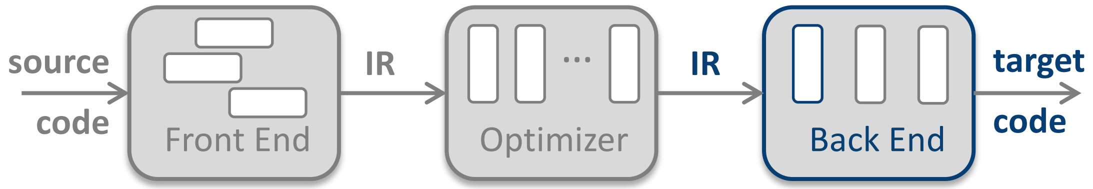
COMP 412
FALL 2018

The ILOC Virtual Machine

(Background Material for all three labs)

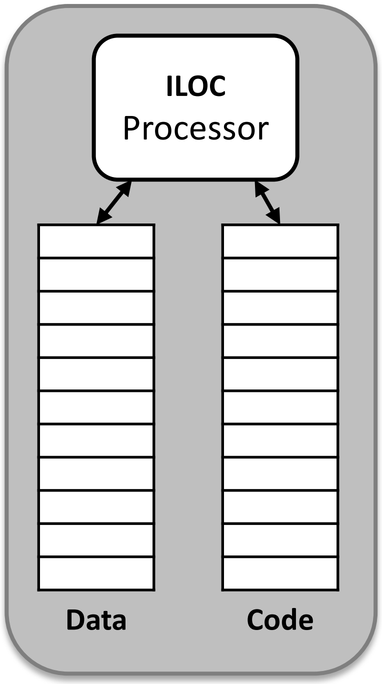
Comp 412



The **ILOC** Virtual Machine

What is the execution model for an ILOC program?

- **ILOC** is the assembly language of a simple, idealized **RISC** processor
  - **ILOC Virtual Machine**
    - → Separate code memory and data memory
      - → Sometimes called a *Harvard architecture*
    - → Sizes of data memory & register set are configurable
    - → Code memory is "large enough" to hold your program
    - → Simple, in-order execution model
  - **ILOC Instruction Set**
    - → Arithmetic operations work on values held in registers
    - → Load & store move values between registers and memory



- To debug the output of your labs, you will use an **ILOC** simulator, a program that mimics the operation of the **ILOC** virtual machine—that is, it is an interpreter for **ILOC**

The term "Harvard architecture" derives from the Harvard Mark 1, an early computer built out of electrical relays.

RISC ≡ <u>R</u>educed <u>I</u>nstruction <u>S</u>et <u>P</u>rocessor

The **ILOC** Subset

See also the Lab 1 handout and Appendix A in EaC2e

Pay attention to the meanings of the ILOC operations

Syntax			Meaning	Latency
load	r_1	=> r_2	$r_2 \leftarrow$ MEM(r_1)	3
store	r_1	=> r_2	MEM(r_2) $\leftarrow$ r1	3
loadI	c	=> r_2	$r_2 \leftarrow$ c	1
add	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 + r_2$	1
sub	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 - r_2$	1
mult	r_1, r_2	=> r_3	$r_3 \leftarrow$ r1 * r_2	1
lshift	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 << r_2$	1
rshift	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 >> r_2$	1
output	c		prints MEM(x) to stdout	1
nop			idles for one cycle	1

ILOC is an abstract assembly language. Each operation, except **nop**, use (or read) one or more values. Each operation, except **output** and **nop**, defines a value.

loadI reads its value from the instruction stream.

load reads both a register and a memory location.

store reads two registers and writes a memory location.

add, **sub**, **mult**, **lshift**, and **rshift** read two registers and write one register.

ILOC Execution

A Simple ILOC Program

```
% cat ex1.iloc
// add two numbers
  loadI 314   => r0
  loadI   0   => r1
  load  r1    => r1
  add   r0,r1 => r2
  loadI 0     => r0
  store r2    => r0
  output 0

% sim ex1.iloc -i 0 14 17
328

Executed 7 instructions and 7 operations in 11 cycles.
```

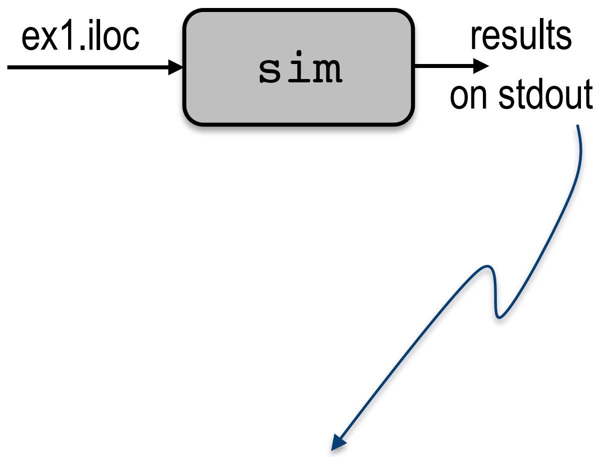
We provide lab-specific simulators for **ILOC** code, that you can use to simulate their execution.

The **–i** option initializes memory, starting at location 0, with the values 14 and 18.

ILOC Execution

A Simple ILOC Program

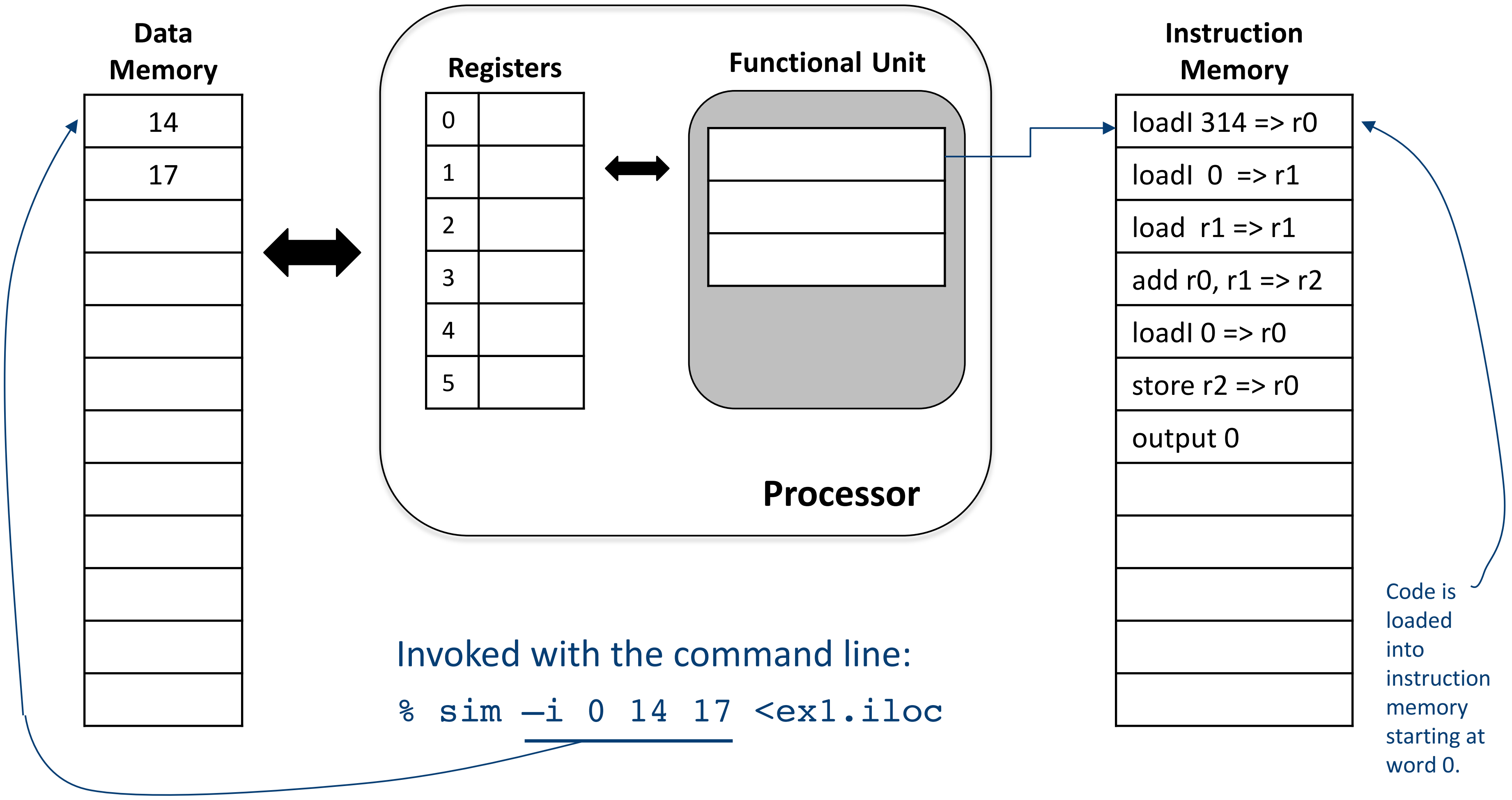
```
% cat ex1.iloc
// add two numbers
   loadI 314    => r0
   loadI   0    => r1
   load  r1     => r1
   add   r0,r1 => r2
   loadI 0      => r0
   store r2     => r0
   output 0
```



```
% sim ex1.iloc -i 0 14 17
328

Executed 7 instructions and 7 operations in 11 cycles.
```

The ILOC Virtual Machine

Before Execution of the ILOC Program Starts



Invoked with the command line:

```
% sim –i 0 14 17 <ex1.iloc
```

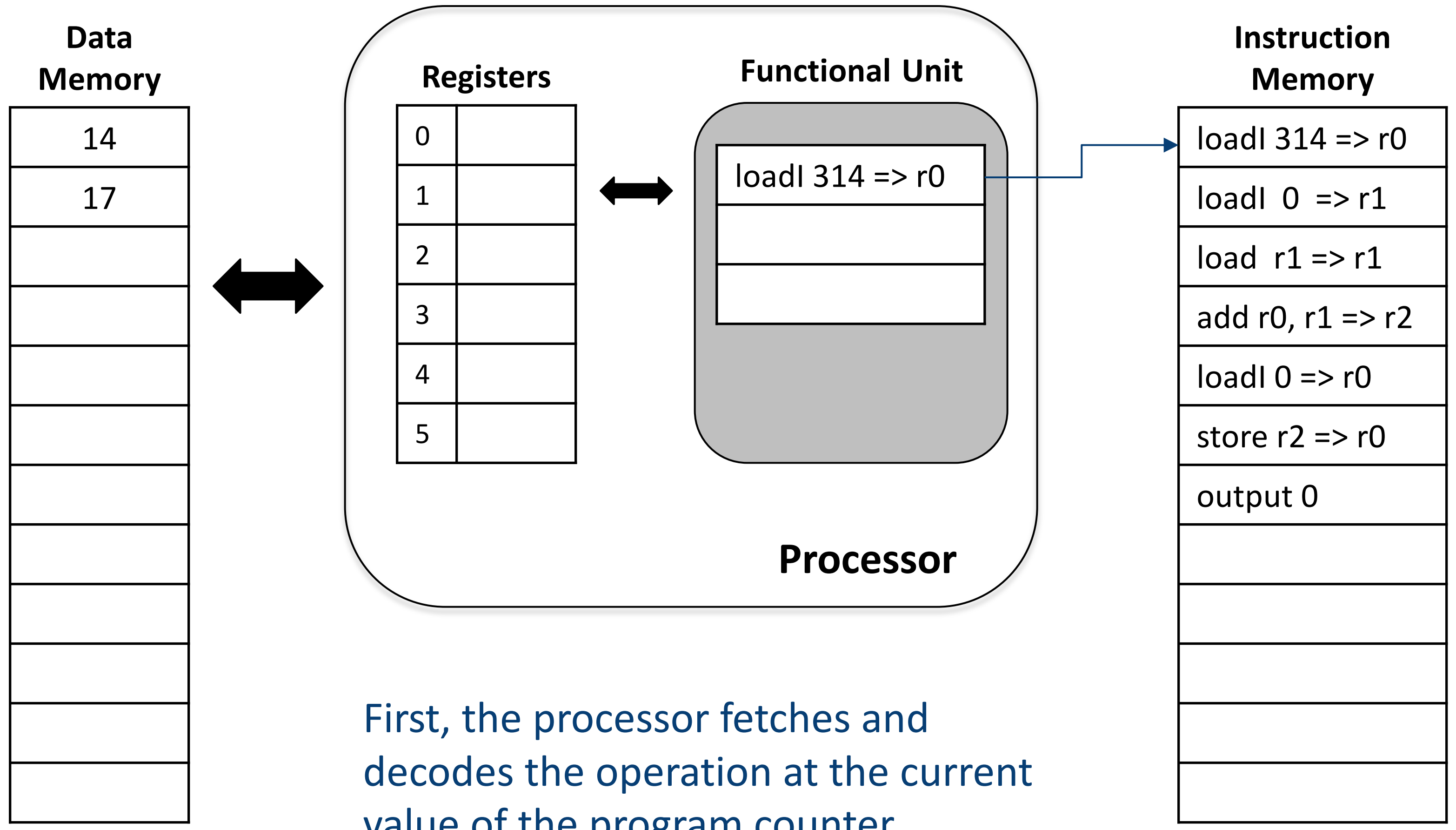
The **ILOC** Virtual Machine

The virtual machine runs through the code, in order

- The basic unit of execution is a “cycle”
- A cycle consists of a fetch phase and an execute phase
  - Execution looks like (fetch, execute) (fetch, execute) …
- Fetch retrieves the next operation from code memory
  - Advances sequentially through the straight-line code
- Execute performs the specified operation
  - Performs one “step” on each “active” operation
  - Multi-cycle operations (e.g., load and store in lab 1) are divided into multiple steps
  - Execution (on the processor’s functional unit) uses a pipeline of operation “steps”
    - → Load and store proceed through three stages or steps in the pipeline
    - → The illustrated example should make this more clear

The ILOC Virtual Machine

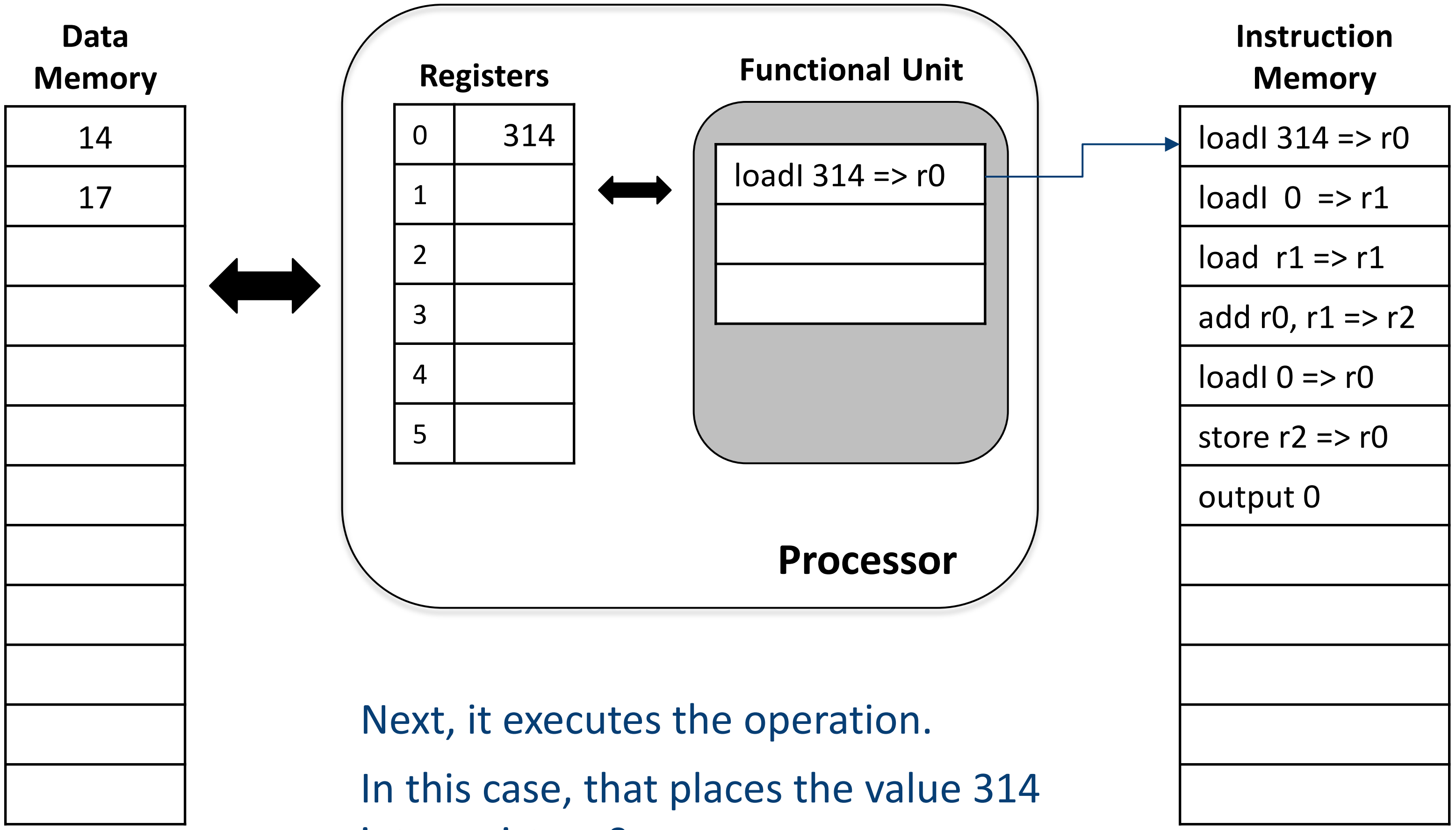
Cycle 0: Fetch Phase



First, the processor fetches and decodes the operation at the current value of the program counter.

