EEE447 Introduction to Microprocessors

Course Overview

Course Schedule

Tuesday 11:40–12:30 **(@A206)**

Thursday 10:40–12:30 **(@A206)**

Textbook

- Embedded Systems: Introduction to ARM Cortex-M Microcontrollers, Fifth Edition, Jonathan W Valvano
- The Definitive Guide to ARM Cortex M3 and Cortex M4 Processors, Third Edition, Joseph Yiu
- Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C (Second Edition), Oct 2015, Yifeng Zhu.

Other references:

- Cortex M4 Technical Reference Manual
- Cortex M4 Devices Generic User Guide
- Tiva TM4C123G Microcontroller Data Sheet
- Tiva TM4C123G Development Board User's Guide
- ARM and Thumb-2 Instruction Set Quick Reference Card

Grading

- MT Exam 1 20 %
- Final Exam 25 %
- Project 15 %
- Quiz+Hw+Att 15 %
- Laboratory Work 25 %

Course objectives

- To develop an in-depth understanding of
  - the operation of microprocessors and microcontrollers,
  - machine language programming,
  - microprocessor interfacing techniques,
  - designing and interfacing of microcontroller-based embedded systems in both hardware and software,
  - basic programming of ARM Cortex chips in assembly language and learning the fundamentals of embedded system design.

- Ultimate goal: to be able to apply this knowledge to more advanced structures.

What we expect of you ...

- A desire to learn about microprocessors, microcontrollers and their applications.

- To do some work:
  - Attend the classes and study regularly,
  - Attend exams,
  - Do the laboratory assignments,
  - Submit homeworks, etc.
  - Complete the project

Course Outline

- Introduction
- ARM Architecture
- Memory concepts, address decoding
- Assembly Programming, detailed instruction set
- Programming Examples
- Stacks, Subroutines, Interrupts
- Timer Subsystem
- Parallel I/O System
- Analog to Digital Conversion System
- Serial System

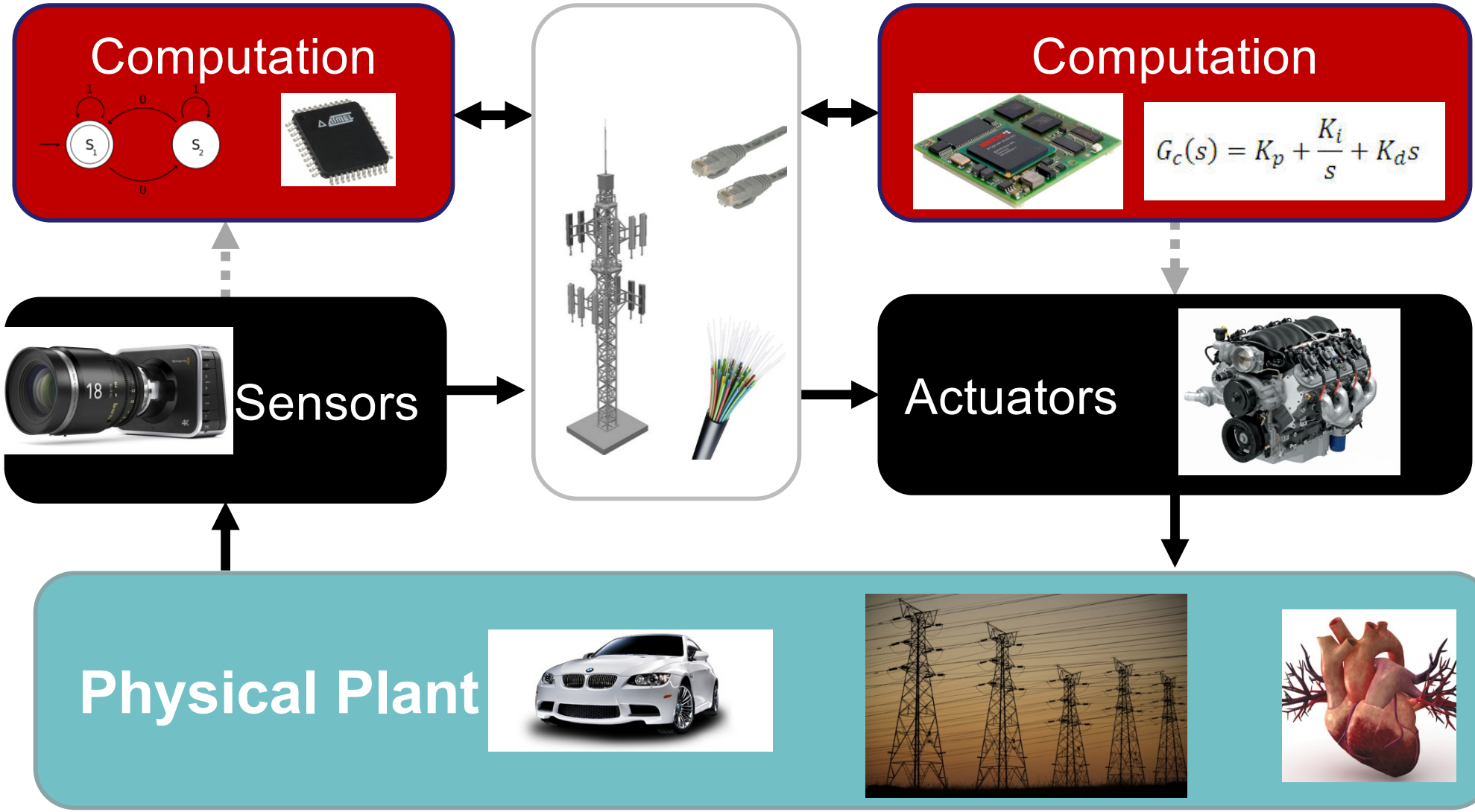
EEE447 Introduction to Microprocessors

Chapter 1

Introduction

(Microprocessor, microcontroller and programming basics)

What do we mean by Embedded Systems?

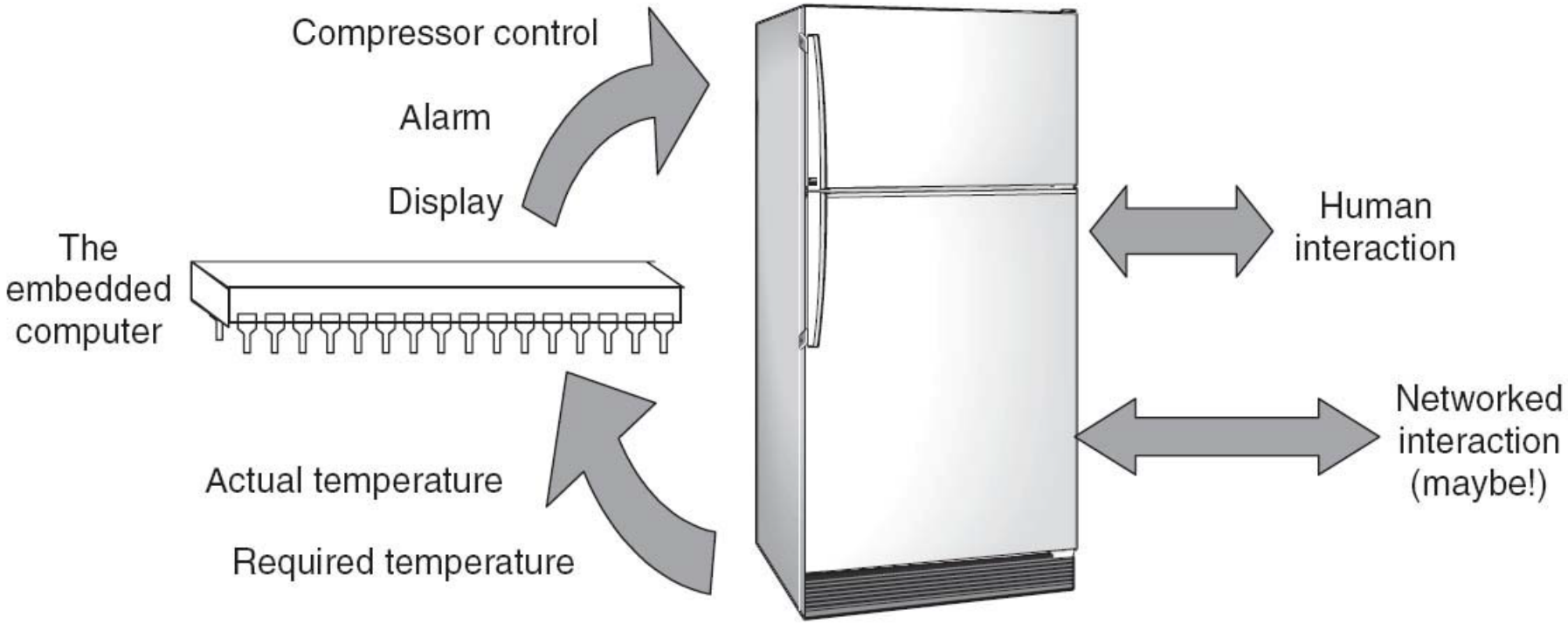


- It is a special purpose system that is used to perform one of few dedicated functions.
- Simply any computer system embedded inside an electronic device but is not itself a general-purpose computer.

  – dedicated to specific function(s)
  – interact with environment
  – real-time requirements

Embedded Systems

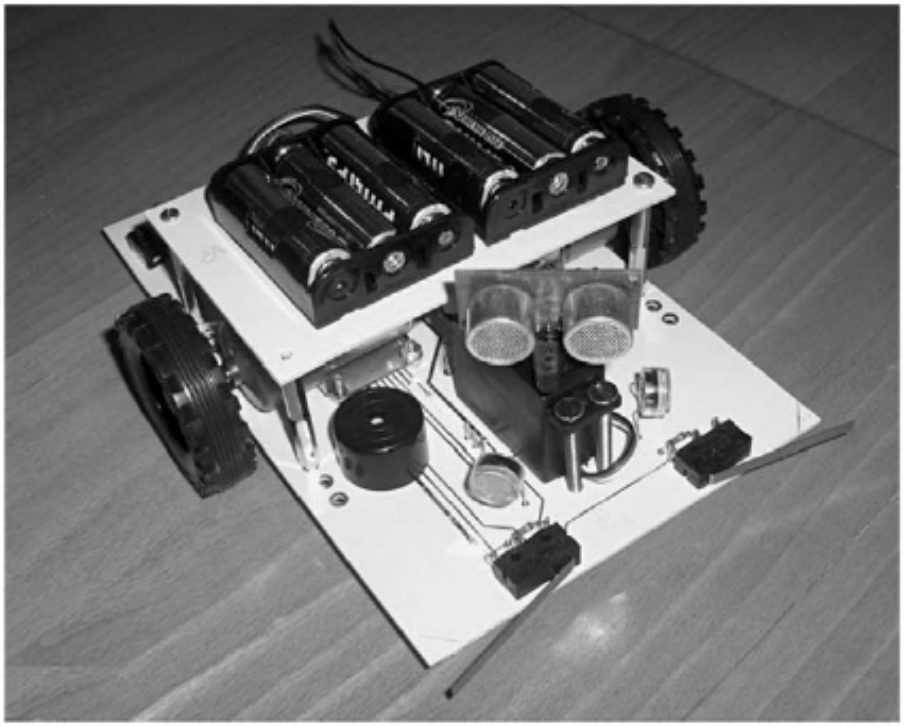
EX:

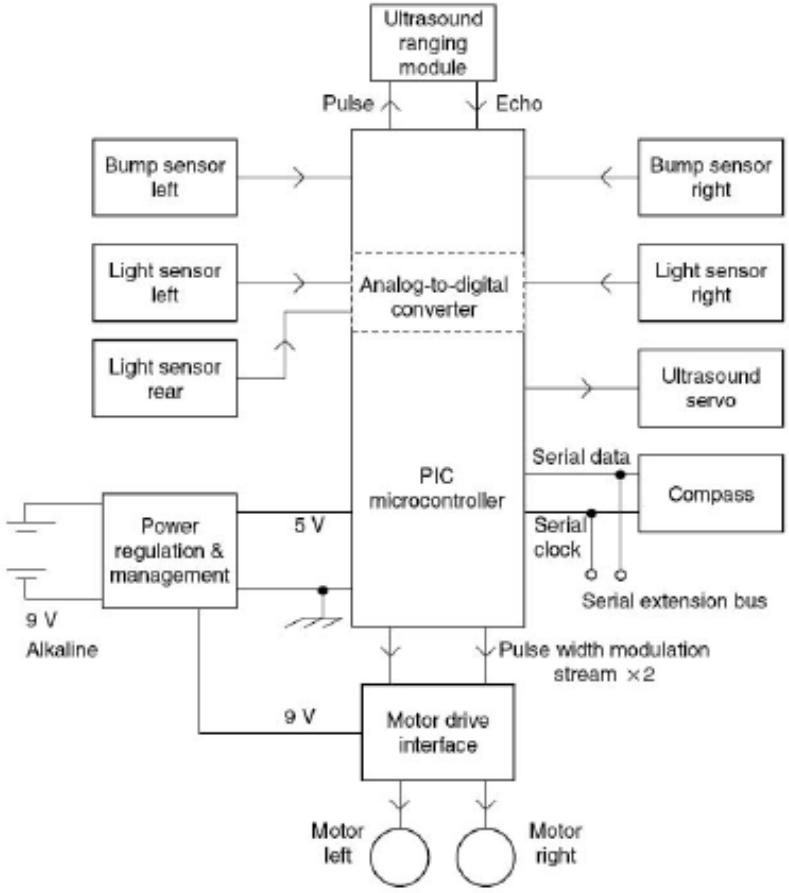


Embedded system example 1: the refrigerator

EX:

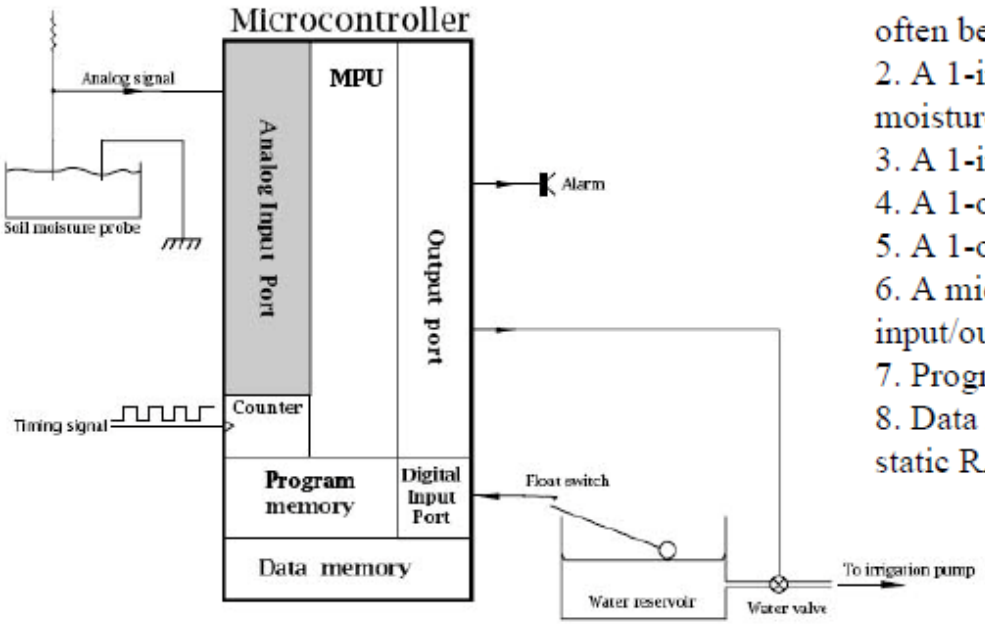
- Autonomous Guided Vehicle (AVG)





EX:

- A greenhouse controller is to monitor an analog signal from a soil moisture probe and if below a certain value turn on a water valve for 5 seconds and off for 5 seconds.

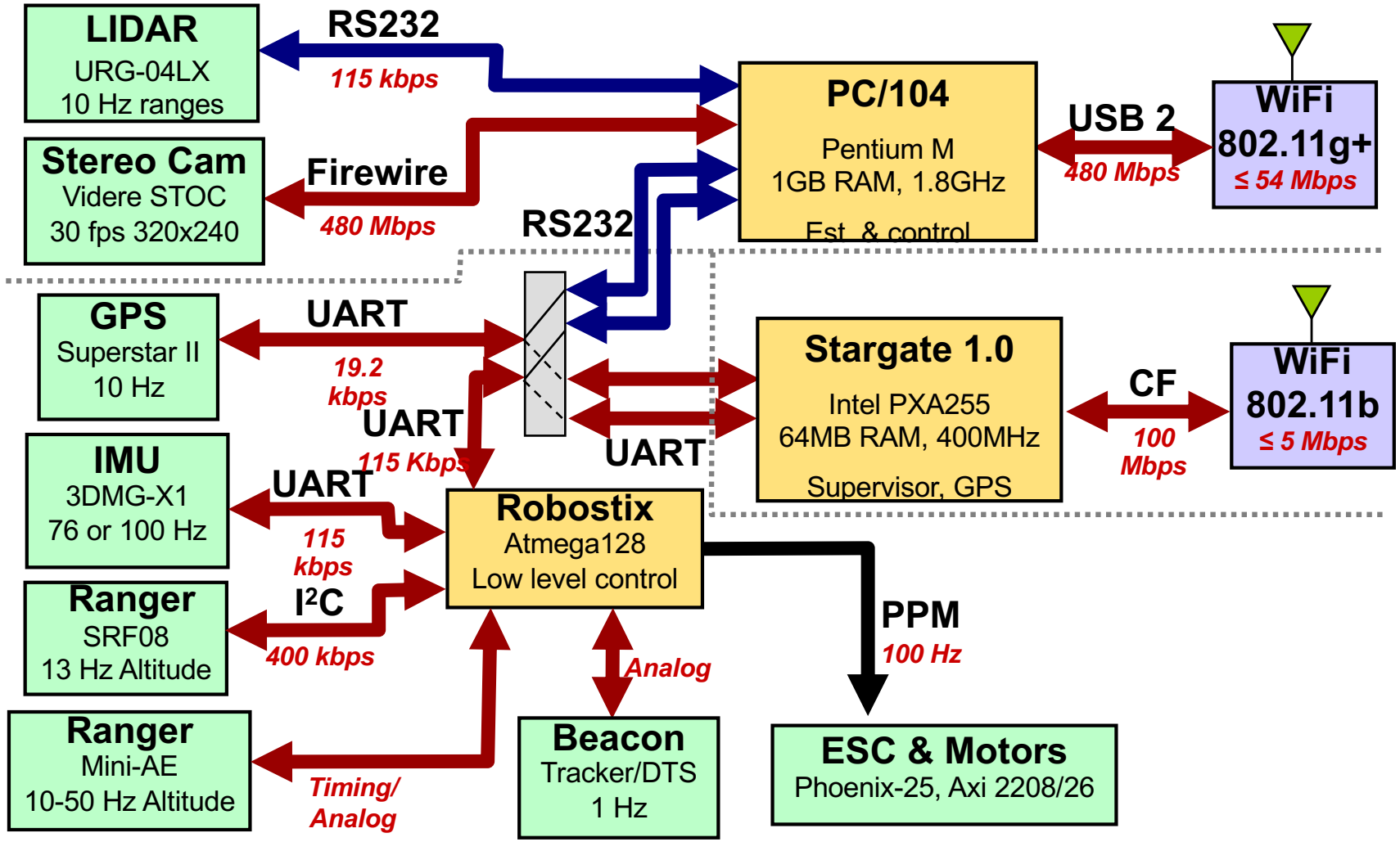


Resource Budget (List of subsystem):

