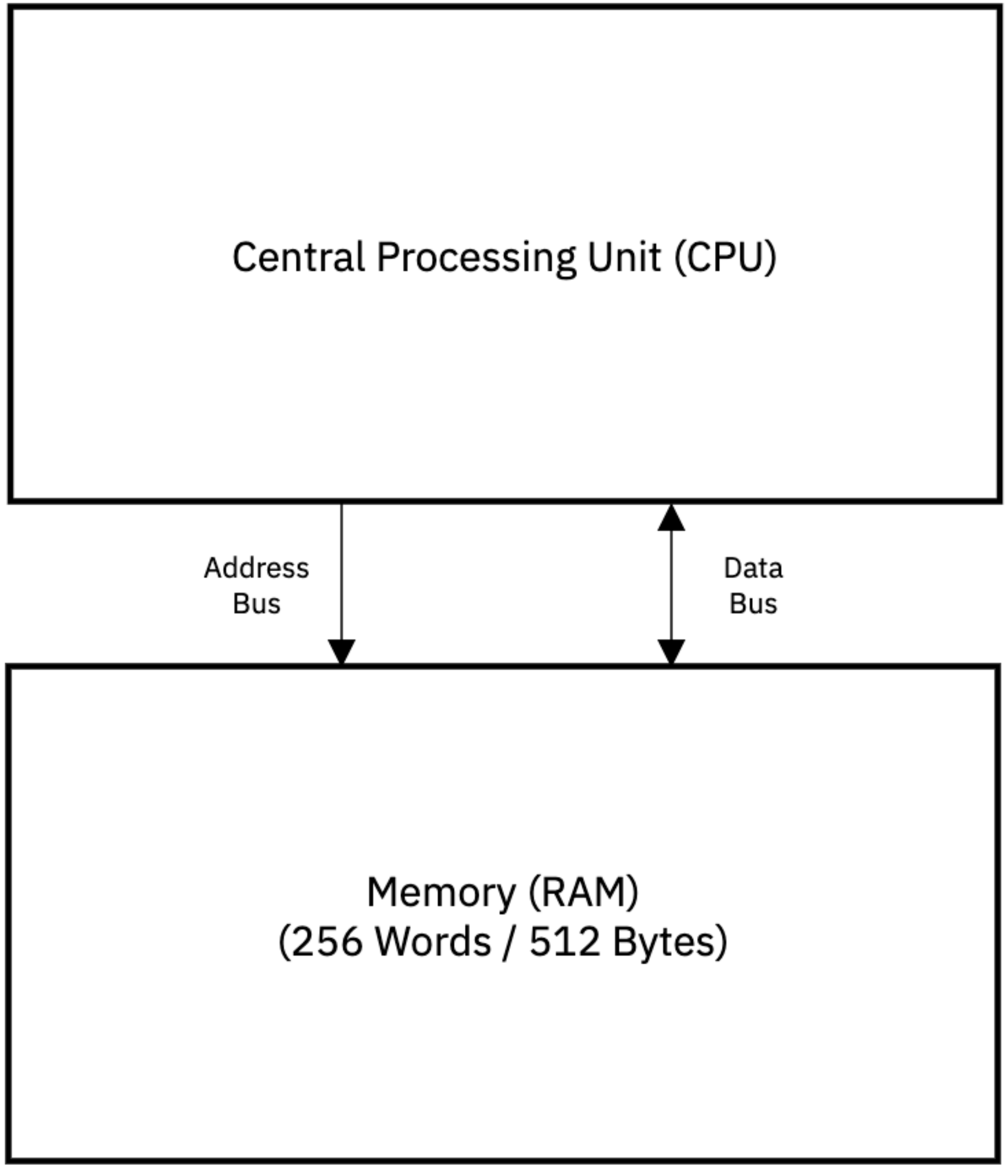
Toy Computer Architecture

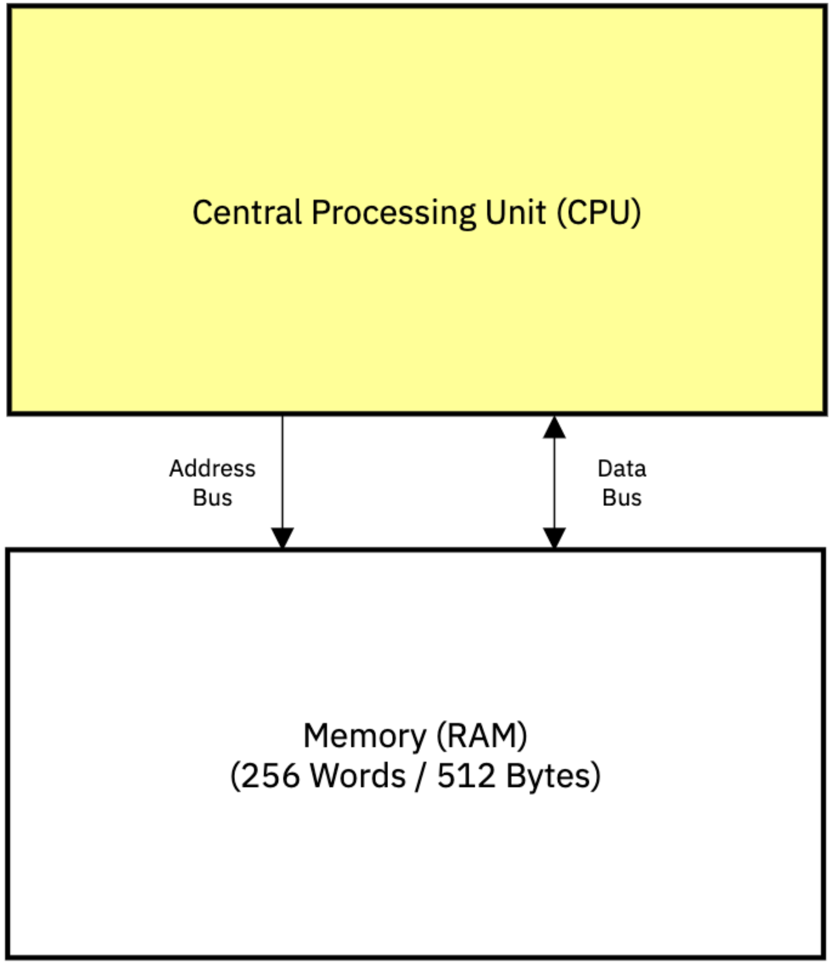
Toy Computer Adapted from:

