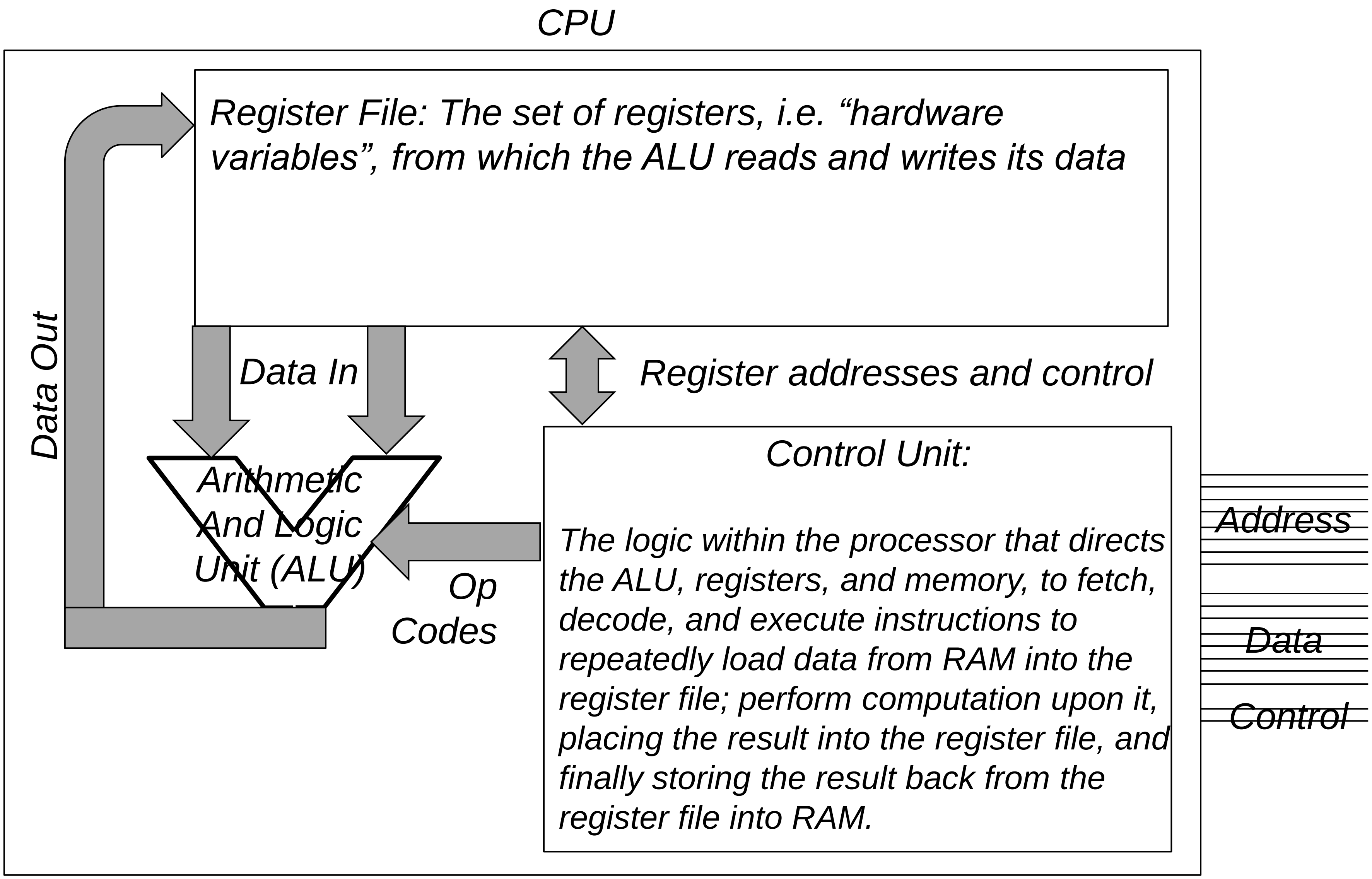
18-100 Introduction to Electrical and Computer Engineering

Lecture 11

Putting it Together: Designing a Computer!

CPU: A Block Diagram

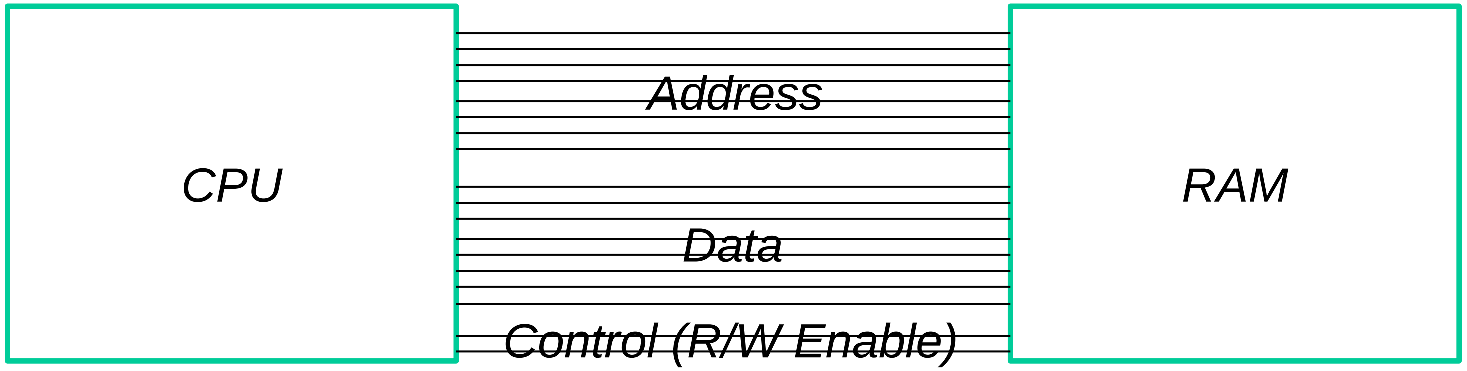

CPU
Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

Register File: The set of registers, i.e. "hardware variables", from which the ALU reads and writes its data

Control Unit:

The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

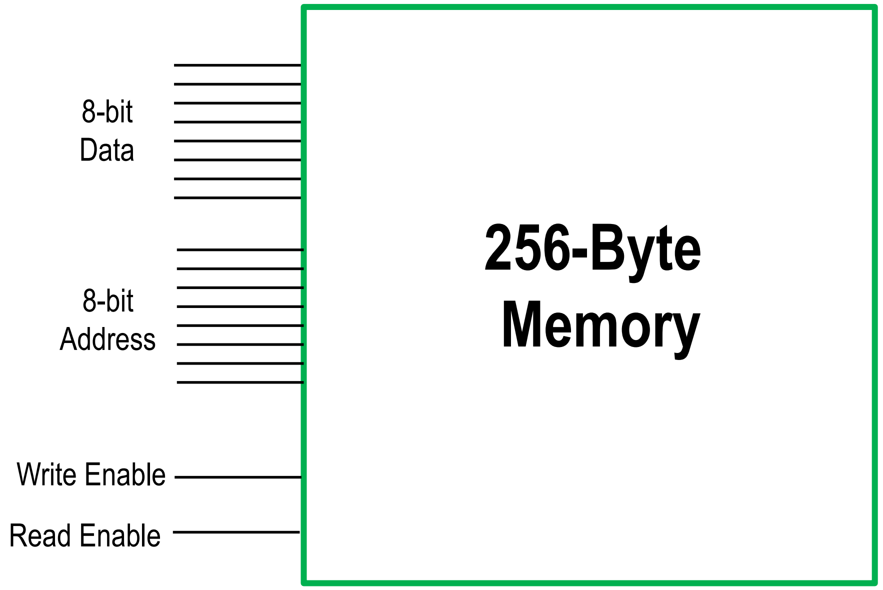
Let’s start out with a *very* simple system model



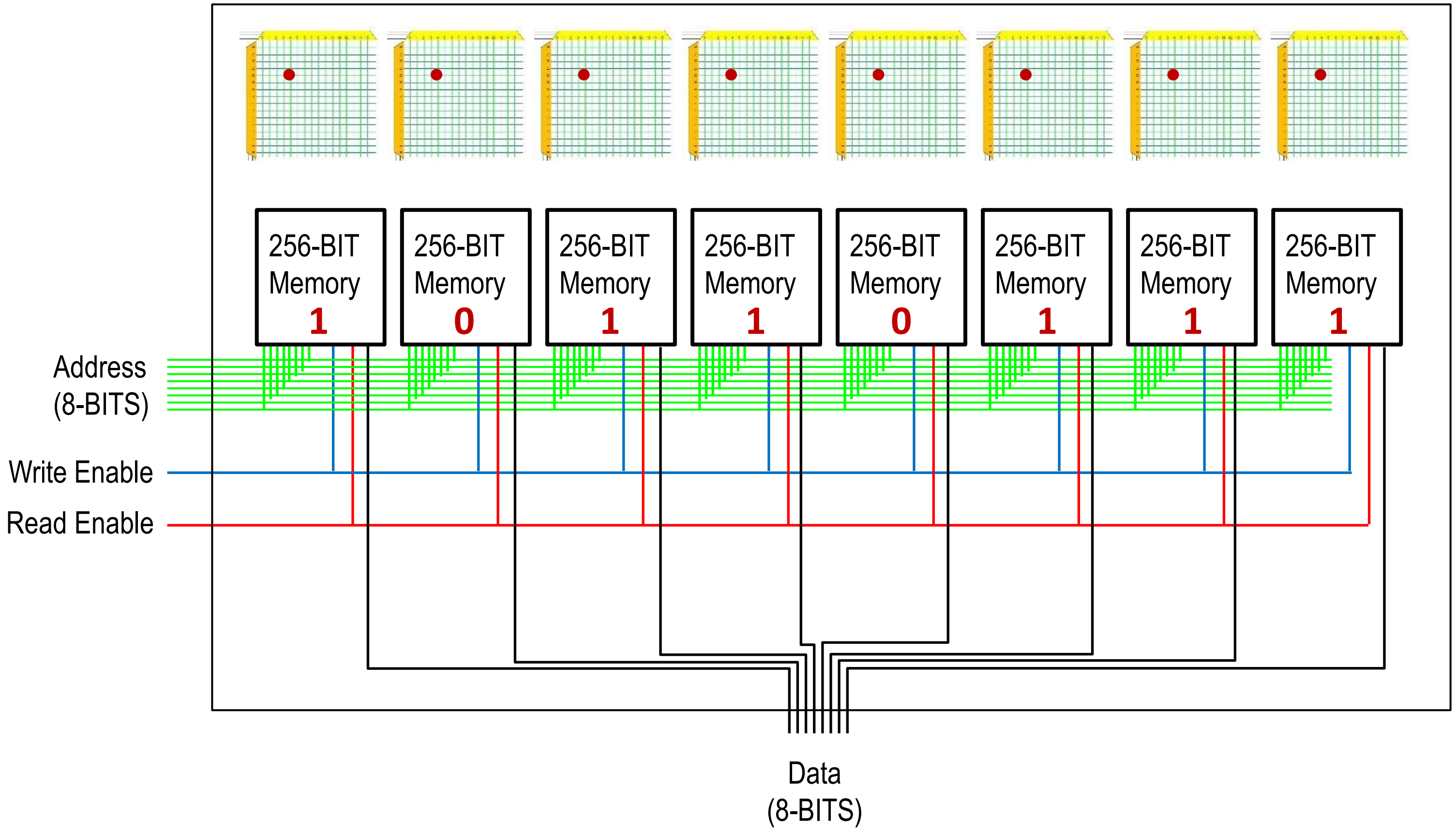
- Imagine a very simple world in which the goal of the central processing unit (CPU) is to execute a sequence of instructions to load values from memory, perform computation upon them, and write the results back to memory.

- Our world looks like the picture above:
  - The CPU controls the lines
  - The control lines tell RAM whether to read or write
  - The address lines tell RAM where to read or write
  - The data lines carry the data from the CPU to RAM or from RAM to the CPU as consistent with the control lines

We know what goes into the RAM box…



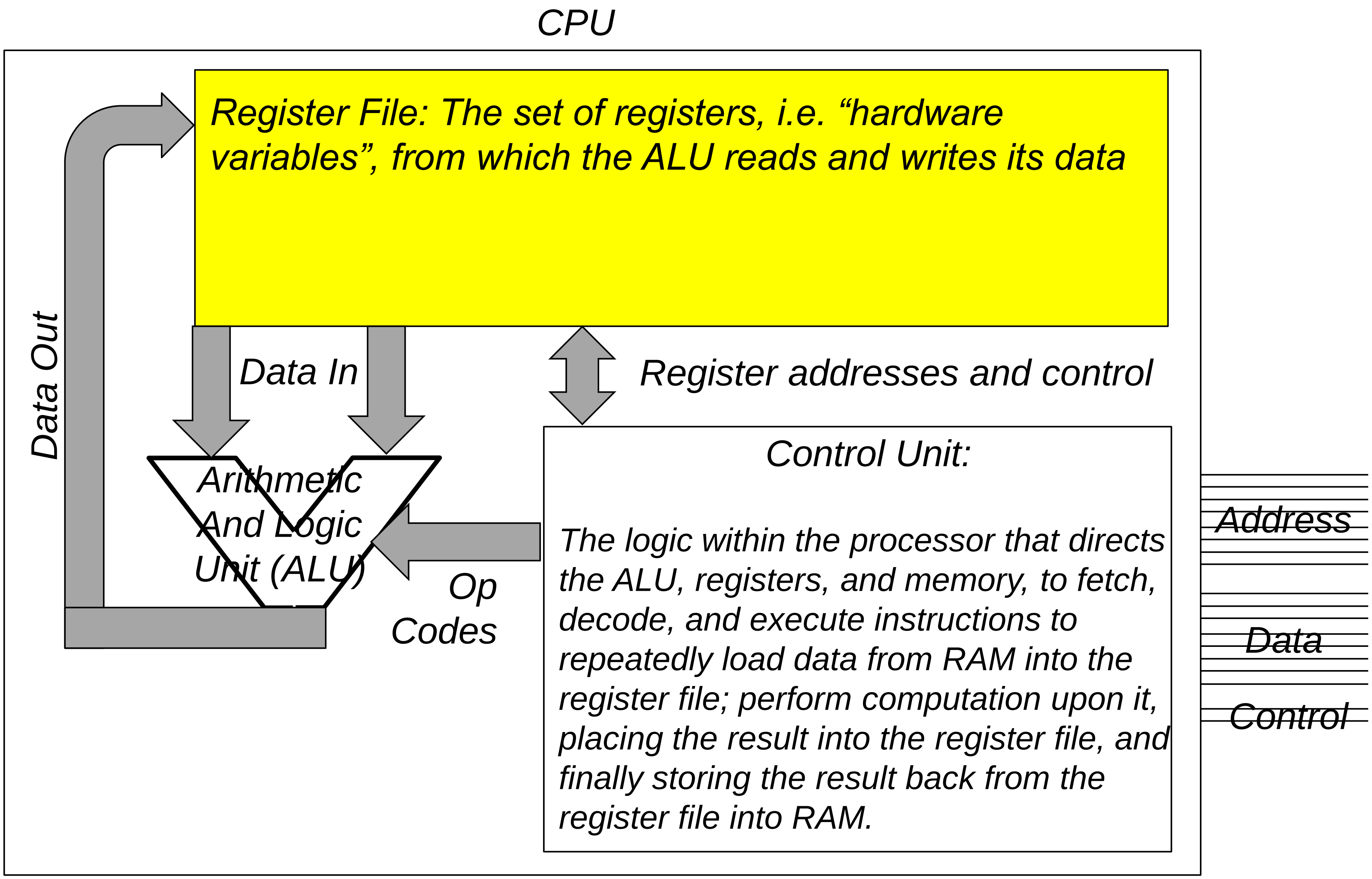
We just built this!



10110111

(Today we'll make use of a smaller 128 byte memory with 7 address bits)

And, we know what goes in this box. We just built it, too!

