

CS-200

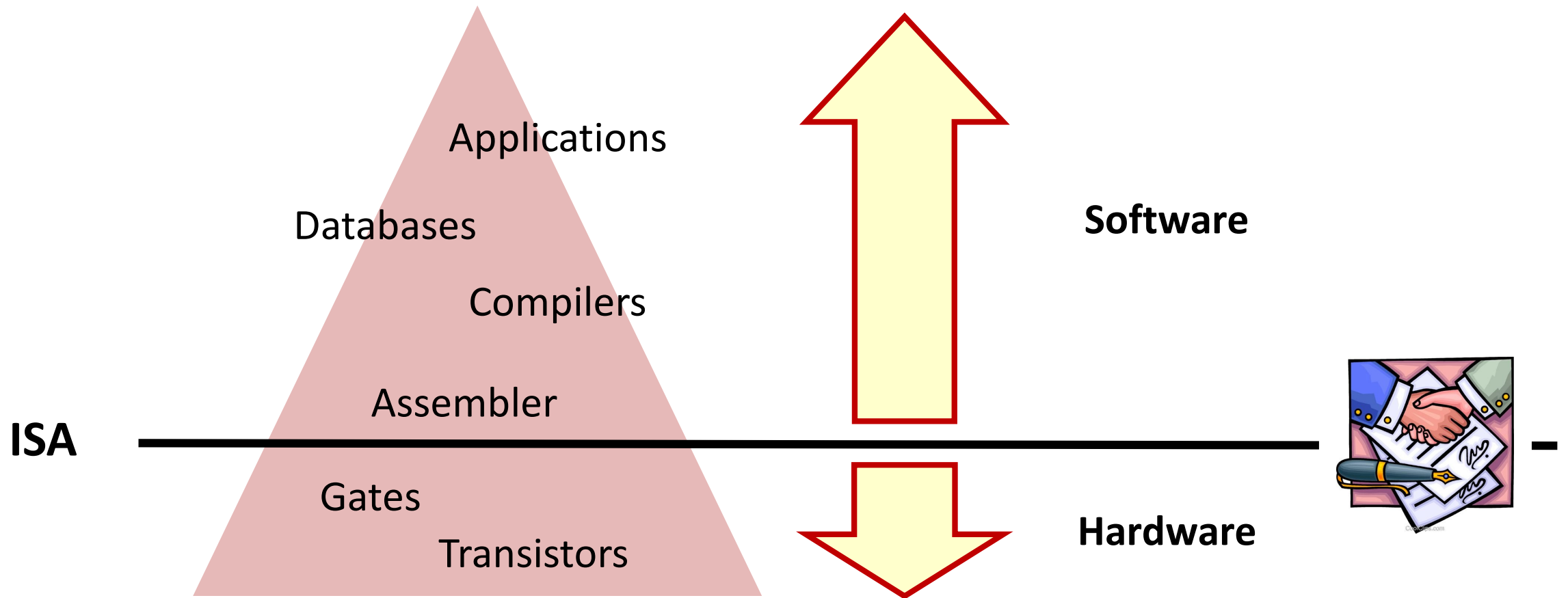
Computer Architecture

Part Ic: Instruction Set Architecture

Memory and Addressing Modes

Paolo Ienne
<paolo.ienne@epfl.ch>

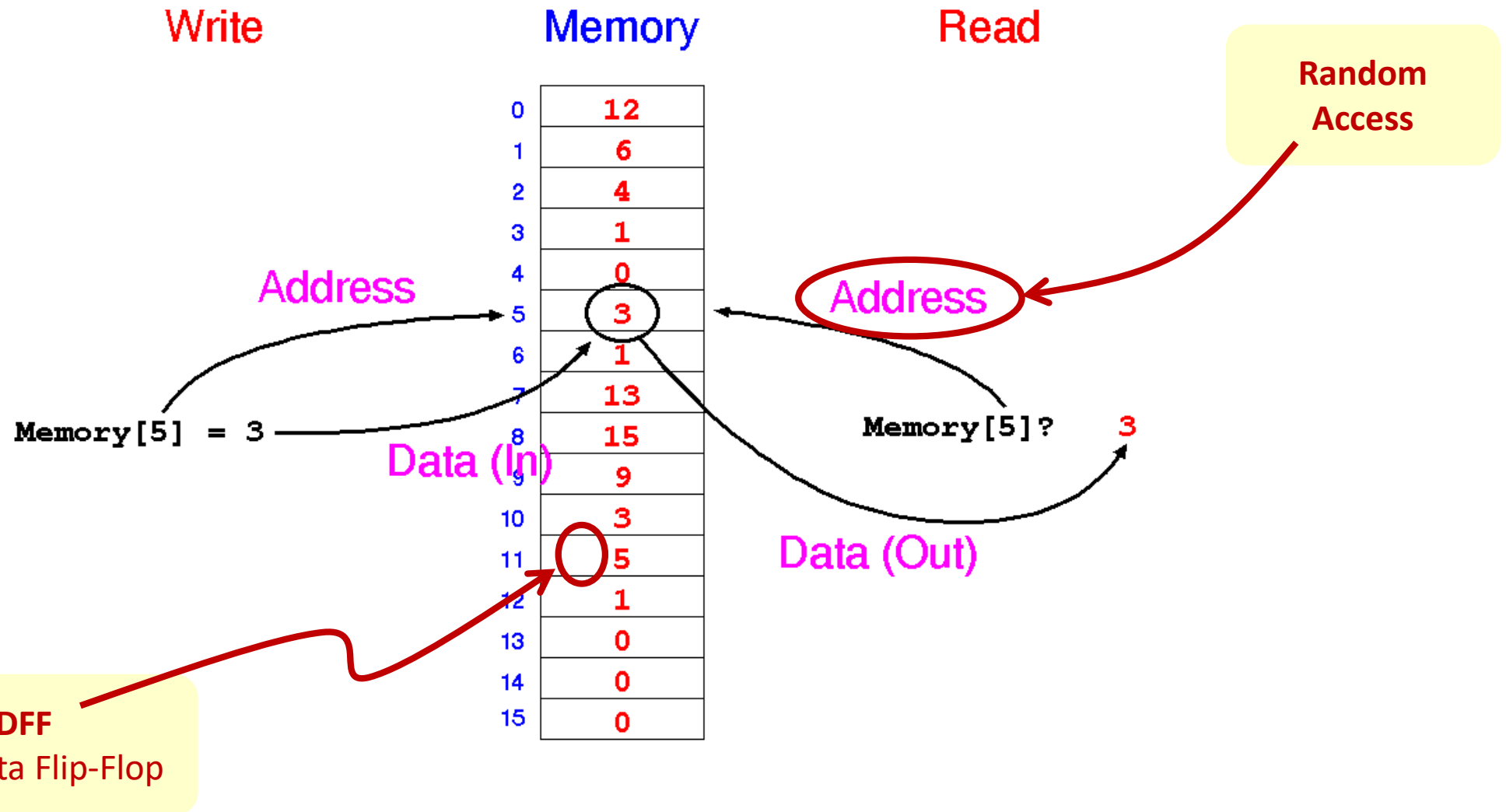
The Contract between HW and SW



Memory

- An incredibly important component of a computing system
 - We store our **programs** in it
 - We store our **data** in it
 - It is often through memory that we will **receive and send out data**
- Memory is a recurrent topic in this course
 - Memory can be **very slow** → Caches
 - Memory is **“finite”** (= relatively small) → Virtual Memory
 - Memory can make an **ISA too complex** → Pipelining

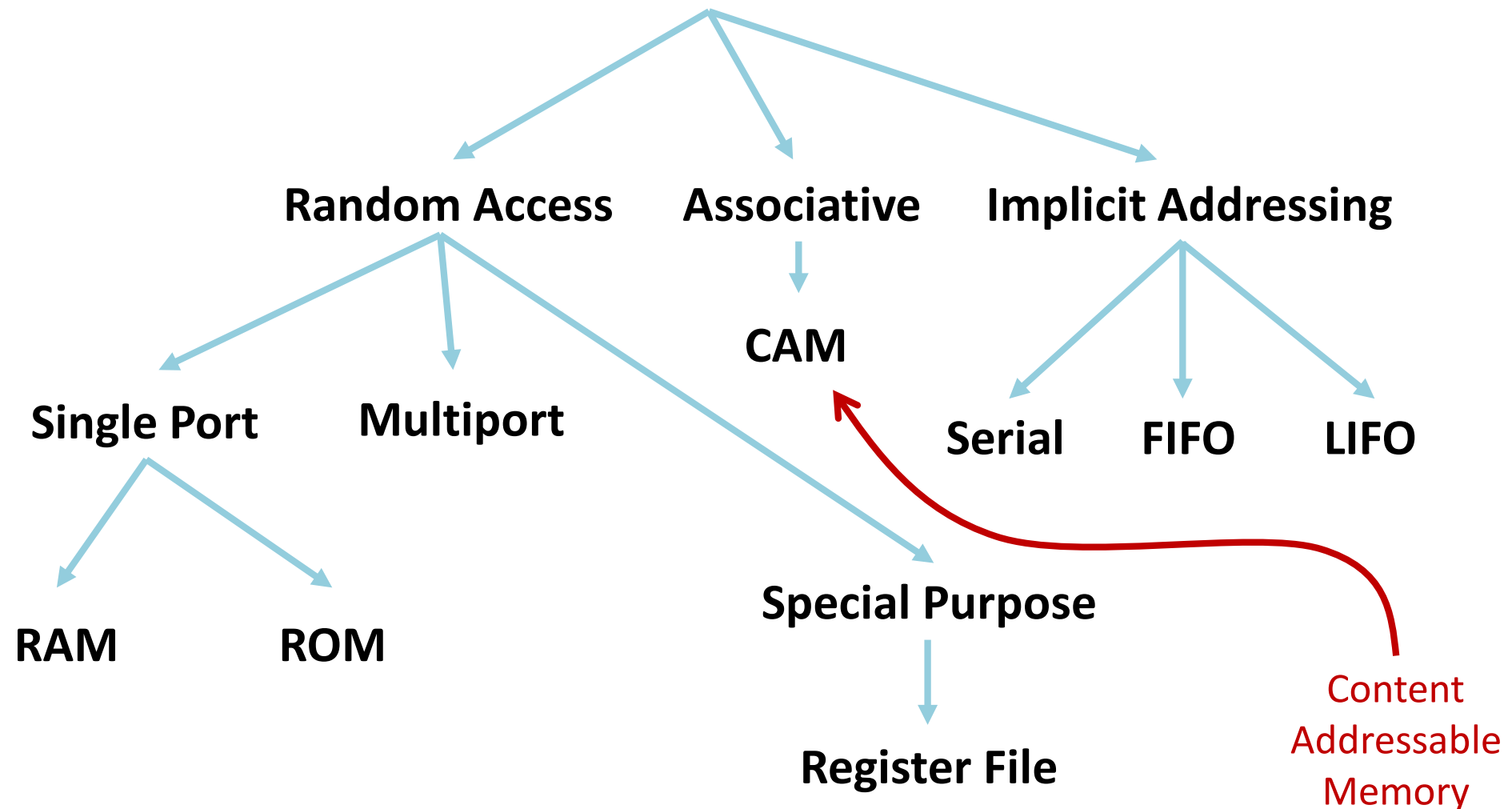
Address and Data



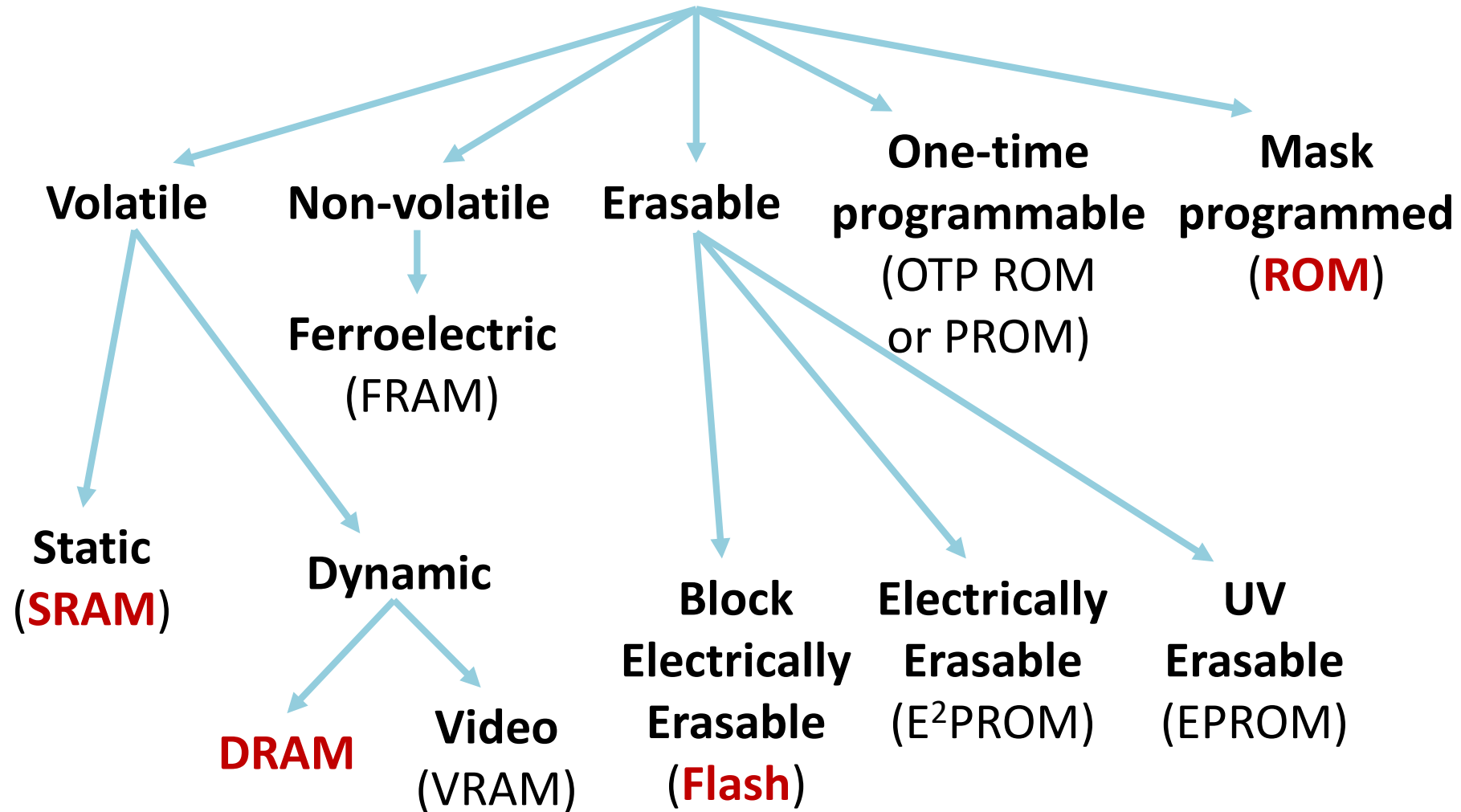
Many Types of Memories

- Different **technologies**
 - SRAM, DRAM, EPROM, Flash, etc.
- Large variations in **capabilities**
 - Capacity, density
 - Speed
 - Writable, permanent, reprogrammable
- Available as discrete devices (all) and as embedded ASIC components (many, increasingly)