1. An input for an external oscillator, connected to a counter/timer to allow the MCU to calculate time. In practice the system clock can often be used by this internal timer to measure duration.
2. A 1-input analog input line to measure the analog signal from the moisture detector
3. A 1-input digital line to check the level of the reservoir water tank
4. A 1-output digital line to open and close the water valve
5. A 1-output digital line to activate the buzzer alarm
6. A microprocessor to do the calculations and to read/write to the input/output ports, respectively
7. Program memory, usually ROM of some kind
8. Data memory for temporary storage of program variables, usually static RAM

EX:

STARMAC quadrotor aircraft

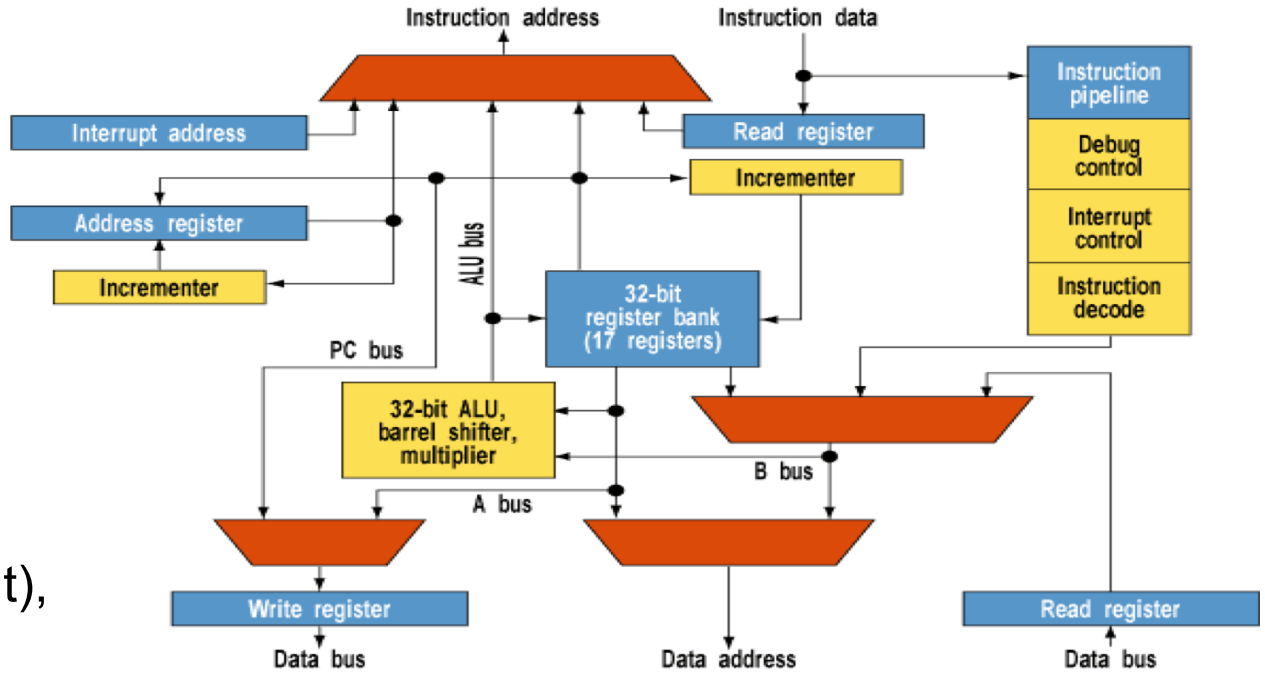


Since this is an introductory course to processors and embedded systems, will look at simpler systems here… but the learning will be applicable and scalable to more complex systems.

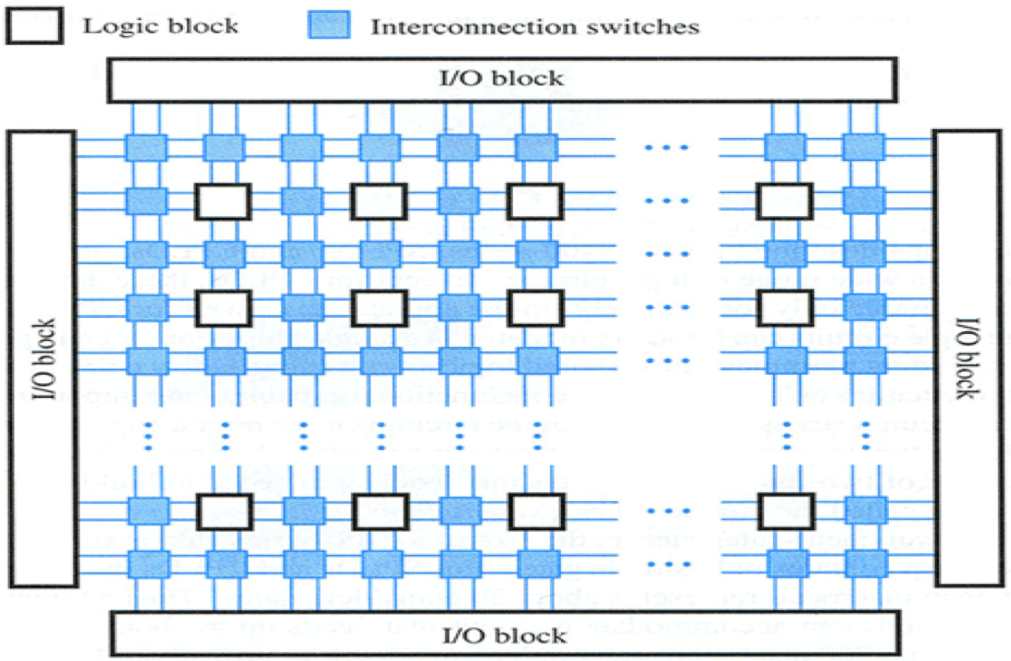
Hardwares for Embedded Processing

By using microcontrollers (or microprocessor based systems):

- Slower (technology dependent),
- only software is needed to be updated.



The Cortex M3's Thumbnail architecture looks like a conventional Arm processor. The differences are found
id the instruction decode that handles only Thumb and Thumb 2 instructions.



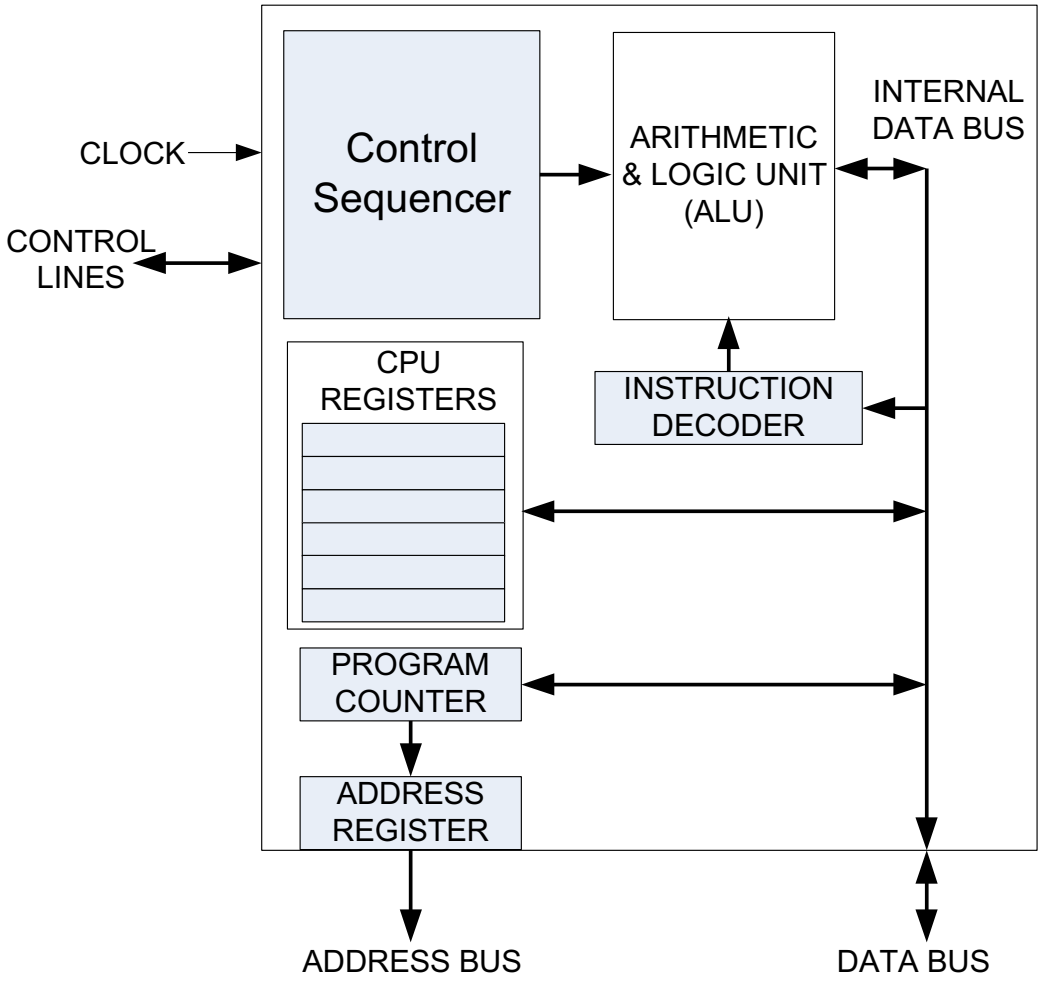
General structure of an FPGA

By using digital circuits (Field Programmable Gate Array - FPGA):

- Faster (only propagation delay),
- functions they perform can't be changed easily.

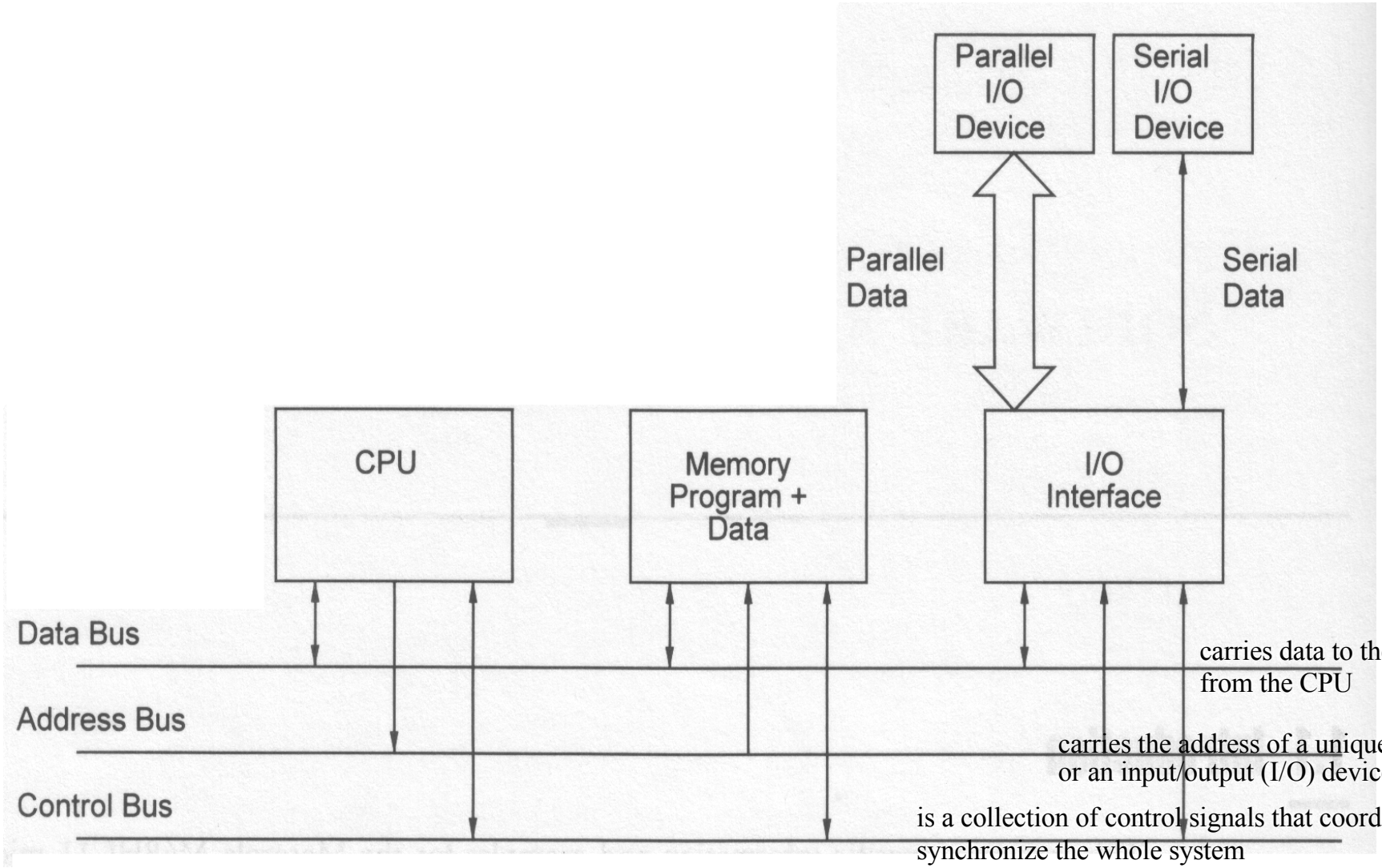
Microprocessor

- A single LSI (large scale integrated) chip, known as Central Processing Unit (CPU)
- Performs all computation
- Without memory and input/output (I/O) interface circuit
- ALU
- Controller
- Registers
- Internal bus



What's a "bus"?

A set of signal lines that carry digital information from source to destination



carries data to the CPU or from the CPU

carries the address of a unique memory location or an input/output (I/O) device

is a collection of control signals that coordinate and synchronize the whole system

Memory

- Memory in a computer system
  - stores the data and instructions of the programs.
  - is organized as a number of locations each of which can hold the same size data value (usually a byte).
- Each location has a unique address.
  - a 16-bit address allows for addressing 2^{16}= 65,536 (64K) memory locations.
  - A 32-bit address allows for addressing 2^{32}=4,294,967,296 (4G) memory locations.
- A transfer of data *from* memory is a read.
- A transfer of data *to* memory is a write.

Input/Output (I/O)

- Input Devices
  - Switches and Keypads
  - Provide binary information to the MPU
- Output devices
  - LEDs and LCDs
  - Receive binary information from the MPU

Registers

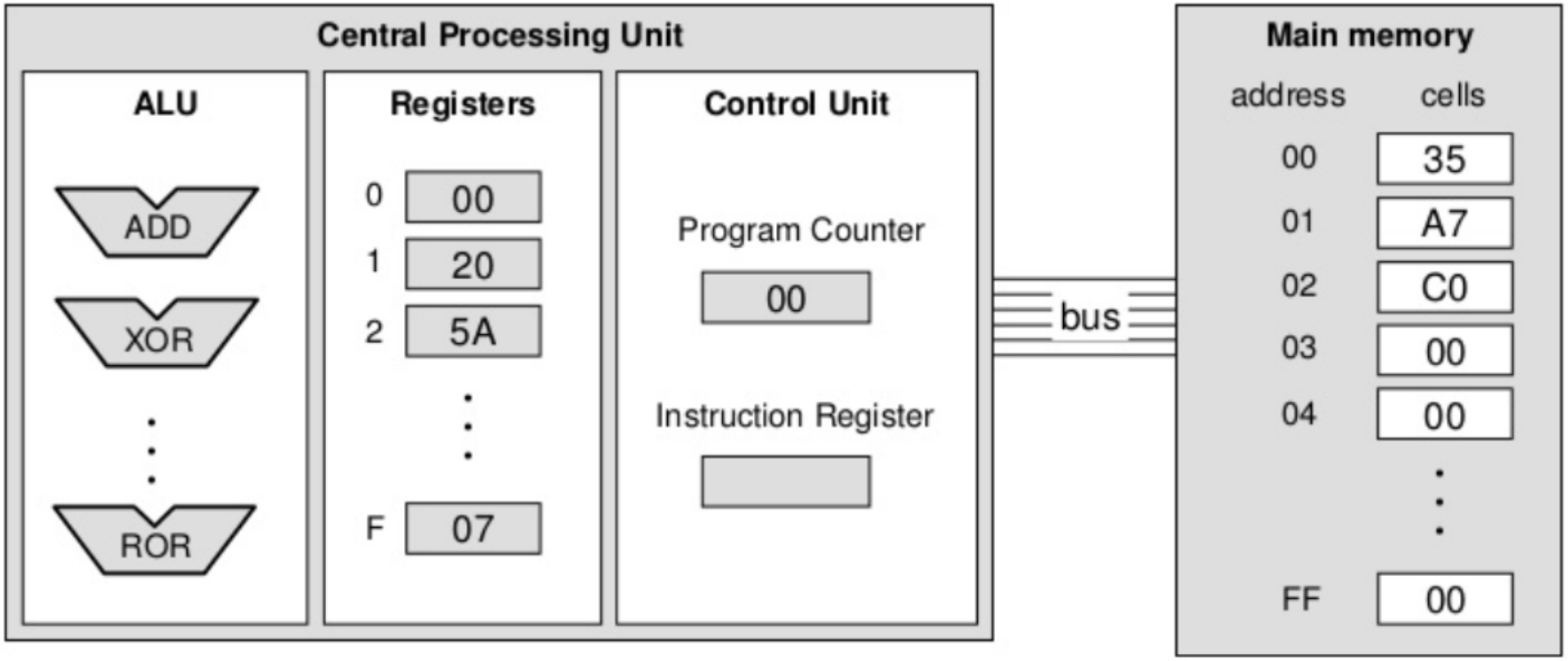
- The most fundamental storage area in the processor
  - is closely located to the processor
  - provides very fast access, operating at the same frequency as the processor clock
  - but is of limited quantity (typically less than 100)
- Most are of the general purpose type and can store any type of information:
  - data –e.g., timer value, constants
  - address –e.g., ASCII table, stack
- Some are reserved for specific purposes
  - program counter (r15 in ARM)
  - program status register (CPSR in ARM)

