CS 31: Introduction to Computer Systems

08: Computer Architecture

02-13-2025

"If you can do logic gates in your head, please confirm you are not a replicant"

http://smbc-comics.com/comic/logic-gates

Announcements

- Lab 3 Checkpoint due today. It will be graded.
- Let me know if your HW group is not added (via email)

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

Check your frequency:

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:

1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success

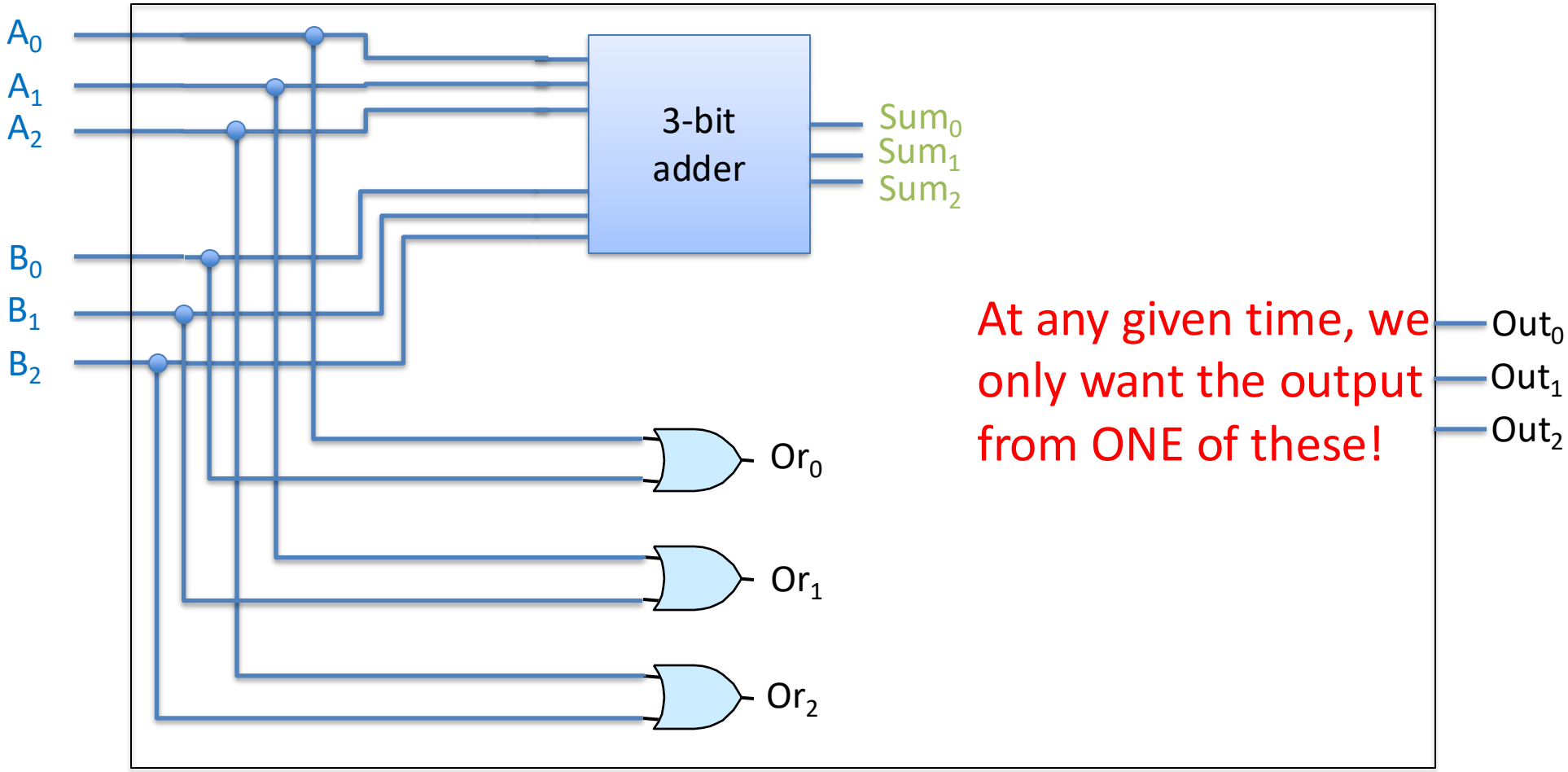
Abstraction!

- Hide away the complex internals of *how* the system functions, and focus on *what* functionality we expect. I.e., the guaranteed output of a system given the set of allowed inputs, and treating the functionality of the system as a black box.

- What are examples of abstractions you have experienced in daily life?

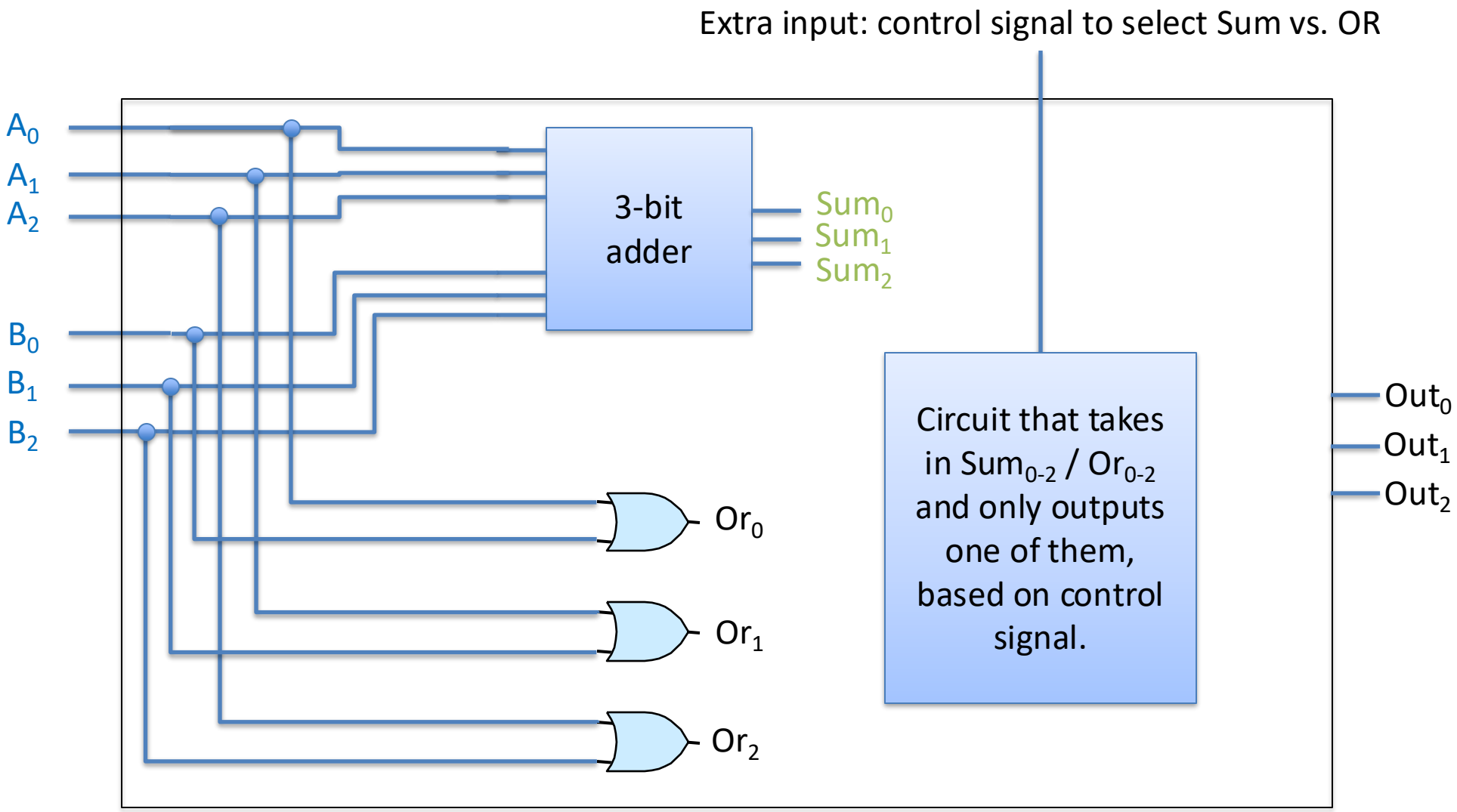
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs A and B:



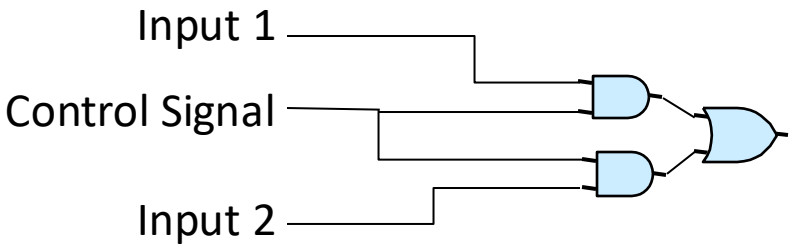
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs A and B:

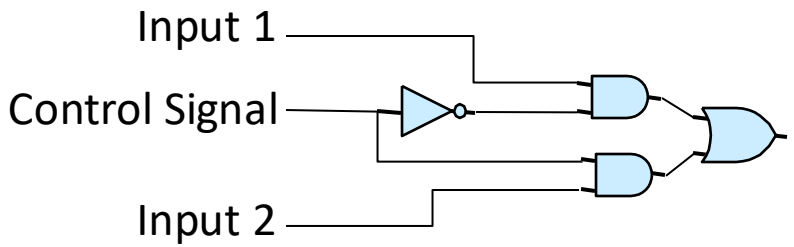


Which of these circuits lets us select between two inputs?

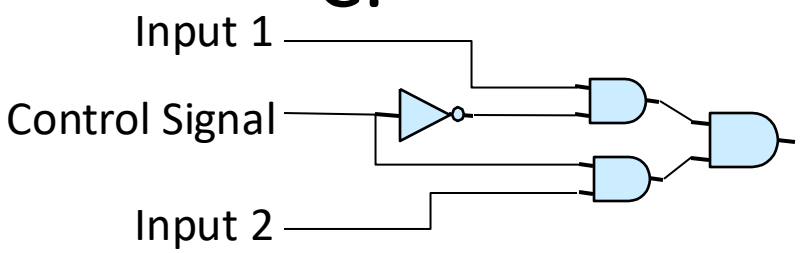
A:



B:

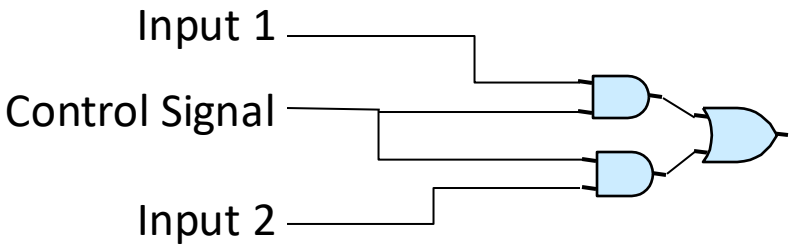


C:

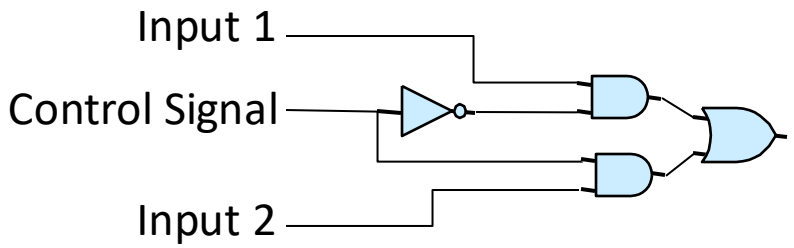


Which of these circuits lets us select between two inputs?

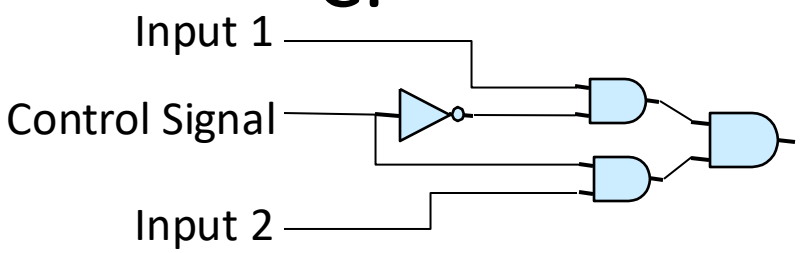
A:



B:

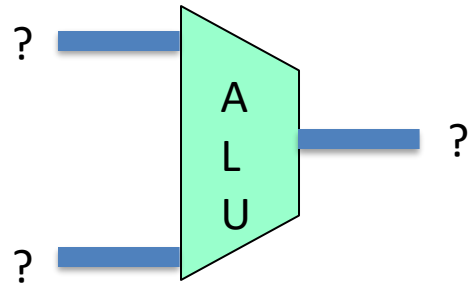


C:



CPU so far…

- We can perform arithmetic!
- Storage questions:
  - Where to the ALU input values come from?
  - Where do we store the result?
  - What does this “register” thing mean?



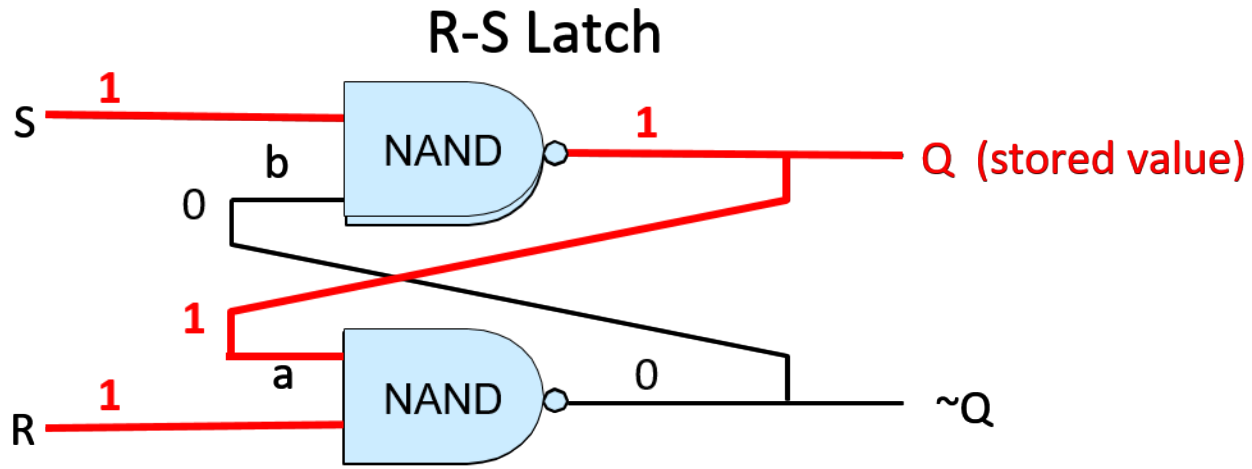
Memory Circuit Goals: Starting Small

- Store a 0 or 1
- Retrieve the 0 or 1 value on demand (read)
- Set the 0 or 1 value on demand (write)

R-S Latch: Stores Value Q

When R and S are both 1: Maintain a value

R and S are never both simultaneously 0

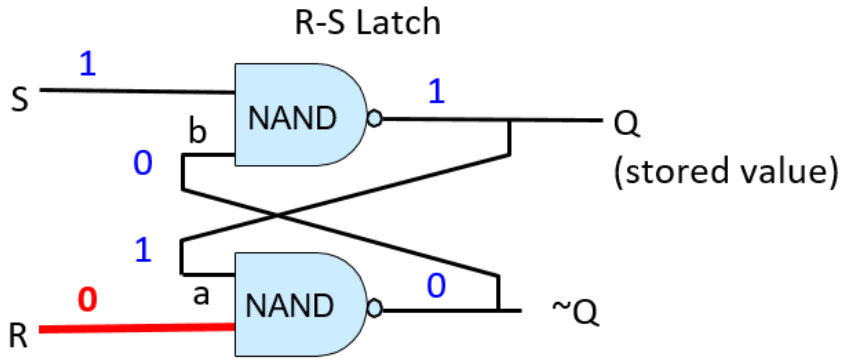


- To write a new value:
  - Set S to 0 momentarily (R stays at 1): to write a 1
  - Set R to 0 momentarily (S stays at 1): to write a 0

R-S Latch: Stores Value Q

Assume that the RS Latch currently stores 1.

To write 0 into the latch, set R's value to 0.

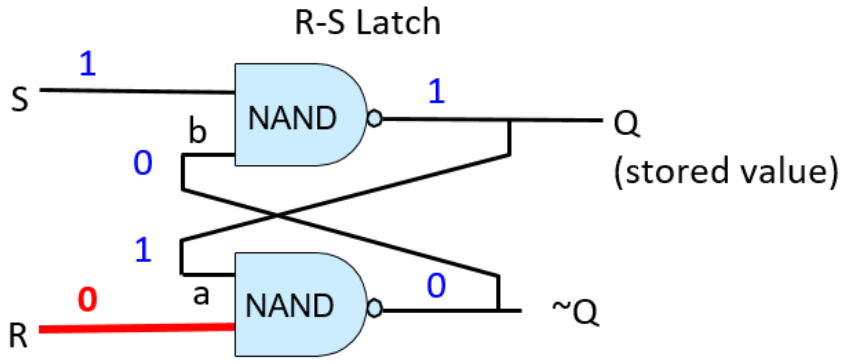


A. Set R to 0 to store 0

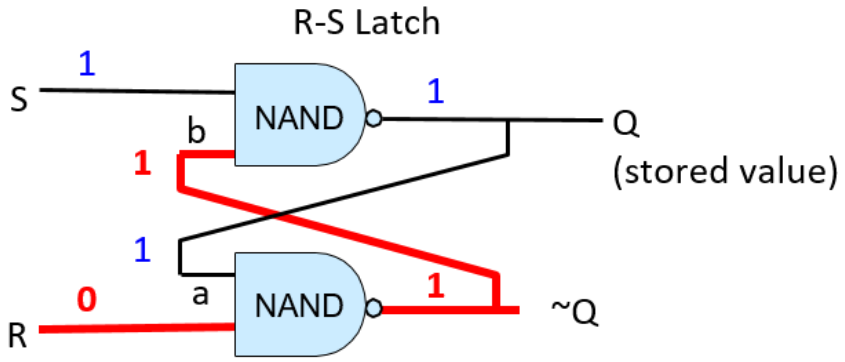
R-S Latch: Stores Value Q

Assume that the RS Latch currently stores 1.

