SoC 101:

*a.k.a., "**Everything you wanted to know about a computer but were afraid to ask**"*

Communicating with Peripherals

(How to build a router…)

Prof. Adam Teman

EnICS Labs, Bar-Ilan University

22 May 2023

Heavily based on the wonderful lecture
"Interfaces: External/Internal, or why CPUs suck"
by Tzachi Noy, 2019

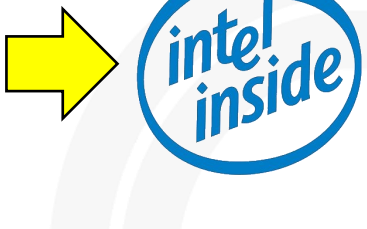
Emerging Nanoscaled Integrated Circuits and Systems Labs

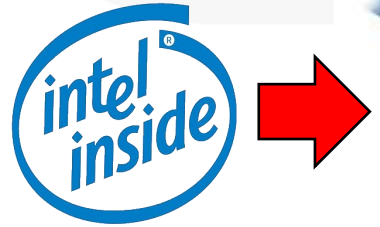
er Kofkin
ineering
Bar-Ilan University

What do a car and a router have in common?

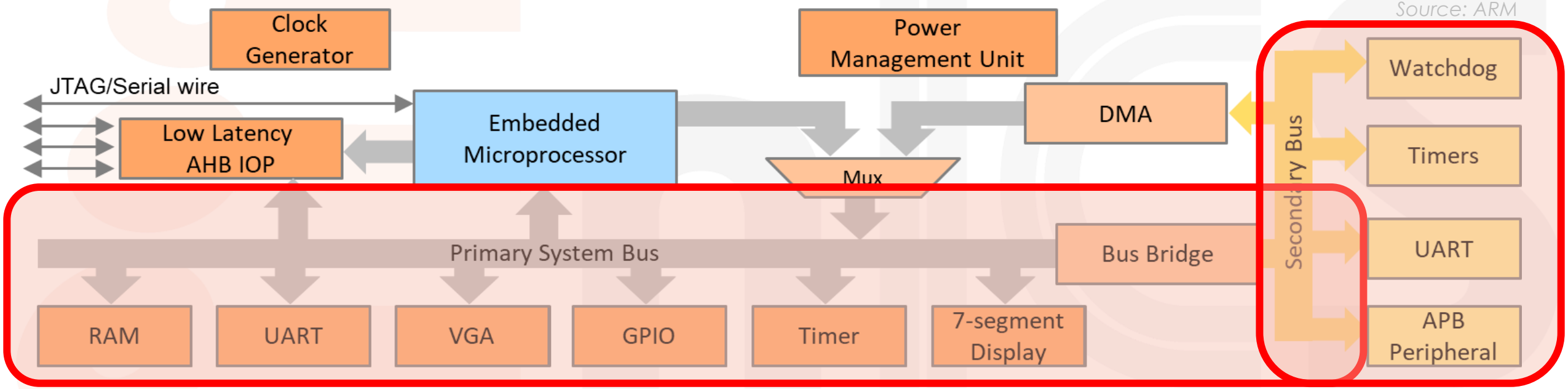
intel inside

Both the car and the router have interfaces



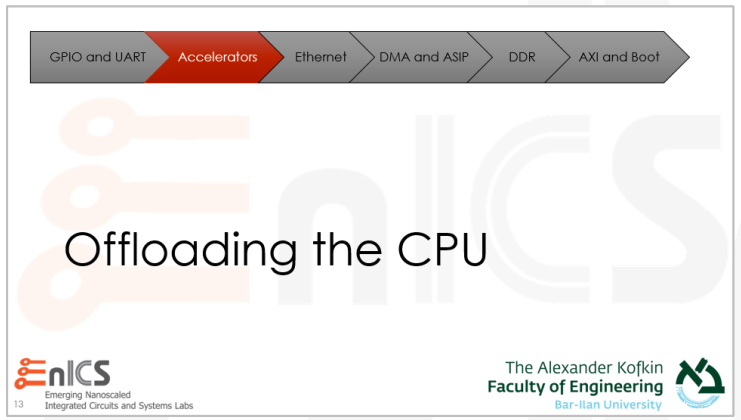


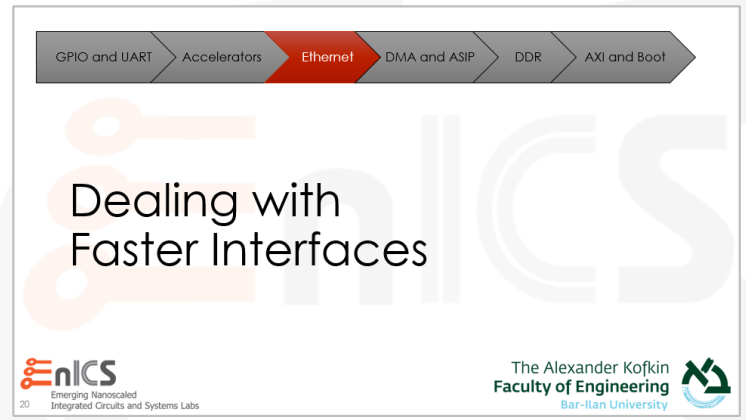
This Lecture



We will "design" a **router** as an example to introduce these concepts

Lecture Outline





Communicating with the outside world

Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

CPU
memory

Reminder: Memory-Mapped I/O

- **Registers and I/O Devices are given an address in the system's memory map:**
  - Everything is treated the same as memory.
  - To communicate with an I/O, we write to and read from these addresses.
  - These are achieved with simple `load` and `store` assembly commands.

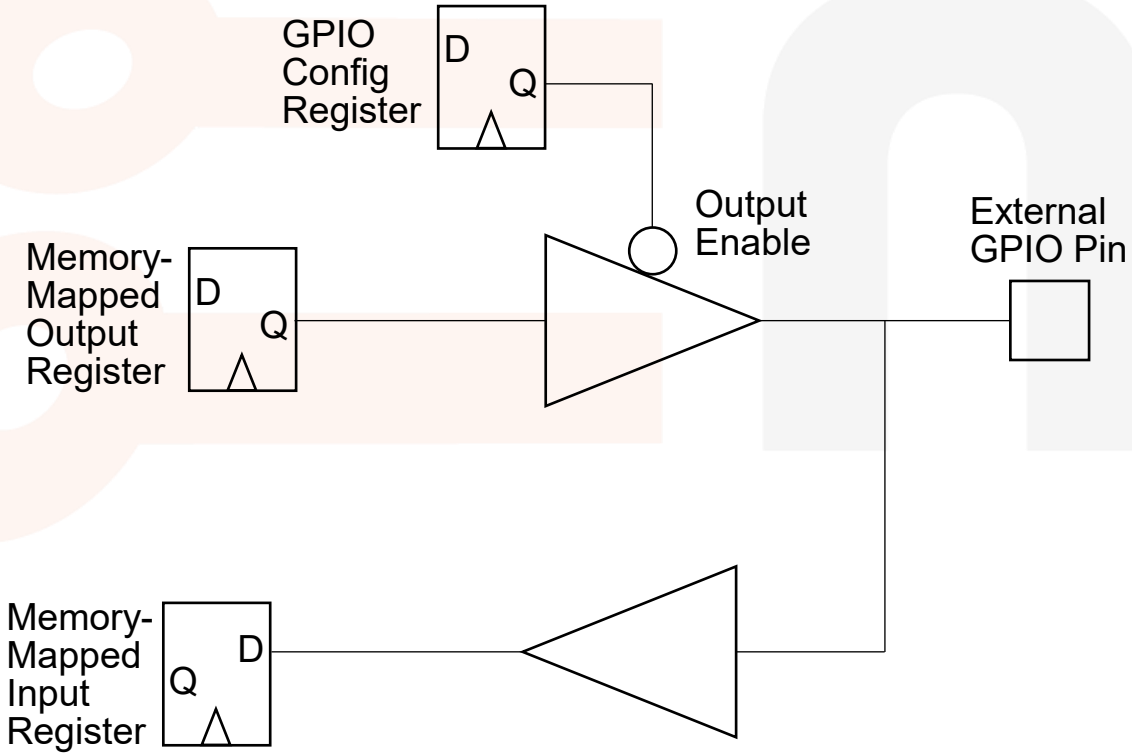
- **In C, we can define two functions, `peek` and `poke`, to accomplish this easily:**
  - Now to access a register, just define its address, and use these functions

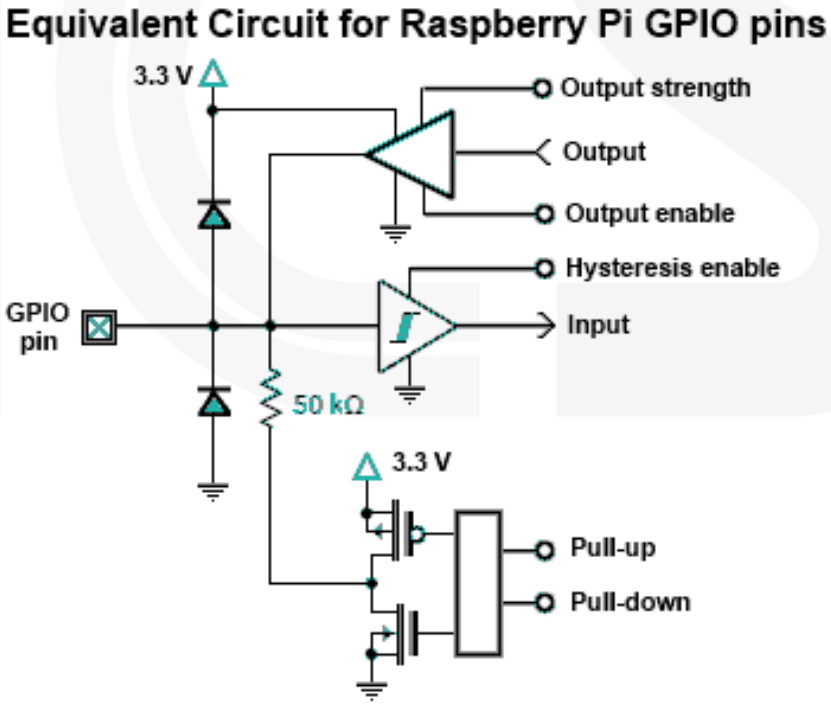
```
int peek (char *location) {
  // Read from a memory-mapped address
  return *location;
}
void poke (char *location, char newval) {
  // Write to a memory-mapped address
  (*location) = newval;
}
```

```
#define DEV1 0x1000
...
dev_status = peek(DEV1)
...
poke(DEV1,8);
```

General Purpose I/O (GPIO)

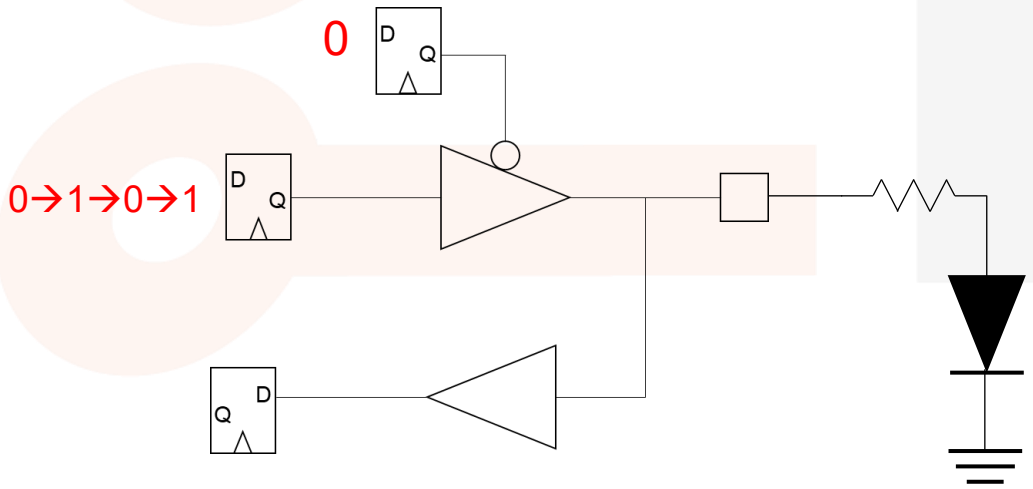
- **Most microcontrollers have a set of general purpose input/output (GPIO) pins.**
  - Can be configured as input pins or output pins.
  - Can be programmed by software for various purposes.





Example: Blinking a LED

- First, configure the GPIO to be an output.
- Next, create an infinite loop that:
  - Toggles the state of the output.
  - Waits for a given period.

