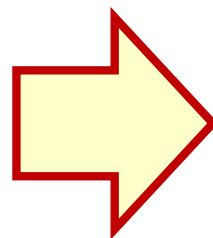
Pseudoinstruction	Base Instruction(s)	Meaning
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if $\neq$ zero
blez rs, offset	bge x0, rs, offset	Branch if $\leq$ zero
bgez rs, offset	bge rs, x0, offset	Branch if $\geq$ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if $\leq$
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if $\leq$ , unsigned

The processor implements only < and  $\geq$   
and the assembler “creates”  $\leq$  and >

# An DO-WHILE Loop



```
do {  
    x5 = x5 >> 1;  
    x6 = x6 + 1;  
} while (x5 != 0);  
...
```



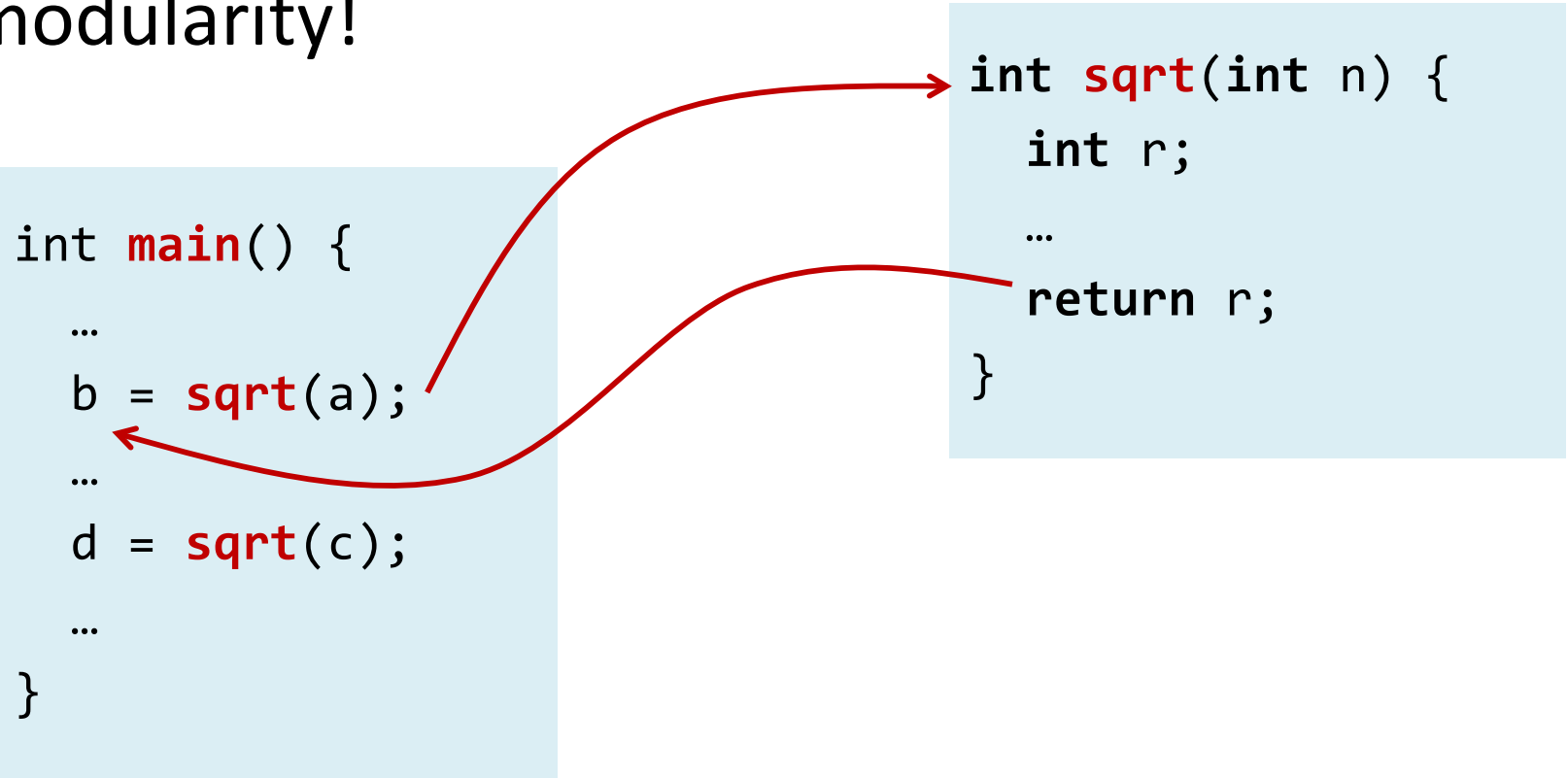
```
.text  
  
loop:  
    srli x5, x5, 1  
    addi x6, x6, 1  
    bnez x5, loop  
  
...
```

# Functions

- Our high-level code is conveniently organized in **functions** (a.k.a. **routines, subroutines, procedures, methods**, etc.)
- Reuse and modularity!

```
int main() {  
    ...  
    b = sqrt(a);  
    ...  
    d = sqrt(c);  
    ...  
}
```

```
int sqrt(int n) {  
    int r;  
    ...  
    return r;  
}
```

The diagram consists of two light blue rectangular boxes containing code. The left box contains the code for a `main` function, and the right box contains the code for a `sqrt` function. Two red curved arrows originate from the `sqrt` function code in the right box. One arrow points from the `sqrt` function definition to the `sqrt(a)` call in the `main` function. The other arrow points from the `sqrt` function definition to the `sqrt(c)` call in the `main` function.