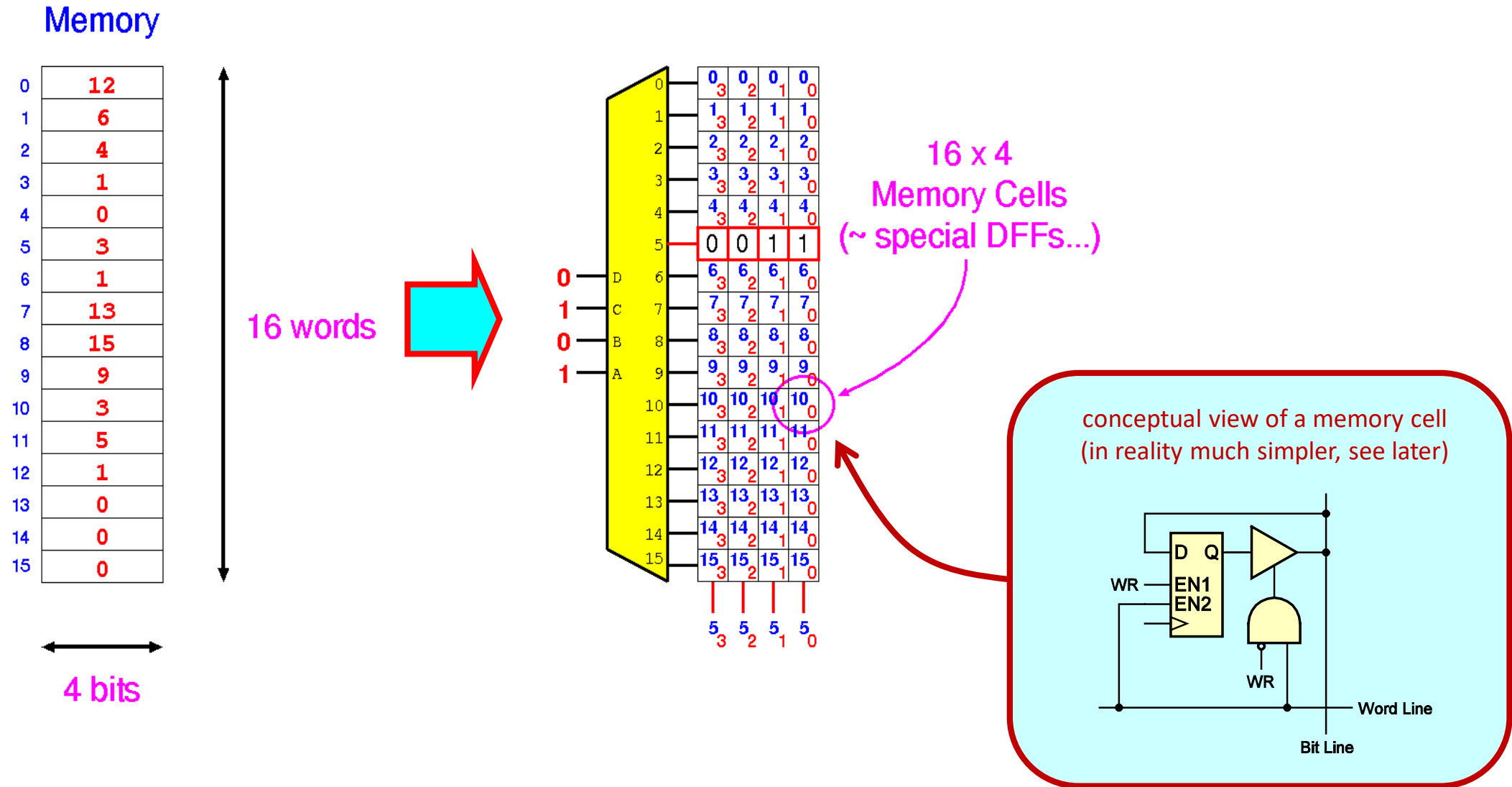
Functional Taxonomy of Memories



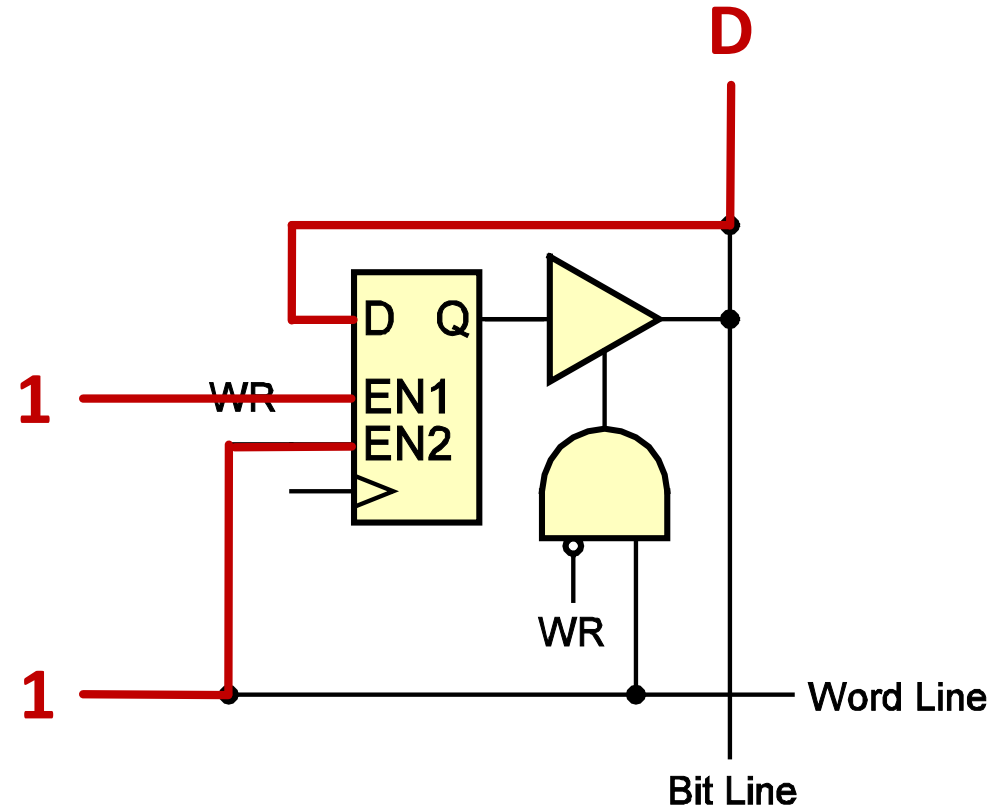
Taxonomy of Random Access Memories



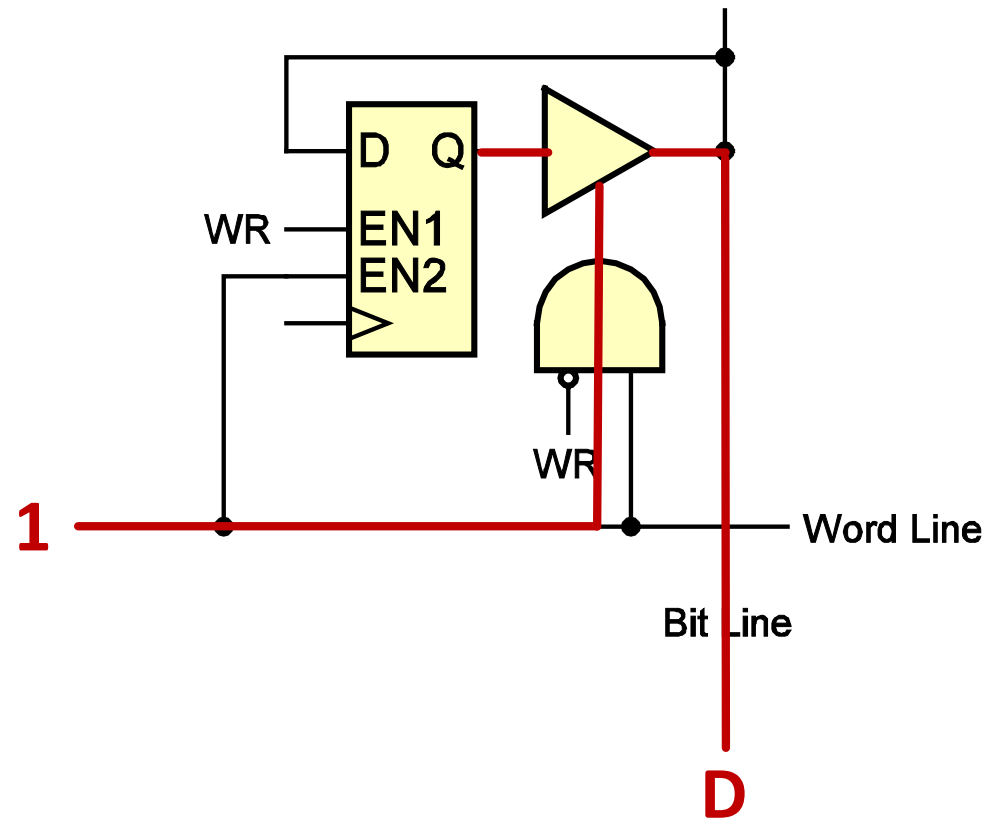
Basic Structure



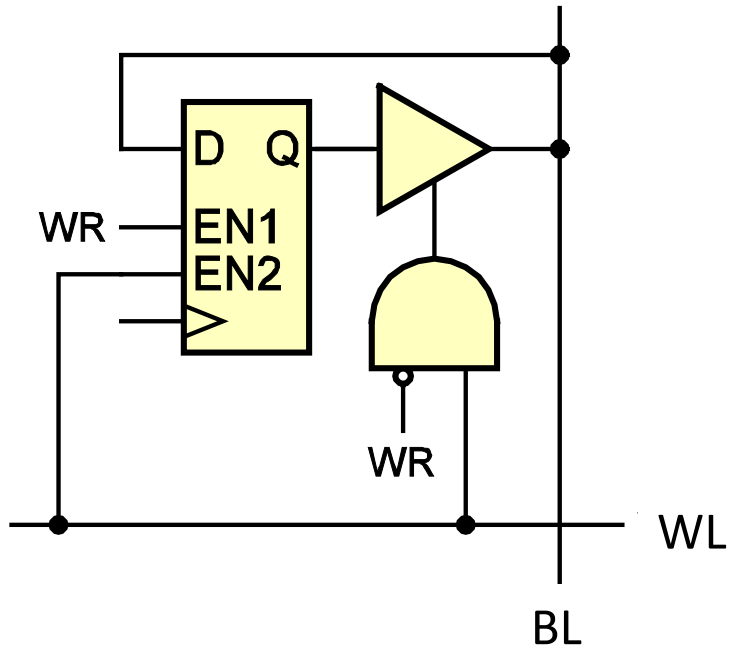
Write



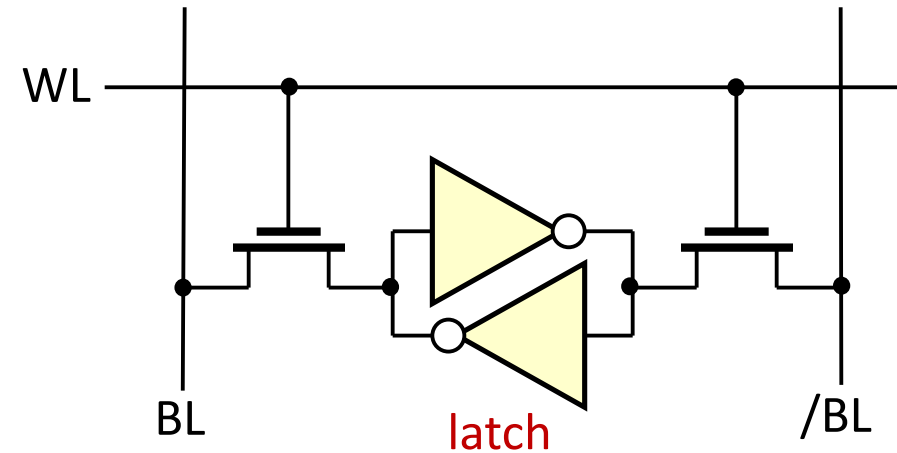
Read



Practical SRAMs



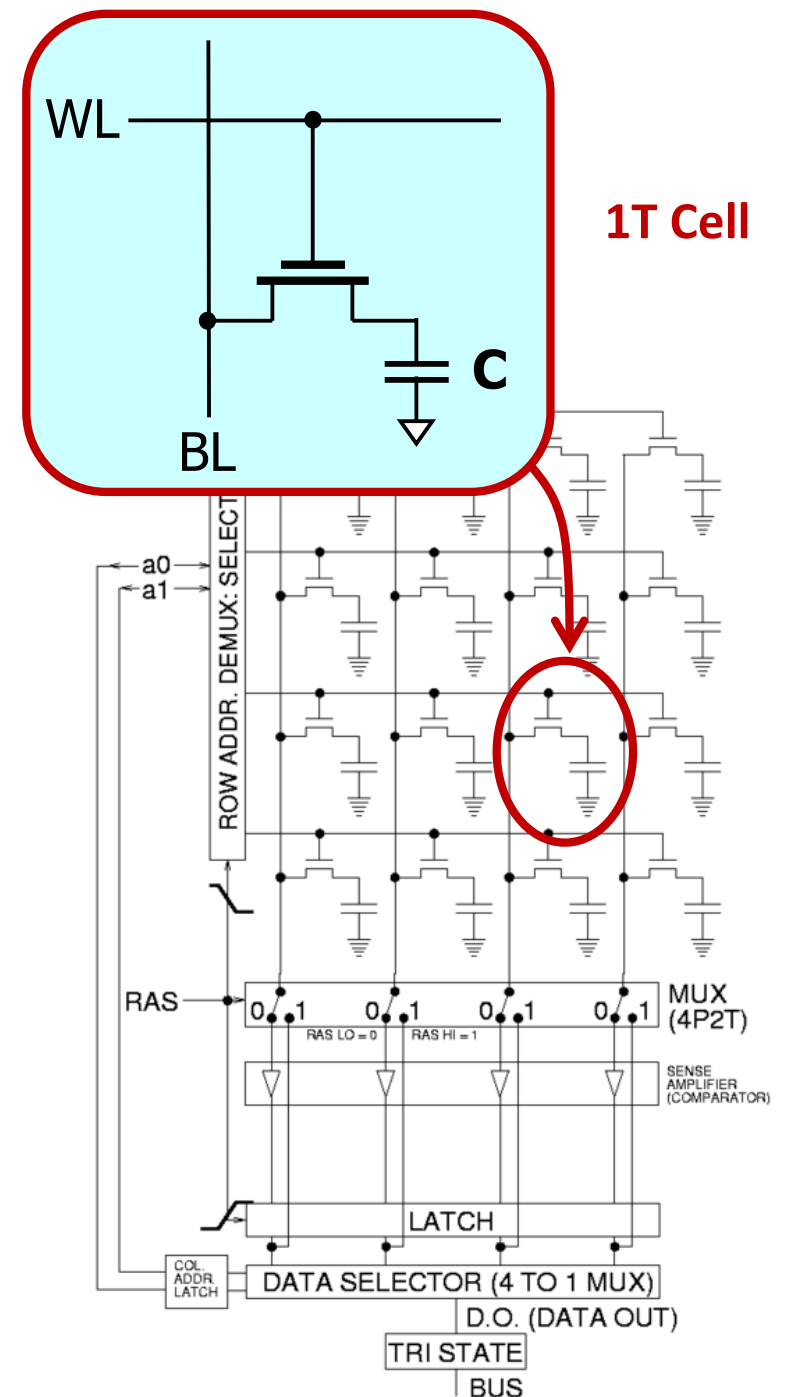
Small, very fast memories
(e.g., maybe Register Files)