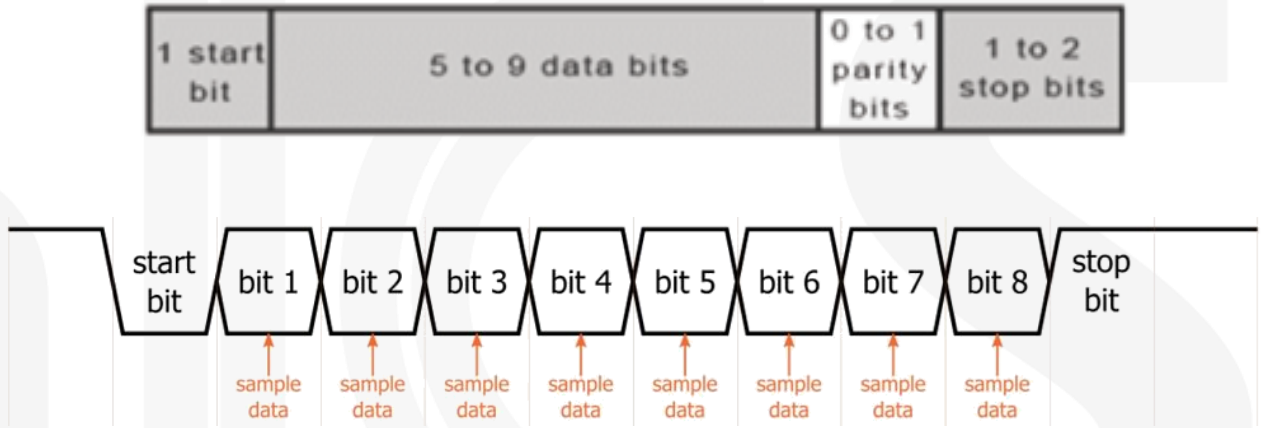


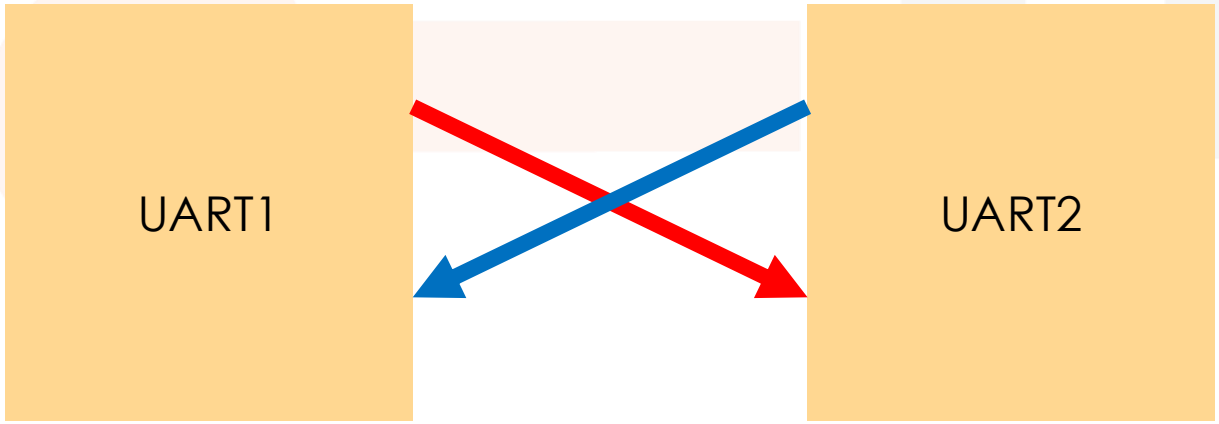

```
#define GPIO_CONFIG_REG 0x10000000
#define GPIO_OUTPUT_REG 0x10000001
#define GPIO_BLINK_PIN 0b00000001
#define BLINK_PERIOD 1000000
int main () {
  // Set LED connected GPIO PIN to output
  toggle_config |= peek(GPIO_CONFIG_REG);
  poke(GPIO_CONFIG_REG,toggle_config);

  while (true) {
    // Toggle the state of the GPIO output register
    output_status = peek(GPIO_OUTPUT_REG);
    poke(GPIO_OUTPUT_REG,
           output_status~GPIO_BLINK_PIN);
    // Wait for a predefined delay
    wait(BLINK_PERIOD);
  }
}
```

Communicating Off-Chip

- **What if we want to communicate with something *more sophisticated* than a LED or a button?**
  - We need a communication protocol.
- **Introducing UART**
  - The Universal Asynchronous Receiver/Transmitter





- **Baud Rate**
  - Number of bits per unit time

$$\text{Baud Rate} = 1/\text{bit time}$$

- **Bandwidth**
  - Data per unit time

$$BW = \frac{\text{data bits}}{\text{frame bits}} \cdot \text{Baud Rate}$$

Can we use UART for our router?

BAUD RATE: 115,200 bits/sec

SAMPLE RATE: 230,400 samples/sec

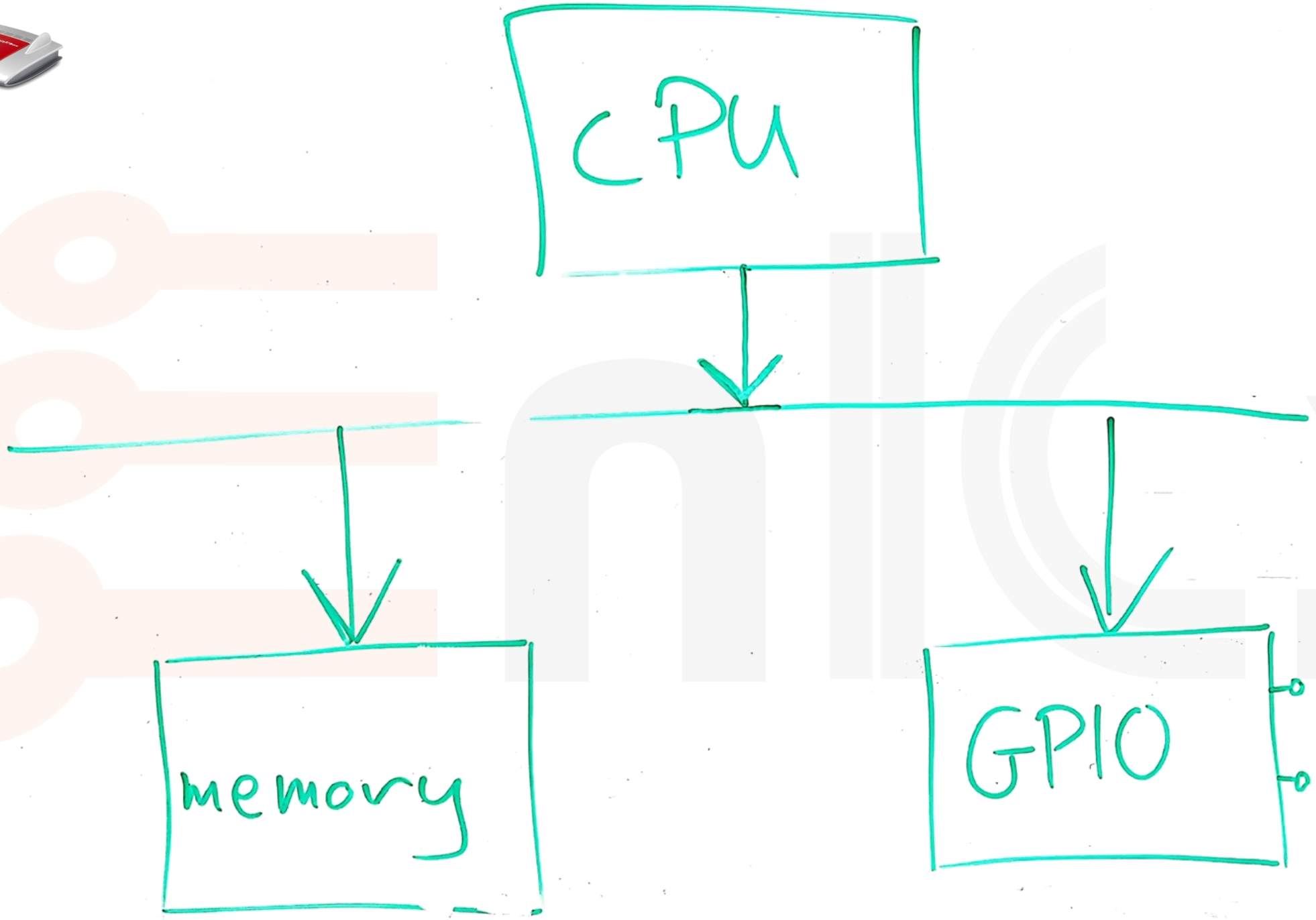
CODE: 40 instructions/sample

OVERHEAD: 9,216,000 instructions/sec

9.2% of CPU time

** Assuming a 100MHz clock frequency*

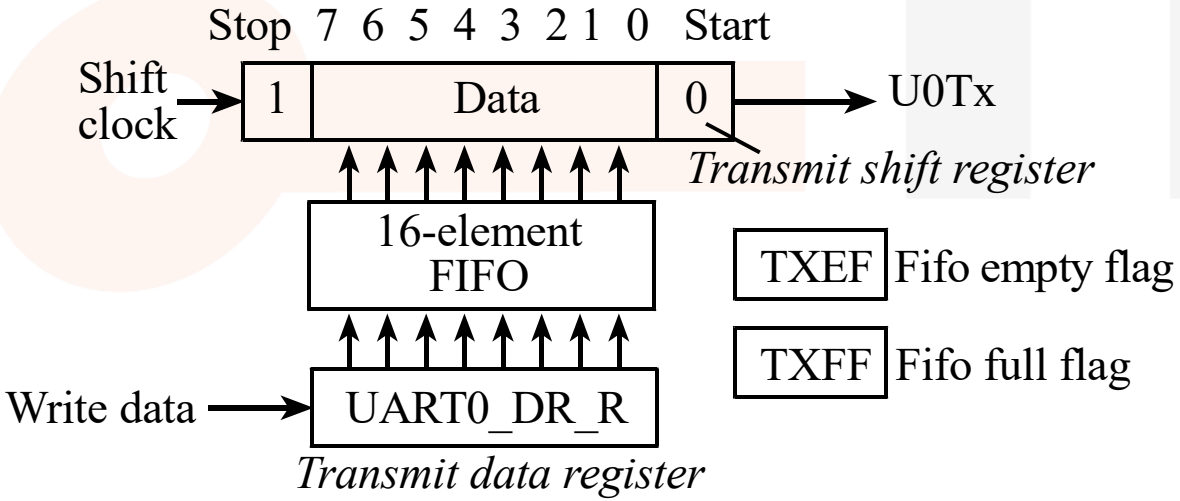
Offloading the CPU


CPU
memory
GPIO

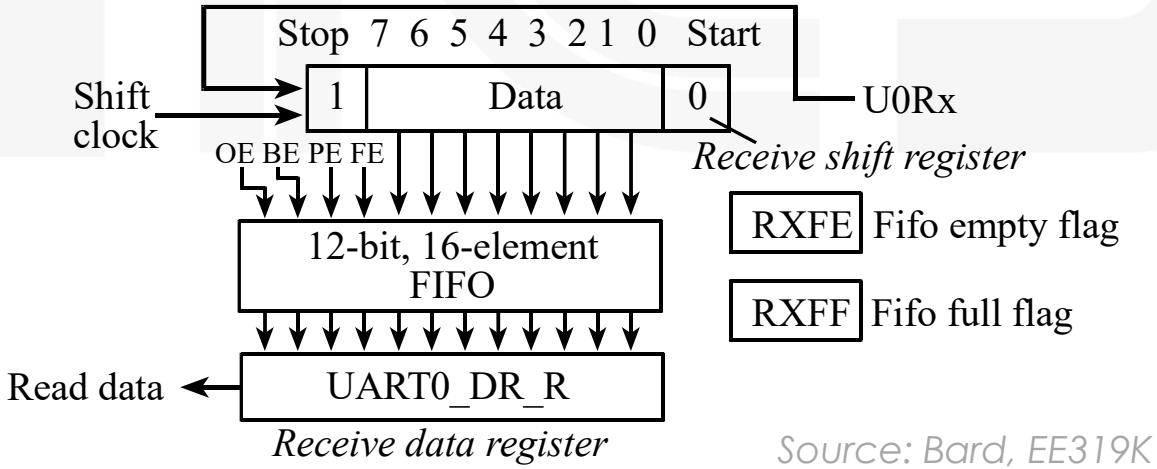
Offload the CPU with a controller

- **UART is a slow serial protocol**
  - One bit is transferred at a time at a low baud rate (e.g., 1200-115200 bits/sec).
- **Integrate a specific UART controller that offloads the CPU**
  - Communicate with the UART through a wider register (e.g., byte, 32-bit).
  - Use a Shift Register to serialize/deserialize parallel data
  - Use a FIFO to buffer several CPU transactions