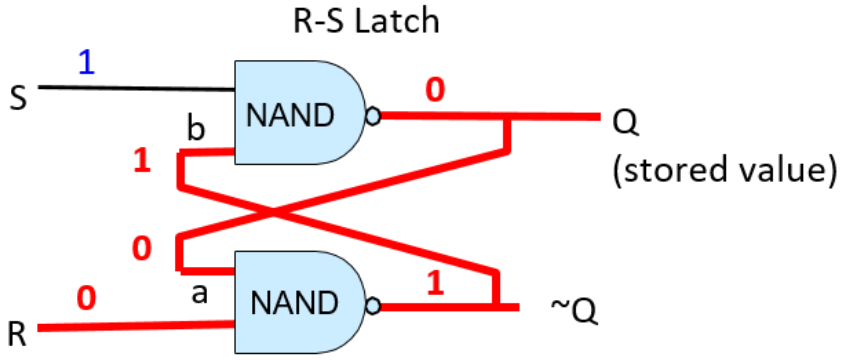
To write 0 into the latch, set R's value to 0 temporarily.



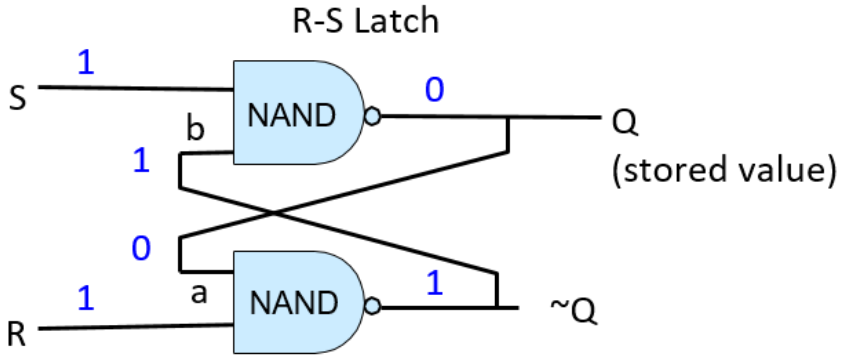
A. Set R to 0 to store 0



B. Changes lower NAND output to 1



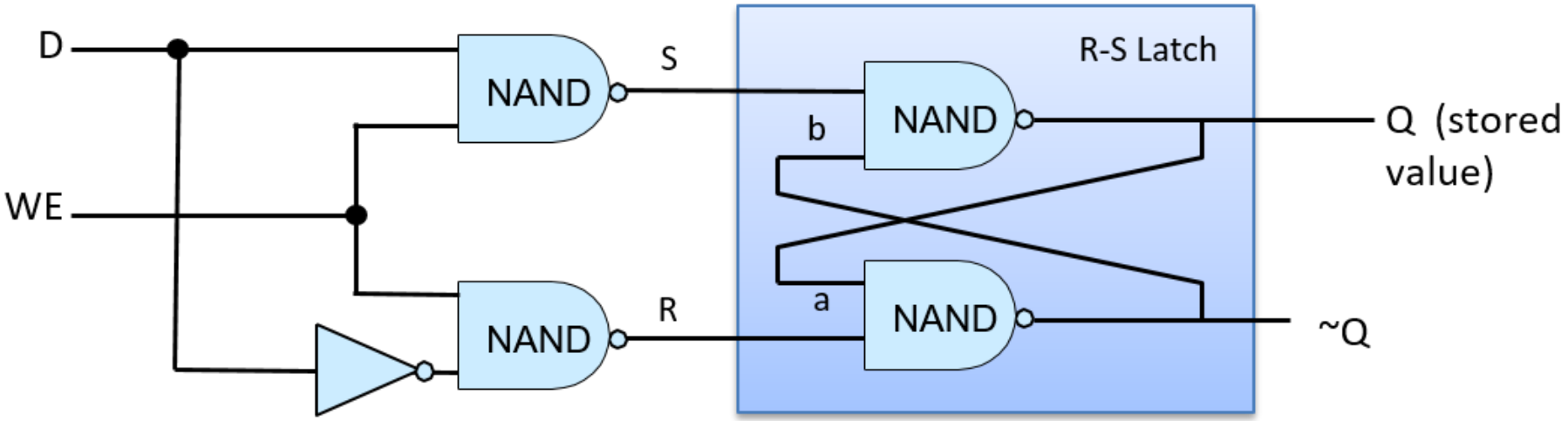
C. Changes upper NAND output to 0



D. R-S Latch Now Stores 0
(R can be set back to 1 and still stores 0)

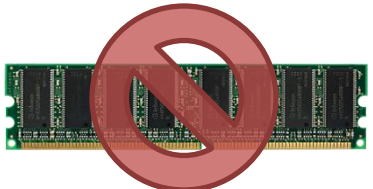
Gated D Latch

Controls S-R latch writing, ensures S & R never both 0



D: into top NAND, ~D into bottom NAND
WE: write-enabled, when set, latch is set to value of D

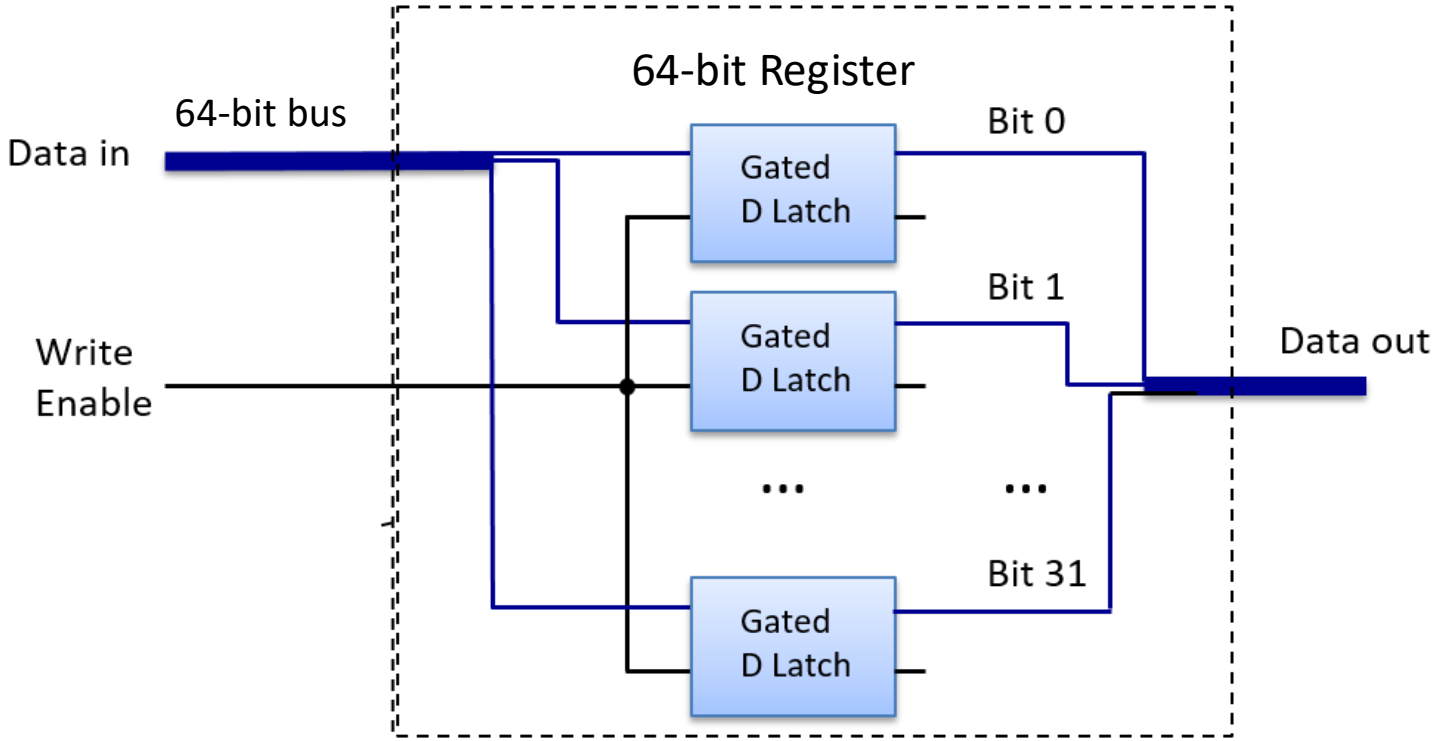
Latches used in registers (up next) and SRAM (caches, later)
Fast, not very dense, expensive
DRAM: capacitor-based



An N-bit Register

- Fixed-size storage (8-bit, 32-bit, 64-bit, etc.)

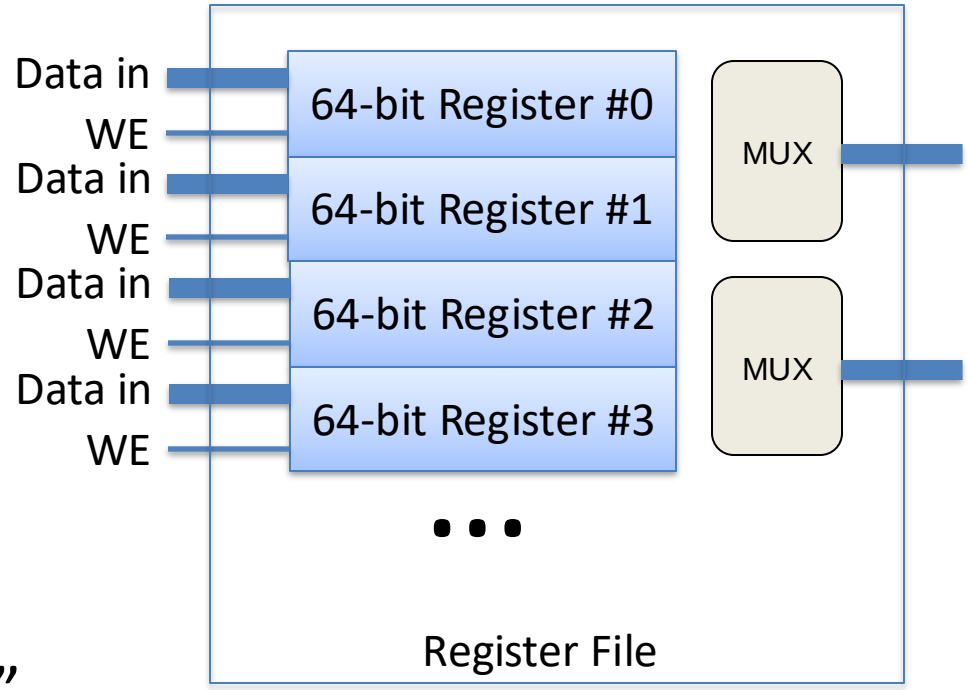
- Gated D latch lets us store one bit
  - Connect N of them to the same write-enable wire!



=

"Register file"

- A set of registers for the CPU to store temporary values.
- This is (finally) something you will interact with!
- Instructions of form:
  - "add R1 + R2, store result in R3"



Memory Circuit Summary

- Lots of abstraction going on here!
  - Gates hide the details of transistors.
  - Build R-S Latches out of gates to store one bit.
  - Combining multiple latches gives us N-bit register.
  - Grouping N-bit registers gives us register file.

- Register file's simple interface:
  - Read R_x's value, use for calculation
  - Write R_y's value to store result

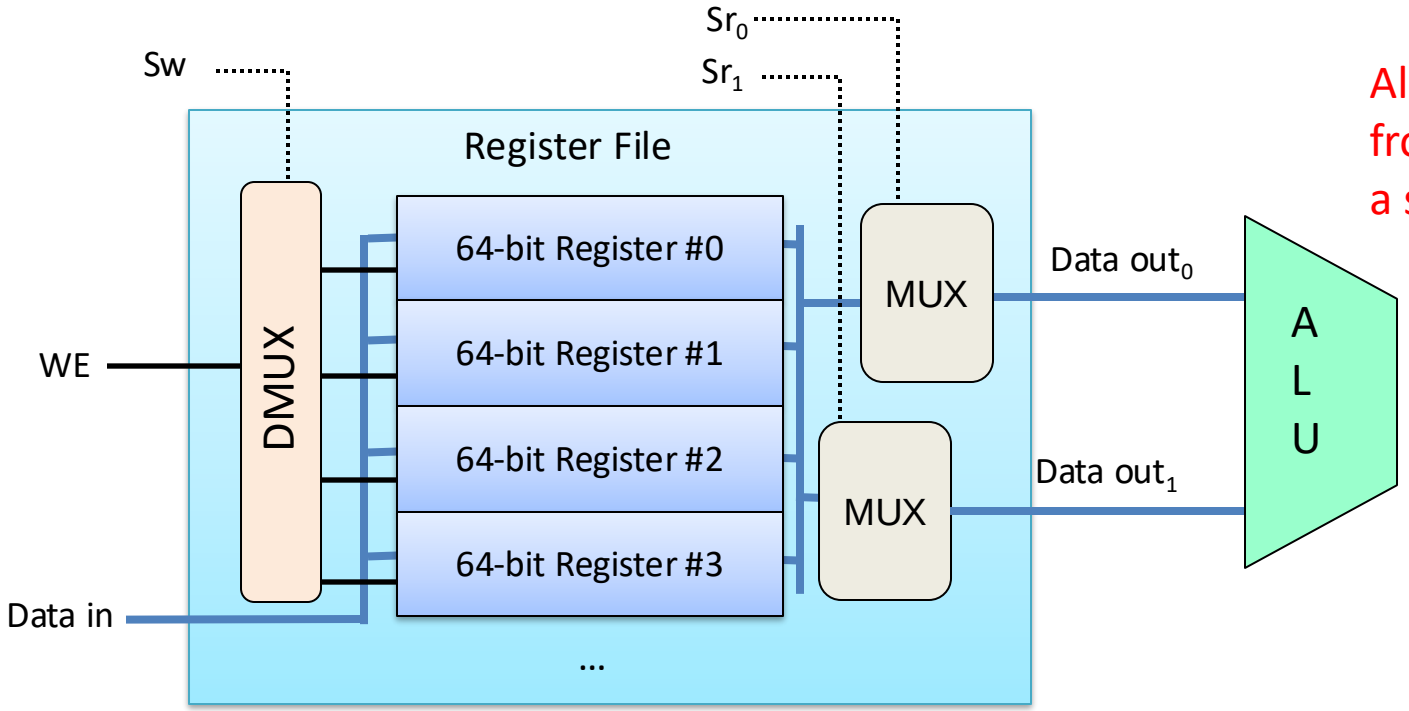
CPU so far…

We know how to store data (in register file).

We know how to perform arithmetic on it, by feeding it to ALU.

Remaining questions:

- Which register(s) do we use as input to ALU?
- Which operation should the ALU perform?
- To which register should we store the result?



All this info comes from the program: a series of instructions.

CPU Game Plan

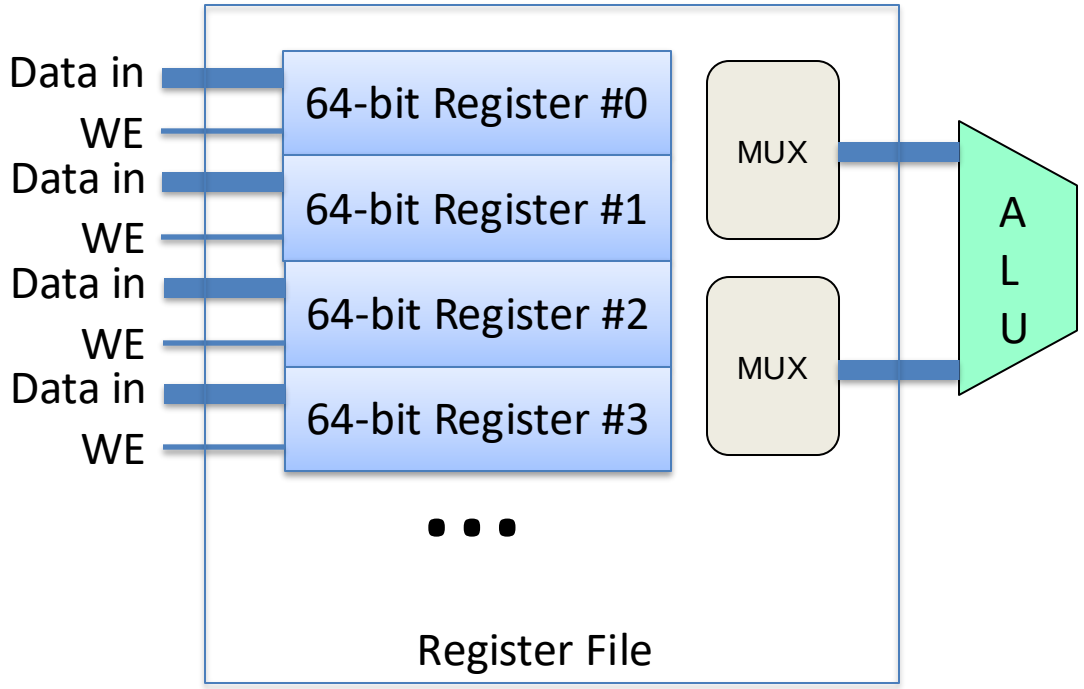
- Fetch instruction from memory
- Decode what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands
- Execute the instruction (arithmetic, etc.)
- Store the result

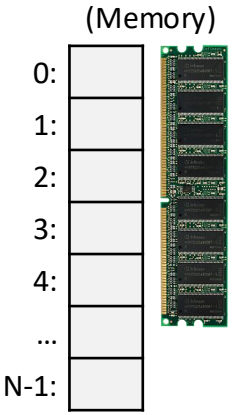
Program State

Let's add two more special registers (not in register file) to keep track of program.