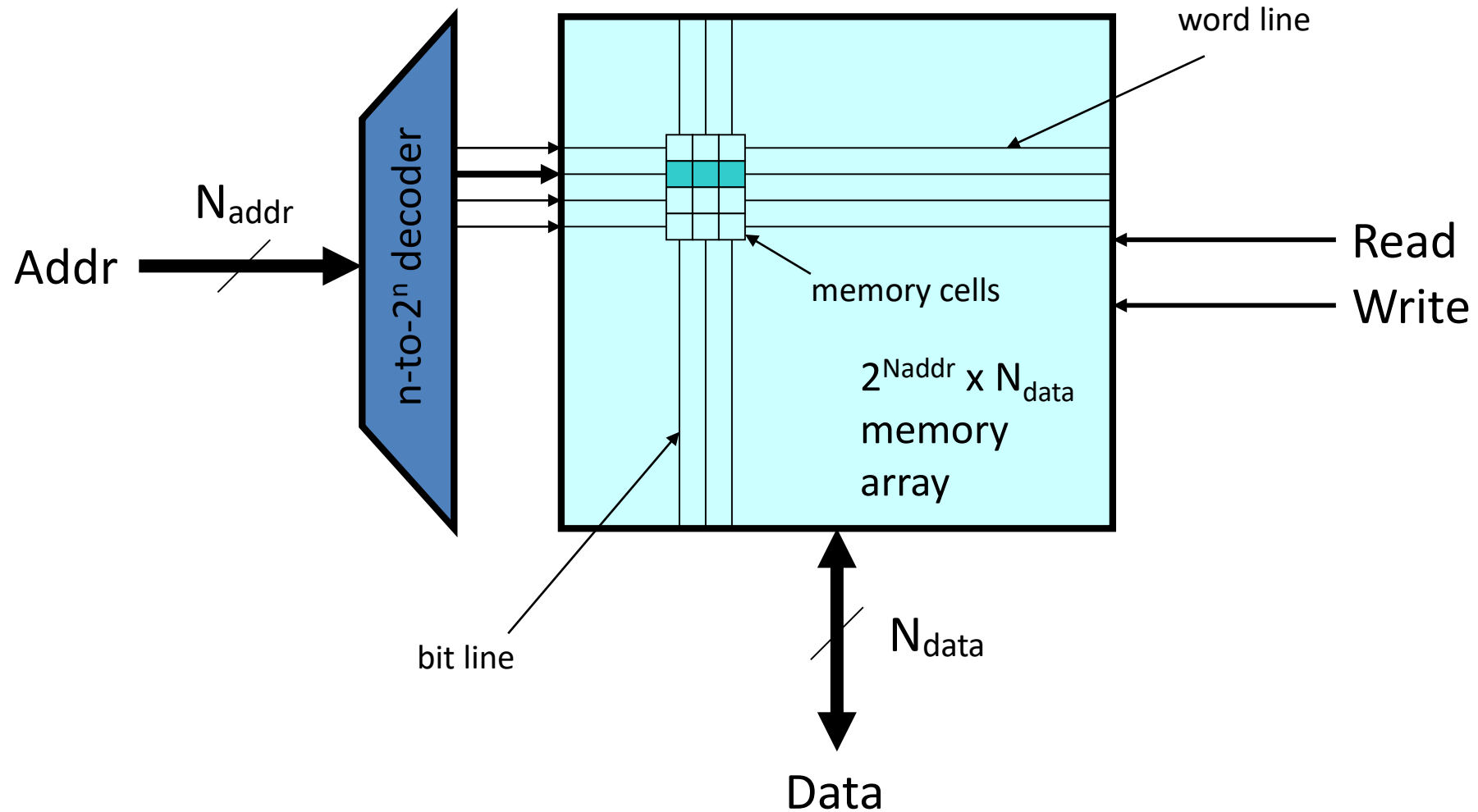
All practical SRAM use a **6T cell**

DRAM

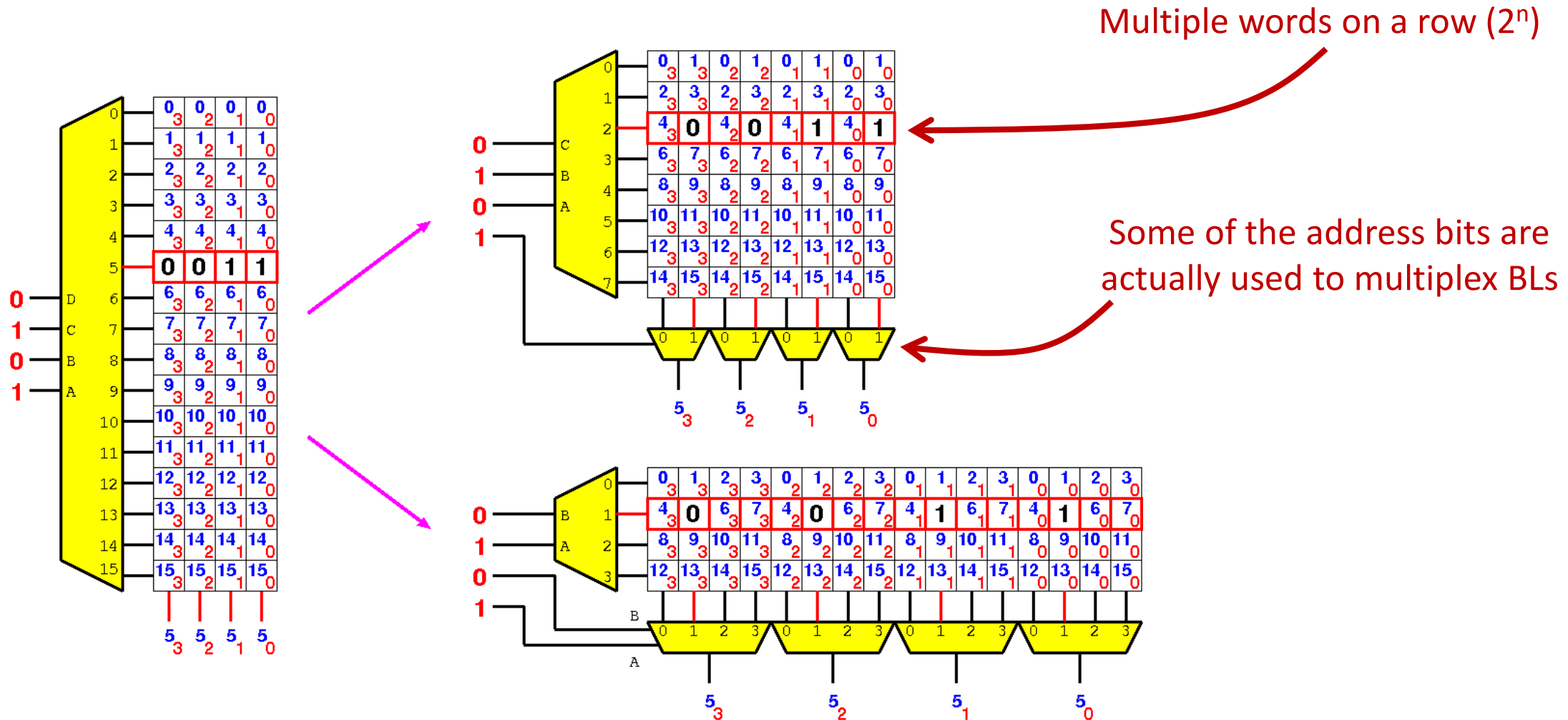
- Dynamic RAMs are the densest (and thus cheapest) form of random-access semiconductor memory
- DRAMs store **information as charge in small capacitors** part of the memory cell
- First patented in 1968 by Robert Dennard, scaled amazingly over decades and was somehow an important ingredient of the progress of computing systems
- Charge **leaks off** the capacitor due to parasitic resistances → every DRAM cell needs a **periodic refresh** (e.g., every ~60 ms) lest it forgets information!



Ideal Random Access Memory Array

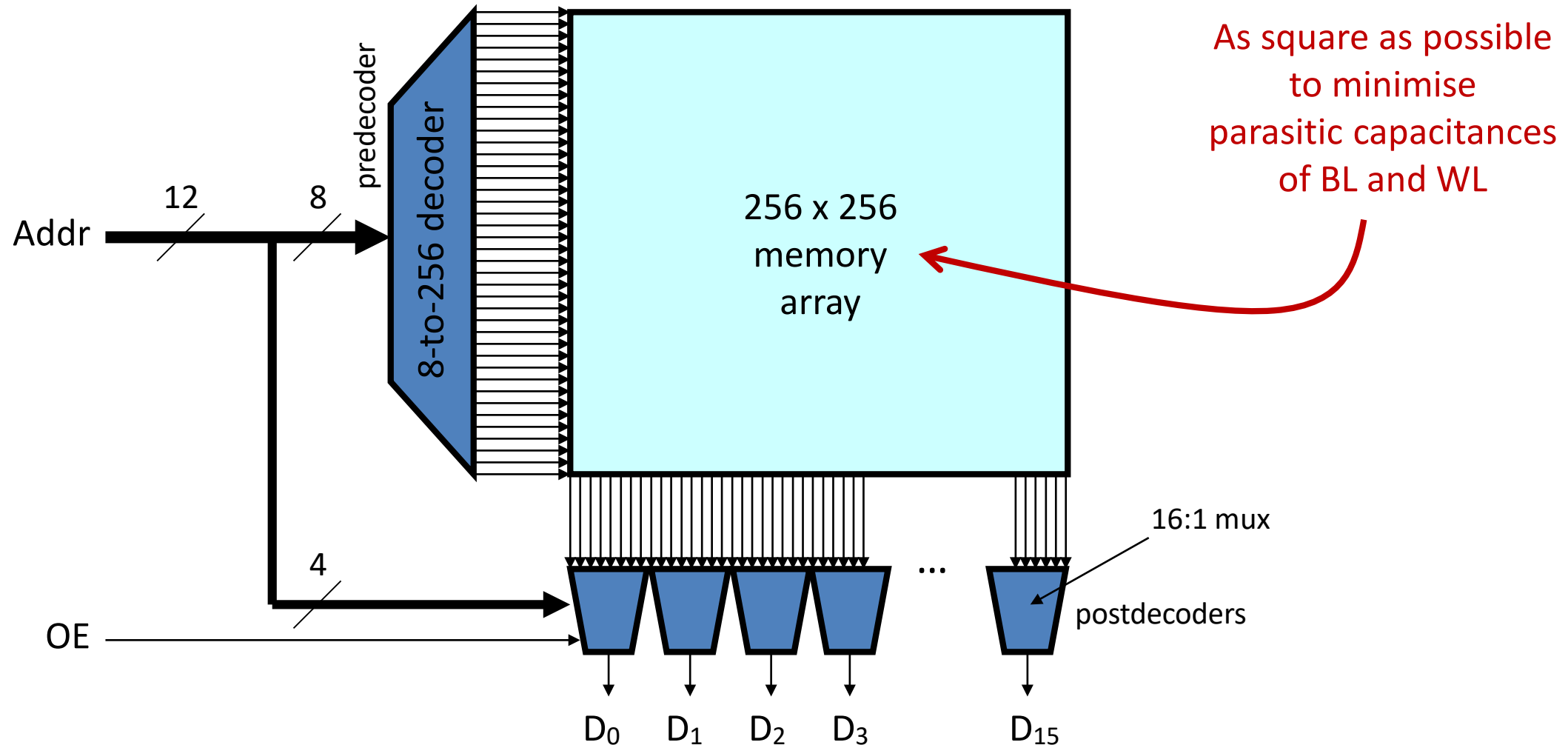


Physical Organisation Can Be Different

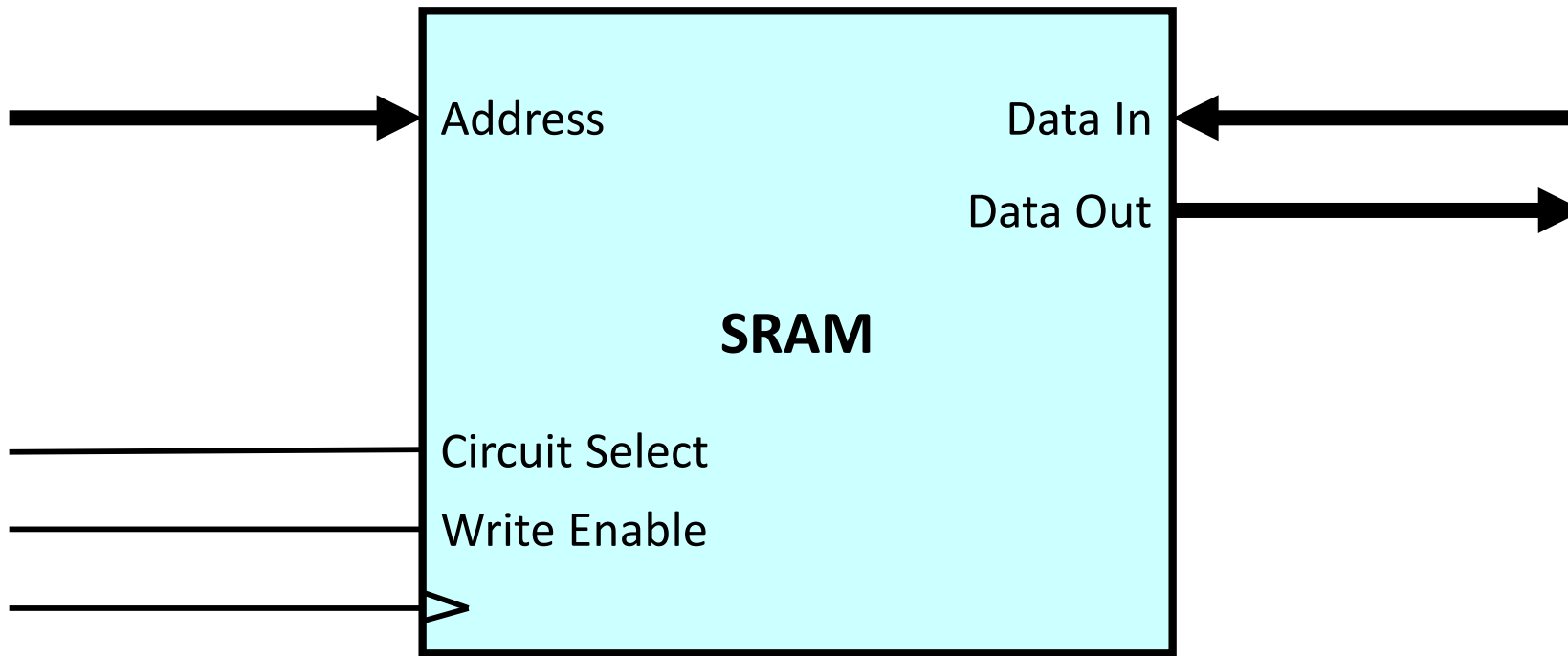


More Realistic ROM Array

E.g., 4K x 16 (= 64 Kbit = 256^2 bit)