The ILOC Virtual Machine

Cycle 0: Execute Phase



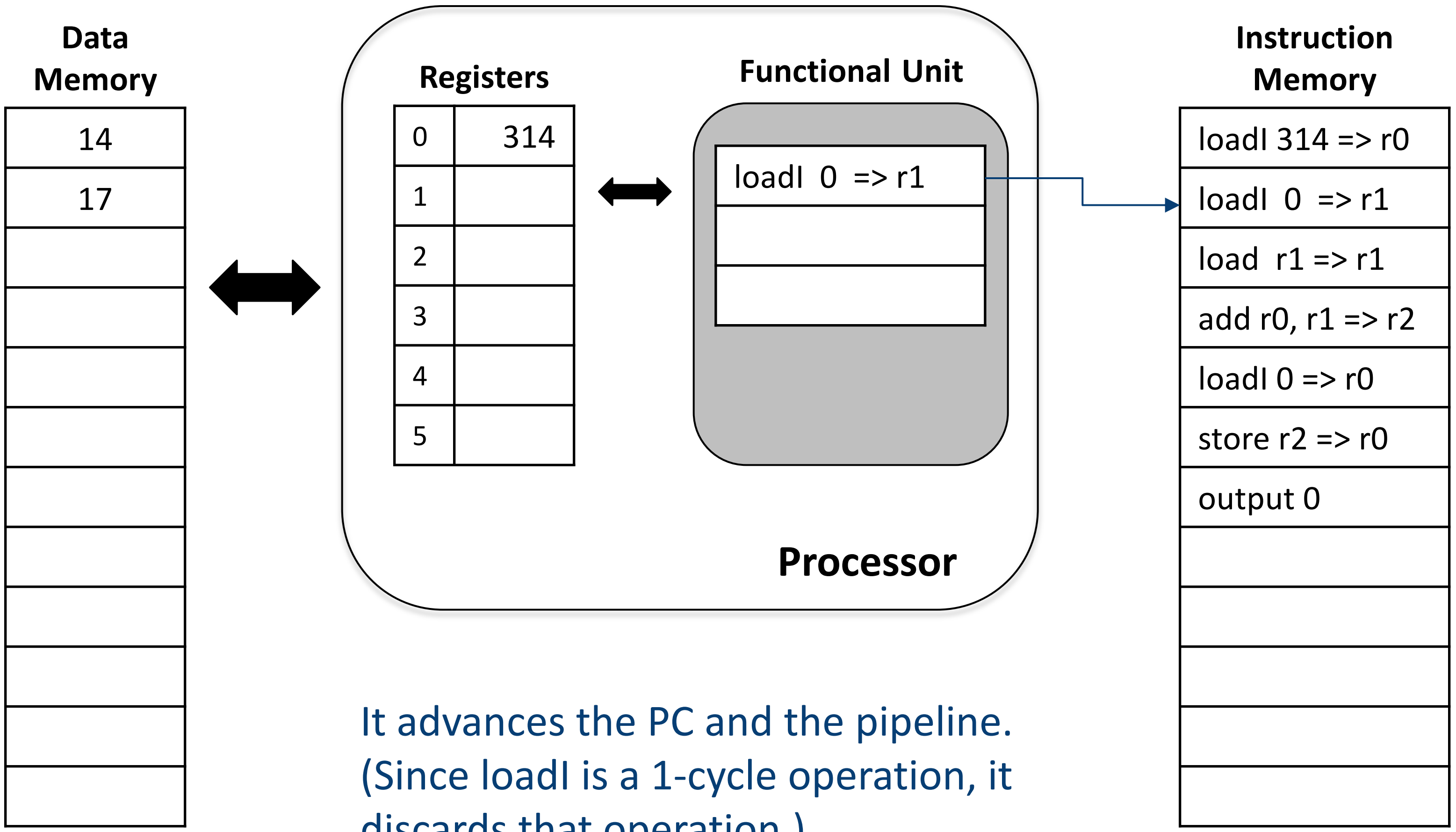
Next, it executes the operation.

In this case, that places the value 314 into register r0.

Trace output: `0:  [loadI 314 => r0 (314)]`

The ILOC Virtual Machine

Cycle 1: Fetch Phase

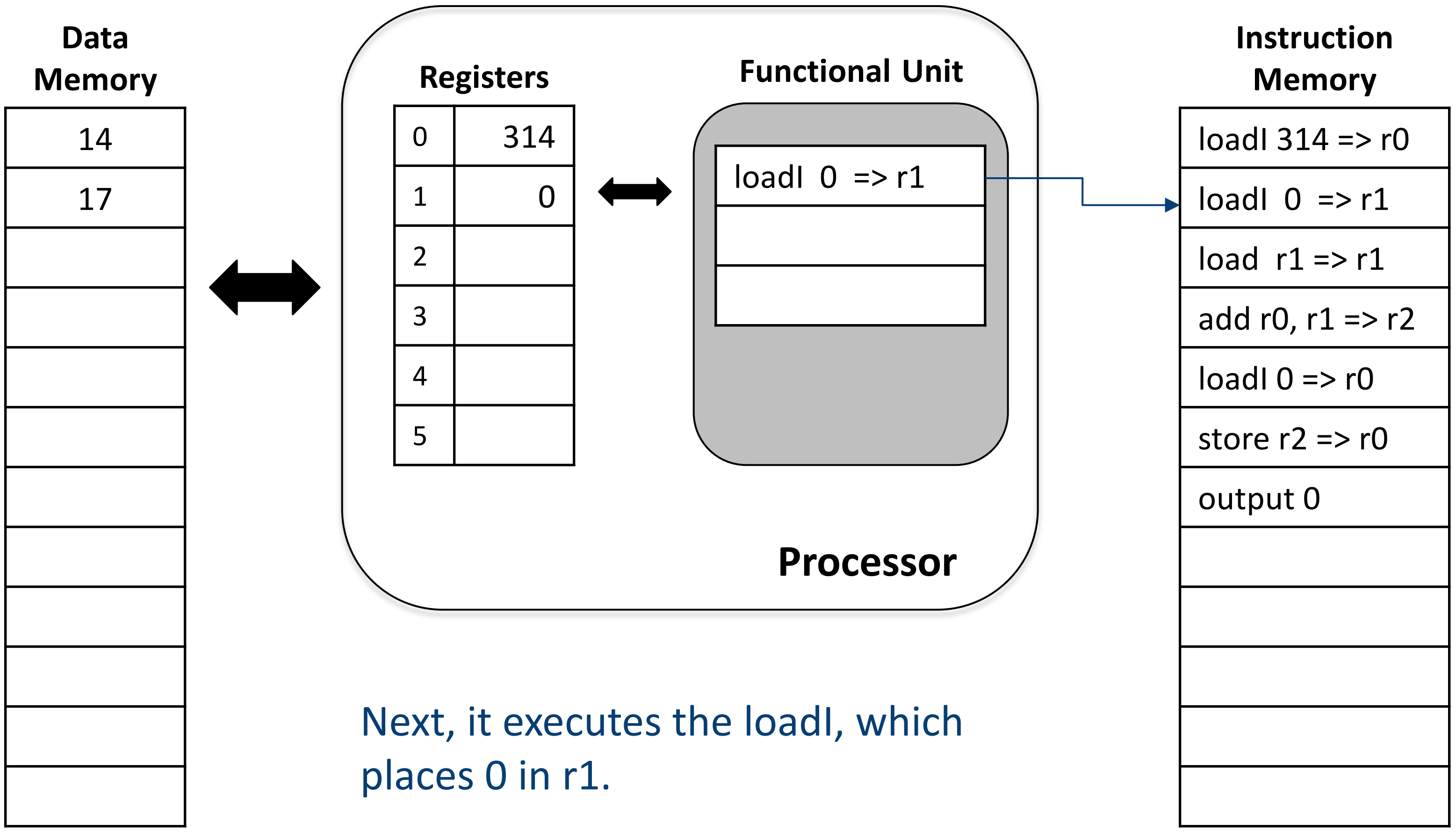


It advances the PC and the pipeline. (Since loadI is a 1-cycle operation, it discards that operation.)

It fetches the next operation.

The ILOC Virtual Machine

Cycle 1: Execute Phase

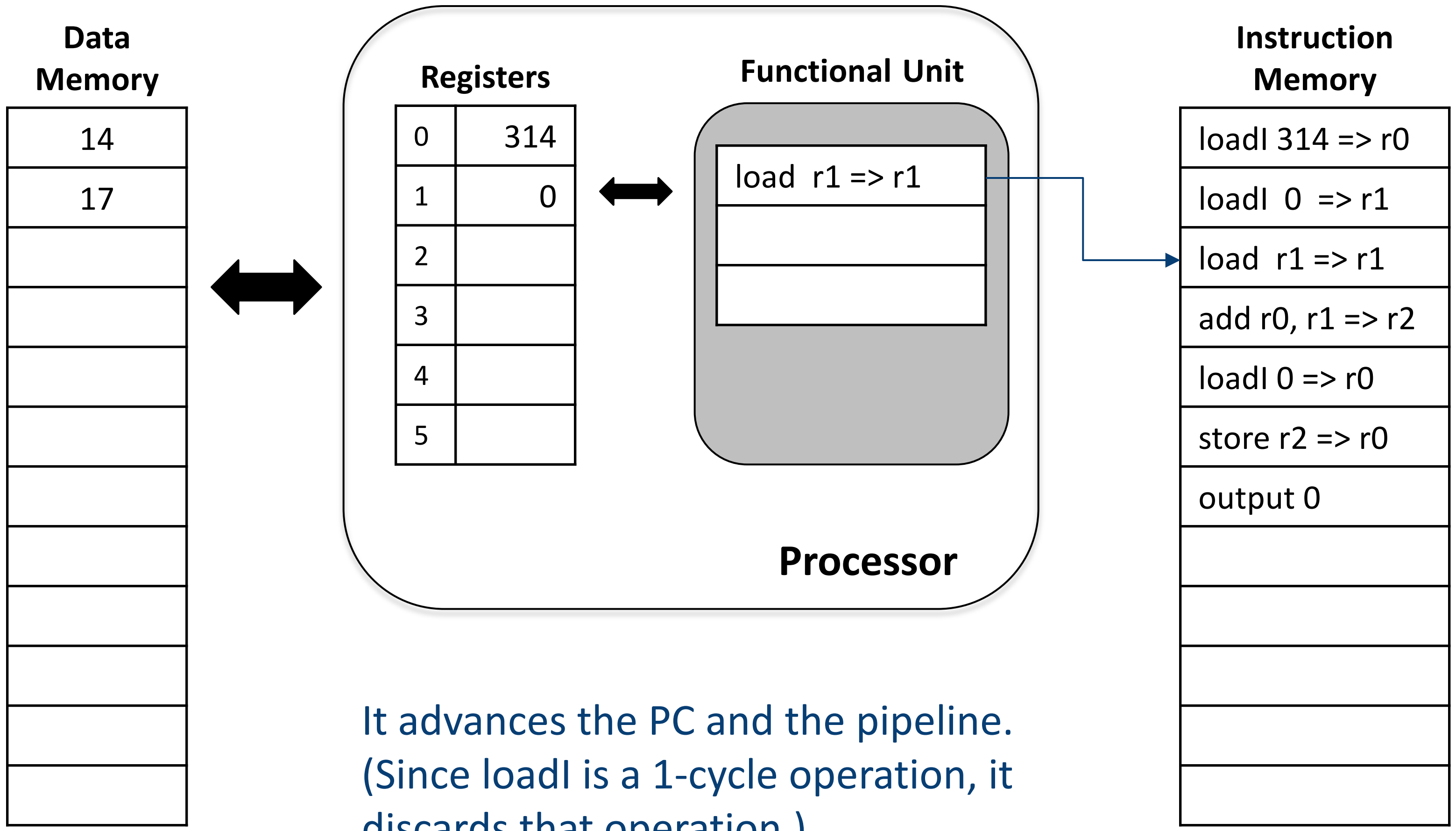


Next, it executes the loadI, which places 0 in r1.