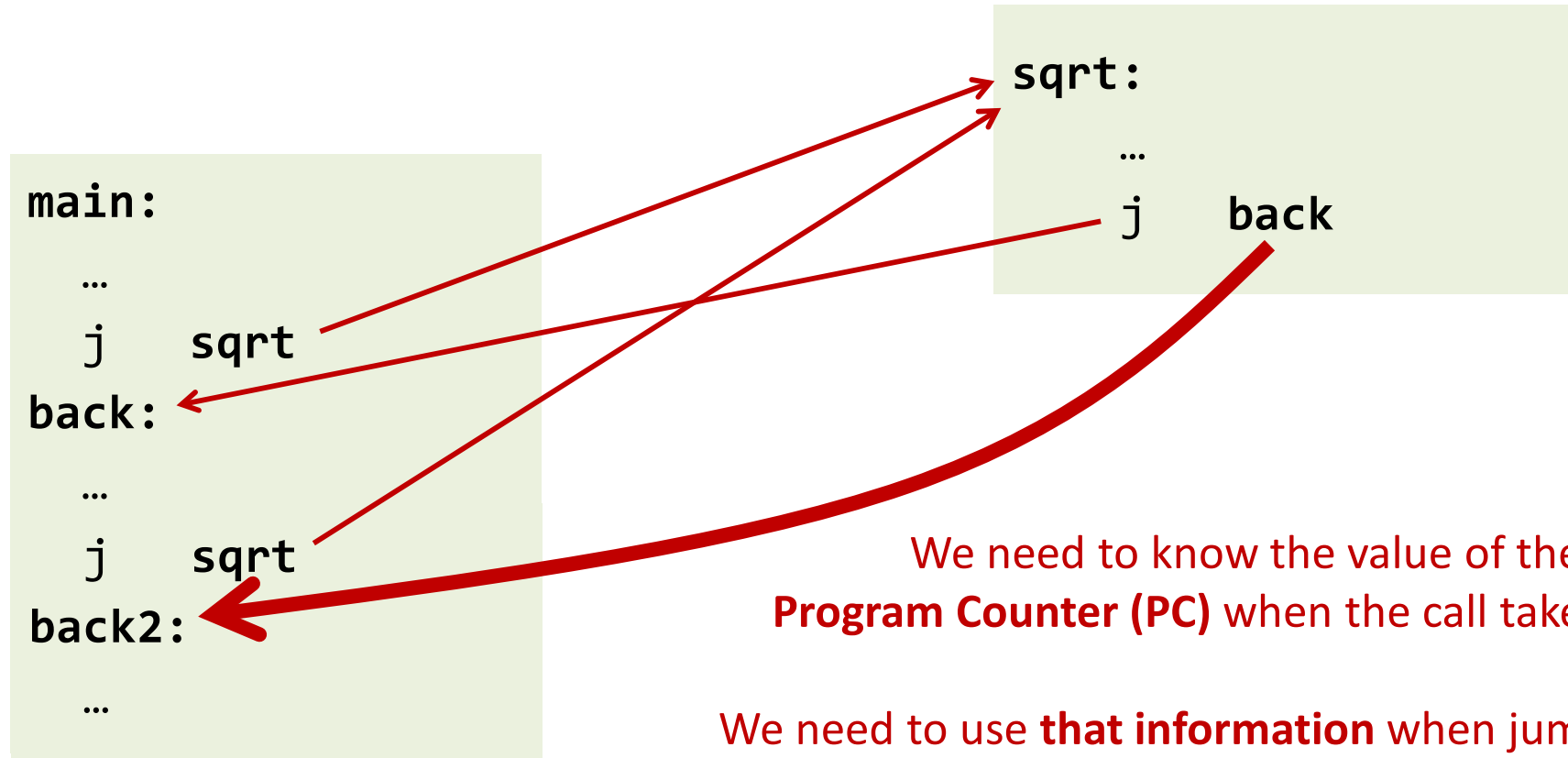
# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

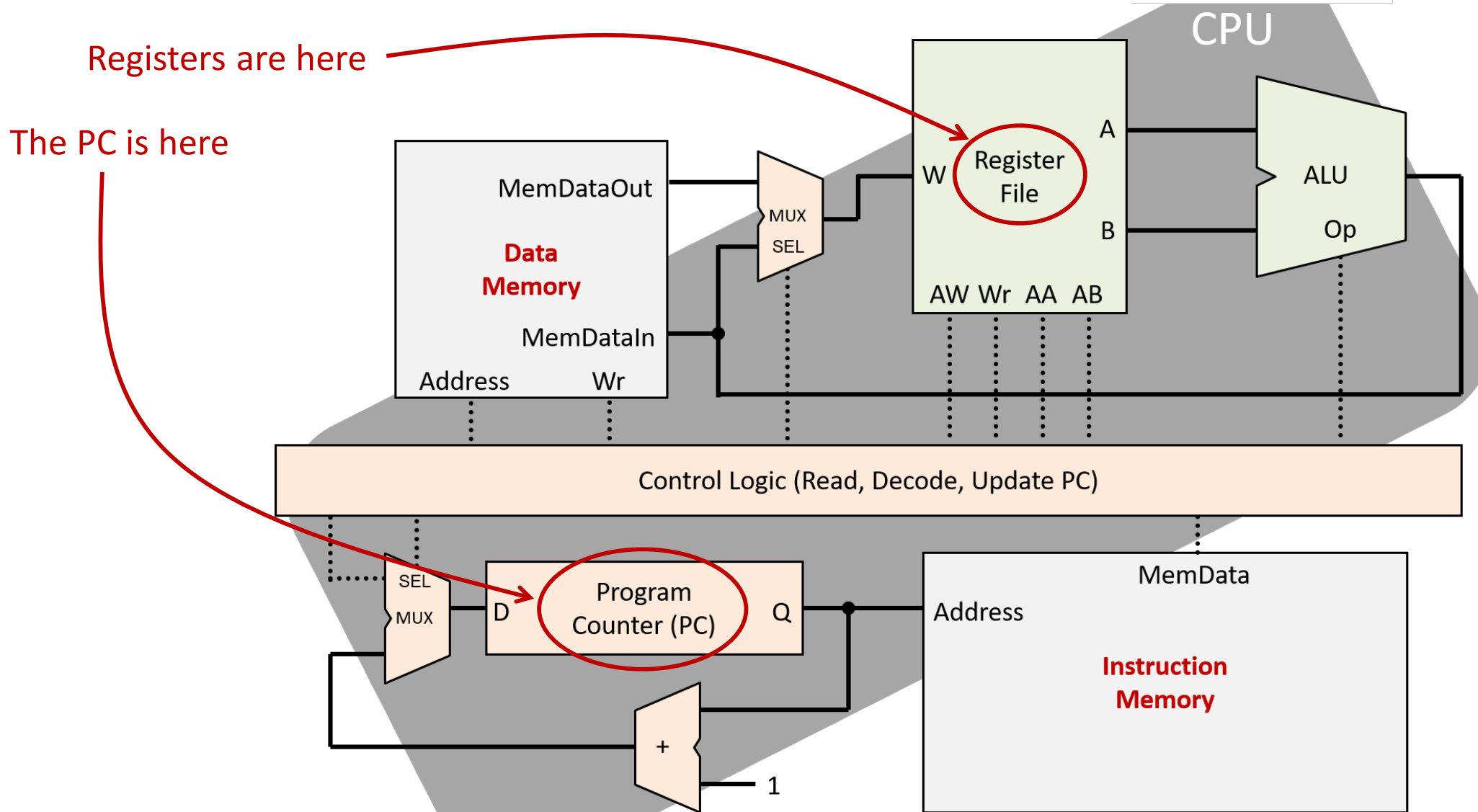
# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

# A Too Simple (i.e., Not Working) Approach



# The PC Is Not a Normal Register



# The Good Approach

**Jump and link,**  
that is, leave PC + 4 into **x1**  
as a “link” to the caller

```
main:  
1200  ...  
1204  jal  x1, sqrt  
1208  ...  
  
2416  ...  
2420  jal  x1, sqrt  
2424  ...
```

```
sqrt:  
3648  ...  
4896  ...  
4900  jr   x1
```

**x1 ← 1208**

**Jump** to the address  
specified in a register

# The Good Approach

**Jump and link,**  
that is, leave PC + 4 into **x1**  
as a “link” to the caller

```
main:  
1200  ...  
1204  jal x1, sqrt  
1208  ...  
  
2416  ...  
2420  jal x1, sqrt  
2424  ...
```

```
sqrt:  
3648  ...  
4896  ...  
4900  jr x1
```

**x1 ← 2424**

**Jump** to the address  
specified in a register

# Only Two Real Jump Instructions!

---

## Jump and link

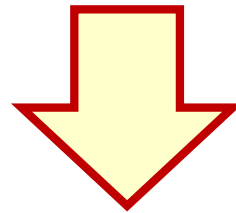
```

jal  rd,imm      rd ← pc + 4
                    pc ← pc + sext(imm << 1)

jalr rd,rs1,imm  rd ← pc + 4
                    pc ← (rs1 + sext(imm)) & (~1)

```

---



In RISC-V, `imm(reg)`  
simply means `reg+imm`

Pseudoinstr.	Base Instruction(s)	Meaning
<code>j offset</code>	<code>jal x0, offset</code>	Jump
<code>jal offset</code>	<code>jal x1, offset</code>	Jump and link
<code>jr rs</code>	<code>jalr x0, 0(rs)</code>	Jump register
<code>jalr rs</code>	<code>jalr x1, 0(rs)</code>	Jump and link register
<code>ret</code>	<code>jalr x0, 0(x1)</code>	Return from subroutine

By **convention** RISC-V uses always  
register `x1` to store the return address

# Register Conventions

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No

We will understand  
this column later...