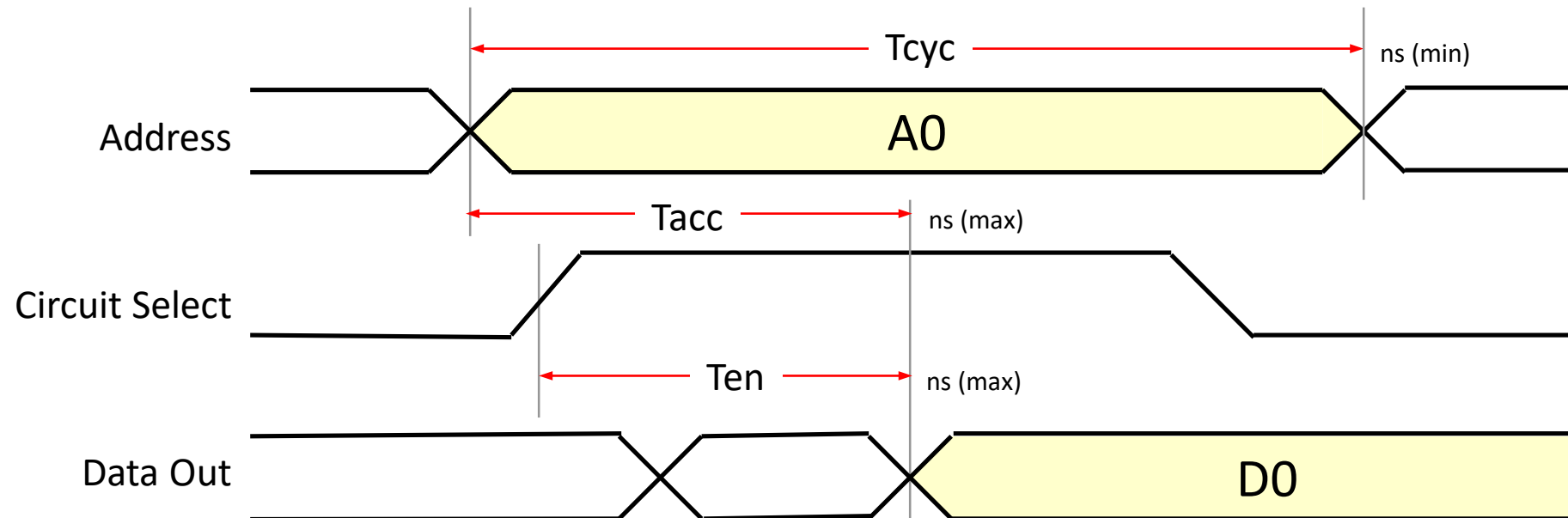


Static RAM Typical Interface

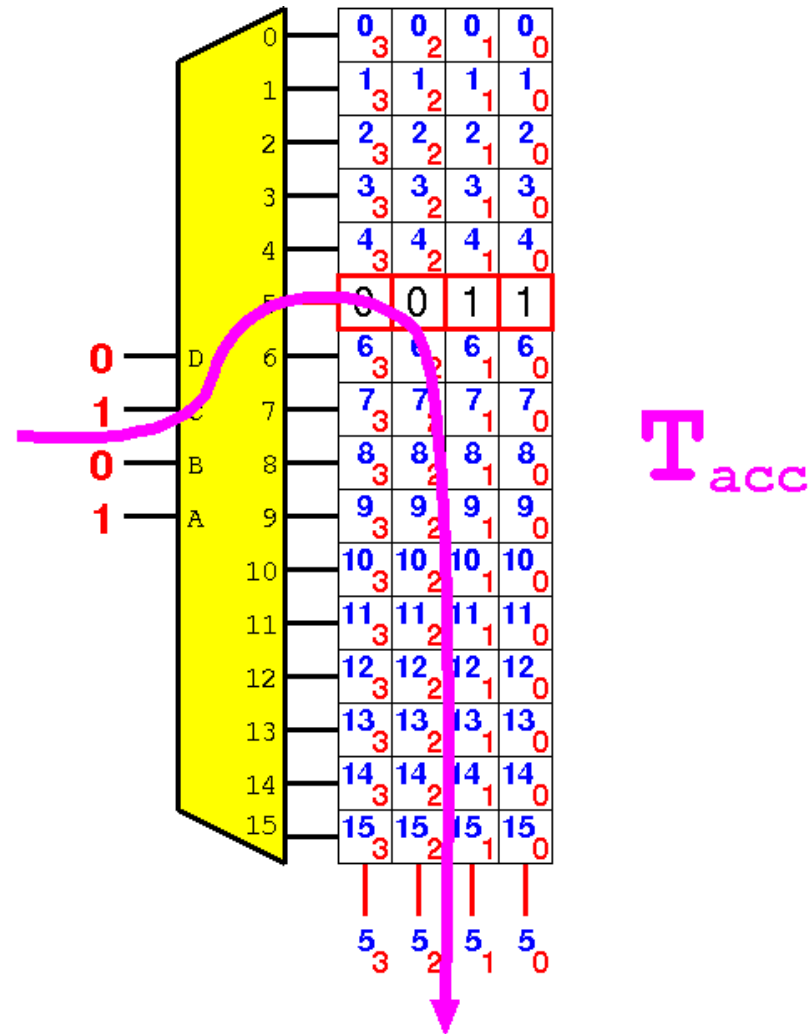


Typical Asynchronous SRAM *Read* Cycle

- Enable the memory, assert the address, and wait for the data
 - Data Out available after a combinational delay T_{acc} = Access Time
- Maximum frequency limited by minimum T_{cyc}

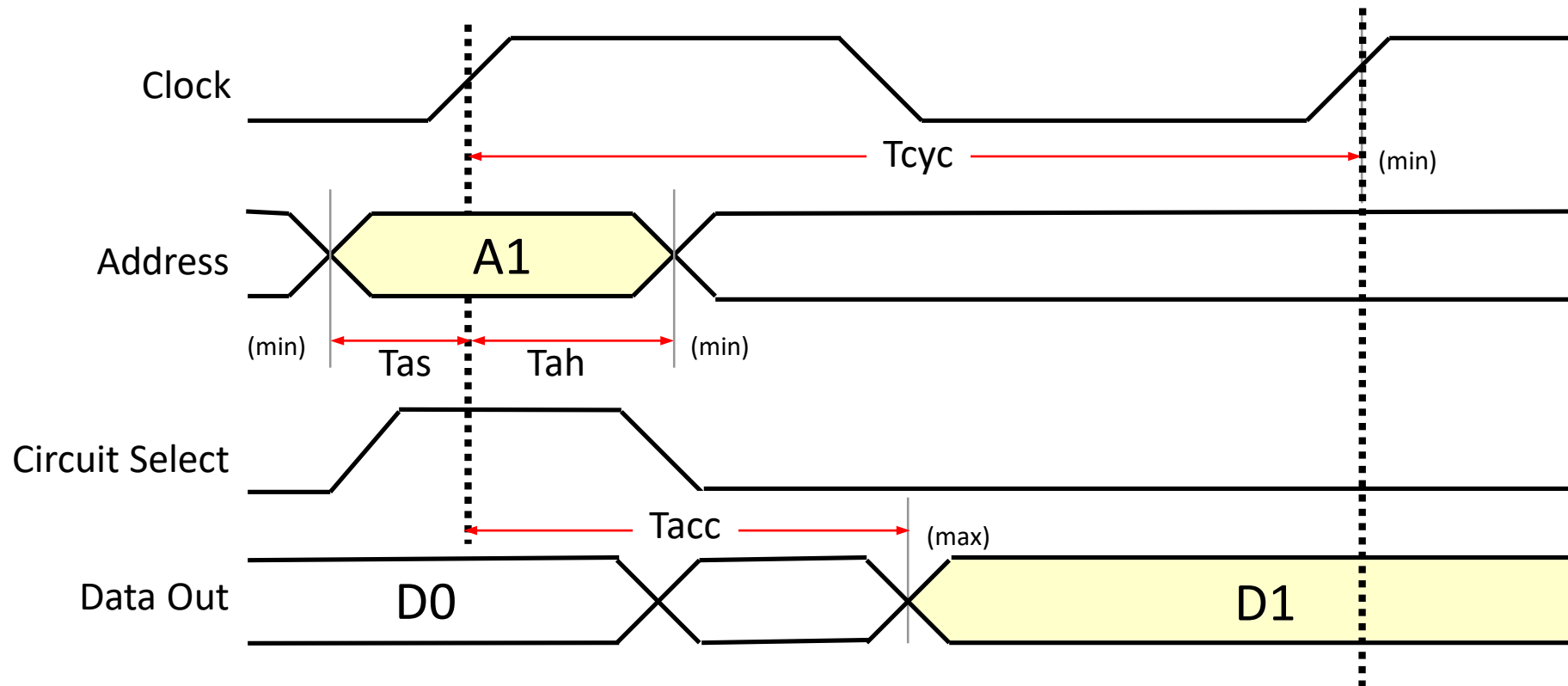


Propagation Delay in the Physical Components



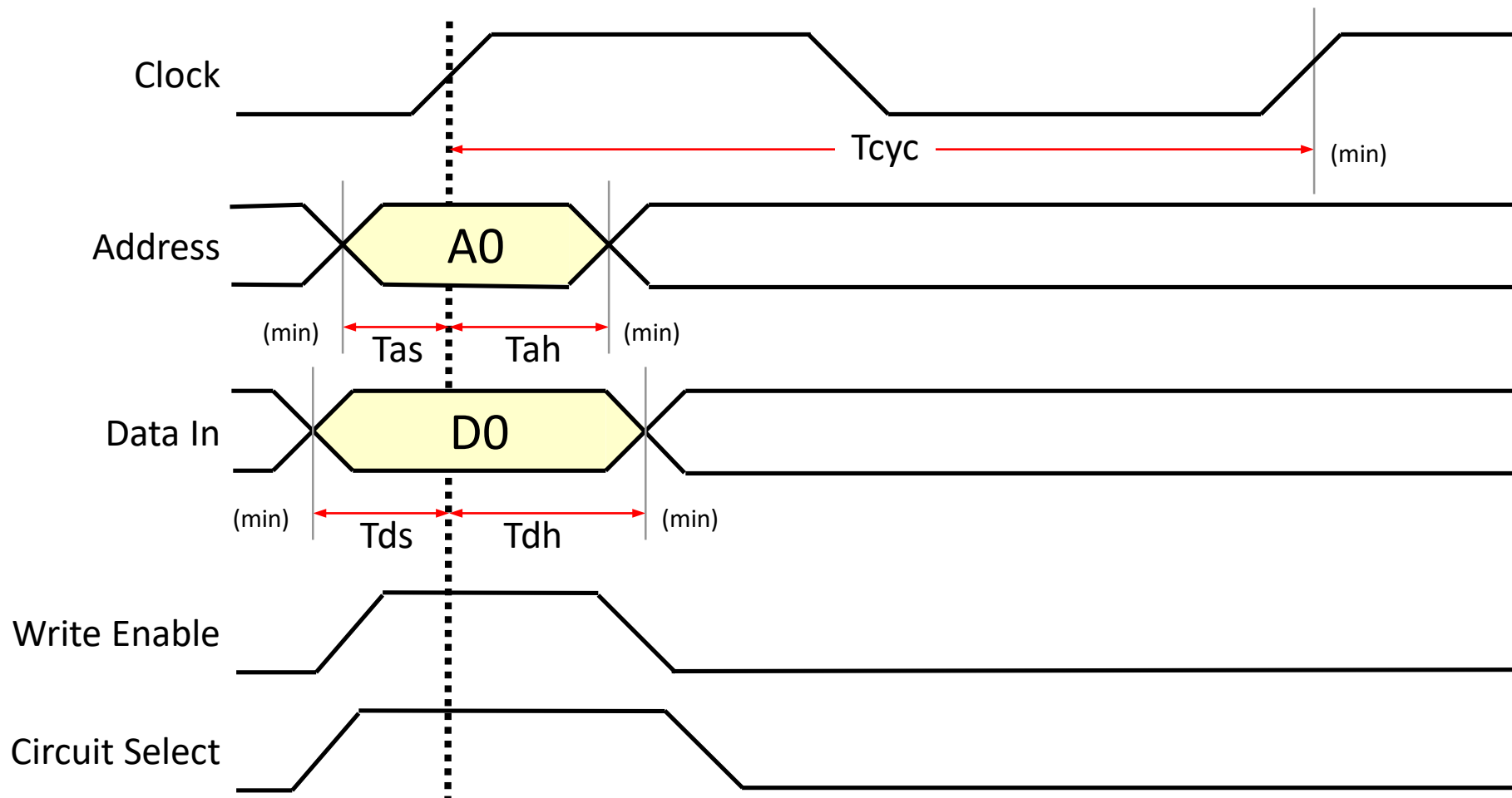
Typical Synchronous SRAM *Read* Cycle

- Everything relative to the clock signal
- Latency is the number of cycles between the address asserted and data available
 - Often one as in this diagram but in some cases (large memories) more