1-byte → 16X speedup
64 byte FIFO→512X speedup



Transmit Parallel-In Serial-Out (PISO)

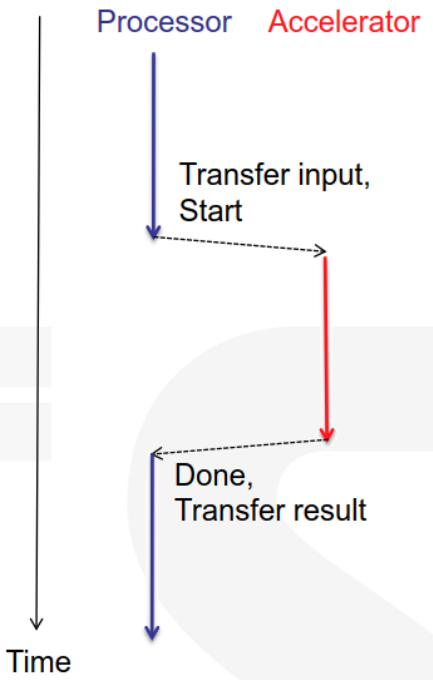


Receive Serial-In Parallel-Out (SIPO)

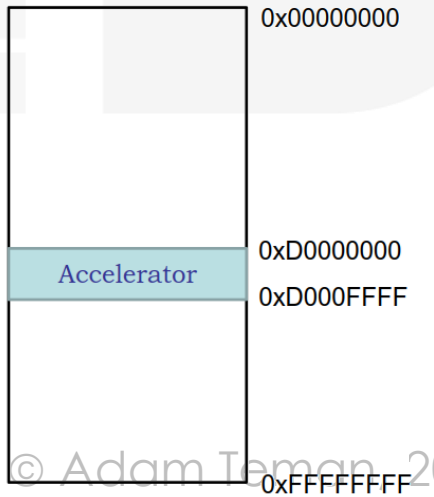
Source: Bard, EE319K

Hardware Acceleration

- **CPUs are general purpose programmable machines**
  - In other words, they are Turing Complete.
  - But CPUs are not great (sometimes terrible) at carrying out certain operations.
- **Offload the CPU by providing dedicated hardware**
  - The dedicated hardware can be designed to efficiently run a specific task → *accelerate* it.
  - The CPU can continue running the program, while the accelerator runs its task.
- **Data transfer achieved through memory mapping**
  - CPU writes/stores inputs and control in memory/registers.
  - Accelerator writes outputs/status in memory/registers.



$$Speedup = \frac{T_{\text{original}}}{T_{\text{unaccel}} + T_{\text{accelerated}} + T_{\text{comm}}}$$



Source: Raghunathan, ECE 695R

But how do we know when it's done?

- **How can the CPU know when a new byte of data is received?**
  - Simple way: "Polling"
  - Check on the status of the UART every so often to see if data has been received (or if it is ready to receive new data)

```
do {
   // Play games
   ...
   // Poll to see if we're there yet.
   status = areWeThereYet();
 } while (status == NO);
```

- **Polling can be carried out with a "busy-wait" loop:**

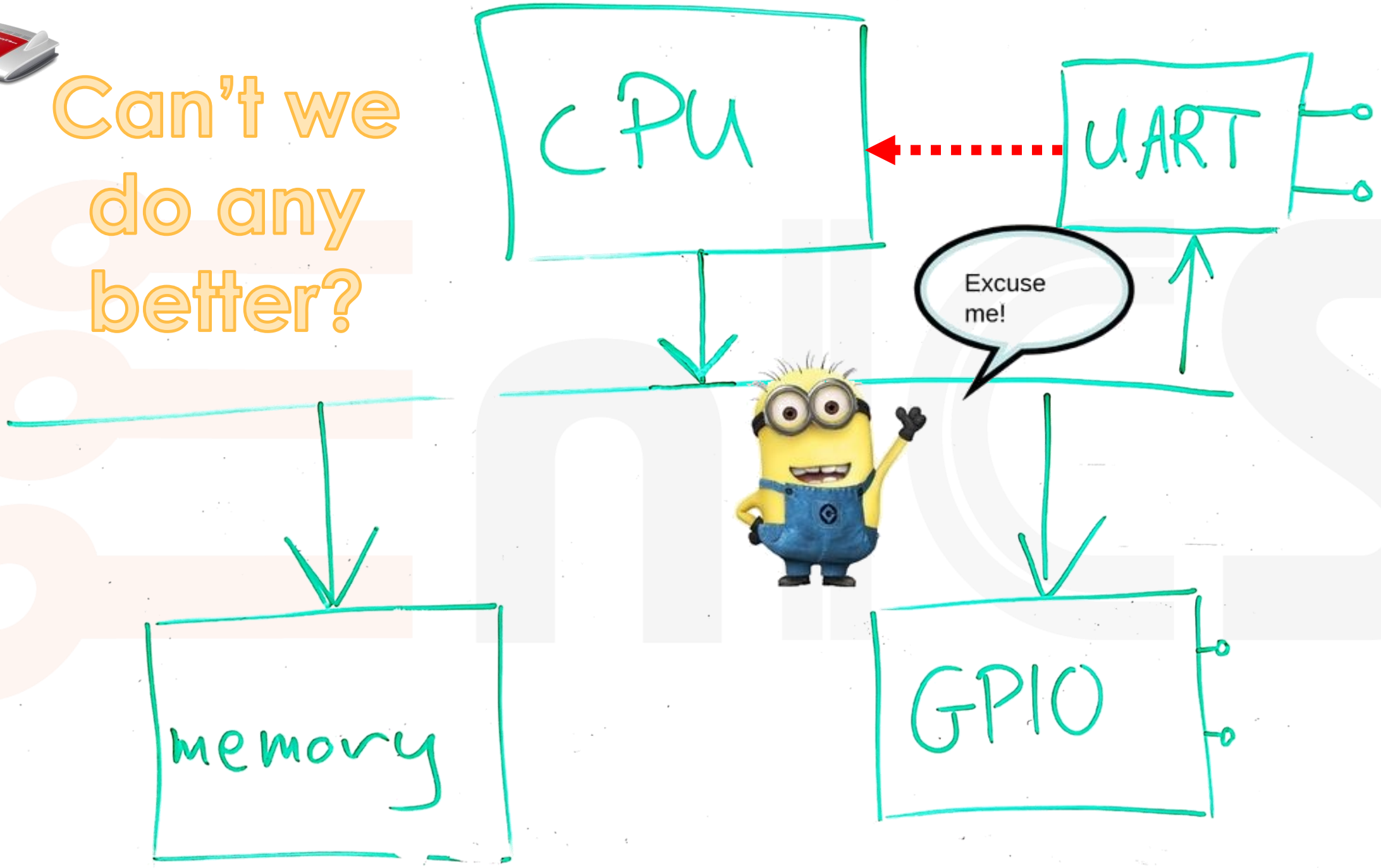
Busy-wait on input from the UART

```
while (TRUE) {
  // Wait until a new character has been read
  while (peek(UART_IN_STATUS)==0);
  // Read the new character
  achar=(char)peek(UART_DATA);
}
```

Busy-wait on writing to the UART

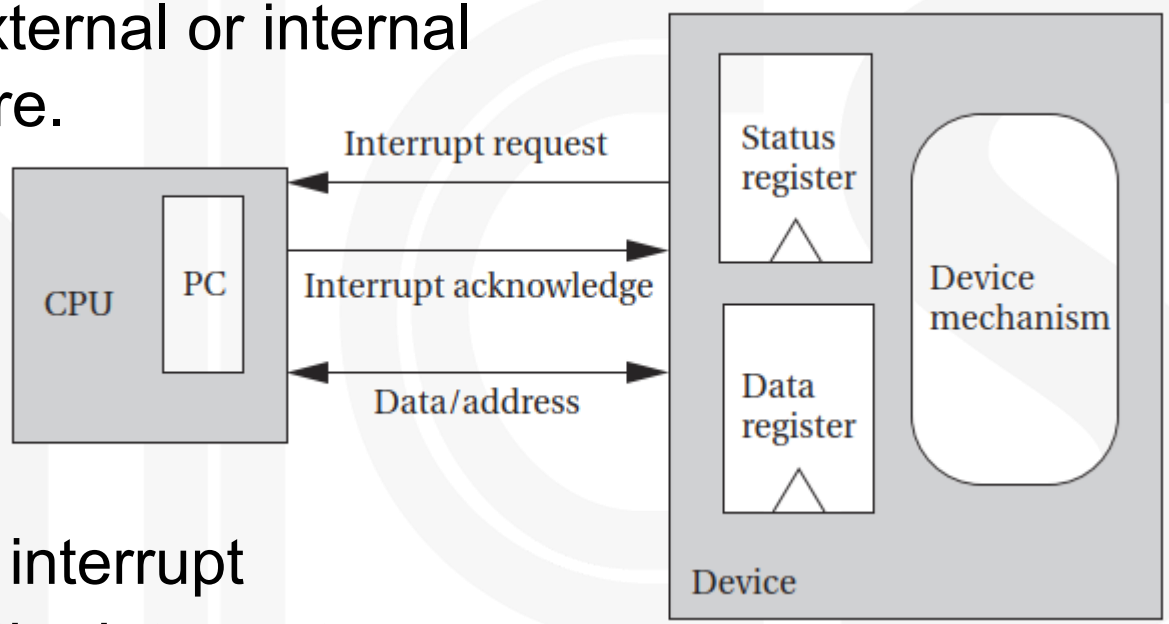
```
current_char = mystring;
// Continue until the end of string
while (*current_char != '\0') {
  // Wait until the UART is ready
  while (peek(UART_OUT_STATUS)!=0);
  // Send character to UART
  poke(UART_DATA_OUT, *current_char);
  // update character pointer
  current_char++;
}
```

Can't we do any better?



Interrupts

- **An interrupt is an asynchronous signal from a peripheral to the processor.**
  - Can be generated from peripherals external or internal to the processor, as well as by software.
  - Frees up the CPU, while the peripheral is doing its job.



Source: Computers as Components

- **Upon receiving an interrupt:**
  - The CPU decides when to *handle* the interrupt
  - When ready, the CPU *acknowledges* the interrupt
  - The CPU calls an interrupt service routine (ISR)
  - Upon finishing, the ISR *returns* and the CPU continues operation