# The Good Approach

**call**

in many other ISAs

```
main:
1200  ...
1204  jal  sqrt
1208  ...

2416  ...
2420  jal  sqrt
2424  ...
```

**sqrt:**

```
3648  ...
4896  ...
4900  ret
```

**ret**

almost universally

# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program

# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program ✓



# Calling a Function from a Function

```
main:
1200  ...
1204  jal  sqrt
1208  ...

2416  ...
2420  jal  sqrt
2424  ...
```

```
sqrt:
3648  ...

4216  add  x5, x7, x8
4220  jal  round
4224  sub  x6, x6, x5

4896  ...
4900  ret
```

```
round:
5248  ...
5252  addi x5, x11, 3

5964  ...
5968  ret
```

Who can use x5?!

# A Very Very Simple Approach

```
main:  
1200    ...  
1204    jal    sqrt  
1208    ...  
  
2416    ...  
2420    jal    sqrt  
2424    ...
```

```
sqrt:  
3648    ...  
  
4216    add    x5, x7, x8  
4220    jal    round  
4224    sub    x6, x6, x5  
  
4896    ...  
4900    ret
```

```
round:  
5248    ...  
5252    addi   x10, x11, 3  
  
5964    ...  
5968    ret
```

**round** can  
**only** use **x10** to **x15**

**sqrt** can  
**only** use **x2** to **x9**

Clearly not scalable...

# What Calling a Function Involves

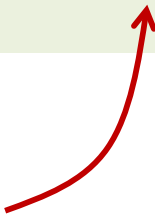
1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program ✓

# A Simple Approach

.data

```
sqrt_save_ra:    .word 0  
sqrt_save_x5:   .word 0
```

Obtain some  
memory space



.text

**sqrt:**

...

```
add x5, x7, x8
```

```
sw ra, (sqrt_save_ra)
```

```
sw x5, (sqrt_save_x5)
```

```
jal round
```

```
lw ra, (sqrt_save_ra)
```


```
lw x5, (sqrt_save_x5)
```

```
sub x6, x6, x5
```

...

```
ret
```

These load/store  
instructions with immediate  
addresses do not exist!



Preserve what we care  
before calling

Restore after returning

# Problem: A Recursive Function!

```
.data

find_child_save_ra:    .word 0

.text

find_child:
...
sw    ra, (find_child_save_ra)
jal   find_child
lw    ra, (find_child_save_ra)
...
ret
```

Back to square one!

Every invocation of `find_child` saves its return address in `find_child_save_ra`, overwriting the value of the caller...



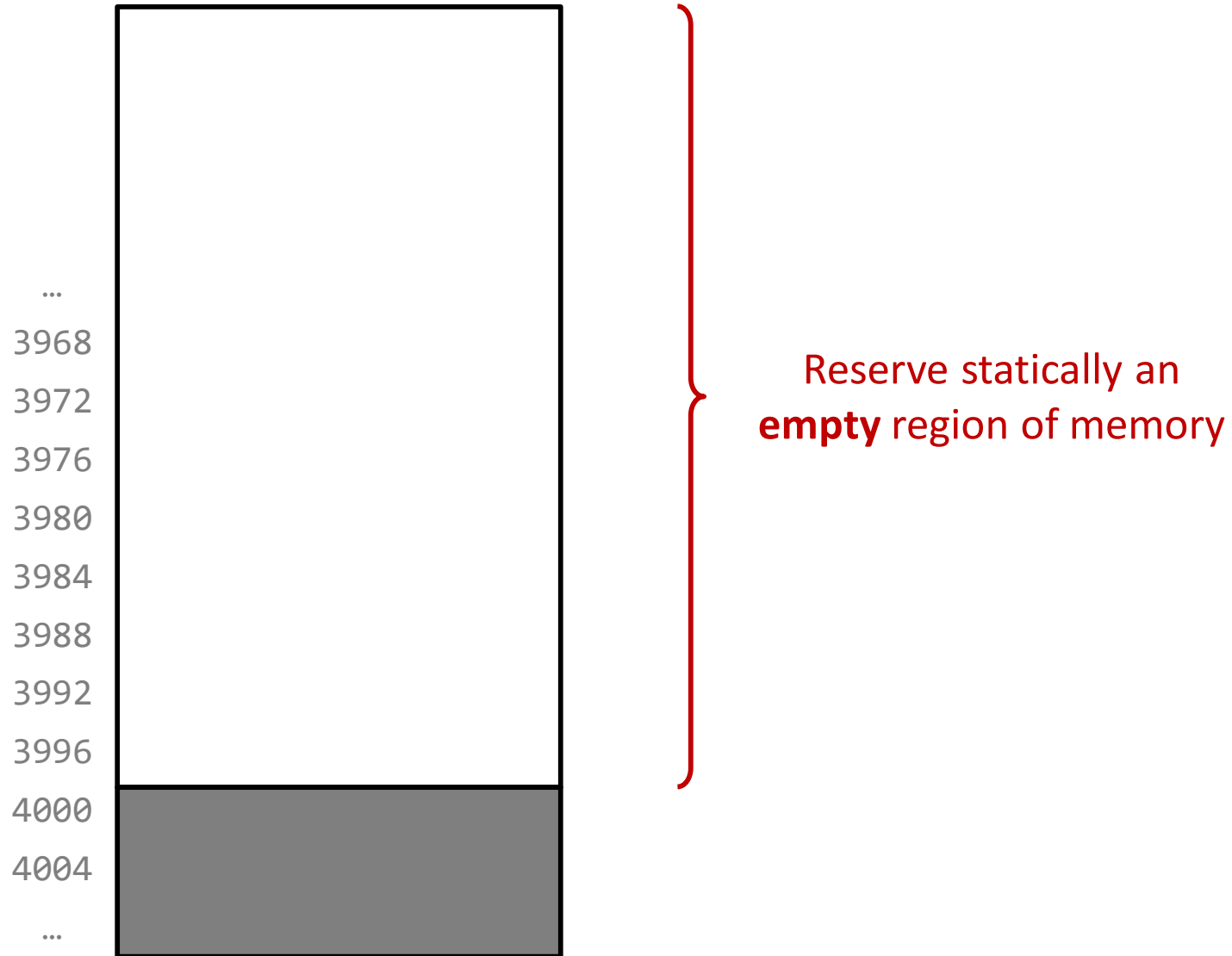
# What Is the Problem?

- Static memory allocation vs. dynamic
  - **Static** → fixed at assembly / compile / program writing time
  - **Dynamic** → fixed during execution
- This is static!