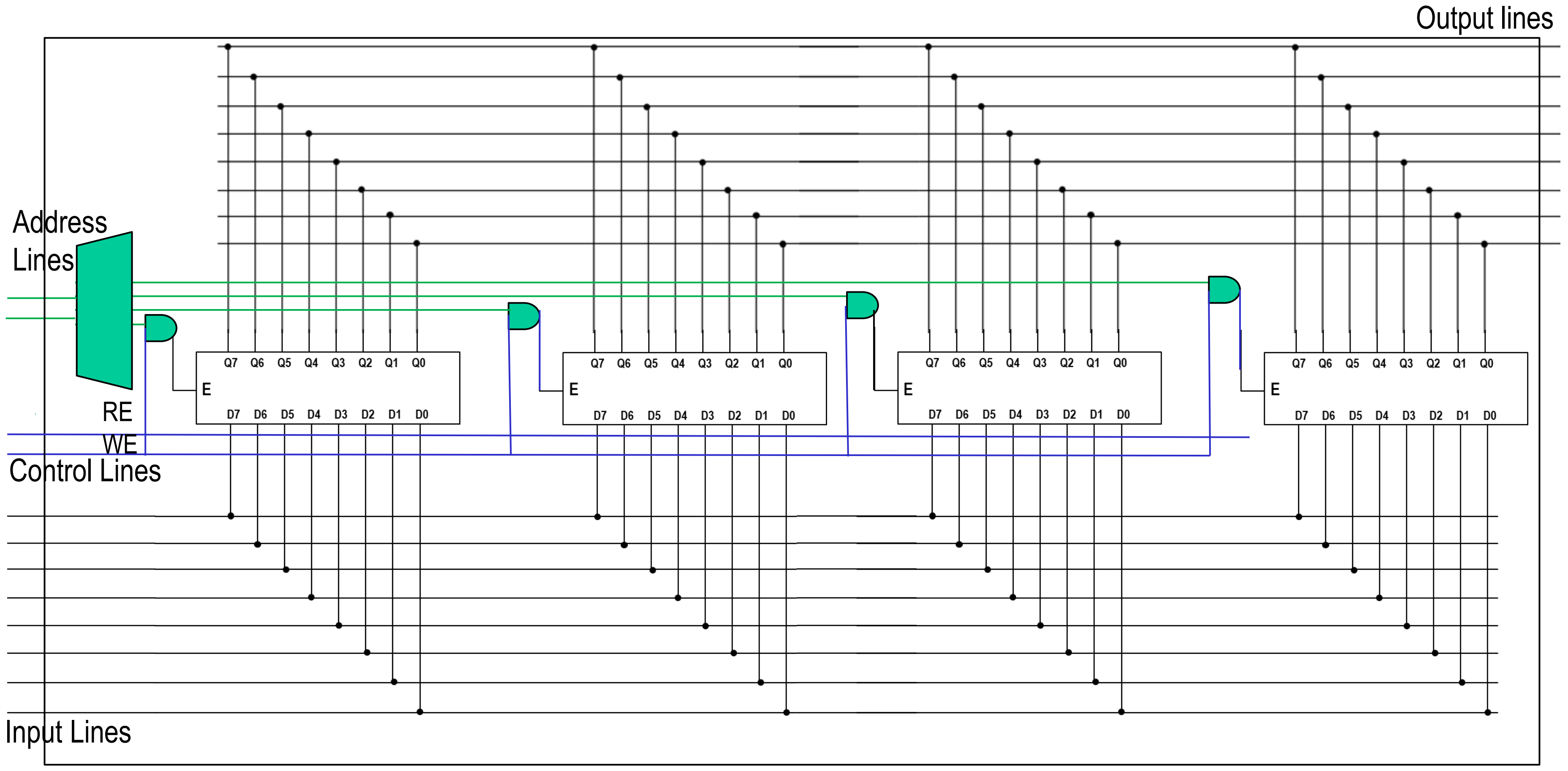

CPU
Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

Register File: The set of registers, i.e. "hardware variables", from which the ALU reads and writes its data

Control Unit:

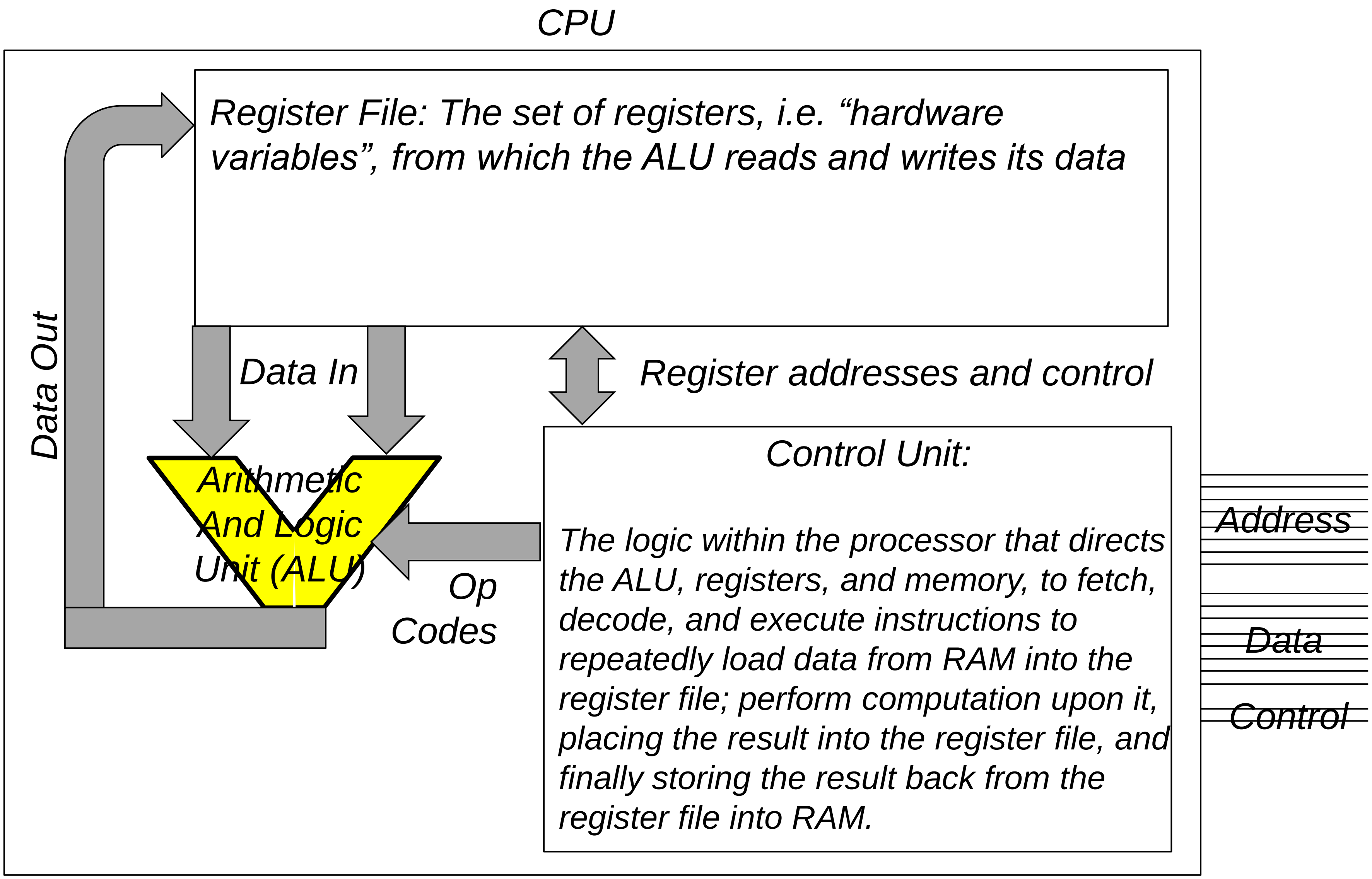
The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

We've designed the register file



- *We can now supply an address to select a register by number, in this case 0-3 in binary*
- *We can direct the register to read or write*
- *It will accept a value from the input lines or send one to the output lines*

So, what goes in this box?

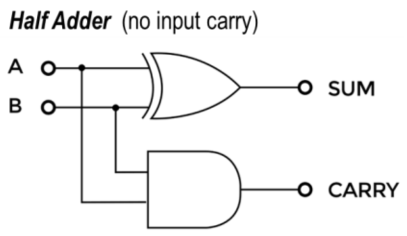

CPU
Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

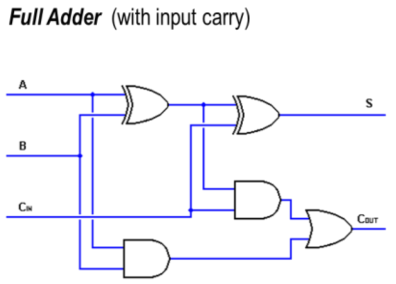
Register File: The set of registers, i.e. “hardware variables”, from which the ALU reads and writes its data

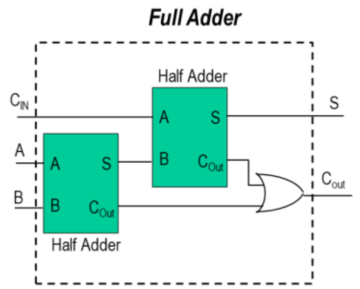
Control Unit:

The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

We know a *little* bit about what can go into the ALU

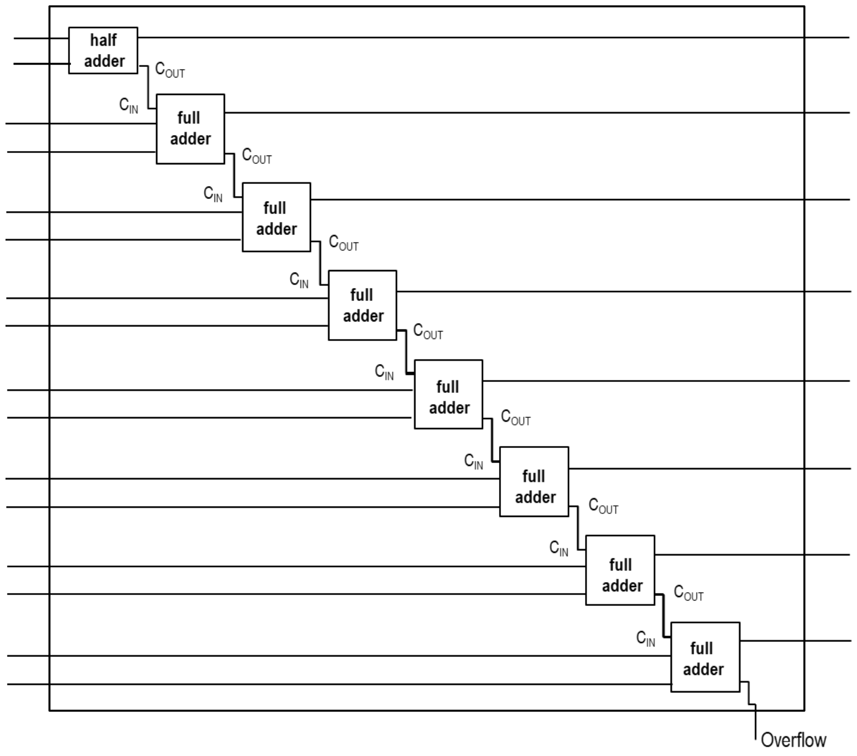






Input lines:

Notice two (2) input registers needed

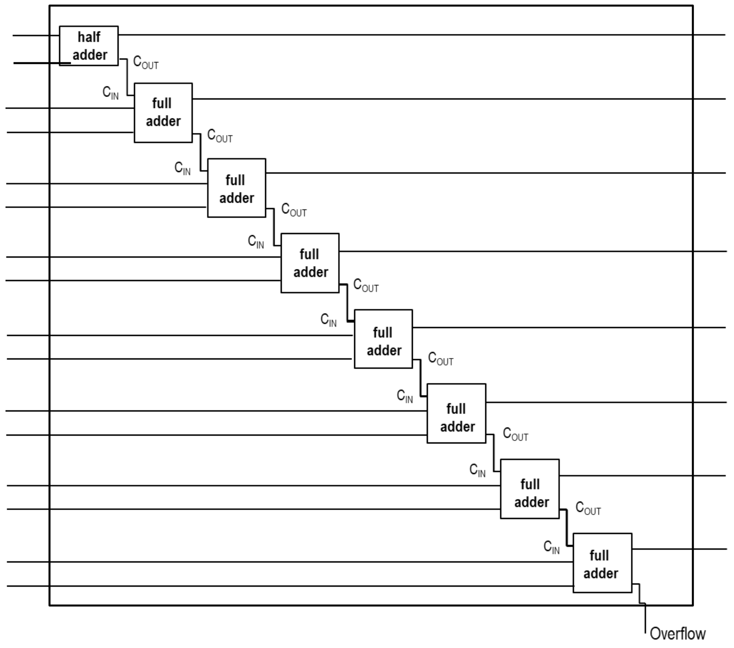


Output lines

How do we make use of three registers? Two input and one output?

Input lines:

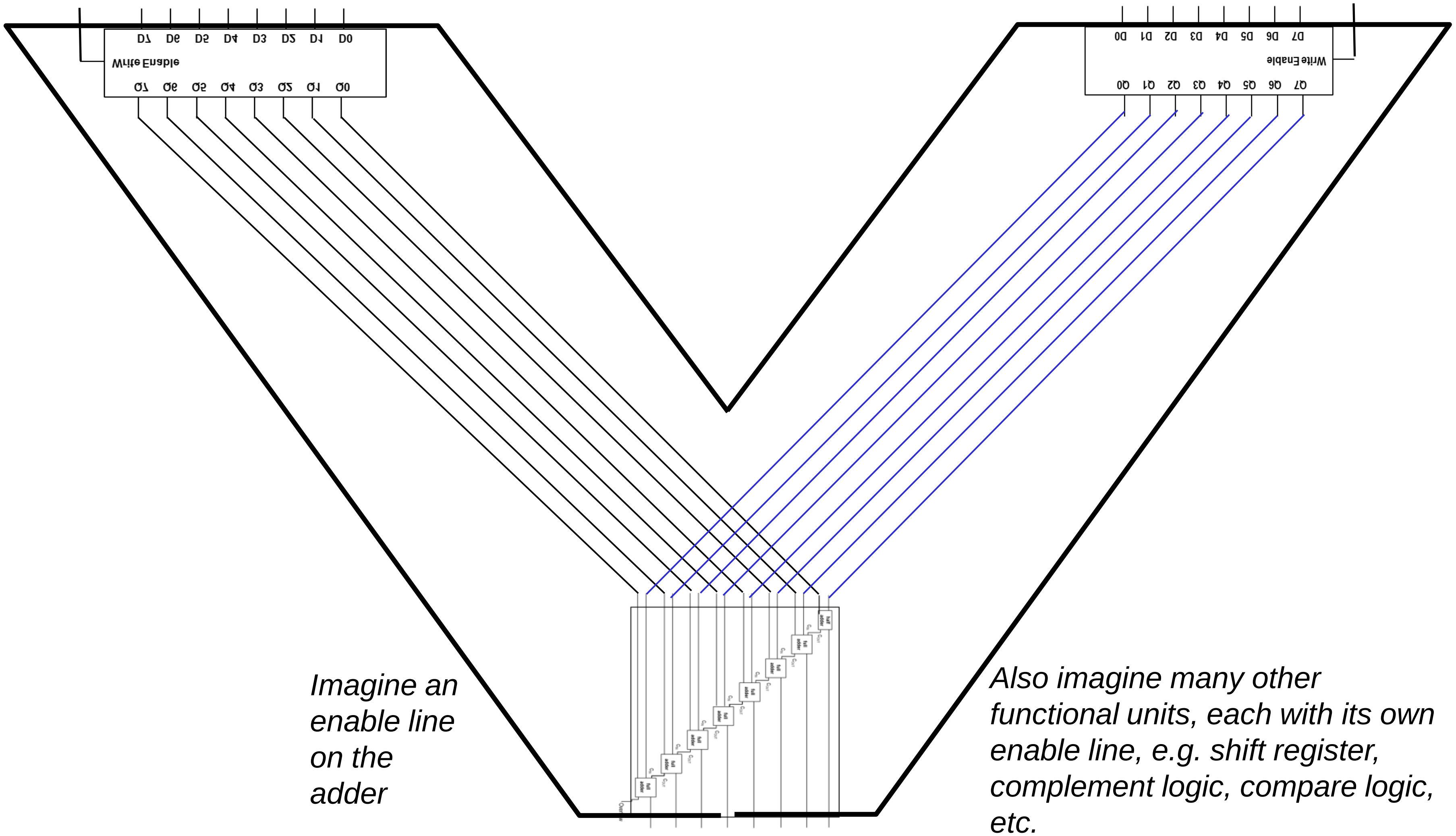
Notice two (2) input registers needed



Output lines:
Notice one register needed

- *The register files in real processors are often “multi-ported”, i.e. they can accept multiple addresses and access multiple registers in parallel, within some limits.*
- *We didn’t design our register file this way.*
- *We’ll assume that our ALU has internal “buffers” for the operands. These are just individual (no need for addressing) registers to hold the values being added.*
- *We’ll do our adds in three steps: Buffer operand 1 (BUF1), Buffer operand 2 (BUF2), and ADD, where the first two instructions get values into the ALU from registers and the ADD does the arithmetic and produces the output.*

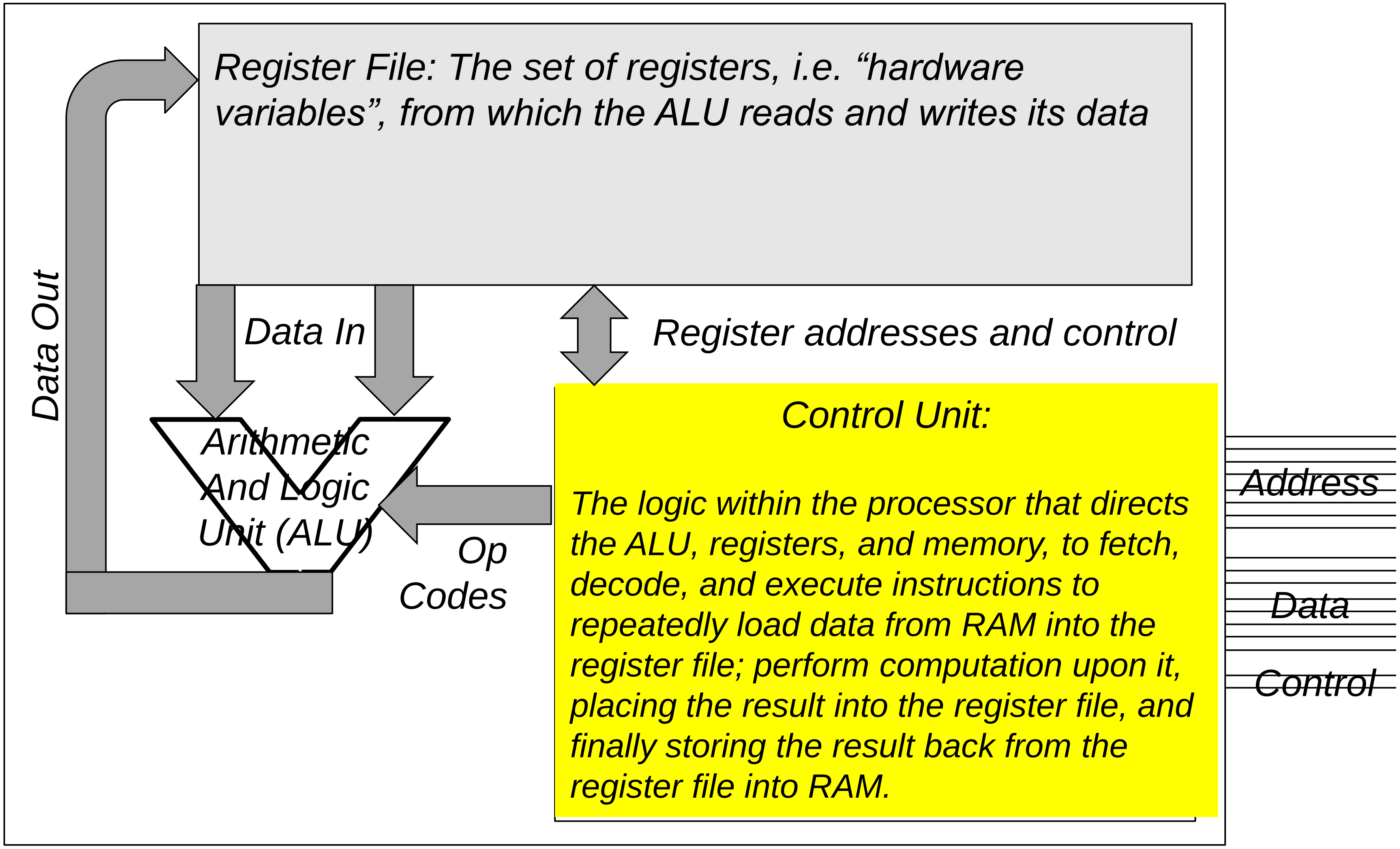
ALU: Our ALU with two internal buffers.