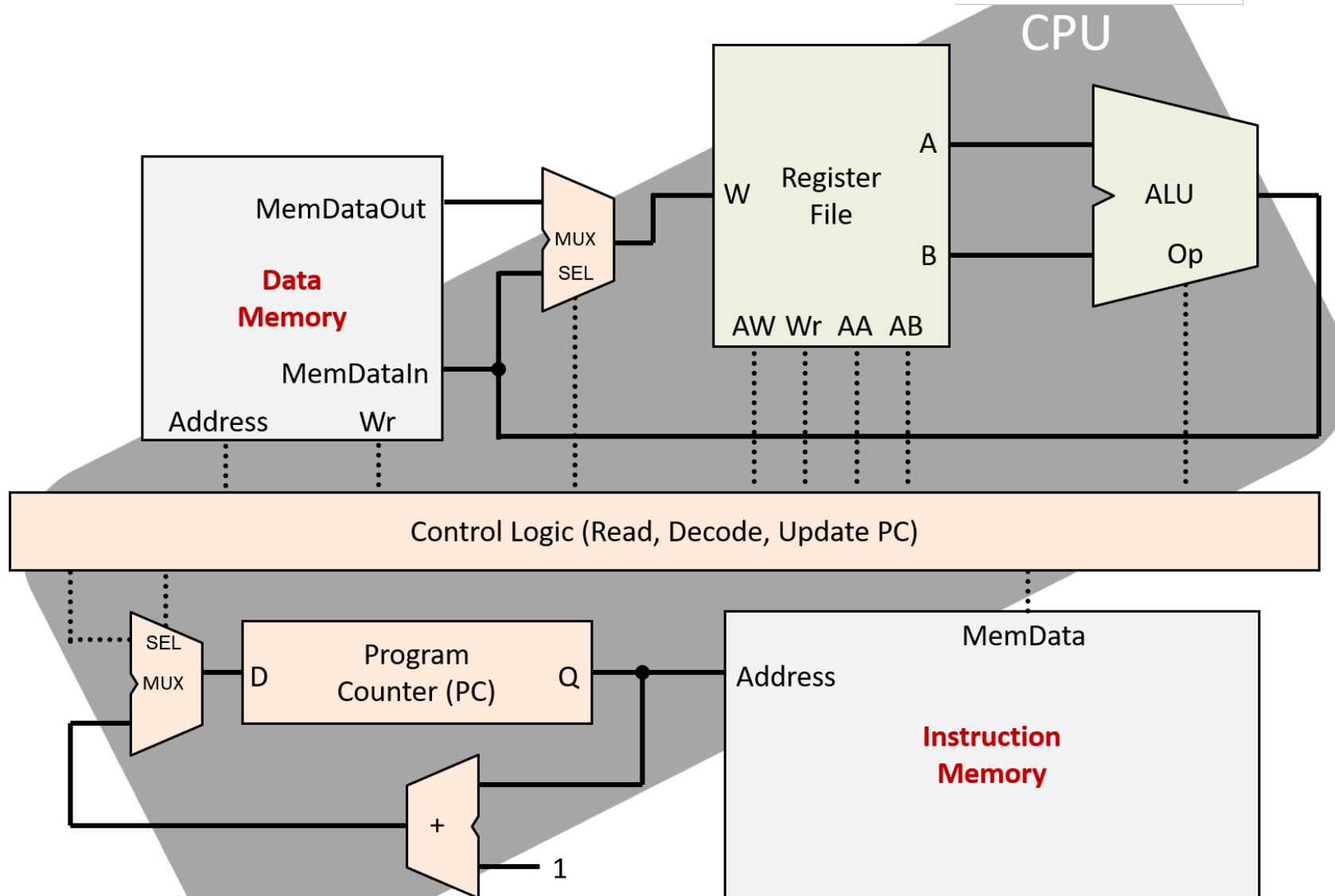


Typical Synchronous SRAM Write Cycle

- Writes on the edge of the clock signal, as a DFF

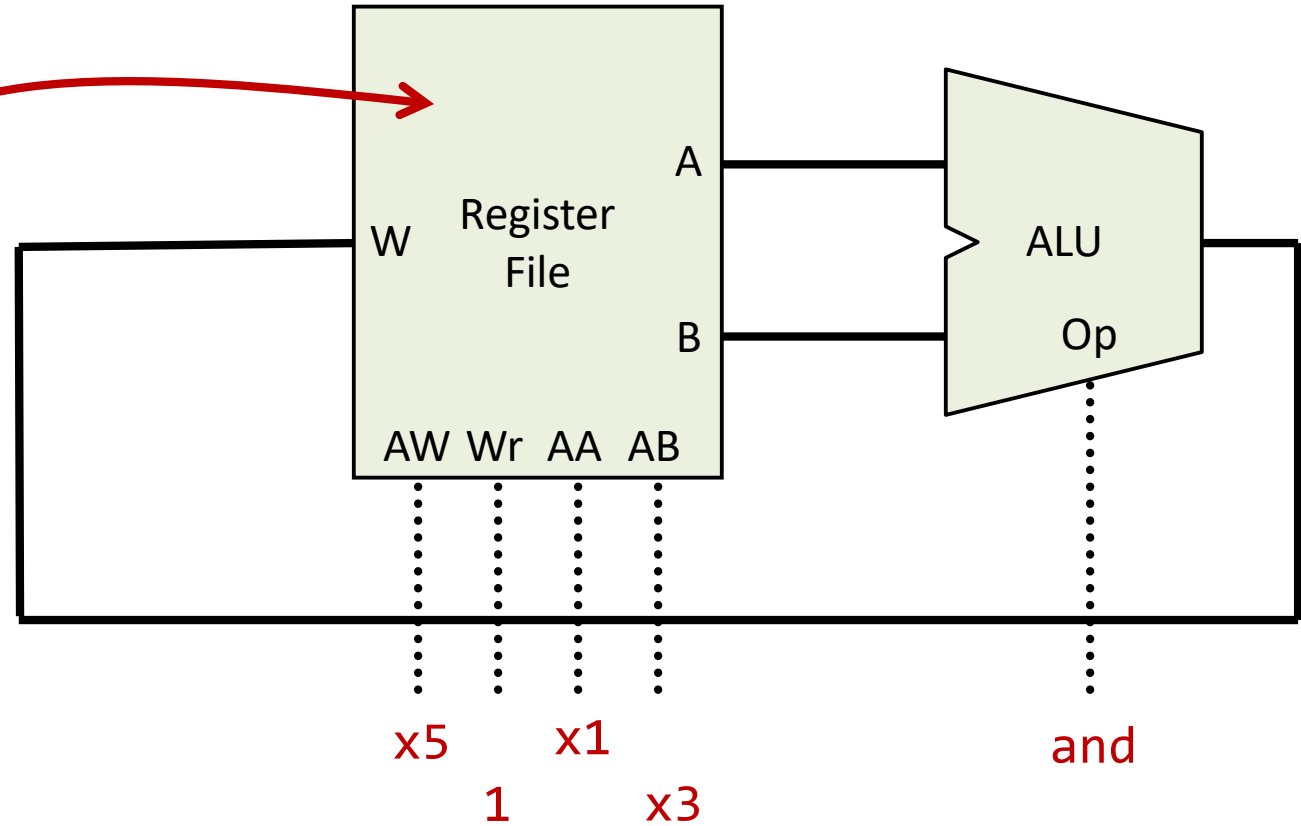


Where is Memory in the Processor?



Arithmetic and Logic Instructions

Too few “variables” for
serious computation



How to bring data
in and out?!

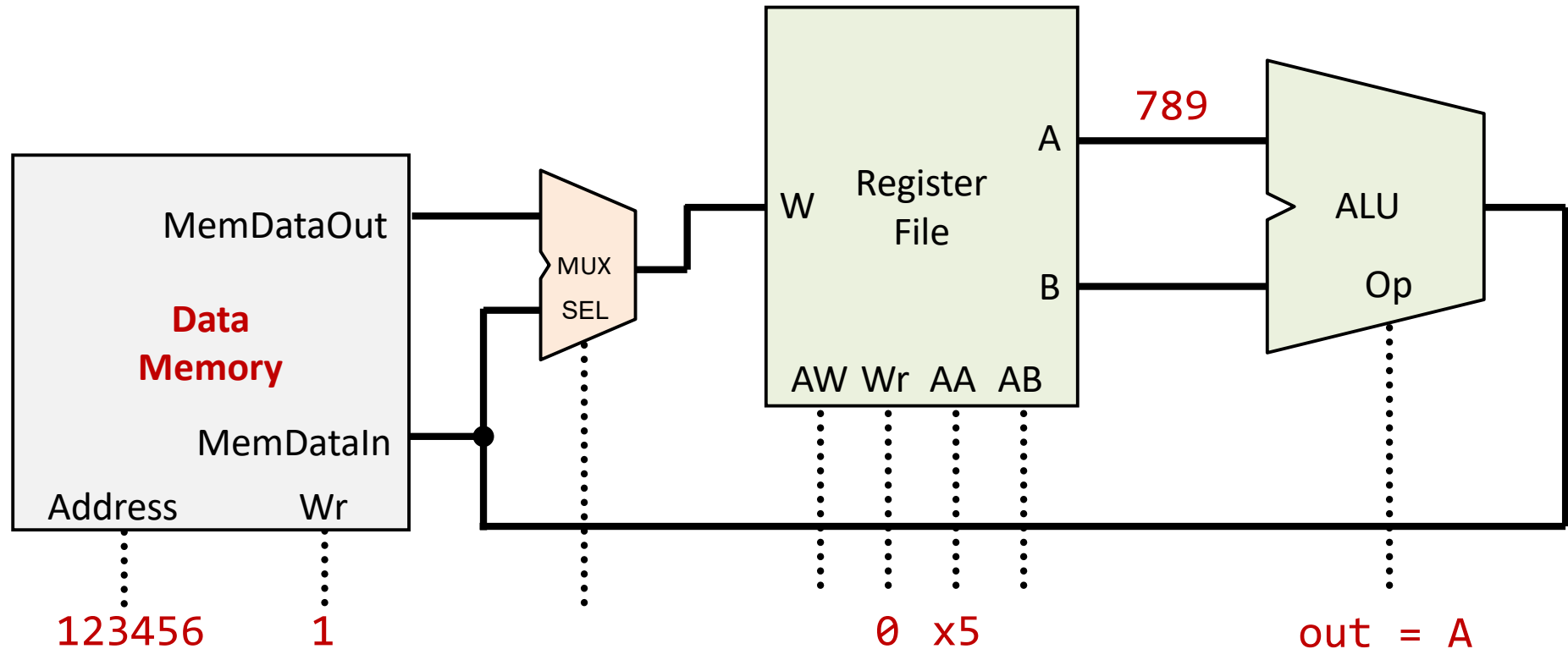
and x5, x1, x3

23



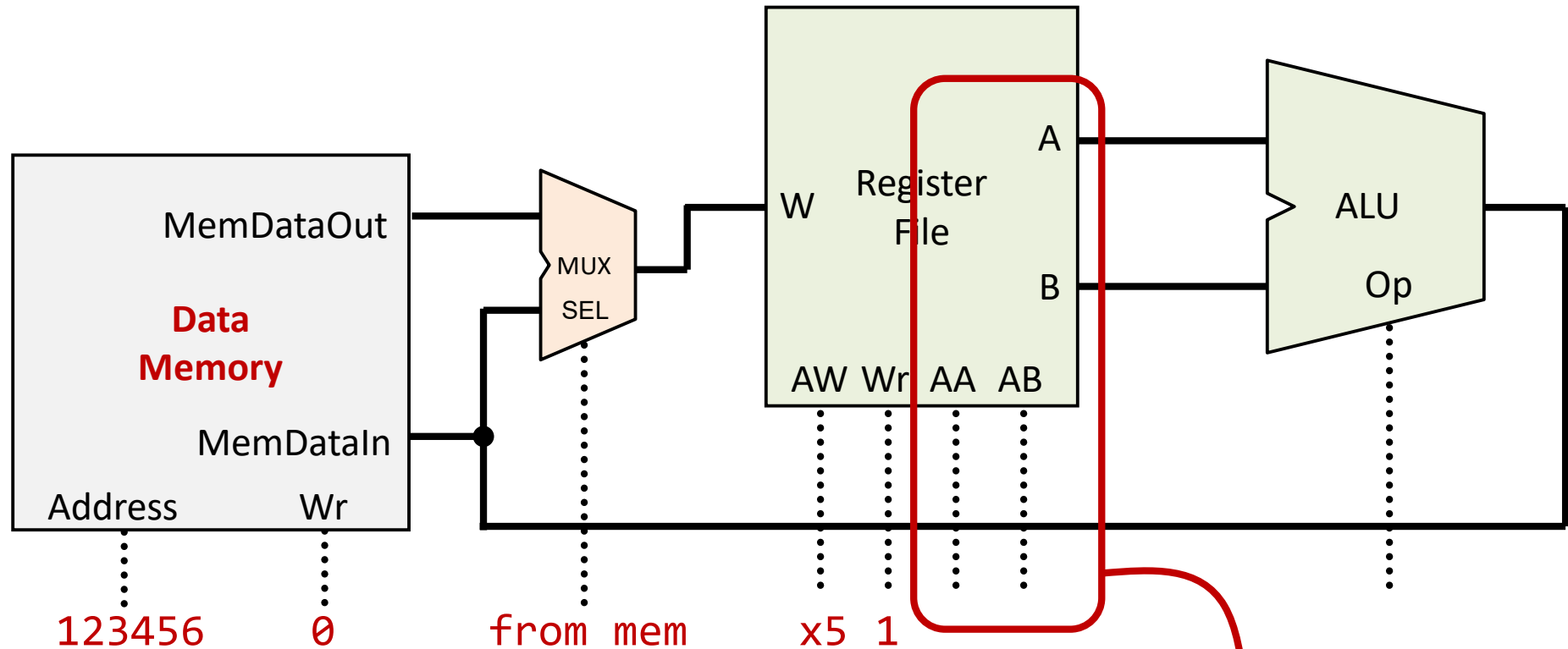
lw x5, (123456)

Store Instructions



sw x5, (123456)

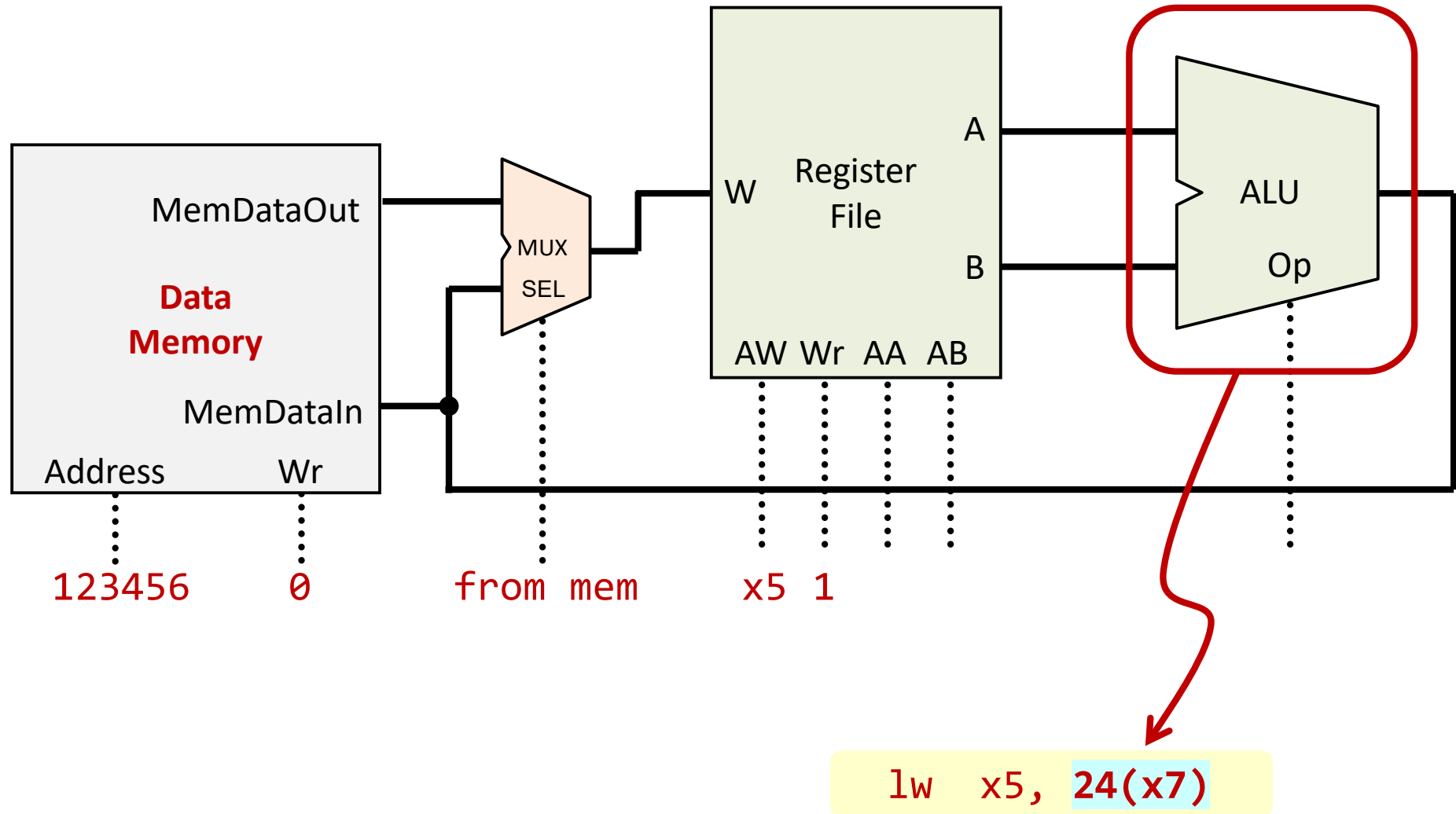
Loads and Store: The RISC-V Way



The address is too big as an **immediate** value!