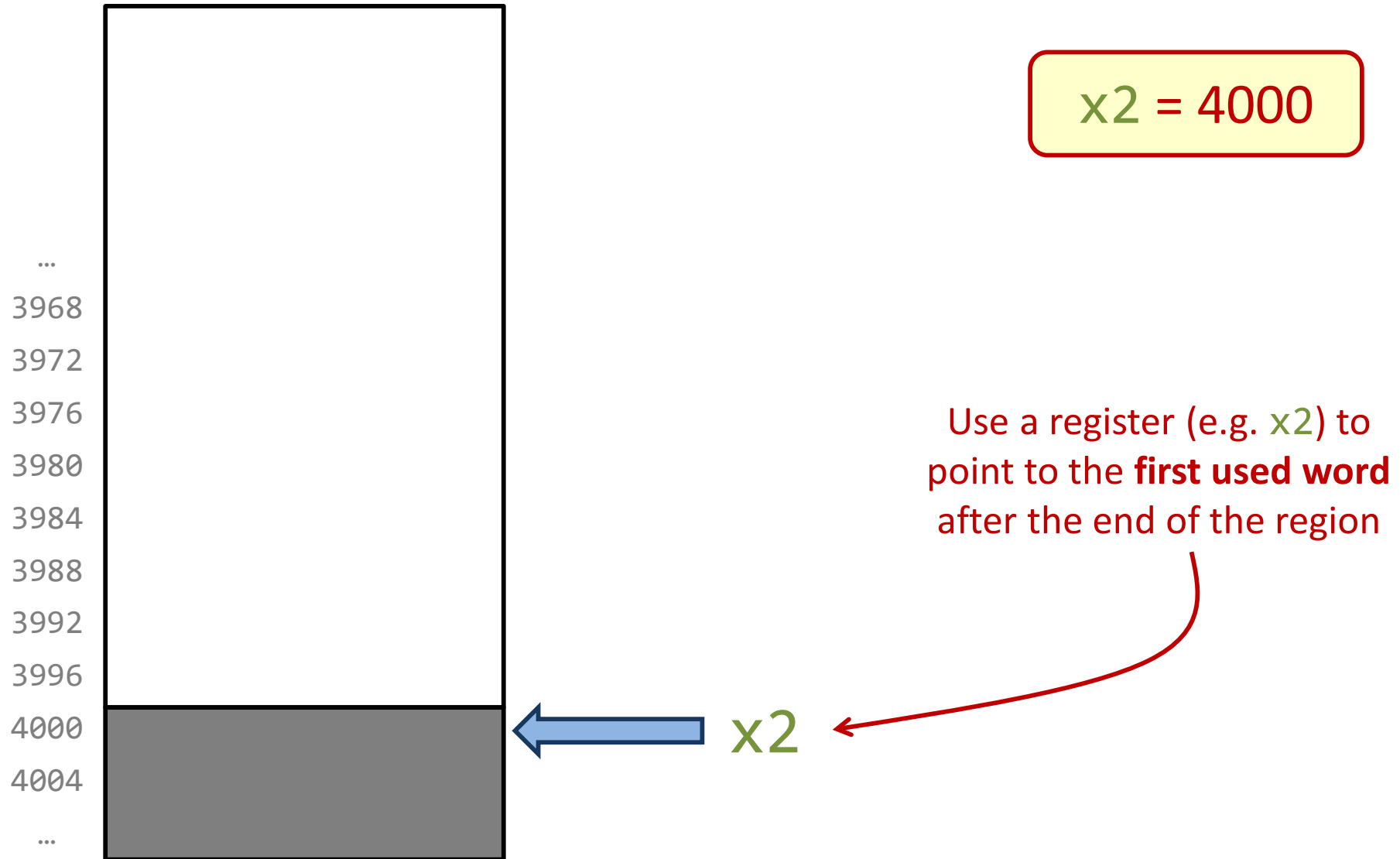
```
.data  
  
find_child_save_ra:    .word 0
```

- **Every invocation** of the function needs a **new place** to store the return address and their data

