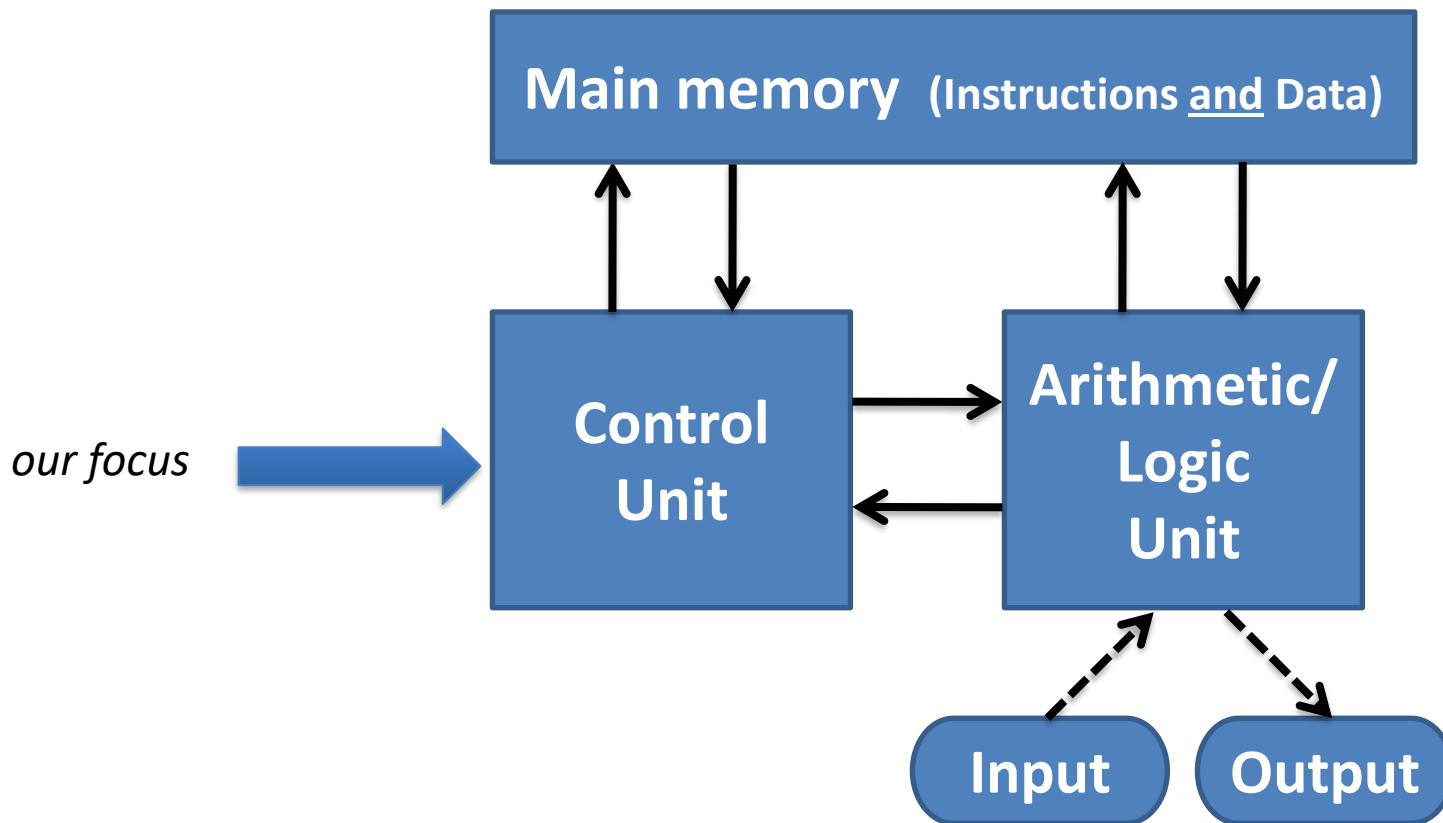


SCC131: Digital Systems

Topic 8: Building the control unit

Reminder of the von Neumann architecture



What *is* the control unit?