Trace output: `1:  [loadI 0 => r1 (0)]`

The ILOC Virtual Machine

Cycle 2: Fetch Phase

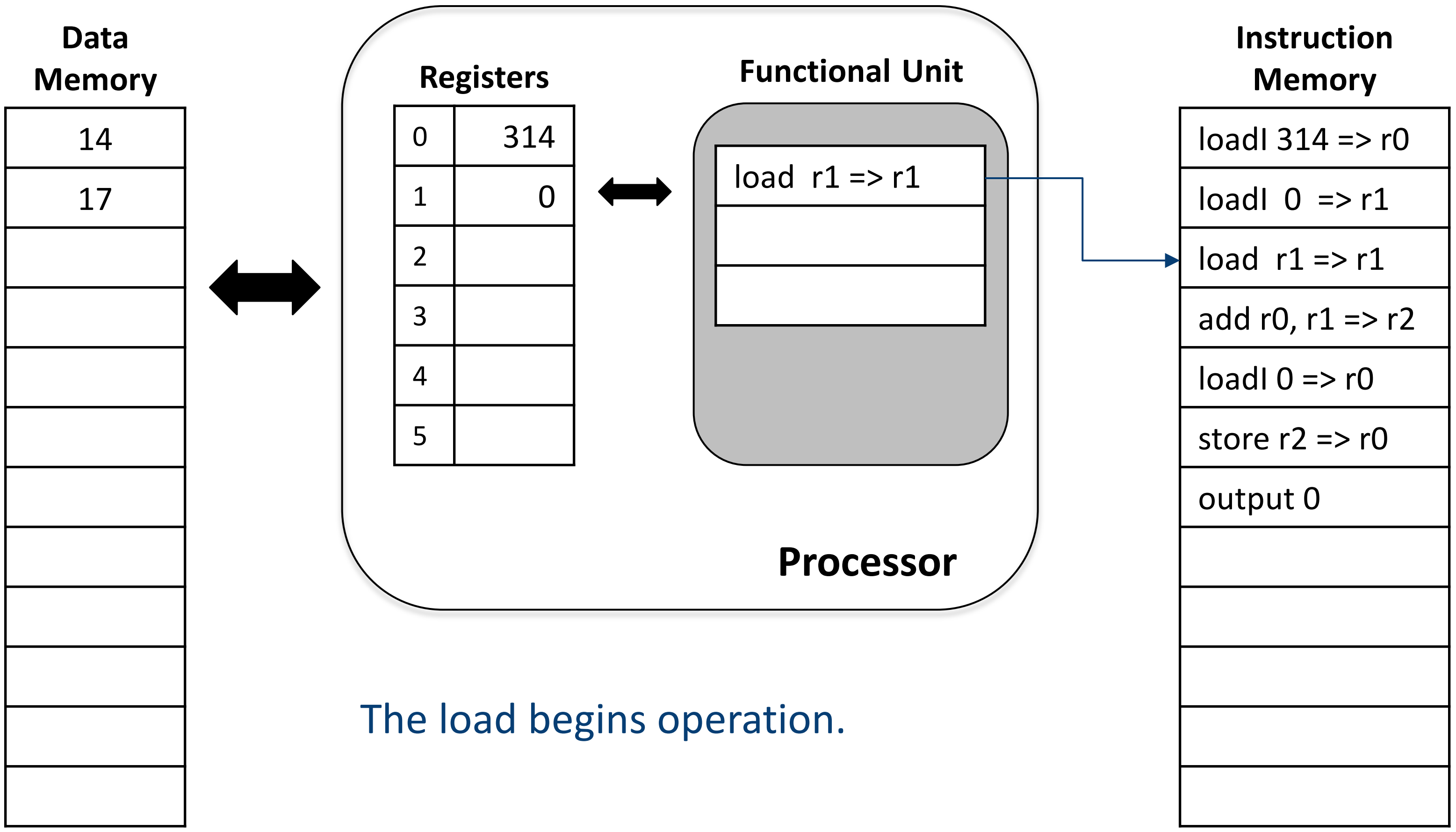


It advances the PC and the pipeline. (Since loadI is a 1-cycle operation, it discards that operation.)

It fetches the next operation.

The ILOC Virtual Machine

Cycle 2: Execute Phase



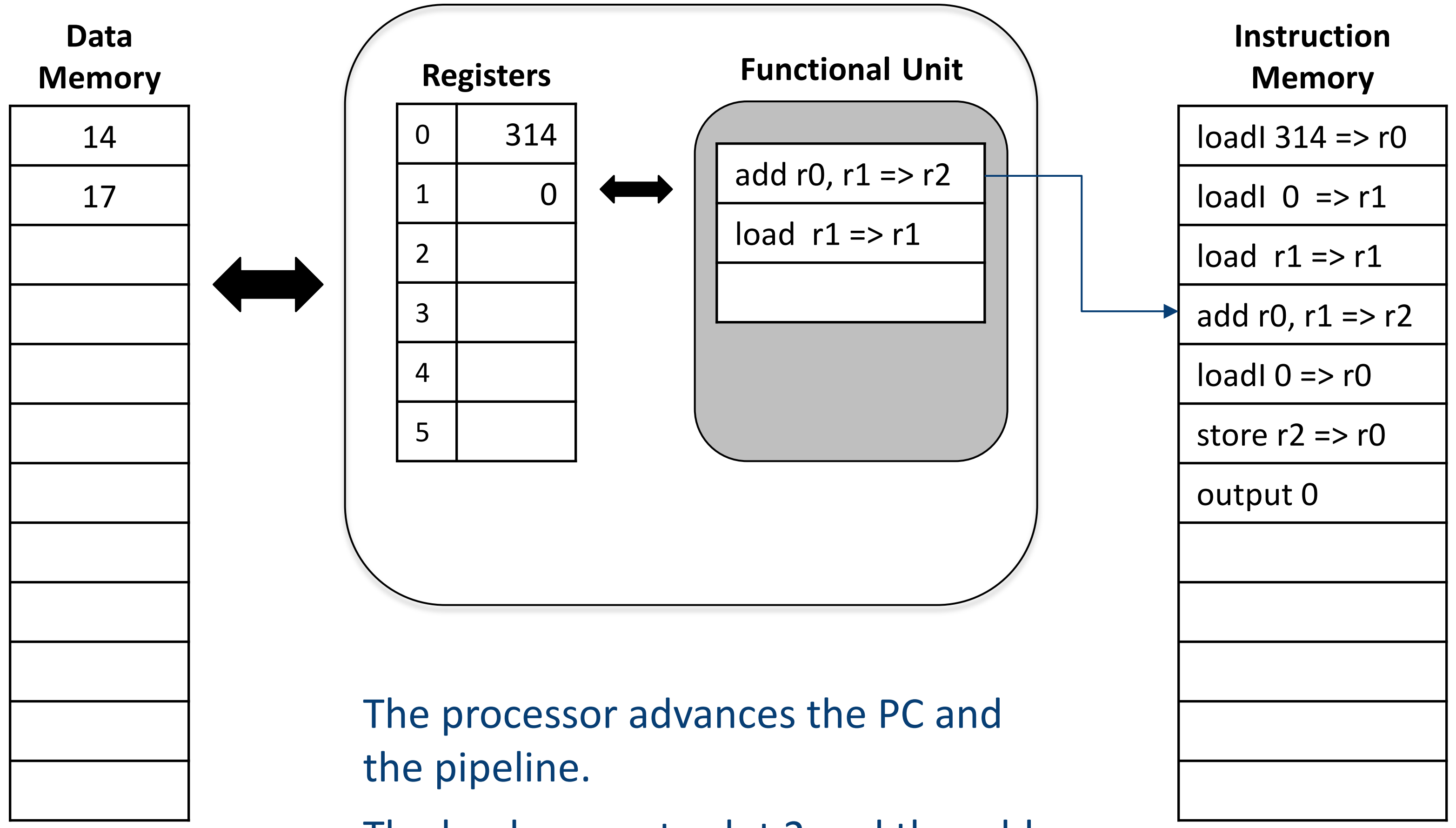
The load begins operation.

Trace output: `2: [load r1 (addr: 0) => r1 (14)`

The ILOC Virtual Machine

pipelined functional unit

Cycle 3: Fetch Phase

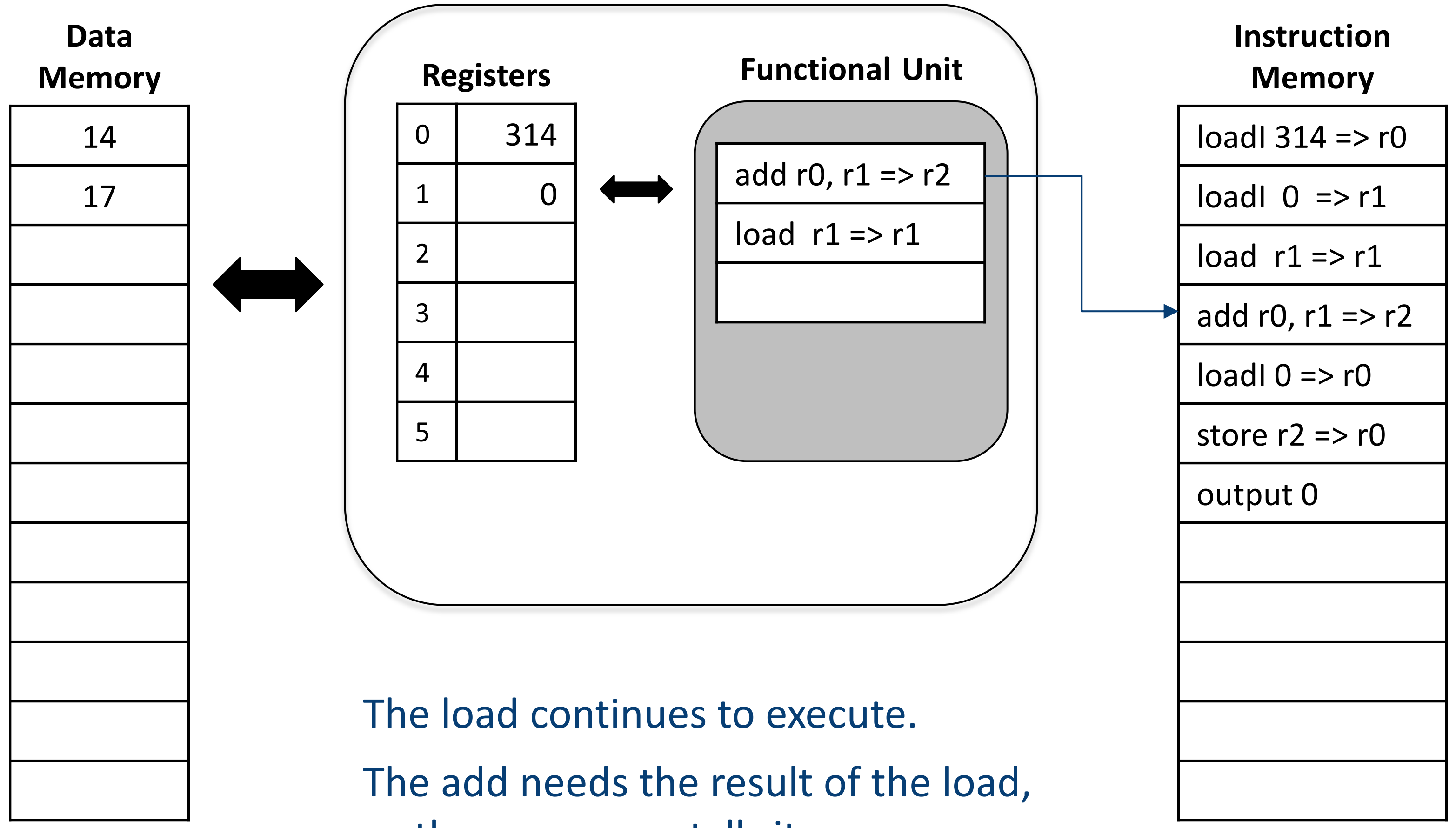


The processor advances the PC and the pipeline.

The load moves to slot 2 and the add fills slot 1.

The ILOC Virtual Machine

Cycle 3: Execute Phase



The load continues to execute.

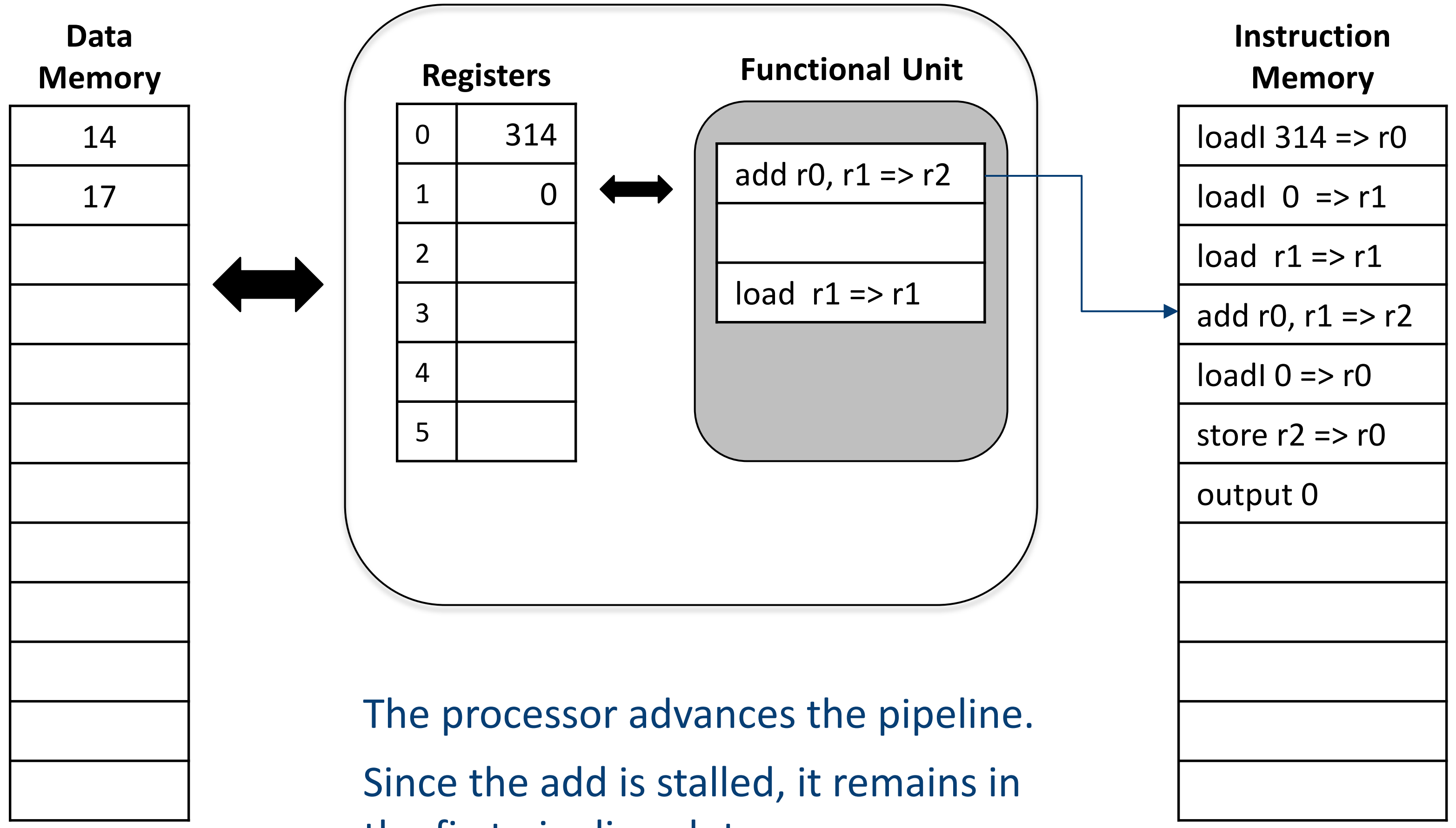
The add needs the result of the load, so the processor stalls it.