Program Counter (PC): **Memory address of next instr**

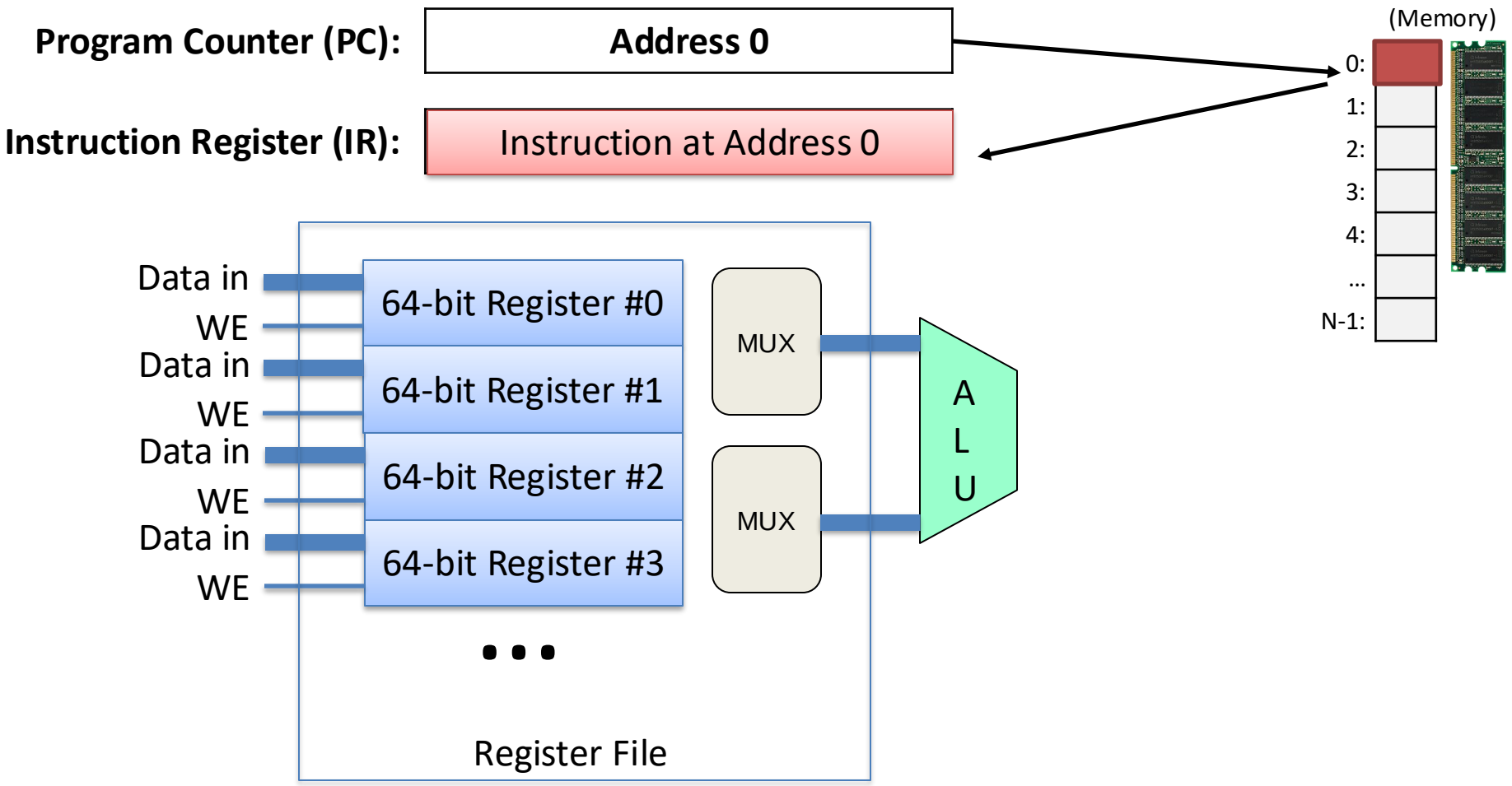
Instruction Register (IR): **Instruction contents (bits)**





Fetching instructions.

Load IR with the contents of memory at the address stored in the PC.

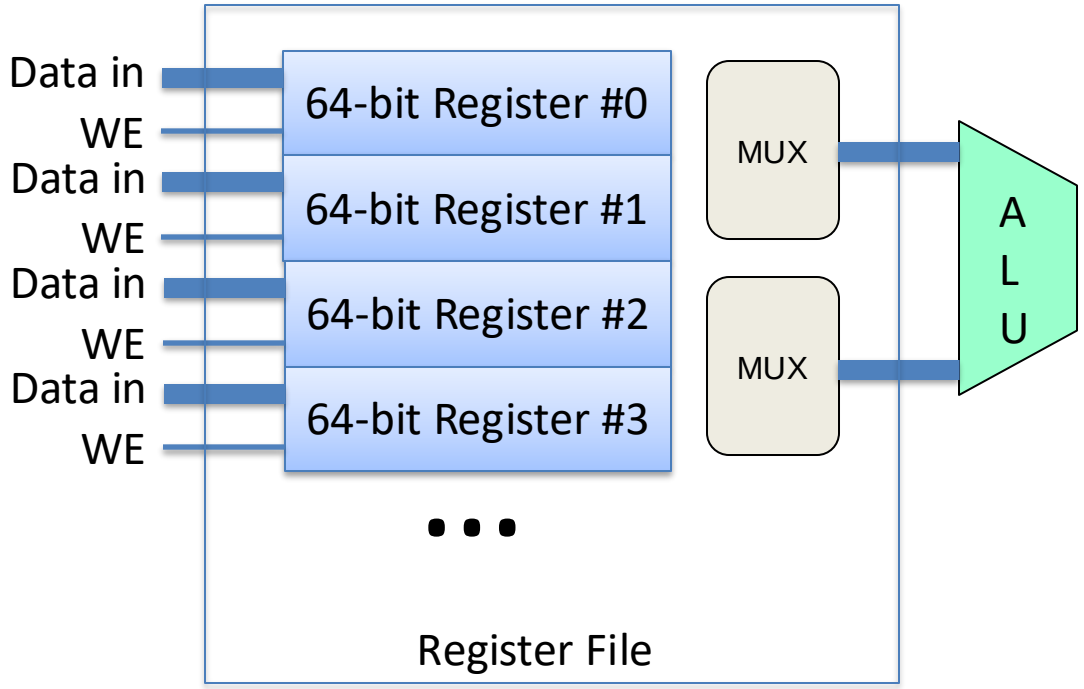


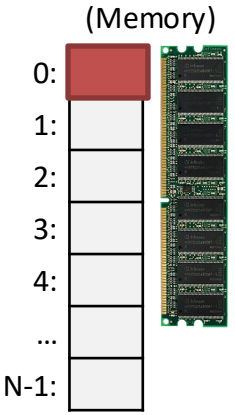
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): **Address 0**

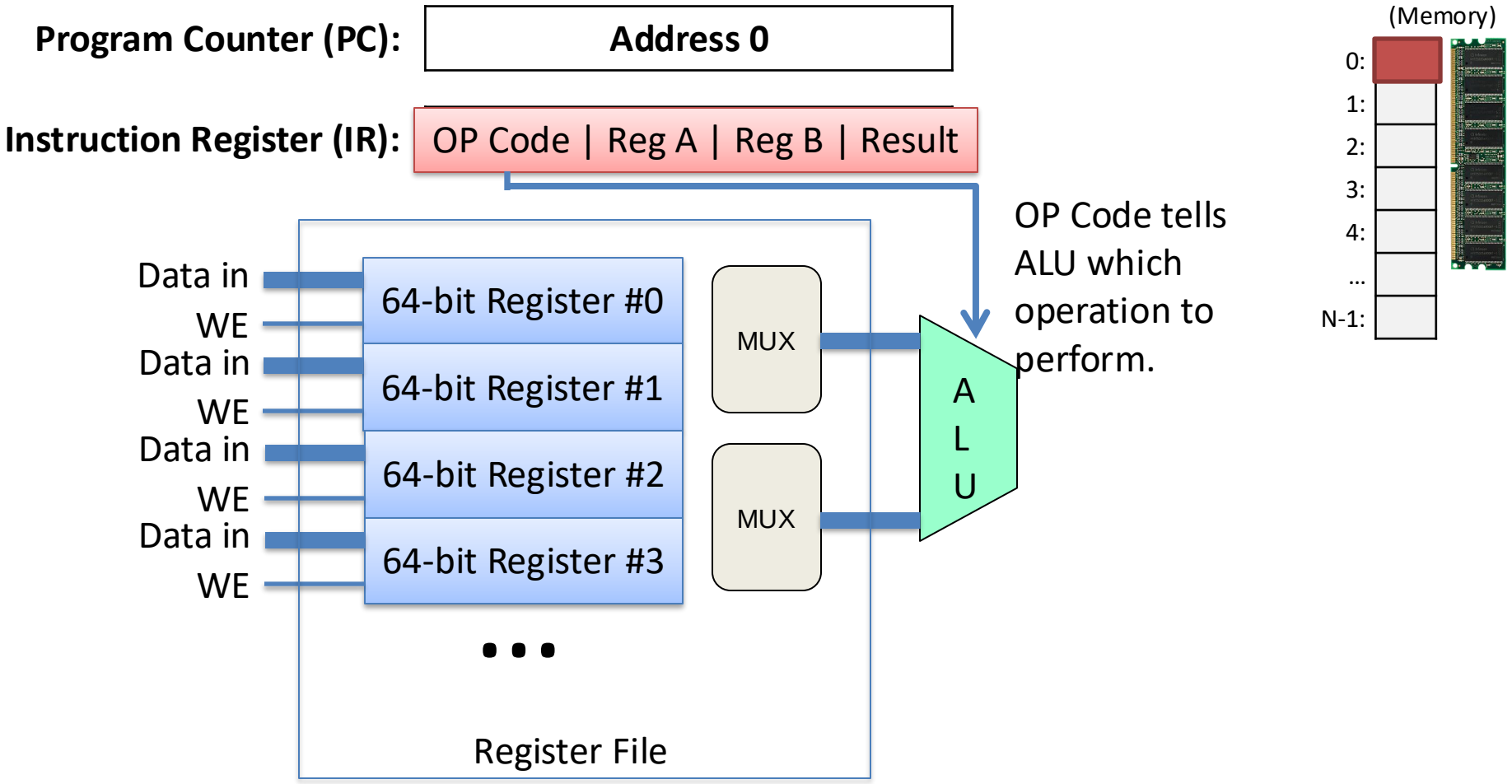
Instruction Register (IR): OP Code | Reg A | Reg B | Result





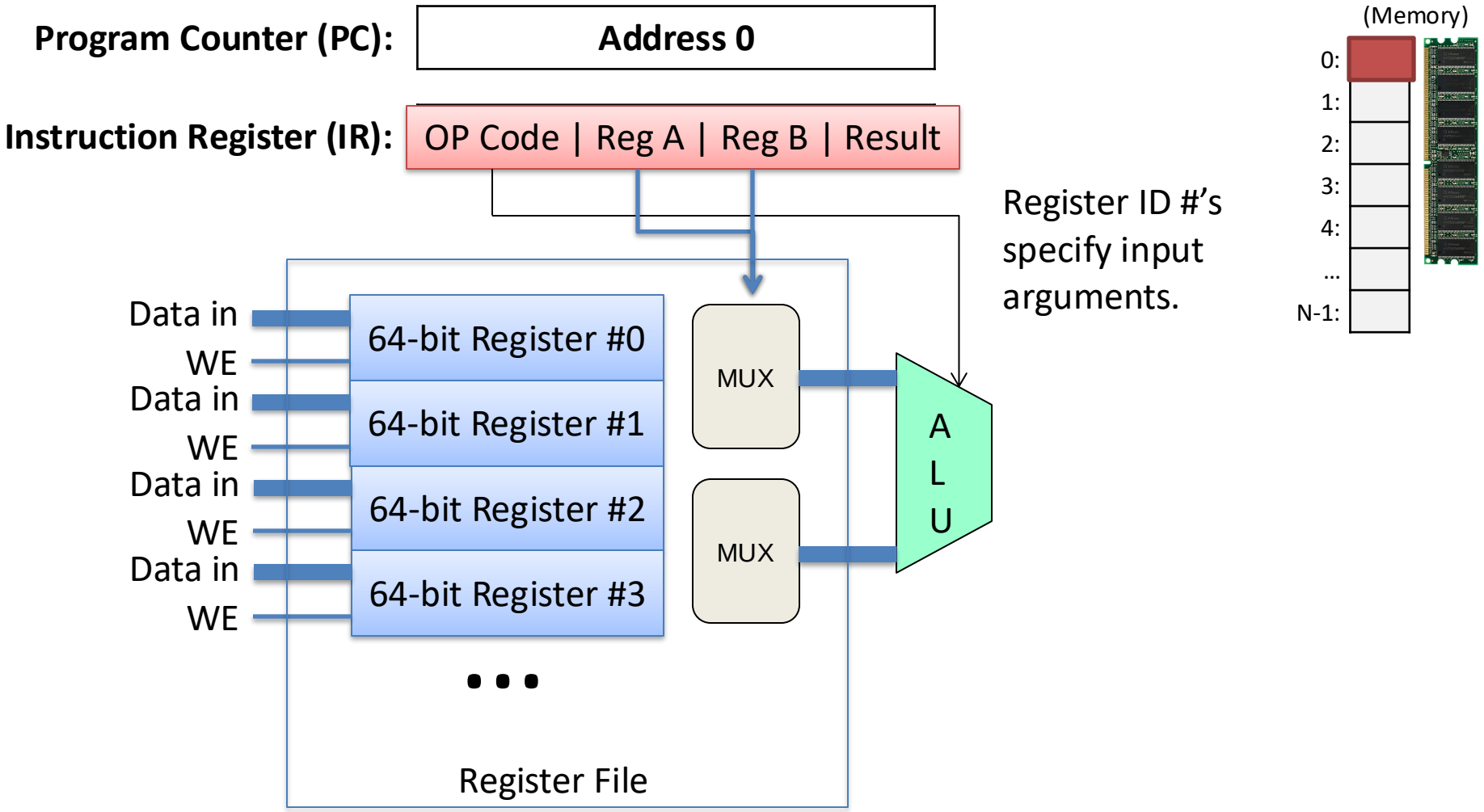
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



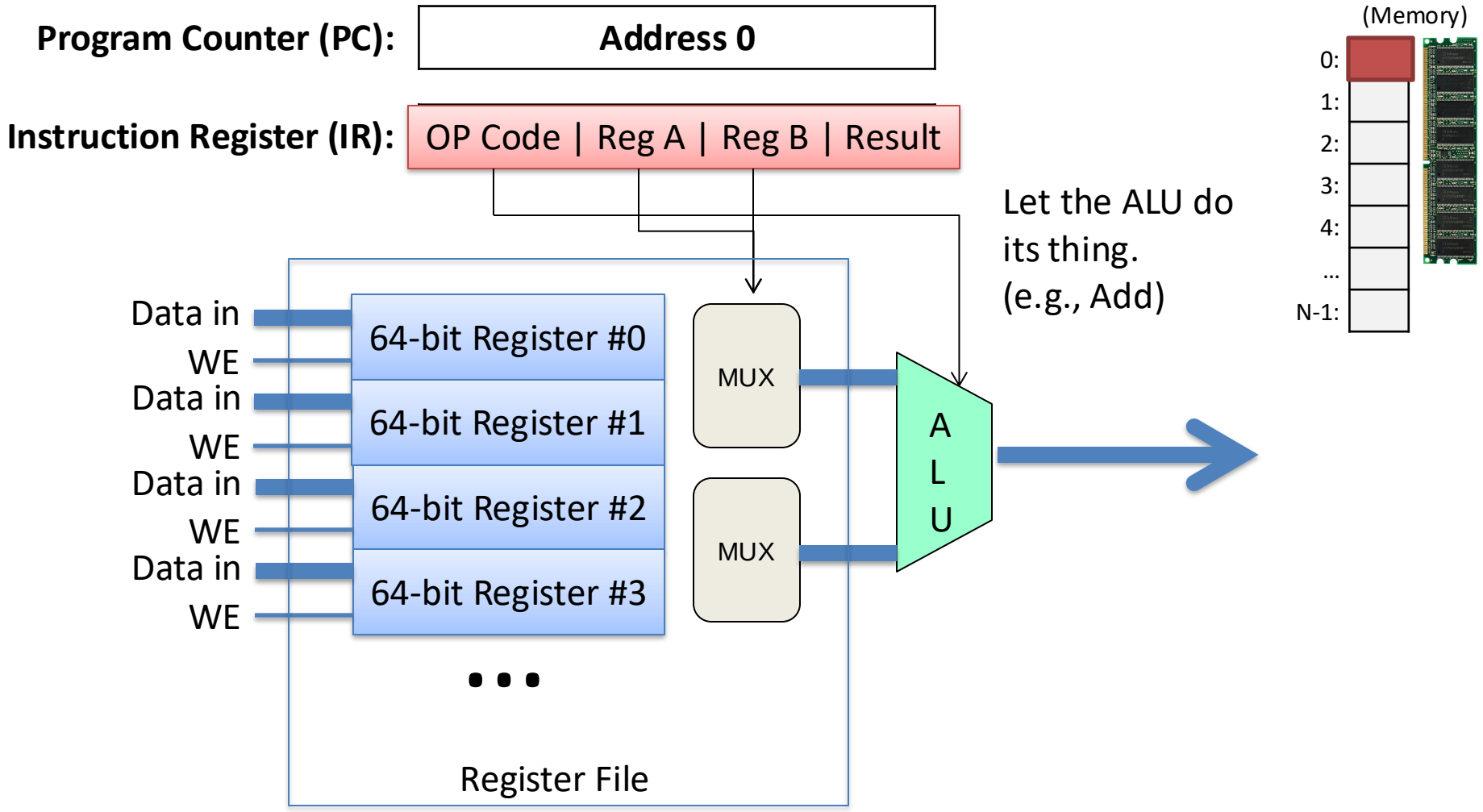
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?