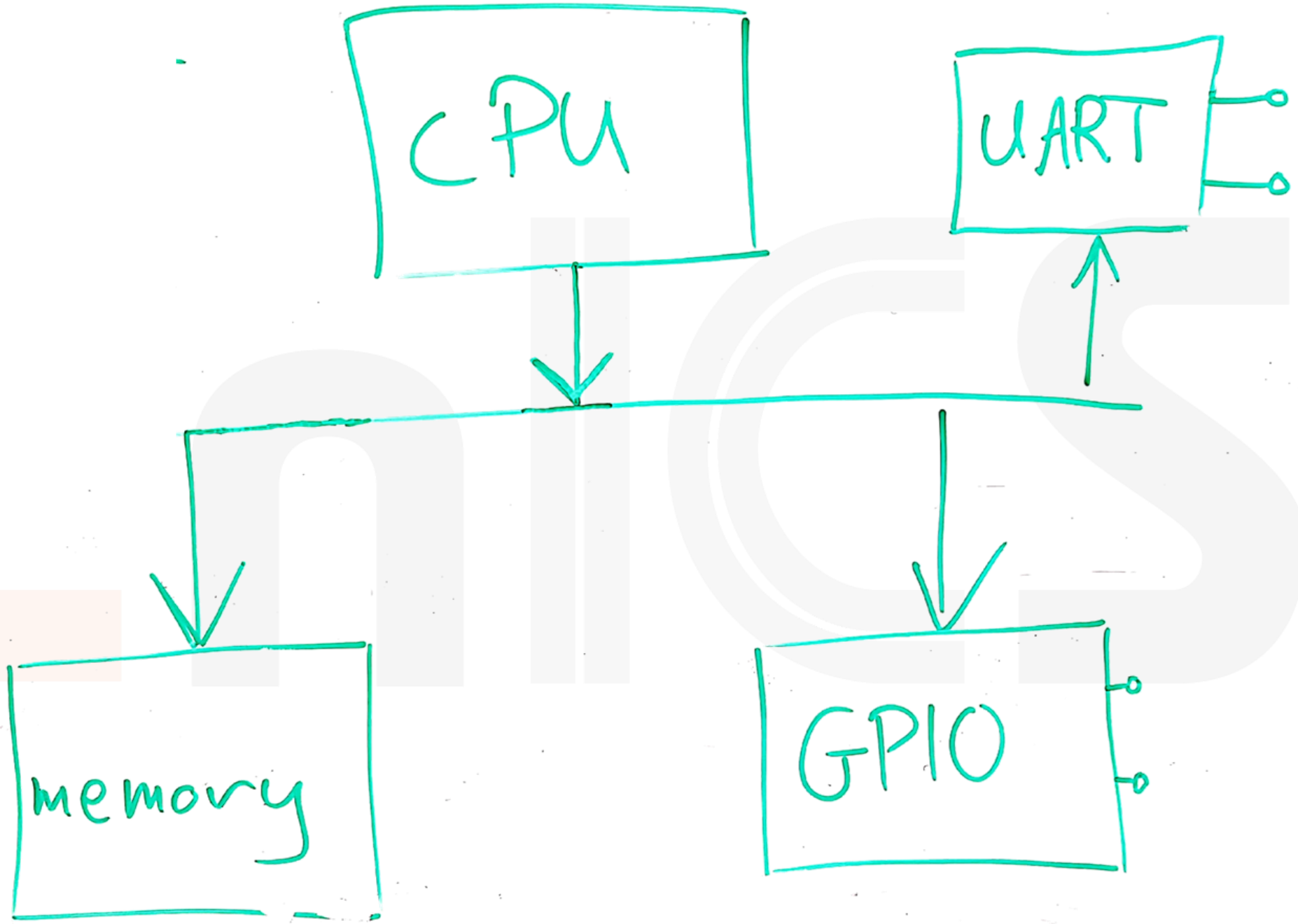
Dealing with Faster Interfaces

Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

UART

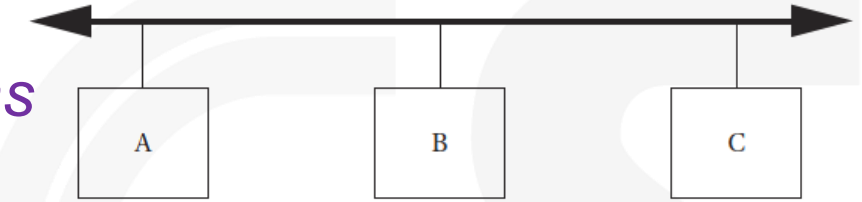
RATE: 115200 bit/sec
0.1152 Mbps

RANGE: 15m


CPU
UART
memory
GPIO

Ethernet

- **Widely used for realization of Local Area Networks (LANs)**
  - Bus with single signal path
  - Originally: Nodes are not synchronized → *Collisions*
  - Arbitration: "Carrier Sense Multiple Access with Collision Detection (CSMA/CD)"
  - If collision → wait for random time → retransmit.
  - Now: switched (point-to-point), fully duplex



Source: Computers as Components

- **Ethernet packet:**
  - Addresses
  - Variable-length data payload: 46 – 1518 bytes

Preamble	Start frame	Destination address	Source address	Length	Data	Padding	CRC

- **Throughput:**
  - 10M = 2.5 x 4bit 100M = 25 x 4bit 1G = 125 x 8bit

Side note: The OSI Model

Ethernet

- **The Open Systems Interconnection (OSI) model defines seven network layers.**
  1. **Physical**: electrical and physical components
  2. **Data Link**: Peer2Peer communication across a singe physical layer.
  3. **Network**: basic routing over the link.
  4. **Transport**: ensure data is delivered in the proper order and without errors across multiple links.
  5. **Session**: interaction of end-user services across a network
  6. **Presentation**: defines data exchange formats including encryption and compression.
  7. **Application**: interface between the network and end-user

Application	End-use interface
Presentation	Data format
Session	Application dialog control
Transport	Connections
Network	End-to-end service
Data link	Reliable data transport
Physical	Mechanical, electrical

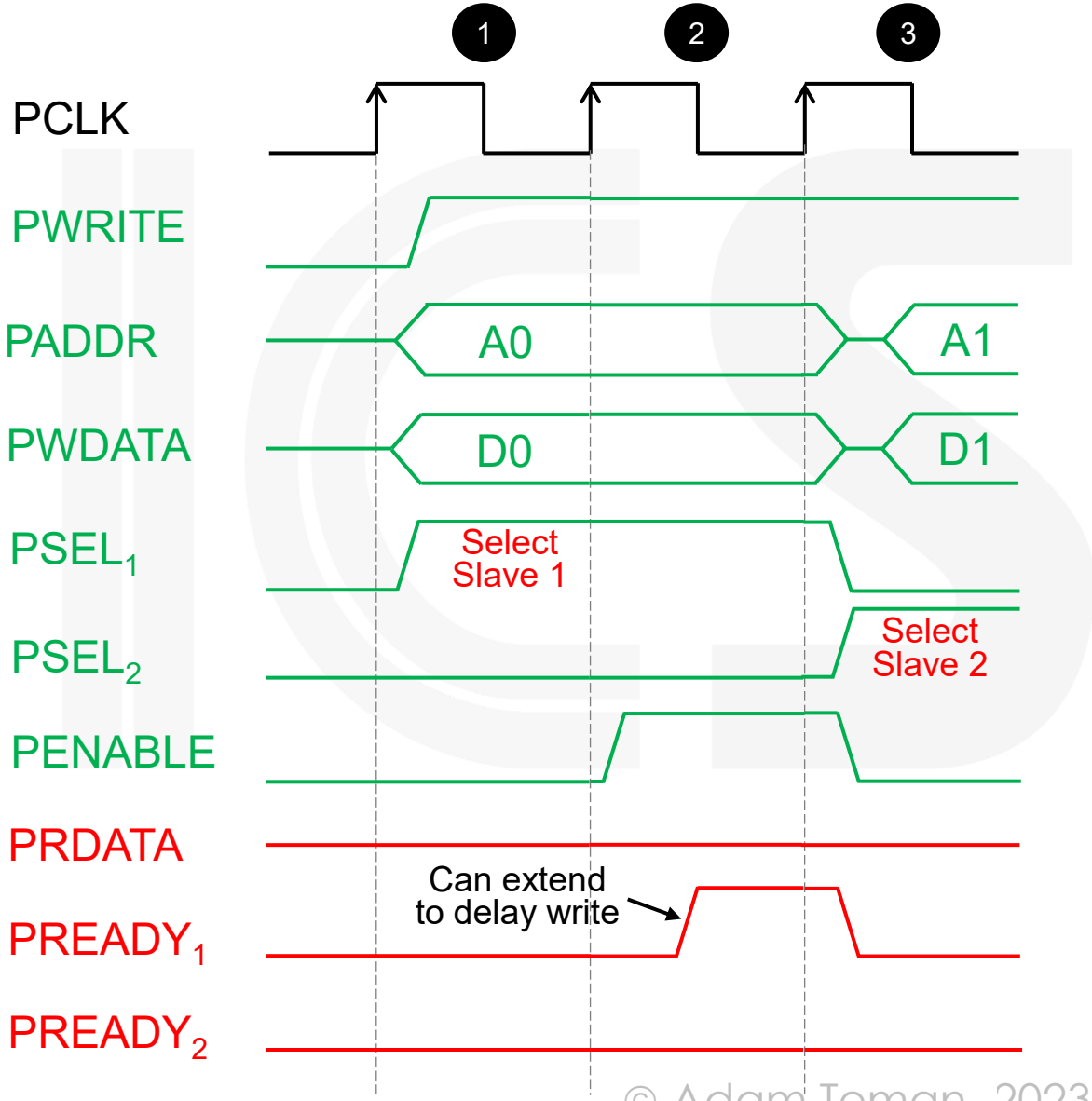
Source: Computers as Components

Ethernet

Let's try a simple interface: APB

- **32-bit bus**
- **Two phase access:**
  - Address (*Setup*) phase
  - Read/Write (*Access*) phase

1. **Setup Phase**
   - PWRITE, PADDR, PWDATA are set
   - PSEL is raised for selected Slave
2. **Access Phase**
   - PENABLE is raised, with other signals held
   - When selected Slave acks, PREADY is raised
3. **Next Transfer**
   - PENABLE is lowered by Master
   - PREADY may be lowered by Slave



Is APB Sufficient?

ETH RATE: 10^9 bits/sec

APB TRANSFER WIDTH: 32 bits ↑

APB RATE: 2 cycles/transfer ↓

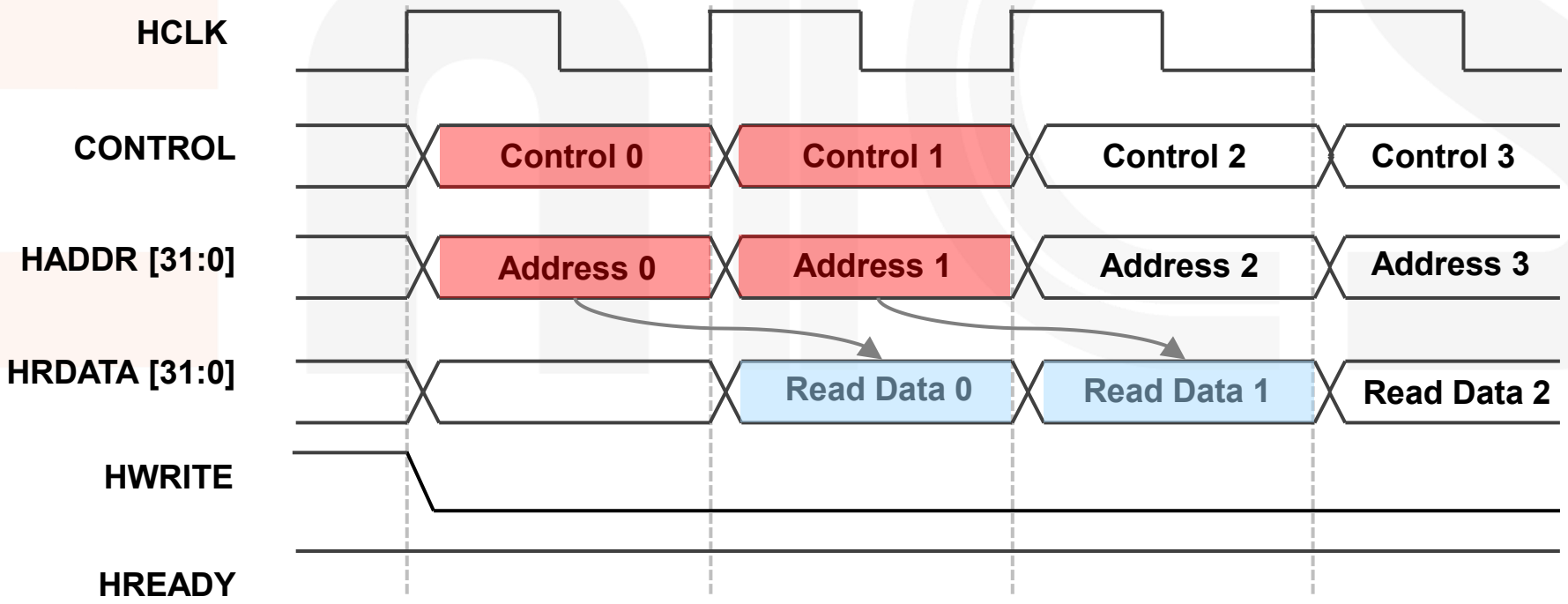
CLOCK: 10^8 cycles/sec

APB THROUGHPUT: 1.6×10^9 bits/sec

BUT, Ethernet is FULL-DUPLEX,
so maximum throughput is actually
2×10^9 bits/sec

So let's make it faster: AHB

- Wider bus (>32 bits)
- Pipelined address and R/W phases (X2 throughput)
- Supports Bursts



ethernet

CPU

UART

AHB

memory

GPIO

Is AHB fast enough?

ETHERNET RATE: 2×10^9 bits/sec

AHB TRANSFER WIDTH: 64 bits

AHB RATE: 1 cycle/transfer

CLOCK: 10^8 cycles/sec

AHB THROUGHPUT: 6.4×10^9 bits/sec

BUT WHAT ABOUT THE CPU?

More Offloading

Emerging Nanoscaled
Integrated Circuits and Systems Labs
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Is AHB fast enough?

ETHERNET RATE: 2×10^9 bits/sec

AHB TRANSFER WIDTH: 64 bits

AHB RATE: 1 cycle/transfer

CLOCK: 10^8 cycles/sec

AHB THROUGHPUT: 6.4×10^9 bits/sec

BUT WHAT ABOUT THE CPU?

Can the CPU support this?