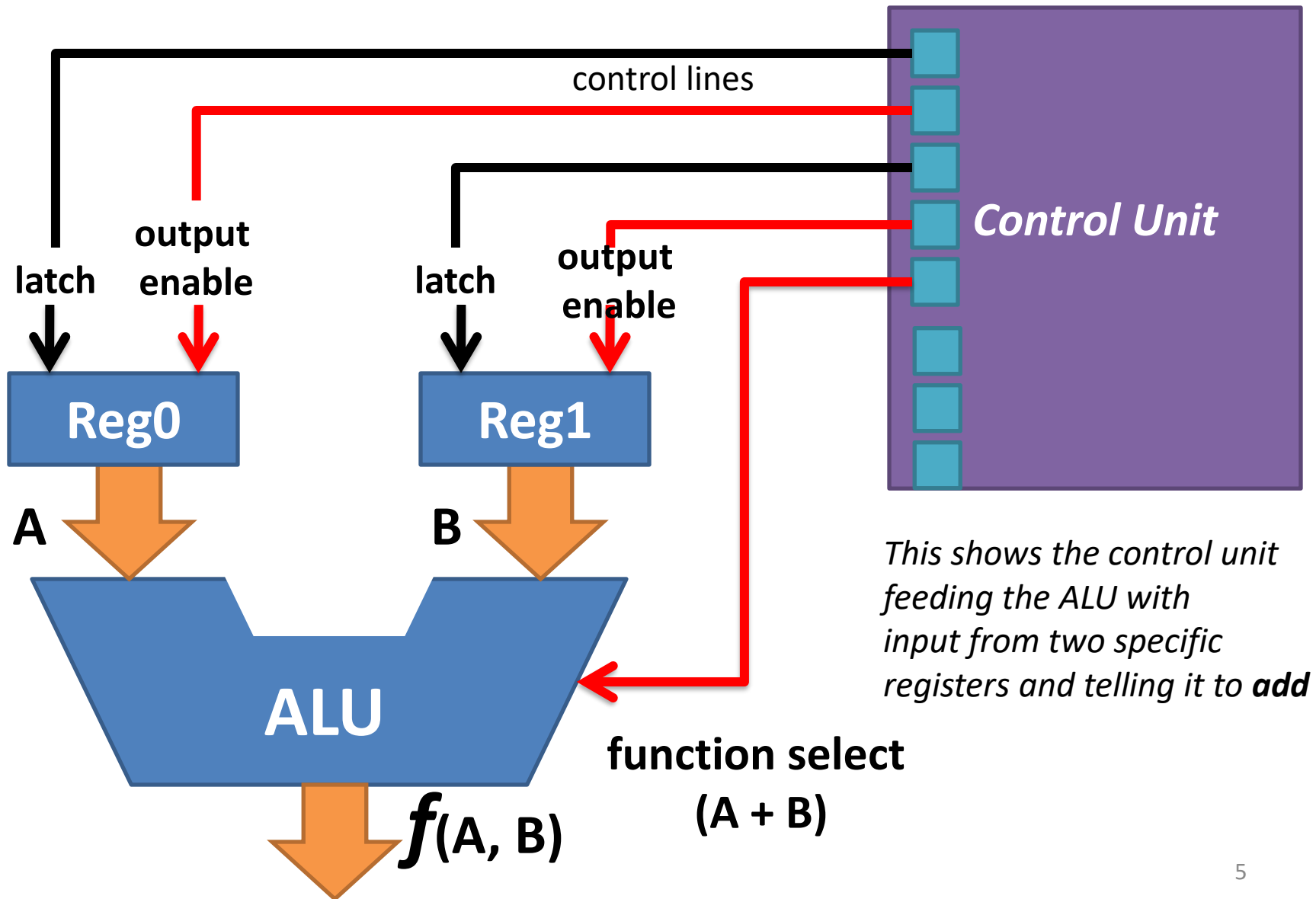
- It can be considered as a “little program” running inside the processor that **endlessly executes the fetch-decode-execute cycle**
- It controls and sequences the other architectural modules* using their respective *control lines*
 - E.g., we’ve already seen control lines such as the following: function select, shift L/R, carry in, latch, output enable, ...
- The control unit is itself driven by a ‘clock’, which gives regular, timed electrical pulses or ‘ticks’
 - Each instruction typically requires multiple ticks

* Primarily the ALU and main memory, but also I/O devices, caches, FP modules, ...