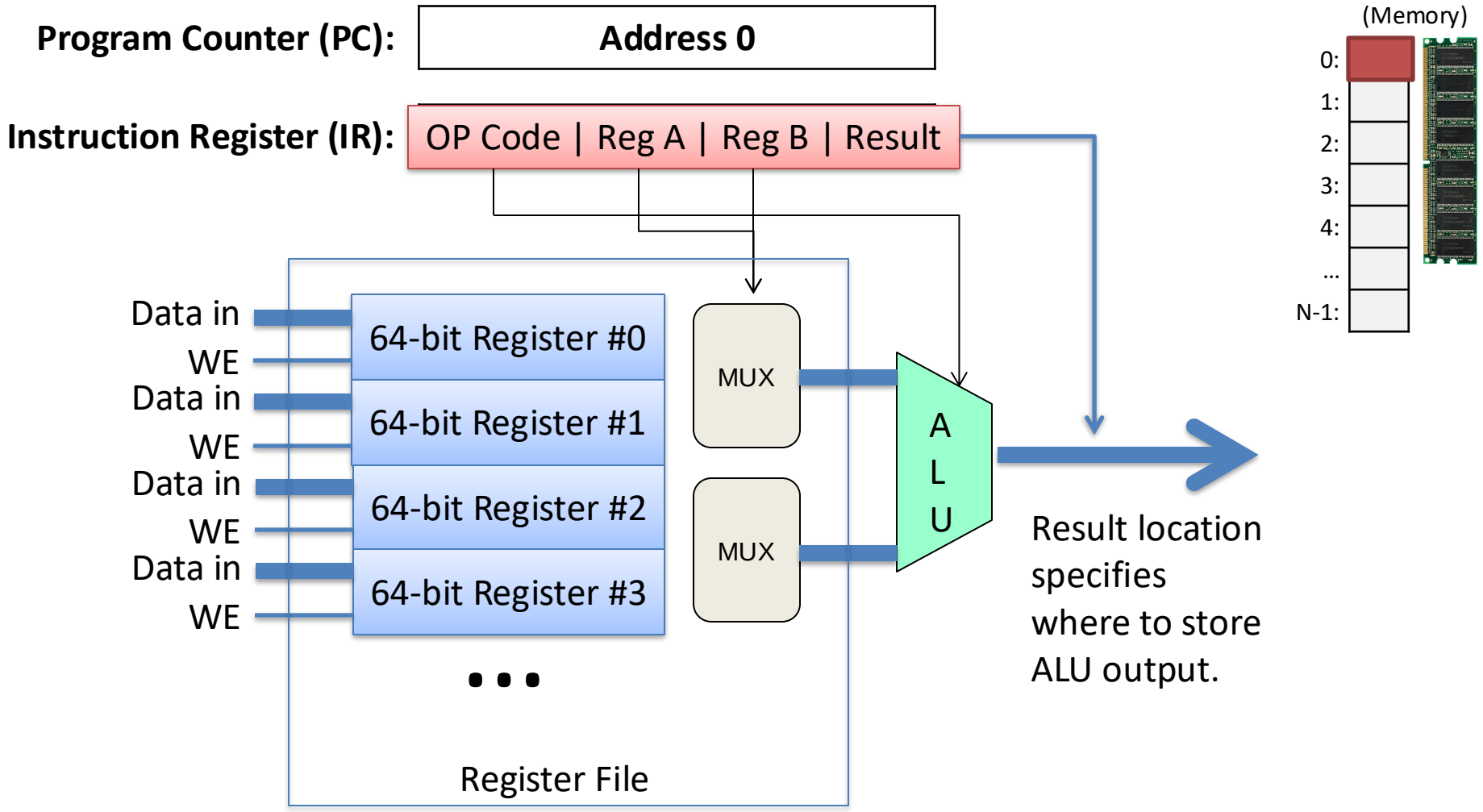
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?



Storing results.

We've just computed something. Where do we put it?



Why do we need a program counter? Can't we just start at 0 and count up one at a time from there?

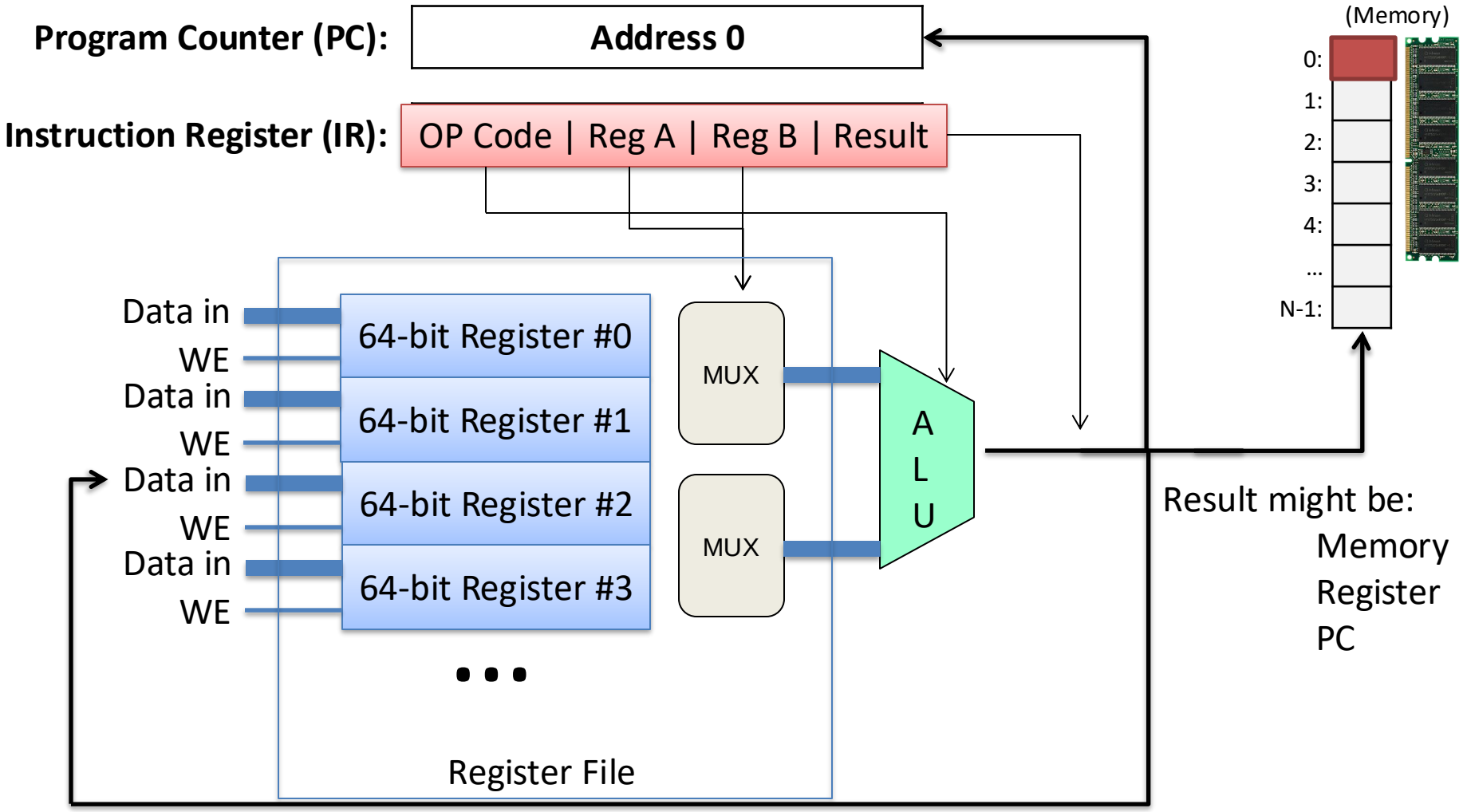
A. We don't, it's there for convenience.
B. Some instructions might skip the PC forward by more than one.
C. Some instructions might adjust the PC backwards.
D. We need the PC for some other reason(s).

Why do we need a program counter? Can't we just start at 0 and count up one at a time from there?

A. We don't, it's there for convenience.
B. Some instructions might skip the PC forward by more than one.
C. Some instructions might adjust the PC backwards.
D. We need the PC for some other reason(s).

Storing results.

Interpret the instruction bits: What operation? Which arguments?



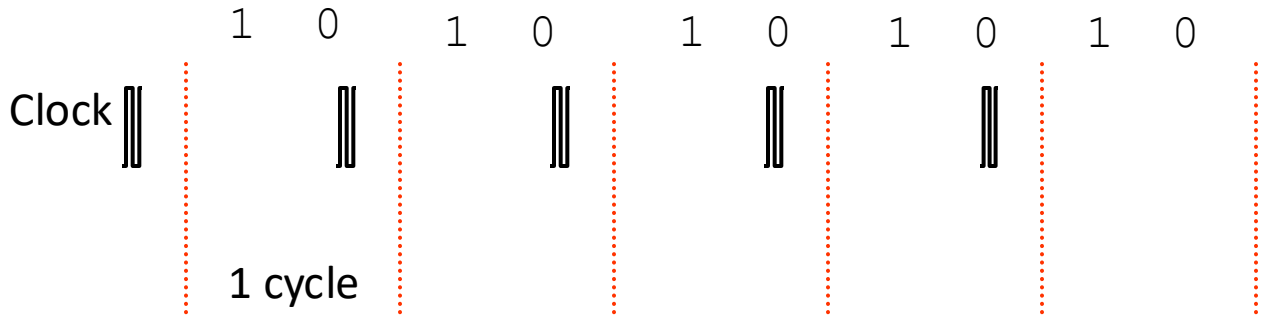
Clocking

- Need to periodically transition from one instruction to the next.

- It takes time to fetch from memory, for signal to propagate through wires, etc.
  - Too fast: don’t fully compute result
  - Too slow: waste time

Clock Driven System

- Everything in is driven by a discrete clock
  - clock: an oscillator circuit, generates hi low pulse
  - clock cycle: one hi-low pair



  - Clock determines how fast system runs
    - Processor can only do one thing per clock cycle
      - Usually just one part of executing an instruction
    - 1GHz processor:

      1 billion cycles/second → 1 cycle every nanosecond

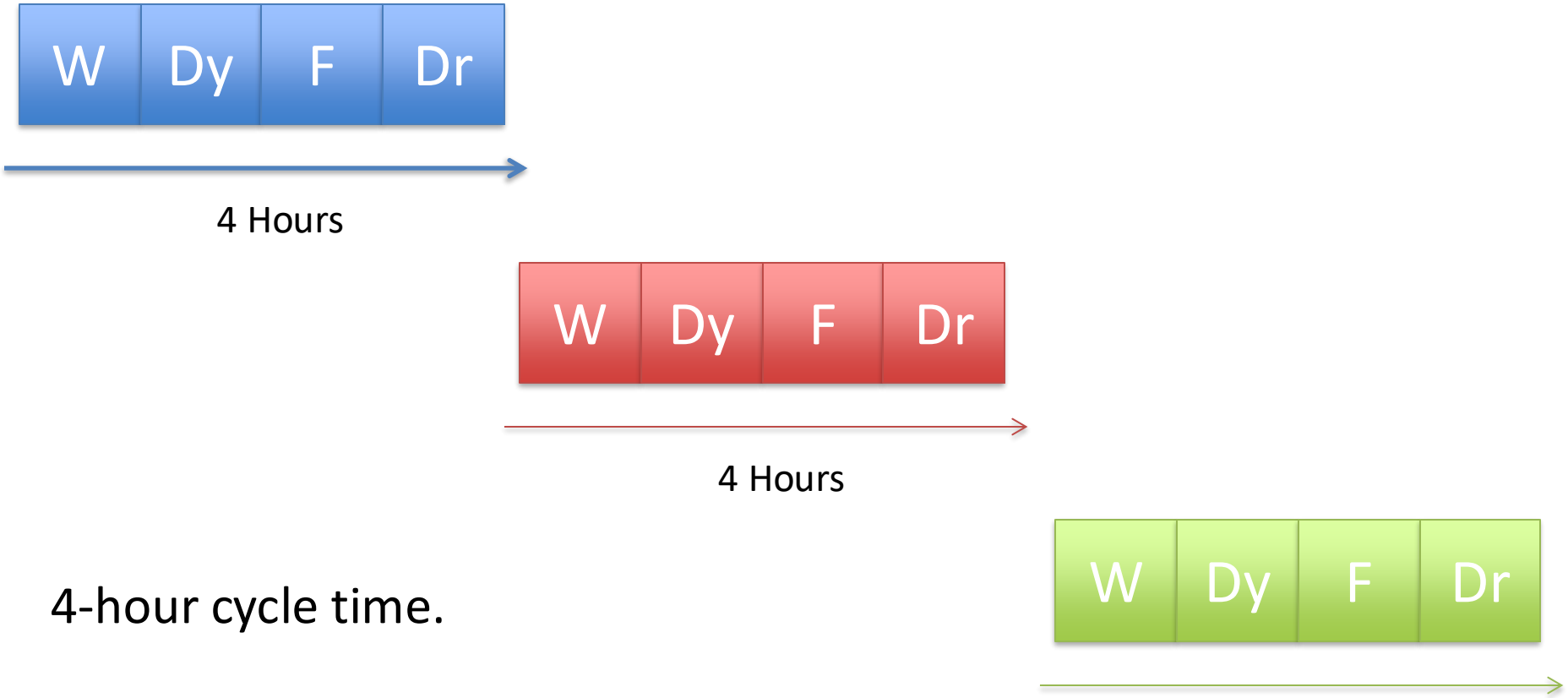
Cycle Time: Laundry Analogy

- Discrete stages: fetch, decode, execute, store

- Analogy (laundry): washer, dryer, folding, dresser

You have big problems if you have millions of loads of laundry to do….

Laundry

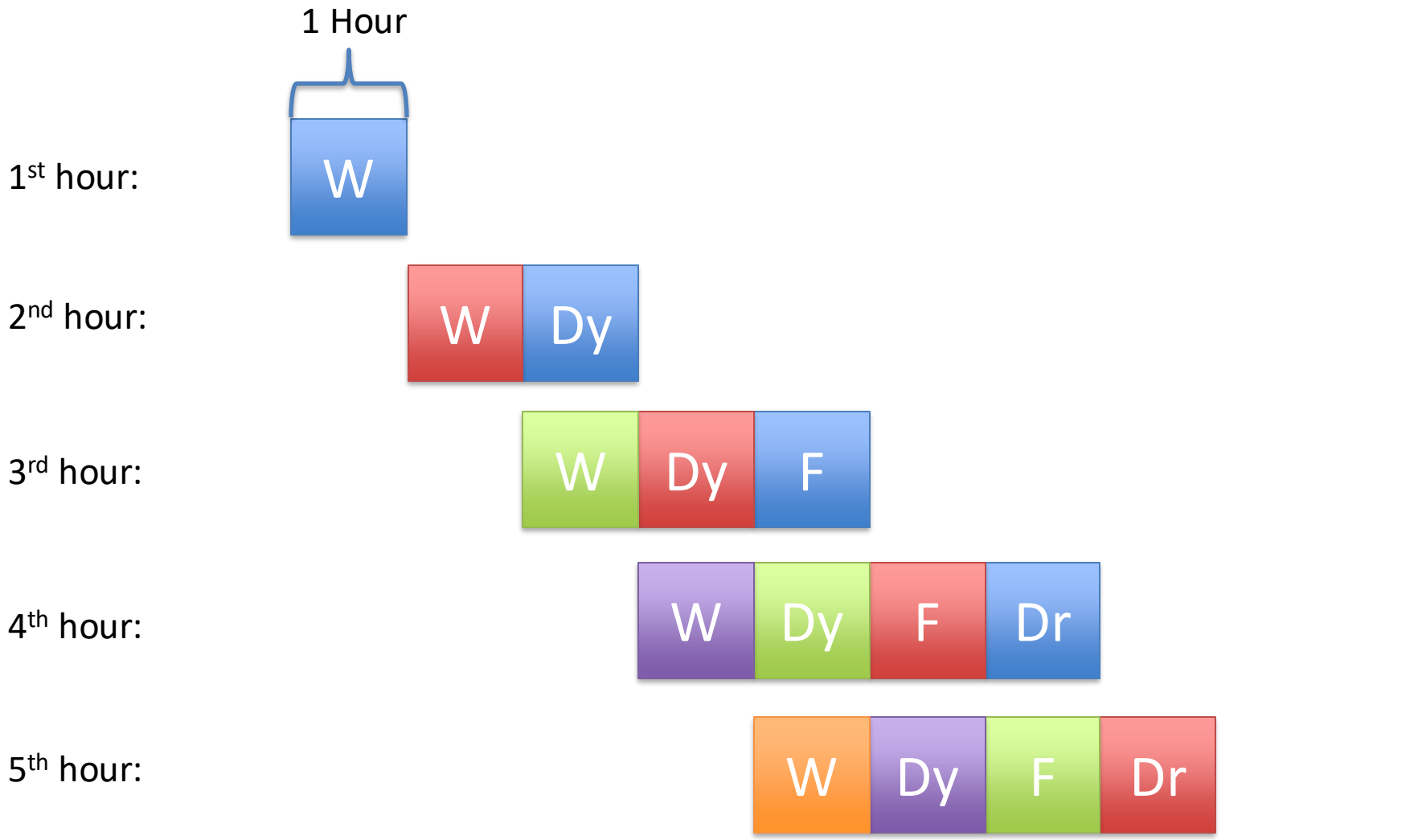


4-hour cycle time.