ETHERNET RATE: 2×10^9 bits/sec

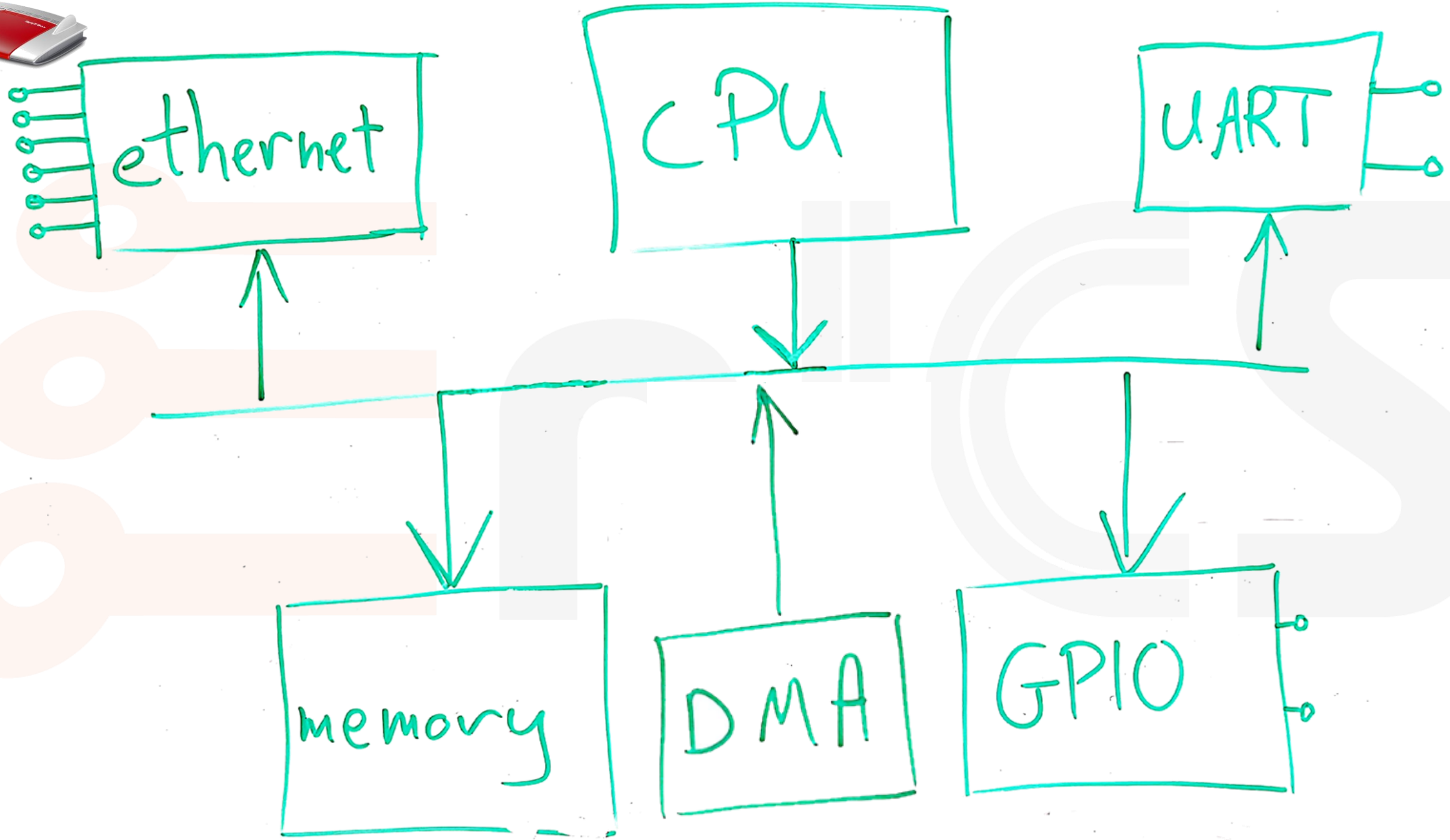
CPU WORD: 32 bits

INSTRUCTIONS PER SW/LW: 3 inst/load

CLOCK: 10^8 cycles/sec

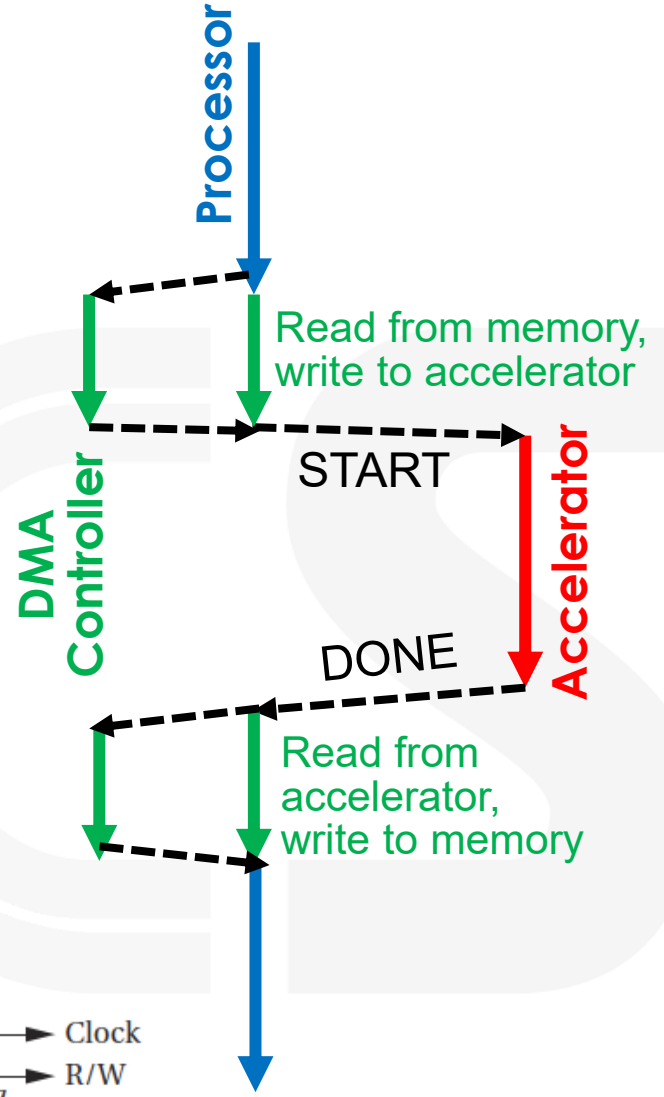
CPU THROUGHPUT: 1.1×10^9 bits/sec

So the CPU is again too slow…

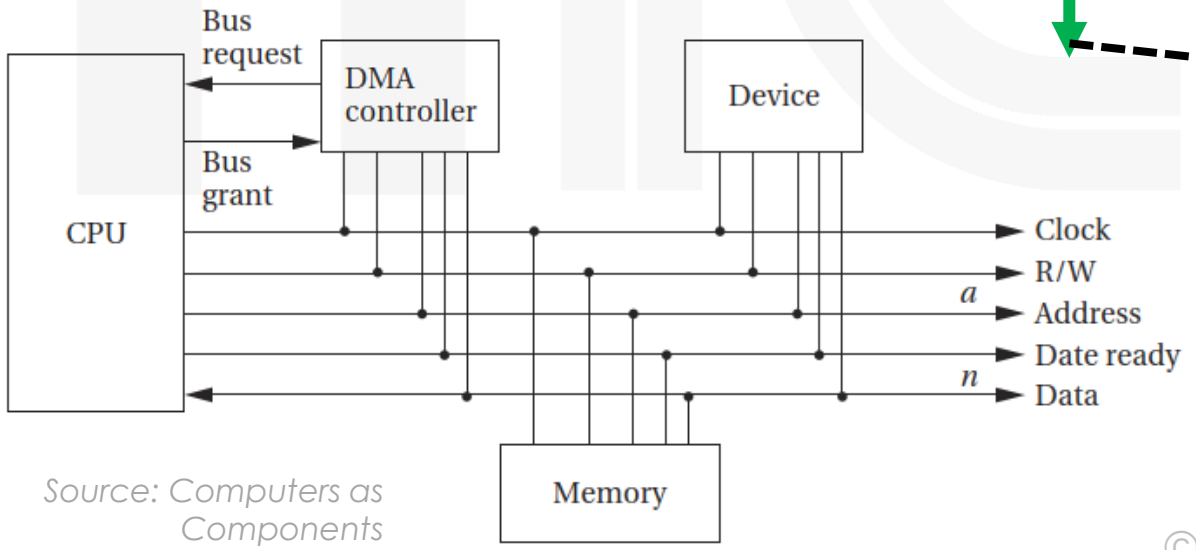

ethernet
CPU
UART
memory
DMA
GPIO

DMA

- **Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU.**
  - A DMA transfer is controlled by a DMA controller that requests control of the bus from the CPU.
  - With control, the DMA controller performs read and write operations directly between devices and memory.



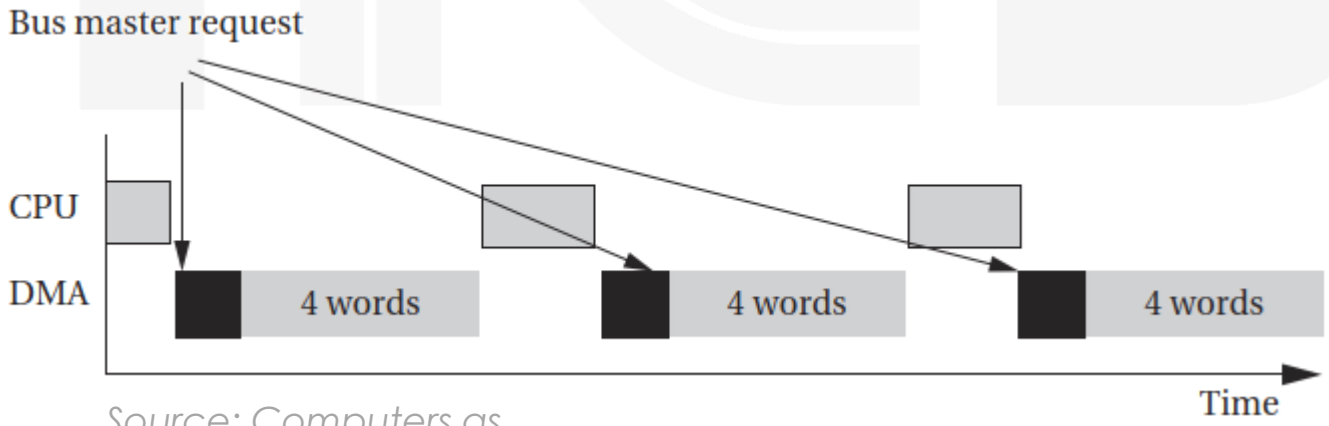
- **DMA adds new signals:**
  - Bus request
  - Bus grant



Source: Computers as Components

DMA Registers

- **The CPU controls the DMA operation through registers in the DMA controller.**
  - Starting address register
  - Length register
  - Status register – to start and stall the DMA
- **After the DMA operation is complete, the DMA controller interrupts the CPU to tell it that the transfer is done.**
- **DMA controllers usually use short *bursts* (e.g., 4-16 words) to only occupy the bus for a few cycles at a time**



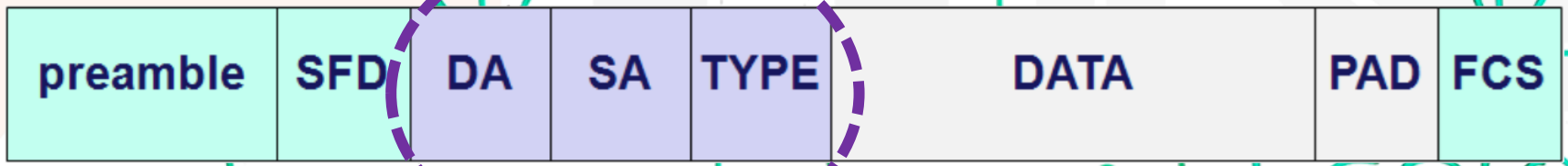
Source: Computers as Components

ethernet

CPU

UART

BUT WHAT ABOUT THE CPU?



memory

DMA

GPIO

data field: min. 46 bytes, max 1500 bytes

Data Rate or Packet Rate?

BIT RATE: 2×10^9 bits/sec

each packet includes 20 bytes of overhead

DATA RATE: BIT-RATE x P/(P+20)

P=64 → $2 \times 0.76 \times 10^9$ bits/sec

P=1518 → $2 \times 0.98 \times 10^9$ bits/sec

Larger packets mean interconnect is busier

PACKET RATE: BIT-RATE / ((P+20) x 8)

P=64 → $2 \times 1.48 \times 10^6$ packets/sec

P=1518 → $2 \times 81.2 \times 10^3$ packets/sec

Smaller packets mean CPU has to do more

How does this affect the CPU?

- **The CPU (in a router) needs to handle the packet**
  - i.e., figure out where to send the packet to.
  - So, all it cares about is *packet rate*
- **How much work can the CPU do on each packet?**
  - For packets with 1518 bytes of data, the packet rate is about 160K packets/sec
  - At 100 MHz → 615 instructions per packet → Not that much
  - For 64 byte packets → 34 instructions per packet → Infeasible!