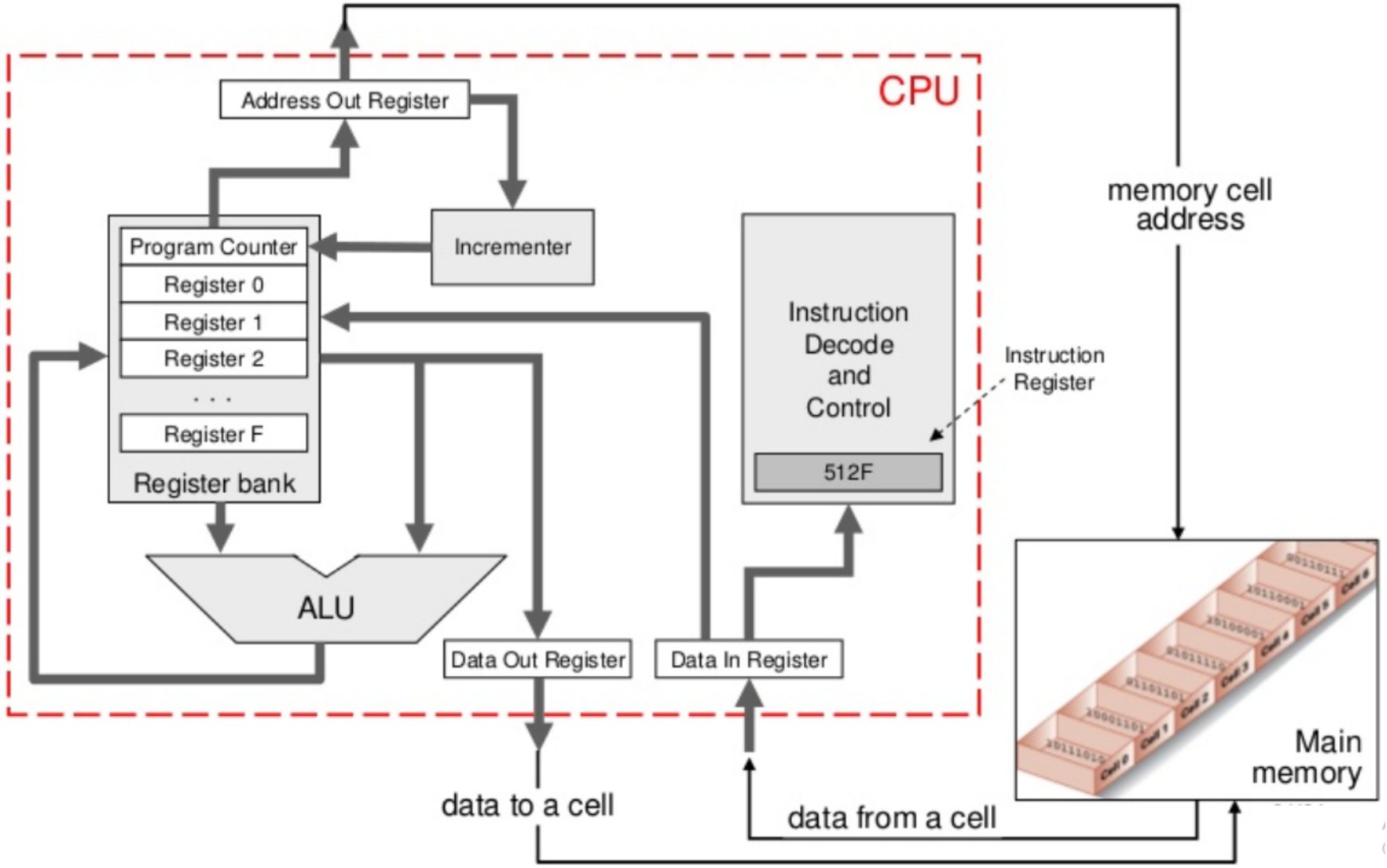
Microcode or Microprogram

- Each instruction is executed by the CPU through a sequence of well-defined steps.

- The recipe of execution per instruction is stored in the CPU's control sequencer, and is called the *Microcode* or *Microprogram. A microprogram* is written during the design phase of the chip and is normally not accessible afterward.

- The available hardware resources combined with the efficiency in microcode determines the number of clock cycles each instruction takes to execute.

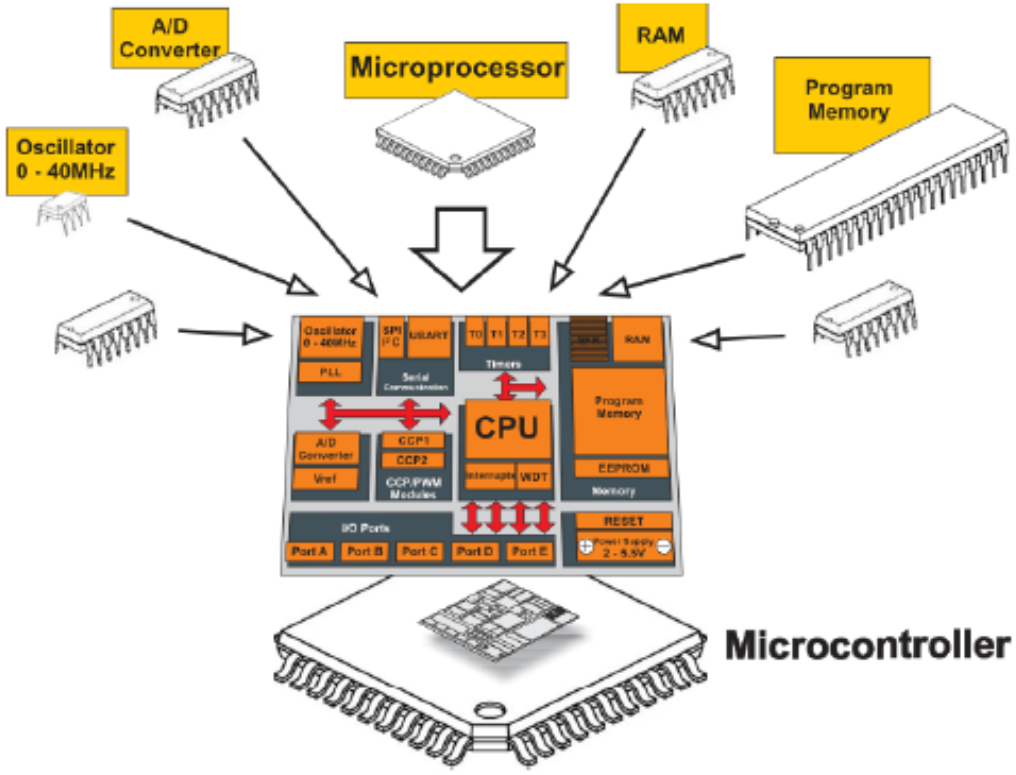

Central Processing Unit
ALU
ADD
XOR
ROR
Registers
0
00
1
20
2
5A
F
07
Control Unit
Program Counter
00
Instruction Register
bus
Main memory
address
cells
00
35
01
A7
02
C0
03
00
04
00
FF
00

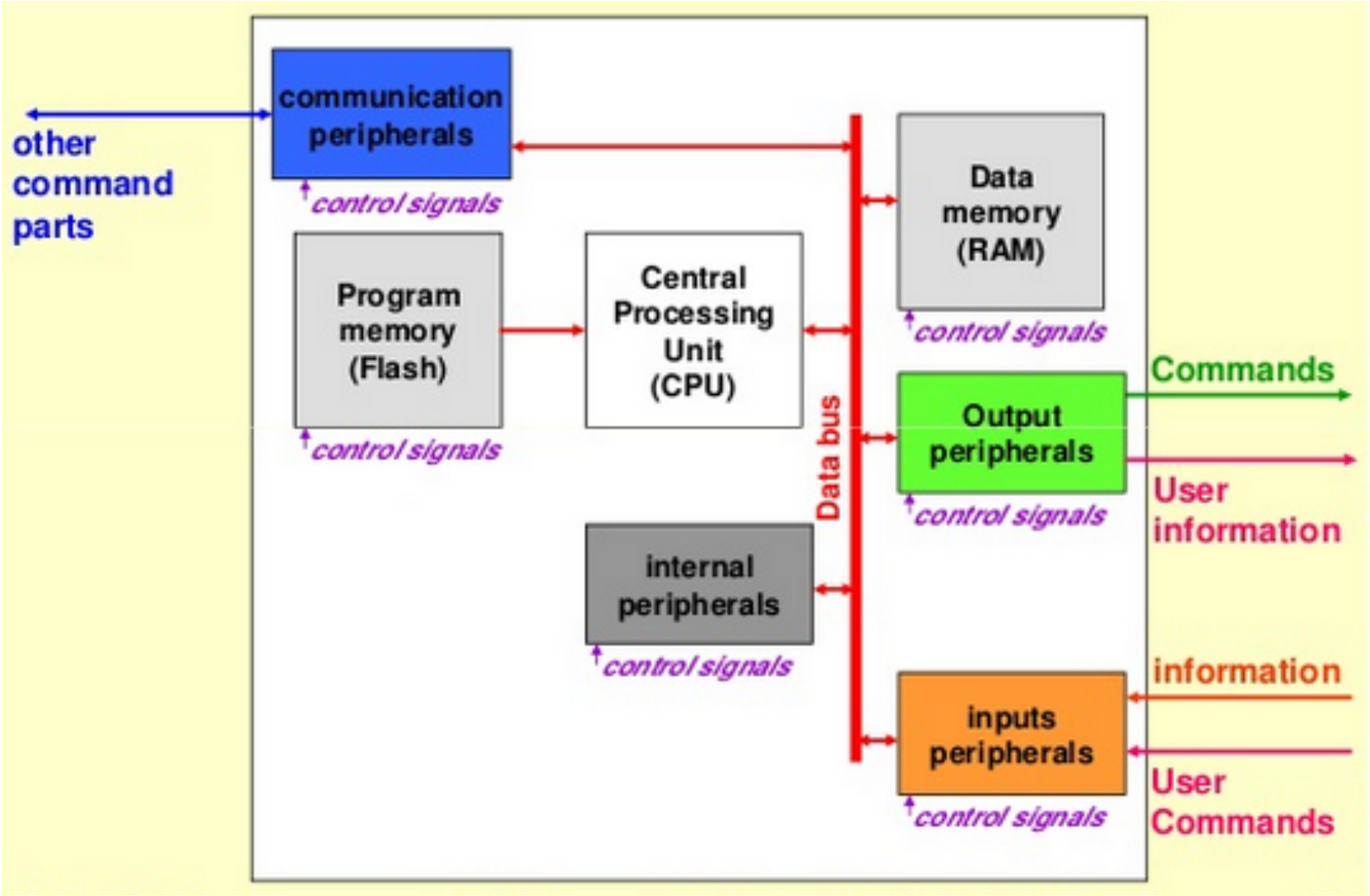

CPU
Address Out Register
memory cell address
Program Counter
Register 0
Register 1
Register 2
. . .
Register F
Register bank
Incrementer
Instruction Decode and Control
Instruction Register
512F
ALU
Data Out Register
Data In Register
data to a cell
data from a cell
Main memory

Microcontroller (MCU)

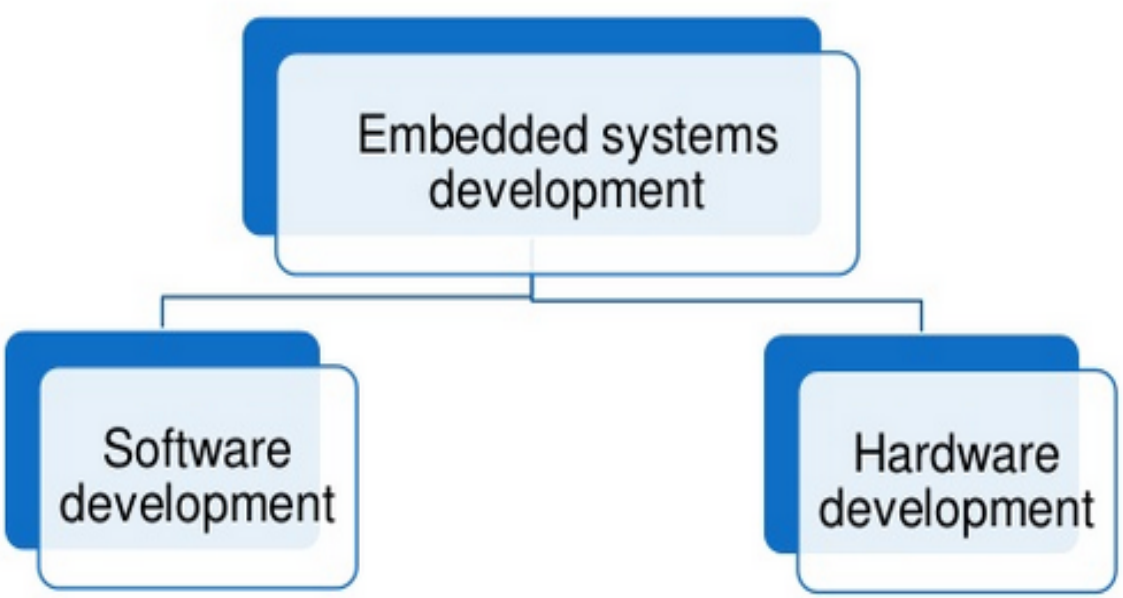
A microcontroller differs from a microprocessor in many ways

- A system where microprocessor along with memory, I/O, and other necessary functionalities are added for **control activities mainly –** all integrated onto a single chip
- Like a microprocessor, a microcontroller needs to be able to compute also, though not necessarily big numbers
- Now the microcontroller has taken over the role of the systems

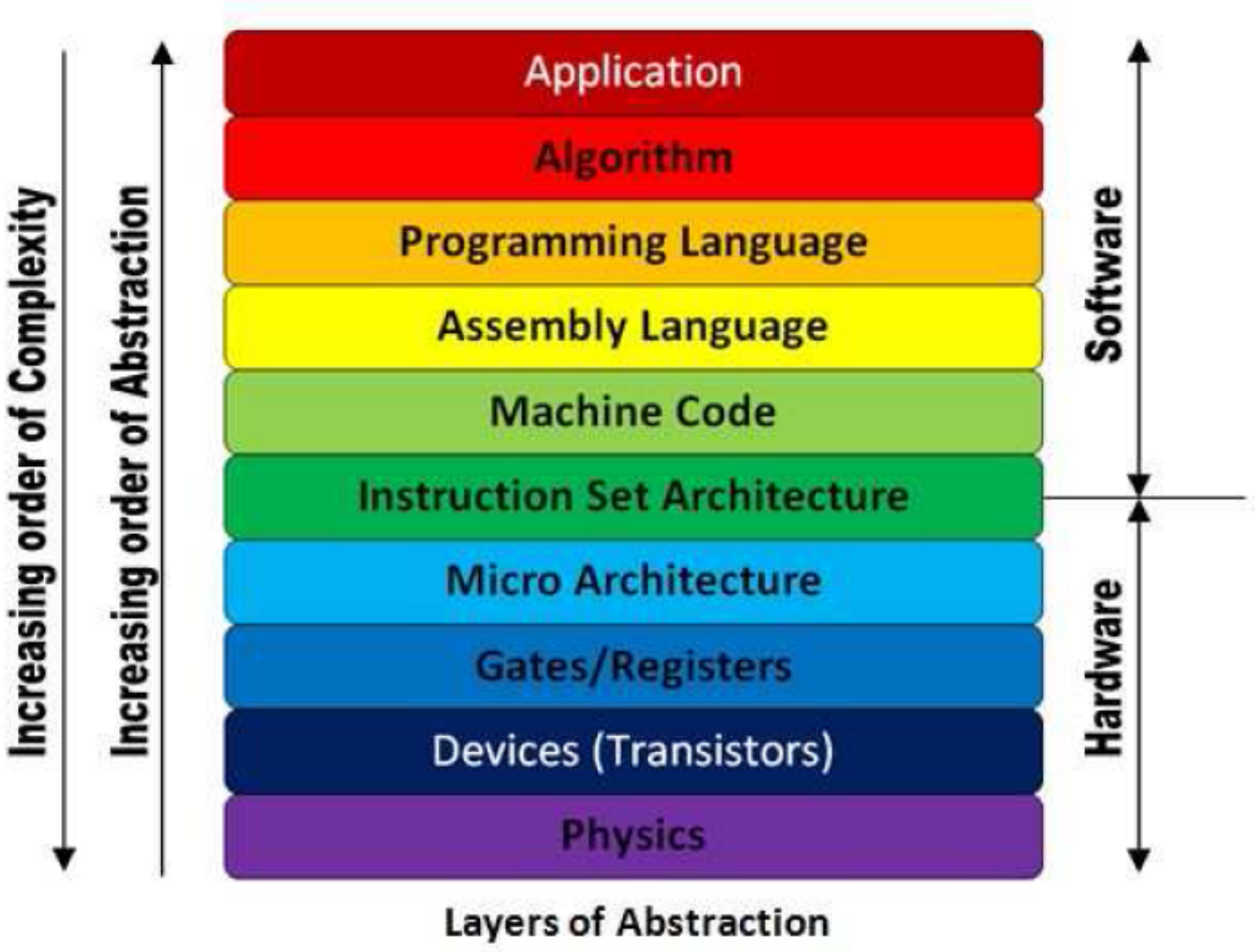



other command parts
communication peripherals
control signals
Program memory (Flash)
control signals
Central Processing Unit (CPU)
Data bus
internal peripherals
control signals
Data memory (RAM)
control signals
Output peripherals
control signals
Commands
User information
inputs peripherals
control signals
information
User Commands