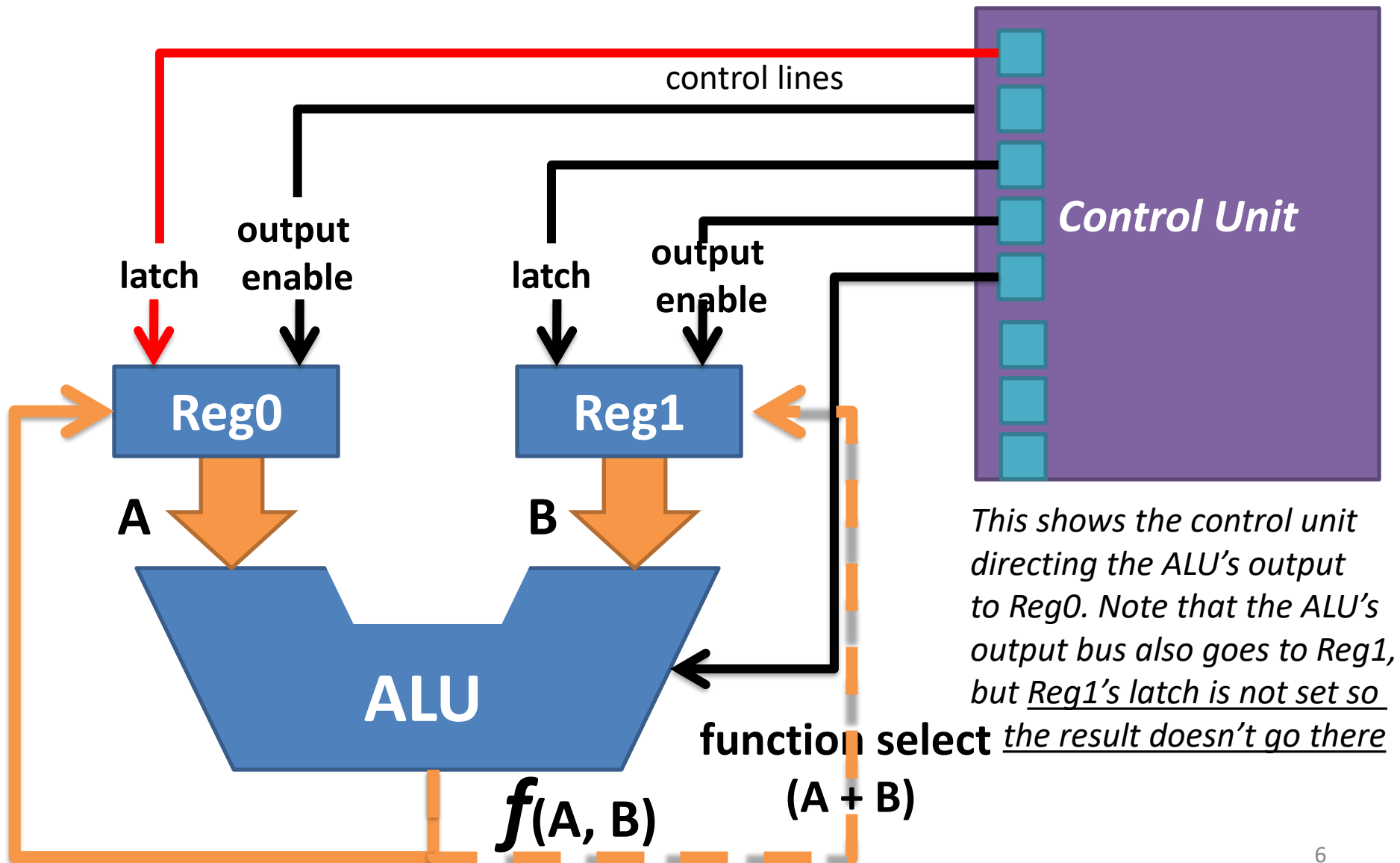
Controlling the ALU and registers

- We've already seen the relevant control lines
 - *Latch* and *output enable* on each register
 - *Function select*, *shift L/R* and *carry in* on the ALU
- Let's now look at the execution of a single instruction:
 - **ADD Reg0, Reg0, Reg1**
 - (This means: Reg0 is to be set to $\text{Reg0} + \text{Reg1}$)
 - We'll assume, for simplicity, that this instruction takes *two* clock ticks to complete...