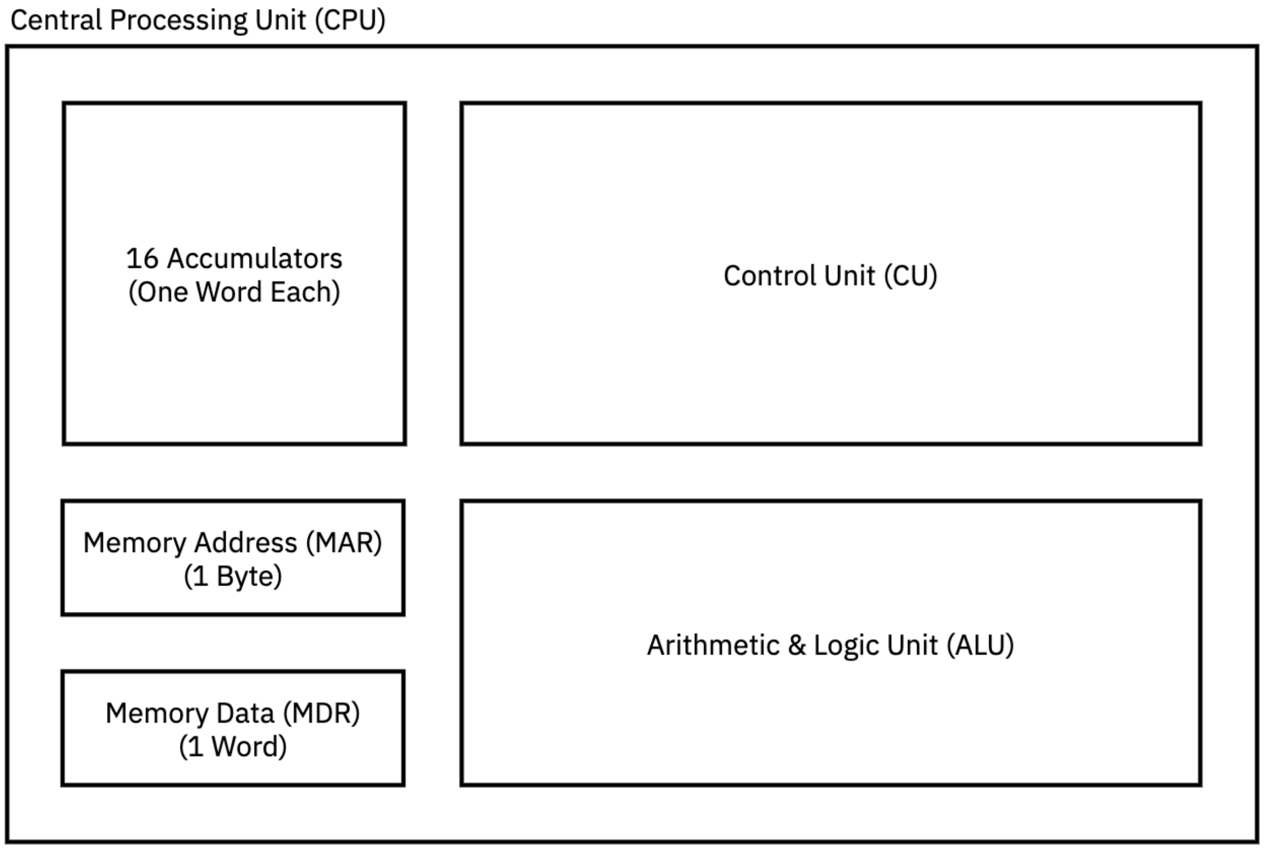
Sedgewick, R. & Wayne, K. (2017)

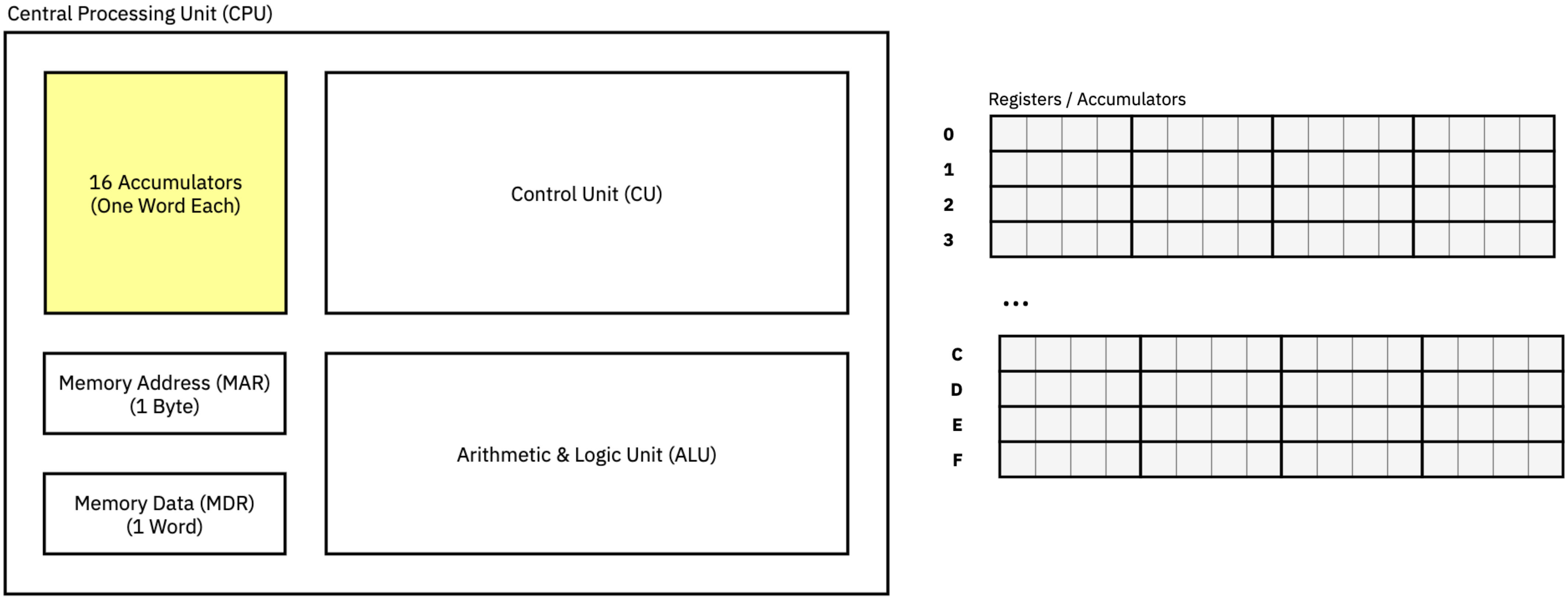
Computer Science: An Interdisciplinary Approach.