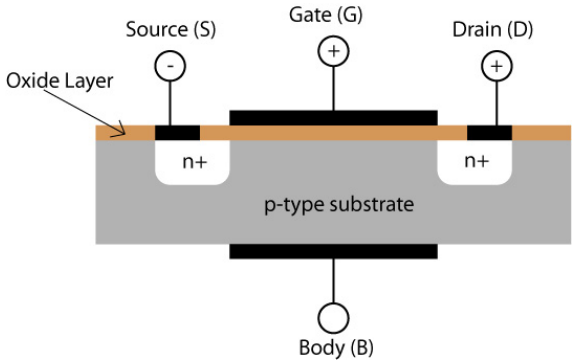
Developing an embedded application

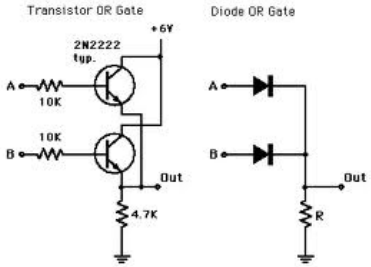


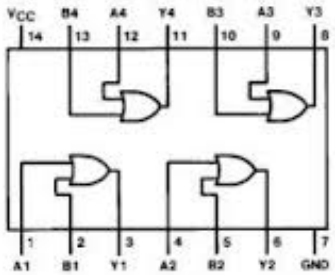
Hardware/Software

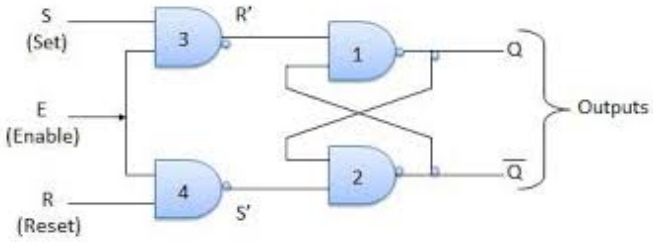


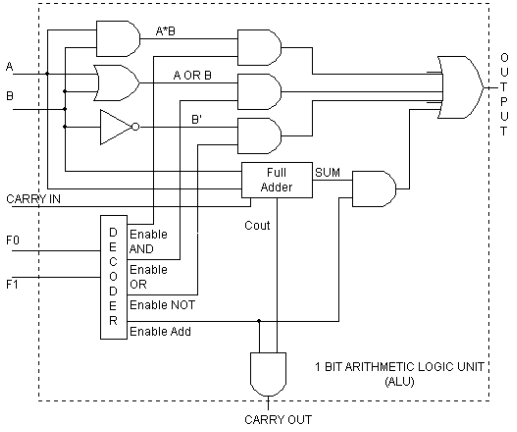
Hardware









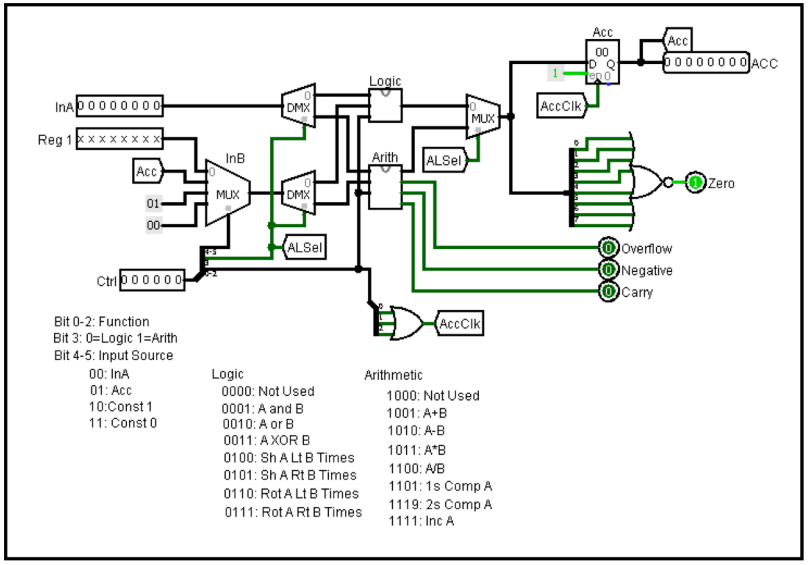


Enable AND = F0'*F1'

Enable OR = F0'* F1

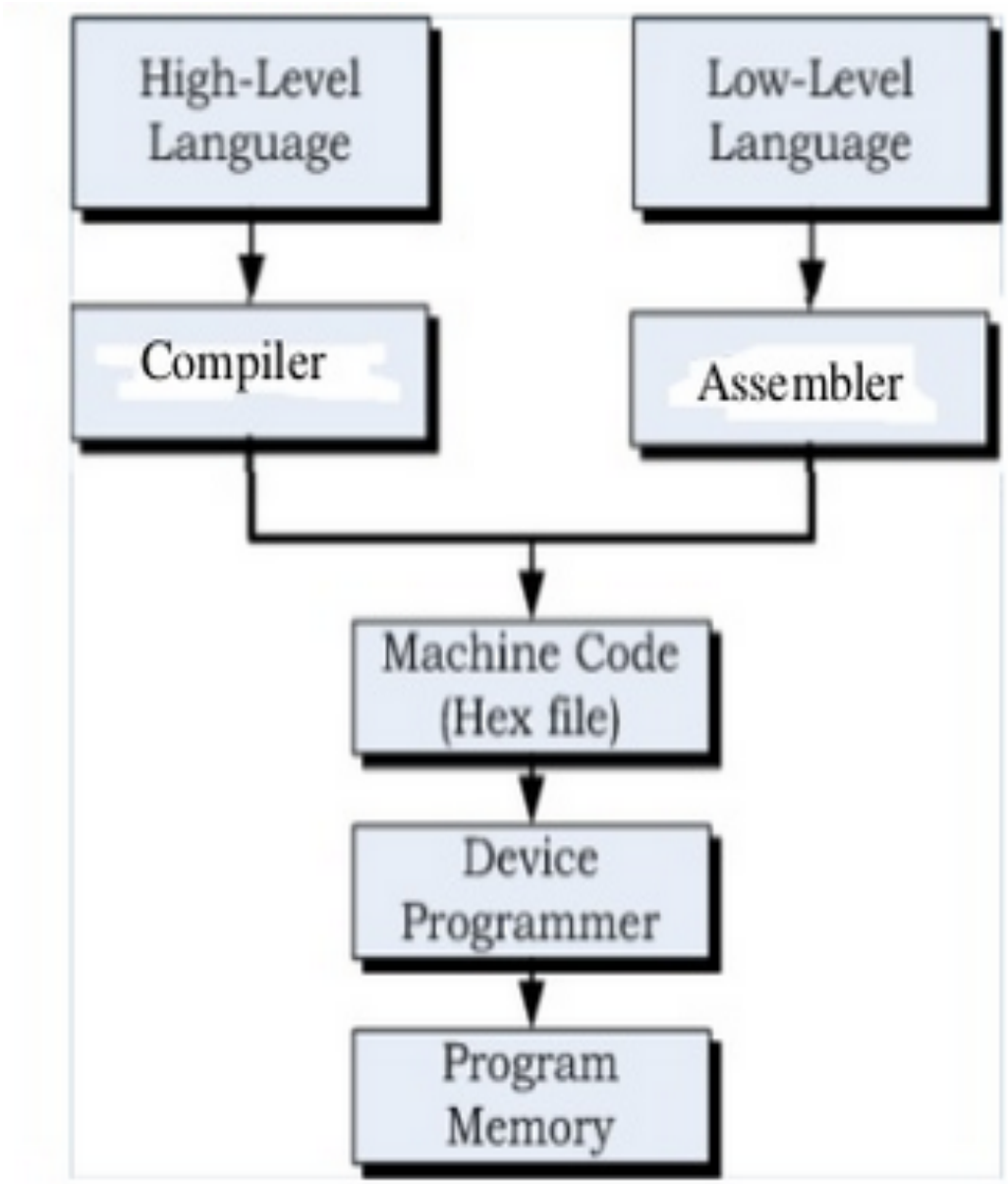
Enable NOT = F0 * F1'

Enable Add = F0 * F1



Software

- Machine Instruction
  - A sequence of binary digits that can be executed by the processor – Hard to understand, program, and debug for human being.
  - E.g. in the ARM Cortex M4, instruction 0010 0010 0110 0100 (in binary, 0x2264 in hexadecimal) puts the number 100 in register #2.
- Assembly Language
  - An assembly instruction is a mnemonic representation of a machine instruction.
  - E.g. ADD represents a register add, ADD R2 R1 R0 means R1 and R0 should be added and the result should be saved to R2.
  - Assembly programs must be translated to binary before it can be executed: Assembler does this translation: ADD R2 R1 R0 -> 0001100 010 001 000 = 0x1888
  - In this course we'll be mostly dealing with Assembly Language.
- High-level Language
  - Syntax of a high-level language is similar to English.
  - A translator is required to translate the program written in a high-level language – this is done by a compiler.
  - High-level languages allow the user to work on the program logic at a more conceptual level (e.g. you don't have to worry about what's in register R2).
  - In this course we will see some C language examples, which are helpful if you understand C, but is not necessary to this course.


High-Level Language
Low-Level Language
Compiler
Assembler
Machine Code (Hex file)
Device Programmer
Program Memory

Abstraction vs reality

Abstraction is good, but don't forget reality!

These abstractions have limits!

- Especially in the presence of bugs
  - Behavior of programs in presence of bugs
    - High-level language model breaks down

Understand assembly -> key to understand what really happens on the machine.

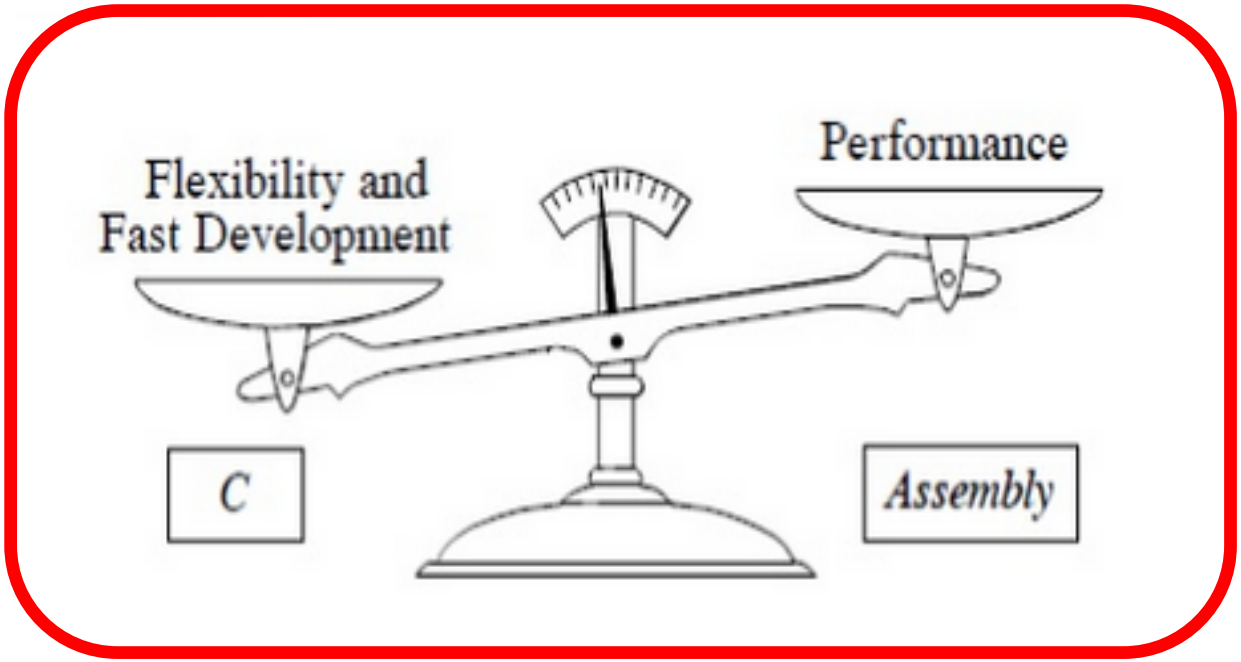
- Need to understand underlying implementations
- Need to have a working understanding of architecture
- Need to know architecture
  - Tuning program performance and understanding sources of program inefficiency.

Implementing system software

- Compiler has machine code as target
- Operating systems must manage process

Why do we learn Assembly?

- Assembly helps you understand how does the processor work
- Assembly program runs faster than high-level language. Performance critical codes must be written in assembly.
  - Use the profiling tools to find the performance bottle and rewrite that code section in assembly
  - Latency-sensitive applications, such as aircraft controller
  - Standard C compilers do not use some operations available on ARM processors, such ROR (Rotate Right) and RRX (Rotate Right Extended).
- Hardware/processor specific code,
  - Processor booting code
  - Device drivers
  - A test-and-set atomic assembly instruction can be used to implement locks and semaphores.
- Cost-sensitive applications
  - Embedded devices, where the size of code is limited
  - E.g. wash machine controller, automobile controllers
- The best applications are written by those who've mastered assembly language or fully understand the low-level implementation of the high-level language statements they're choosing.


Flexibility and
Fast Development
Performance
C
Assembly

Levels of Program Code

C Program

```
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10; i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

Compile →

Assembly Program