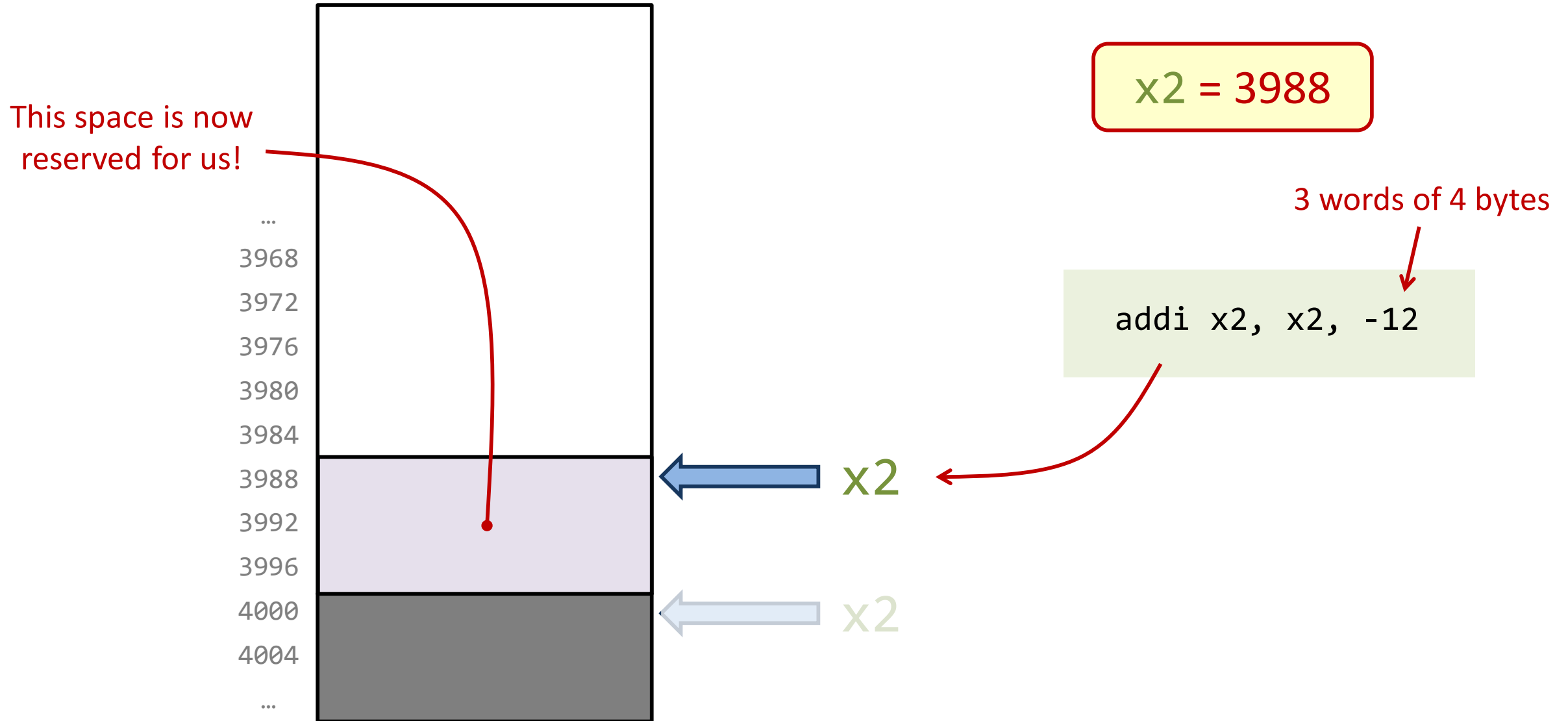
# The Idea of a Stack



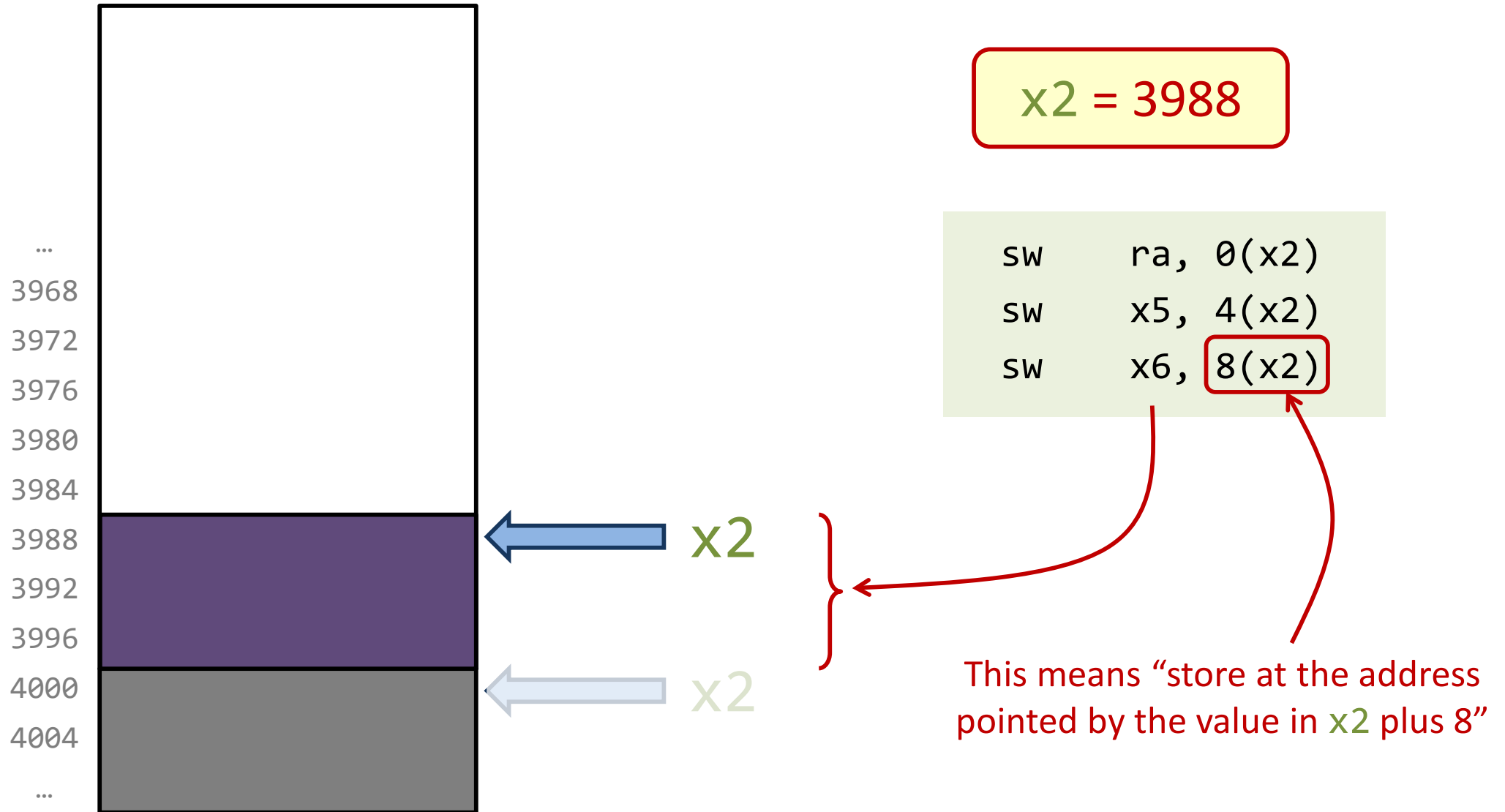
# The Idea of a Stack



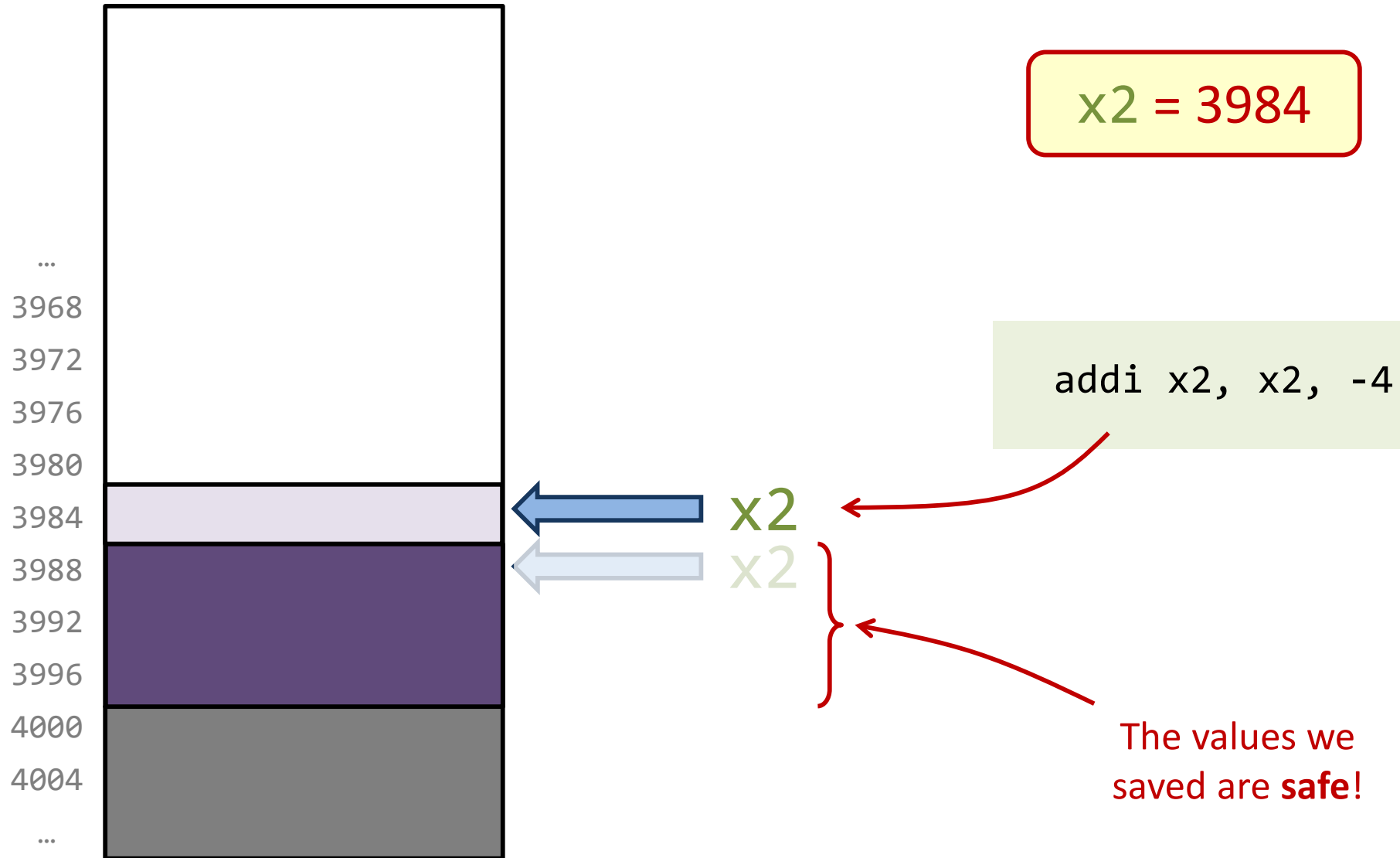
# Dynamically Allocating Space (e.g., 3 Words)