- **What can we do???**
  - Trivial solution: Raise the frequency
  - Still not enough: Add additional CPUs
  - Better solution: Integrate dedicated hardware (Accelerators and ASIPs)

A typical router SoC

PARSER

CLASSIFIER

POLICER

TRAFFIC MANAGEMENT

ASIP 0

ASIP 1

ASIP 2

HOST

I/F 0

I/F 1

I/F 2

I/F 3

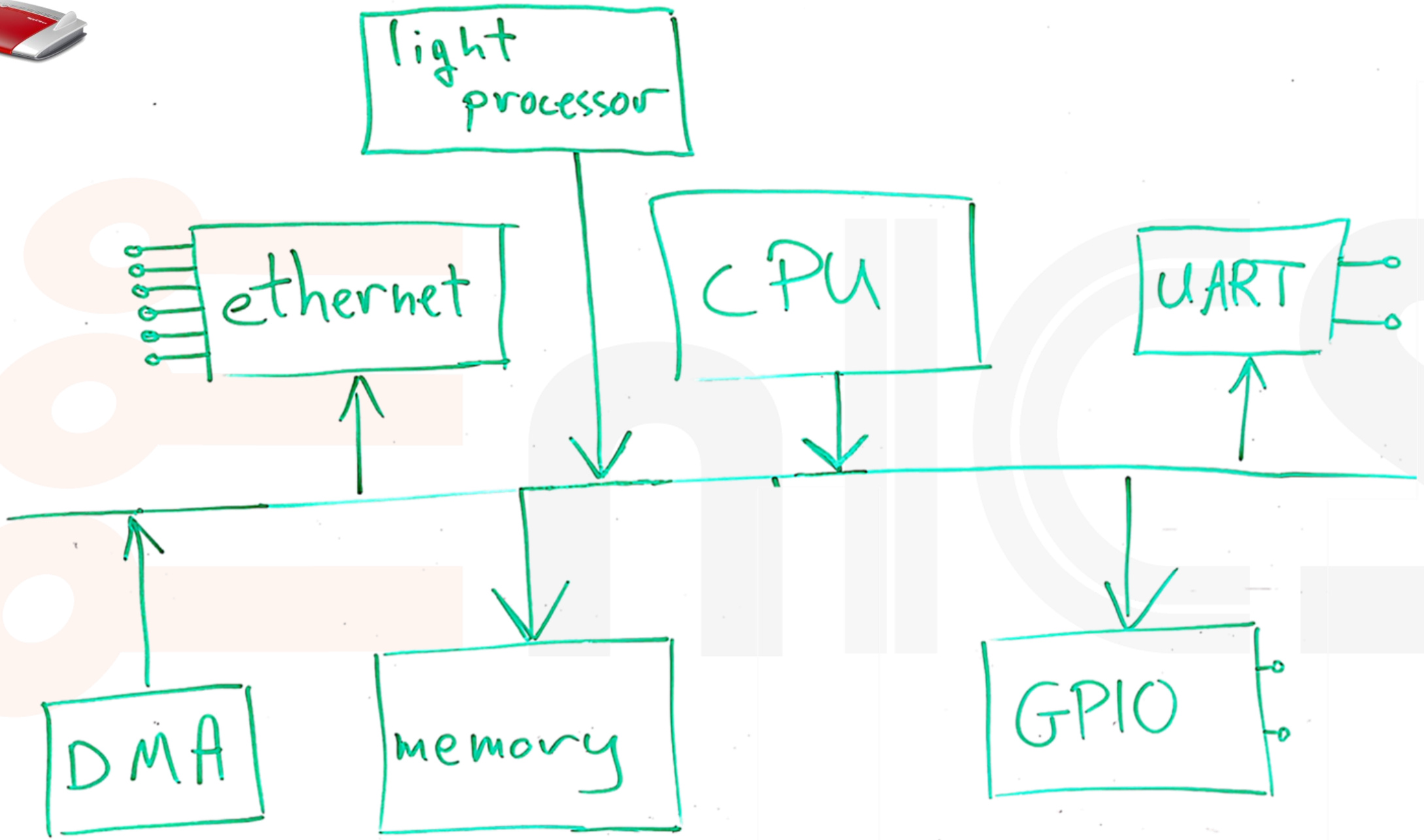
I/F 4

I/F 5

Memory

Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University


light processor
ethernet
CPU
UART
DMA
memory
GPIO

But what about memory?

- **We now have a processor that can communicate with peripherals, with an off-chip network, etc.**

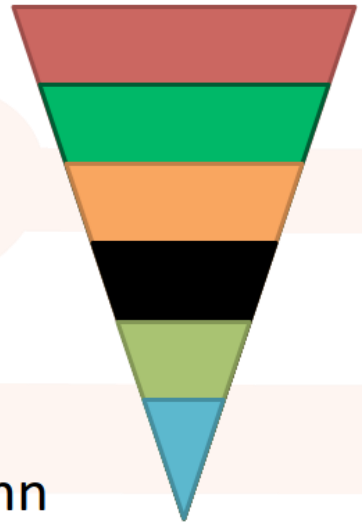
But what about memory?

- **Our router needs a lot of memory:**
  - To buffer packets
  - To store routing tables
  - To host the operating system
  - …
- **The on-chip memory (~MB) is nowhere near enough.**

We need to use DRAM

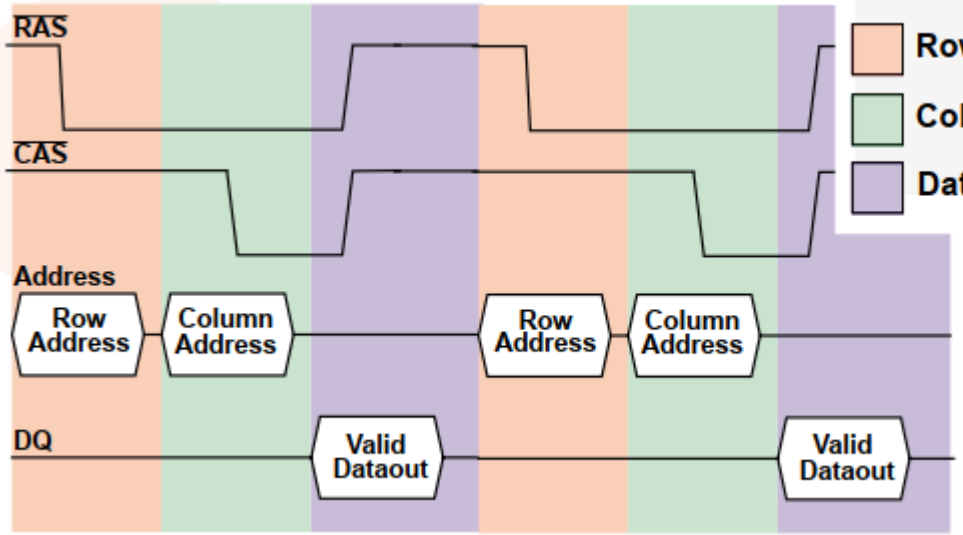
DRAM Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column

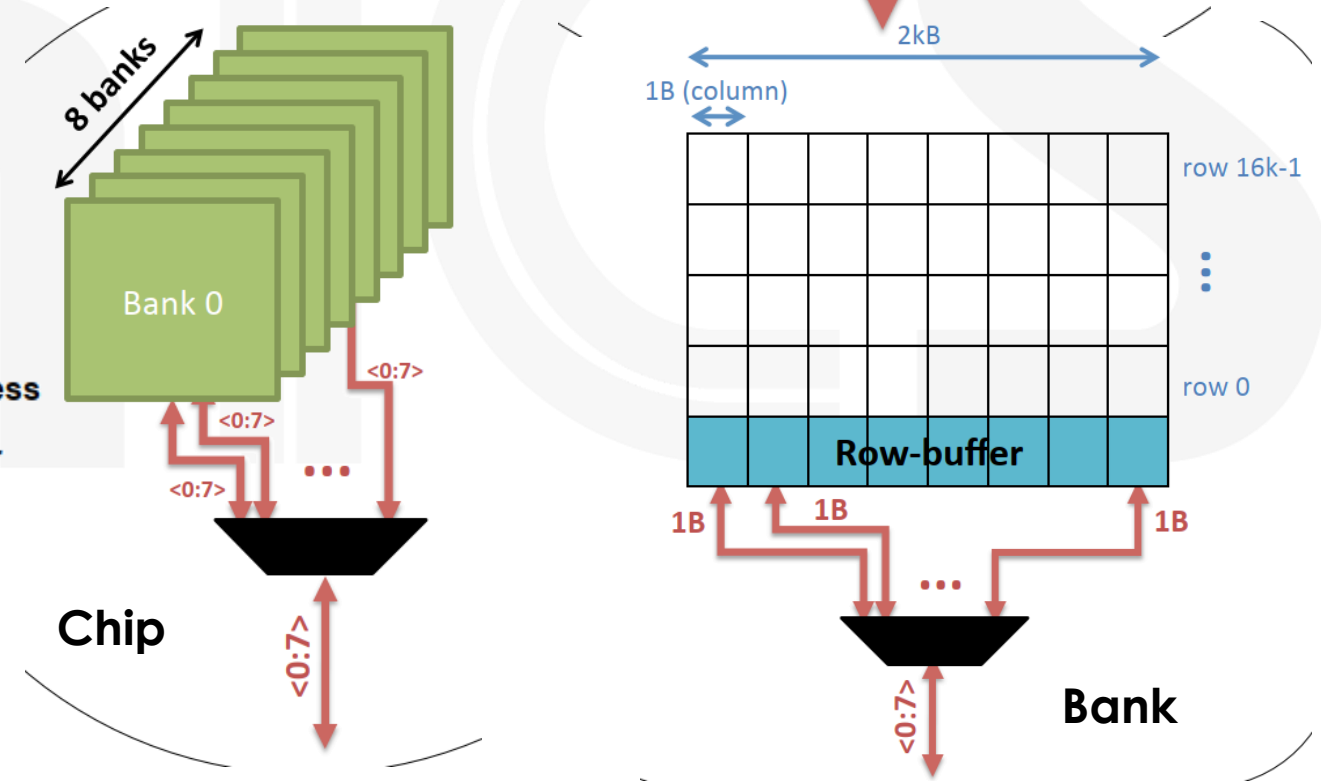


Source: Onur Mutlu

DIMM (Dual in-line memory module)