```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

Assemble →

Machine Program

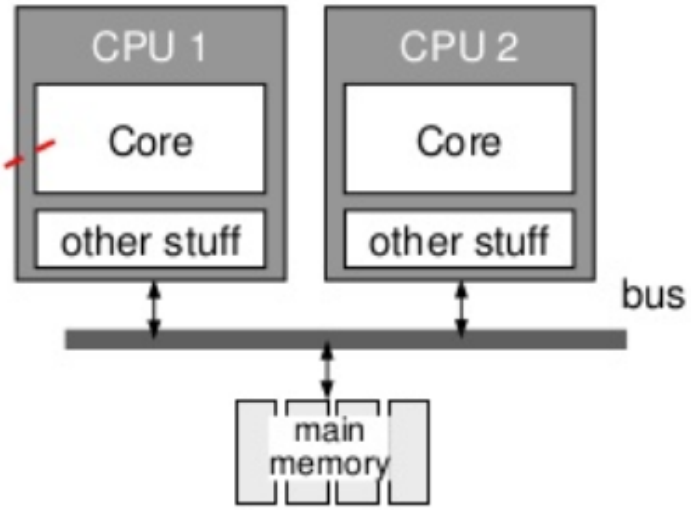
```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

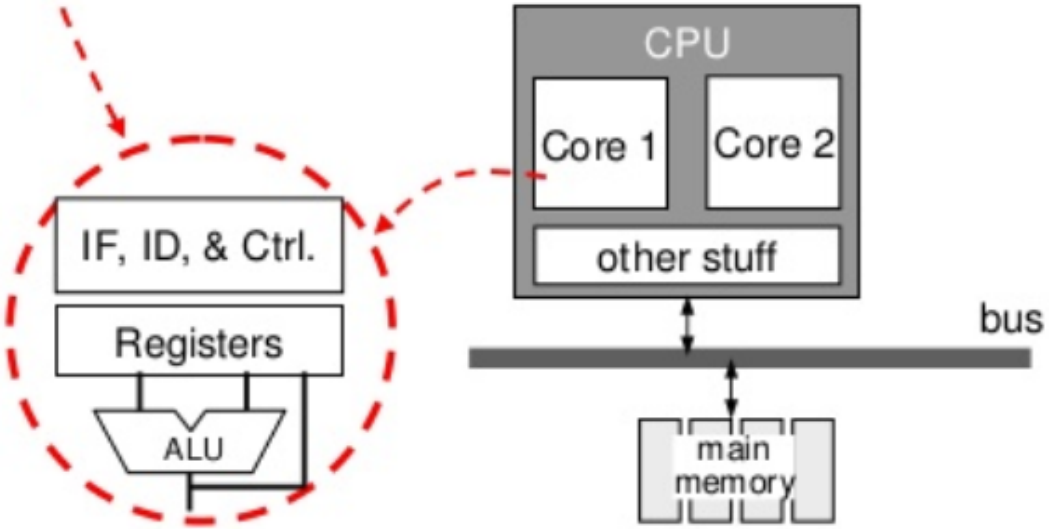
Improving CPU architecture

- Gradually, different memory types have been added on to the same chip
- At the same time, the CPU became more powerful and faster (from 8-bit to 16- and 32-bit devices)
- Parallel processing: Multiple operations simultaneously within CPU or across CPU

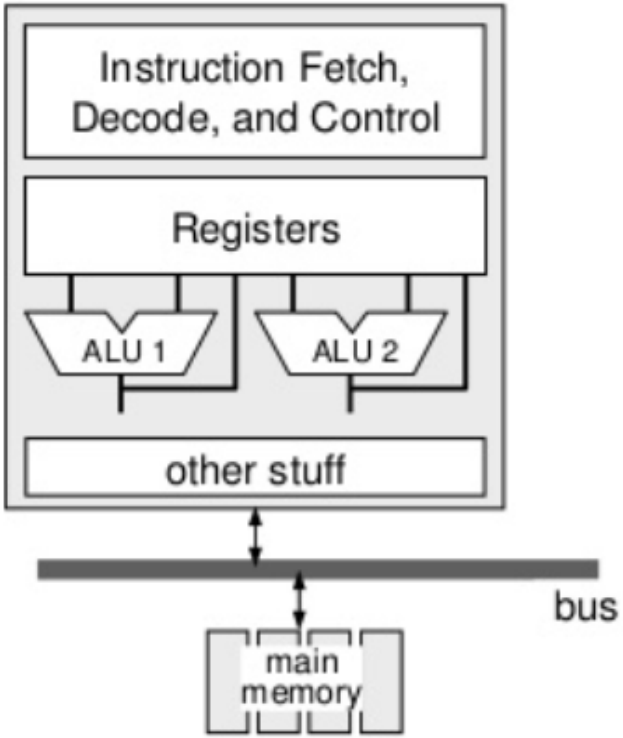
❑ Multi-CPU Architecture:



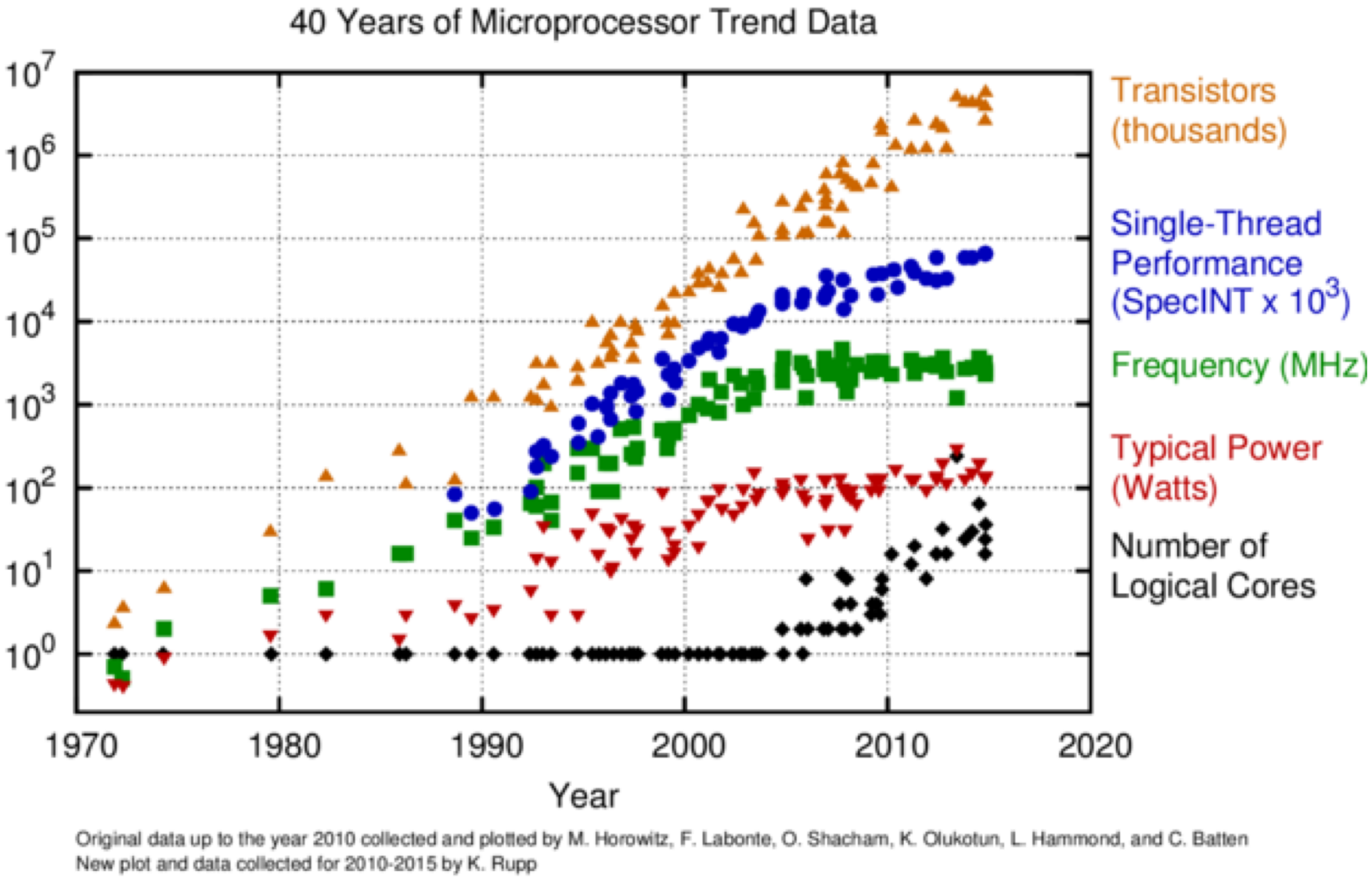
❑ Multi-Core Architecture:



❑ Multi-ALU Architecture:

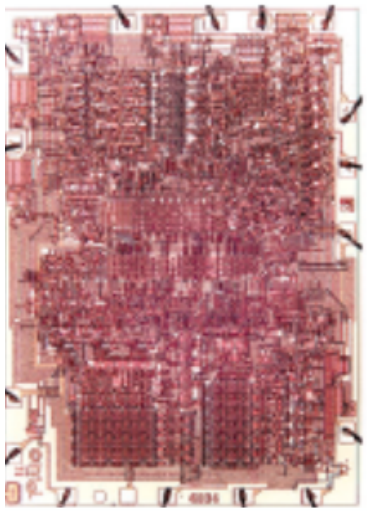


Note: "other stuff" could be interface logic, cache, MMU, timer, … etc.

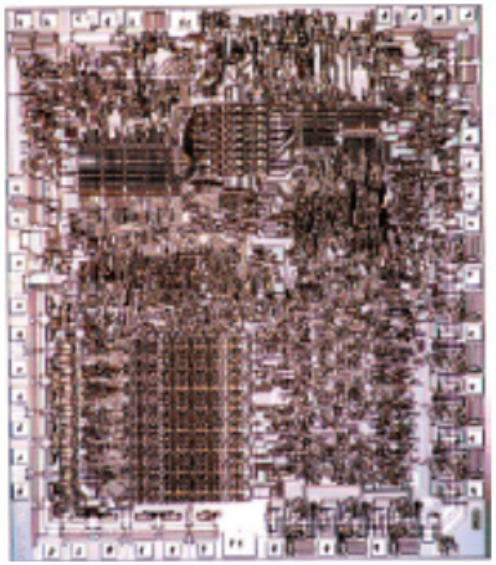


https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

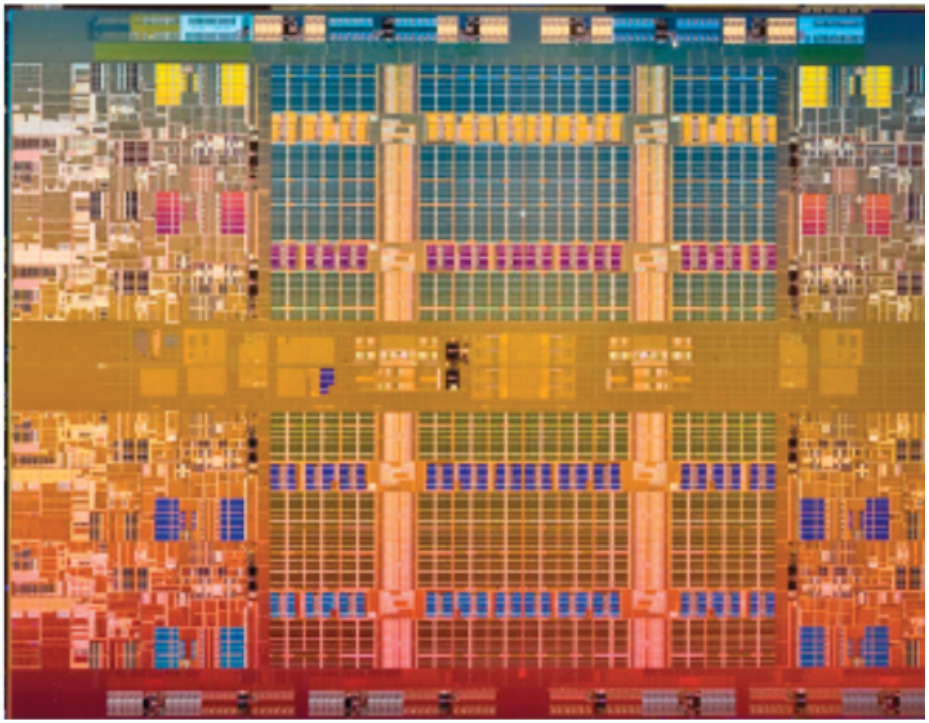
The first microprocessors appeared in Nov 1971 (Intel 4004)



Intel 4004, 1971
1 core, no cache
23K transistors

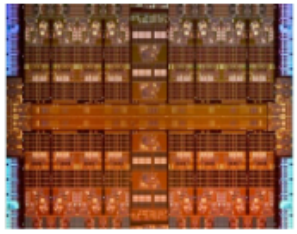


Intel 8008, 1978
1 core, no cache
29K transistors



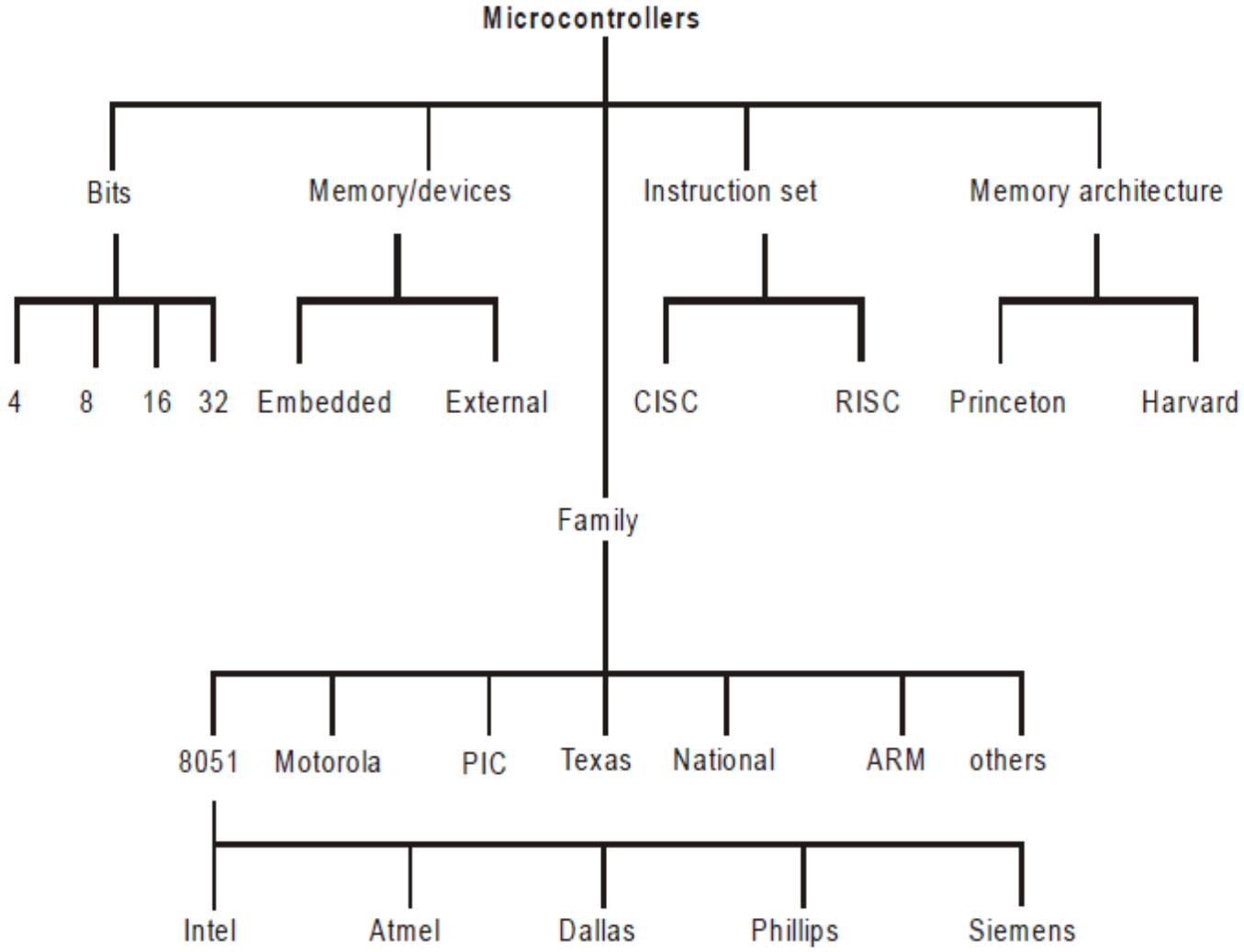
Intel Nehalem-EX, 2009
8 cores, 24MB cache
2.3B transistors

Figure credit: Shekhar Borkar, Andrew A. Chien, The Future of Microprocessors. Communications of the ACM, Vol. 54 No. 5, Pages 67-77 10.1145/1941487.1941507.



Oracle SPARC M7 (2015)
32 cores; > 10B transistors

Microcontollers



Most common microcontrollers

- 8-bit microcontrollers
  - Motorola
  - AVR
  - PIC
  - HCS12
  - 8051
- 32-bit microcontrollers
  - ARM
  - PIC32

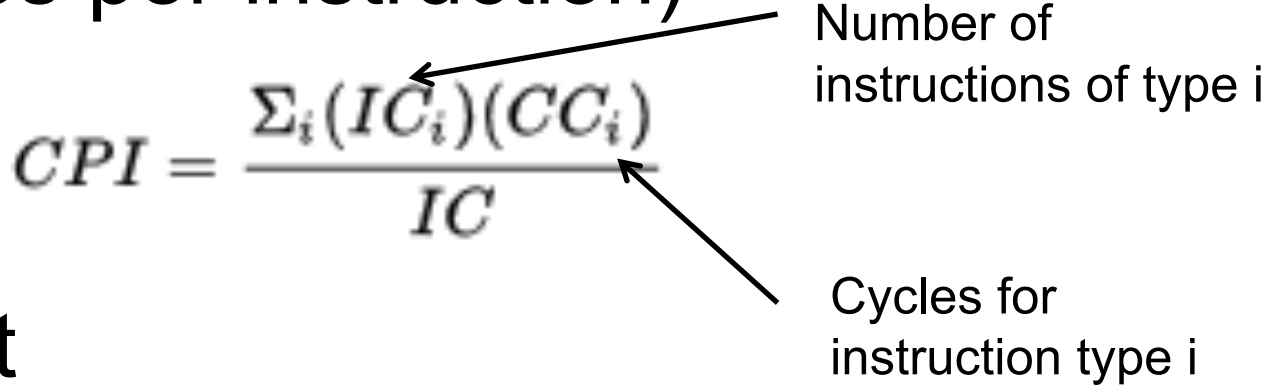
How to choose?

- Peripherals and interface features
  - Number of I/O ports
  - Serial communication modules
  - Peripherals like timer, ADC, PWM, etc
- Memory size requirements of the application
- Processing speed requirements
- Low power requirements
- Chip package
- Operation conditions (voltage, temperature, electromagnetic interference)
- Cost and availability
- Software development tool support and development kits
- Future upgradability
- Firmware packages and firmware* security
- Availability of application notes, design examples, and support

*__firmware__ is a specific class of computer software that provides the low-level control for the device's specific hardware

Processor efficiency

- MIPS (Million Instructions per Second)
  - Many reported IPS values have represented "peak" execution rates on artificial instruction sequences
  - Synthetic benchmarks such as Dhrystone are now generally used - DMIPS
- CPI (cycles per instruction)



- MIPS/watt
  - The energy required to run a particular benchmark

Why ARM?

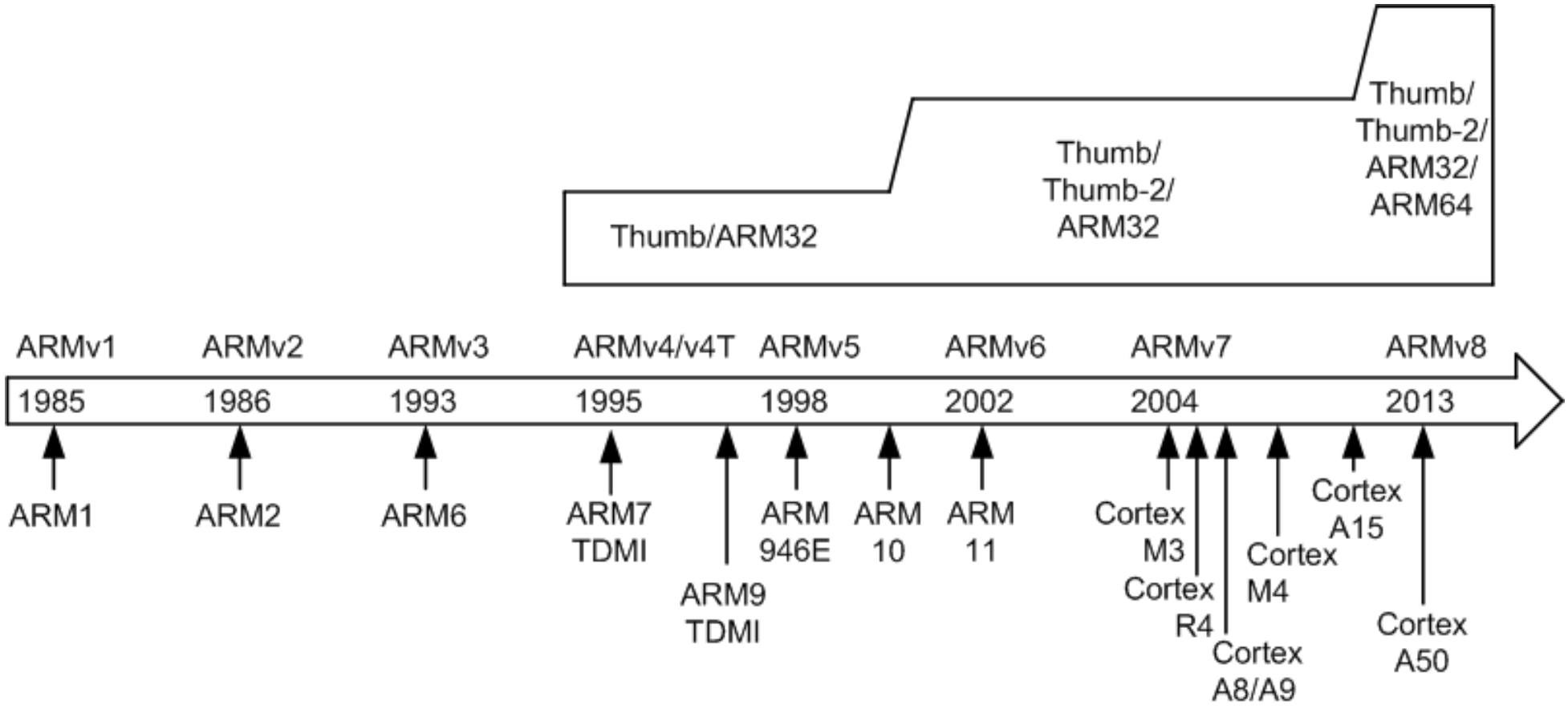
- ARM is one of the most licensed and thus widespread processor cores in the world
- Used especially in portable devices due to low power consumption and reasonable performance (MIPS / watt)

ARM History

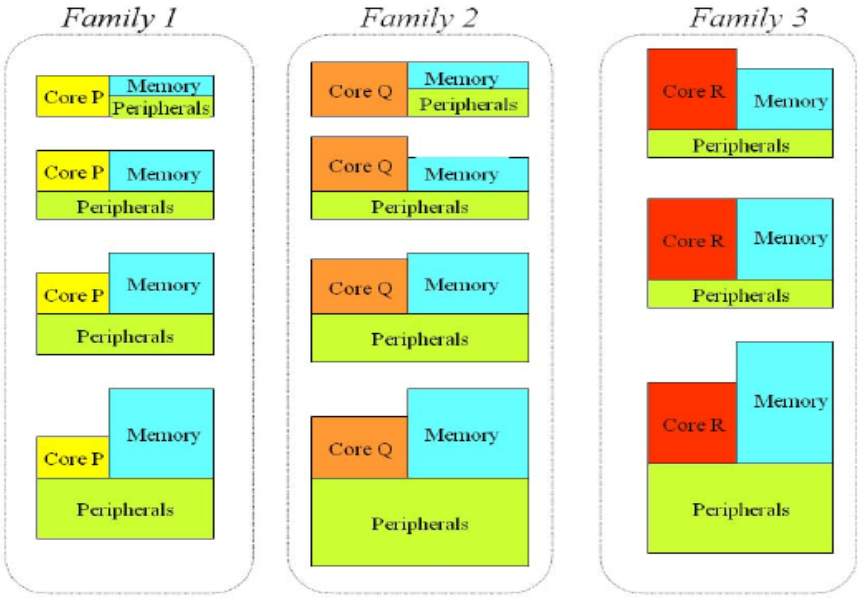
Year	Billion units	Relative size
2015	15	
2014	12	
2013	10	
2012	8.7	
2011	7.9	
2010	6.1	
2009	3.9	
2008	4.0	
2007	2.9	
2006	2.4	
2005	1.662	
2004	1.272	
2003	0.782	
2002	0.456	
2001	0.420	
2000	0.367	
1999	0.175	
1998	0.051	
1997	0.009	
Total	**78.094**	

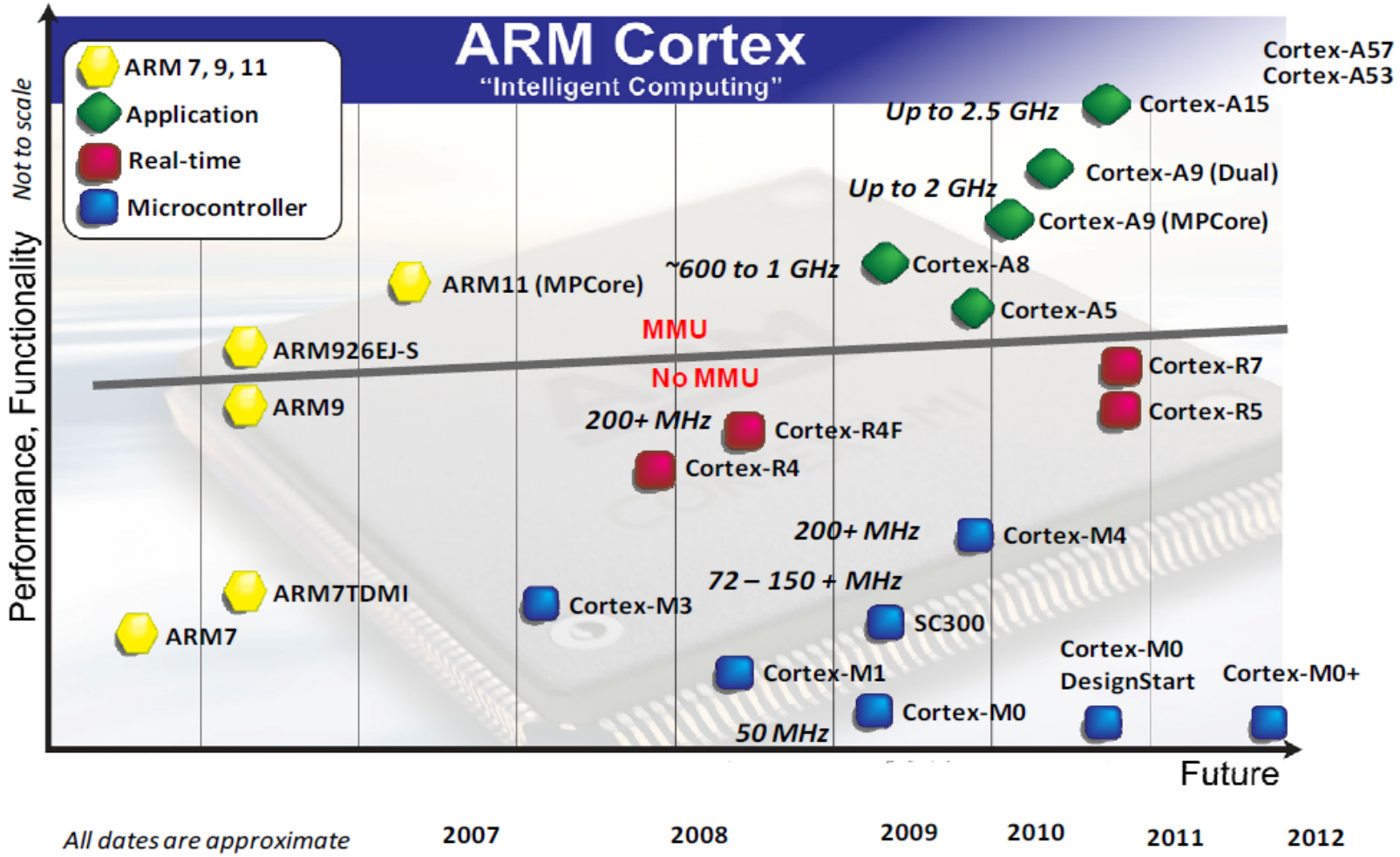
- Acorn Computers
  - British computer company based in Cambridge.
  - Set-up in 1978 by Chris Curry and Hermann Hauser.
  - Most famous computer was the BBC Micro (1980)
  - First RISC machine – 1985
  - First RICS based home computer – Archimedes - 1987
- ARM