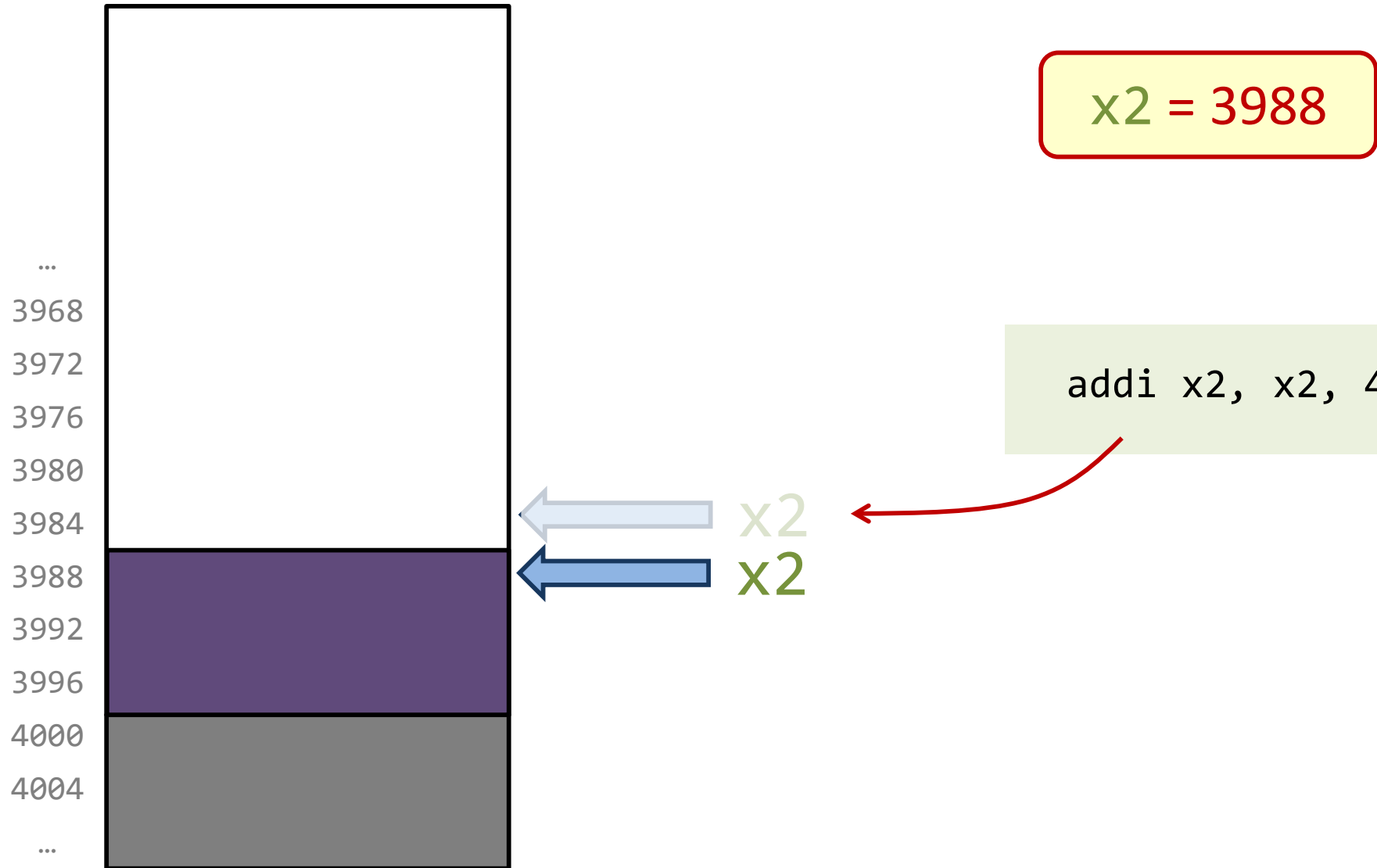
# Using the Space



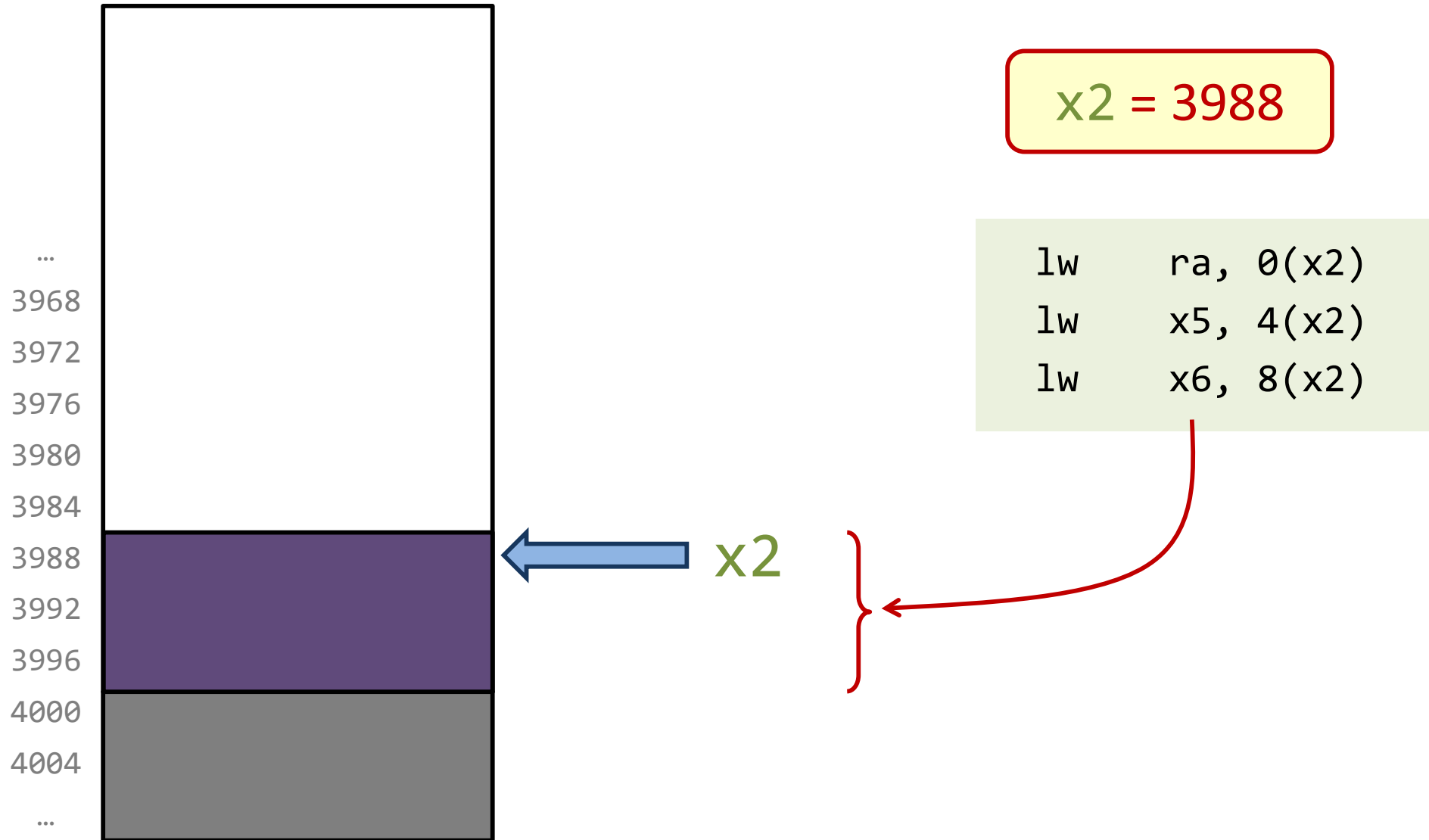
# Dynamically Allocating More Space



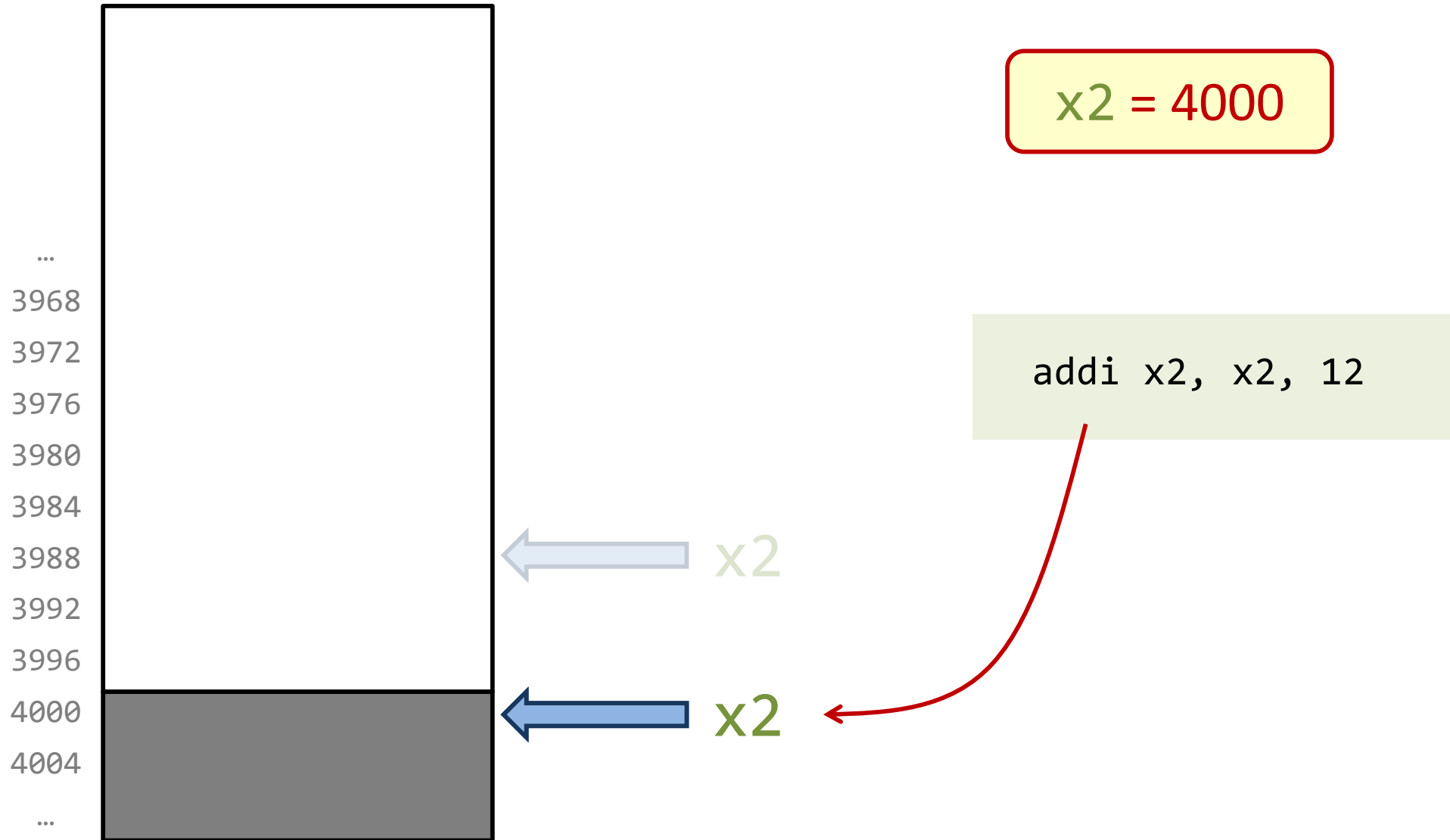
# Deallocating Space Is Simple



# Retrieve Saved Values



# More Deallocation



# Stack: Limited but Effective

- The simplest form of **dynamic memory allocation**
- Simplicity comes with a big limitation
  - **Last-In, First-Out (LIFO)**
  - Deallocation order must be the inverse of the allocation order!
- Yet, **perfectly matched** to our application (**function calls**)
- Other needs will require much more complex approaches that generally need more “management” and need to deal with **garbage collection**

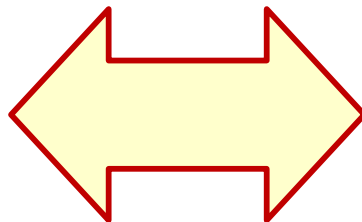
# Stack Pointer

- This is so important that we are going to devote a register to this purpose and **everybody** will comply with our **conventions**

Register	ABI Name	Description	Preserved across call?
x2	sp	Stack pointer	Yes

- Other architectures have special instructions to place stuff on the stack (**push**) and to retrieve it (**pop**)

PUSH AX



```
addi    sp, sp, -4