Finishes a laundry load every cycle.

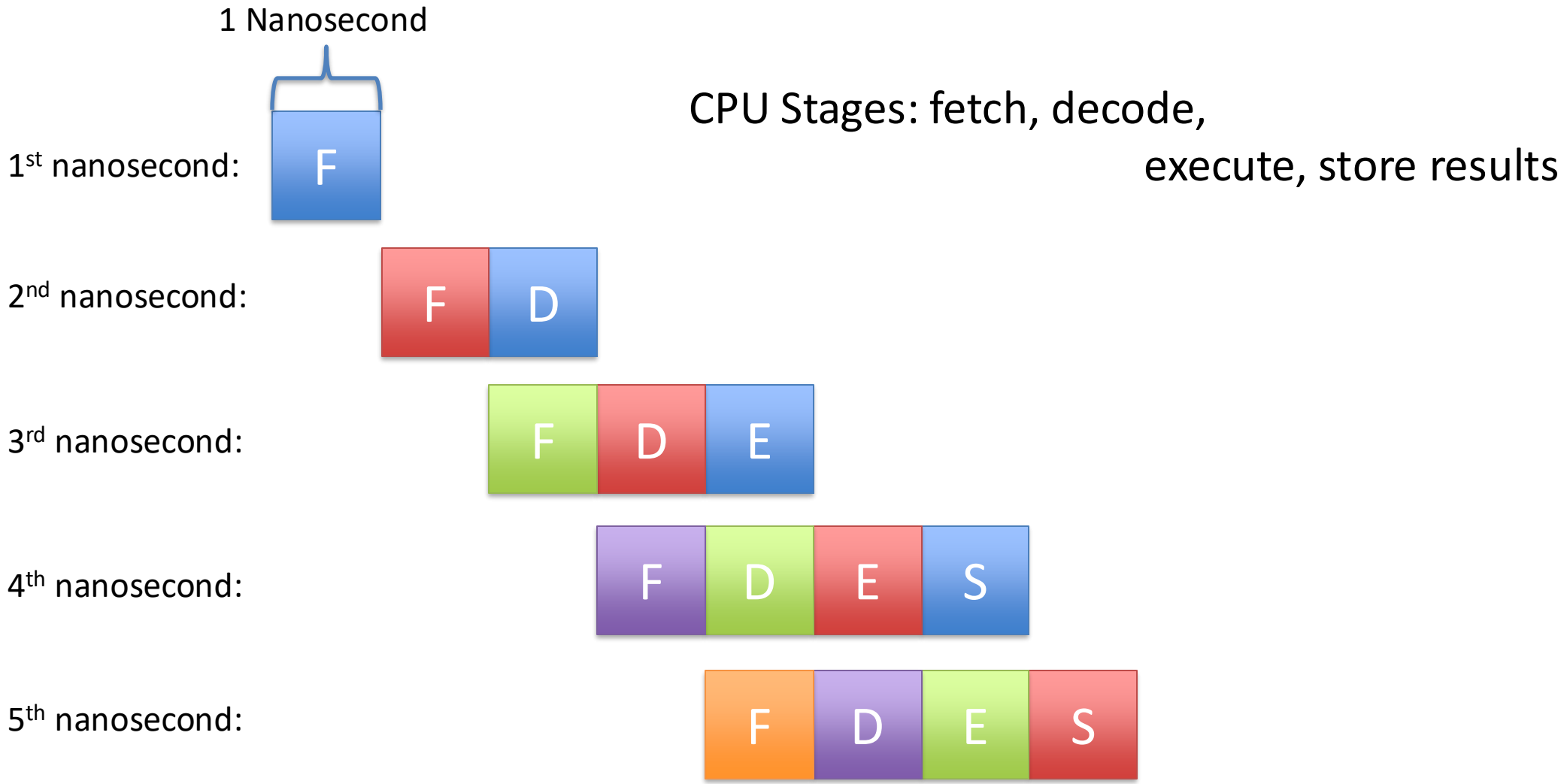
(6 laundry loads per day)

Pipelining (Laundry)



Steady state: One load finishes every hour!
(Not every four hours like before.)

Pipelining (CPU)



CPU Stages: fetch, decode, execute, store results

Steady state: One instruction finishes every nanosecond!
(Clock rate can be faster.)

Pipelining

(For more details about this and the other things we talked about here, take architecture.)

Overview

- How to reference the location of a variable in memory
- Where variables are placed in memory
- How to make this information useful
  - Allocating memory
  - Calling functions with pointer arguments

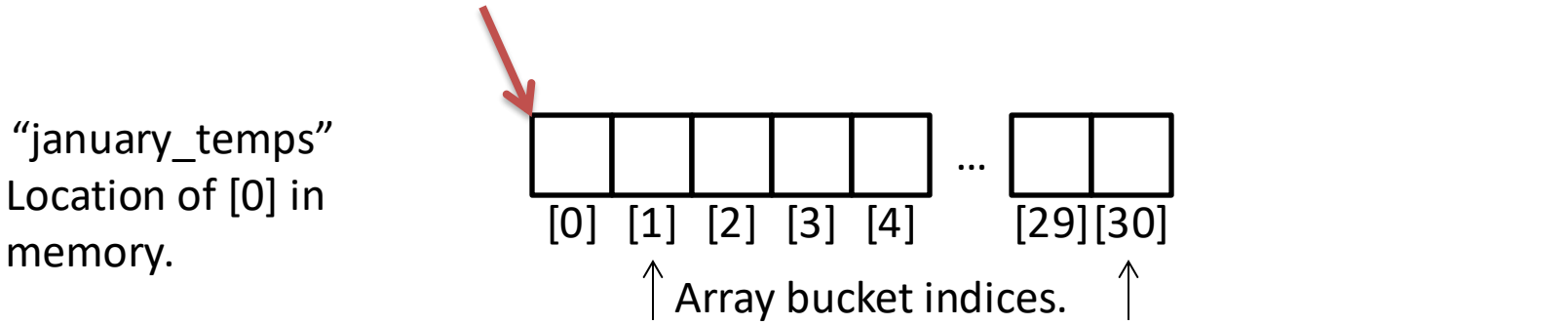
Pointers

- Pointer: A variable that stores a reference to (the address of) a memory location.
- Pointer: sequence of bits that should be interpreted as an index into memory.
- Where have we seen this before?

- A pointer is like a mailing address, it tells you where a variable is **located in memory**.

Recall: Arrays

```
int january_temps[31];  // Daily high temps
```



- Array variable name means, to the compiler, the beginning of the memory chunk. (address)

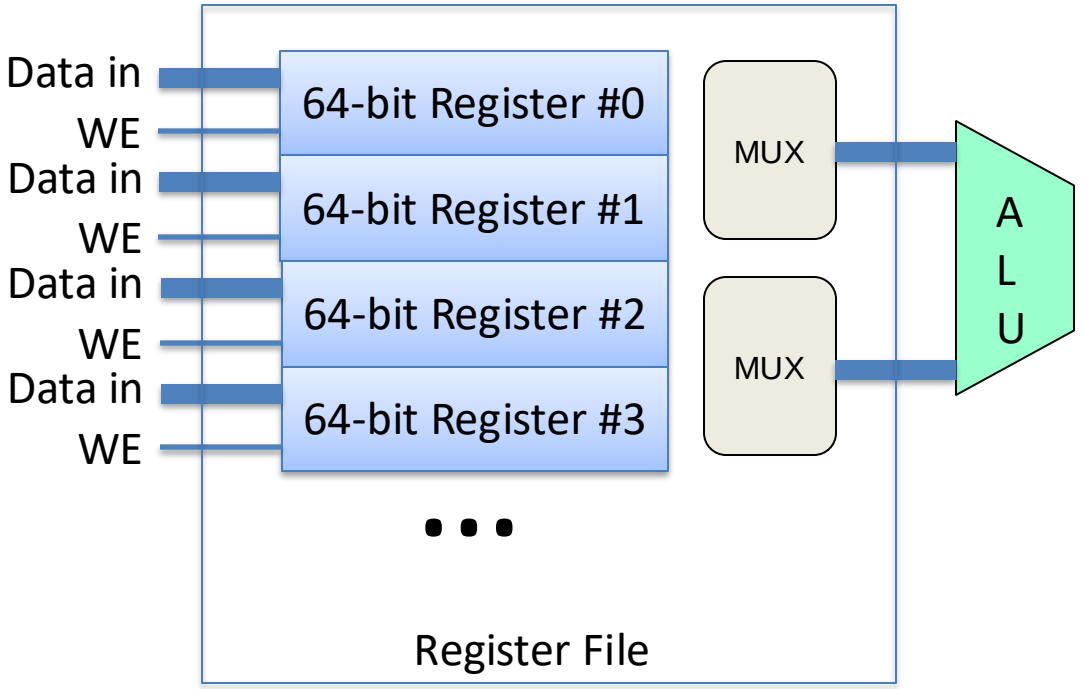
Recall: Program Counte

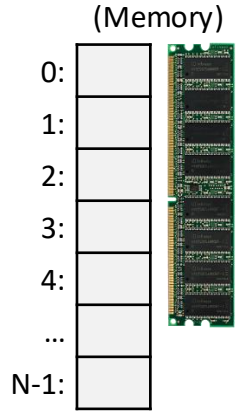
X86_64 refers to the PC as %rip.

Instruction **P**ointer

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

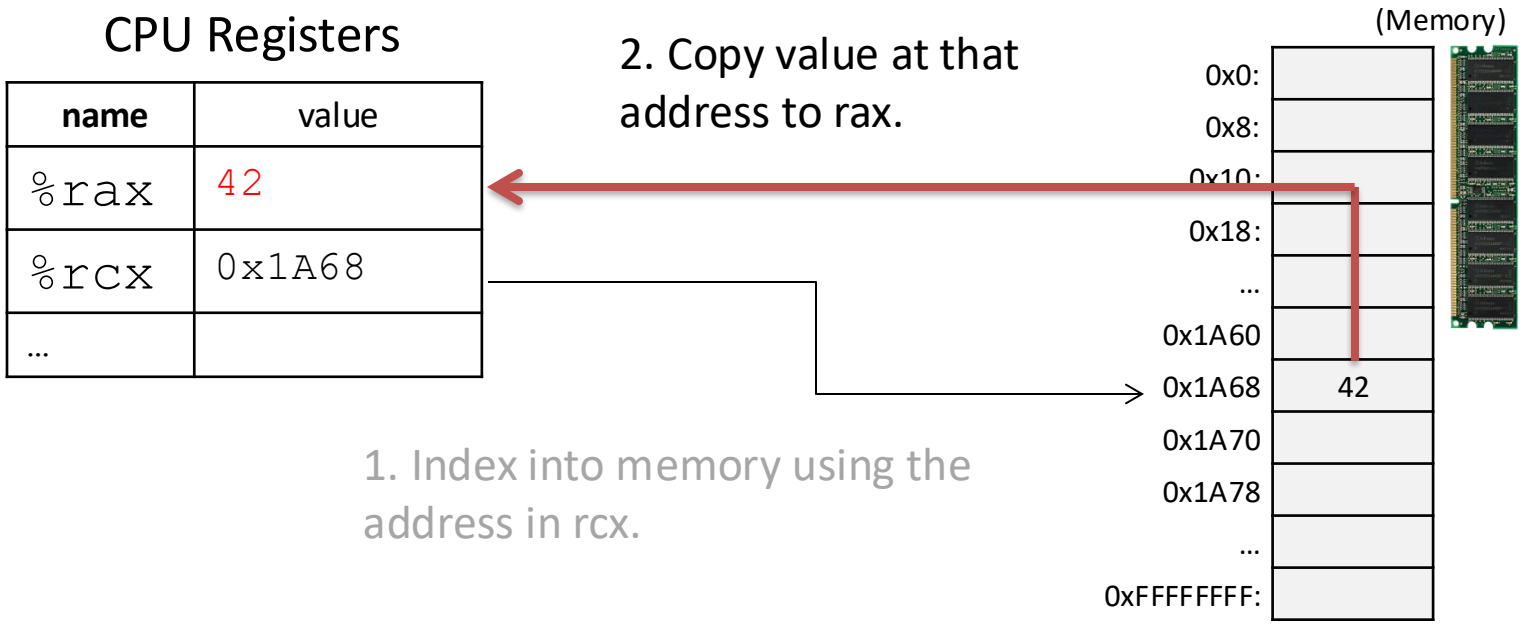




Recall: Addressing Mode: Memory

`movl (%rcx), %rax`

– Use the address in register %rcx to access memory, store result in register %rax



Pointers in C

- Like any other variable, must be declared:
  - Using the format: `type *name;`
- Example:
  - `int *myptr;`
  - This is a promise to the compiler:
    - This variable holds a memory address. If you follow what it points to in memory (dereference it), you'll find an integer.
- A note on syntax:
  - `int* myptr;` `int * myptr;` `int *myptr;`
  - These all do the same thing. (note the * position)

Dereferencing a Pointer

- To follow the pointer, we *dereference* it.
- Dereferencing re-uses the * symbol.
- If `iptr` is declared as an integer pointer,
  `*iptr` will follow the address it stores to find an integer in memory.

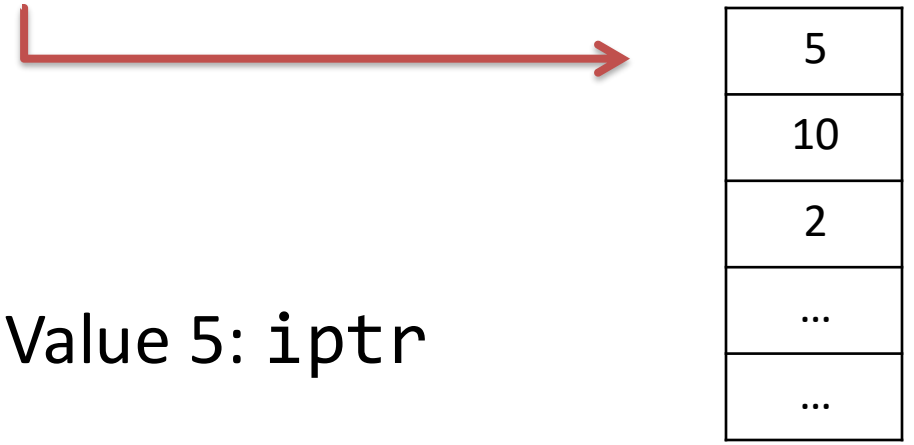
Putting a * in front of a variable…

- When you *declare* the variable:
  - Declares the variable to be a pointer
  - It stores a memory address

- When you *use* the variable (dereference):
  - Like putting [] around a register name
  - Follows the pointer out to memory
  - Acts like the specified type (e.g., int, float, etc.)