Tick 1



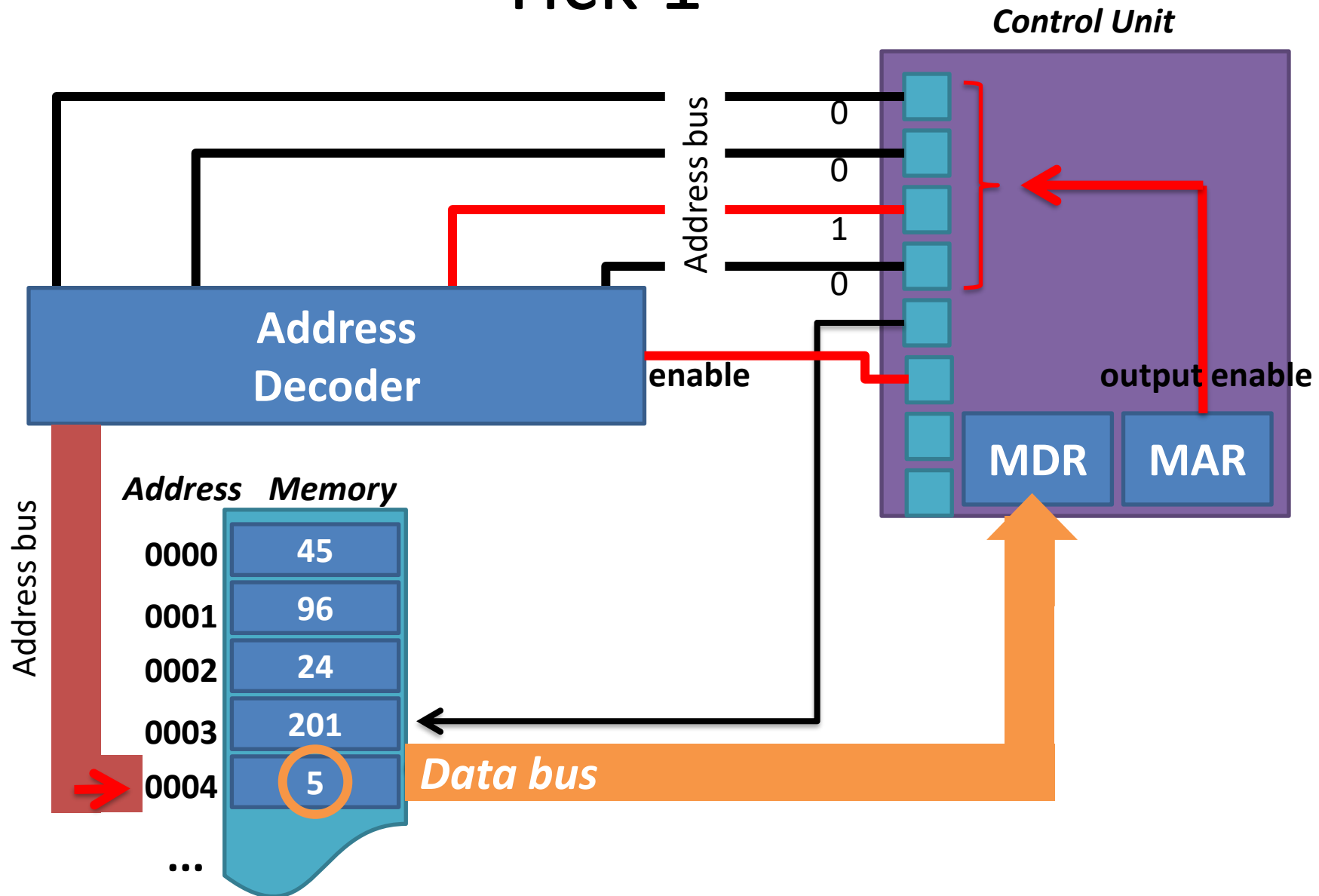
Tick 2



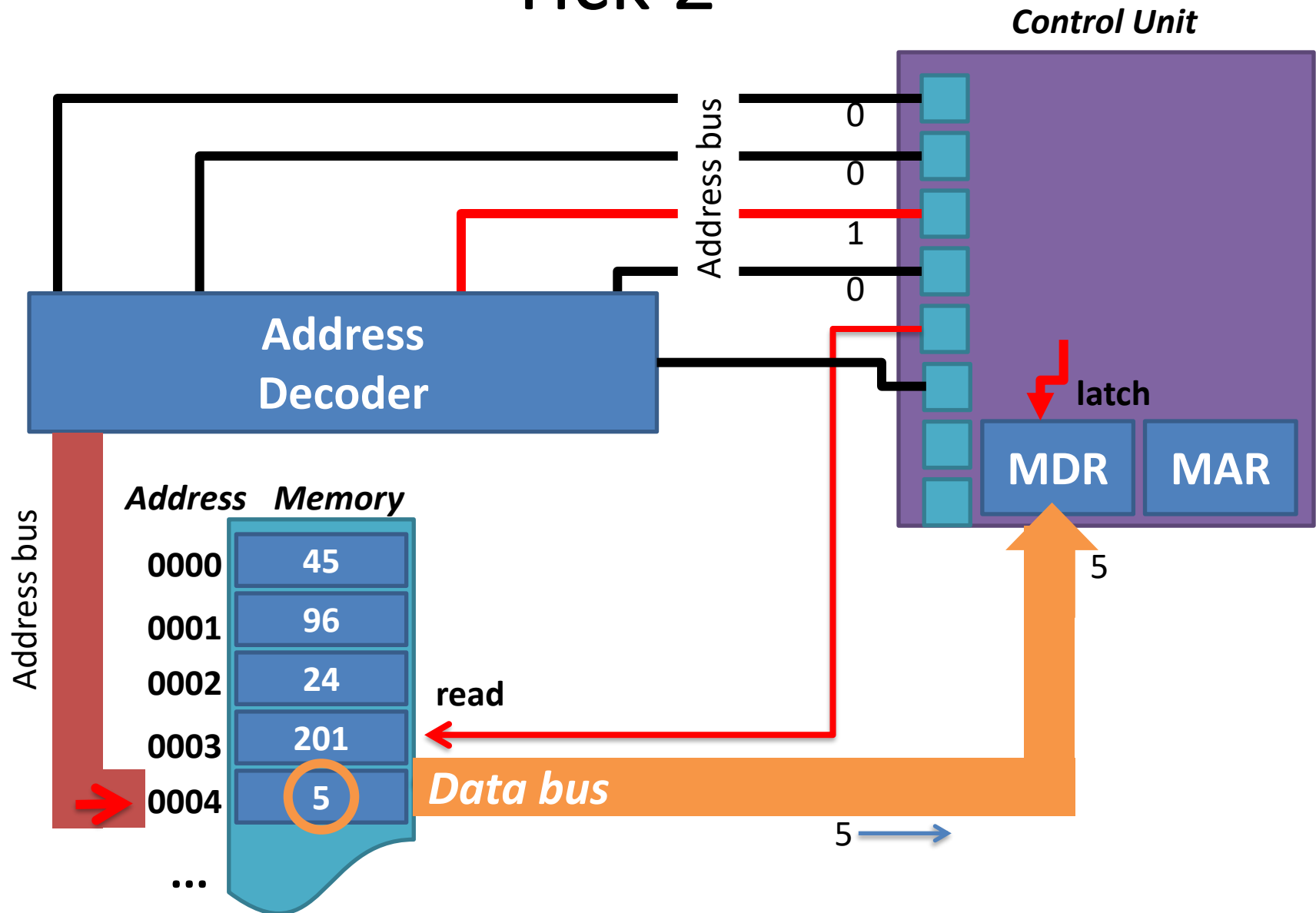
Controlling main memory

- The currently-executing instruction is held in a special register:
 - **Instruction Register (IR)** – holds the instruction currently being executed
 - (Recall that we've already seen other 'special registers': flags, PC and SP)
- In addition, the control unit employs the following special registers to enable it to control memory:
 - **Memory Address Register (MAR)** – as the current instruction executes, this holds the memory address from which processor will read some data, or to which it will write some data (depending on what the instruction does)
 - **Memory Data Register (MDR)** – as the current instruction executes, this holds data going to/ coming from the memory address in the MAR
- Let's see how the control unit might 'sequence' the reading of the contents of the memory address indicated by MAR, into MDR ...