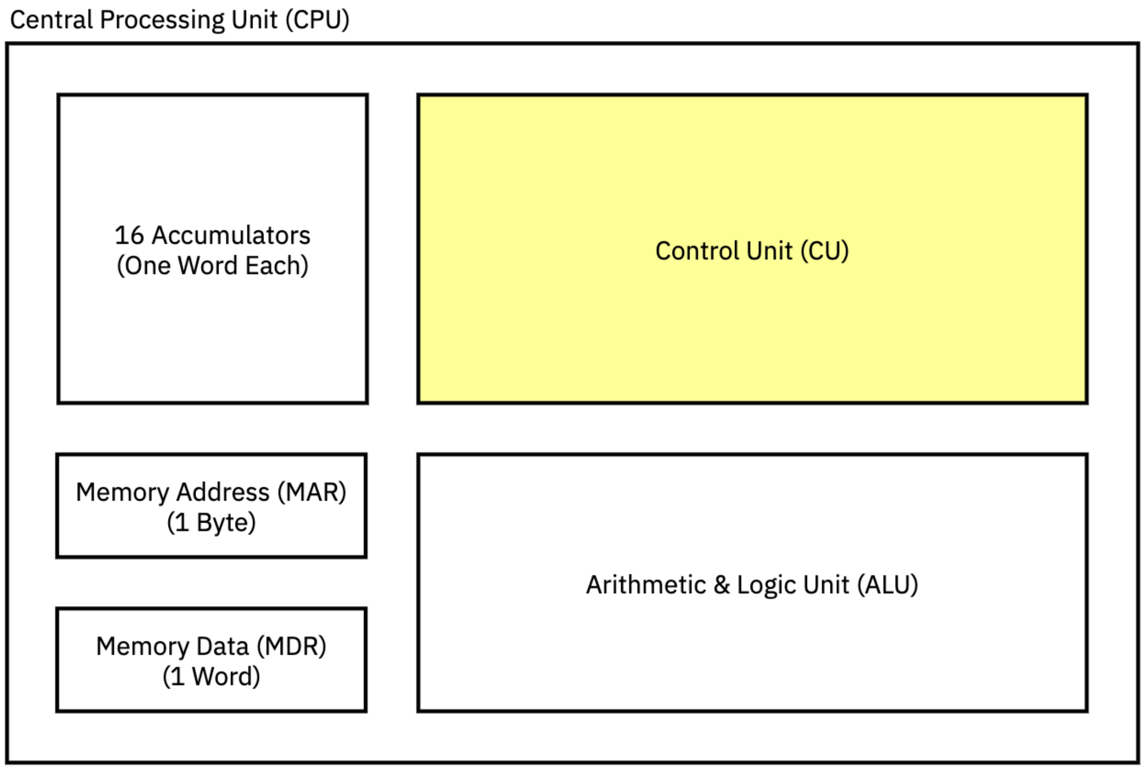
Pearson Education

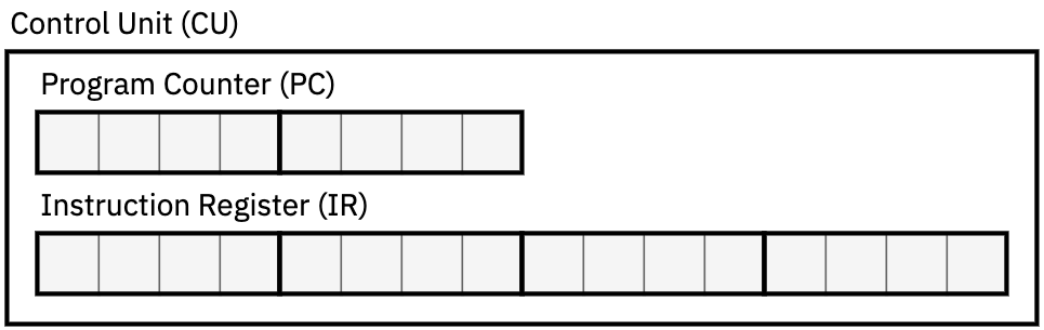

Central Processing Unit (CPU)
Address
Bus
Data
Bus
Memory (RAM)
(256 Words / 512 Bytes)

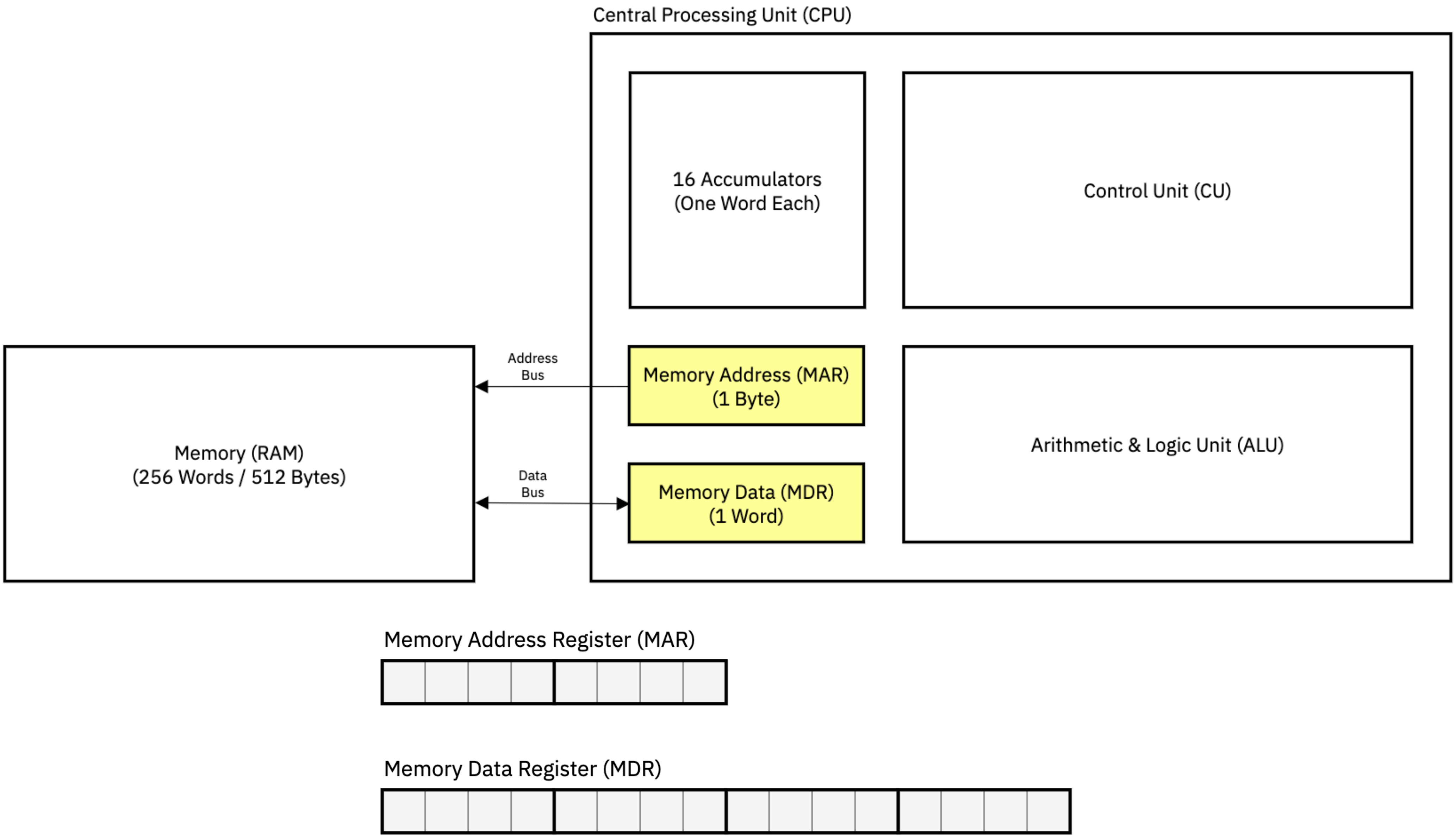

Central Processing Unit (CPU)
Address Bus
Data Bus
Memory (RAM)
(256 Words / 512 Bytes)