Suppose we set up a pointer like the one below. Which expression gives us 5, and which gives us a memory address?

```
int *iptr = (the location of that memory);
```

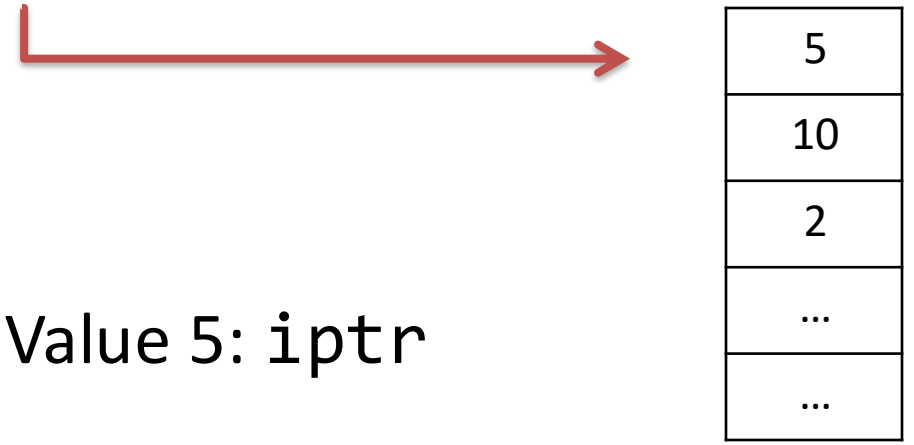


A. Memory address: `*iptr`, Value 5: `iptr`

B. Memory address: `iptr`, Value 5: `*iptr`

Suppose we set up a pointer like the one below. Which expression gives us 5, and which gives us a memory address?

```
int *iptr = (the location of that memory);
```



A. Memory address: `*iptr`, Value 5: `iptr`

B. Memory address: `iptr`, Value 5: `*iptr`

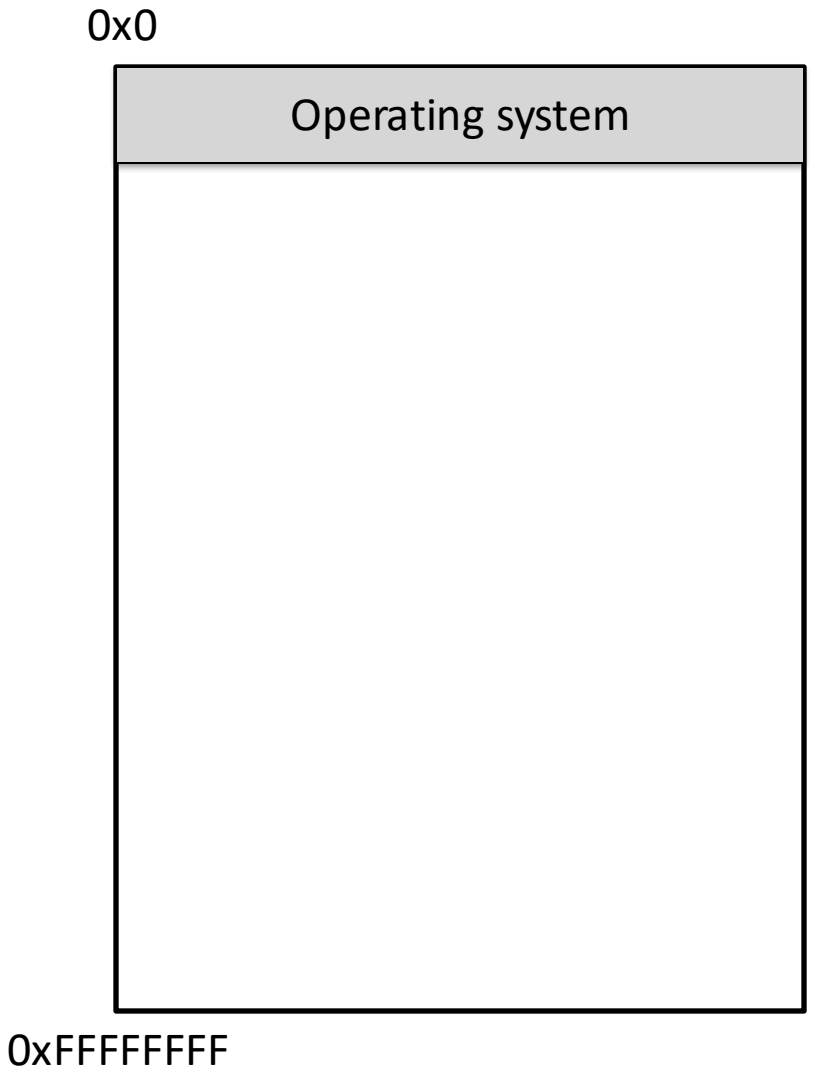
So, we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable (&)
  2. Copy some other pointer
  3. Allocate some memory and point to it

- First, let's look at how memory is organized.
  (From the perspective of one executing program.)

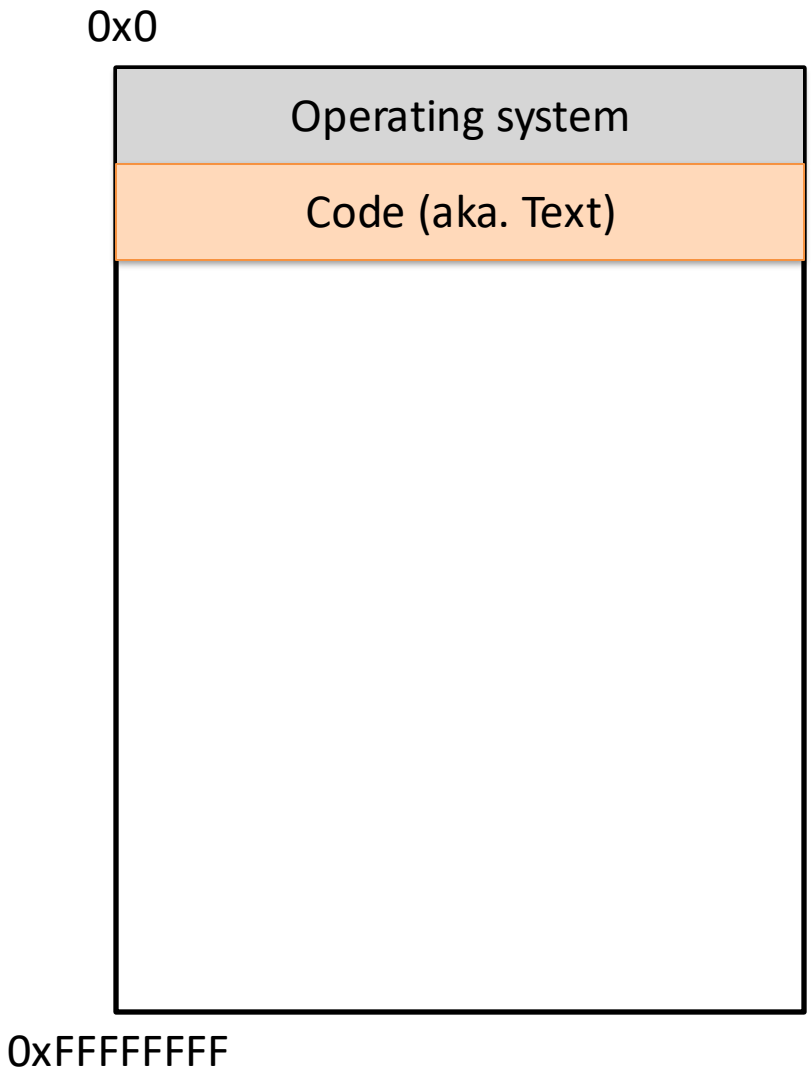
Memory

- Behaves like a big array of bytes, each with an address (bucket #).
- By convention, we divide it into regions.
- The region at the lowest addresses is usually reserved for the OS.



Memory - Text

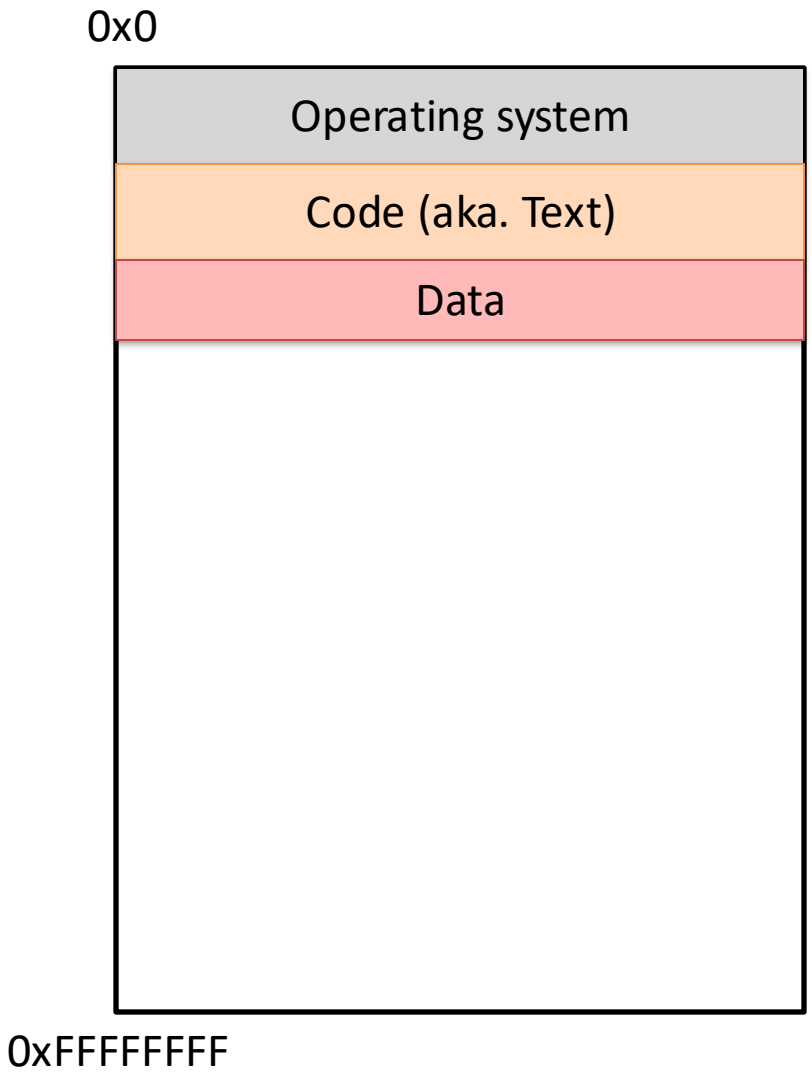
- After the OS, we store the program's code.
- Instructions generated by the compiler.



Memory – (Static) Data

- Next, there's a fixed-size region for static data.

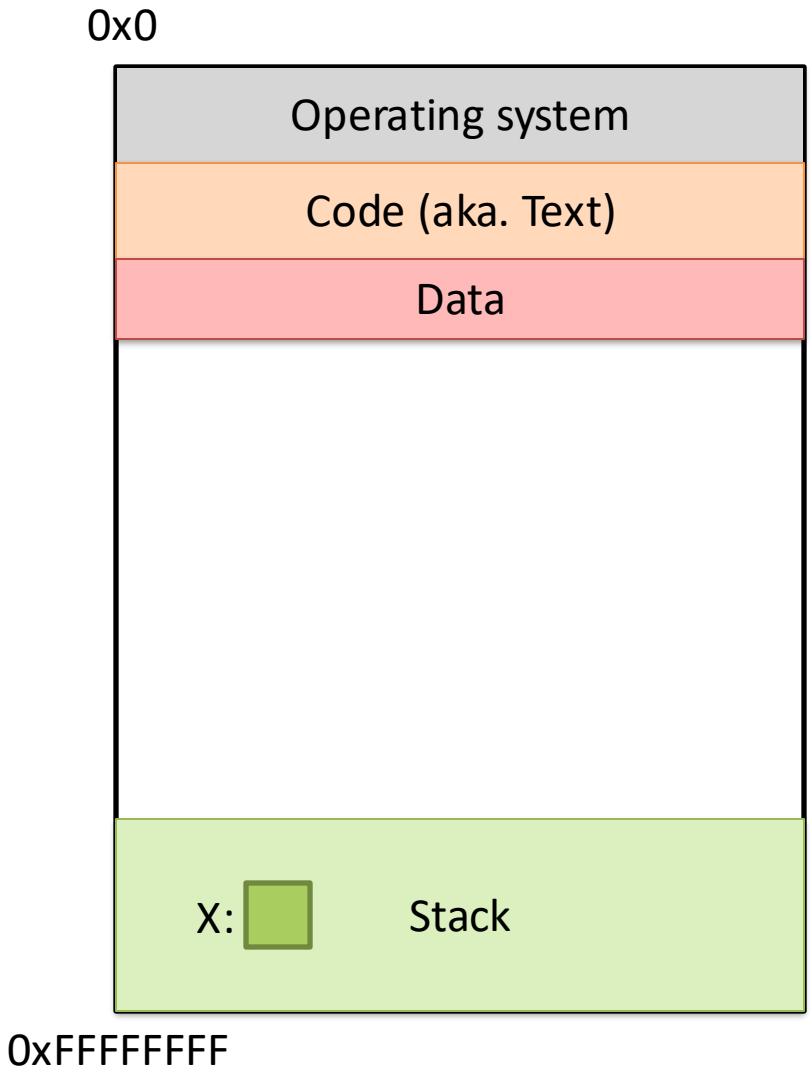
- This stores static variables that are known at compile time.
  - Global variables
  - Static (hard-coded) strings



Memory - Stack

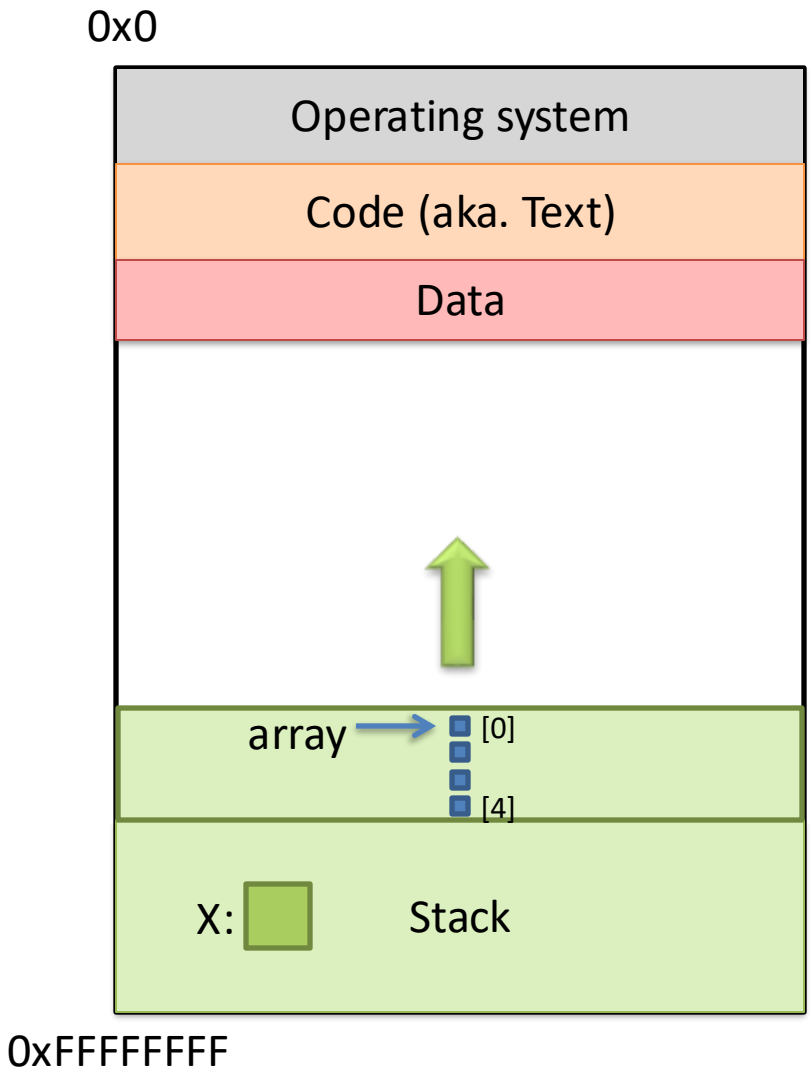
- At high addresses, we keep the stack.

- This stores local (automatic) variables.
  - The kind we've been using in C so far.
  - e.g., `int x;`



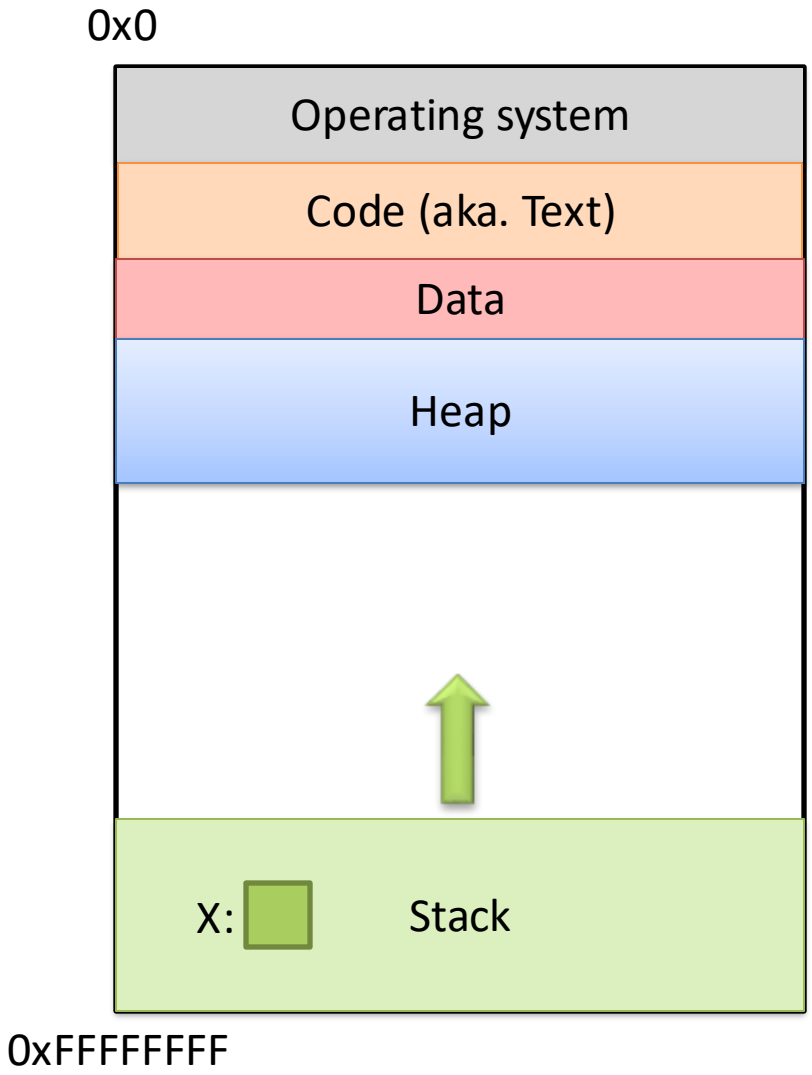
Memory - Stack

- The stack grows upwards towards lower addresses (negative direction).
- Example: Allocating array
  - int array[4];
- (Note: this differs from Python.)