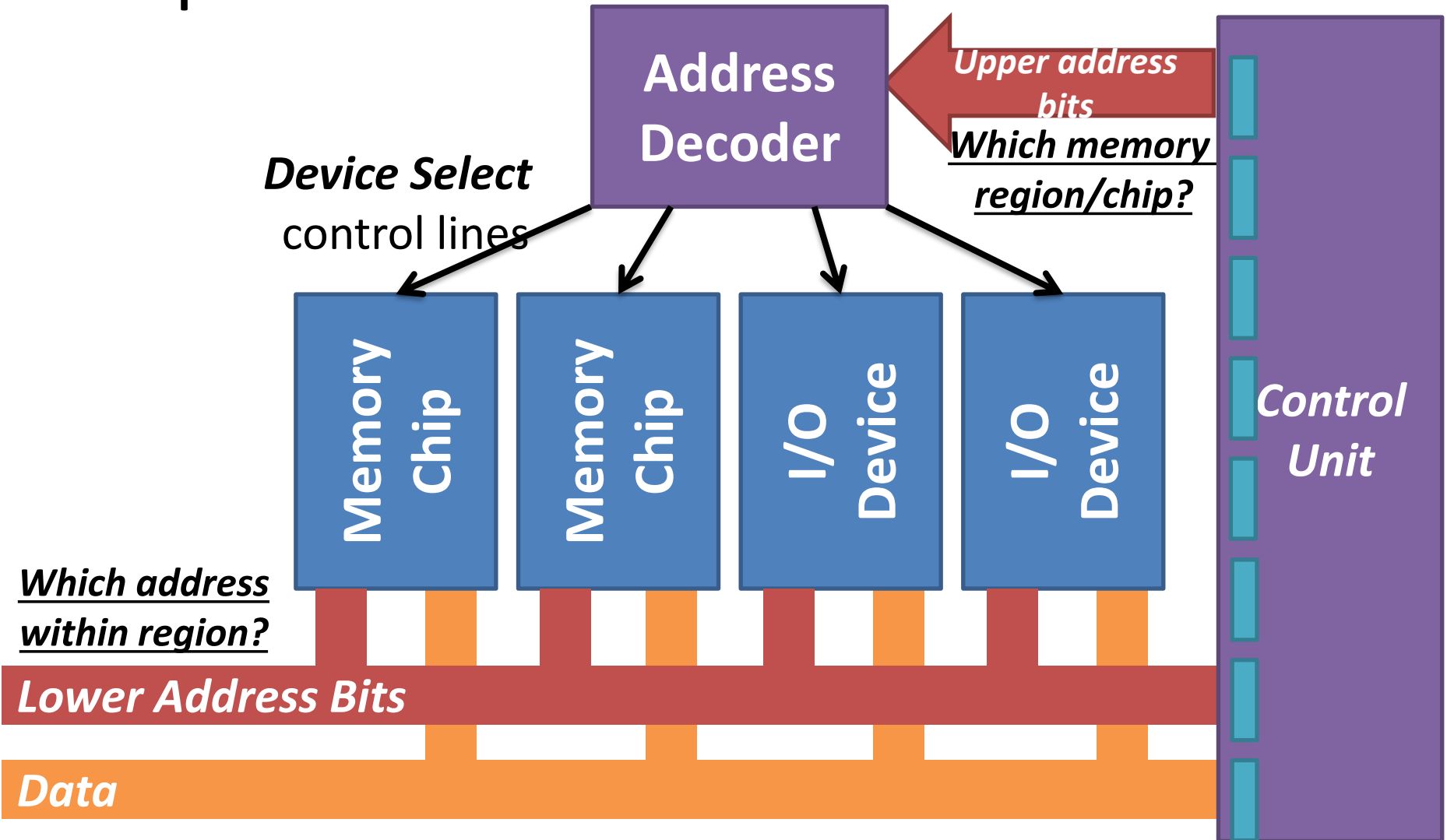
Tick 1



Tick 2



In practice, it may be a bit more complex...



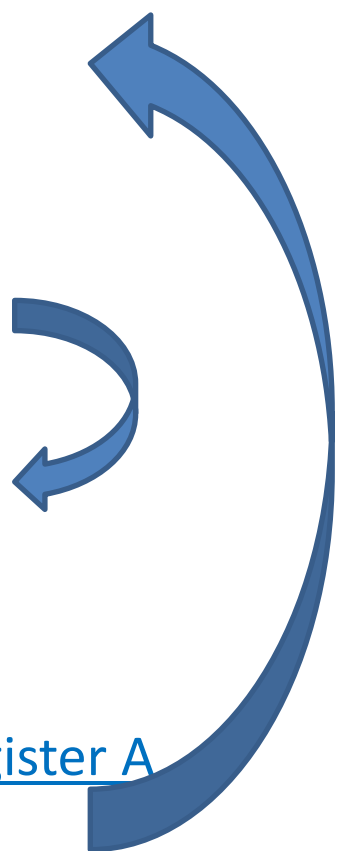
Each memory chip will need a similar decoder internally for the lower bits¹⁰

Tracing the fetch-decode-execute cycle

- So, we can see that the control unit repeatedly executes the fetch-decode-execute cycle, setting control lines as required
- Let's look at a more complete example... Let's assume:
 - An “**ADDA** C” instruction that adds constant C to Reg. A
 - That the operation code (op-code) of ADDA is 1F (in hex)
 - That we have an 8-bit data bus
- **Let's trace ADDA 5...**

1. Read byte from memory pointed to by PC *[Reads byte 1F into the IR]*
2. Decode the instruction in IR (Instruction Register) *[1F → add constant to A]*
3. Advance PC to point to operand byte *[Moves past instruction 1F]*
4. Read operand byte (5) from memory *[See “Controlling main memory” slide]*
5. Add value 5 to Register A *[See “Controlling ALU and registers” slide]*
6. Advance PC ready for next instruction *[Moves PC past operand]*
7. **Go back to step 1 to do the next instruction...**

“1F05” (ADDA 5) in yet more detail...