sw      x5, 0(sp)
```

# Spilling Registers to Memory

```
...  
addi sp, sp -8  
sw x8, 0(sp)  
sw x9, 4(sp)  
...
```

```
# freely use  
# x8 and x9
```

```
...  
lw x9, 4(sp)  
lw x8, 0(sp)  
addi sp, sp, 8  
...
```

Whoever needs to free registers, can obtain some space from the stack

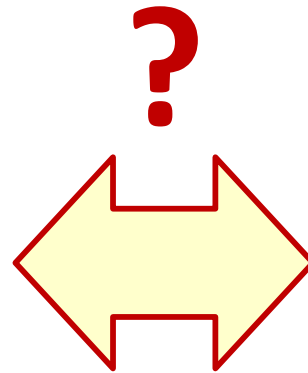
In particular, functions can save their return address (if they call other functions)

```
sqrt:  
addi sp, sp -4  
sw ra, 0(sp)  
...  
# freely call  
# other functions  
...  
lw ra, 0(sp)  
addi sp, sp, 4  
ret
```

# Registers across Functions

Functions **change registers**  
and callers save their stuff

```
...  
addi  sp, sp -4  
sw    x20, 0(sp)  
      # sqrt changes x20  
jal   sqrt  
lw    x20, 0(sp)  
addi  sp, sp, 4  
...
```



Functions **preserve registers**

```
sqrt:  
  addi  sp, sp -4  
  sw    x20, 0(sp)  
  ...  
      # uses freely x20  
  ...  
  lw    x20, 0(sp)  
  addi  sp, sp, 4  
  ret   # x20 is preserved
```

It does not matter (much) provided that **everyone agrees...**

# Preserving Registers: The RISC-V Way

- A bit of both

Of course!