`lw x5, (x7)`

Loads and Store: The RISC-V Way



A Load/Store Architecture

- Instructions reading and writing in memory do **just that** and **nothing else**
- It is a typical feature of **Reduced Instruction Set Computer (RISC)** processors, whose advantages will become clear later in CS-200

Load					
lw	rd,imm(rs1)	$rd \leftarrow \text{mem}[\text{rs1} + \text{sxt}(\text{imm})]$	I	0x2	0x03
Store					
sw	rs2,imm(rs1)	$\text{mem}[\text{rs1} + \text{sxt}(\text{imm})] \leftarrow \text{rs2}$	S	0x2	0x23

More Addressing Modes? Not in RISC-V!

- Register

`add x0, x1, x2`

→ `x0 = x1 + x2;`

- Immediate

`add x0, x1, 123`

→ `x0 = x1 + 123;`

- Direct or Absolute

`add x0, x1, (1234)`

→ `x0 = x1 + mem[1234];`

- Register Indirect

`add x0, x1, (x2)`

→ `x0 = x1 + mem[x2];`

- Displacement or Relative

`add x0, x1, 123(x2)`

→ `x0 = x1 + mem[x2 + 123];`

Syntax here looks like RISC-V but most of these instructions do not exist in RISC-V

More Addressing Modes? Not in RISC-V!

- Base or Indexed

`add x0, x1, i5(x2)`

→ `x0 = x1 + mem[x2 + i5];`

- Auto-increment or -decrement

`add x0, x1, (x2+)`

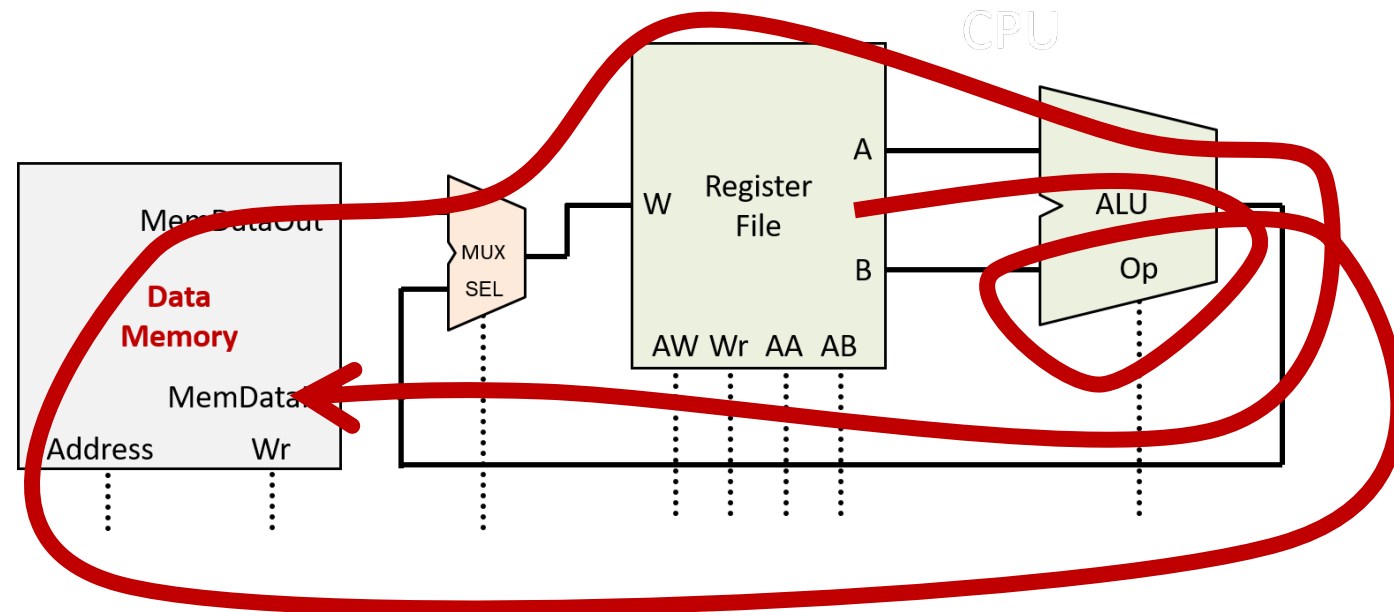
→ `x0 = x1 + mem[x2];`

- PC-Relative

`add x0, x1, 123(pc)`

→ `x0 = x1 + mem[pc + 123];`

An Example from x86/x64



`ADD DWORD PTR [EBX + ESI*4 + 16], EAX`

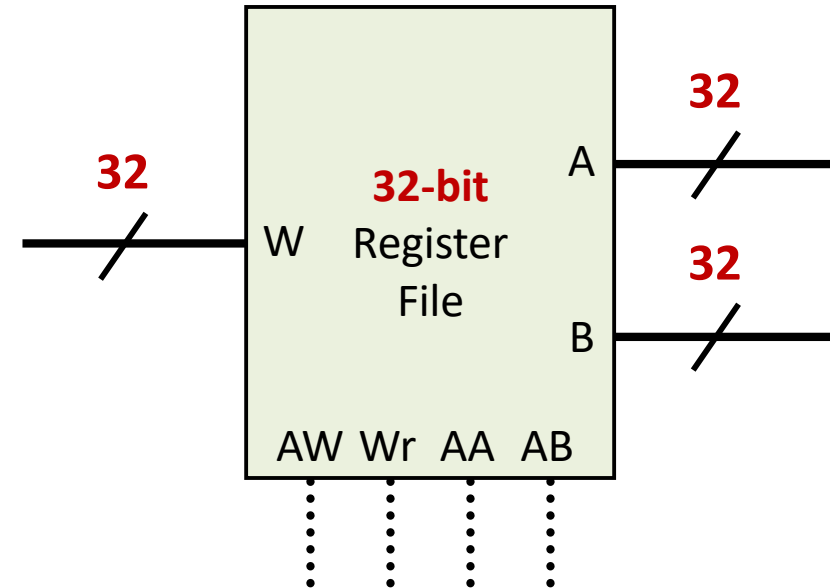
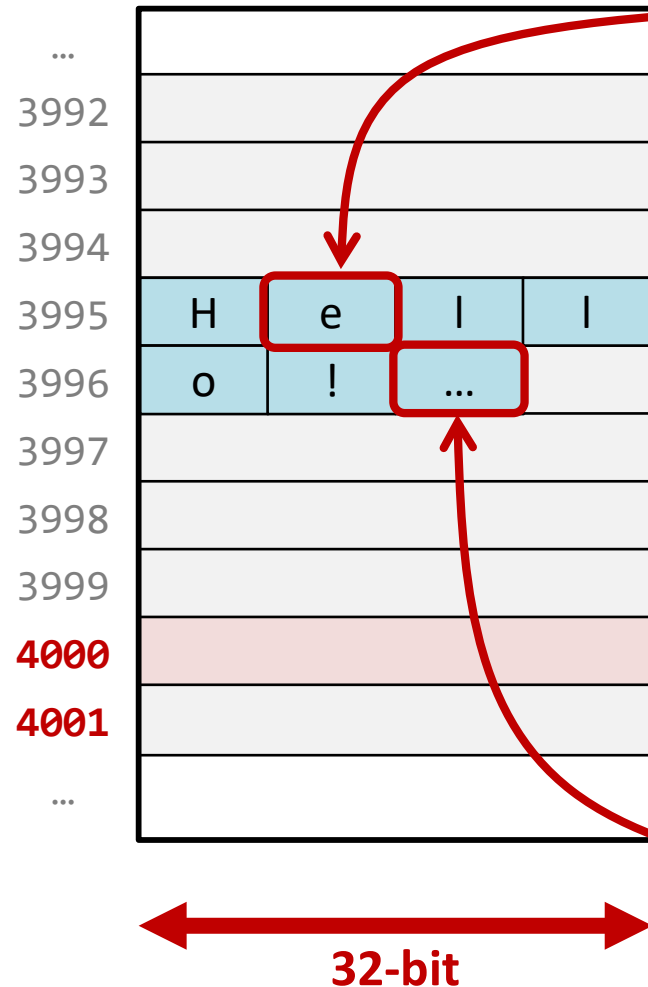
x86/x64
registers

This roughly means: **read memory** at address $EBX + ESI \cdot 4 + 16$, **add** EAX to it, and **write the result back** into memory (at the same address)

Word Addressed Memory

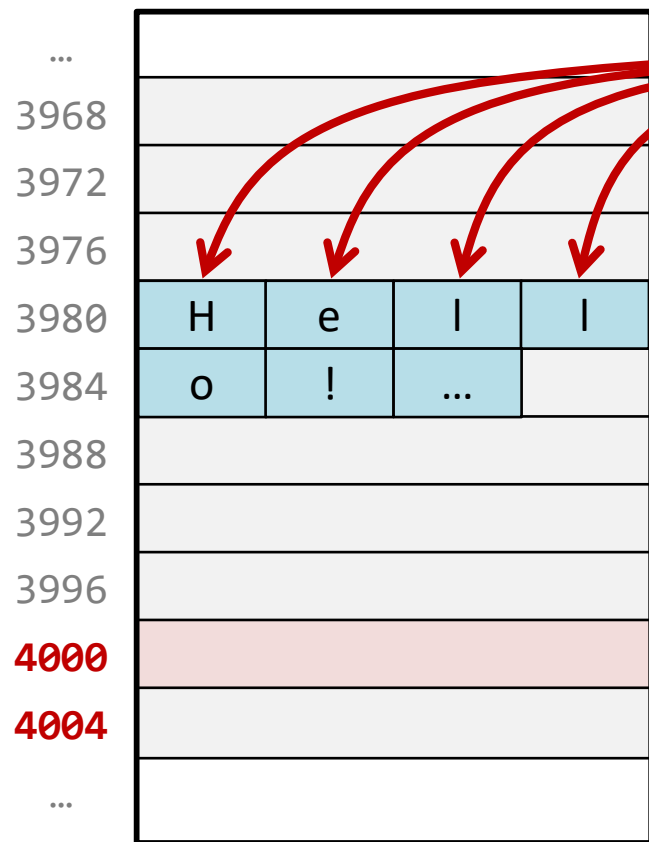
Yet, **bytes** (8-bit data) are quite **important!**

Disks are organized in bytes,
network packets are bytes...



How do I identify this?!
“The third byte of 3996”?!

Byte Addressed Memory



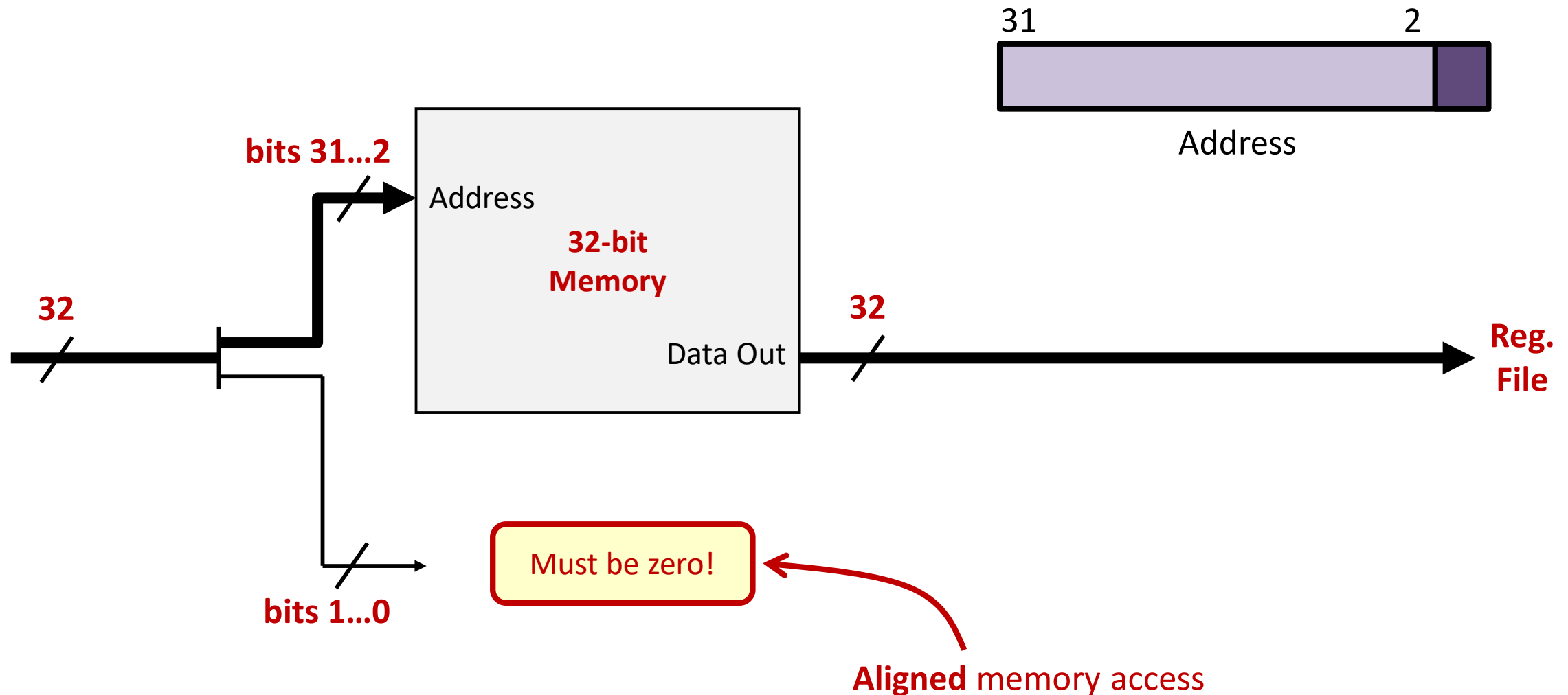
We will identify these bytes as
3980, 3981, 3982, and 3983

Number **bytes** and **not words**!