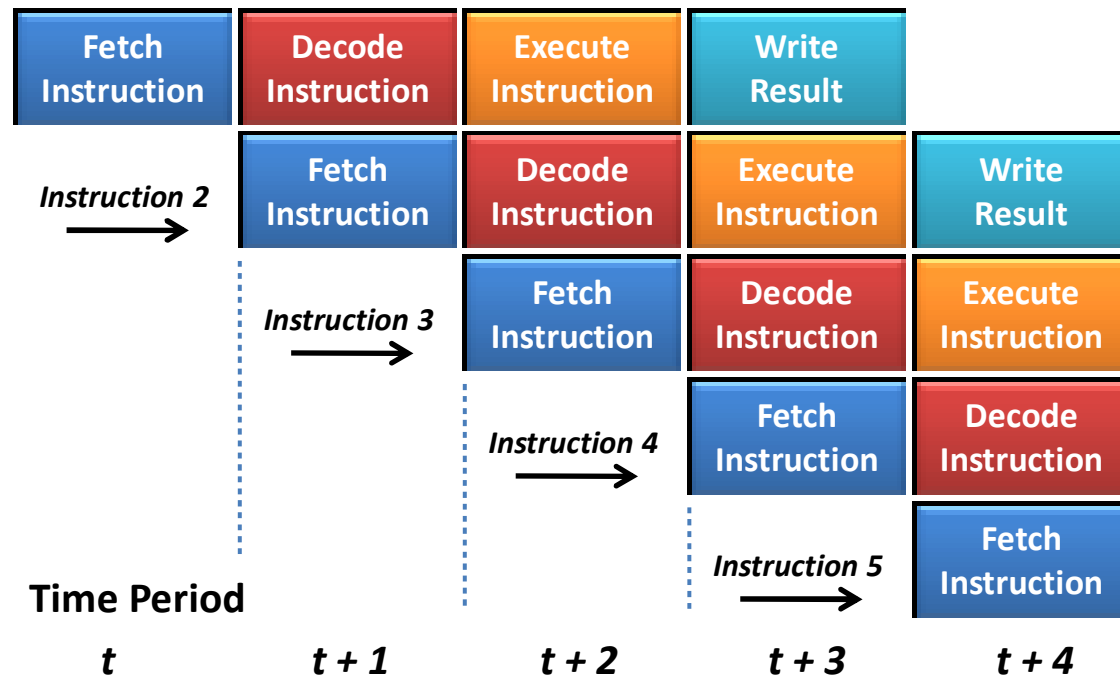
- **Fetch and decode**
 - MAR = PC Get address of instruction
 - MDR = (MAR) Read op-code (**1F**)
 - IR = MDR Transfer to Instruction Register
 - *Decode instruction:* op-code 1F → **add constant to A**
 - **Execute ADDA**
 - PC = PC + 1 Advance over op-code
 - MAR = PC Get address of operand
 - MDR = (MAR) Read operand into MDR (**05**)
 - A = A + MDR Use ALU to add value in MDR to register A
 - PC = PC + 1 Advance PC to next instruction
- 

FSM-based vs microcoded control units

- The control unit's fetch-decode-execute loop can be implemented in **two** alternative ways:
 1. As a ***finite state machine (FSM)***: hard-wired sequential logic
(built directly in terms of NAND gates etc)
 - high performance, but expensive and hard to evolve
 2. As ***microcode***: a sequence of **micro-instructions** in a **micro-memory**
 - slightly lower performance but much more flexible
(e.g. manufacturer can “easily” add a new instruction)

Pipelining

- Pipelining is a widely-used way to exploit inherent **parallelism** inside the control unit to **speed up** the fetch-decode-execute cycle
 - If we can split the cycle into n sub-stages, we get n x speedup!
 - Cf. human workers on a factory production line



Pipeline hazards

- There are various categories of “things that can go wrong”, known as *pipeline hazards*
 - **Control hazards**
 - Occur when a control-transfer instruction changes the flow of execution
 - **Data hazards**
 - Occur when instruction n depends on a result from instruction $n-1$
 - Or when two parts of the pipeline need access to the same data
 - **Structural hazards**
 - Occur when two parts of the pipeline need access to the same piece of hardware (e.g. ALU, address decoder, ...)
- When encountered, such hazards may cause the pipeline to “stall”; so we must “flush” it to continue
 - This might considerably reduce the apparent 4 x speedup
 - Pipeline design is a very complex and tricky business!

Summary

- We know what the control unit is and what it does
- We understand how the control unit controls the ALU, registers and memory via **control lines**
- We understand the function of the control unit's **special registers** for interaction with main memory
- We have a pretty clear understanding of how the control unit executes the **fetch-decode-execute** cycle
- We understand the distinction between **FSM-based** and **microcode-based** control, and their pros and cons
- We understand how **pipelining** can significantly speed up the fetch-decode-execute cycle, but that it is vulnerable to various “hazards”