  - Founded in November 1990 by Acorn Comp, Apple, VLSI Tech.
  - Licencing – 1992
  - ARM holdings float on to Nasdaq and London Stock Exch. – 1998
  - 1 billion ARM based chips have been shipped – 2002
  - Energy efficient Cortex chips have been announced – 2004
  - 10 billion chips shipped (**95%** of smartphones, **35%** of digital televisions and set-top boxes and **10%** of mobile computers) – 2009
  - 64-bit capable ARMv8 architecture is introduced – 2011
  - ARM named as fifth most innovative company in the world – 2014
  - ARM reported profit of $448.4 million – 2015
  - SoftBank acquired ARM for $31 billion - 2016

History of ARM processors



- ARM has several designs which can be grouped in different architectures, families and cores.
  - The architectures are the specifications, i.e. the set of registers, instructions and operation modes that should be supported by implementations of the architecture.
  - The core is a specific implementation of an architecture, i.e. the actual blue-print of the transistors and other discrete parts needed to create a ARM CPU.
  - A family is a specific detailed implementation of an architecture, i.e. the actual hardware details needed to create an ARM core.




ARM Cortex
"Intelligent Computing"
ARM 7, 9, 11
Application
Real-time
Microcontroller
Performance, Functionality
Not to scale
Cortex-A57
Cortex-A53
Up to 2.5 GHz
Cortex-A15
Cortex-A9 (Dual)
Up to 2 GHz
Cortex-A9 (MPCore)
~600 to 1 GHz
Cortex-A8
ARM11 (MPCore)
Cortex-A5
MMU
ARM926EJ-S
No MMU
Cortex-R7
ARM9
Cortex-R5
200+ MHz
Cortex-R4F
Cortex-R4
200+ MHz
Cortex-M4
72 – 150 + MHz
ARM7TDMI
Cortex-M3
SC300
ARM7
Cortex-M0
DesignStart
Cortex-M0+
Cortex-M1
Cortex-M0
50 MHz
Future
All dates are approximate
2007
2008
2009
2010
2011
2012

Family	Architecture	Cores
ARM7TDMI	ARMv4T	ARM7TDMI(S)
ARM9 ARM9E	ARMv5TE(J)	ARM926EJ-S, ARM966E-S
ARM11	ARMv6 (T2)	ARM1136(F), 1156T2(F)-S, 1176JZ(F), ARM11 MPCore™
Cortex-A	ARMv7-A	Cortex-A5, A7, A8, A9, A15
Cortex-R	ARMv7-R	Cortex-R4(F)
Cortex-M	ARMv7-M	Cortex-M3, M4
	ARMv6-M	Cortex-M1, M0
NEW !	ARMv8-A	64 Bit

Features	ARM7TDMI-S	Cortex-M3
Architecture	ARMv4T (von Neumann)	ARMv7-M (Harvard)
ISA Support	Thumb / ARM	Thumb / Thumb-2
Pipeline	3-Stage	3-Stage + branch speculation
Interrupts	FIQ / IRQ	NMI + 1 to 240 Physical Interrupts
Interrupt Latency	24-42 Cycles	12 Cycles
Sleep Modes	None	Integrated
Memory Protection	None	8 region Memory Protection Unit
Dhrystone	0.95 DMIPS/MHz (ARM mode)	1.25 DMIPS/MHz
Power Consumption	0.28mW/MHz	0.19mW/MHz
Area	0.62mm2 (Core Only)	0.86mm2 (Core & Peripherals)*

Where to use?

ARM 7 – Introduced in 1994, used for simple 32-bit devices.

ARM9 – Was most popular ARM family (over 5 billion sold), used for smarthphones, HDD controllers, set top box, etc.

ARM11 - Extreme low power, many of today's smartphones.

Cortex M-series: Cost-sensitive solutions for microcontroller applications, system clock<200MHz

- Cortex M0 – Ultra low-power, oltra low gate count,
- Cortex M1 – First ARM processor designed specifically for implementation in FPGAs.
- Cortex M3 – High performance and energy efficiency. Microcontroller applications.
- Cortex M4 – Embedded processor for DSP
- Cortex M7 – 2x Performance of M4

Cortex R-series: Exceptional performance for real-time applications, system clock<600MHz

- Cortex R4 - First embedded real-time processor based on the ARMv7-R architecture for high-volume deeply-embedded System-on-Chip applications such as hard disk, wireless baseband processors, electronic control units for automotive systems.
- Cortex R5 – Extends the feature set of Cortex-R4, incereased efficiency and reliability.