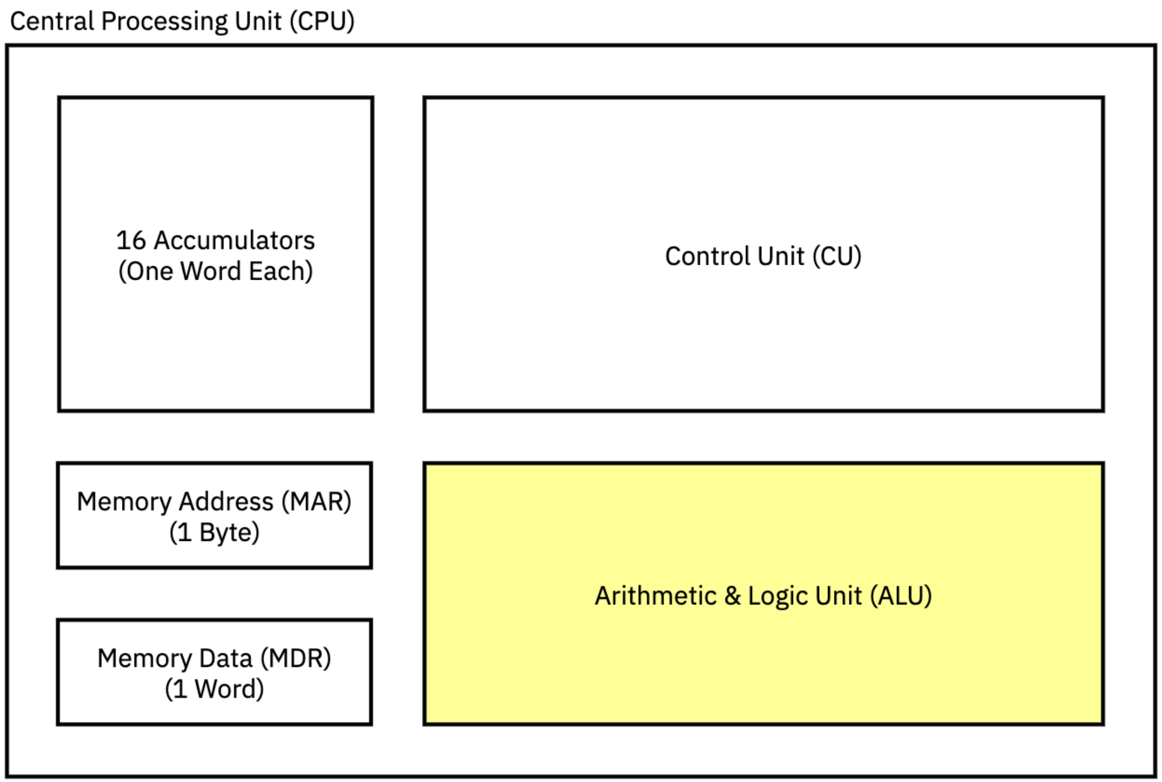

Central Processing Unit (CPU)
16 Accumulators
(One Word Each)
Control Unit (CU)
Memory Address (MAR)
(1 Byte)
Memory Data (MDR)
(1 Word)
Arithmetic & Logic Unit (ALU)

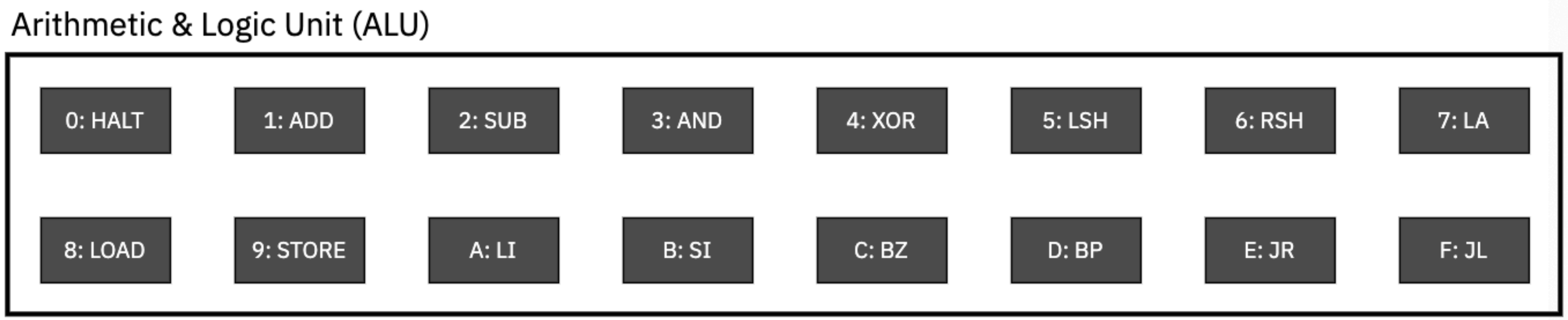

Central Processing Unit (CPU)
16 Accumulators
(One Word Each)
Control Unit (CU)
Memory Address (MAR)
(1 Byte)
Memory Data (MDR)
(1 Word)
Arithmetic & Logic Unit (ALU)
Registers / Accumulators
0
1
2
3
...
C
D
E
F