Imagine an enable line on the adder

Also imagine many other functional units, each with its own enable line, e.g. shift register, complement logic, compare logic, etc.

So, what goes in this box?

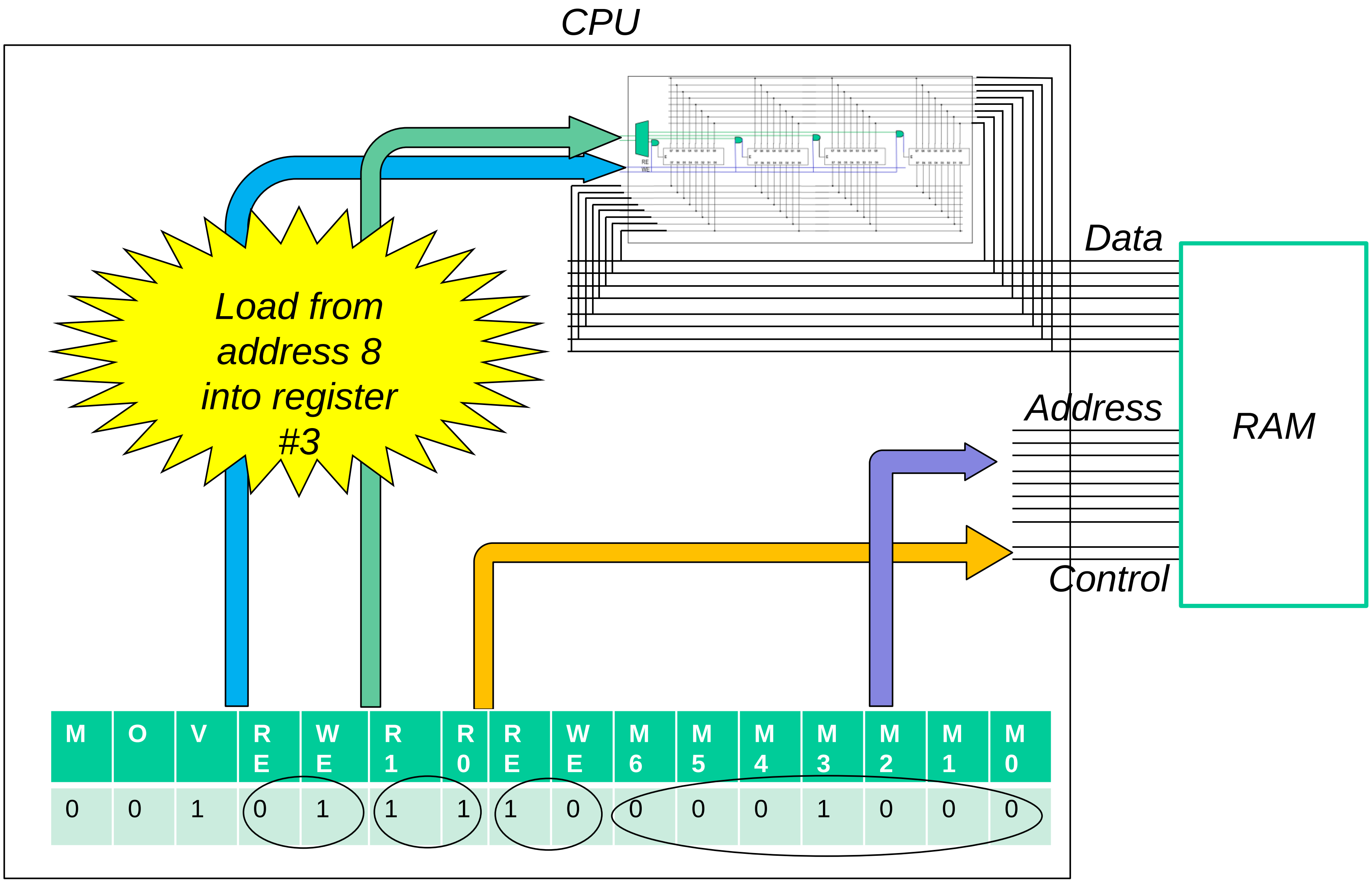

Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

Register File: The set of registers, i.e. "hardware variables", from which the ALU reads and writes its data

Control Unit:

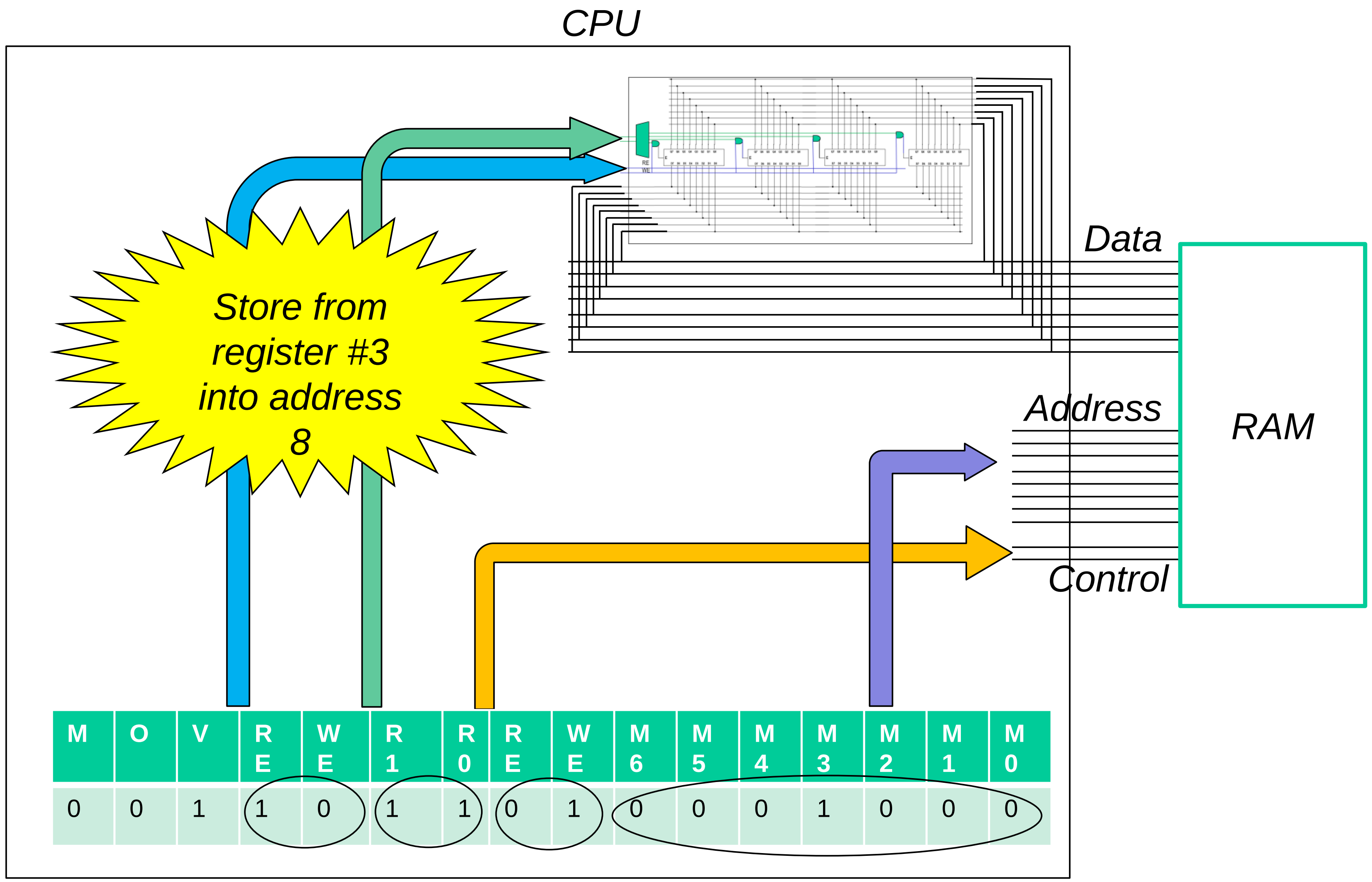
The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

How do we load a value from RAM into a register



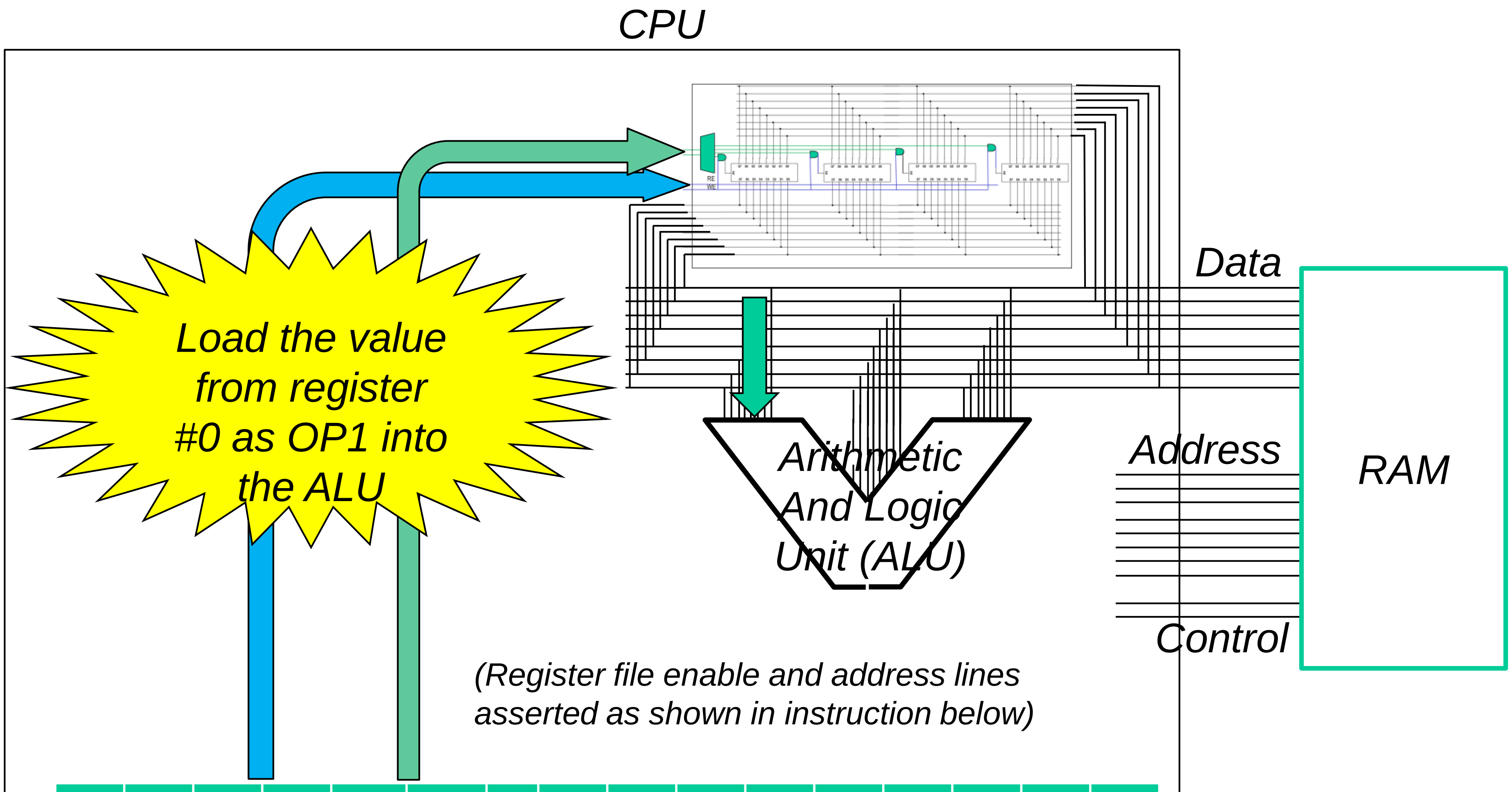
M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0

How do we store a value from a register into RAM



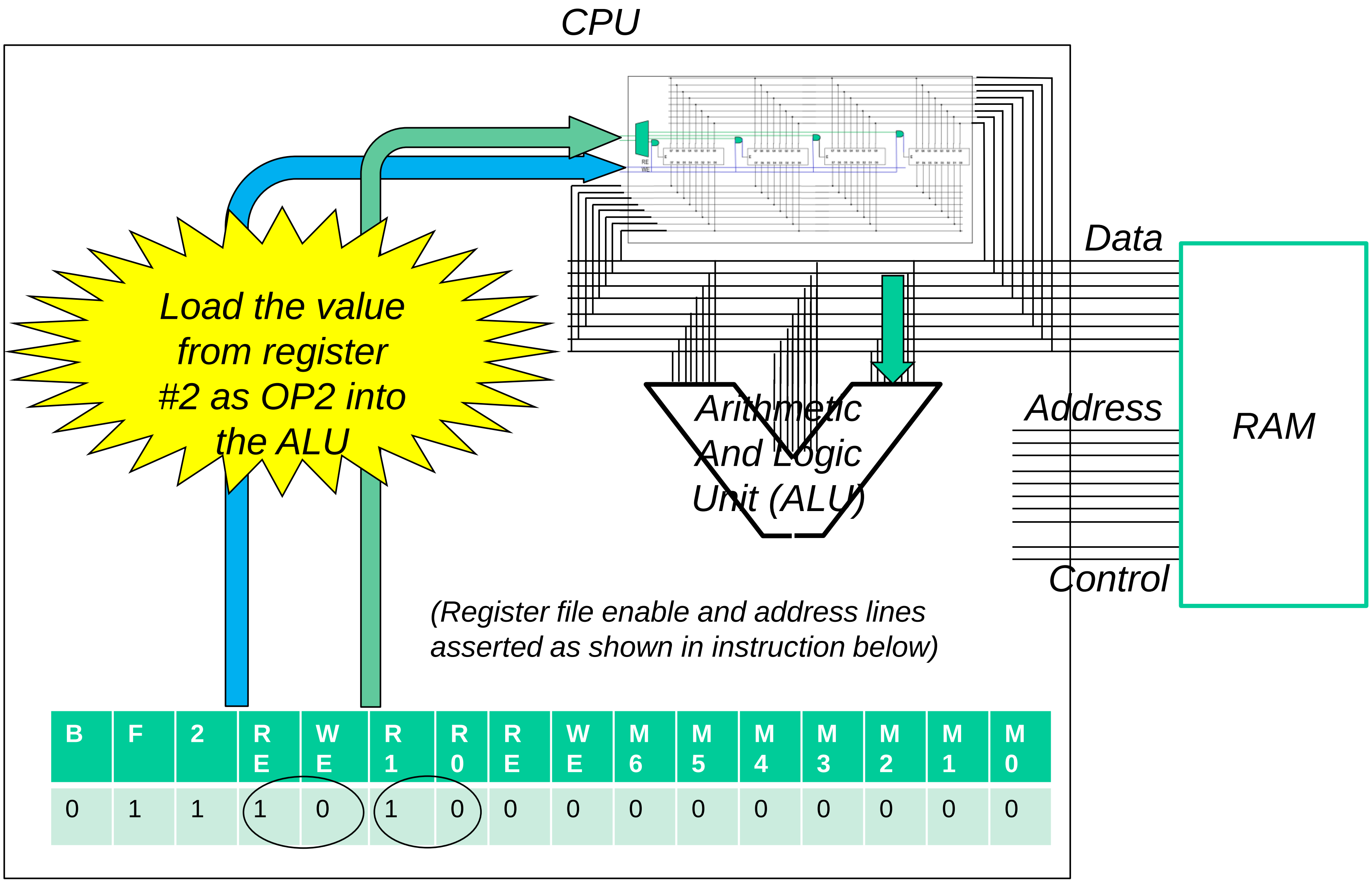
M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	1	0	1	1	0	1	0	0	0	1	0	0	0