- Cortex R7 – High-performance dual core

Cortex A-series: High performance processors for open Operating sys, system clock>1GHz

- Cortex A5 – Power and cost sensitive applications, smartphones.
- Cortex A8 – Suitable for high-end phones, printers, DTVs
- Cortex A9 – 1 to 4 cores. High performance-low power devices
- Cortex A15 – Ultra low-power. Suitable for mobile computing, wireless infrastructure

ARM Architecture: For Diverse Embedded Processing Needs

Cortex - A

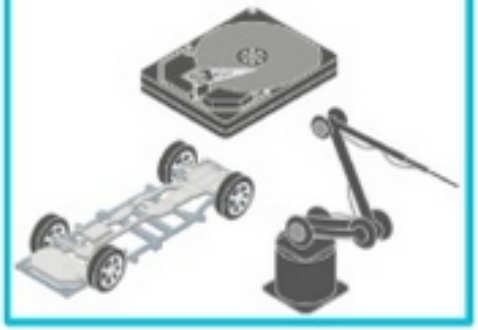
Highest performance

Optimised for
rich operating systems

Cortex - R

Fast response

Optimised for
high performance,
hard real-time applications

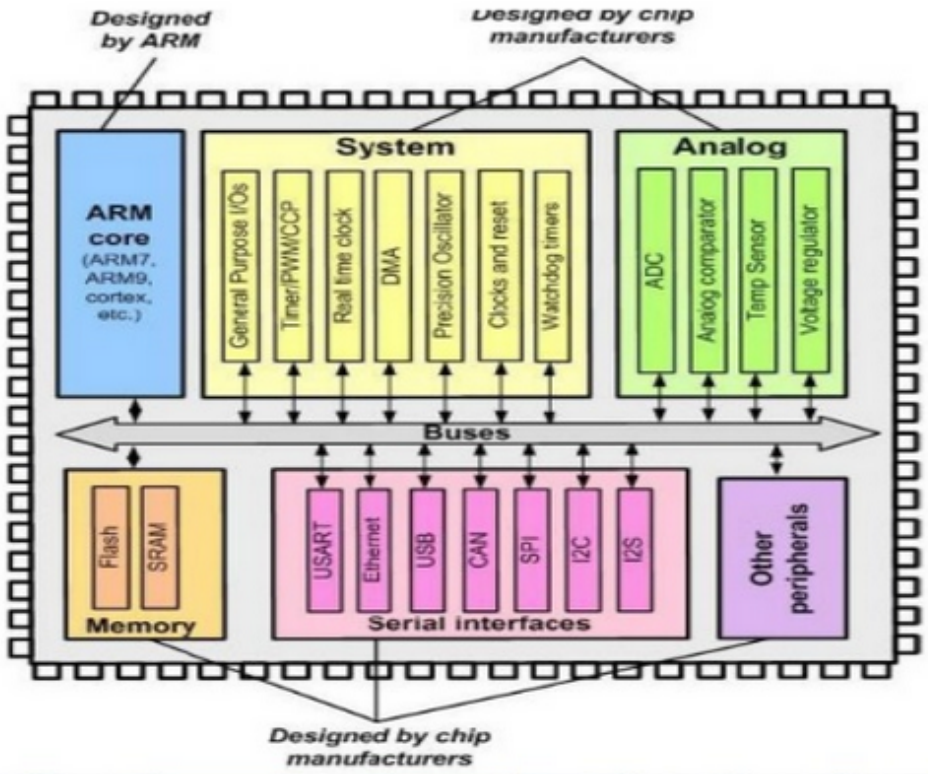


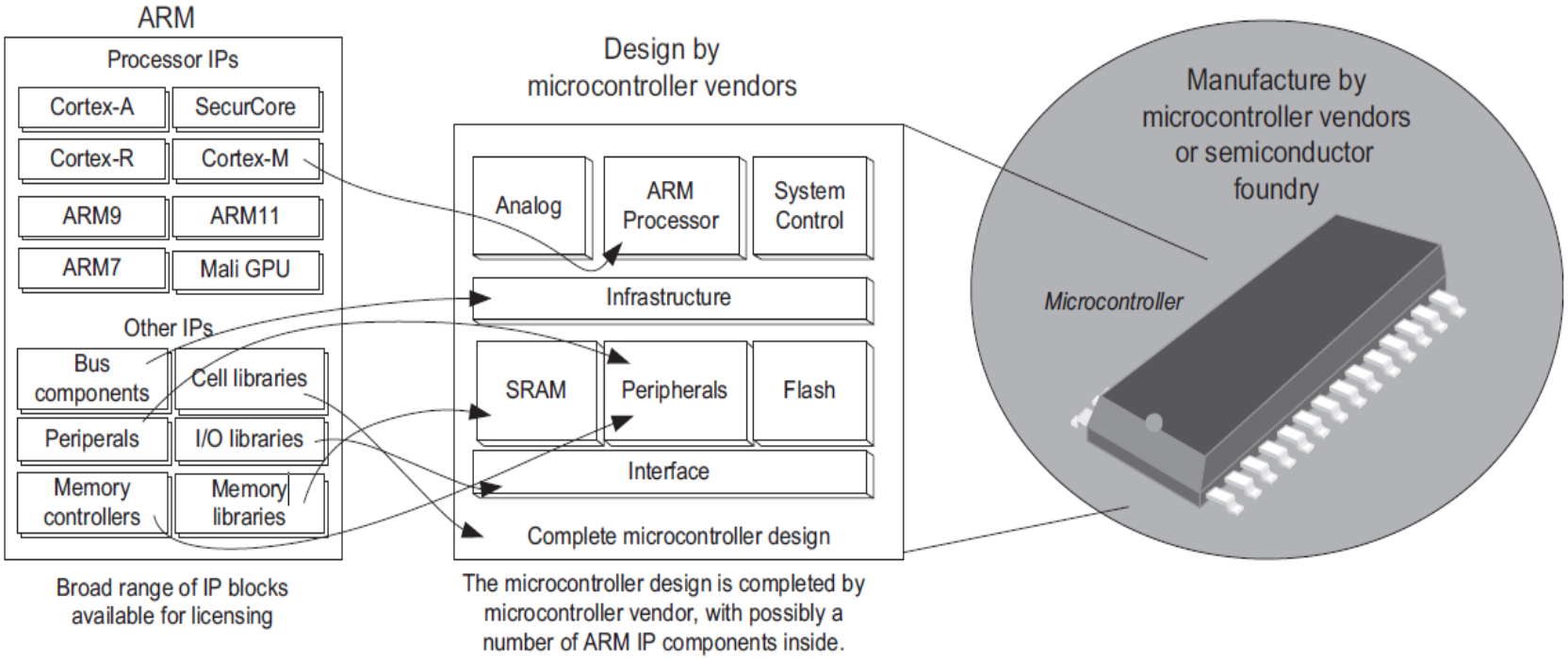
Cortex - M

Smallest/lowest power

Optimised for
discrete processing and
microcontrollers

Microcontrollers with ARM core





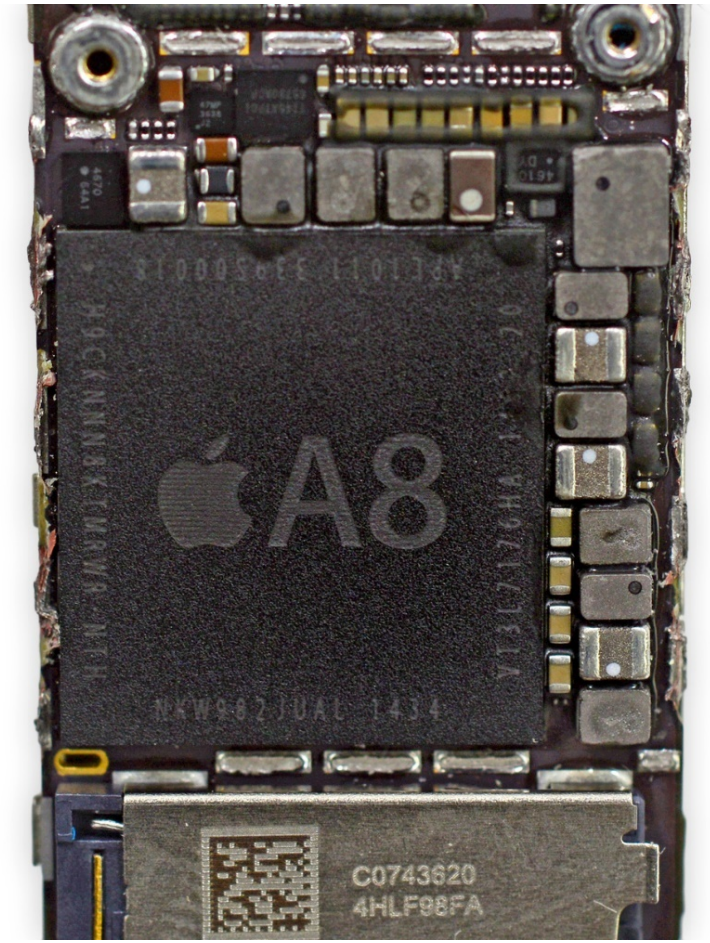
Sample ARM powered products

iPhone 5

http://www.ifixit.com

The A6 processor is the first Apple System-on-Chip (SoC) to use a custom design, based off the **ARMv7** instruction set.

iPhone 6



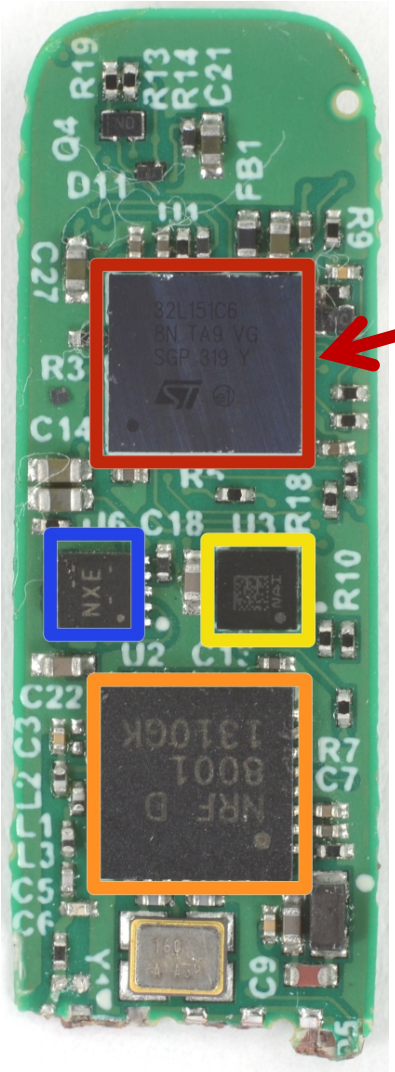
The A8 processor is the first 64-bit ARM based SoC. It supports **ARM A64, A32, and T32** instruction set.

http://www.ifixit.com

Apple Watch

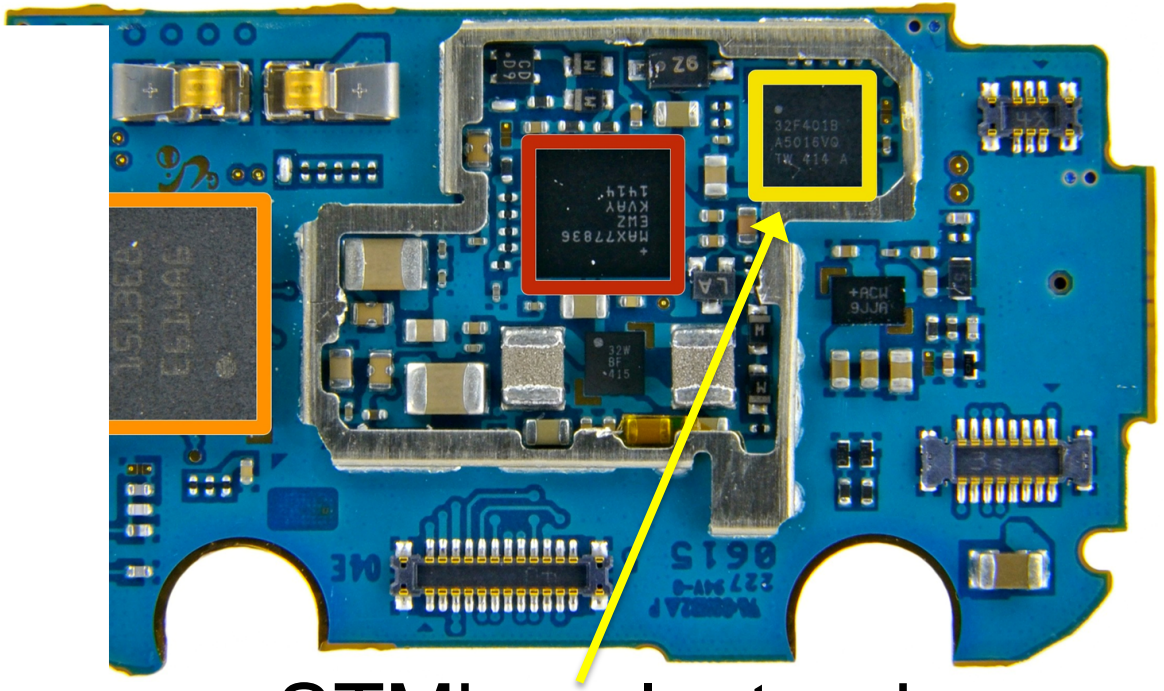
- Apple S1 Processor
  - **32-bit ARMv7-A** compatible
  - # of Cores: **1**
  - CMOS Technology: 28 nm
  - L1 cache 32 KB data
  - L2 cache 256 KB
  - GPU PowerVR SGX543

Fitbit Flex Teardown



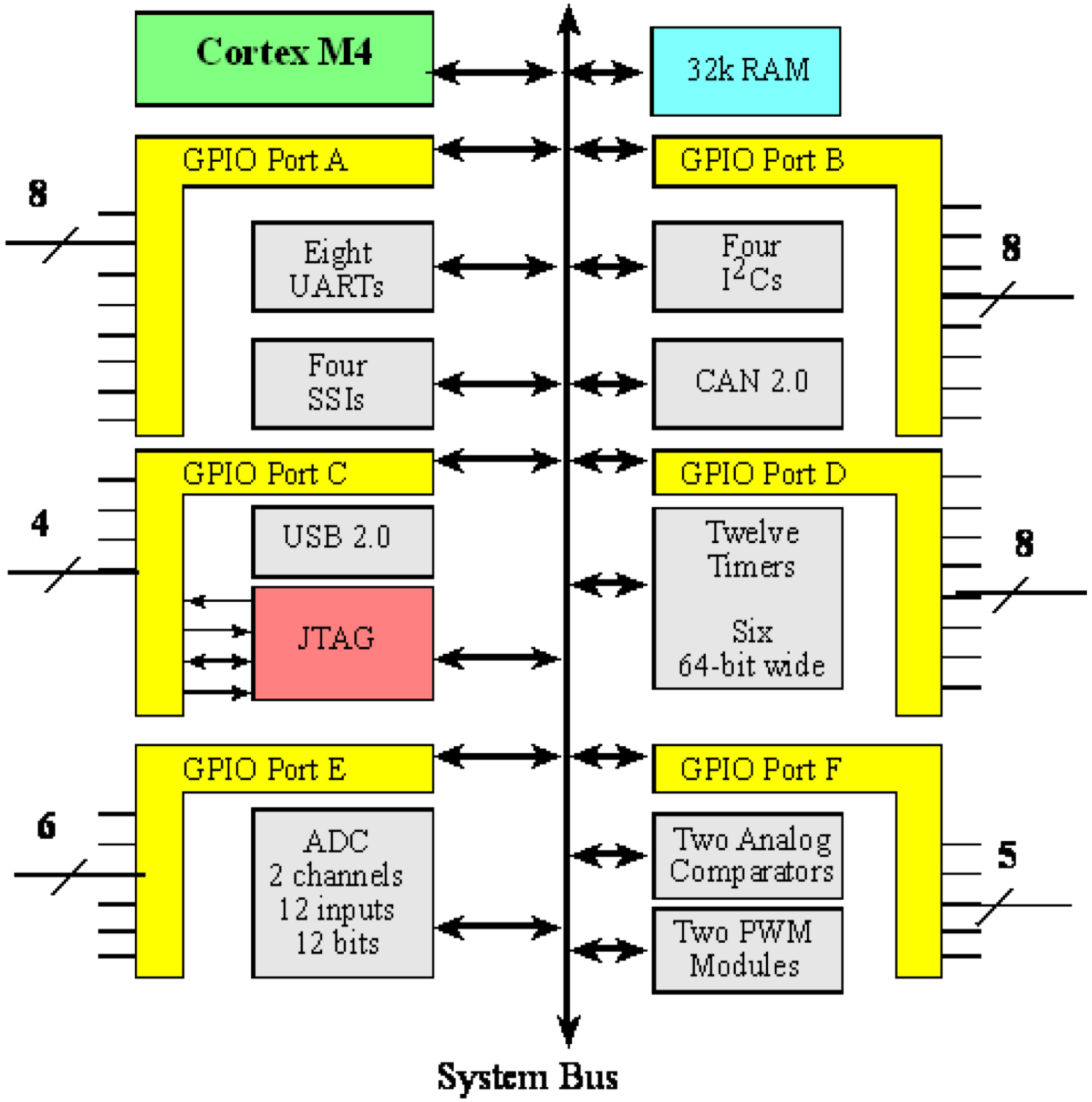
STMicroelectronics 32L151C6 Ultra Low Power ARM Cortex M3 Microcontroller

Samsung Galaxy Gear

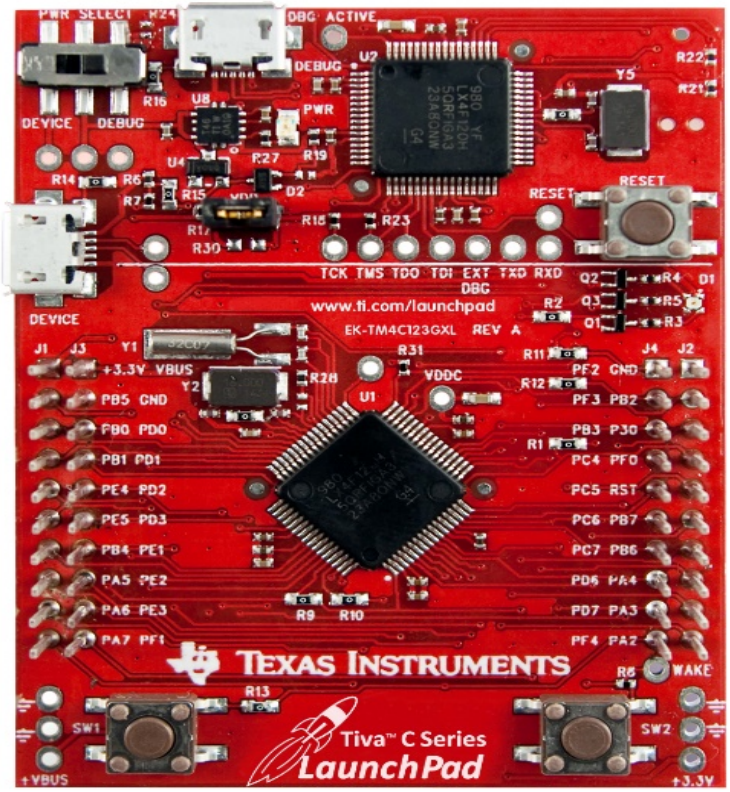


- STMicroelectronics STM32F401B **ARM-Cortex M4** MCU with 128KB Flash

We will use: Texas instruments Tiva C Series TM4C123G board including ARM Cortex 4



80MHz internal clock
16MHz for usual operations



Feature	Description
Performance	
Core	ARM Cortex-M4F processor core
Performance	80-MHz operation; 100 DMIPS performance
Flash	256 KB single-cycle Flash memory
System SRAM	32 KB single-cycle SRAM
EEPROM	2KB of EEPROM
Internal ROM	Internal ROM loaded with TivaWare™ for C Series software
Security	
Communication Interfaces	
Universal Asynchronous Receivers/Transmitter (UART)	Eight UARTs
Synchronous Serial Interface (SSI)	Four SSI modules
Inter-Integrated Circuit (I^2C)	Four I^2C modules with four transmission speeds including high-speed mode
Controller Area Network (CAN)	Two CAN 2.0 A/B controllers
Universal Serial Bus (USB)	USB 2.0 OTG/Host/Device
System Integration	
Micro Direct Memory Access (µDMA)	ARM® PrimeCell® 32-channel configurable µDMA controller
General-Purpose Timer (GPTM)	Six 16/32-bit GPTM blocks and six 32/64-bit Wide GPTM blocks
Watchdog Timer (WDT)	Two watchdog timers
Hibernation Module (HIB)	Low-power battery-backed Hibernation module
General-Purpose Input/Output (GPIO)	Six physical GPIO blocks
Advanced Motion Control	
Pulse Width Modulator (PWM)	Two PWM modules, each with four PWM generator blocks and a control block, for a total of 16 PWM outputs.
Quadrature Encoder Interface (QEI)	Two QEI modules
Analog Support	
Analog-to-Digital Converter (ADC)	Two 12-bit ADC modules, each with a maximum sample rate of one million samples/second
Analog Comparator Controller	Two independent integrated analog comparators
Digital Comparator	16 digital comparators
JTAG and Serial Wire Debug (SWD)	One JTAG module with integrated ARM SWD
Package Information	
Package	64-pin LQFP