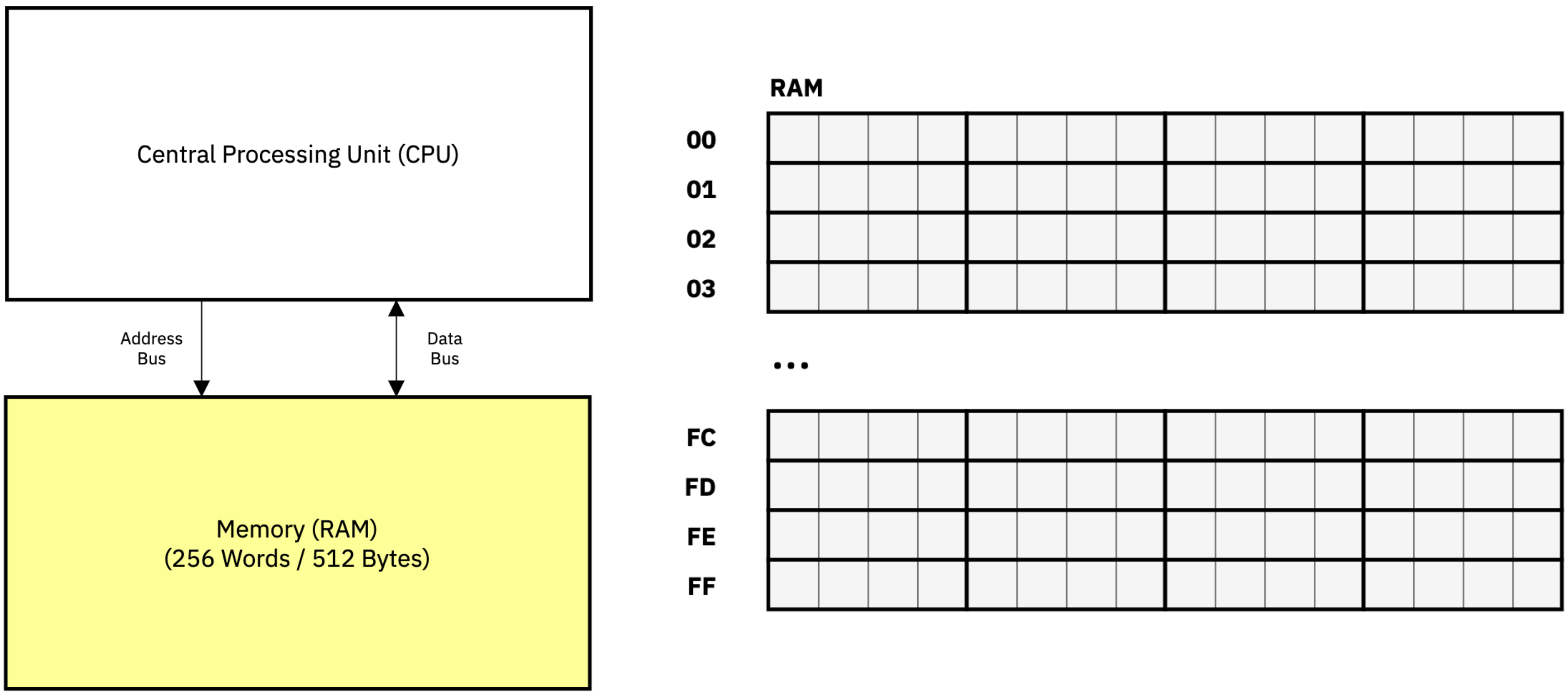

Central Processing Unit (CPU)
16 Accumulators
(One Word Each)
Control Unit (CU)
Memory Address (MAR)
(1 Byte)
Arithmetic & Logic Unit (ALU)
Memory Data (MDR)
(1 Word)


Control Unit (CU)
Program Counter (PC)
Instruction Register (IR)

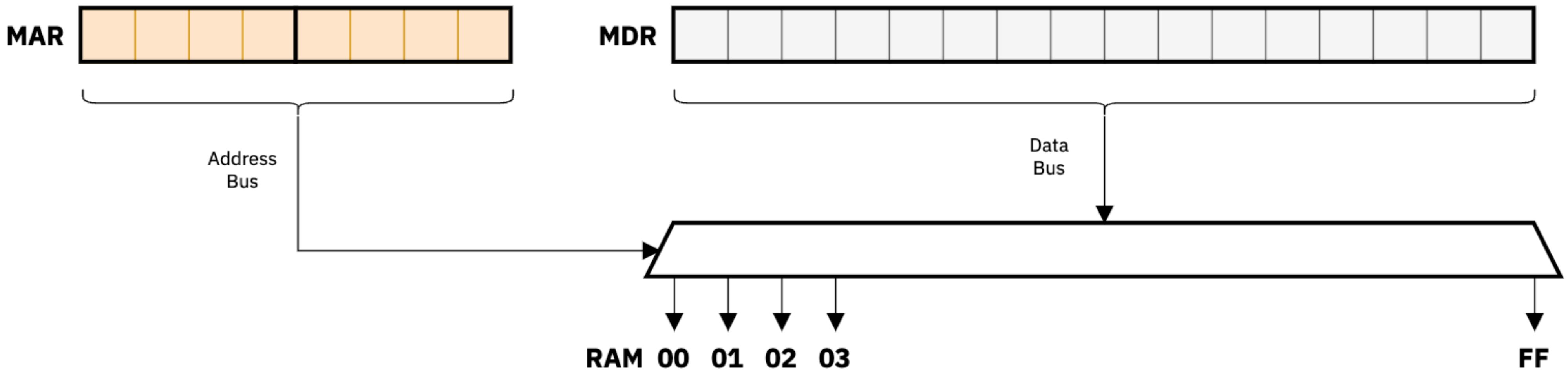

Central Processing Unit (CPU)
16 Accumulators
(One Word Each)
Control Unit (CU)
Address
Bus
Memory Address (MAR)
(1 Byte)
Memory (RAM)
(256 Words / 512 Bytes)
Data
Bus
Memory Data (MDR)
(1 Word)
Arithmetic & Logic Unit (ALU)
Memory Address Register (MAR)
Memory Data Register (MDR)


Central Processing Unit (CPU)
16 Accumulators
(One Word Each)
Control Unit (CU)
Memory Address (MAR)
(1 Byte)
Memory Data (MDR)
(1 Word)
Arithmetic & Logic Unit (ALU)

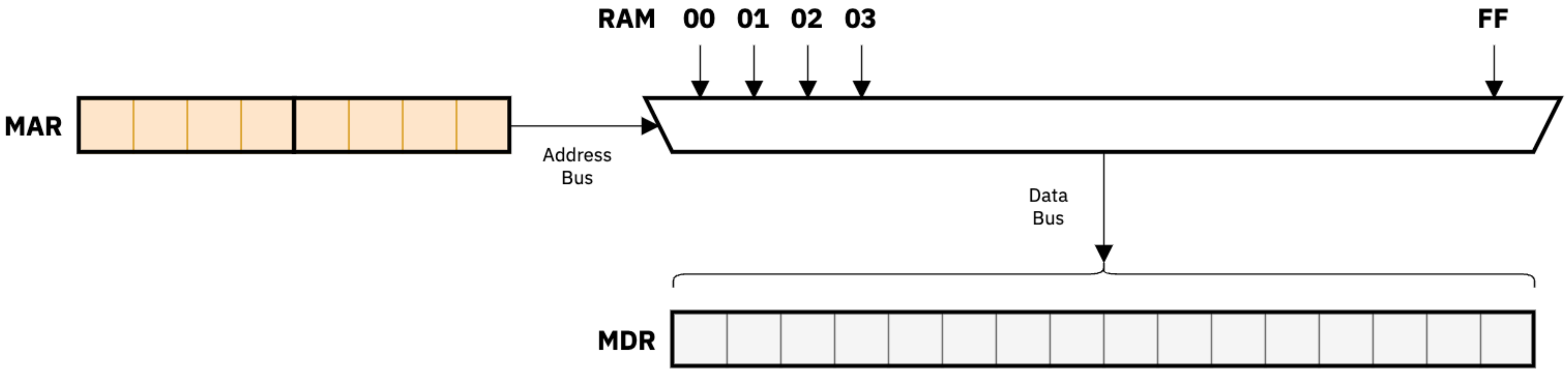

Arithmetic & Logic Unit (ALU)
0: HALT
1: ADD
2: SUB
3: AND
4: XOR
5: LSH
6: RSH
7: LA
8: LOAD
9: STORE
A: LI
B: SI
C: BZ
D: BP
E: JR
F: JL