Trace output: `3: [ stall ]`

"stall" means to hold the op for another cycle

The ILOC Virtual Machine

Cycle 4: Fetch Phase

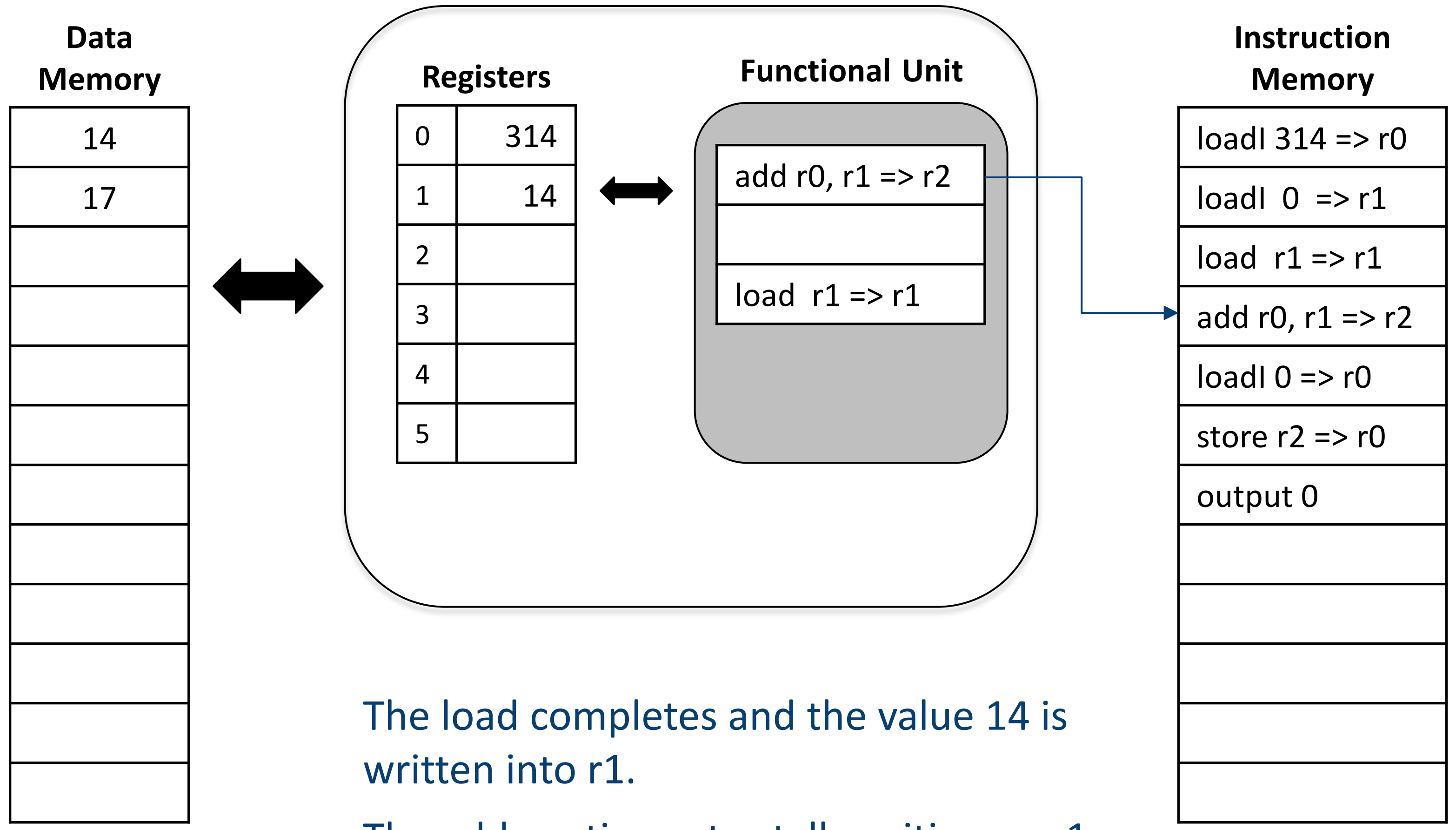


The processor advances the pipeline.

Since the add is stalled, it remains in the first pipeline slot.

The ILOC Virtual Machine

Cycle 4: Execute Phase



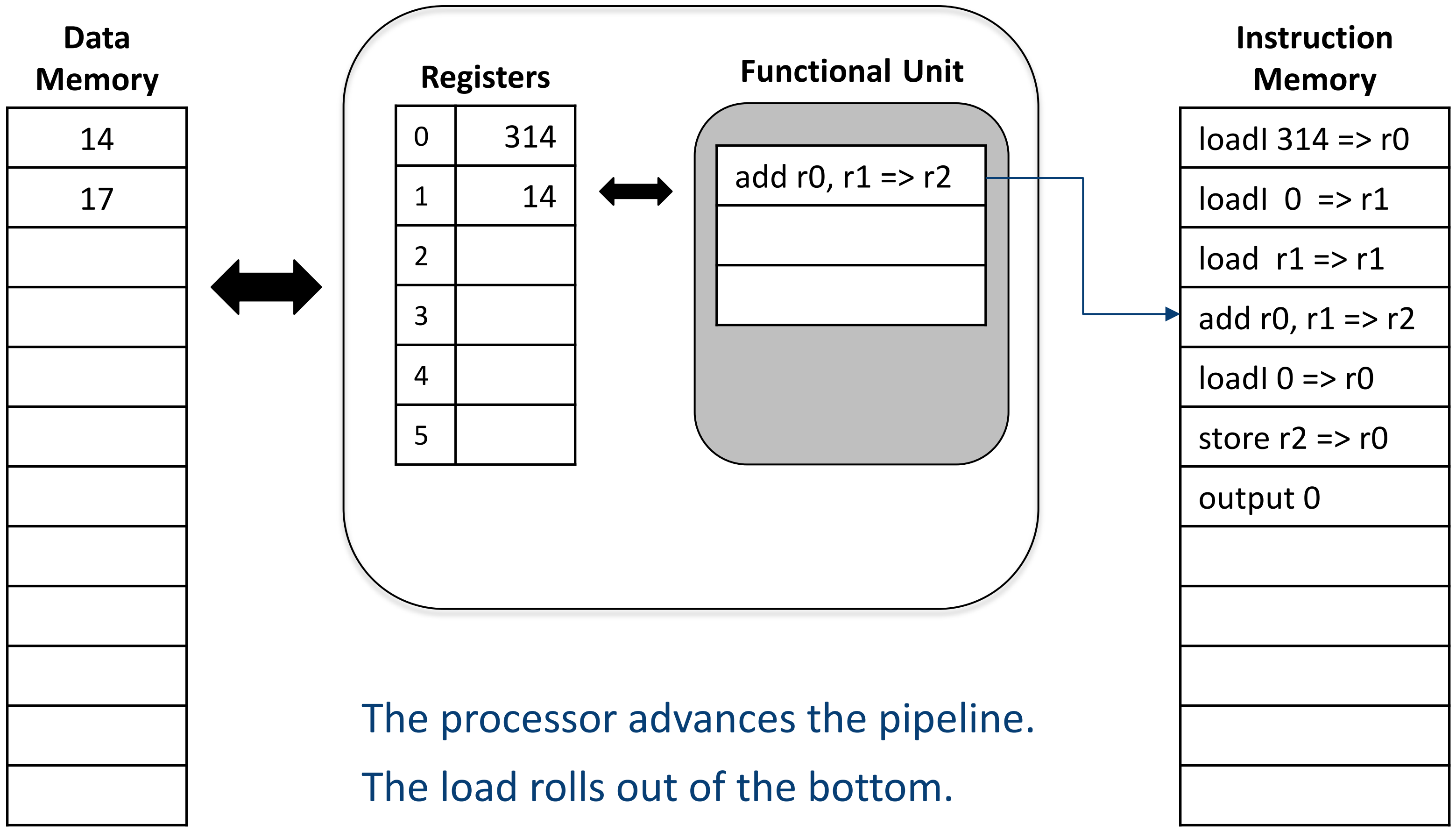
The load completes and the value 14 is written into r1.

The add continues to stall, waiting on r1.

Trace output: `4: [ stall ] *2`

The ILOC Virtual Machine

Cycle 5: Fetch Phase



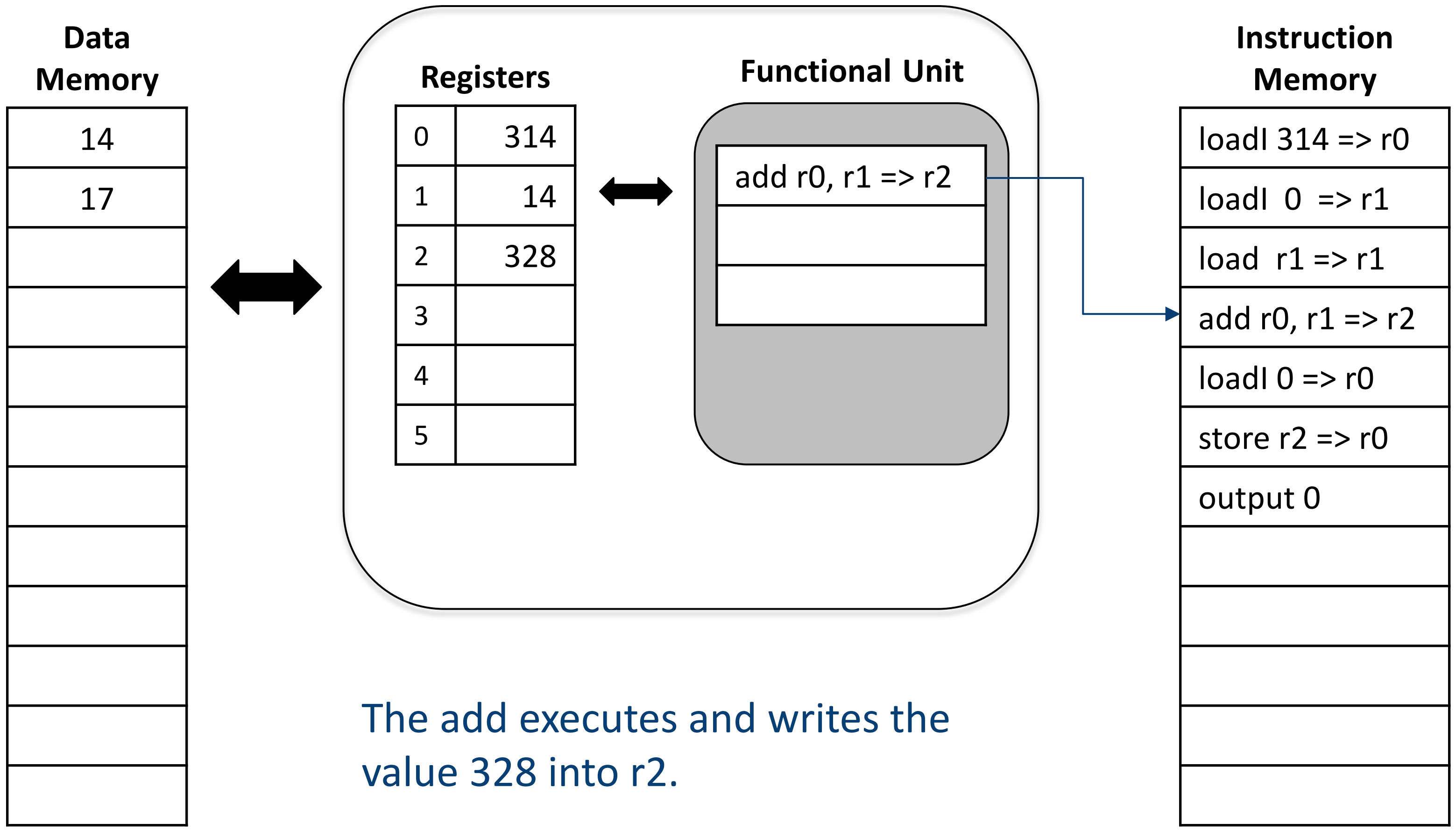
The processor advances the pipeline.

The load rolls out of the bottom.

The add remains in slot 1.

The ILOC Virtual Machine

Cycle 5: Execute Phase

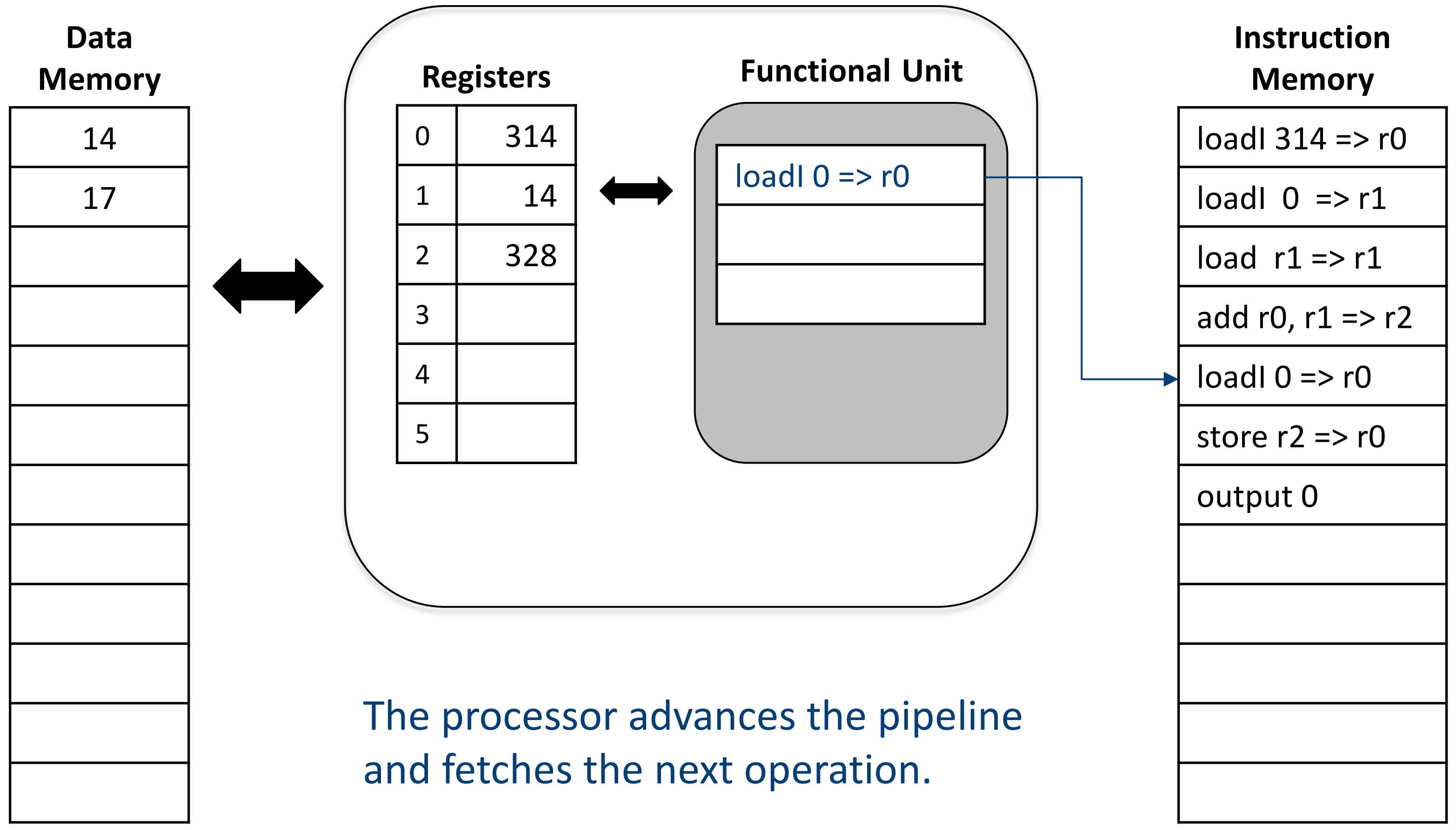


The add executes and writes the value 328 into r2.