ALU: Buffer Operand 1


CPU
Load the value from register #0 as OP1 into the ALU
Arithmetic And Logic Unit (ALU)
Data
Address
Control
RAM
(Register file enable and address lines asserted as shown in instruction below)

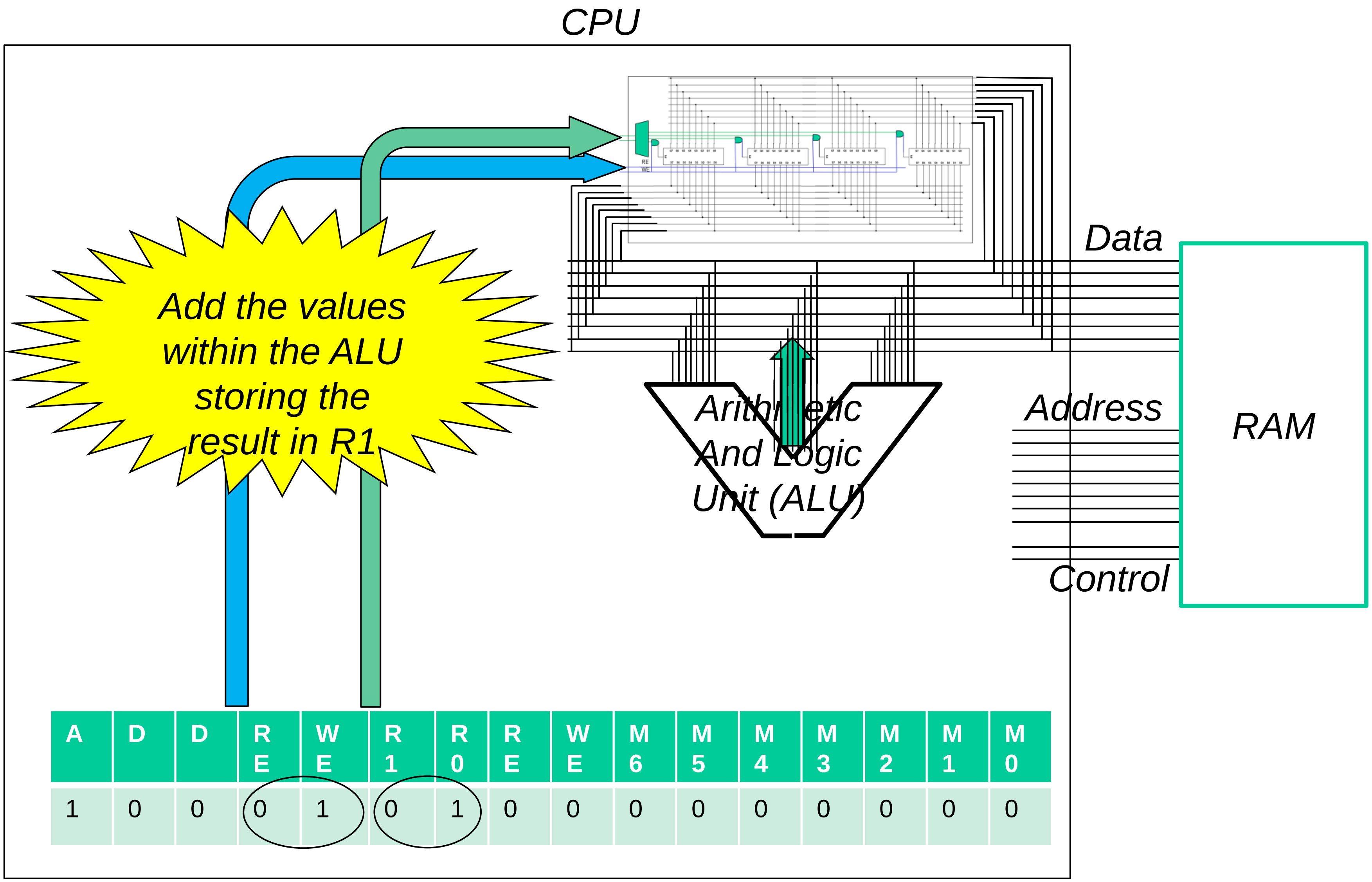
B	F	1	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

ALU: Buffer Operand 2


CPU
Data
Address
Control
RAM
Arithmetic And Logic Unit (ALU)
Load the value from register #2 as OP2 into the ALU
(Register file enable and address lines asserted as shown in instruction below)

B	F	2	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0

ALU: Do the Add


CPU
Add the values within the ALU storing the result in R1
Arithmetic And Logic Unit (ALU)
Data
Address
Control
RAM
RE
WE

A	D	D	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

What can our CPU do (so far)?

M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0

MOV %R1, MEM8

M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	1	0	1	1	0	1	0	0	0	1	0	0	0

MOV MEM8, %R3

B	F	1	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

BUF1 %R0

B	F	2	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0

BUF2 %R2

A	D	D	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

ADD %R1

But, How do we choose? How do we act upon these bits?

M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0

MOV %R1, MEM8

M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	1	0	1	1	0	1	0	0	0	1	0	0	0

MOV MEM8, %R3

B	F	1	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

BUF1 %R0

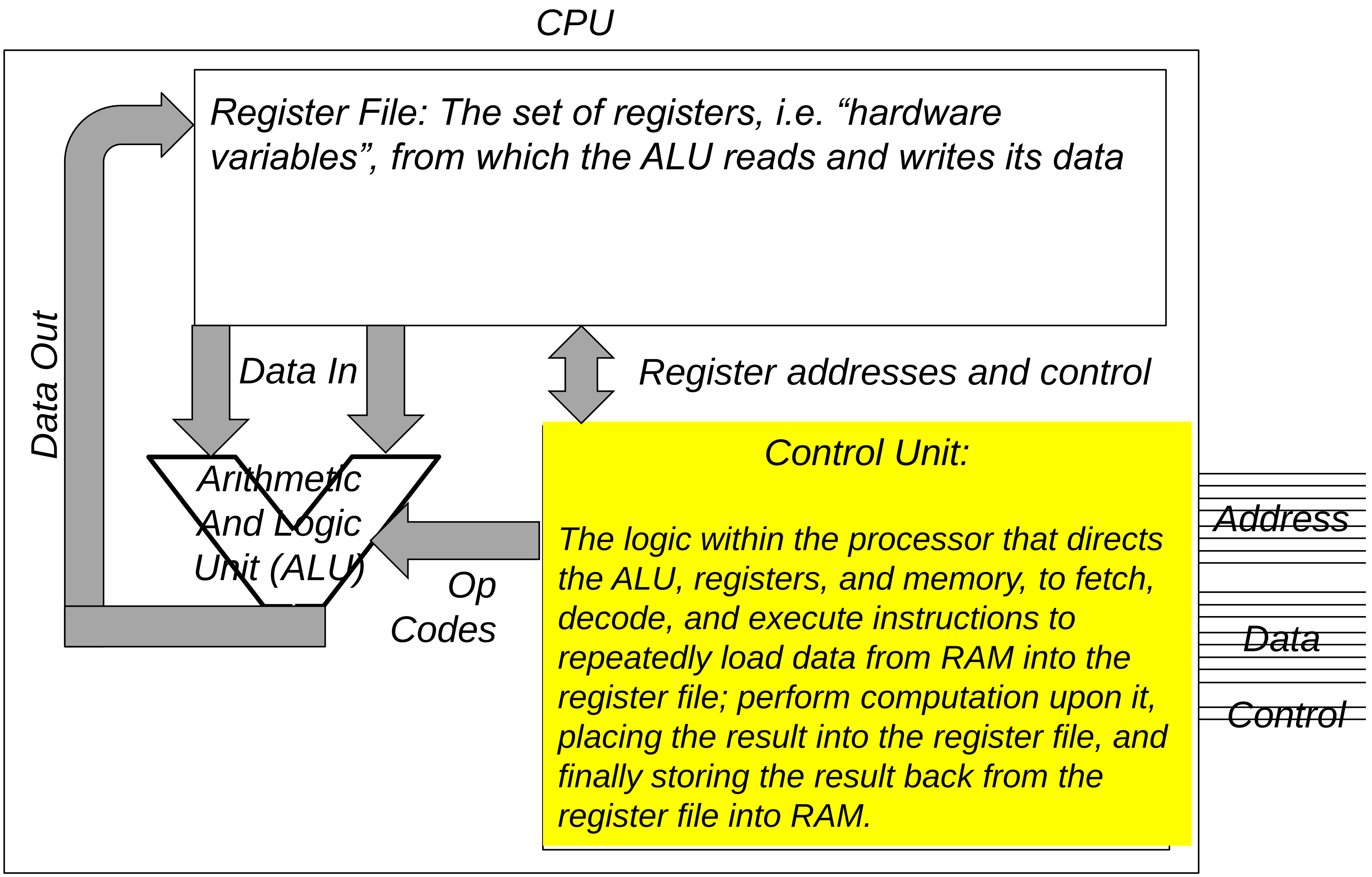
B	F	2	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0

BUF2 %R2

A	D	D	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

ADD %R1

CPU: A Block Diagram

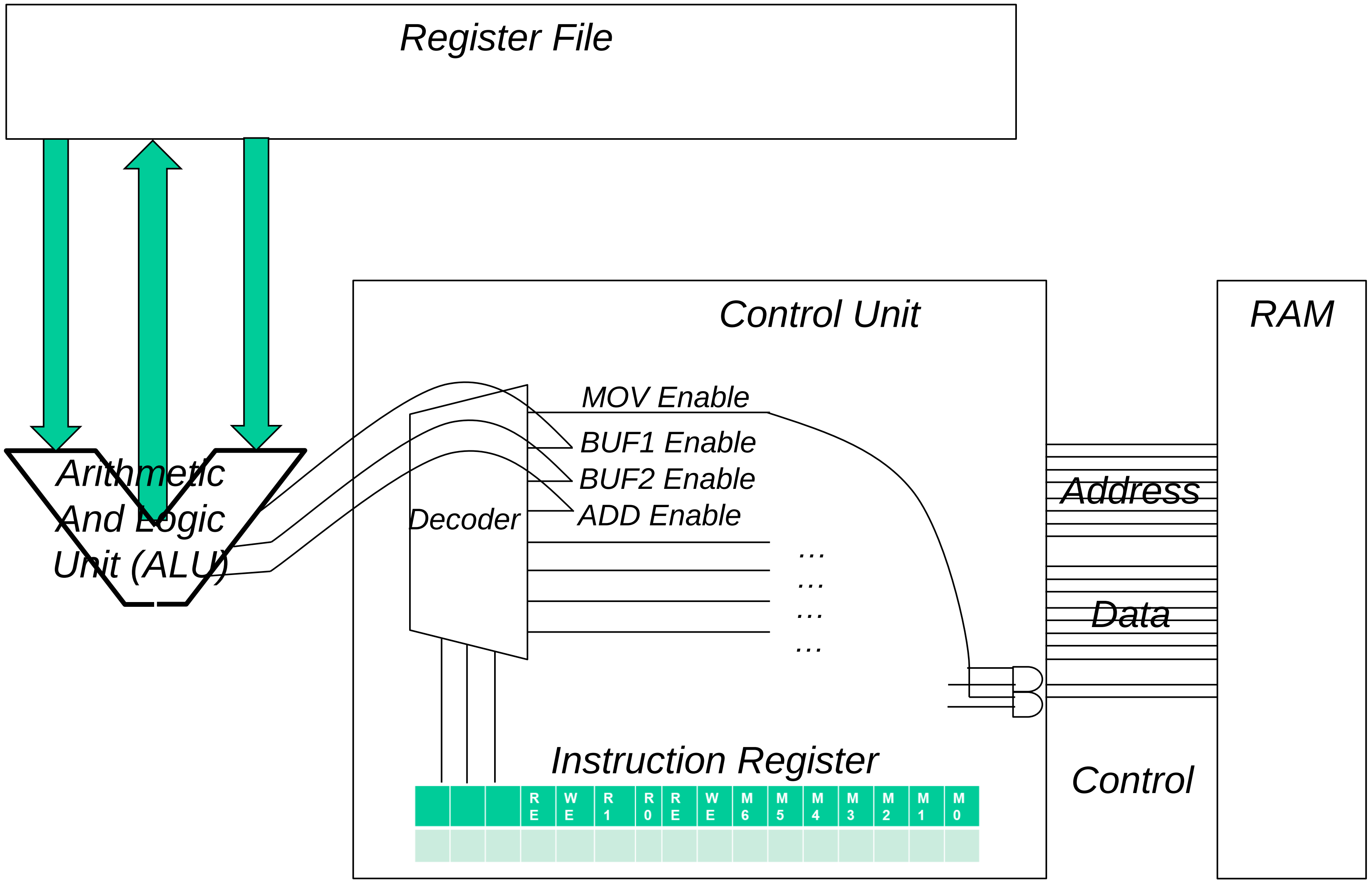

CPU
Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

Register File: The set of registers, i.e. "hardware variables", from which the ALU reads and writes its data

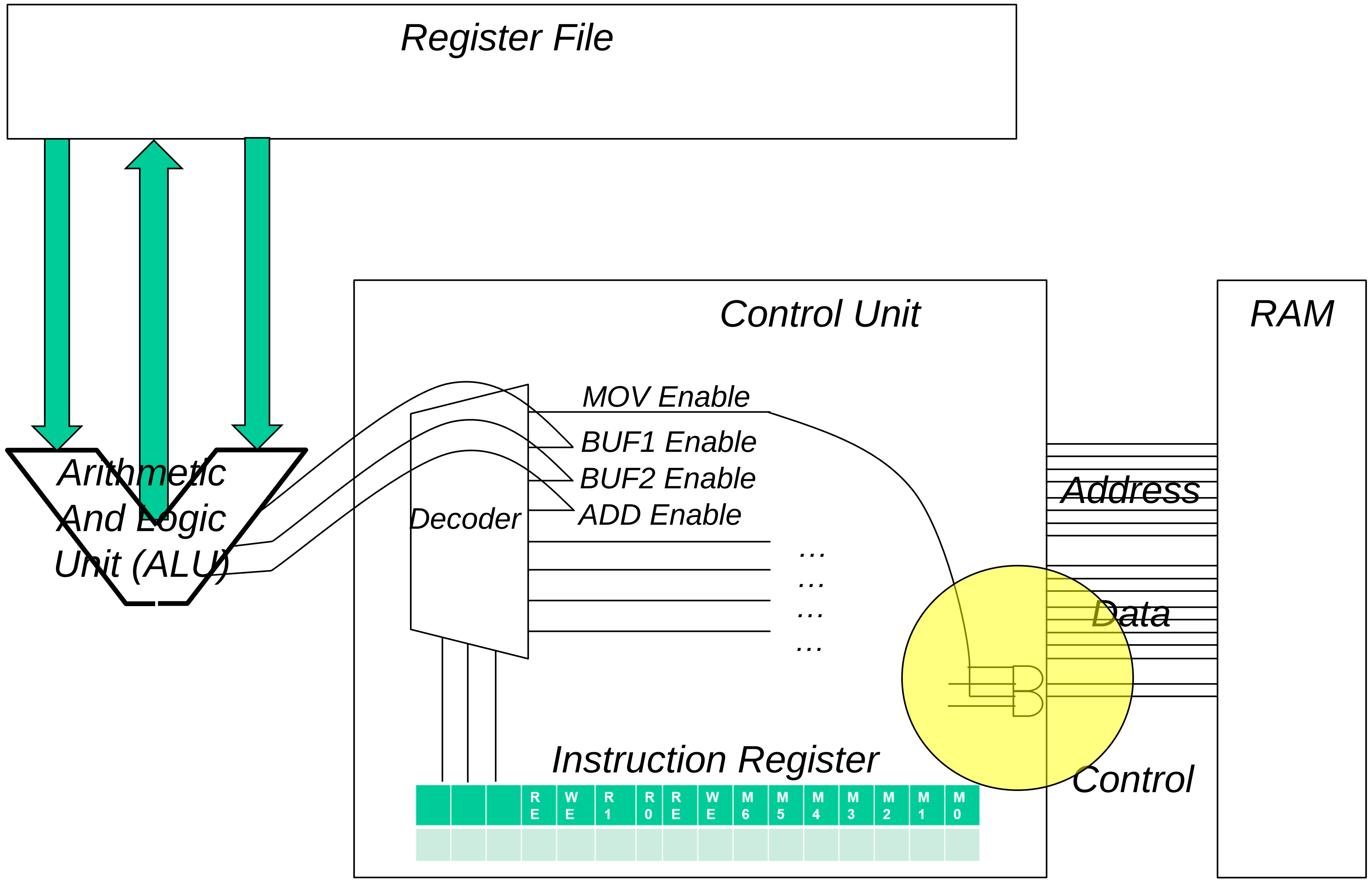
Control Unit:

The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

Control: Decoding an Instruction



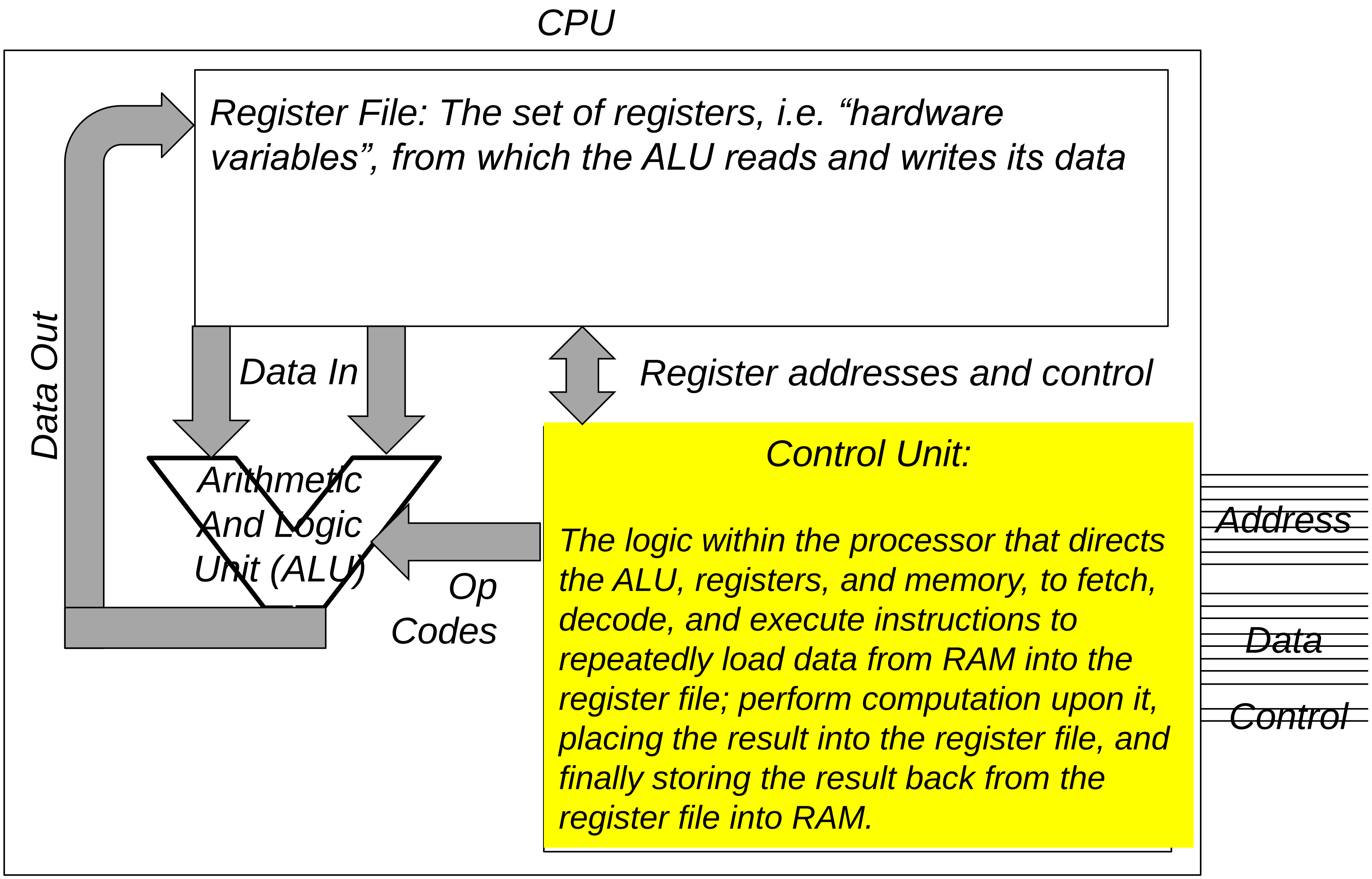
Control: Decoding an Instruction



Control: Enabling/Disabling the ALU operations

- *Enable line can switch to enable or disable power to the circuits and/or isolate the inputs and outputs.*
- *Consider for example using MOSFETs (our new friends).*

CPU: A Block Diagram (Let’s keep talking about control)


CPU
Data Out
Data In
Arithmetic And Logic Unit (ALU)
Op Codes
Register addresses and control
Address
Data
Control

Register File: The set of registers, i.e. “hardware variables”, from which the ALU reads and writes its data

Control Unit:

The logic within the processor that directs the ALU, registers, and memory, to fetch, decode, and execute instructions to repeatedly load data from RAM into the register file; perform computation upon it, placing the result into the register file, and finally storing the result back from the register file into RAM.

How do we get an instruction into the instruction register?

F	T	C	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FETCH MEM0

M	O	V	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0

MOV %R1, MEM8

M	O	V	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	0	1	1	0	1	1	0	1	0	0	0	1	0	0	0

MOV MEM8, %R3

B	F	1	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

BUF1 %R0

B	F	2	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0

BUF2 %R2

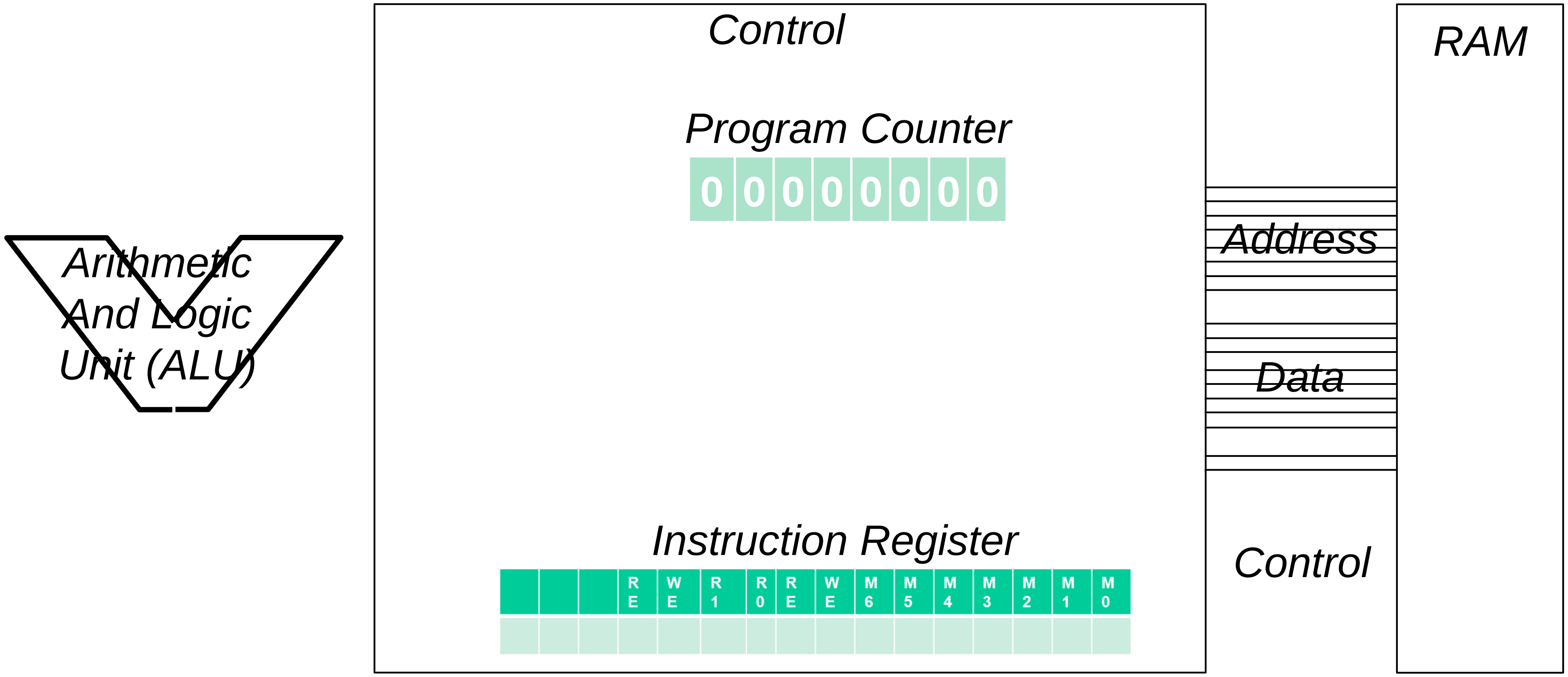
A	D	D	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

ADD %R1

How does an instruction get loaded and executed?

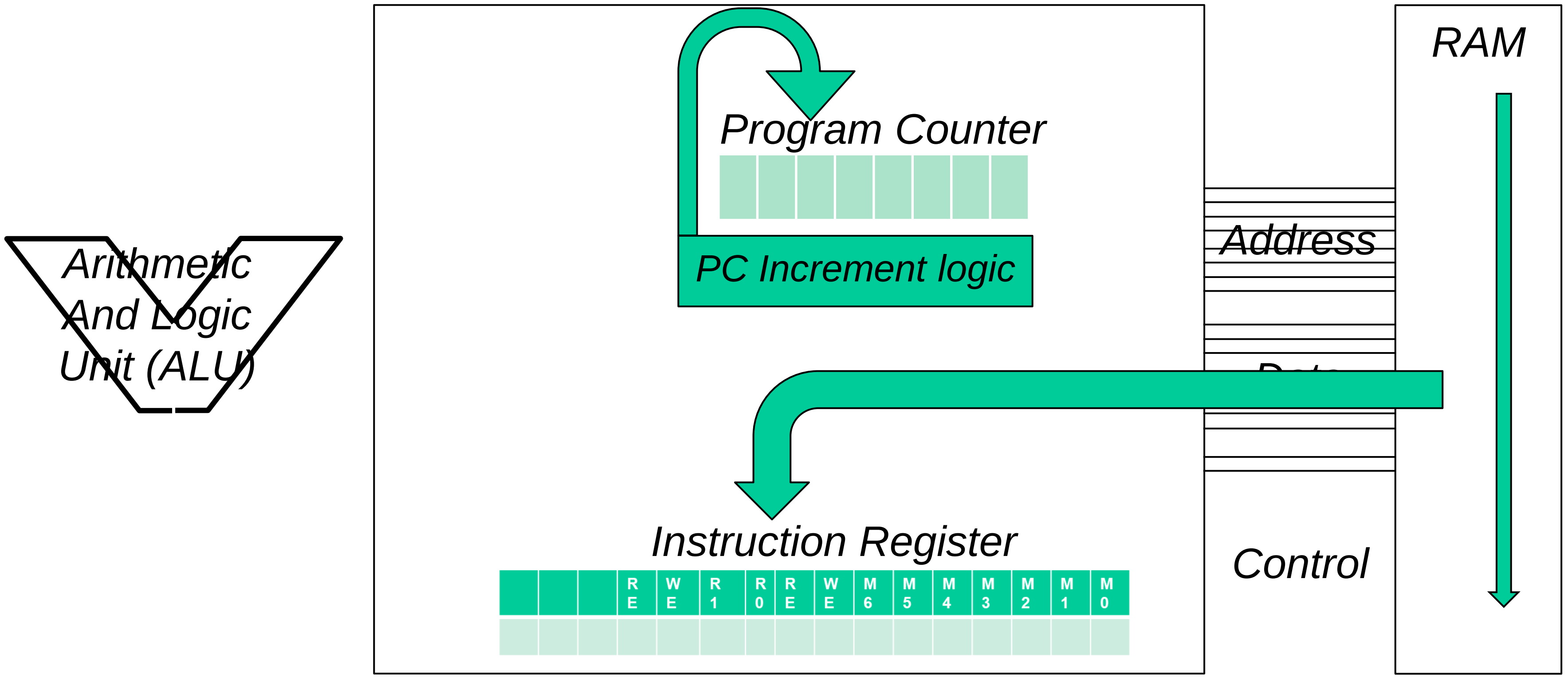
Register File

There is a special purpose register, the program counter, a.k.a instruction pointer, that keeps track of the address in RAM of the next instruction to execute. At boot, it is initialized to an established address.

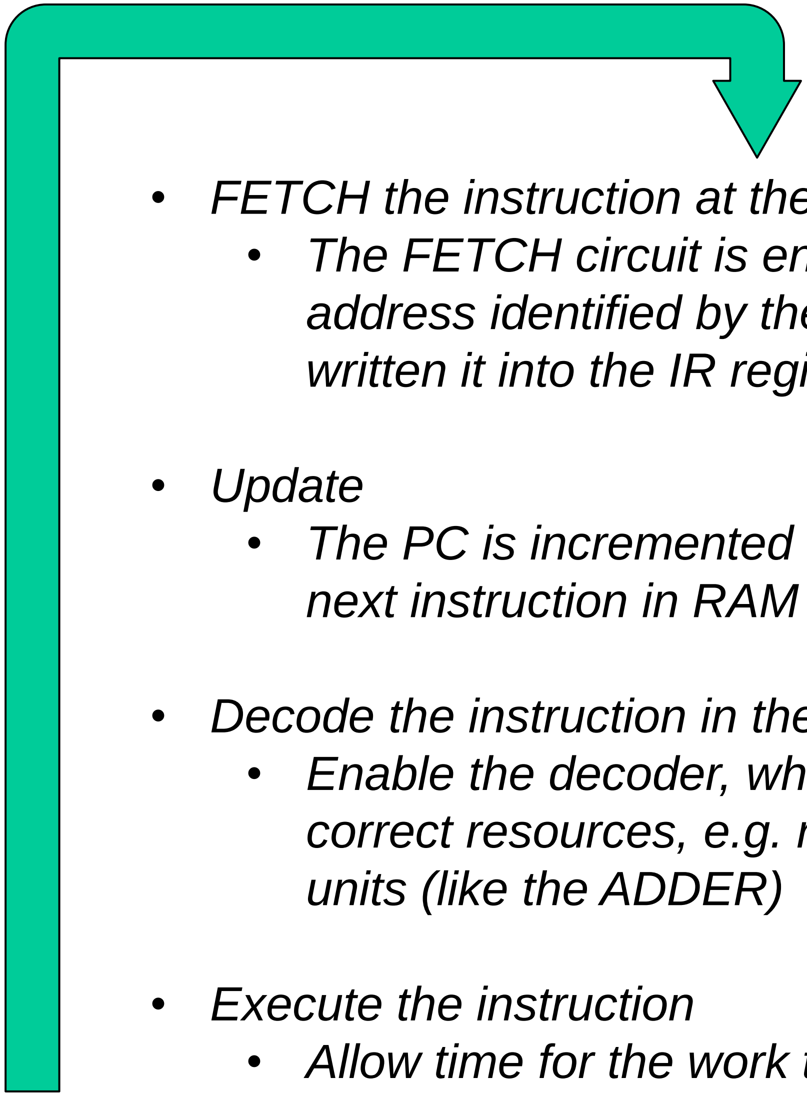


How does an instruction get loaded and executed?

The processor is hard-wired to FETCH the instruction from the PC into the IR, and then increment the PC such that it points to the next instruction in RAM.



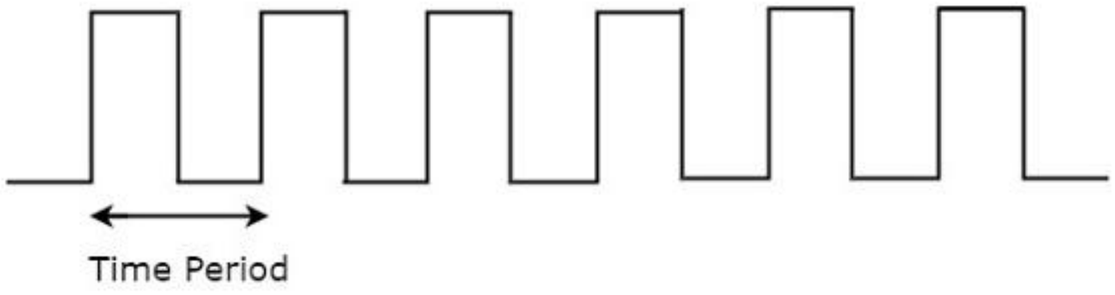
Processor Execution Cycle



- *FETCH the instruction at the address given by the PC*
  - *The FETCH circuit is enabled: The memory address identified by the PC is read from RAM and written it into the IR register*
- *Update*
  - *The PC is incremented such that it points to the next instruction in RAM (to be ready for next time)*
- *Decode the instruction in the IR*
  - *Enable the decoder, which in turn enables the correct resources, e.g. registers, RAM, functional units (like the ADDER)*
- *Execute the instruction*
  - *Allow time for the work to be done and results to settle in registers or RAM*

What drives the cycle?

- *A CLOCK generates a square wave (high, low, high, low, …) at a fixed interval*
- *By observing this clock signal, different parts of the CPU can do things at the right time, in coordination with each other.*
- *O, let's imagine a simple model where a clock signal is used to time our four events*



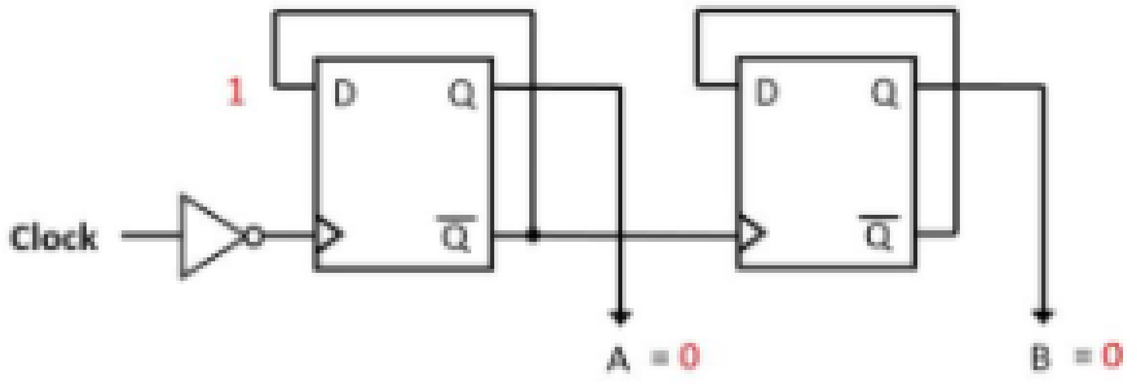
How do we get from a clock signal to coordinating events?