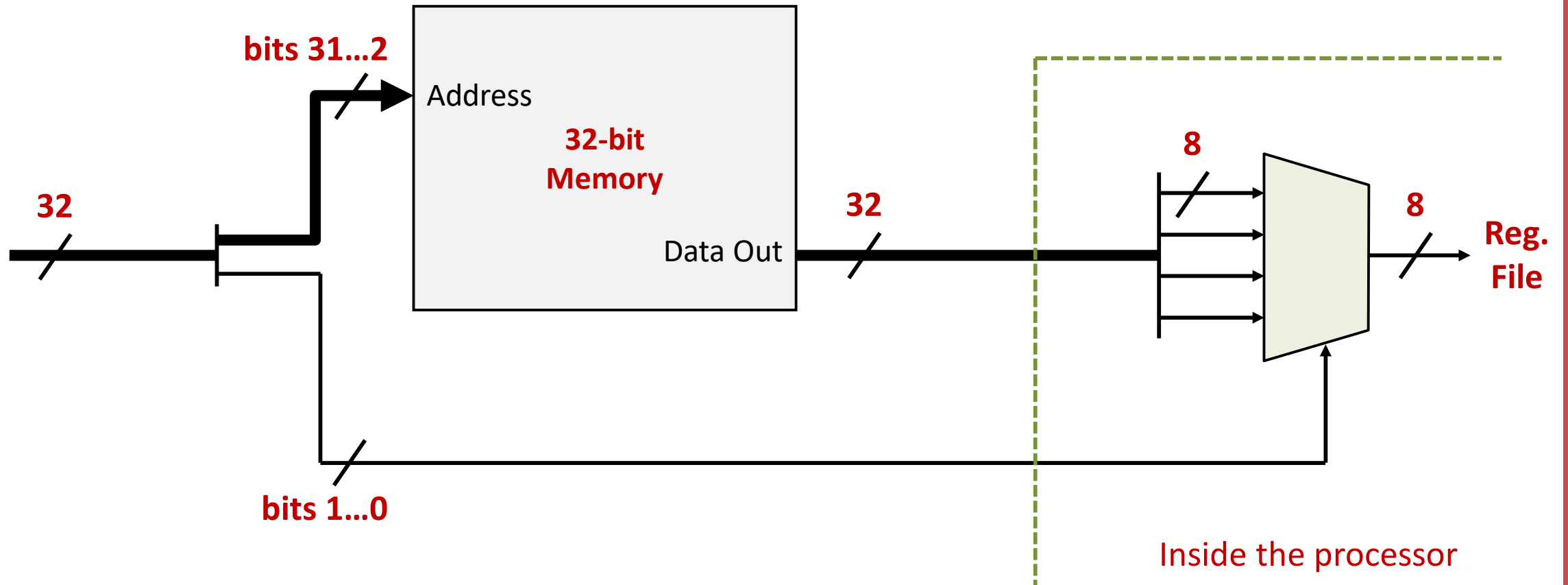
All 32-bit words are placed at addresses
that are **multiple of four**

32-bit

Loading Words (**lw**) and Instructions



Loading Bytes (1b)



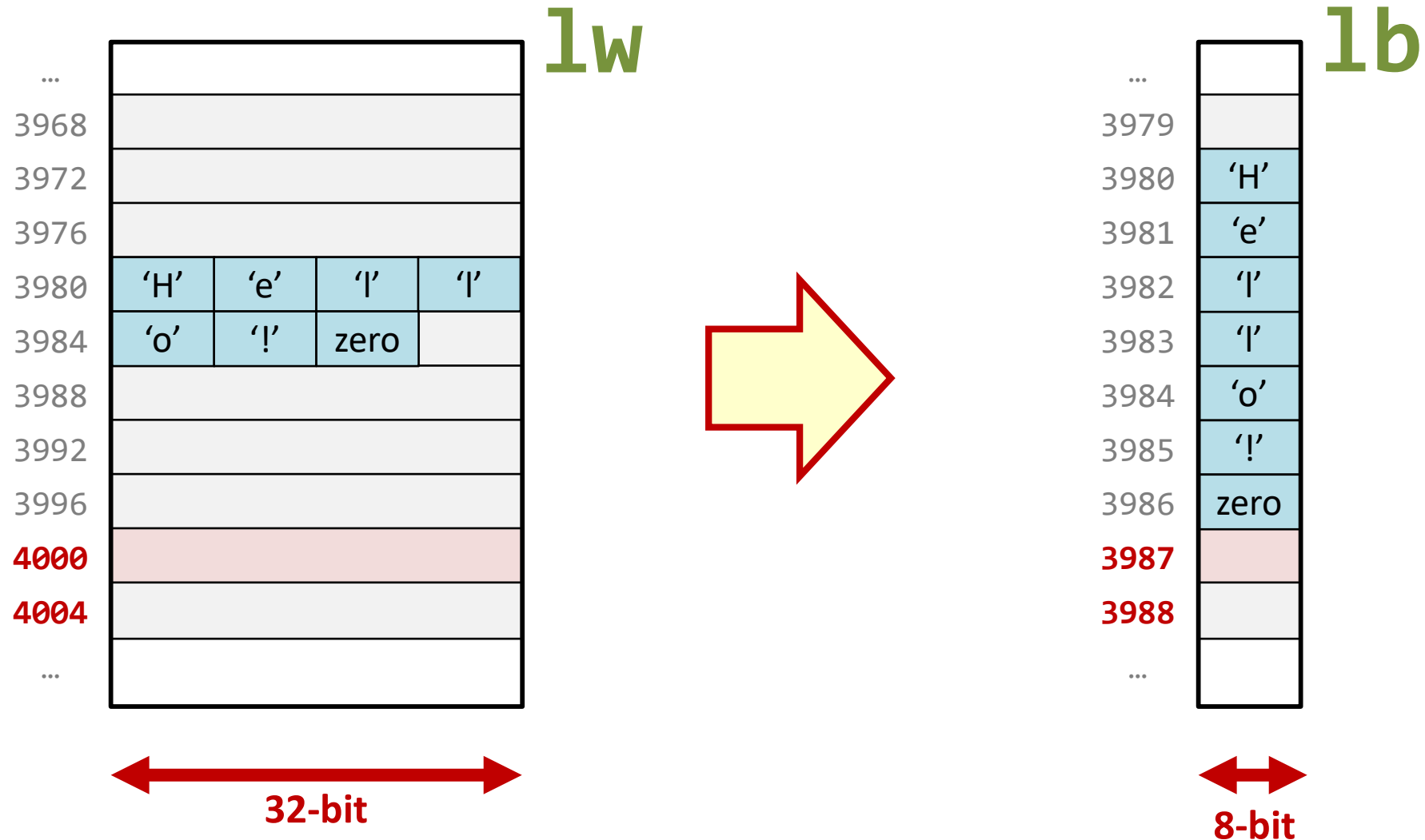
A Few More Load/Store Instructions

- Access **bytes** (and half-words) as if memory were made of bytes

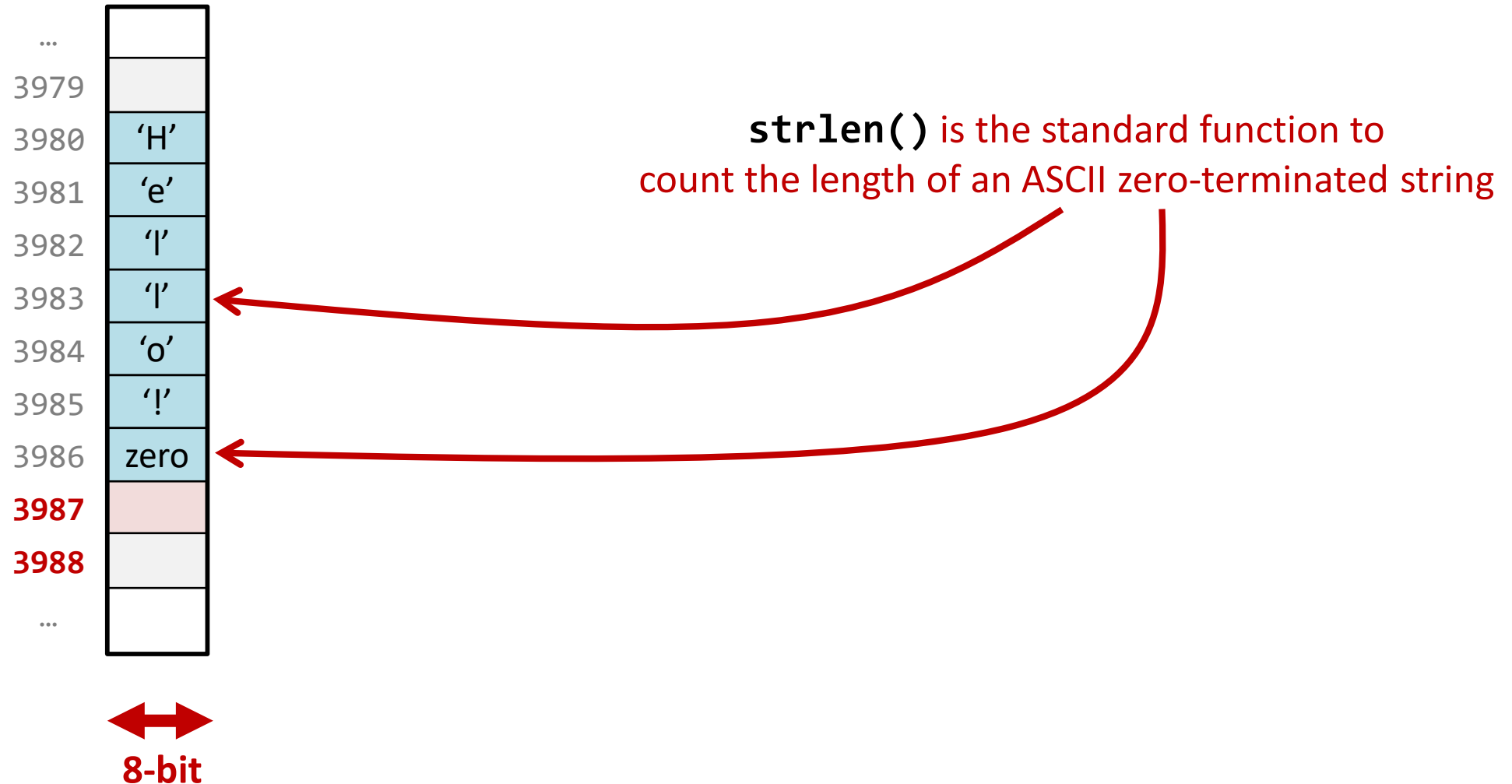
Load						
lb	rd, imm(rs1)	$rd \leftarrow \text{sext}(\text{mem}[\text{rs1} + \text{sext}(\text{imm})][7 : 0])$	I	0x0	0x03	
lbu	rd, imm(rs1)	$rd \leftarrow \text{zext}(\text{mem}[\text{rs1} + \text{sext}(\text{imm})][7 : 0])$	I	0x4	0x03	
lh	rd, imm(rs1)	$rd \leftarrow \text{sext}(\text{mem}[\text{rs1} + \text{sext}(\text{imm})][15 : 0])$	I	0x1	0x03	
lhu	rd, imm(rs1)	$rd \leftarrow \text{zext}(\text{mem}[\text{rs1} + \text{sext}(\text{imm})][15 : 0])$	I	0x5	0x03	
Store						
sb	rs2, imm(rs1)	$\text{mem}[\text{rs1} + \text{sext}(\text{imm})] \leftarrow \text{rs2}[7 : 0]$	S	0x0	0x23	
sh	rs2, imm(rs1)	$\text{mem}[\text{rs1} + \text{sext}(\text{imm})] \leftarrow \text{rs2}[15 : 0]$	S	0x1	0x23	

When putting something smaller (e.g., a byte)
into a larger register (32-bit), extension matters:
zero extend **unsigned** numbers and **sign** extend **signed** numbers

Access Memory as It Is More Suitable!



Example: Count Characters in a String




Example: Count Characters in a String

strlen:

```
mv    t0, a0      # Copy the pointer (a0) into t0 to traverse the string
li    t1, 0        # t1 will hold the length (initialized to 0)
```

loop:

```
lbu    t2, 0(t0)   # Load byte at address t0 into t2
beq    t2, zero, end # If t2 is 0 (null byte), we are done
addi   t1, t1, 1    # Increment the length counter (t1)
addi   t0, t0, 1    # Point to the next character in the string
j      loop        # Repeat the loop
```

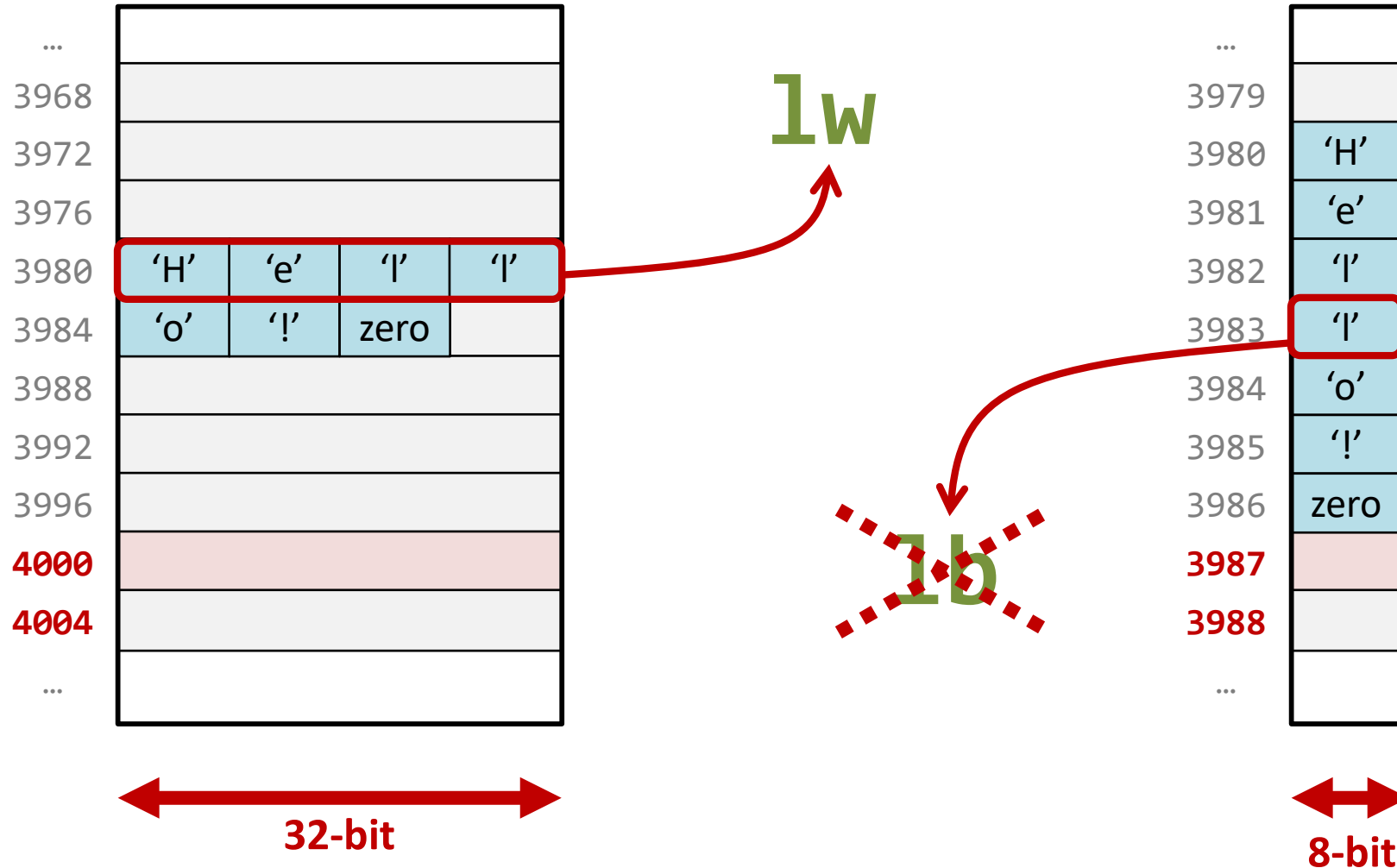


end:

```
mv    a0, t1      # Move the length (t1) into a0 as the return value
ret                    # Return to caller
```