Trace output: `5: [add r0 (314), r1 (14) => r2 (328)]`

The ILOC Virtual Machine

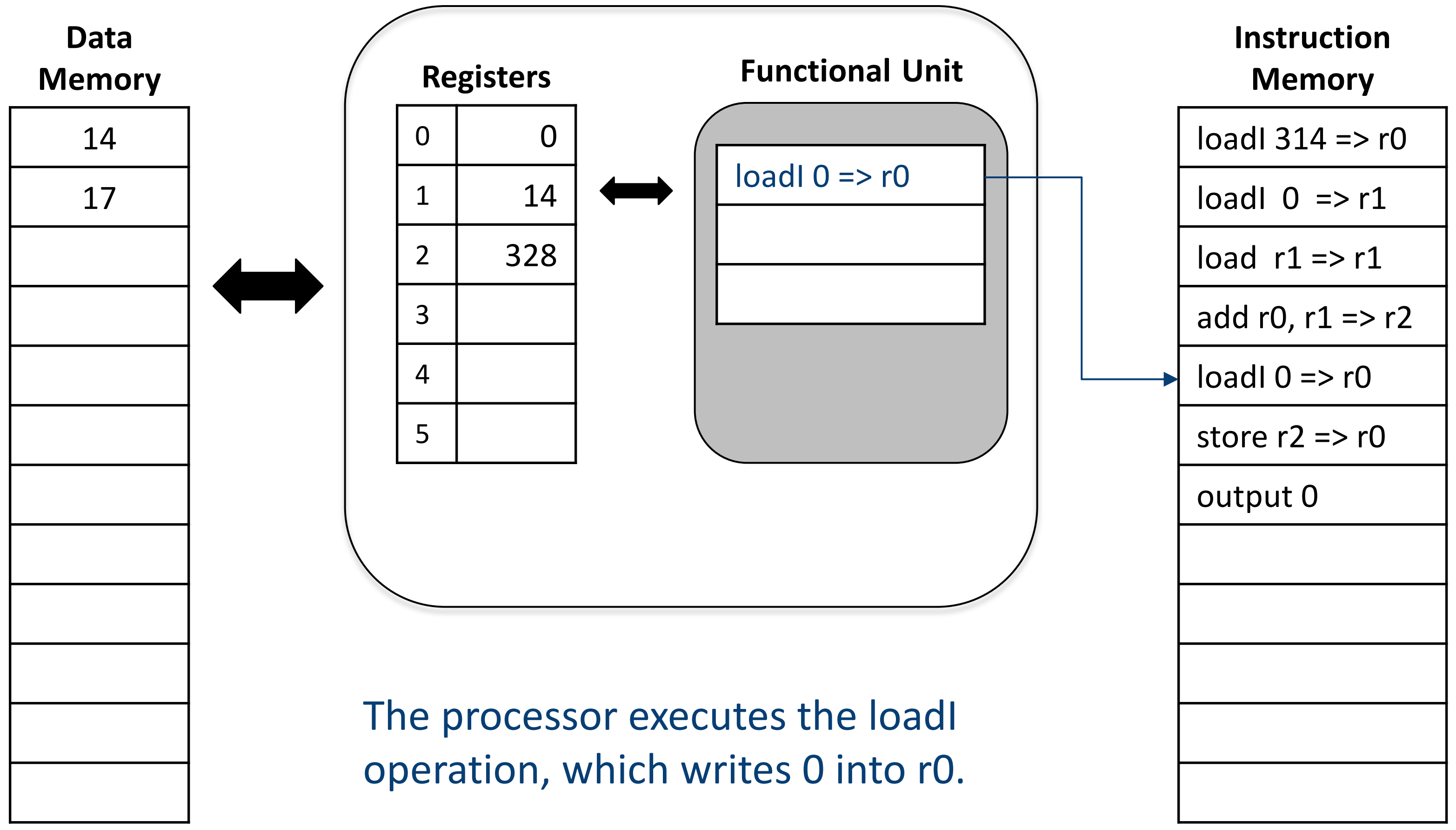
Cycle 6: Fetch Phase



The processor advances the pipeline and fetches the next operation.

The ILOC Virtual Machine

Cycle 6: Execute Phase

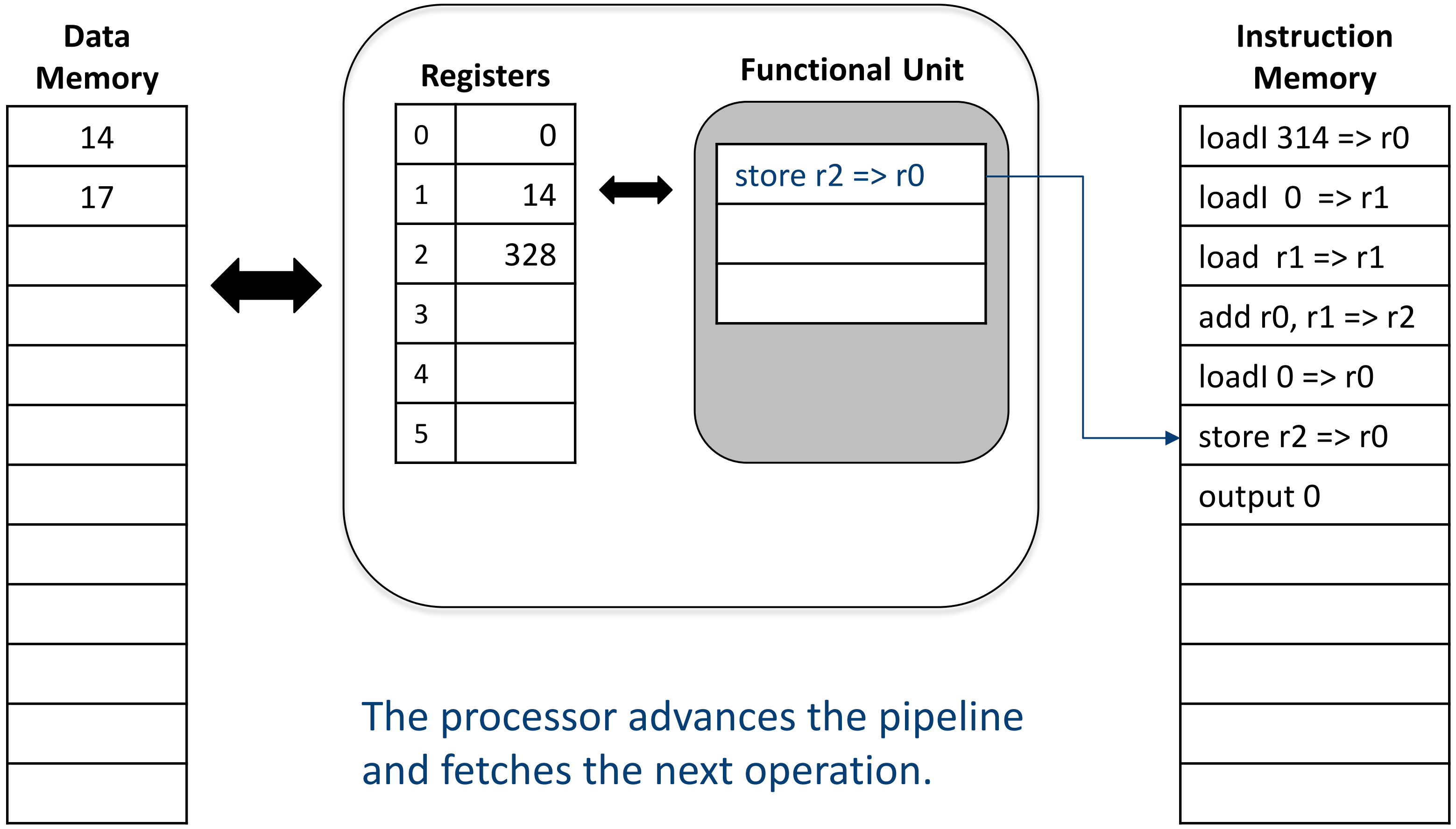


The processor executes the loadI operation, which writes 0 into r0.

Trace output: `6: [loadI 0 => r0 (0)]`

The ILOC Virtual Machine

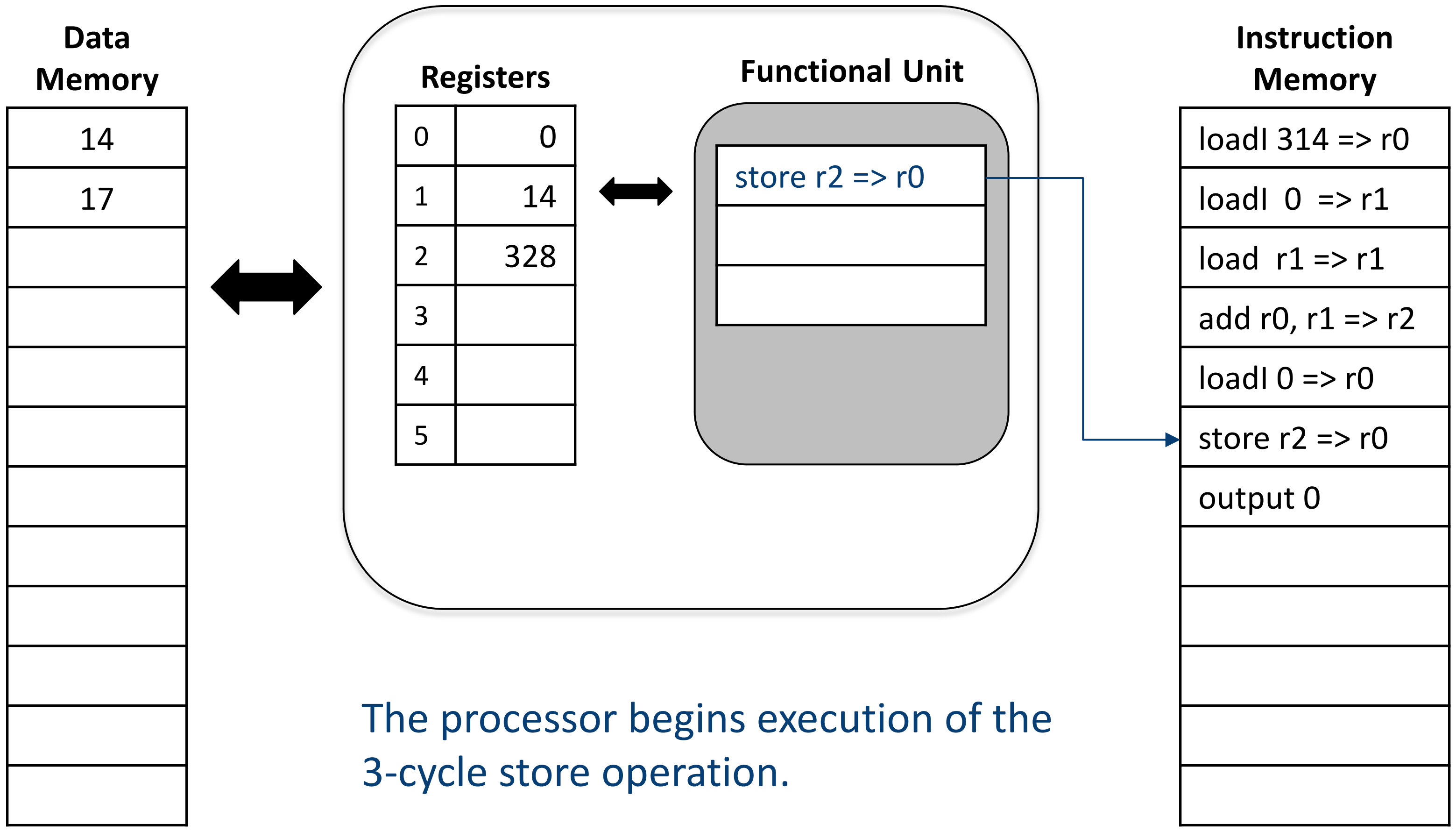
Cycle 7: Fetch Phase



The processor advances the pipeline and fetches the next operation.

The ILOC Virtual Machine

Cycle 7: Execute Phase

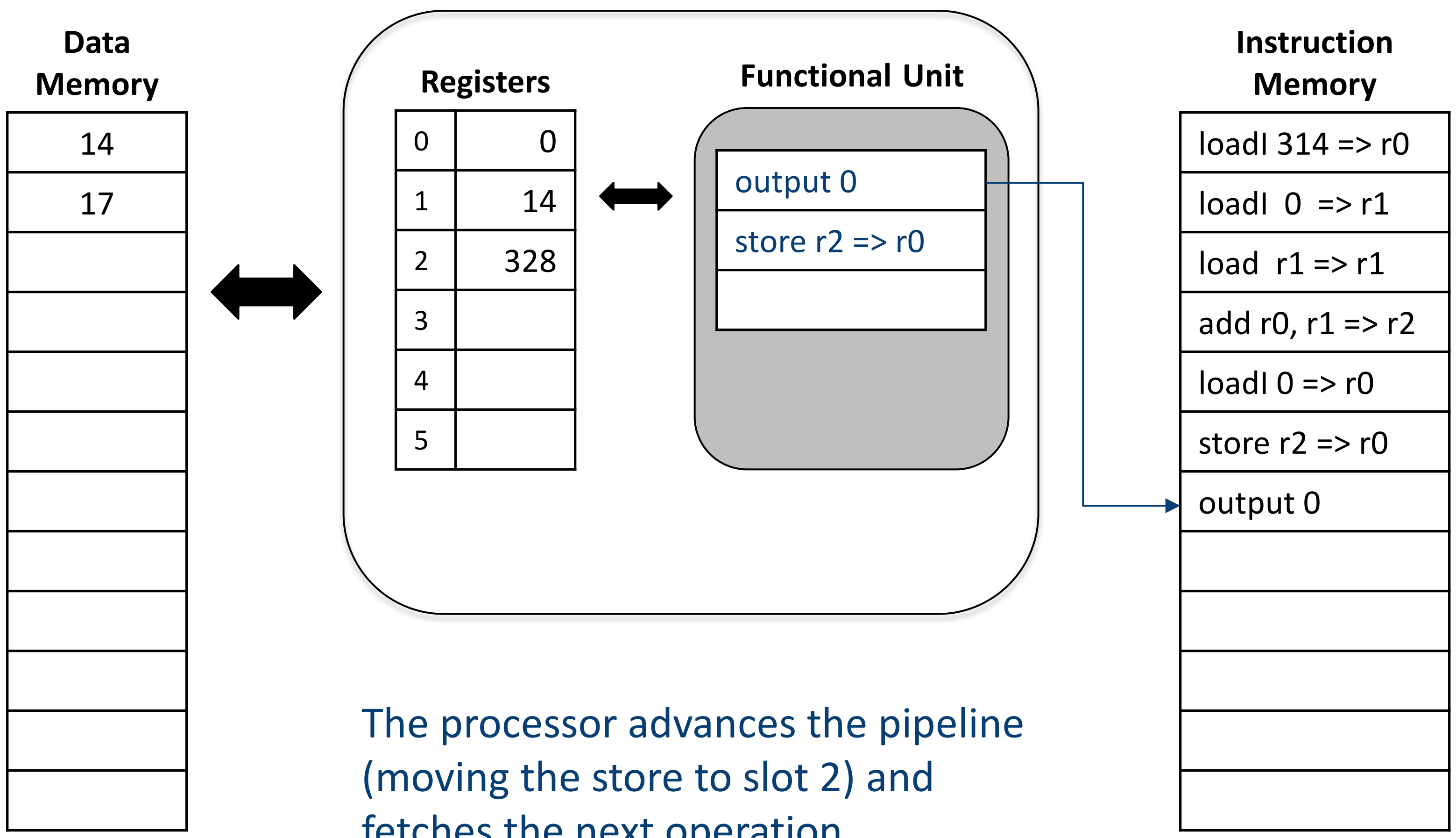


The processor begins execution of the 3-cycle store operation.