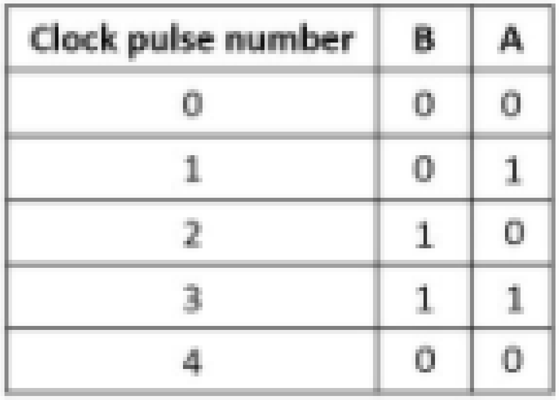
- We want to be able to cycle through four states:
  - 00 Fetch
  - 01 Update
  - 10 Decode
  - 11 Execute

- So, we can use a 2-bit adder:
  - We start at 00
  - We add 1 each clock cycle
  - When it gets to 11 and increments, it resets to 0
  - Then it repeats from there

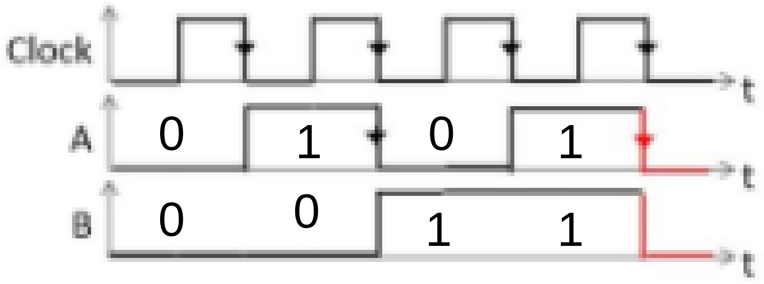
How to build a 2-bit adder?

- *We can use 2 flip-flops*





Clock pulse number	B	A
0	0	0
1	0	1
2	1	0
3	1	1
4	0	0

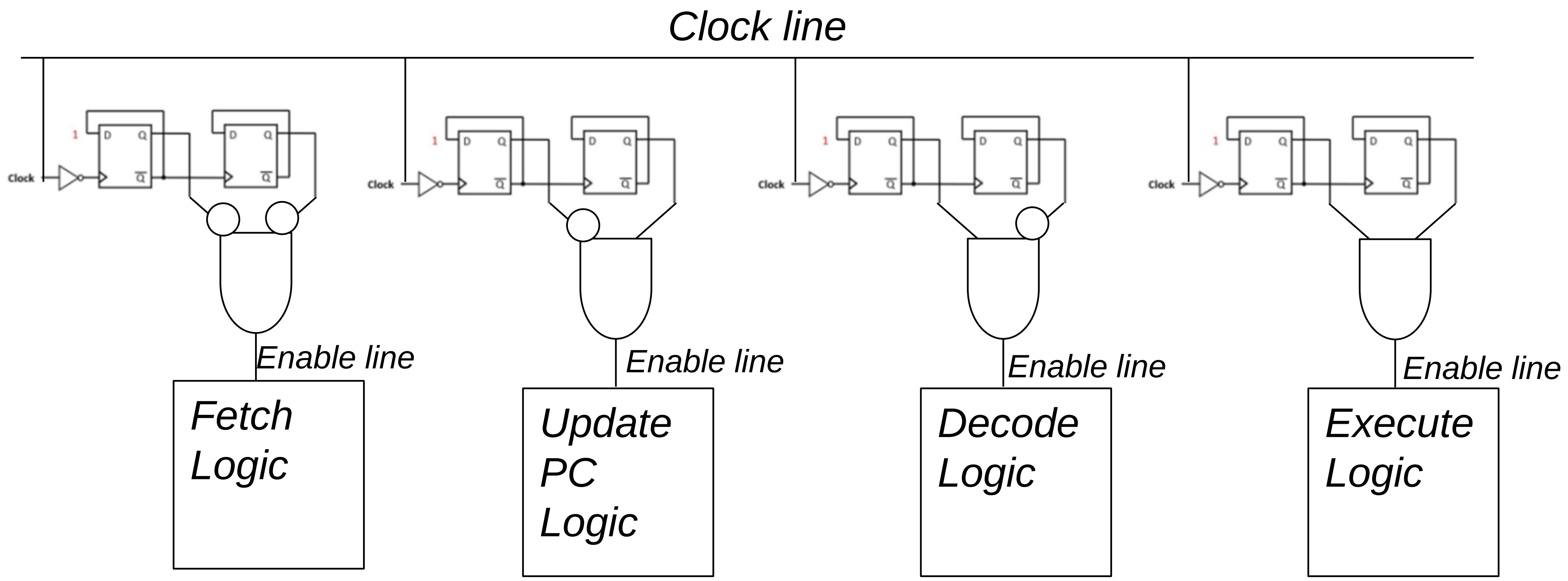


https://www.quora.com/How-do-I-design-a-2-bit-up-down-counter-using-d-flip-flop

Now, how do we coordinate our processor?

- *We want to be able to cycle through four states:*
  - *00 Fetch*
  - *01 Update*
  - *10 Decode*
  - *11 Execute*

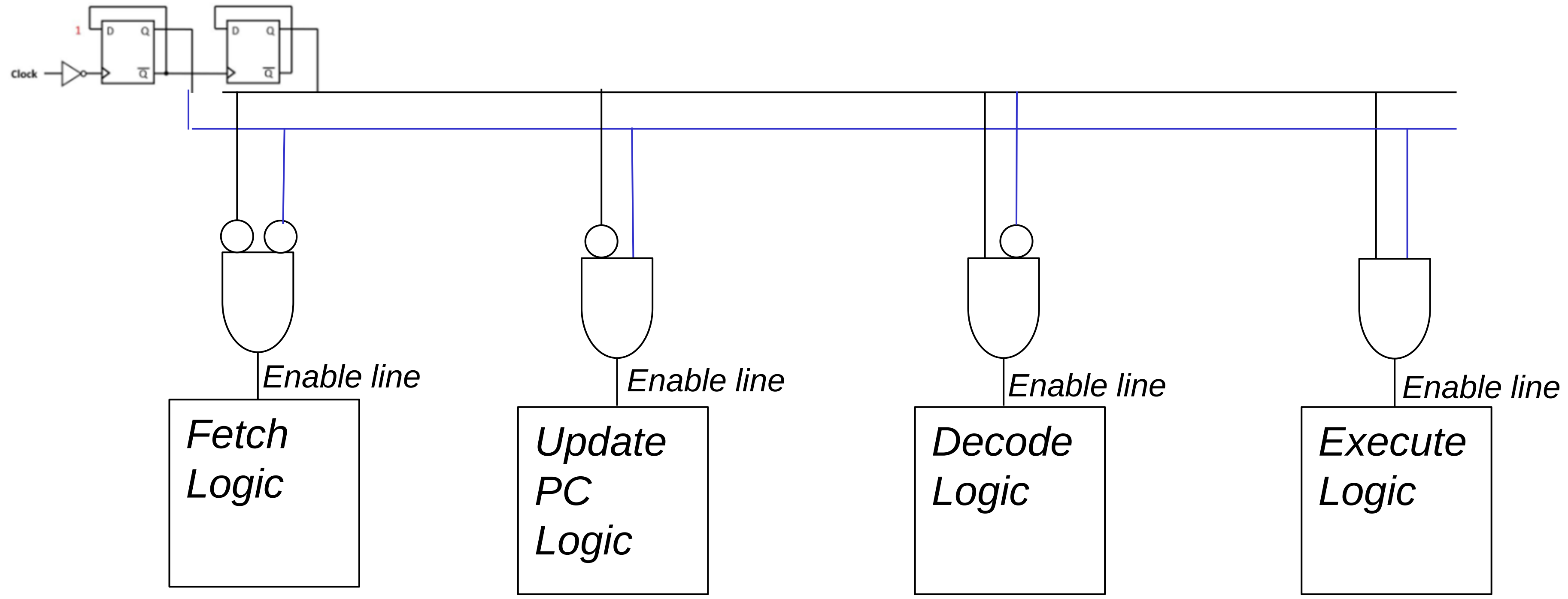
- *Fetch_enable =* $\overline{A}$ *&* $\overline{B}$
- *Update_enable =* $\overline{A}$ *& B*
- *Decode_enable = A &* $\overline{B}$
- *Execute_enable = A & B*



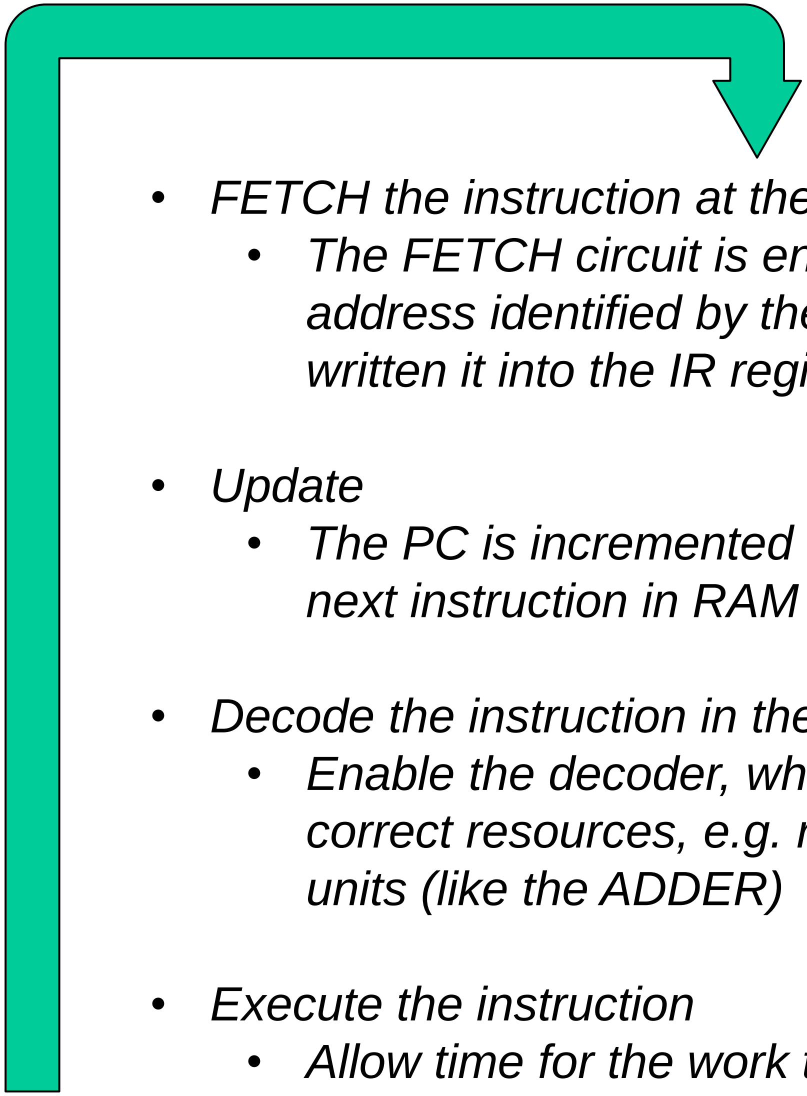
Alternative design: Now, how do we coordinate our processor?

- *We want to be able to cycle through four states:*
  - *00 Fetch*
  - *01 Update*
  - *10 Decode*
  - *11 Execute*

- *Fetch_enable =* $\overline{A}$ *&* $\overline{B}$
- *Update_enable =* $\overline{A}$ *& B*
- *Decode_enable = A &* $\overline{B}$
- *Execute_enable = A & B*



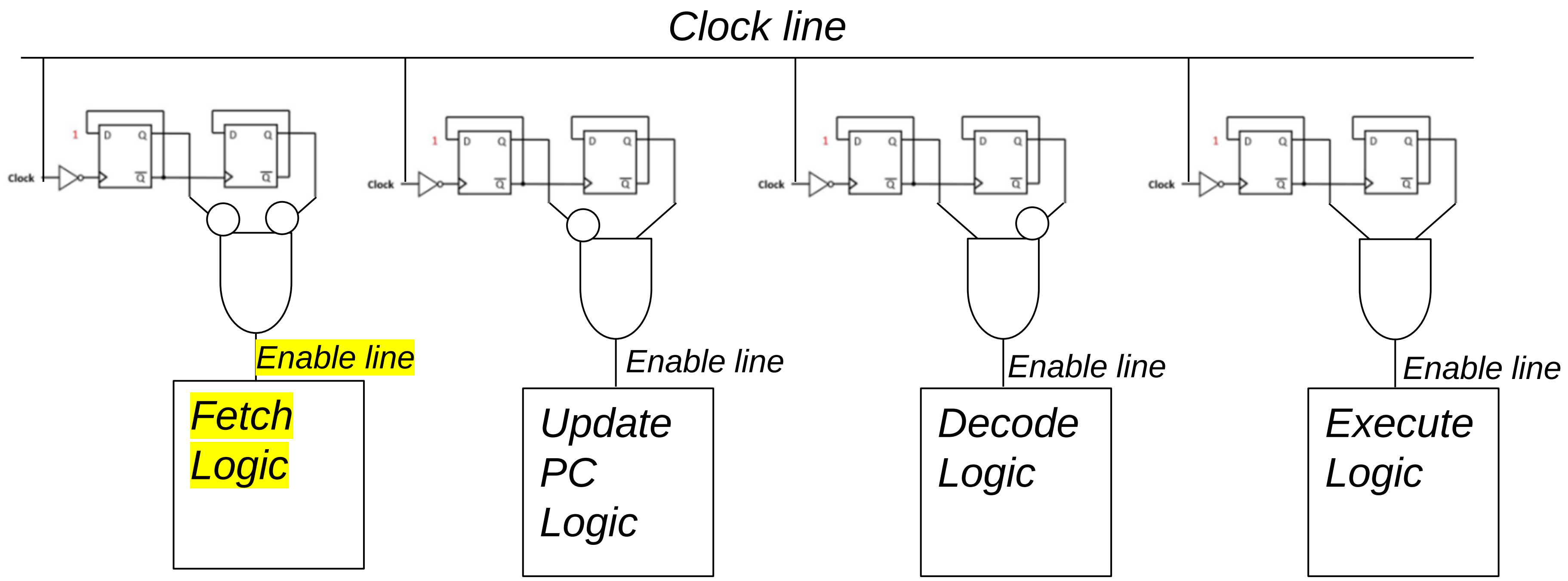
Let’s do it!



- *FETCH the instruction at the address given by the PC*
  - *The FETCH circuit is enabled: The memory address identified by the PC is read from RAM and written it into the IR register*
- *Update*
  - *The PC is incremented such that it points to the next instruction in RAM (to be ready for next time)*
- *Decode the instruction in the IR*
  - *Enable the decoder, which in turn enables the correct resources, e.g. registers, RAM, functional units (like the ADDER)*
- *Execute the instruction*
  - *Allow time for the work to be done and results to settle in registers or RAM*

Adder state is 00: Instruction referenced by PC is fetched into IR

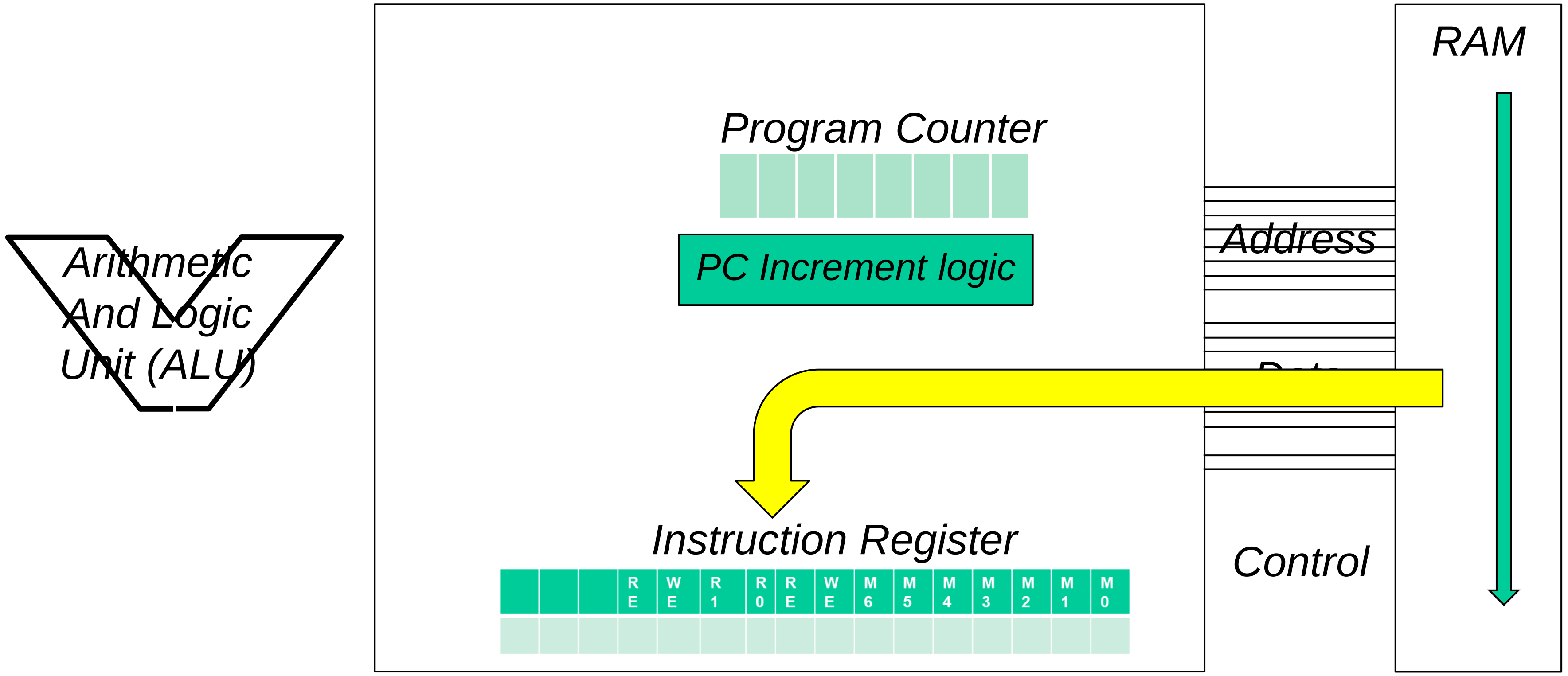
- *We want to be able to cycle through four states:*
    - *00 Fetch*
    - *01 Update*
    - *10 Decode*
    - *11 Execute*
- *Fetch_enable =* $\overline{A}$ *&* $\overline{B}$
- *Update_enable =* $\overline{A}$ *& B*
- *Decode_enable = A &* $\overline{B}$
- *Execute_enable = A & B*



00: Fetch circuit is enabled.