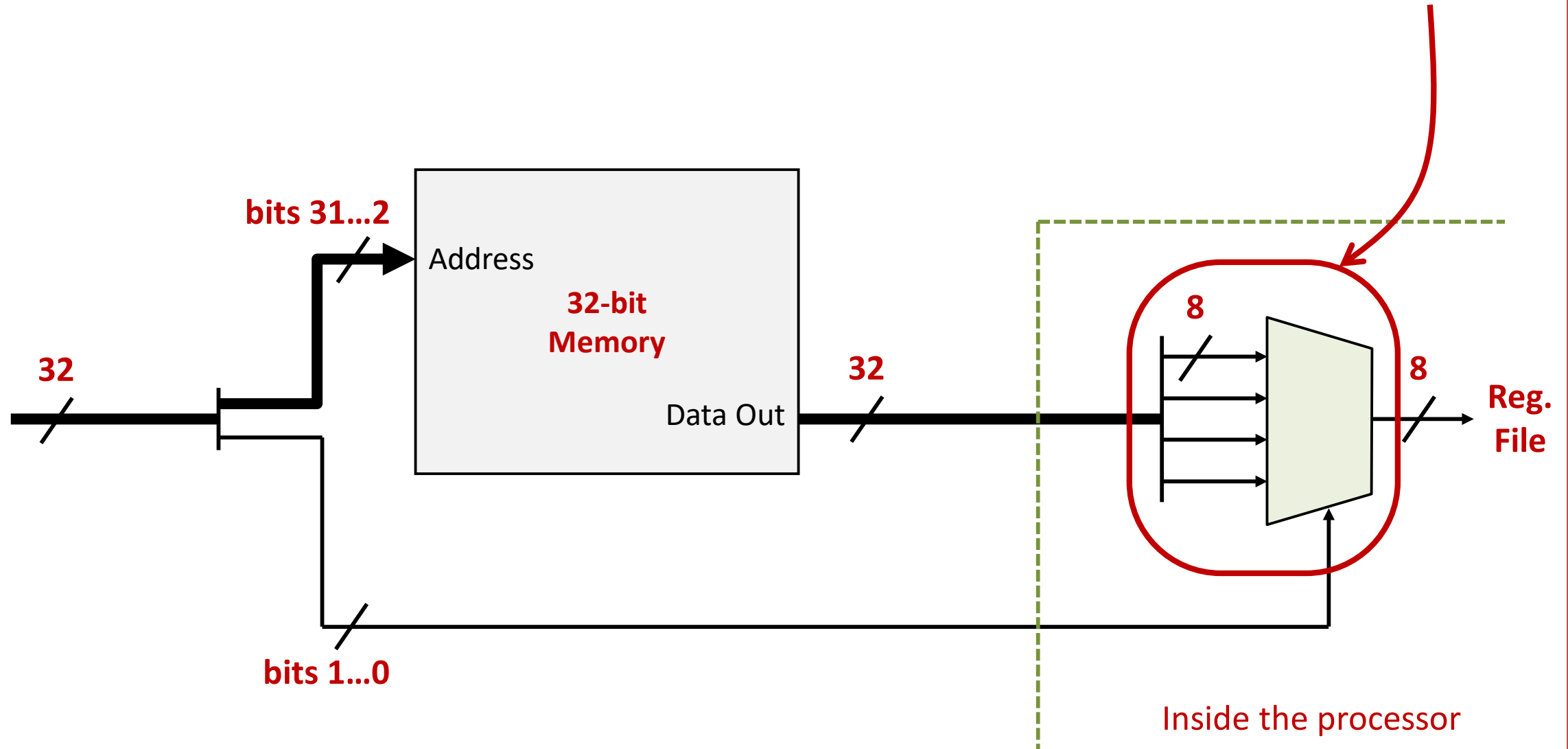
Could this be **lb**? Which one is correct?

And Using **lw** instead of **lb**?

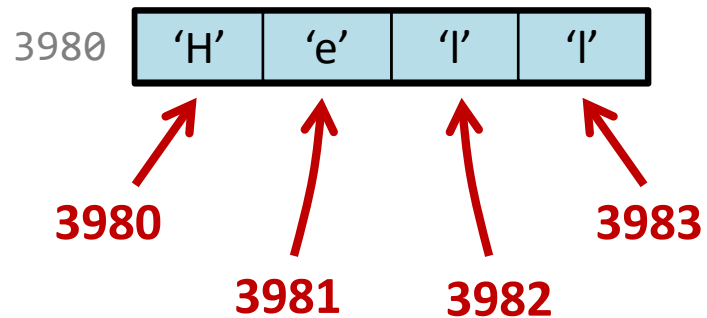
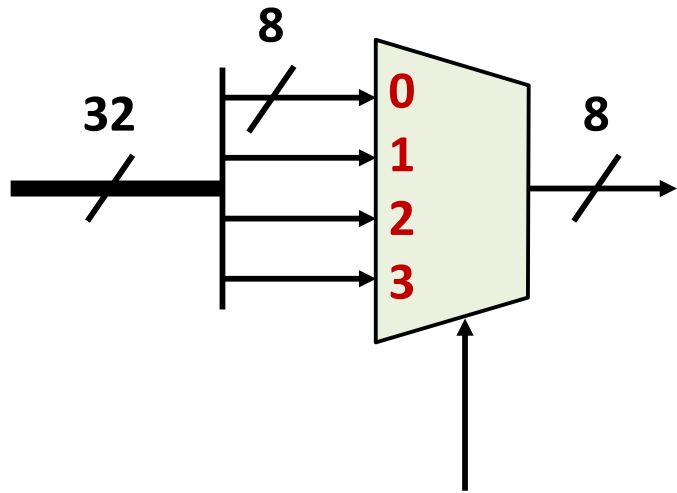


Loading Bytes (1b)

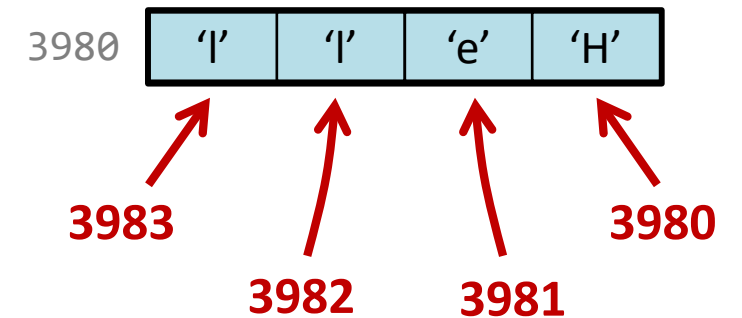
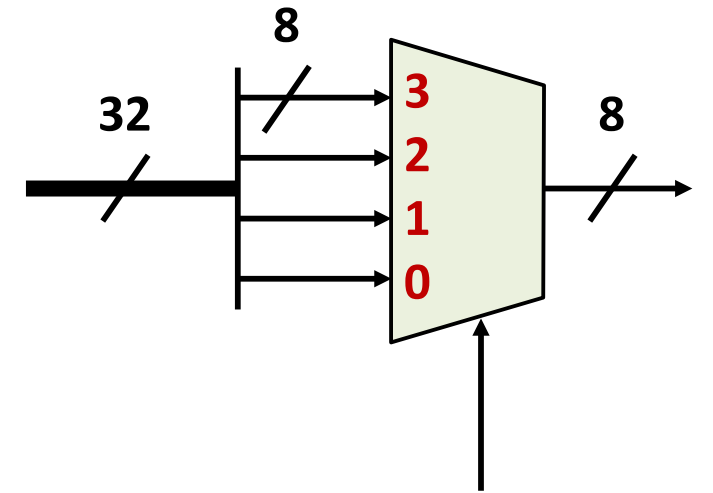
How is this
connected,
exactly?



Which Byte Where?!

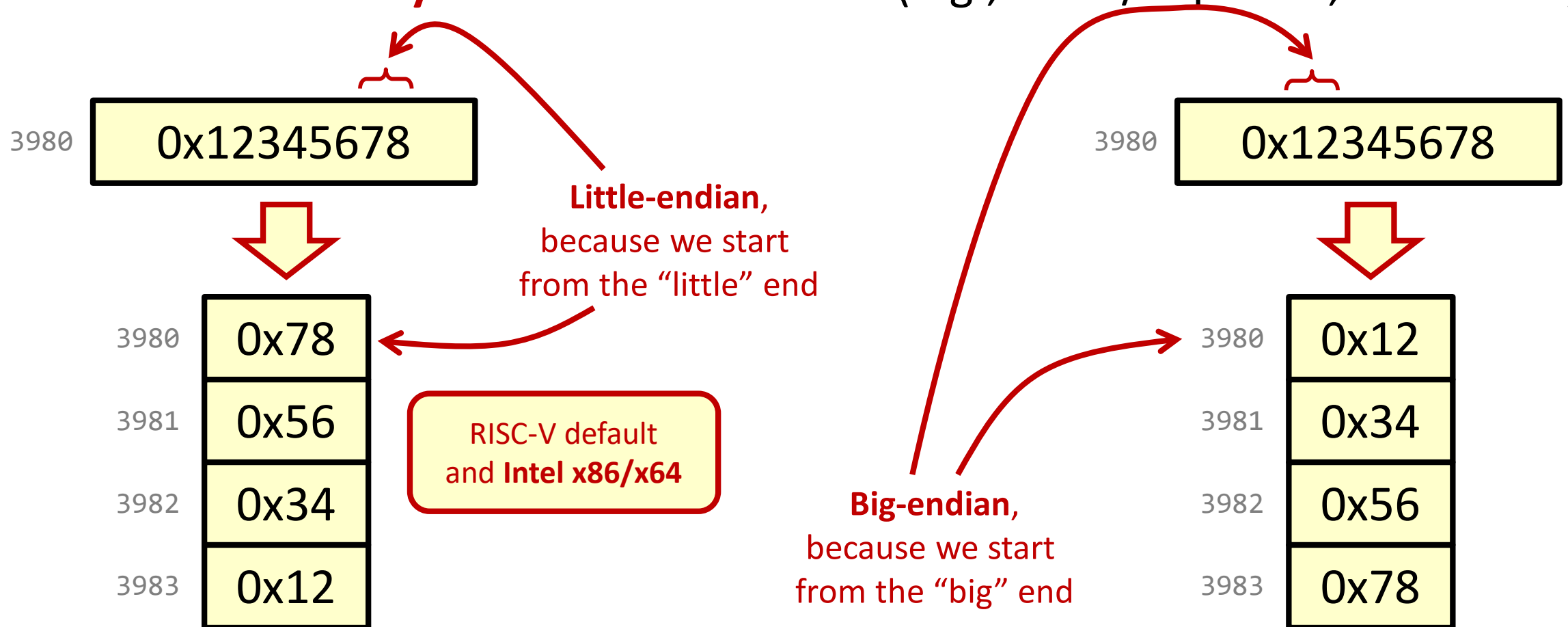


It does **not** matter...
...but we need to decide!

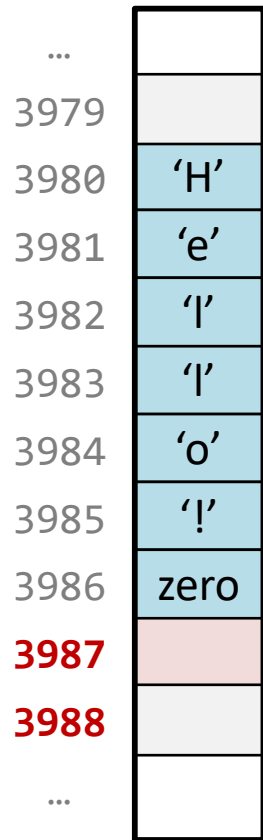


Little Endian or Big Endian?

- It only matters if the same data are accessed **both as words and bytes** or if two **different systems** access the data (e.g., a TCP/IP packet, a WAV file)



Example: `strlen()` with only `lw`



8-bit

Let's assume that the string is **word-aligned**;
that is, the first letter is at an address that is
a multiple of four

And let's do it for a **little-endian** processor

Example: strlen() with only lw

strlen:

```
li    t1, 0           # t1 will hold the length (initialized to 0)
```

next_word:

```
li    t2, 4           # t2 will count the bytes in a loaded word (four)
lw    t3, 0(t0)        # Load four bytes at address t0 into t3
```

next_byte:

```
andi  t4, t3, 0xff     # Move the "little-end" in t4
beq    t4, zero, end    # If t4 is 0 (null byte), we are done
addi   t1, t1, 1        # Increment the length counter (t1)
srli   t3, t3, 8        # Prepare the next byte of the word in the "little-end" (t3)
addi   t2, t2, -1       # One byte left in the loaded word
bnez   t2, next_byte    # If more bytes in t3, check the next
addi   a0, a0, 4        # Else point to the next word of characters in the string
j      next_word        # Repeat the loop
```

end:

```
mv     a0, t1          # Move the length (t1) into a0 as the return value
ret
```

As an exercise, can you write this for a **big-endian** processor?

References

- Patterson & Hennessy, COD – RISC-V Edition
 - **Chapter 2** and, in particular, **Sections 2.3** and **2.9**
 - **Sections 5.1** and **5.2**