Trace output: `7: [store r2 (328) => r0 (addr: 0)]`

The ILOC Virtual Machine

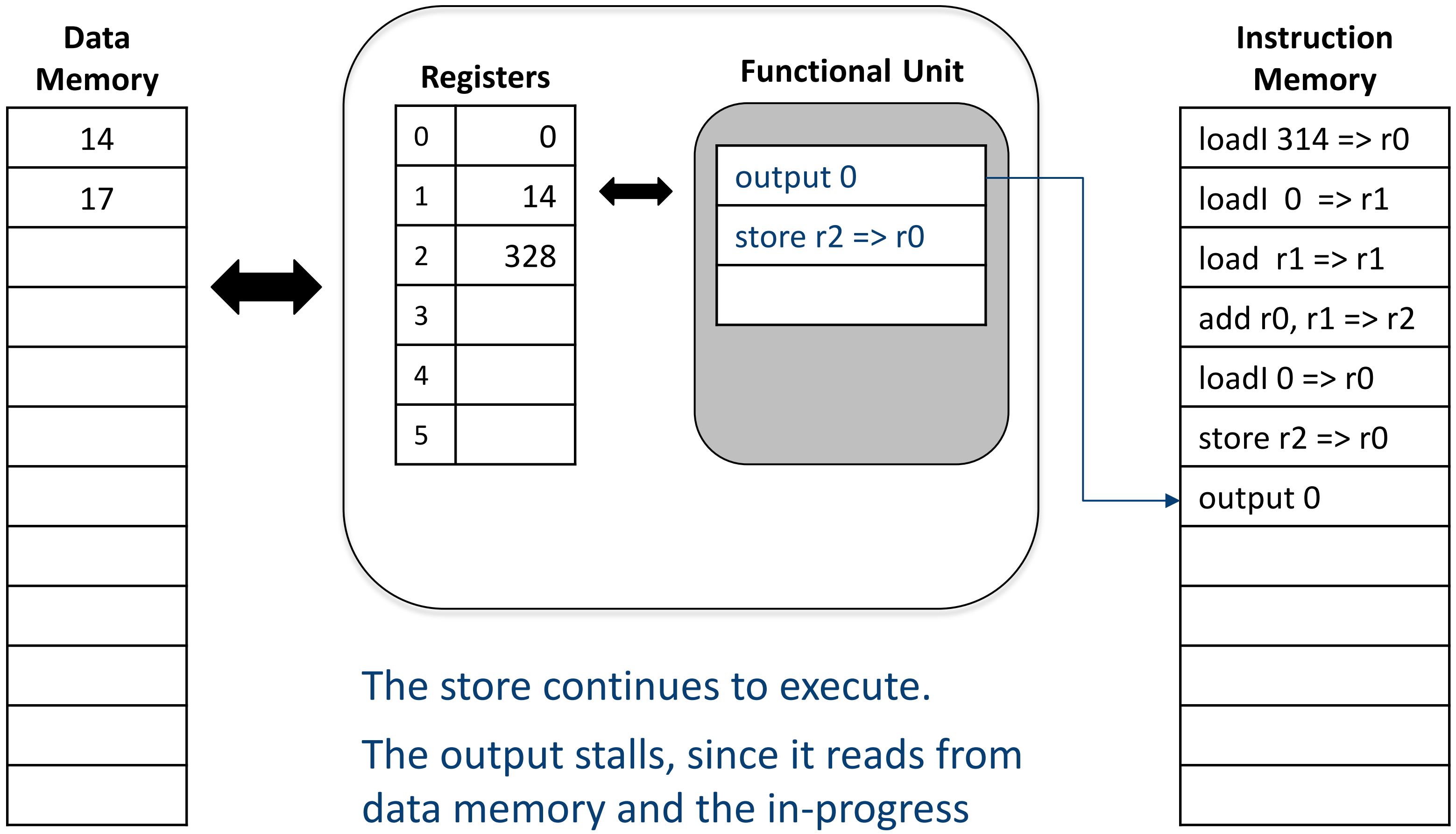
Cycle 8: Fetch Phase



The processor advances the pipeline (moving the store to slot 2) and fetches the next operation

The ILOC Virtual Machine

Cycle 8: Execute Phase



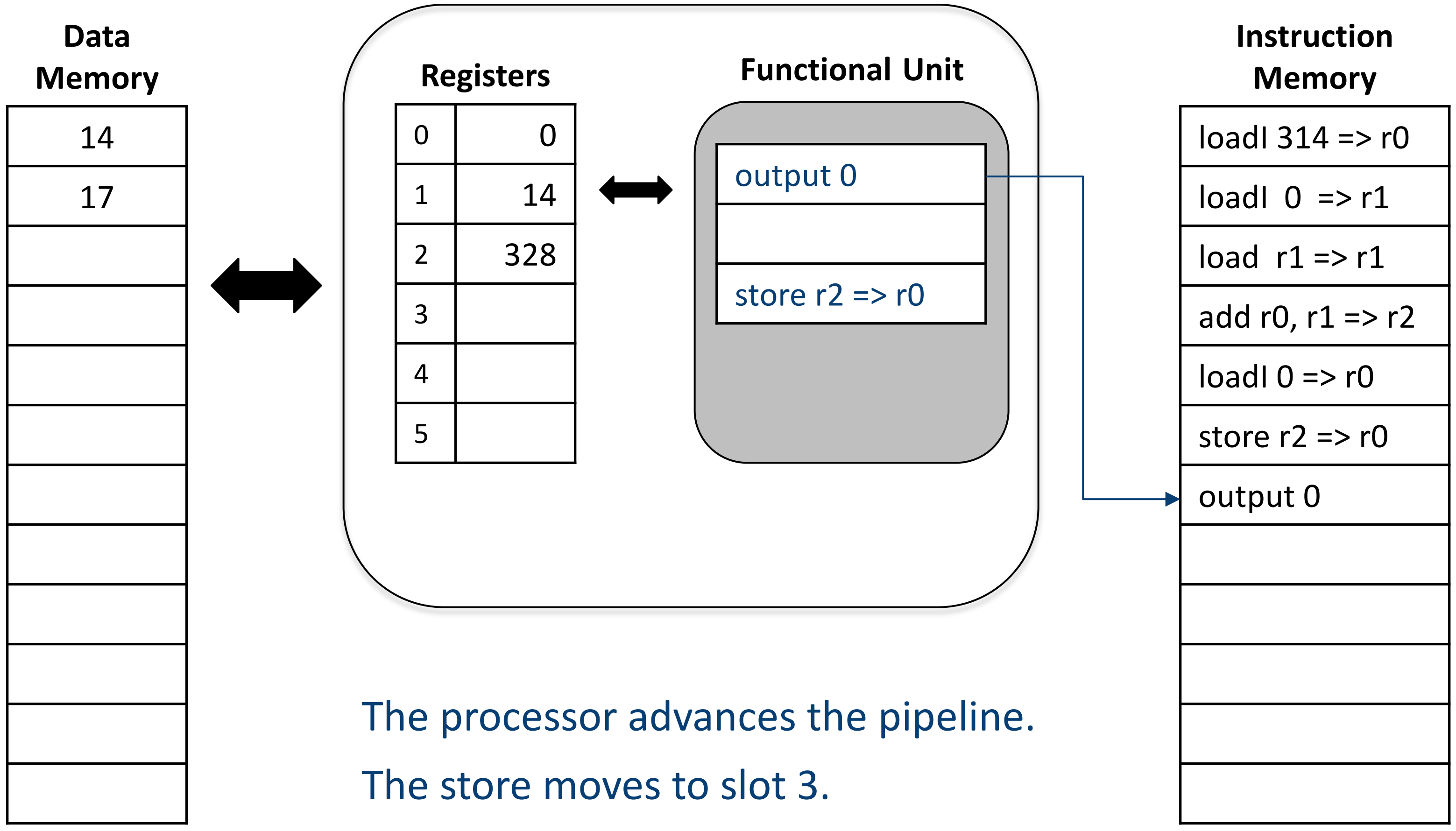
The store continues to execute.

The output stalls, since it reads from data memory and the in-progress store writes to data memory.

Trace output: `8: [ stall ]`

The ILOC Virtual Machine

Cycle 9: Fetch Phase



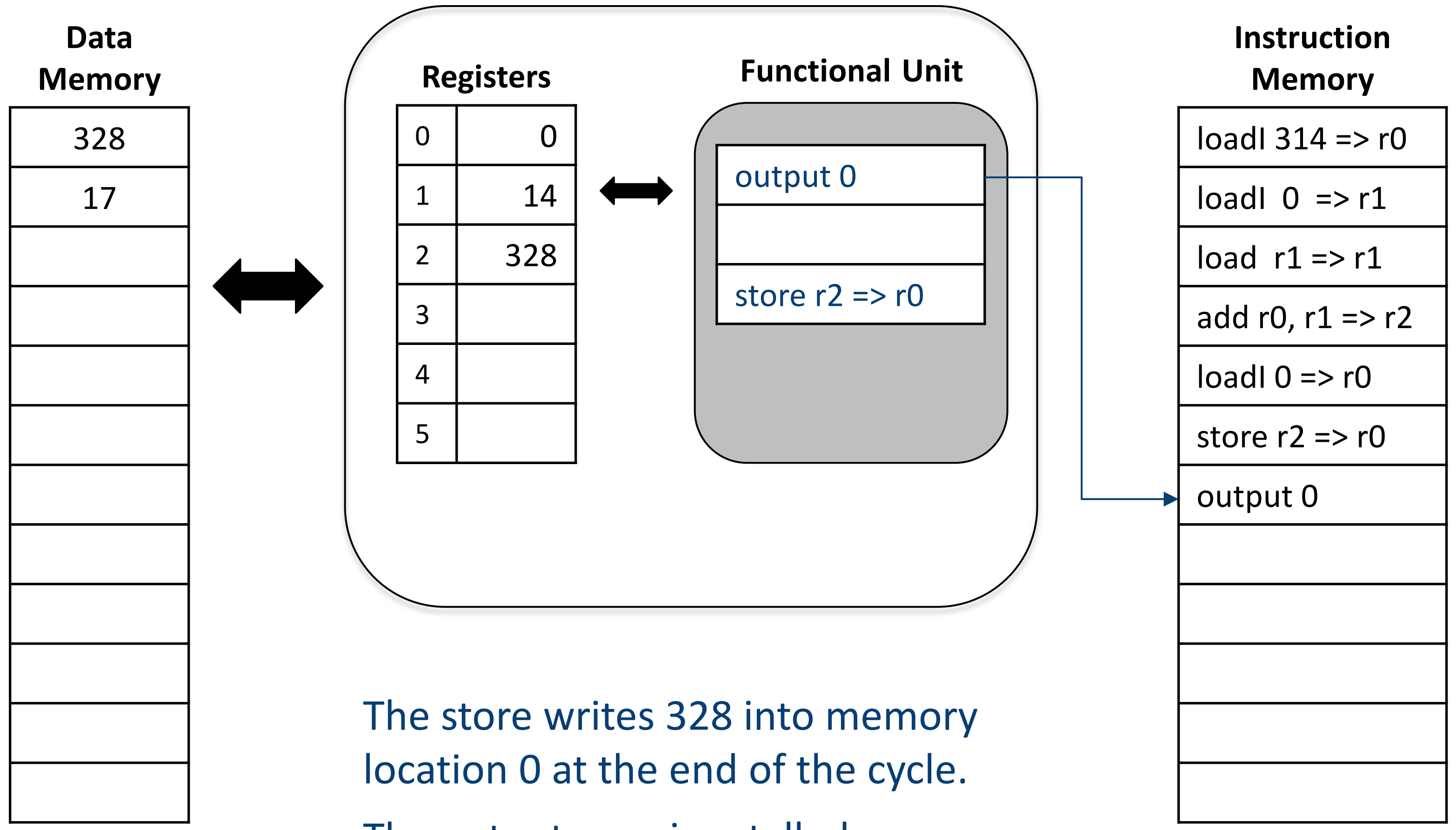
The processor advances the pipeline.

The store moves to slot 3.

The stalled output operation remains in slot 1, waiting for the store to finish.

The ILOC Virtual Machine

Cycle 9: Execute Phase



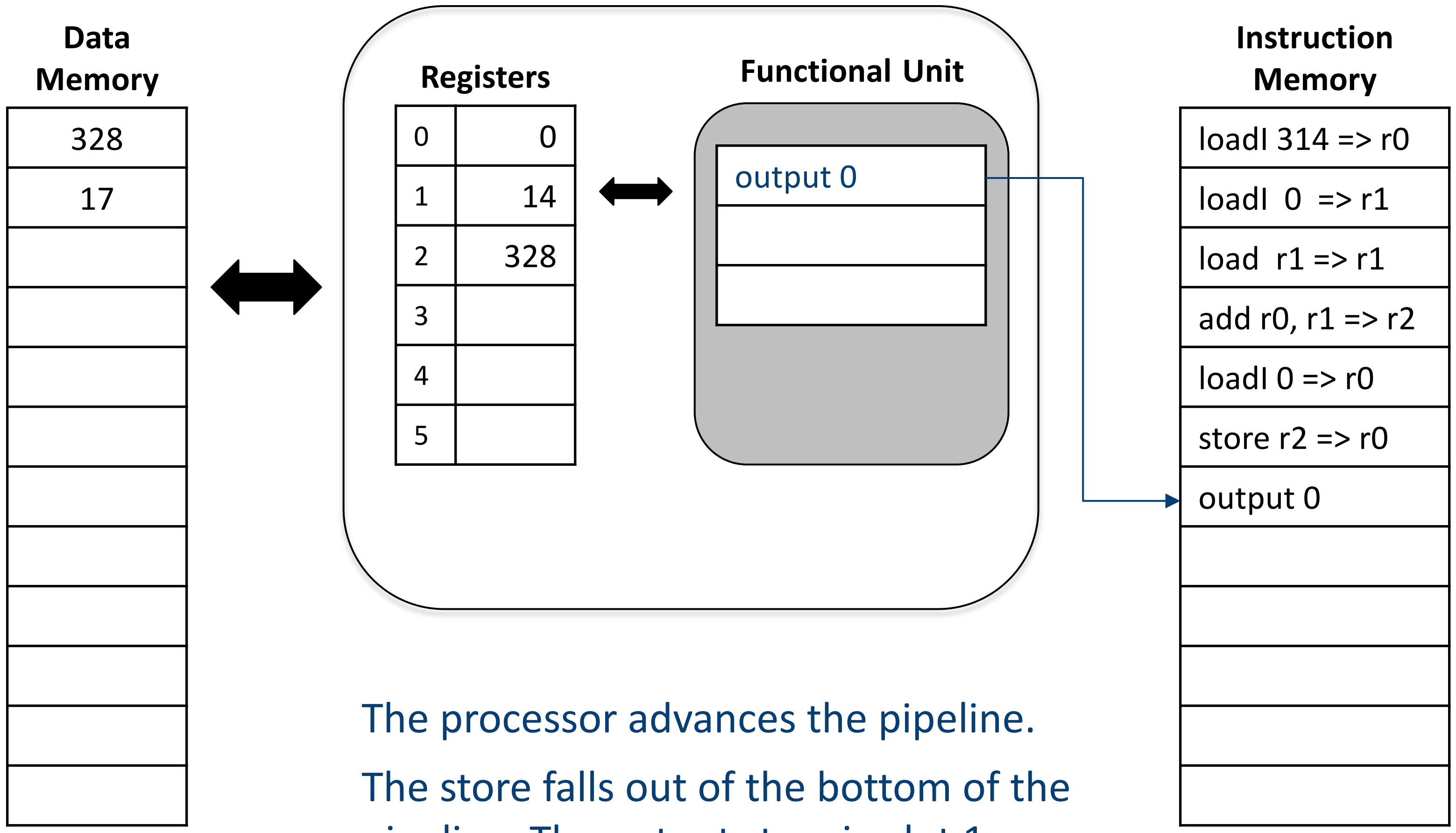
The store writes 328 into memory location 0 at the end of the cycle.