Memory - Heap

- The heap stores dynamically allocated variables.
- When programs explicitly ask the OS for memory, it comes from the heap.
  - malloc() function



If we can declare variables on the stack, why do we need to dynamically allocate things on the heap?

A. There is more space available on the heap.

B. Heap memory is better. (Why?)

C. We may not know a variable's size in advance.

D. The stack grows and shrinks automatically.

E. Some other reason.

If we can declare variables on the stack, why do we need to dynamically allocate things on the heap?

A. There is more space available on the heap.

B. Heap memory is better. (Why?)

C. We may not know a variable's size in advance. (Primary reason)

D. The stack grows and shrinks automatically. (Return from function: can't return large chunk of memory safely)

E. Some other reason.

"Static" vs. "Dynamic"

Static

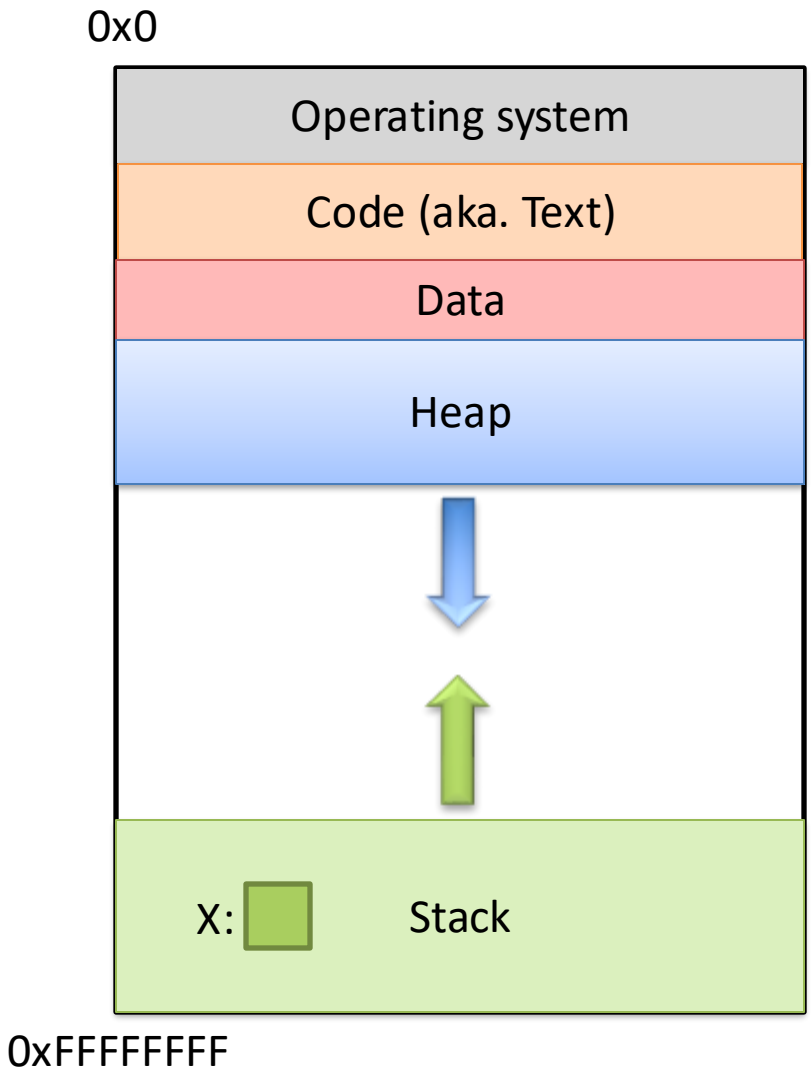
- The compiler can know in advance.
- The size of a C variable (based on its type).
- Hard-coded constants.

Dynamic

- The compiler cannot know -- must be determined at run time.
- User input (or things that depend on it).
- E.g., create an array where the size is typed in by user (or file).

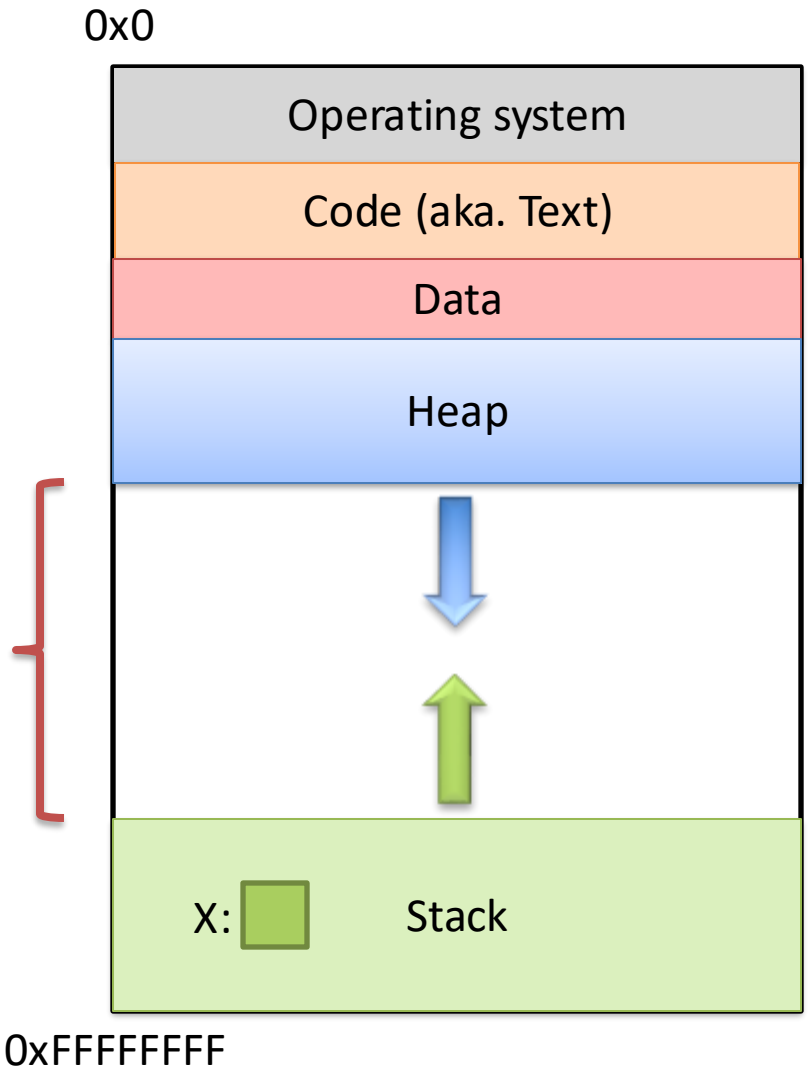
Memory - Heap

- The heap grows downwards, towards higher addresses.
- I know you want to ask a question…



Memory - Heap

- "What happens if the heap and stack collide?"
- This picture is not to scale – the gap is huge.
- The OS works really hard to prevent this.
  - Would likely kill your program before it could happen.



Which region would we expect the PC register (program counter) to point to?

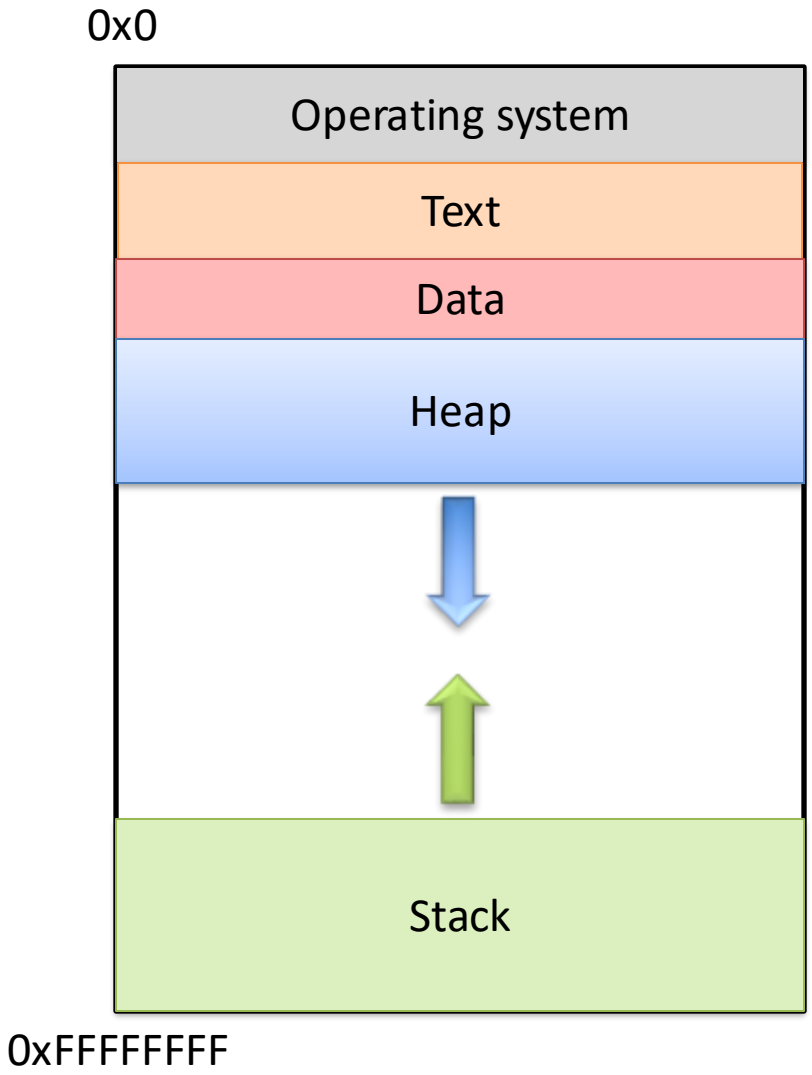
A. OS

B. Text

C. Data

D. Heap

E. Stack



Which region would we expect the PC register (program counter) to point to?

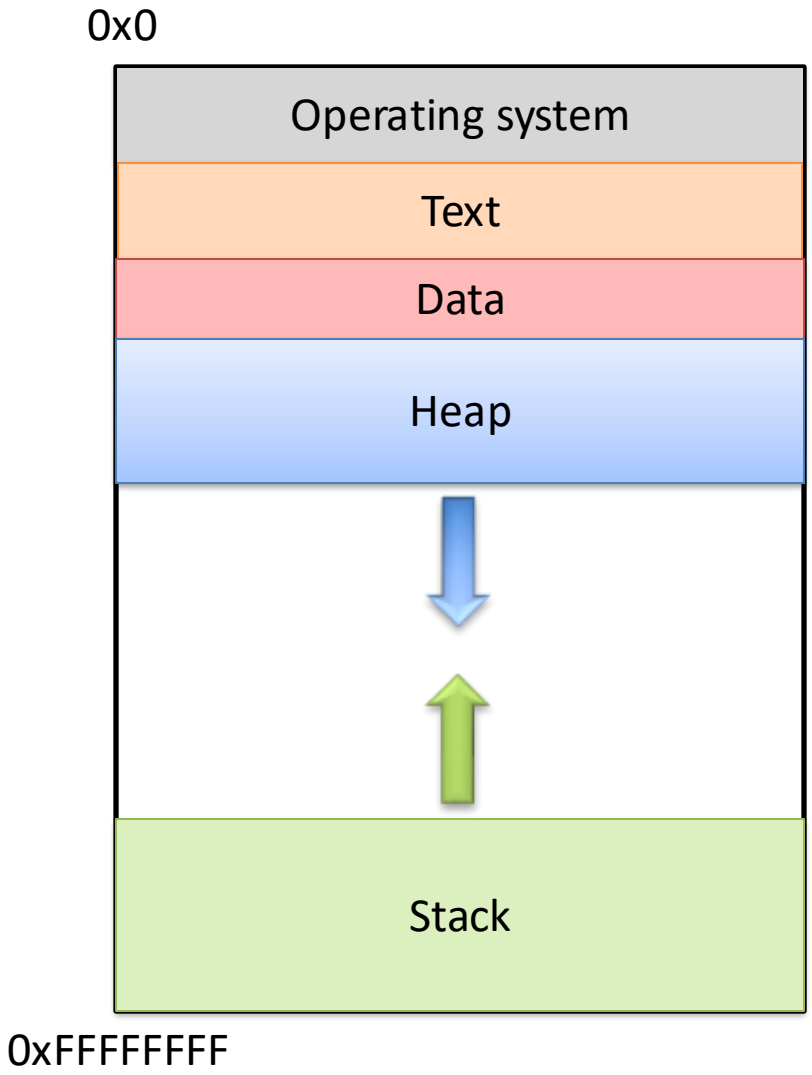
A. OS

B. Text

C. Data

D. Heap

E. Stack



What should happen if we try to access an address that's NOT in one of these regions?

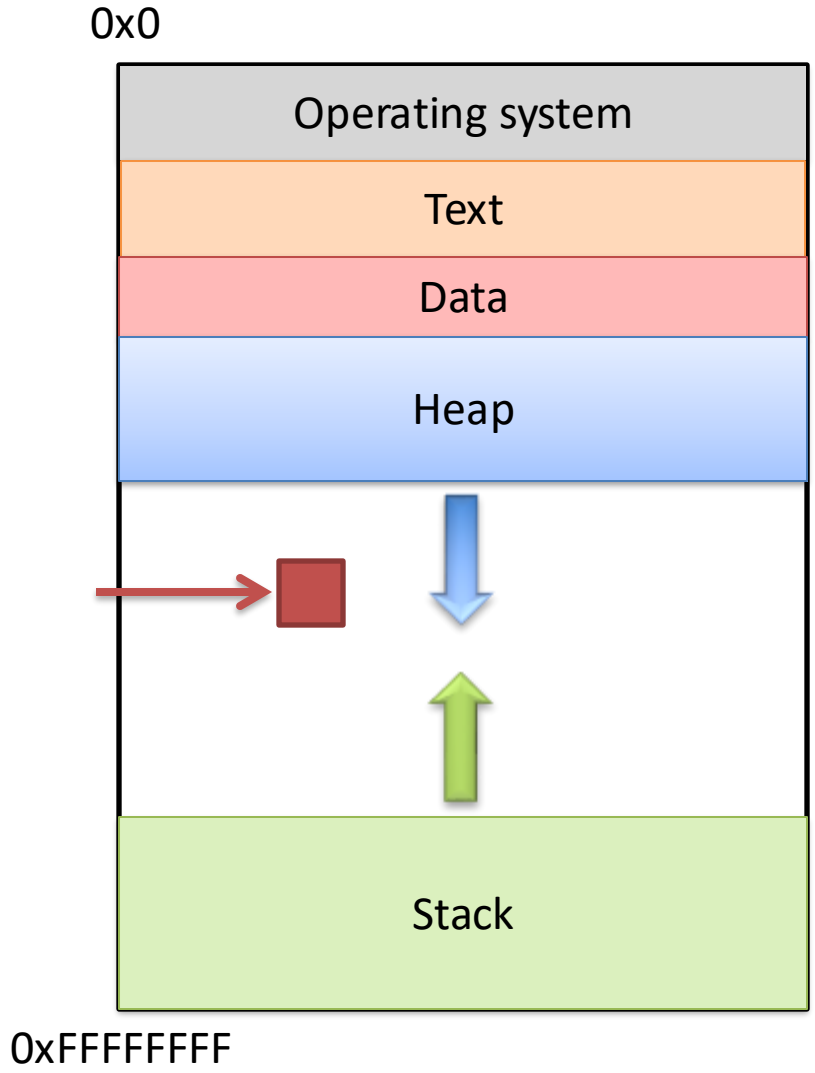
A. The address is allocated to your program.

B. The OS warns your program.

C. The OS kills your program.

D. The access fails, try the next instruction.

E. Something else



What should happen if we try to access an address that's NOT in one of these regions?