Central Processing Unit (CPU)
Address Bus
Data Bus
Memory (RAM)
(256 Words / 512 Bytes)
RAM
00
01
02
03
...
FC
FD
FE
FF

Storing to RAM



Loading from RAM

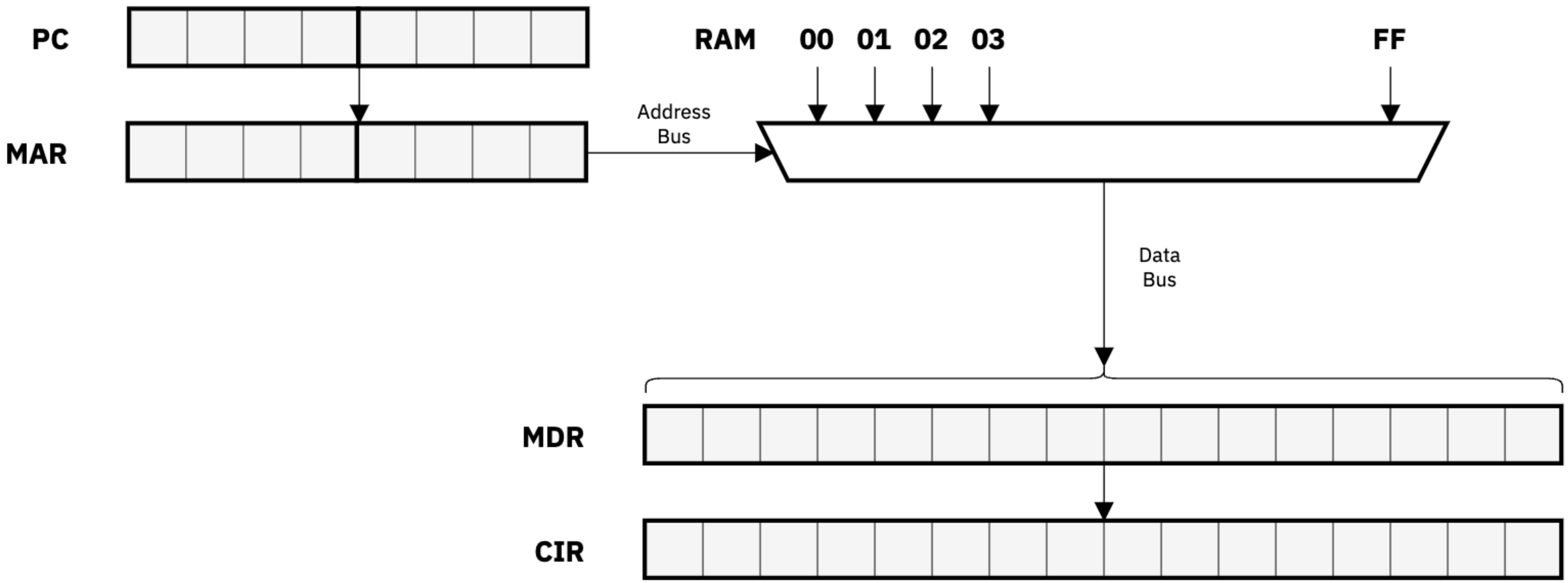


How are programs executed?

Machine Instruction Cycle

1. Fetch
2. Decode
3. Execute

1. Fetch



1. Fetch

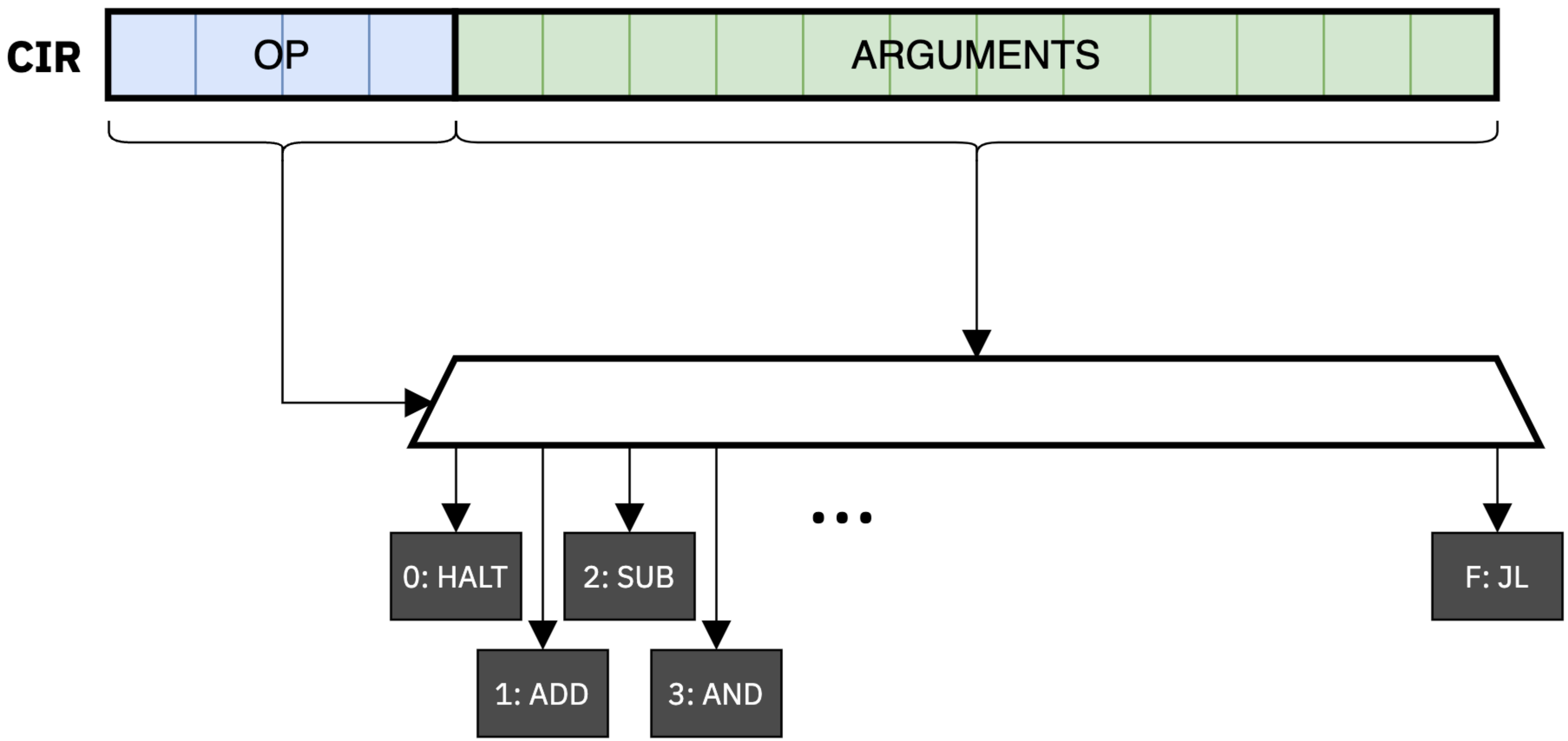
(i) Address copied from PC to MAR.