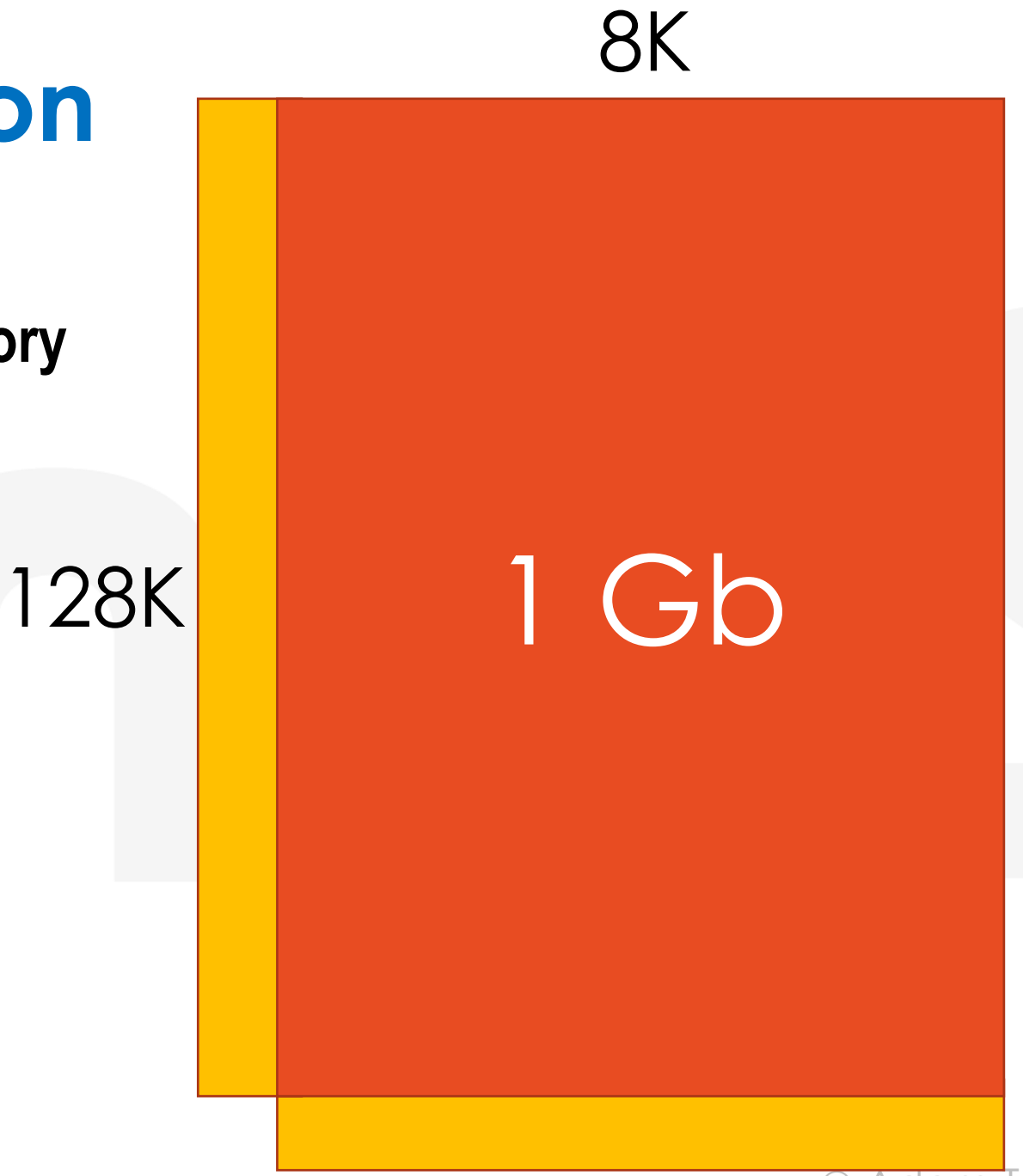


Source: Bruce Jacob



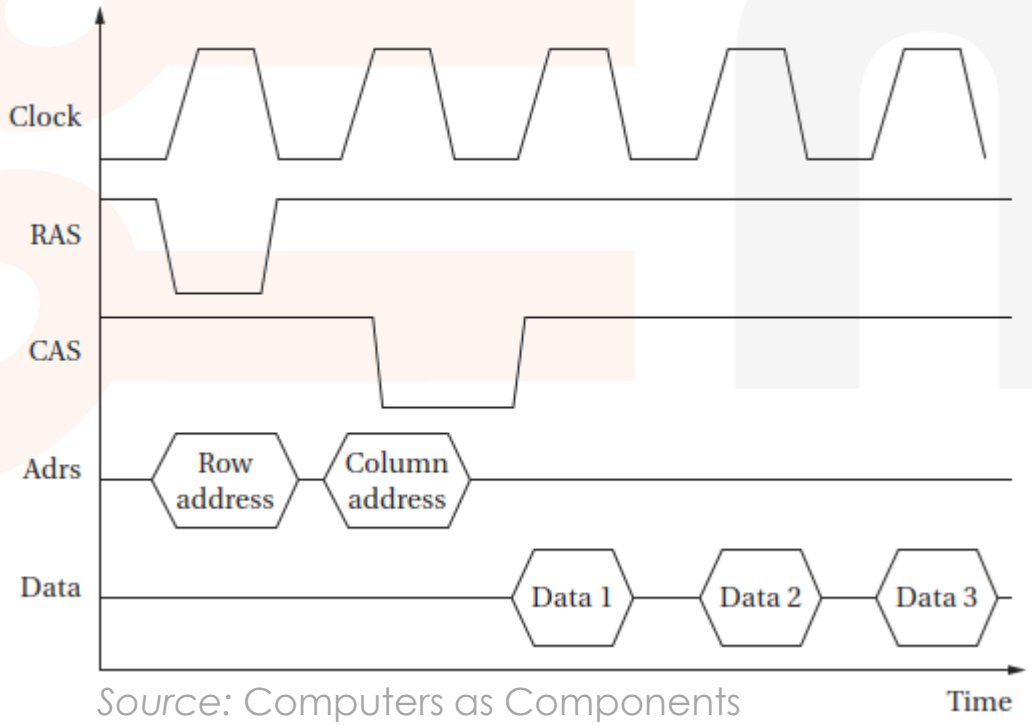
DRAM Organization

- **So, for a 1GB DIMM** (i.e., 8 chips)**, we need chips with 1Gb of memory**
  - e.g., 128k x 8k
- **But that is a lot of rows…**

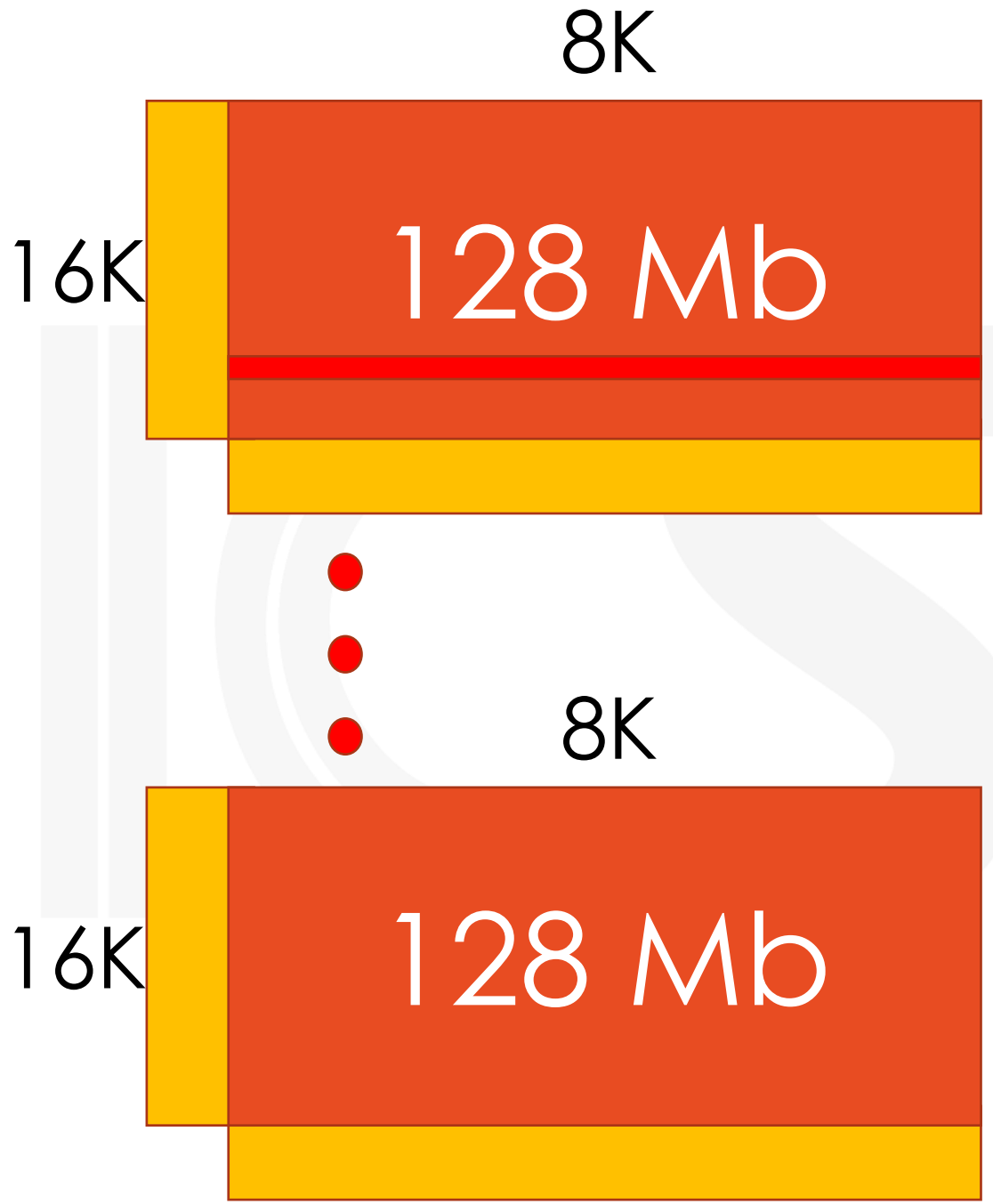


DRAM Banks

- **So, we break it into 8 banks of 16k rows**
  - And readout an entire row to a buffer
  - Once the row is buffered, we can directly access any byte in the buffer.

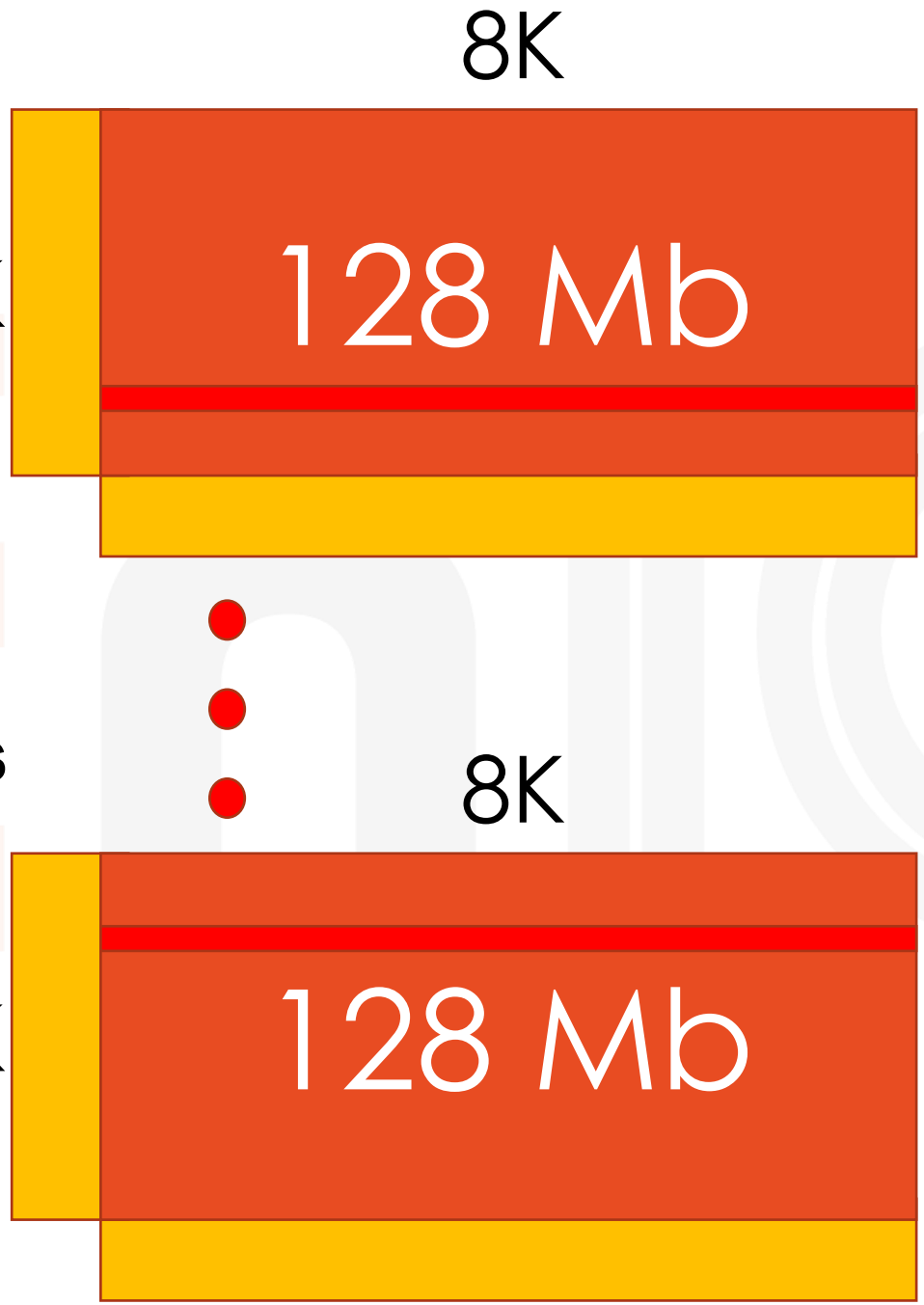


Source: Computers as Components



DRAM Banks

- Reordering the memory accesses can improve throughput
- Add a dedicated DDR controller to optimize memory access patterns