Register File

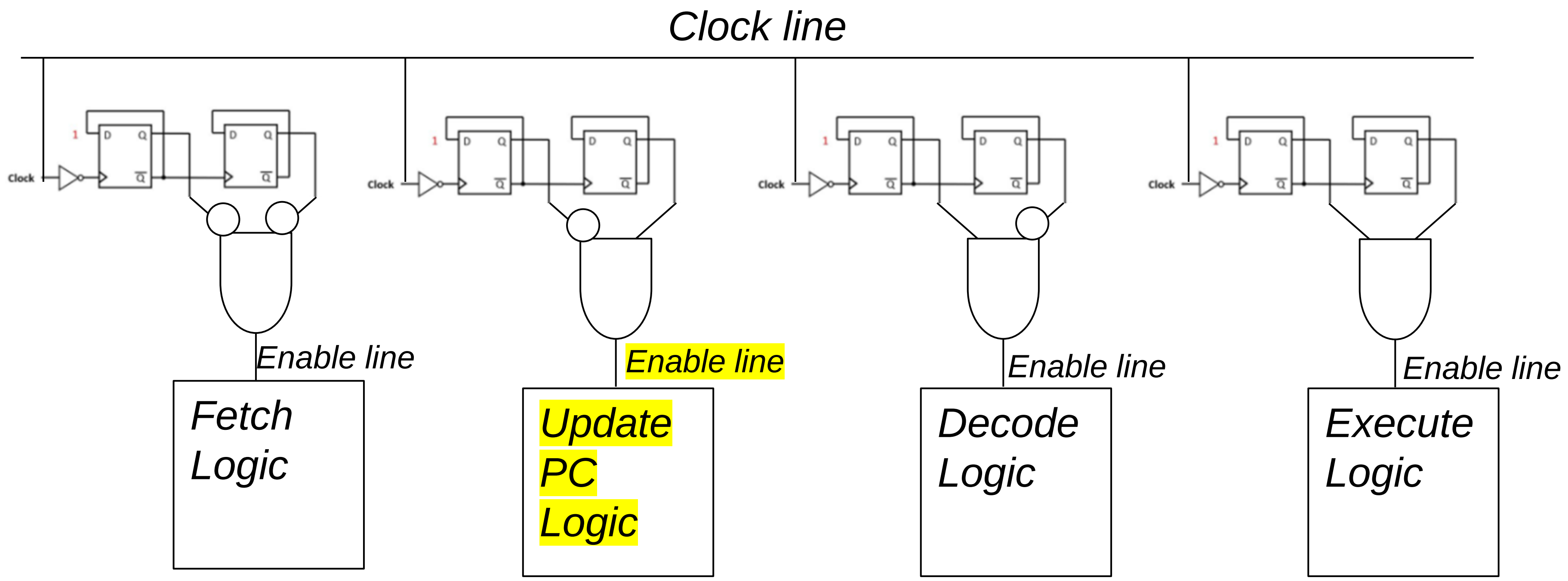
The processor is hard-wired to FETCH the instruction from the PC into the IR, and then increment the PC such that it points to the next instruction in RAM.



Adder state is 01: PC is incremented to set up for next fetch later on

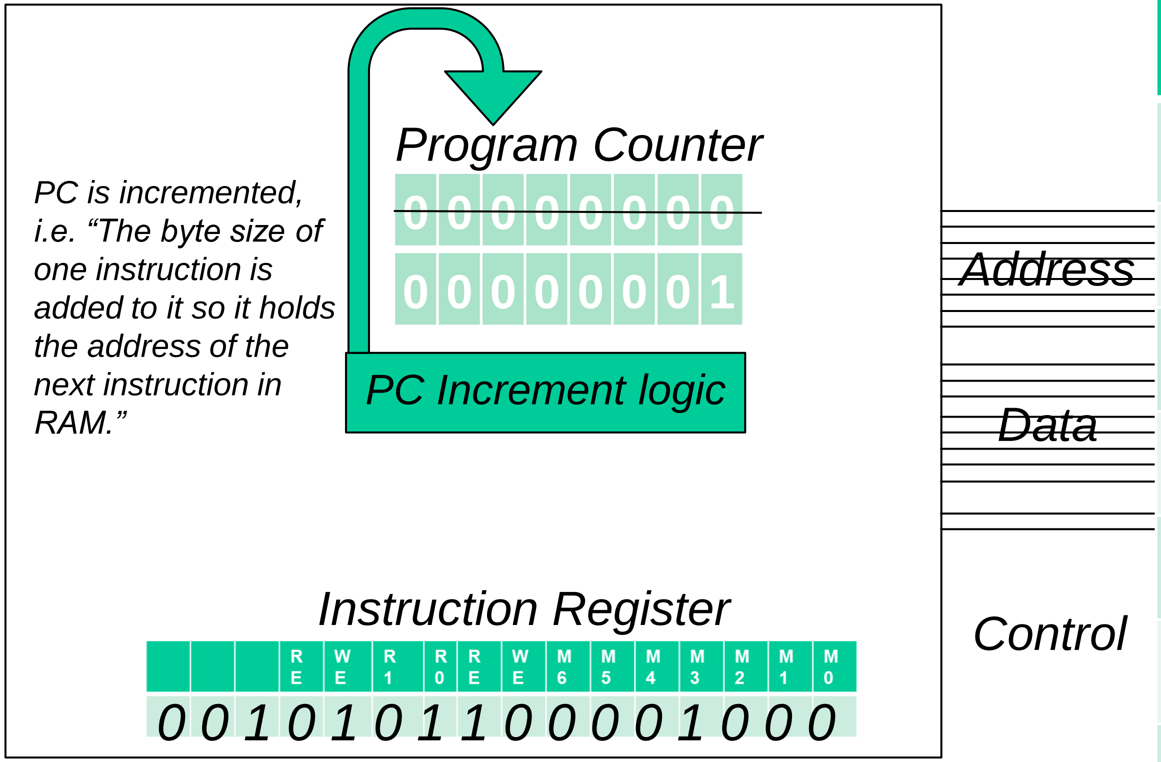
- We want to be able to cycle through four states:
  - 00 Fetch
  - 01 Update
  - 10 Decode
  - 11 Execute

- Fetch_enable = $\overline{A}$ & $\overline{B}$
- Update_enable = $\overline{A}$ & B
- Decode_enable = A & $\overline{B}$
- Execute_enable = A & B



Adder state is 01: PC is updated

0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0


Program Counter
~~0 0 0 0 0 0 0 0~~
0 0 0 0 0 0 0 1
PC Increment logic
PC is incremented, i.e. "The byte size of one instruction is added to it so it holds the address of the next instruction in RAM."
Address
Data
Control
Instruction Register
RE WE R1 R0 RE WE M6 M5 M4 M3 M2 M1 M0
0 0 1 0 1 0 1 1 0 0 0 0 1 0 0 0

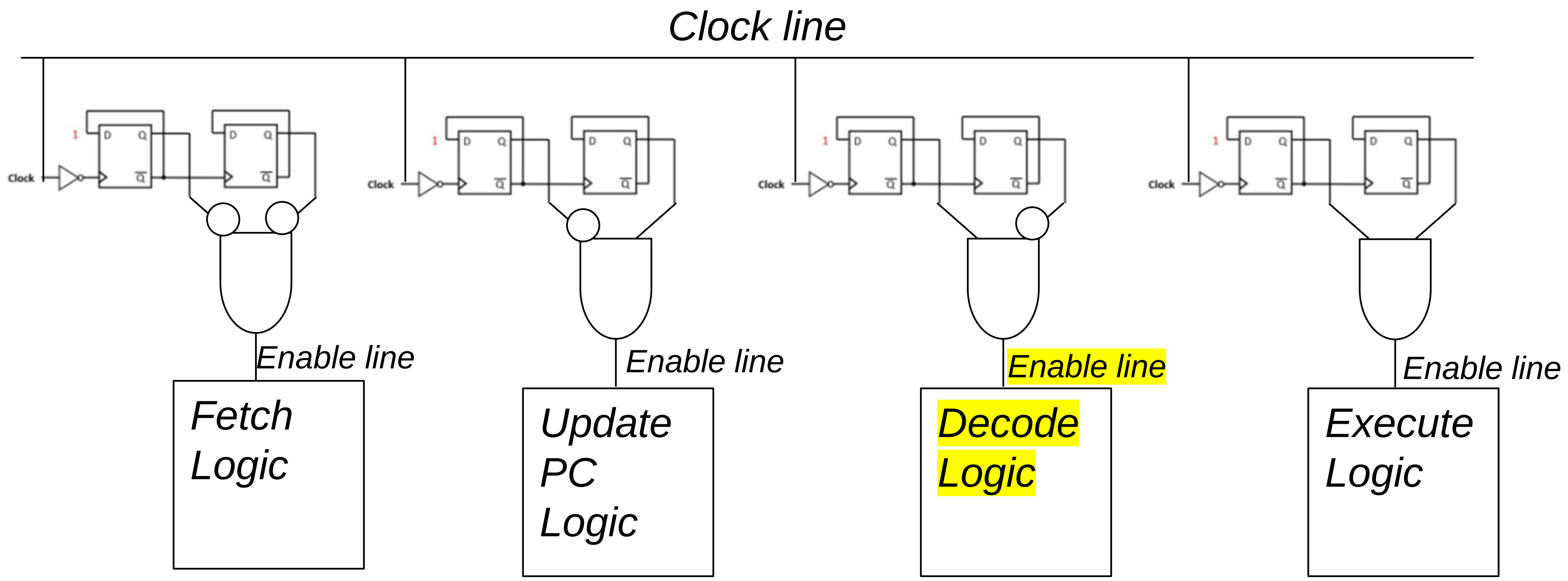
Addr	Contents
0	0010101100001000
1	0010111100001000
2	
3	
4	
5	
…	

Note that the PC is updated, but the IR won't be updated until the fetch.

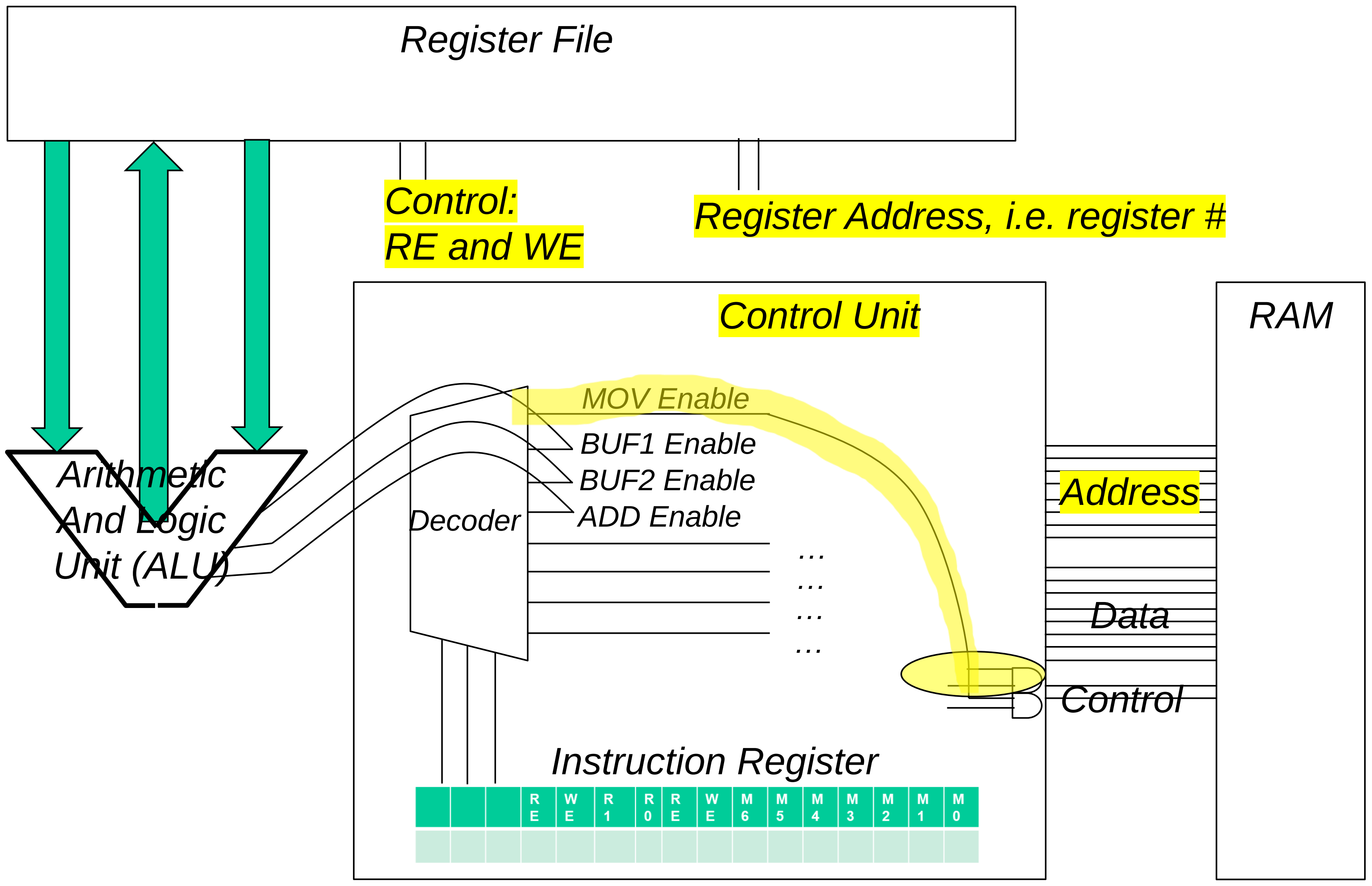
Adder state is 10: Instruction is decoded, control and enable lines are set

- *We want to be able to cycle through four states:*
  - *00 Fetch*
  - *01 Update*
  - *10 Decode*
  - *11 Execute*