The output remains stalled.

Trace output: `9: [ stall ] *7`

The ILOC Virtual Machine

Cycle 10: Fetch Phase

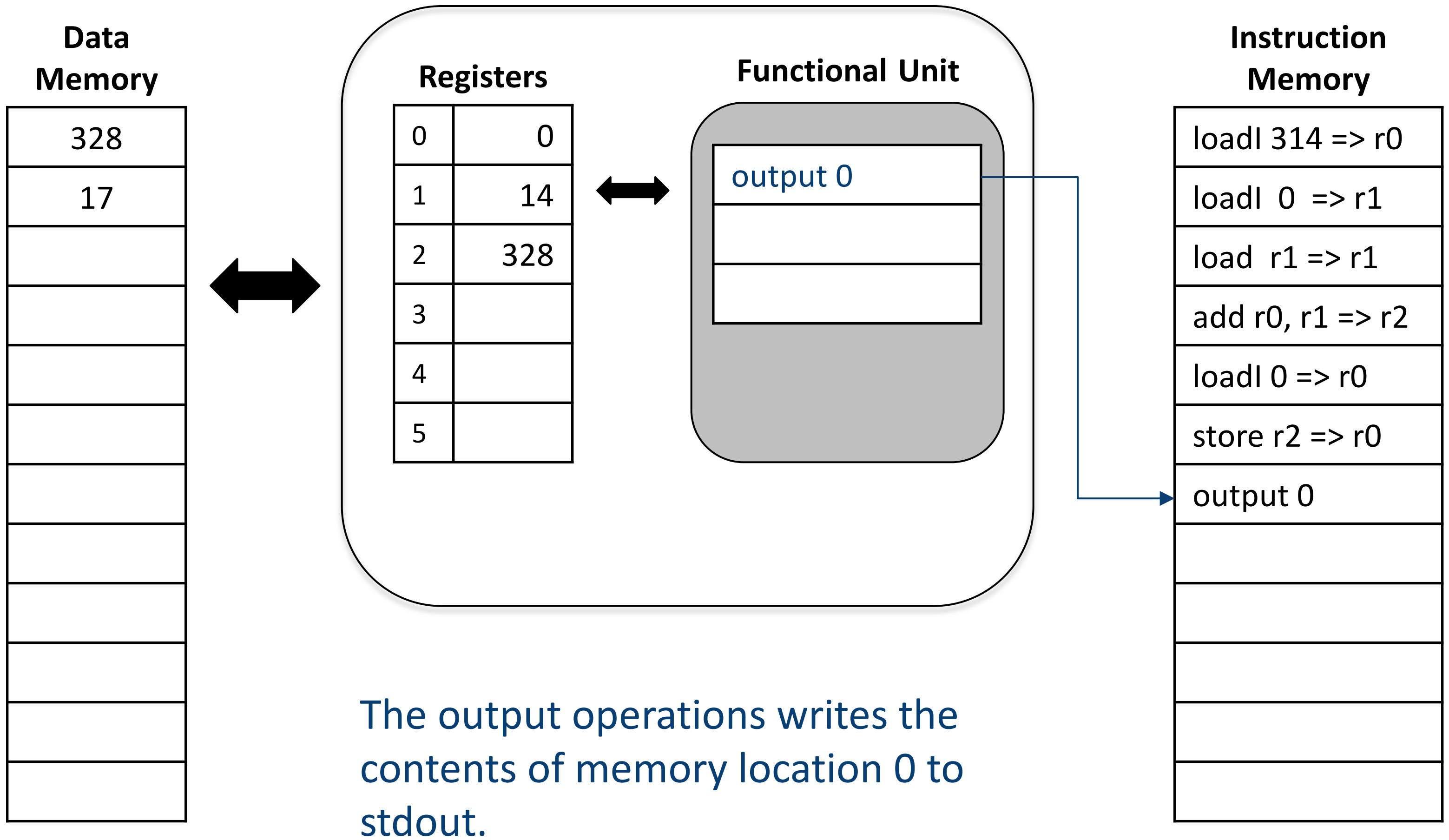


The processor advances the pipeline.

The store falls out of the bottom of the pipeline. The output stays in slot 1.

The ILOC Virtual Machine

Cycle 10: Execute Phase



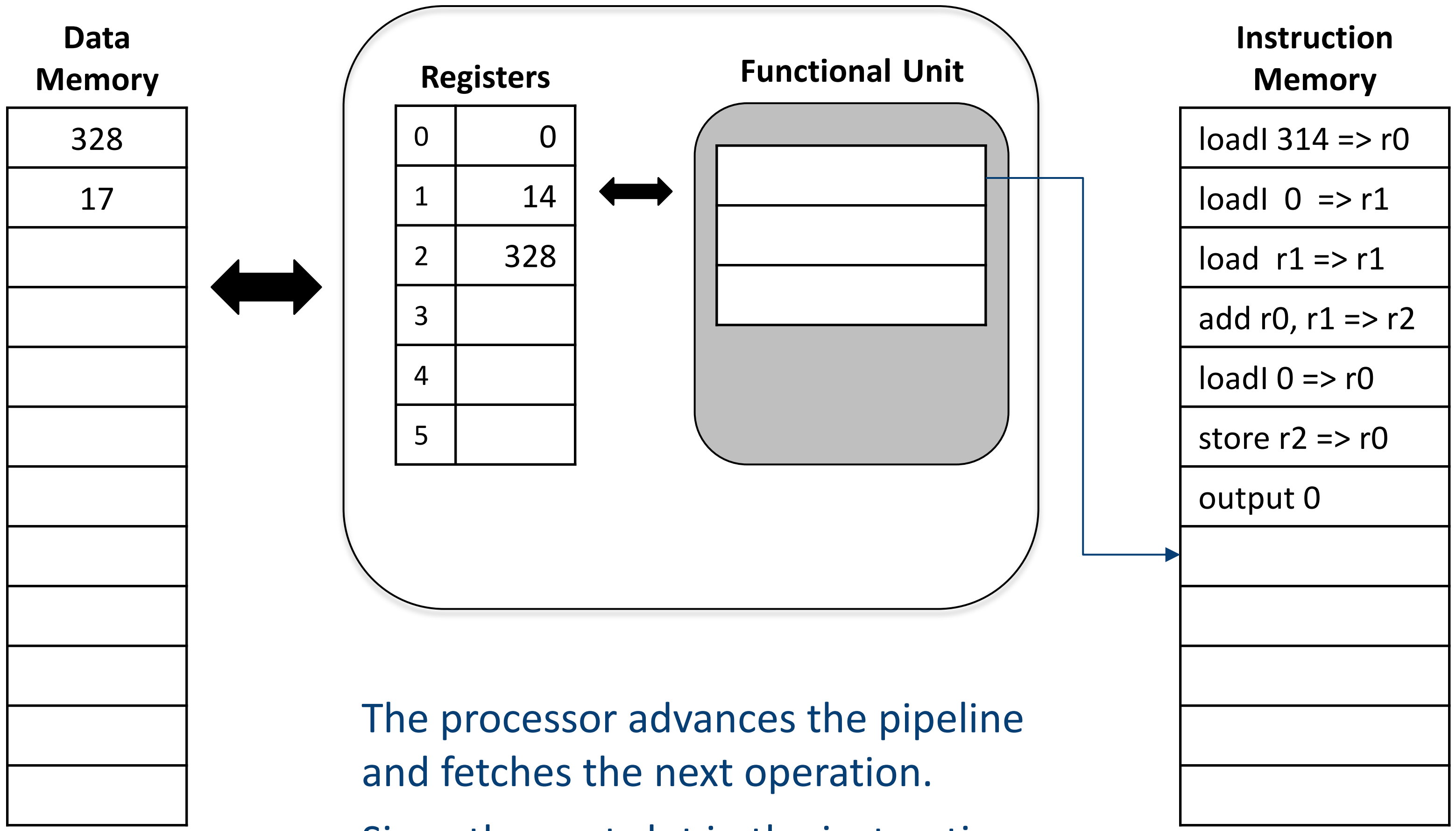
The output operations writes the contents of memory location 0 to stdout.

Trace output:

```
10:  [output 0 (328)]
output generates => 328
```

The ILOC Virtual Machine

Cycle 11: Fetch Phase



The processor advances the pipeline and fetches the next operation.

Since the next slot in the instruction memory is invalid, the processor halts.

ILOC Execution

This execution is captured in the trace provided by the simulator

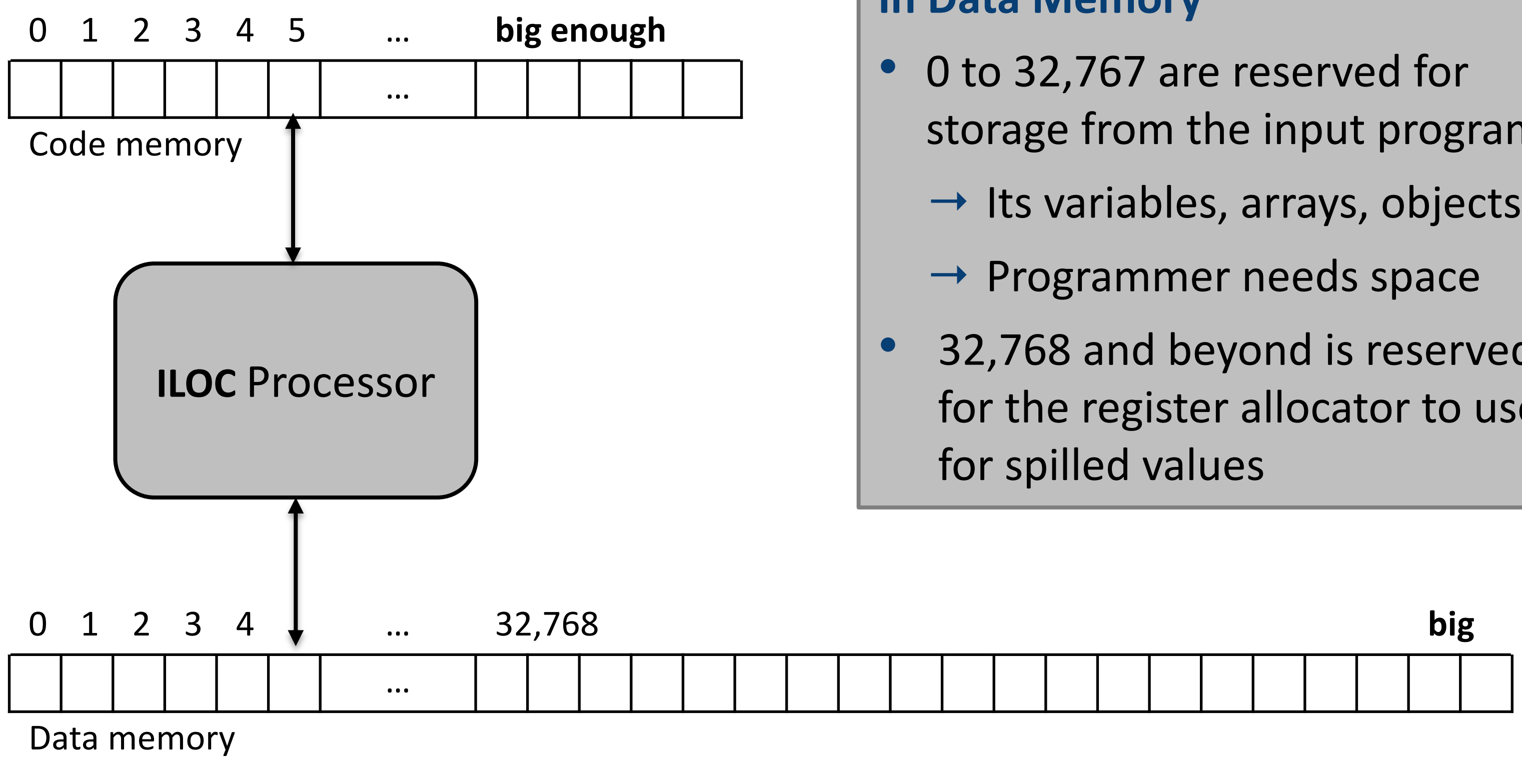
```
% cat ex1.iloc
// add two numbers
  loadI 314   => r0
  loadI   0   => r1
  load  r1    => r1
  add   r0,r1 => r2
  loadI 0     => r0
  store r2    => r0
  output 0
%
```

```
% sim -t ex1.iloc –i 0 14 17
ILOC Simulator, Version 412-2015-1
Interlock settings memory registers branches

0:   [loadI 314 => r0 (314)]
1:   [loadI 0 => r1 (0)]
2:   [load r1 (addr: 0) => r1 (14)]
3:   [ stall ]
4:   [ stall ] *2
5:   [add r0 (314), r1 (14) => r2 (328)]
6:   [loadI 0 => r0 (0)]
7:   [store r2 (328) => r0 (addr: 0)]
8:   [ stall ]
9:   [ stall ] *7
10:  [output 0 (328)]
output generates => 328

Executed 7 instructions and 7 operations
in 11 cycles.
```

Compare the simulator's trace output against the preceding slides.

The ILOC Virtual Machine

The Memory Model in the ILOC Virtual Machine



In Data Memory

- 0 to 32,767 are reserved for storage from the input program
  - → Its variables, arrays, objects
  - → Programmer needs space
- 32,768 and beyond is reserved for the register allocator to use for spilled values

"big" can be changed from the command line. Zero cannot.

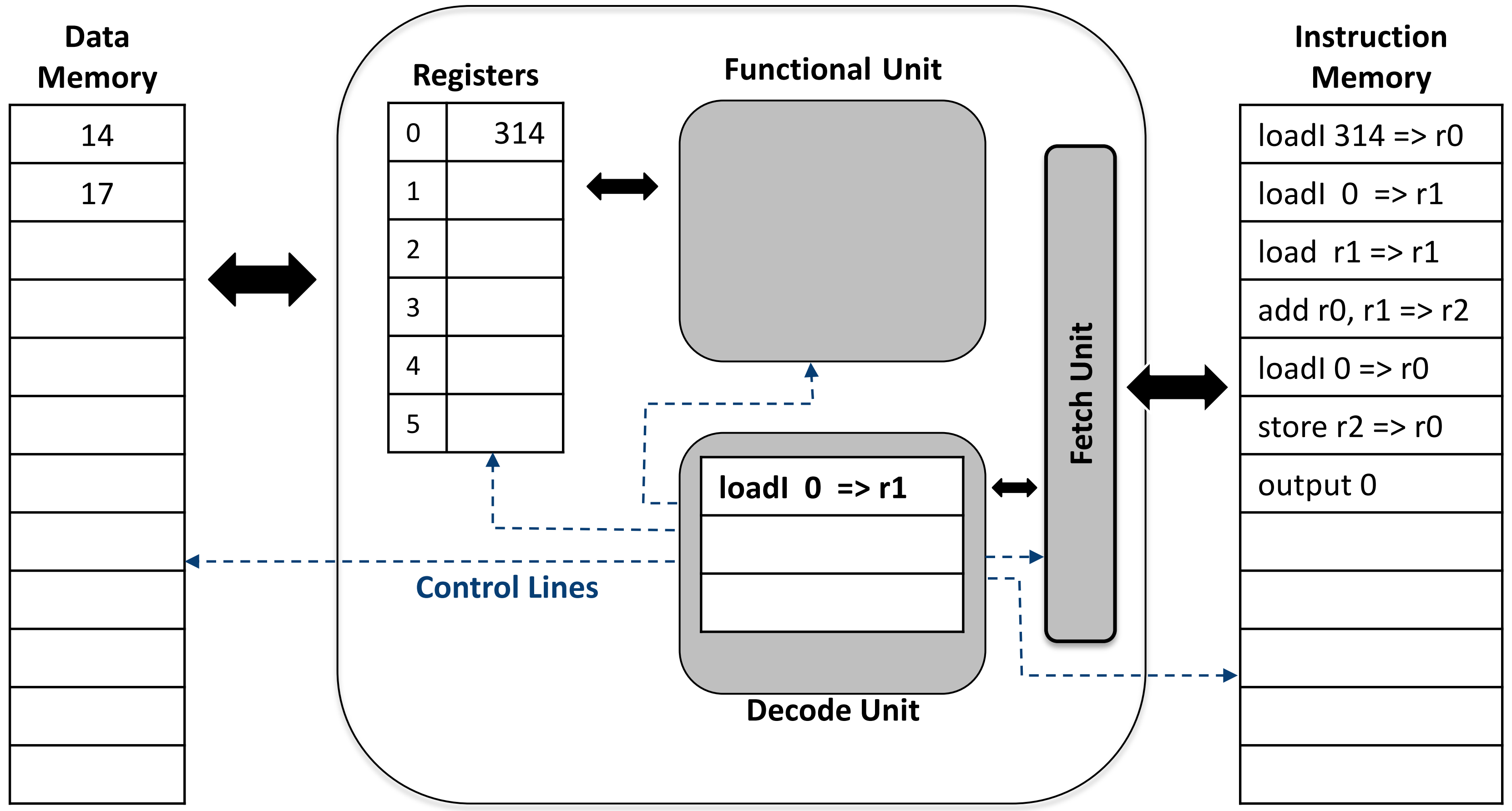
Does Real Hardware Work This Way?