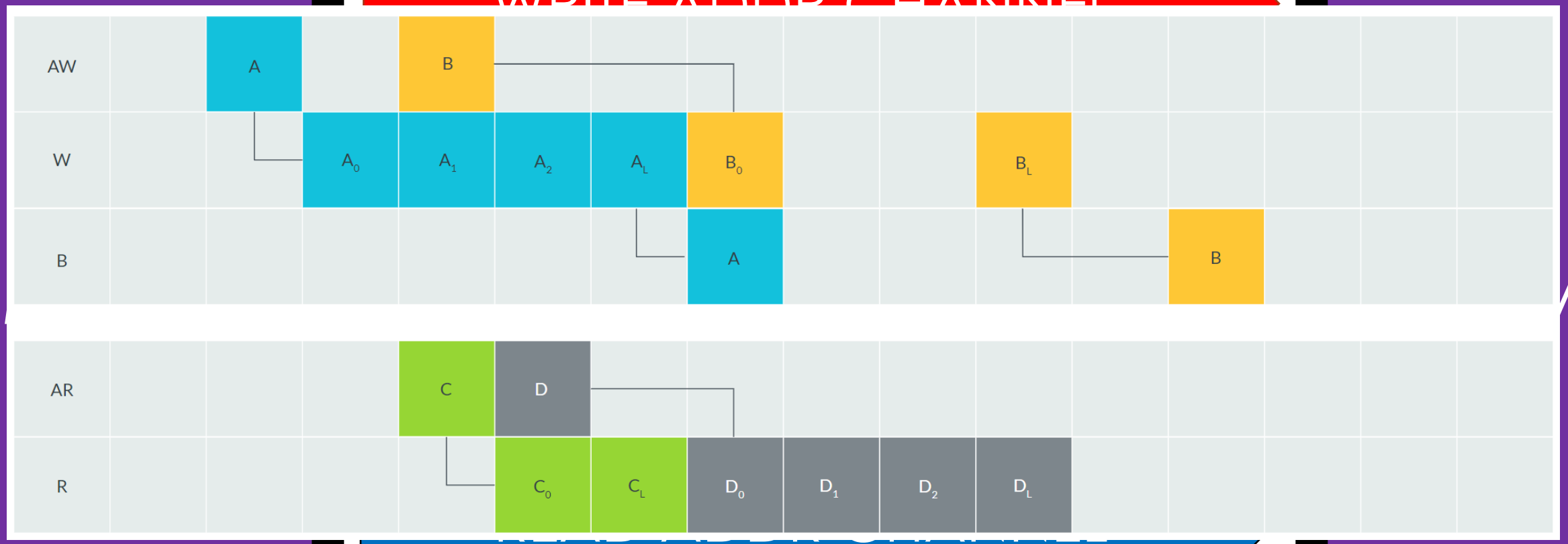


B0/R0
B1/R1
B0/R0
B1/R1
B0/R0
B1/R1
B0/R3
B0/R0
B1/R1
B1/R2
B0/R3
B1/R1
B1/R2

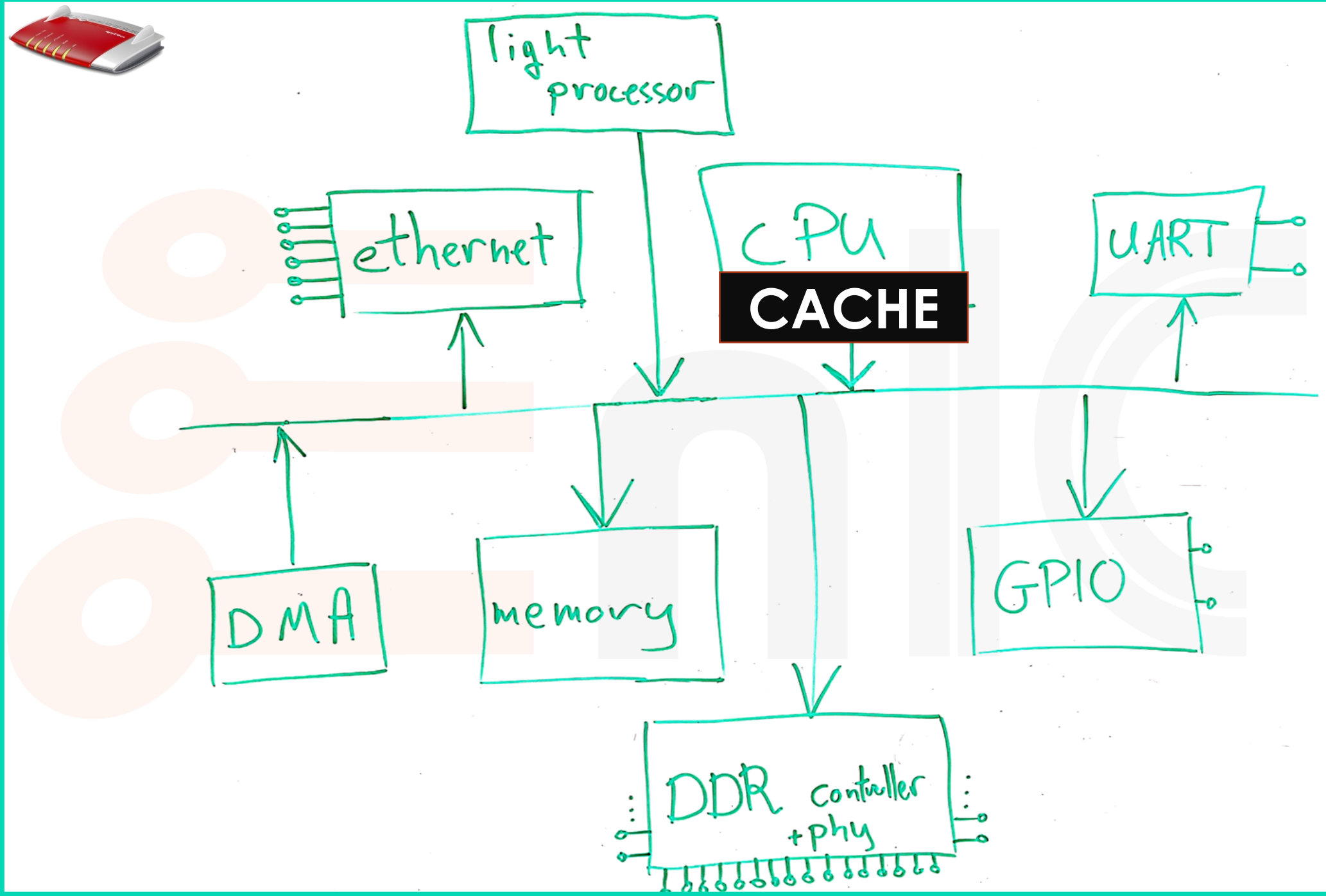
Finishing our Design

In AHB, we always have to wait until a transaction is finished before starting a new one…

AXI



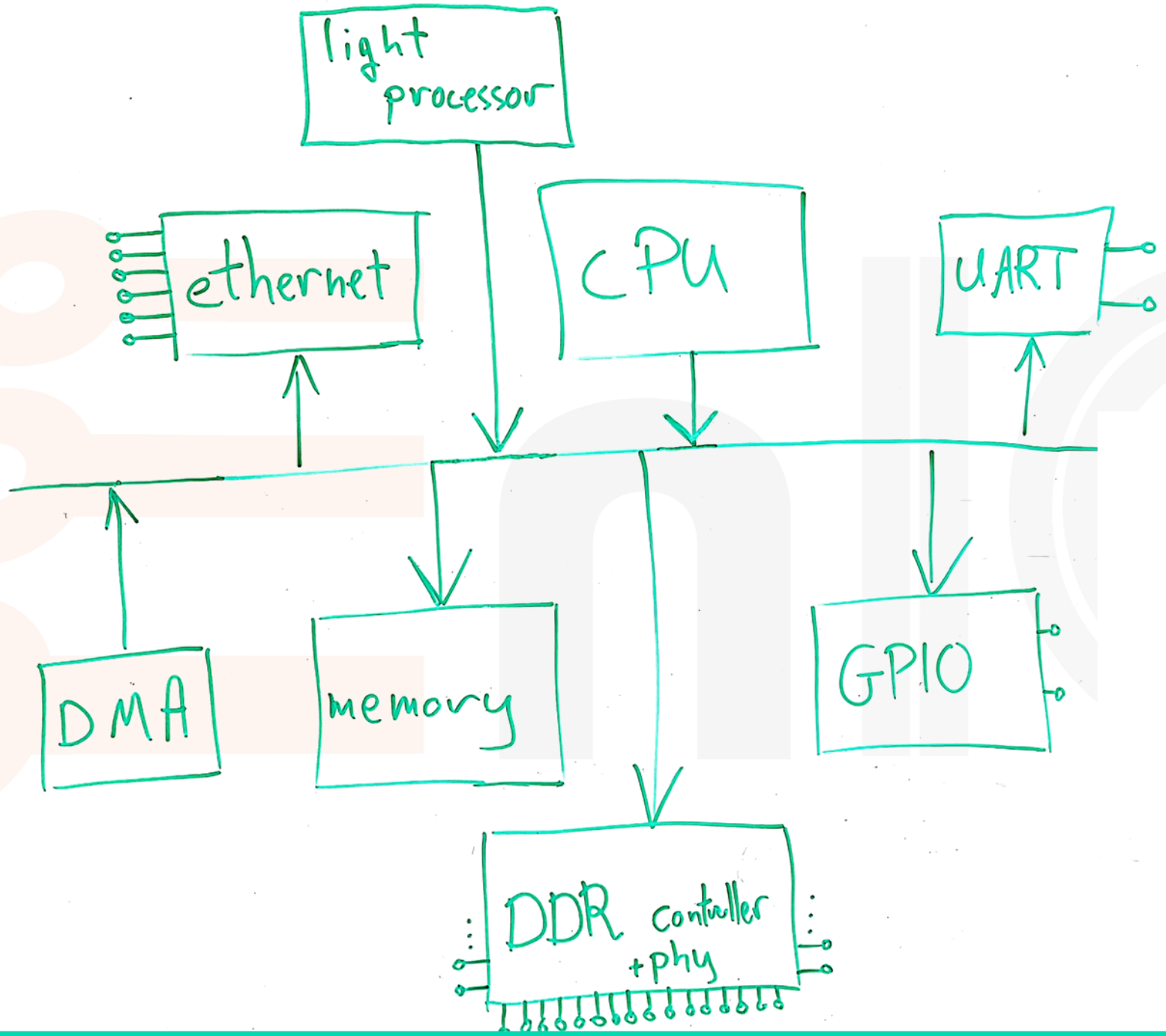
READ DATA CHANNEL


light processor
ethernet
CPU
CACHE
UART
DMA
memory
GPIO
DDR controller +phy

But what happens on Startup?

Source: Intel

- **Start by reading from a BootROM**
  - A small piece of memory, which contains the very first code that is executed upon reset.
  - Either hard-wired (mask-ROM) or rewriteable (embedded Flash)
  - Can use bootstraps/fuses to change configuration.
- **Then move on to the Bootloader**
  - Usually stored on rewriteable flash (i.e., SD card)
  - Configures the chip and some of the peripherals
  - Loads the end application (e.g., OS) from storage (flash, SSD, HDD)
  - Passes control to the end application
- **The BootROM and bootloader can be combined**
- **The BootROM of an x86 system is called the BIOS**


light processor
ethernet
CPU
UART
DMA
memory
GPIO
DDR controller +phy

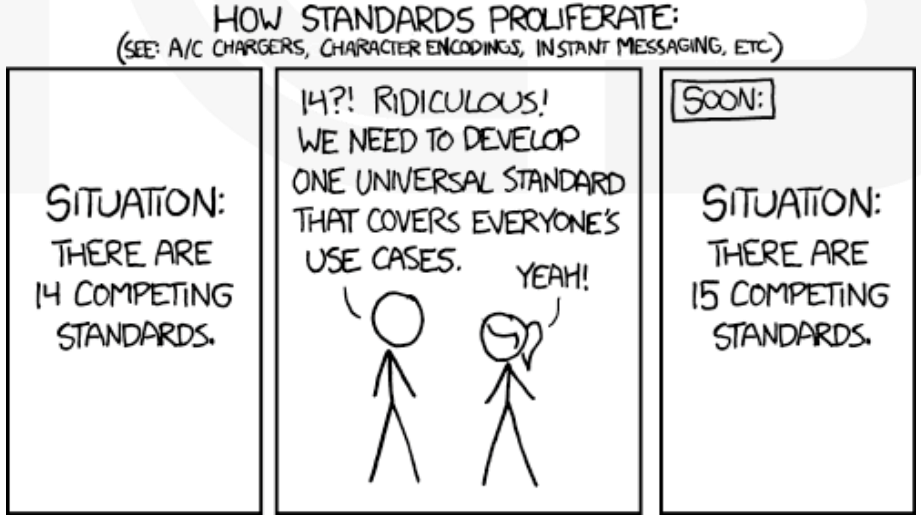
Conclusions

- Processors are great at processing.
- They are not so great at data movement.
- They are not so great at doing simple tasks.

- For a well-defined task, dedicated hardware will never lose to a processor.

- With Processors we gain flexibility.
- Software development is faster than Hardware.
- Bugs in Software are much cheaper to fix.

Parks and Recreation. Source: Giphy



Source: xkcd

Main References

- Tzachi Noy, “Interfaces: External/Internal, or why CPUs suck”, 2019
- Wolf, “Computer as Components - Principles of Embedded Computing System Design,” Elsevier 2012