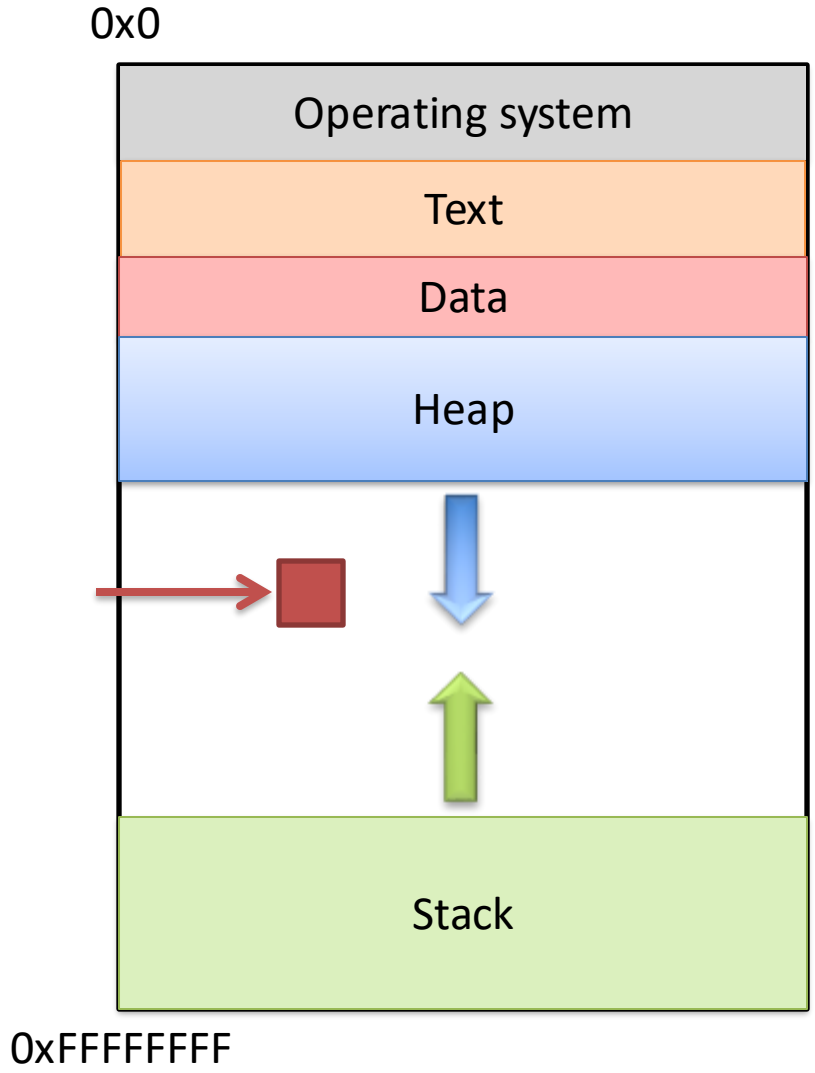
A. The address is allocated to your program.

B. The OS warns your program.

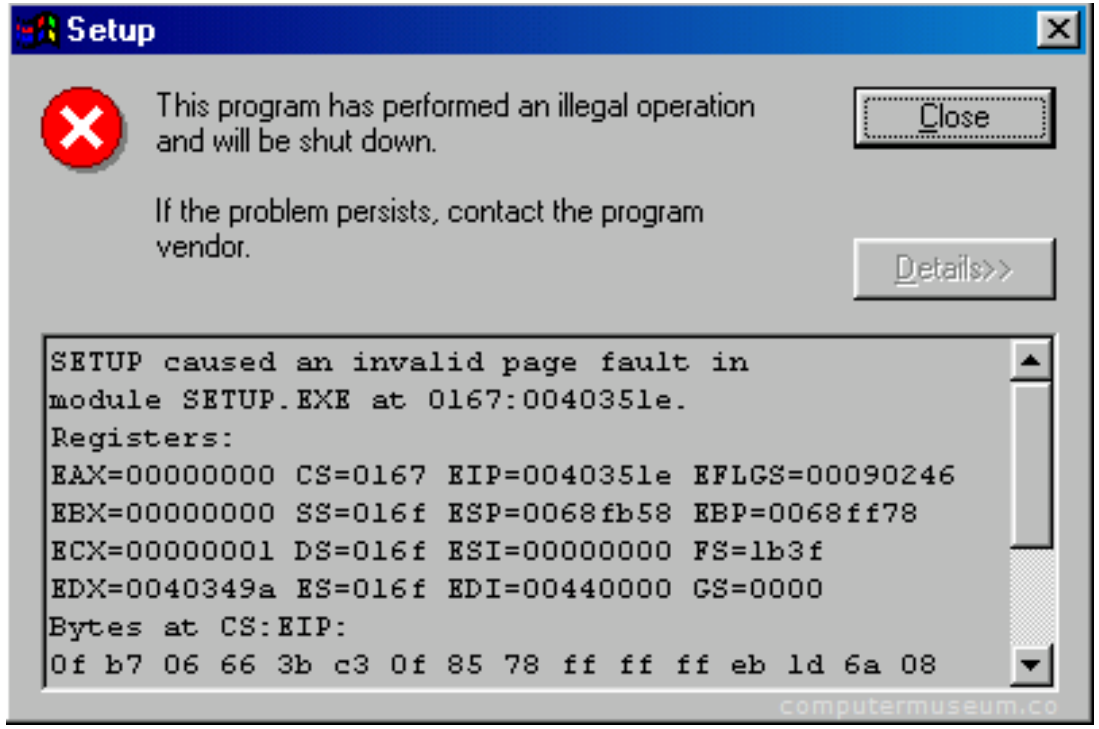
C. The OS kills your program.

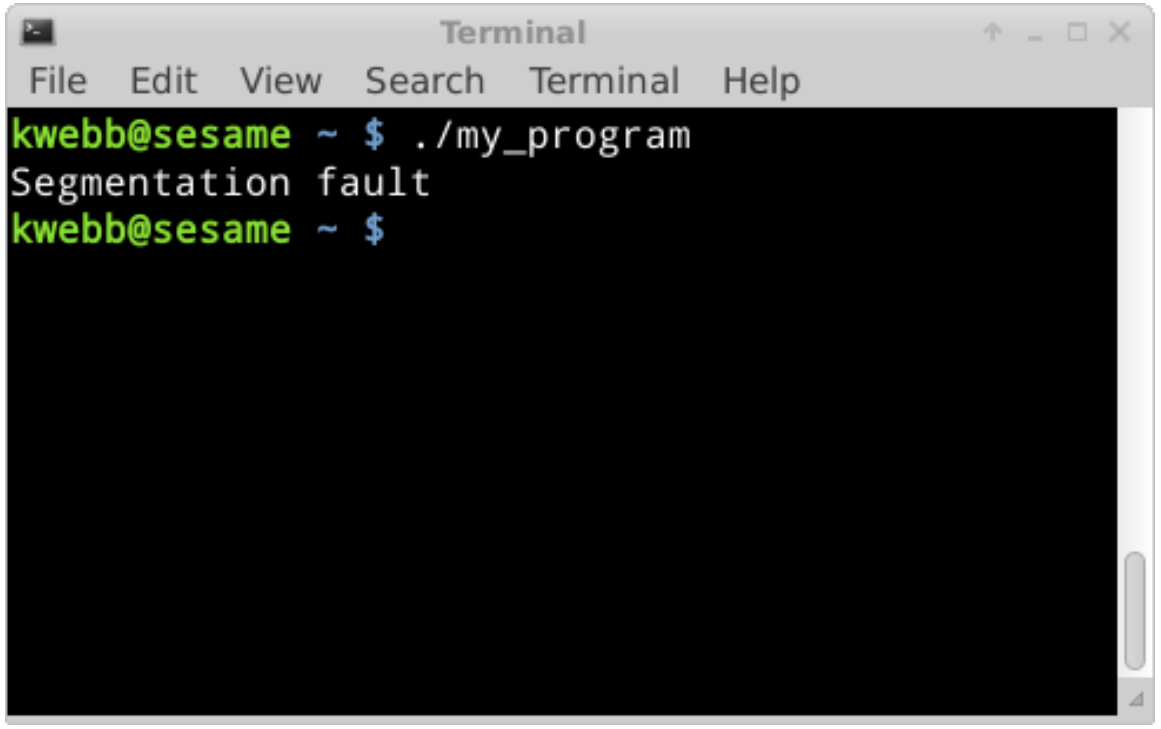
D. The access fails, try the next instruction.

E. Something else



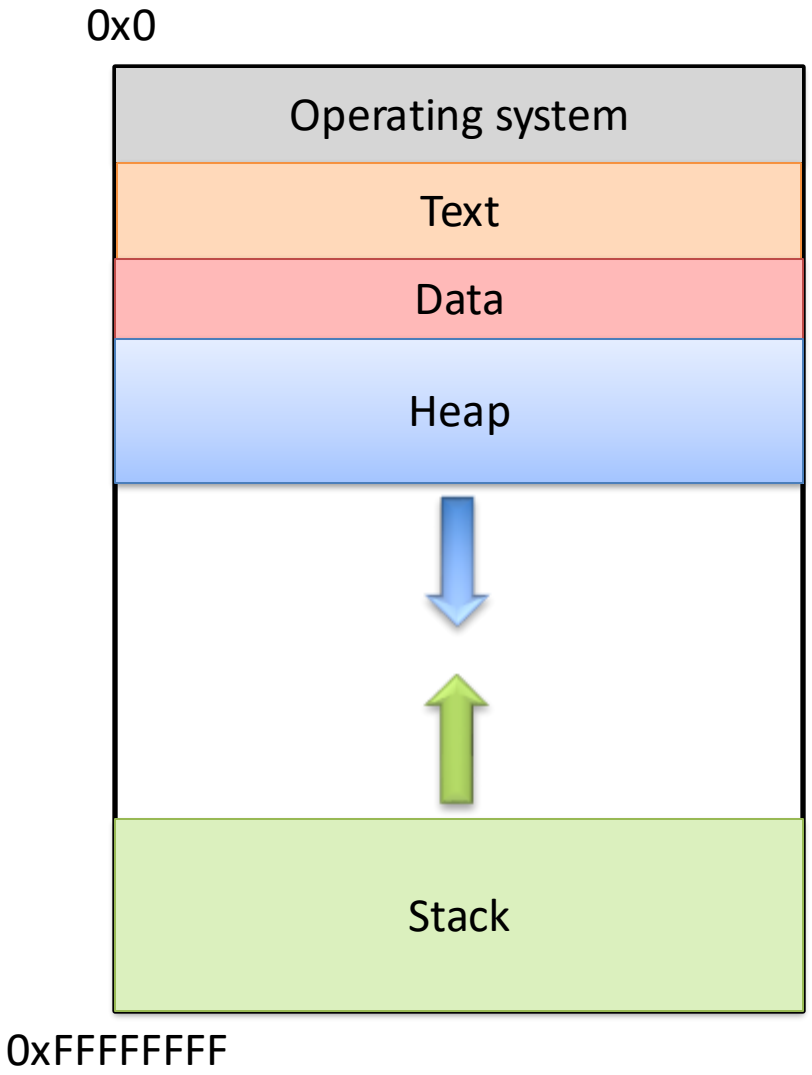
Segmentation Violation





Segmentation Violation

- Each region also known as a memory segment.
- Accessing memory outside a segment is not allowed.
- Can also happen if you try to access a segment in an invalid way.
  - OS not accessible to users
  - Text is usually read-only



So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

The Address Of (&)

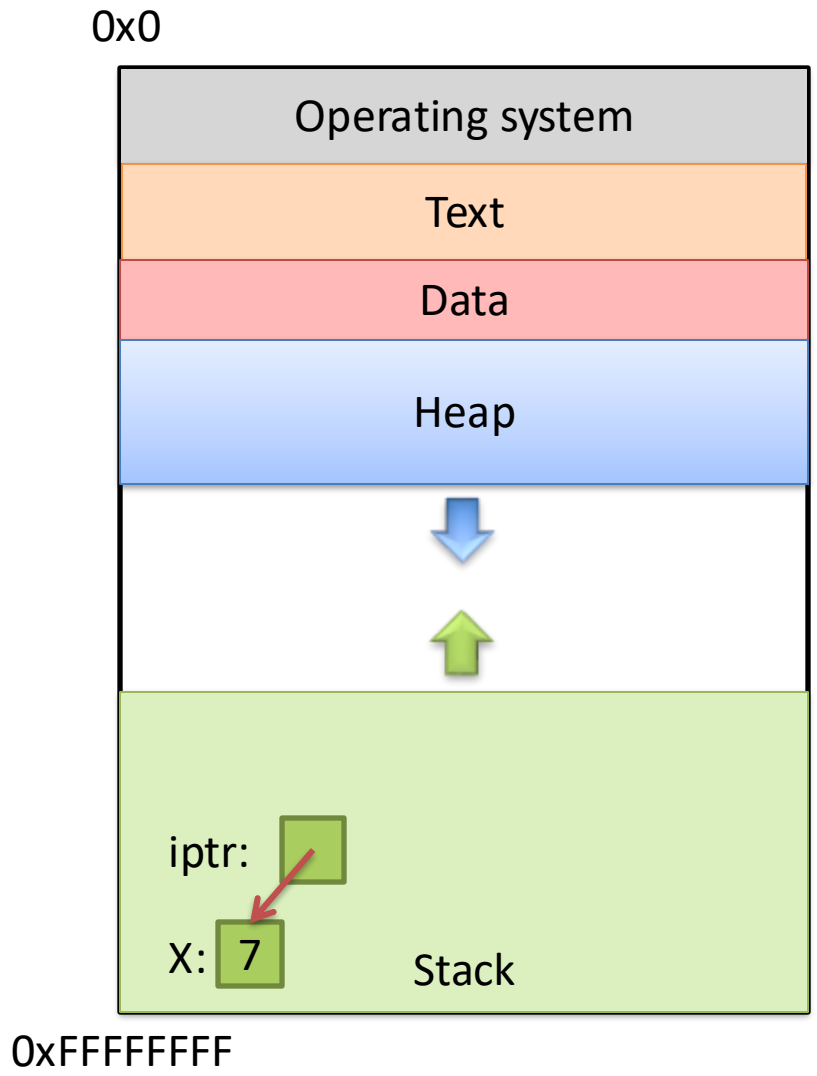
- You can create a pointer to anything by taking its address with the *address of* operator (&).

The Address Of (&)

```
int main(void) {
   int x = 7;
   int *iptr = &x;




   return 0;
}
```

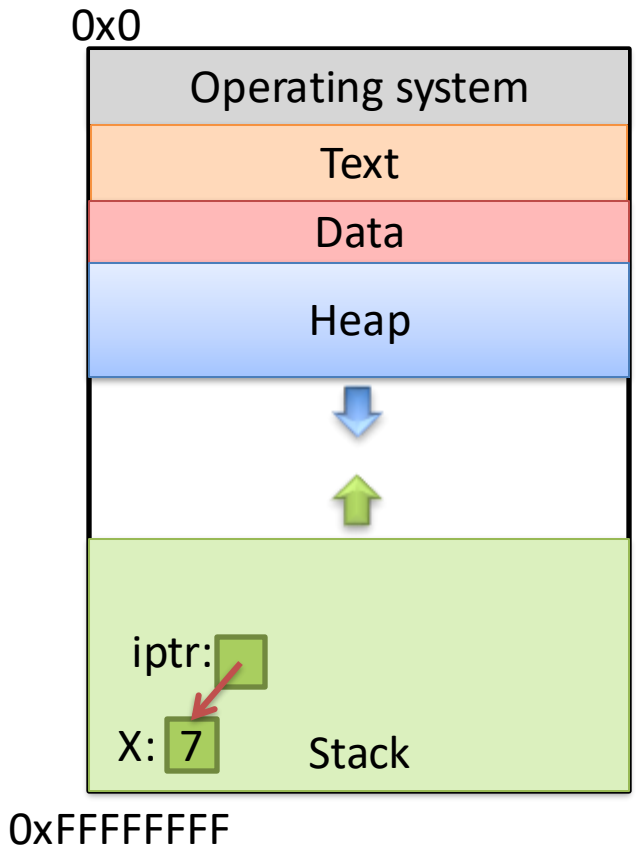


What would this print?

```
int main(void) {
  int x = 7;
  int *iptr  = &x;
  int *iptr2 = &x;

  printf("%d %d ", x, *iptr);
  *iptr2 = 5;
  printf("%d %d ", x, *iptr);

  return 0;
}
```



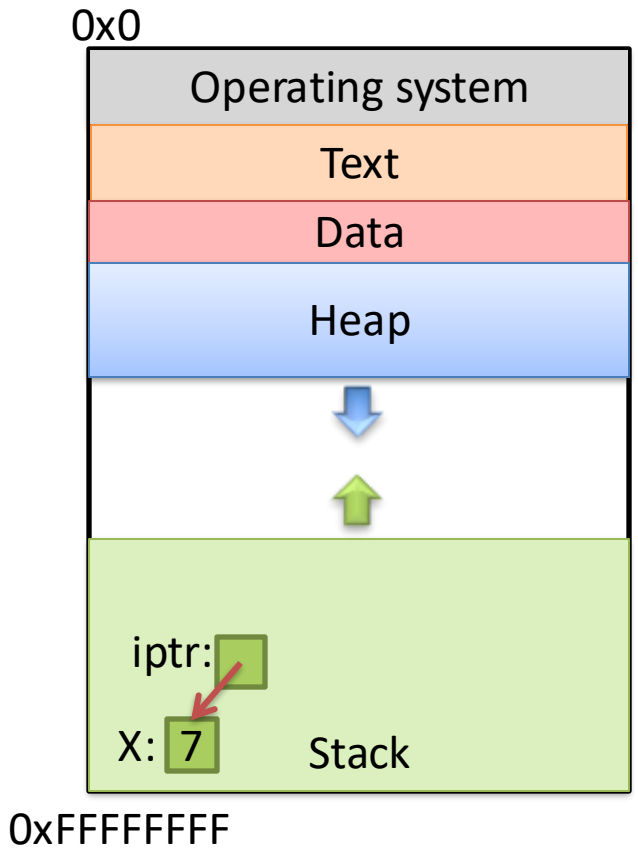
A. 7 7 7 7 B. 7 7 7 5 C. 7 7 5 5 D. Something else

What would this print?

```
int main(void) {
   int x = 7;
   int *iptr  = &x;
   int *iptr2 = &x;

   printf("%d %d ", x, *iptr);
   *iptr2 = 5;
   printf("%d %d ", x, *iptr);

   return 0;
}
```



A. 7 7 7 7 B. 7 7 7 5 C. 7 7 5 5 D. Something else