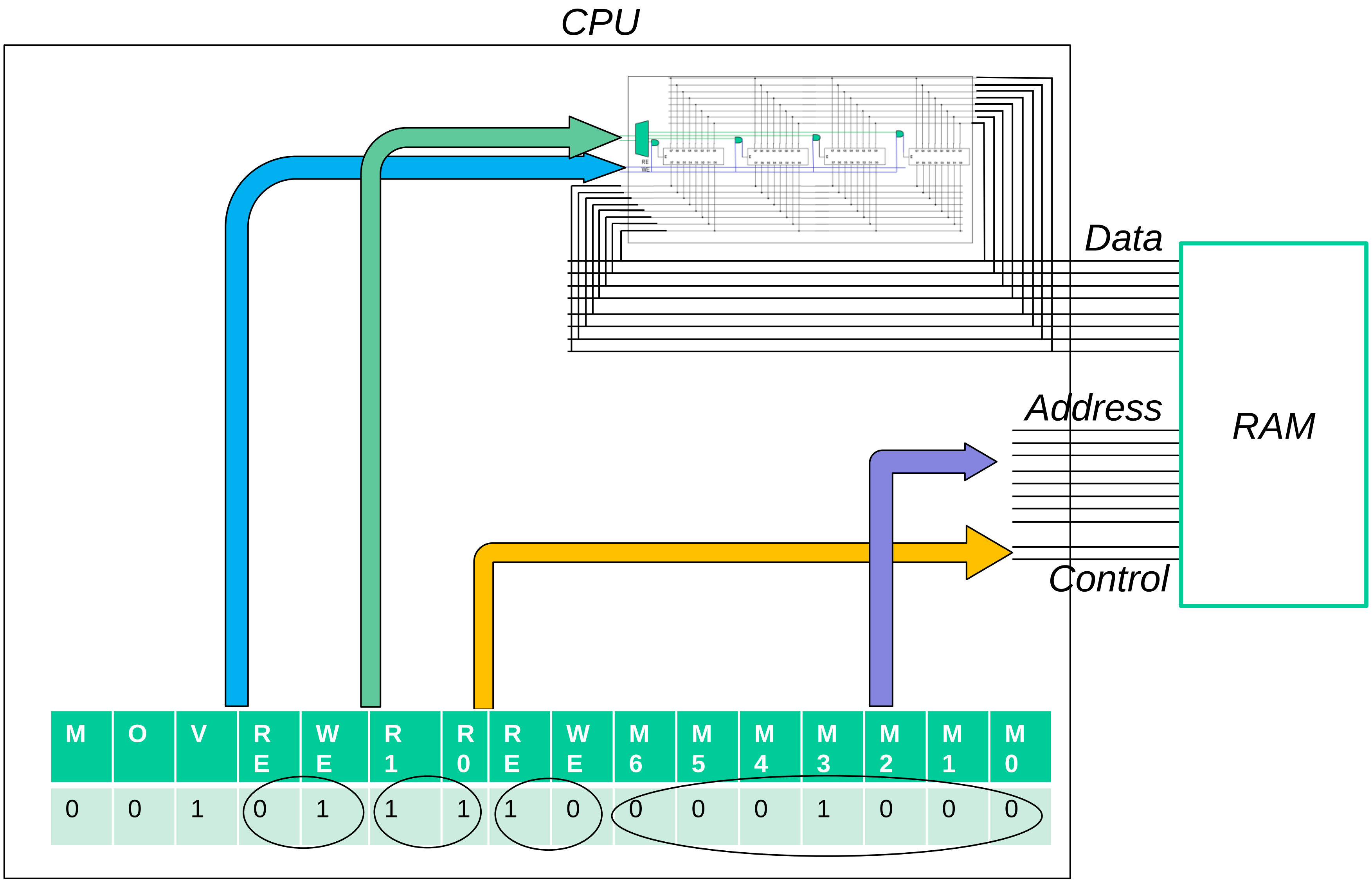
- *Fetch_enable = $\overline{A}$ & $\overline{B}$*
- *Update_enable = $\overline{A}$ & B*
- *Decode_enable = A & $\overline{B}$*
- *Execute_enable = A & B*



Adder state 10: Enable and address lines are set



Adder state 10: Enable and address lines are set

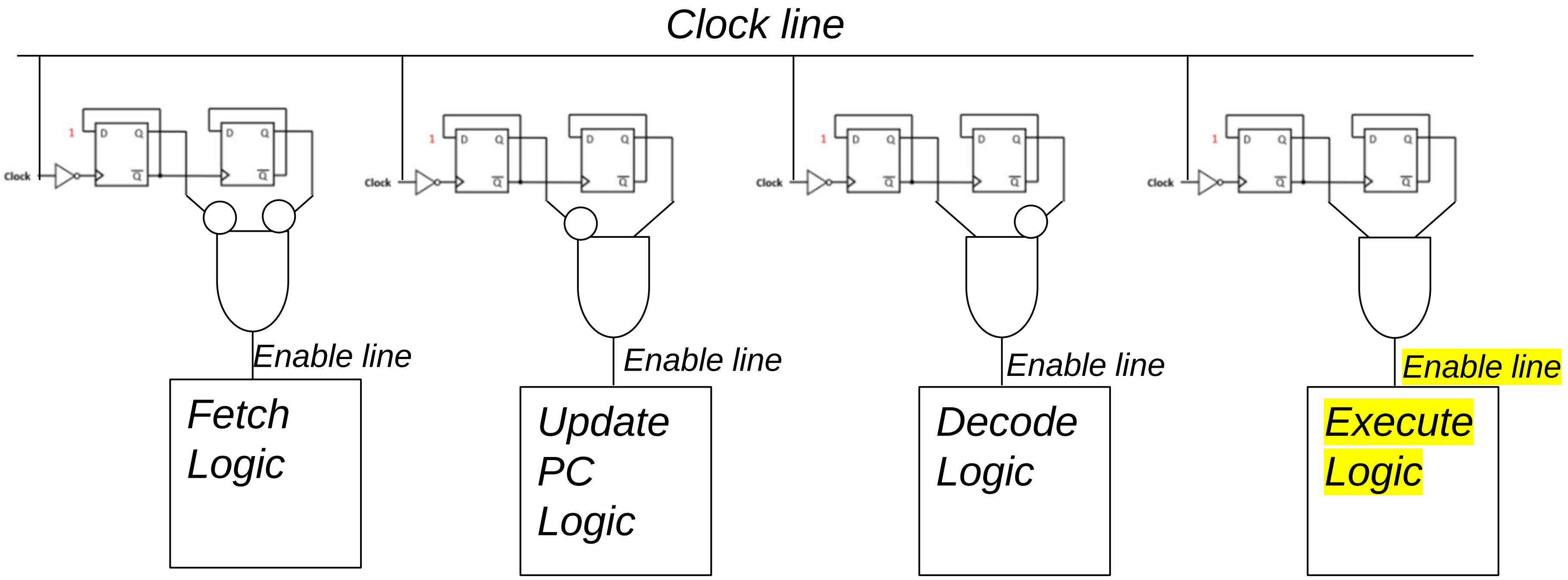


M	O	V	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0

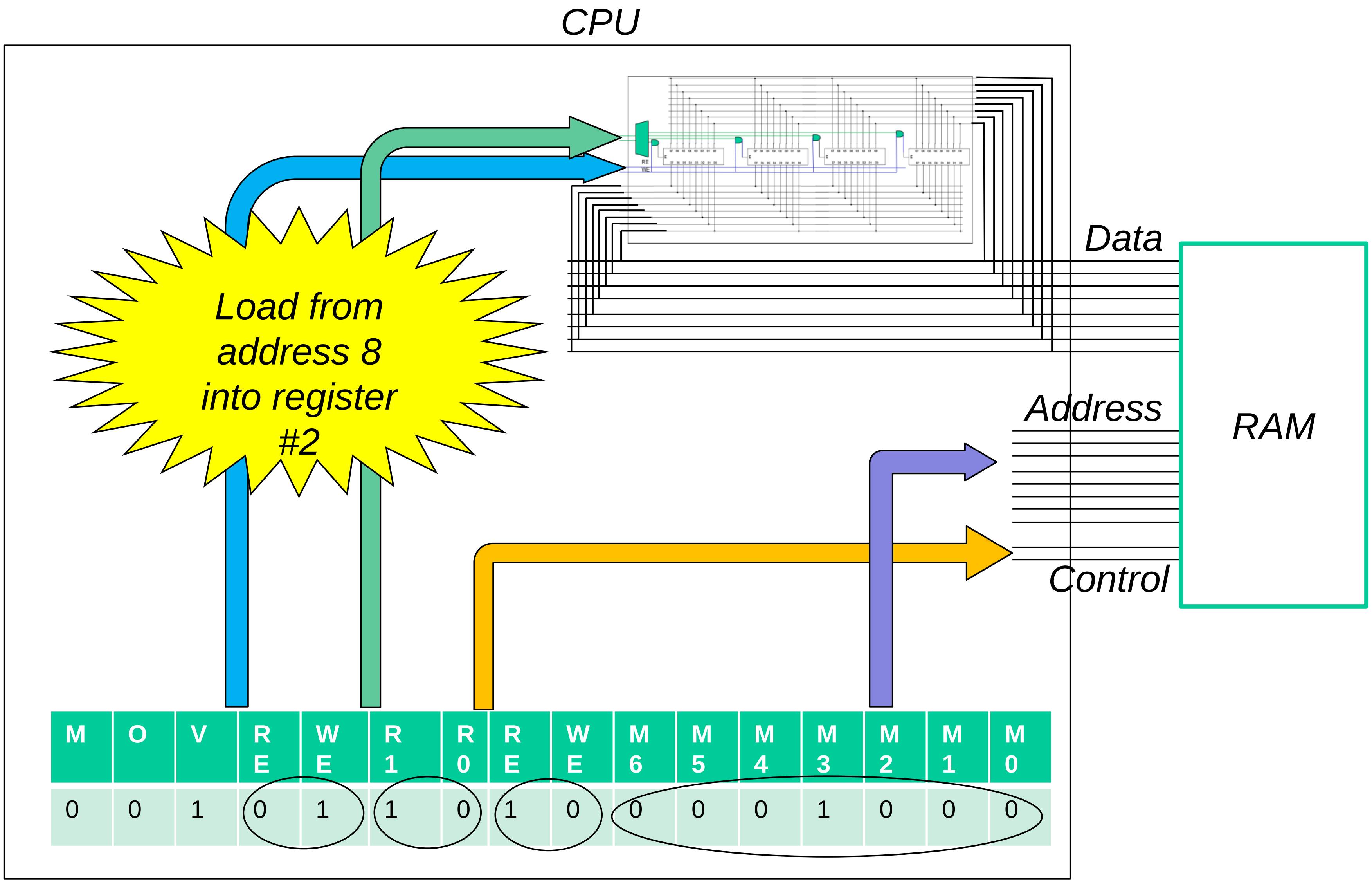
Adder state is 11: Time is allowed for execution

- *We want to be able to cycle through four states:*
    - *00 Fetch*
    - *01 Update*
    - *10 Decode*
    - *11 Execute*

- *Fetch_enable = $\overline{A}$ & $\overline{B}$*
- *Update_enable = $\overline{A}$ & B*
- *Decode_enable = A & $\overline{B}$*
- *Execute_enable = A & B*



Adder state is 11: Time is allowed for execution



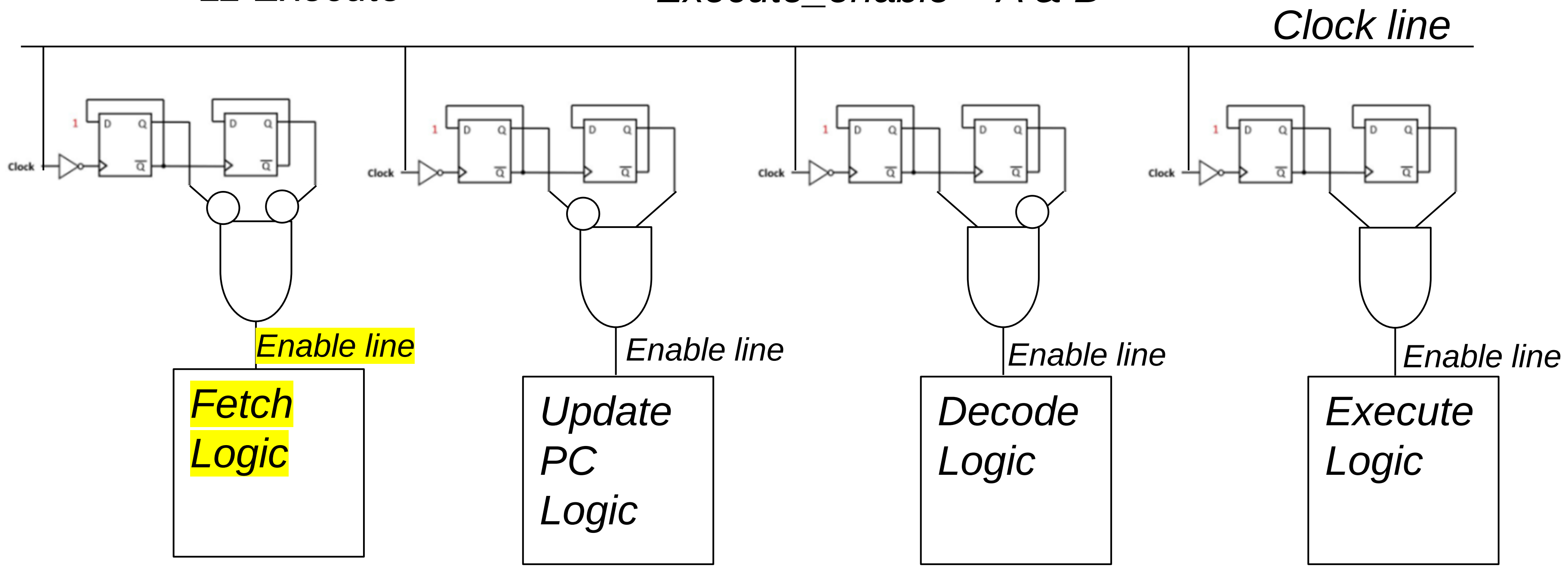
M	O	V	R E	W E	R 1	R 0	R E	W E	M 6	M 5	M 4	M 3	M 2	M 1	M 0
0	0	1	0	1	1	0	1	0	0	0	0	1	0	0	0

Adder state is 00: The instruction at the previously updated PC is loaded into the IR

- *11 + 1 has brought us back to 00 (Fetch).*
- *The cycle begins again, but this time with the PC updated during the most recent 01 Update phase.*

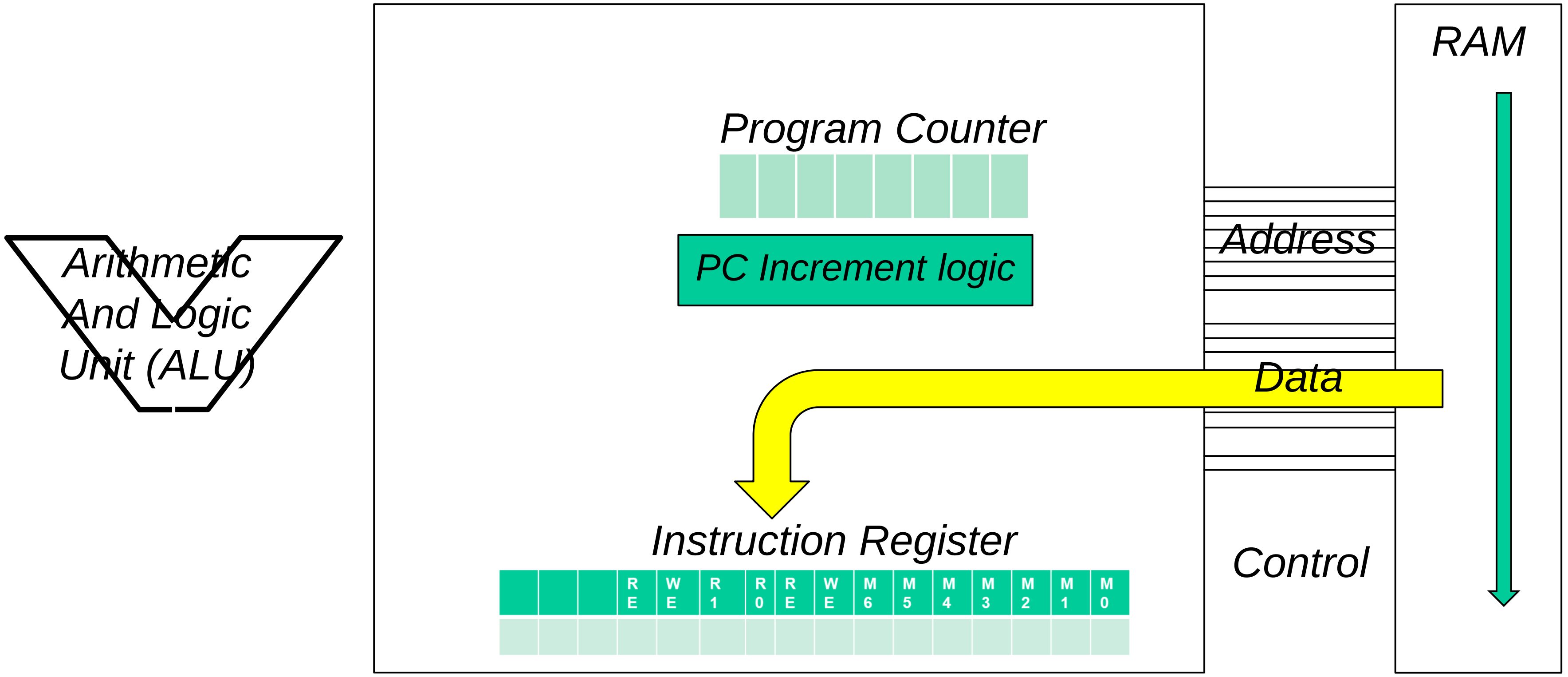
- *00 Fetch*
- *01 Update*
- *10 Decode*
- *11 Execute*

- *Fetch_enable = $\overline{A}$ & $\overline{B}$*
- *Update_enable = $\overline{A}$ & B*
- *Decode_enable = A & $\overline{B}$*
- *Execute_enable = A & B*



00: Fetch circuit is enabled.

The processor is hard-wired to FETCH the instruction from the PC into the IR, and then increment the PC such that it points to the next instruction in RAM.



Jump Instructions

- What if we don't want to execute the instruction immediately following the current instruction, i.e. we don't want to execute the instruction at the incremented PC?
  - If statement
  - Switch statement
  - Loop
  - Function call
  - Etc
- A JMP "jump" instruction loads a specific value into the PC, over-writing the value placed there during the increment phase.
  - The example below sets the PC to 0x55, such that the next fetch will occur at that address and the cycle will continue from there.

Instruction Register

JMP	RE	WE	R1	R0	RE	WE	M6	M5	M4	M3	M2	M1	M0
1010	0	0	0	0	0	0	1	0	1	0	1	0	1