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

Saved registers are  
**callee** saved

Temporary registers are  
**caller** saved

# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program ✓

# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources
7. Return control to the calling program ✓

# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs ✓
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources ✓
7. Return control to the calling program ✓

# Passing Arguments: Option 1

- Use some particular registers, both for the **arguments** and for the returned **result**
- We can do it **ad-hoc**...
  - **sqrt** gets the argument in **x5** and returns the result in **x6**
- ...or we can have some **convention**
  - All functions pass arguments in registers **x10** to **x17** and return the result in **x10**
- Can this be insufficient? **More arguments** than allocated registers? What if we have 10 arguments for **x10** to **x17**?

# Passing Arguments: Option 2

```
...  
addi sp, sp, -4  
sw    s3, 0(sp)      # s3 = arg  
call  sqrt  
...  
  
sqrt:  
addi sp, sp, -4  
sw    ra, 0(sp)  
...  
lw    t4, 4(sp)     # t4 = arg  
...  
ret
```

Offset from **sp** changes because **sp** changes

- We can put them on the **stack**
- **Universal** solution (the stack is “unlimited”)
- **More work** than simply using registers, though...
- Many commercial processors do that

# Passing Arguments: Option 2 (+ fp)

```
...  
addi sp, sp, -4  
sw    s3, 0(sp)      # s3 = arg  
call  sqrt  
...
```

sqrt:

```
mv    fp, sp  
addi sp, sp, -4  
sw    ra, 0(sp)  
...  
lw    t4, 0(fp)      # t4 = arg  
...  
ret
```

- In addition, one can have another register (**Frame Pointer**; **fp** or **x8** in RISC-V) point to the same location as **sp** on entry
- Code may be more **readable**:
  - **sp** changes inside the function and so do relative offsets
  - Offsets with respect to the **fp** are **fixed**
- Use of **fp** is **optional** and even varies among users and compilers

# Passing Arguments: The RISC-V Way

- A bit of both
  - **Some registers reserved** for the arguments and return value(s)

Register	ABI Name	Description	Preserved across call?
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No

- Rest goes on the **stack**

# Summary of RISC-V Register Conventions

Not covered  
in CS-200

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/ <del>alternate link register</del>	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

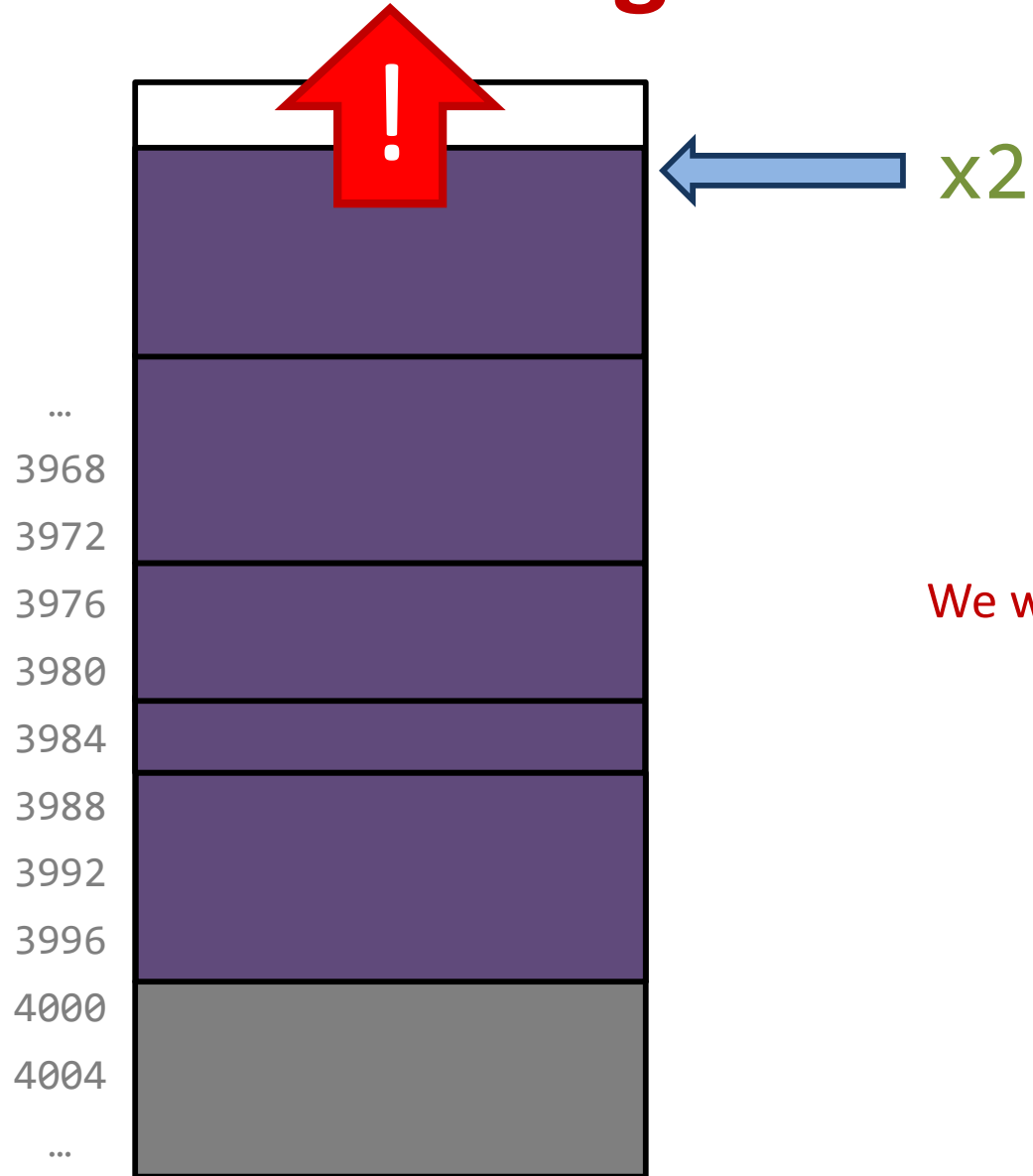
# What Calling a Function Involves

1. Place arguments where the called function can access them
2. Jump to the function ✓
3. Acquire storage resources the function needs ✓
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program
6. Release any local storage resources ✓
7. Return control to the calling program ✓

# What Calling a Function Involves

1. Place arguments where the called function can access them ✓
2. Jump to the function ✓
3. Acquire storage resources the function needs ✓
4. Perform the desired task of the function ✓
5. Communicate the result value back to the calling program ✓
6. Release any local storage resources ✓
7. Return control to the calling program ✓

# Something to Worry About...



We will definitely need to find a solution!  
(In a few weeks' time...)

# References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 2** and, in particular, **Sections 2.6, 2.7, and 2.8**