(ii) Data at address in MAR copied to MDR.

(iii) Data in MDR copied to CIR.

(iv) Value in PC incremented.

2. Decode

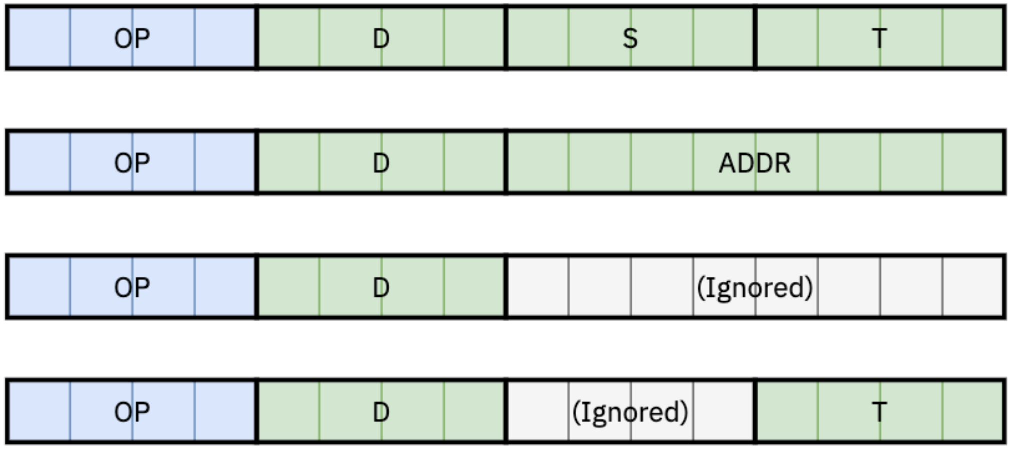


2. Decode

Part of the data in CIR determines which circuit in the ALU the rest of the data in the CIR is sent to.

3. Execute

CIR



0	Halt	-
1	Add	R[D] <- R[S] + R[T]
2	Subtract	R[D] <- R[S] - R[T]
3	Bitwise And	R[D] <- R[S] & R[T]
4	Bitwise Xor	R[D] <- R[S] ^ R[T]
5	Left Shift	R[D] <- R[S] << R[T]
6	Right Shift	R[D] <- R[S] >> R[T]
7	Load Address	R[D] <- ADDR
8	Load	R[D] <- M[ADDR]
9	Store	M[ADDR] <- R[D]
A	Load Indirect	R[D] <- M[R[T]]
B	Store Indirect	M[R[T]] <- R[D]
C	Branch Zero	if R[D] == 0 PC <- ADDR
D	Branch Positive	if R[D] > 0 PC <- ADDR
E	Jump Register	PC <- R[D]
F	Jump & Link	R[D] <- PC; PC <- ADDR

3. Execute

Helpful calculations or effects are achieved through the hardwired circuits in the ALU.

Some of the instructions involve interactions with the MAR and MDR to **load** data from memory to registers or to **store** data from registers to memory.

```
0 Halt            -
1 Add             R[D] <- R[S] + R[T]
2 Subtract        R[D] <- R[S] - R[T]
3 Bitwise And     R[D] <- R[S] & R[T]
4 Bitwise Xor     R[D] <- R[S] ^ R[T]
5 Left Shift      R[D] <- R[S] << R[T]
6 Right Shift     R[D] <- R[S] >> R[T]
7 Load Address    R[D] <- ADDR
8 Load            R[D] <- M[ADDR]
9 Store           M[ADDR] <- R[D]
A Load Indirect   R[D] <- M[R[T]]
B Store Indirect  M[R[T]] <- R[D]
C Branch Zero     if R[D] == 0 PC <- ADDR
D Branch Positive if R[D] > 0 PC <- ADDR
E Jump Register   PC <- R[D]
F Jump & Link     R[D] <- PC; PC <- ADDR
```

Input / Output

Most computers have an additional bus to interact with input / output devices — like keyboards, mice, touch screens, printers, hard drives and networks.

Toy Computer performs “magic” I/O via interactions with special memory addresses.

Special Load Addresses

```
8 Load          R[D] <- M[ADDR]
A Load Indirect R[D] <- M[R[T]]
```

ADDR / R[T]	**Loads**
F0	User Input
FA	Random Word

Special Store Addresses

```
9 Store          M[ADDR] <- R[D]
B Store Indirect M[R[T]] <- R[D]
```

ADDR / R[T]	**Outputs**
F1	R[D] in binary
F2	R[D] in octal
F3	R[D] in hexadecimal
F4	R[D] in denary
F5	R[D] as ascii character
F6	A new line
F7	R[D] as binary pattern
F8	Computer State
F9	Computer State as Machine Code

Machine Language

Machine language allows us to specify what data should be saved and where in the computer's memory the data should be saved.

Toy Machine Language uses hexadecimal and has only got two instructions:

- PC: *addr* (Sets the value stored to the PC)
- *addr*: *data* (Stores *data* to address *addr*)

Example

Write a Toy Machine Language program that inputs two integers from the user and outputs the sum of the two integers.

Example

Write a Toy Machine Language program that inputs two integers from the user and outputs the sum of the two integers.

1. Load the first value into a register, say R[0].
2. Load the second value into a register, say R[1]
3. Perform the addition and store the result into a register, say R[2].
4. Output the result.

Example

Write a Toy Machine Language program that inputs two integers from the user and outputs the sum of the two integers.

Address	Data
PC:	00
00:	80F0
01:	81F0
02:	1201
03:	92F4
04:	0000

Example

Write a Toy Machine Language program that inputs two integers from the user and outputs the sum of the two integers.