In fact, the ILOC model is fairly close to reality

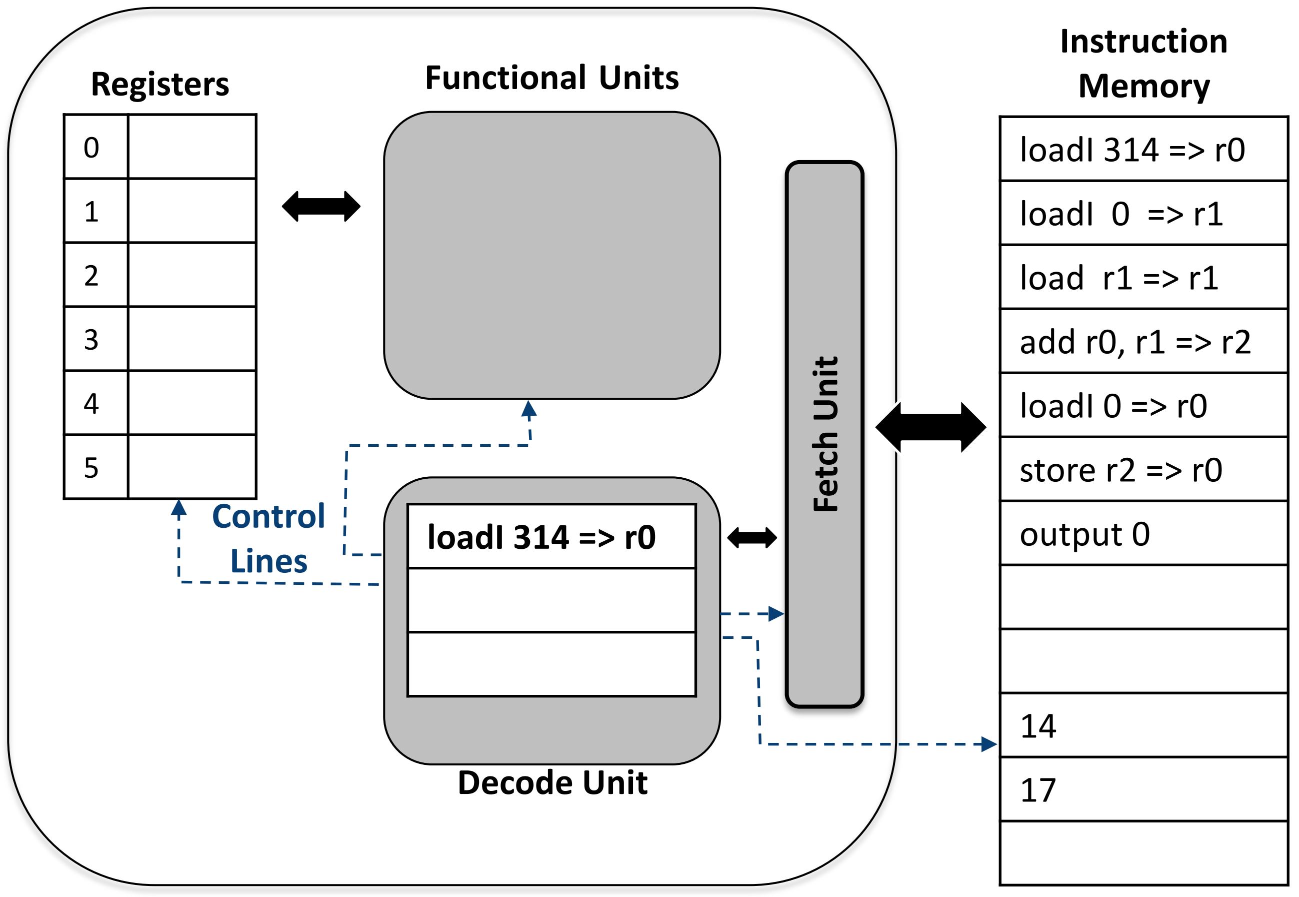
- Real processors have a “fetch, decode, execute” cycle
  - Fetch brings operations into a “buffer” in the decode unit
  - Decode deciphers the bits and sends control signals to the functional unit(s)
  - Execute clocks the functional unit through one pipeline cycle
- “Fetch, decode, execute” is construed as a single cycle
  - In reality, the units run concurrently
- Fetch unit works to deliver “enough” operations to the fetch unit
  - “enough” is defined, roughly, as one op per functional unit per cycle
- Decode unit is, essentially, combinatorial logic (&, therefore, fast)
- Execute unit performs complex operations
  - Multiply and divide are algorithmically complex operations
  - Pipeline units break “long” operations into smaller subtasks

More Realistic Drawing

Separate Fetch-Decode-Execute



What about processors like **core i8** or **ARM**?



One "processing core"

Modern processors typically have unified instruction and data memory.

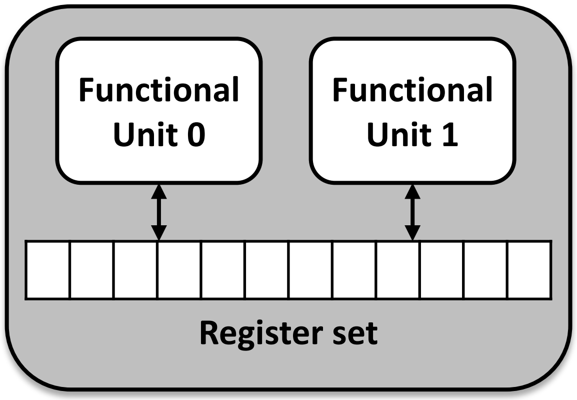
- Operate on a fetch-decode-execute cycle
- Complex, cache-based memory hierarchies
- Multiple pipelined functional units
- Multiple cores

"Modified Harvard Architecture": separate pathways for code and data, but one store

What about processors like **core i7 or ARM**?

Modern processors often have multiple functional units?

- For Lab 1, the **ILOC** simulator has one functional unit
- In Lab 3, the simulator will have two functional units
  - Some operations run on unit 0, some run on unit 1, some run on either unit 0 or unit 1
- The basic model is the same
  - Fetch then execute
  - Number of operations executed in a single cycle depends on the order in which they are encountered *and* the dependences between operations



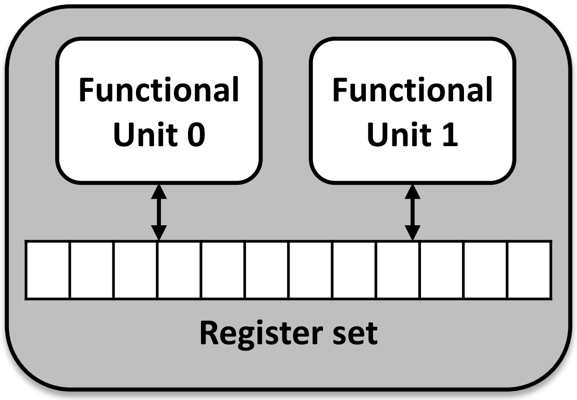
The Lab 3 documentation addresses these issues for **ILOC**

The Lab 3 simulator trace shows action in both units

What about processors like **core i7 or ARM**?

What happens to the execution model with multiple functional units?

- One operation executes on each functional unit
- The complication arises in the processor's fetch and decode units
  - Fetch unit must be retrieve several operations
  - Fetch & decode must collaborate to decide where they execute
    - → Fixed, position-based scheme leads to **VLIW** system
    - → Dynamic scheme leads to superscalar systems
  - More complex decode unit costs more transistors and more power



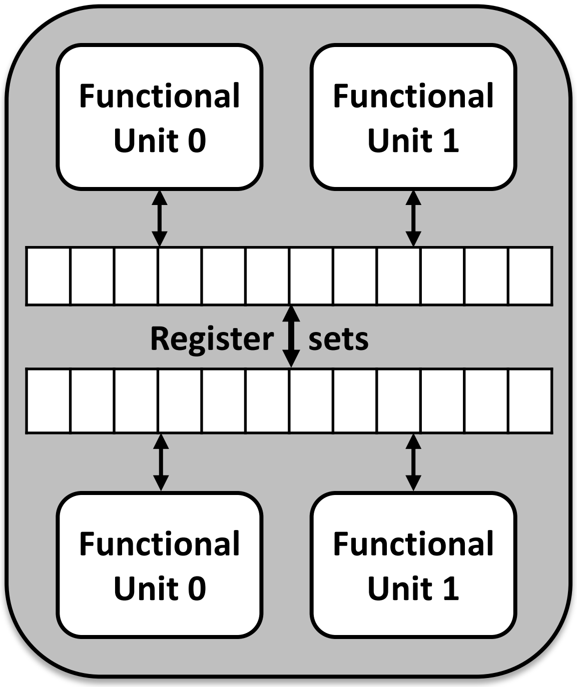
- Processors with multiple functional units need code with multiple independent (unrelated) operations in each cycle
  - *Instruction Level Parallelism* (*or* ***ILP***)
  - *See Lab 3 in **COMP 412***

VLIW is *Very Long Instruction Word* computer

What about processors like **core i7 or ARM**?

When the number of functional units gets large …

- At some point, the network to connect register sets to functional units gets too deep
  - Transmission time through the multiplexor can come to dominate processor cycle time
  - More functional units would slow down the processor's fundamental clock speed
- Architects have adopted "partitioned register" designs that have multiple register sets with limited bandwidth between the register sets
  - Adds a new problem to code generation: the placement of operands
  - Need to place each operation on a functional unit that can access the data
    - → *Or, need to insert code to transfer the data (& ensure that a register is available for it in the new register set)*



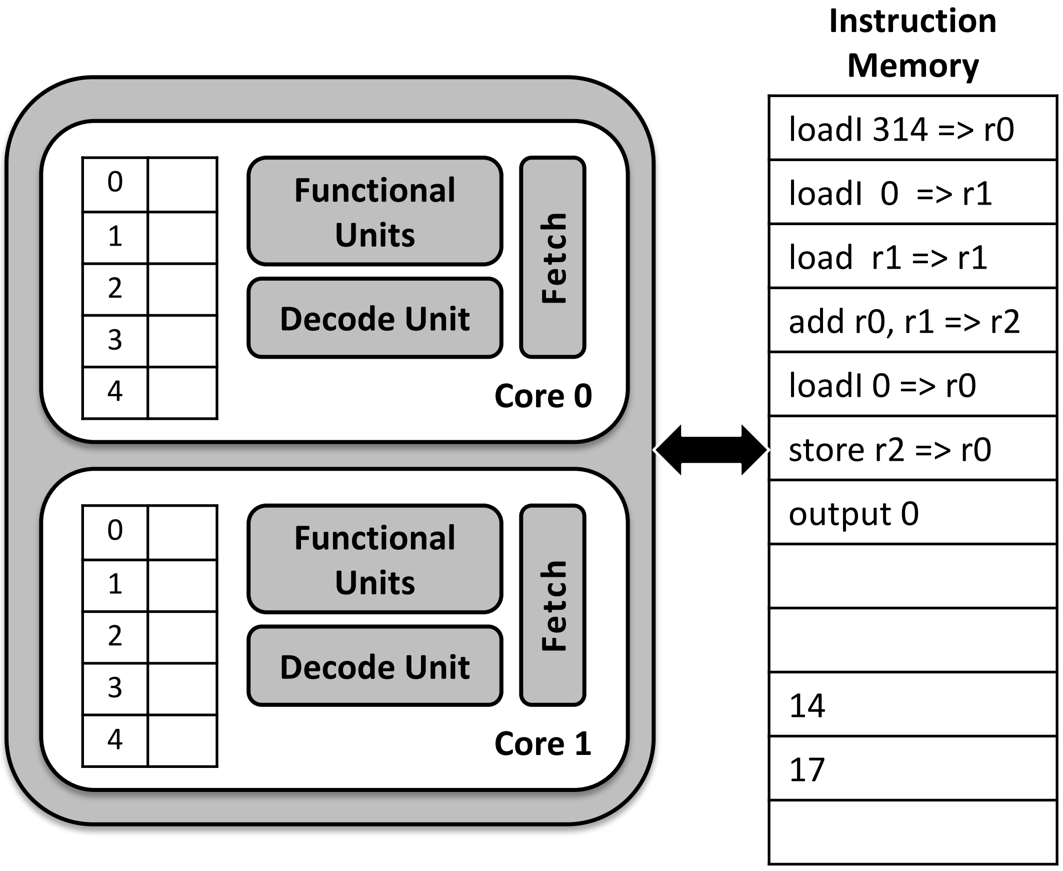
And the fetch and decode units get even more complex ..

What’s Next After Multiple Functional Units?

As processor complexity grows, the yield on performance for a given expenditure of chip real estate (or power) shrinks

- A core with eight functional units might be bigger than four cores with two functional units
  - The interconnects between fetch, decode, register sets, (caches,) and functional units become even more complex
- At some point, it is easier to put more cores on a chip than bigger cores
  - Stamp out more simpler cores rather than fewer complex cores
  - Easier design problem
  - Lower power consumption
  - Better ratio of performance to chip area (and power)
- A great idea, if the programmer, language, and compiler can find
  - Enough thread-level parallelism to keep all the cores busy
  - Enough instruction-level parallelism (within each thread) to keep the functional units busy

What About Multiple Cores?



Modern multicore processors have 2 to many (4, 8, 32) cores

- Require lots of parallelism for best performance
- Major limitation is memory bandwidth
  - 1 / (# cores) ?
  - Bandwidth may impose some practical limits on the use of all those cores

What’s Next After Multiple Functional Units?

What happens to the execution model in a multicore processor?

- Execution within a thread follows the single core model
  - Fetch, decode, & execute with (possibly) multiple functional units
  - Single threads have simple behavior
- Individual threads operate independently
  - Language (& processor) usually provide synchronization between threads
  - Need synchronization to share data and communicate control
- See **COMP 322** and **COMP 515**