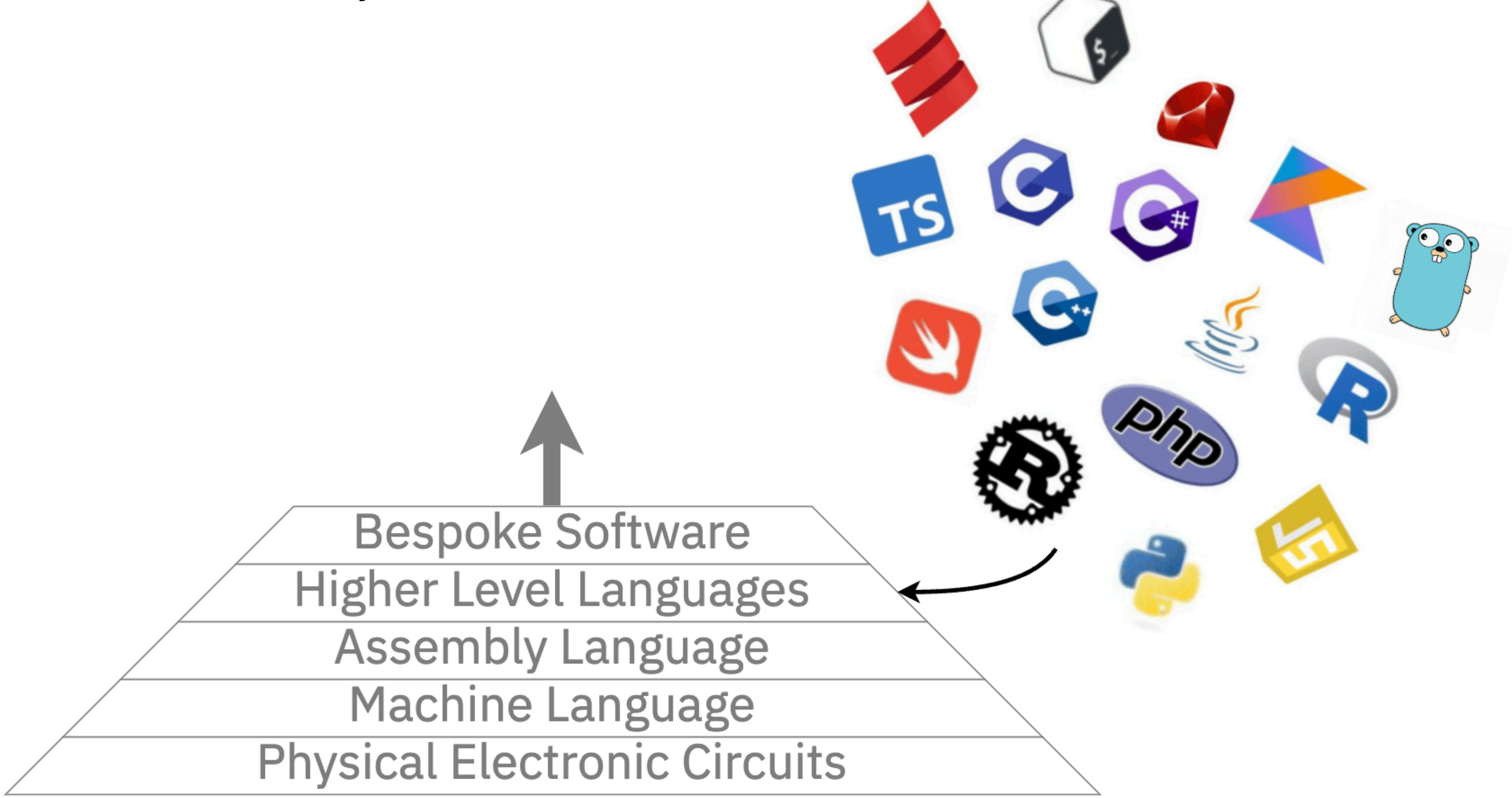
Address	Data	
PC:	00	Sets PC to zero.
00:	80F0	R[0] ← M[F0] (Reads user input)
01:	81F0	R[1] ← M[F0] (Reads user input)
02:	1201	R[2] ← R[0] + R[1]
03:	92F4	M[F4] ← R[2] (Outputs as denary)
04:	0000	Halts

Toy Assembly

Layers of Abstraction

What is abstraction, and how is abstraction central to computational thinking?

Layers of Abstraction



Assembly Language

- Uses *mnemonics*, such as **ld** (for load) and **st** (for store), instead of explicitly specifying the underlying architecture that instructions are referencing.

- Allows us to give important memory addresses *labels* so that we can refer to these addresses without using explicit address values. (This also means we don't need to worry about updating memory addresses in our code when we insert or edit!)

- Each assembly instruction is directly translated to a sequence (one or more) of machine instructions.

Toy Assembly

Program Control

By default, the address in the PC is incremented as part of the fetch-decode-execute cycle. *Program control* instructions can conditionally or unconditionally change the address in the PC. They are:

- *halt*
- *jz*
- *jp*
- *jmp*
- *call*
- *ret*

Toy Assembly

Moving Data

Some instructions **load** data to registers, **store** data to memory or **move** data between registers:

- *ld*
- *st*
- *mv*

Toy Assembly

Operations

Special purpose circuits in the ALU perform operations on data:

- **Arithmetic:**

  add, sub

- **Logic:**

  and, or, xor, not

- **Bit Shifting:**

  lsh, rsh

Toy Assembly

Labels

Labels allow us to give addresses mnemonic names.

In Toy Assembly, the special name **.main** labels the starting position of the PC.

Toy Assembly

Input

In Toy Computer, input is loaded to registers from special memory addresses; in Toy Assembly, we use the mnemonics:

- *.rand*
- *.input*
- *.string*

Toy Assembly

Output

In Toy Computer, output is stored to special memory addresses; in Toy Assembly, we use the mnemonics:

- *.char*
- *.bin*
- *.oct*
- *.den*
- *.hex*

Toy Assembly

Data

Special data instructions include:

- *.data*
- *.ascii*

Example

Here is an assembly *Hello World!* program:

```
greet: .ascii "Hello, world!"
       .main
       ld %0 greet
loop:  ld %1 [%0]
       jz %1 done
       .char %1
       add %0 1
       jmp loop
done:  .line
       halt
```

Example

Here is an assembly *Hello World!* program:

```
greet: .ascii "Hello, world!"
       .main
       ld %0 greet
loop:  ld %1 [%0]
       jz %1 done
       .char %1
       add %0 1
       jmp loop
done:  .line
       halt
```

```
PC: 0e
00: 0048;    72    'H'    0e: 7000; R[0] <- 0  (*)
01: 0065;   101    'e'    0f: a100; R[1] <- M[R[0]]
02: 006c;   108    'l'    10: c116; if (R[1] == 0) PC <- 16
03: 006c;   108    'l'    11: 91f5; M[f5] <- R[1]
04: 006f;   111    'o'    12: 7e01; R[e] <- 1
05: 002c;    44    ','    13: 100e; R[0] <- R[0] + R[e]
06: 0020;    32    ' '    14: 7f0f; R[f] <- f
07: 0077;   119    'w'    15: ef00; PC <- R[f]
08: 006f;   111    'o'    16: 90f6; M[f6] <- R[0]
09: 0072;   114    'r'    17: 0000
0a: 006c;   108    'l'
0b: 0064;   100    'd'
0c: 0021;    33    '!'
0d: 0000
```