Operating Range (Ambient)	Industrial (-40°C to 85°C) temperature range Extended (-40°C to 105°C) temperature range

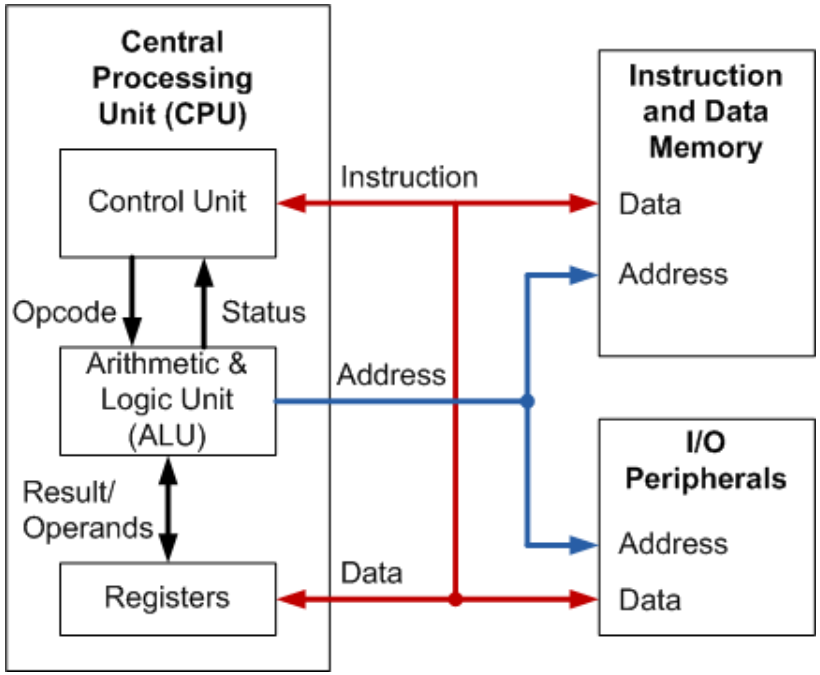
ARM Cortex-M4

- Introduced in 2010
- Designed with a large variety of highly efficient signal processing features
- Features extended single-cycle multiply accumulate instructions, optimized SIMD (single instruction multiple data) arithmetic, saturating arithmetic and an optional Floating Point Unit.
- High Performance Efficiency
  - 1.25 DMIPS/MHz (DhrystoneMillion Instructions Per Second / MHz) at the order of μWatts / MHz
- Low Power Consumption
  - Longer battery life –especially critical in mobile products
- Enhanced Determinism
  - The critical tasks and interrupt routines can be served quickly in a known number of cycles

Computer Architectures

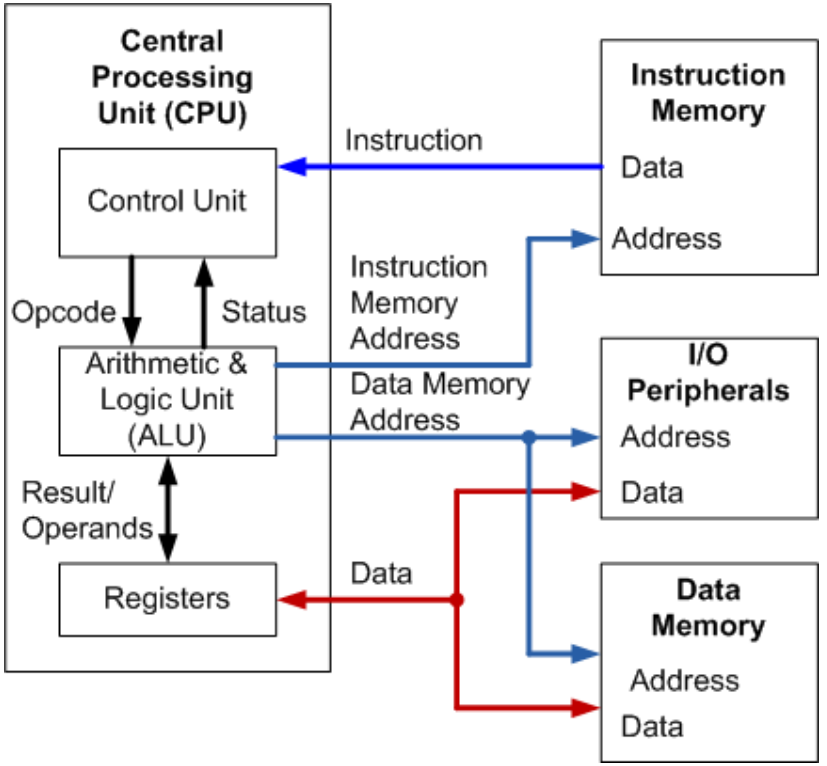
Von-Neumann/Princeton

Instructions and data are stored in the same memory.

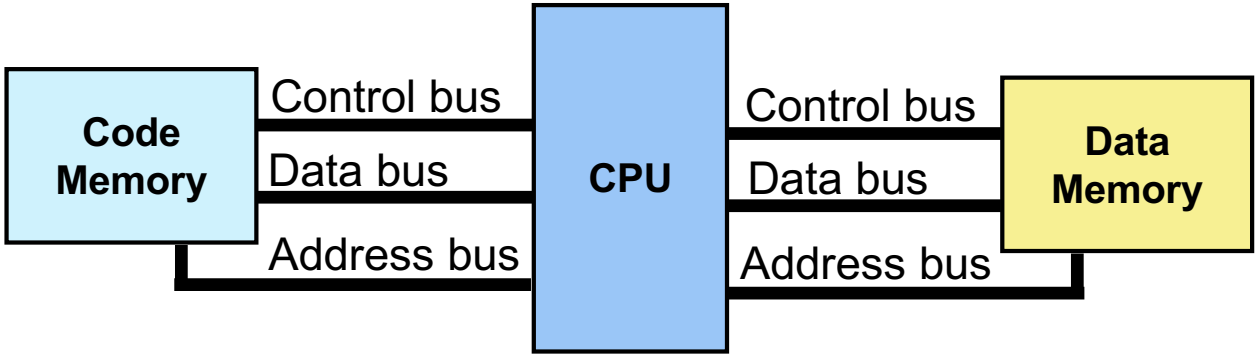


Harvard

Data and instructions are stored into separate memories.

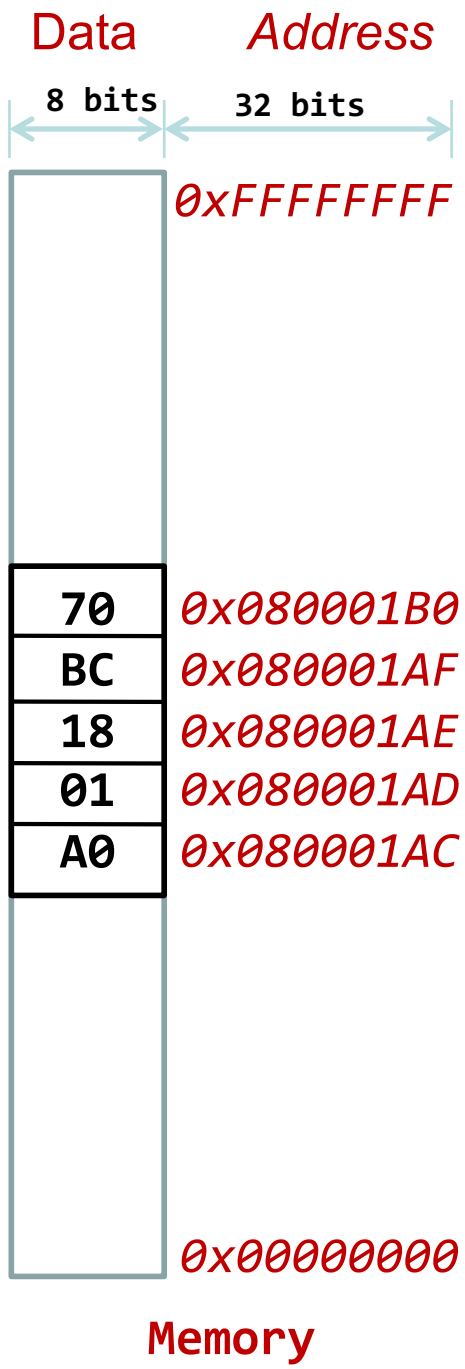


- Cortex M4 - Harvard architecture
  - Separated data bus and instruction bus
  - Advantage: opcodes and operands can go in and out of the CPU together.
  - Disadvantage: leads to more cost in general purpose computers

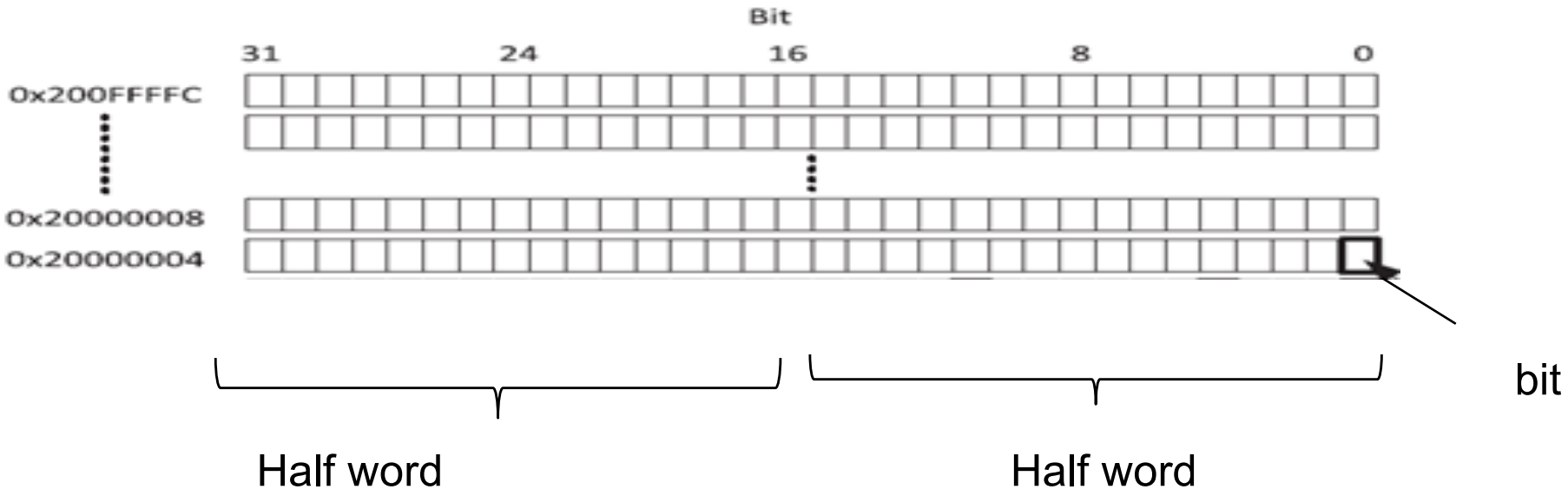


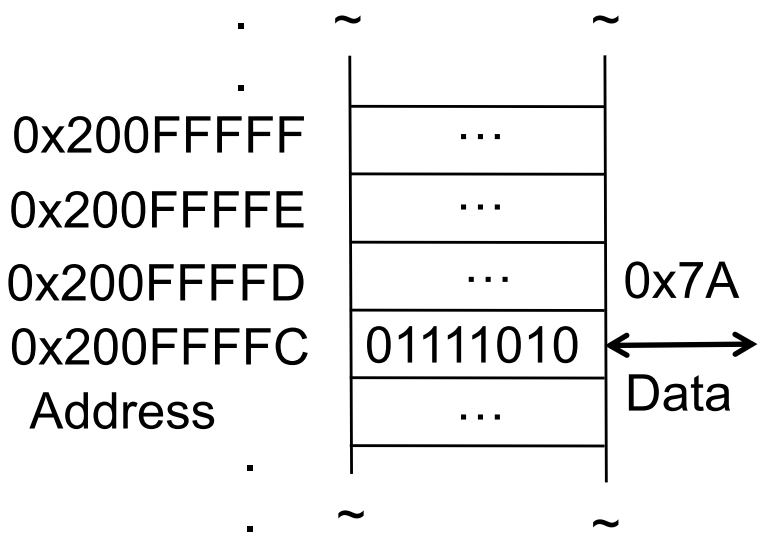
Memory

- Memory is arranged as a series of “locations”
  - Each location has a unique “address”
  - Each location holds a byte (***byte-addressable***)
  - e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., `112`
- The number of locations in memory is limited
  - e.g. 4 GB of RAM
  - 1 Gigabyte (GB) = 2^{30} bytes
  - 2^{32} locations ➔ 4,294,967,296 locations!
- Values stored at each location can represent either program data or program instructions
  - e.g. the value `0x70` might be the code used to tell the processor to add two values together



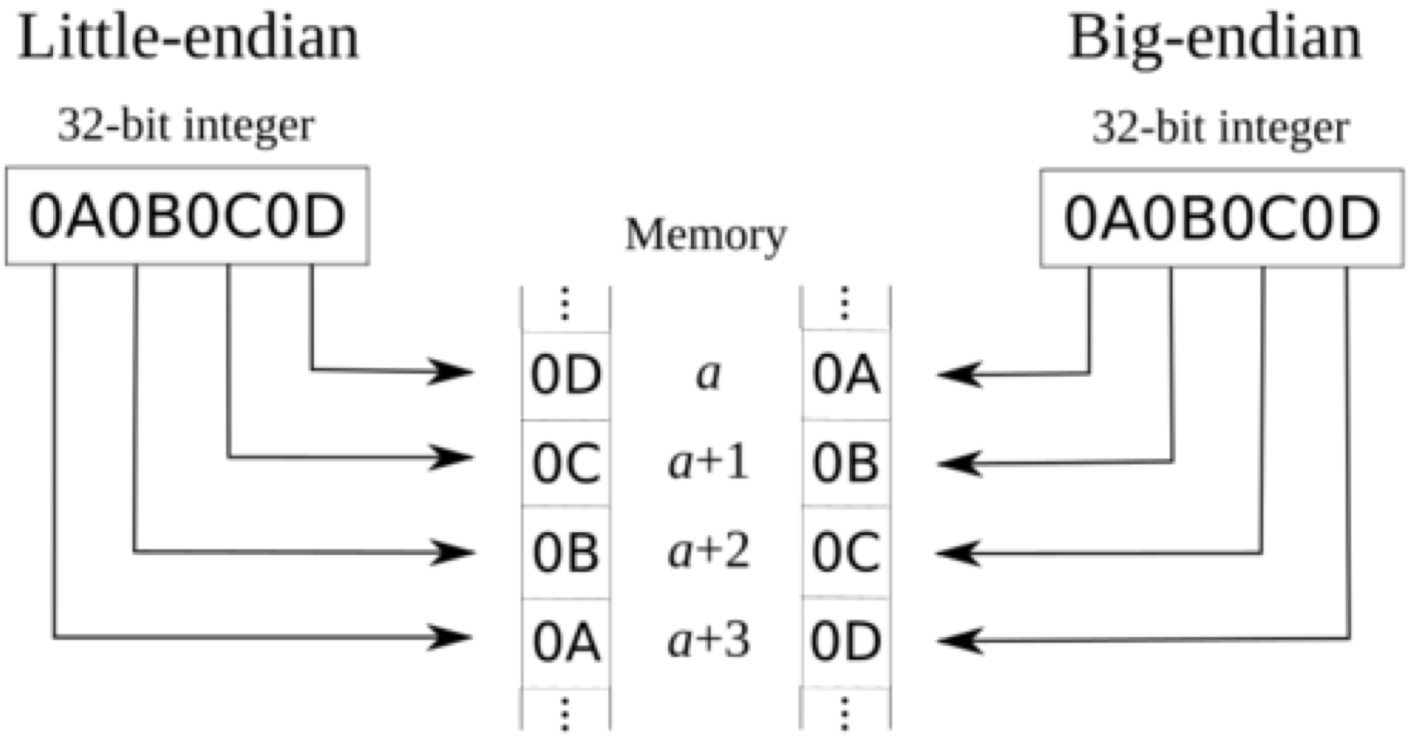
32 bit


Bit
31
24
16
8
0
0x200FFFFC
0x20000008
0x20000004
bit
Half word
Half word


0x200FFFFF
0x200FFFFE
0x200FFFFD
0x200FFFFC
Address
…
…
…
01111010
…
0x7A
Data

Endianness

- In the Little Endian format, the least significant byte (LSB) is stored in the lowest address of the memory, with the most significant byte (MSB) stored in the highest address location of the memory. (ARM is Little Endian by default)
- In the Big Endian format, the least significant byte (LSB) is stored in the highest address of the memory, with the most significant byte (MSB) stored in the lowest address location of the memory.



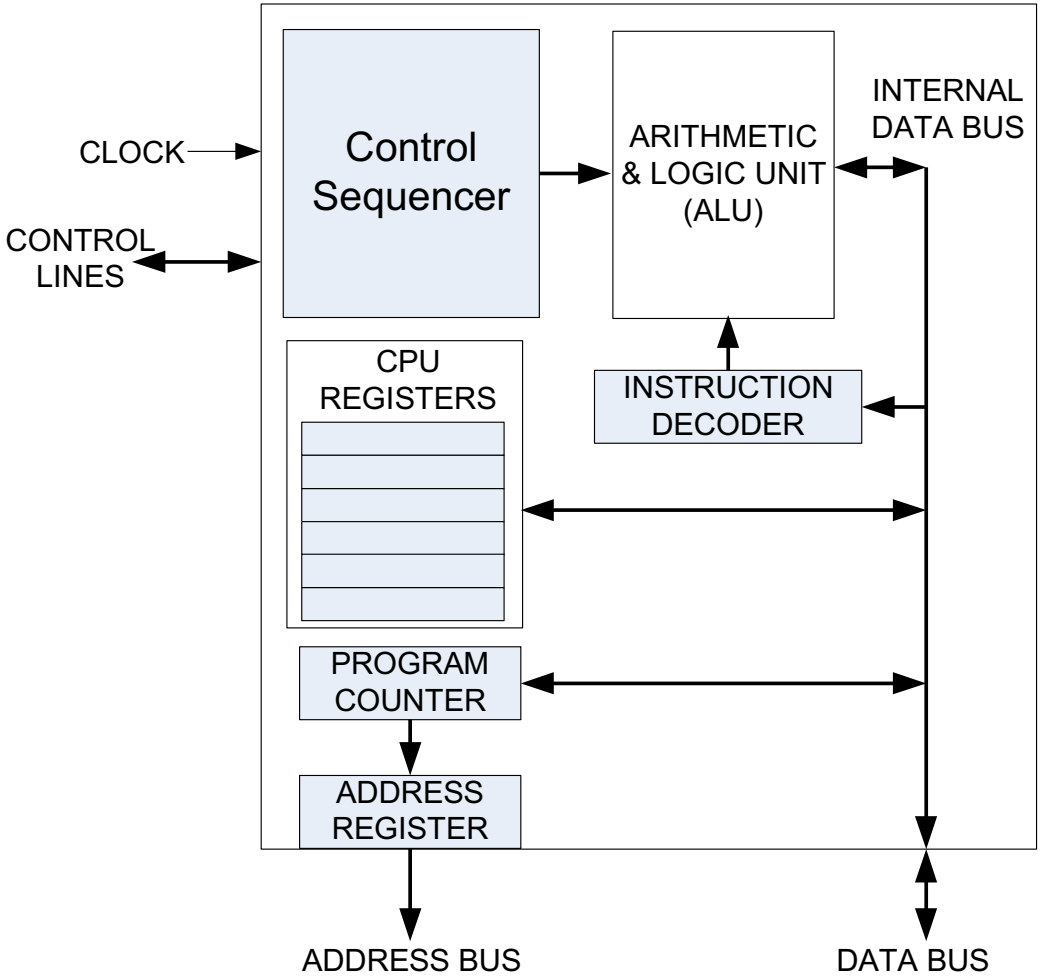
Program Execution

Remember: Microcode or Microprogram

- Each instruction is executed by the CPU through a sequence of well-defined steps.
- The recipe of execution per instruction is stored in the CPU's control sequencer, and is called the *Microcode* or *Microprogram. A microprogram* is written during the design phase of the chip and is normally not accessible afterward.
- The available hardware resources combined with the efficiency in microcode determines the number of clock cycles each instruction takes to execute.

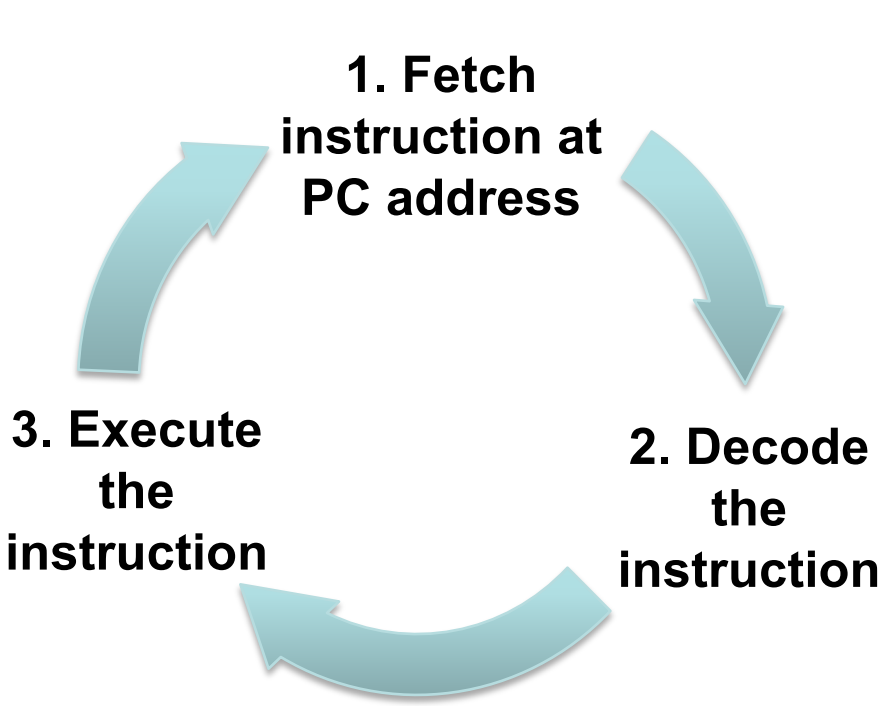
Program Execution

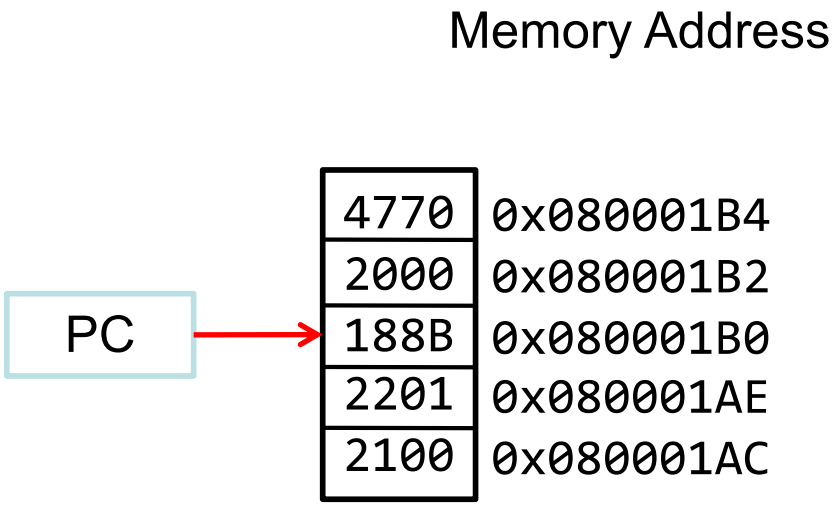
Remember: CPU architecture



Program Execution

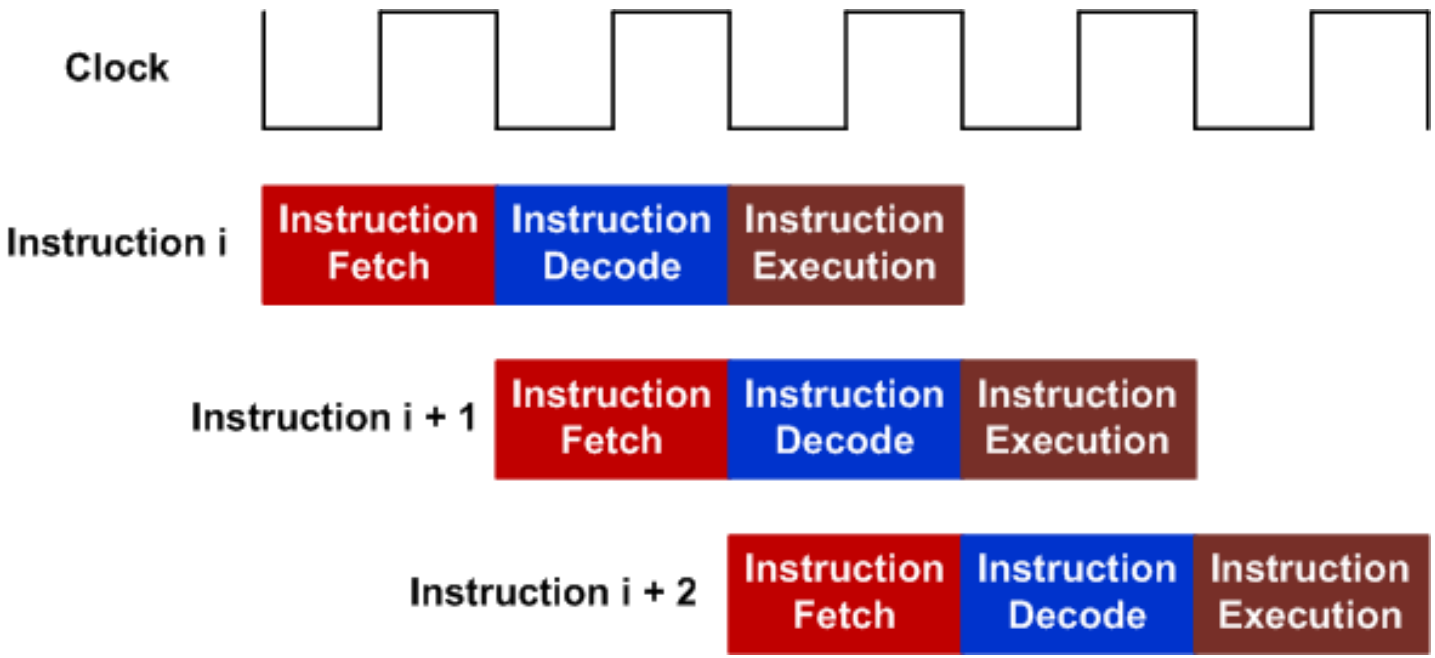
- **Program Counter (PC)** is a register that holds the memory address of the next instruction to be fetched from the memory.





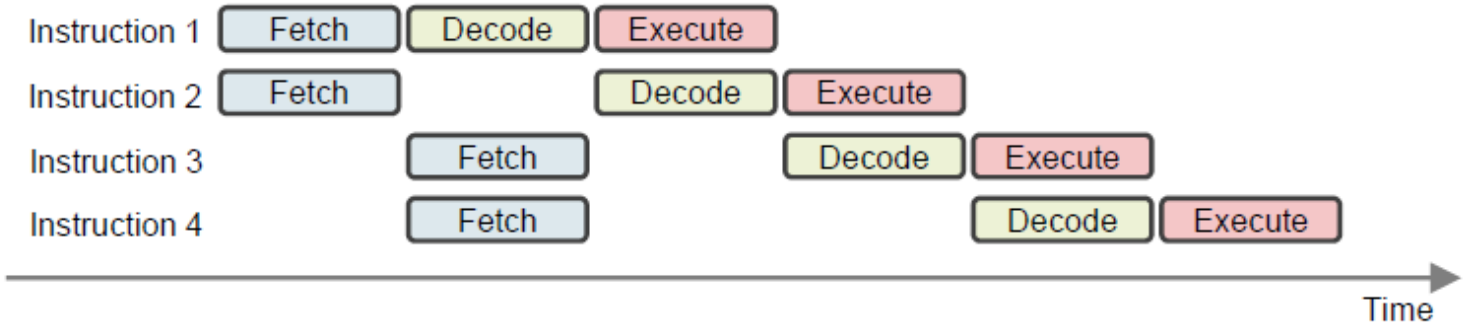
Three-state pipeline: Fetch, Decode, Execution

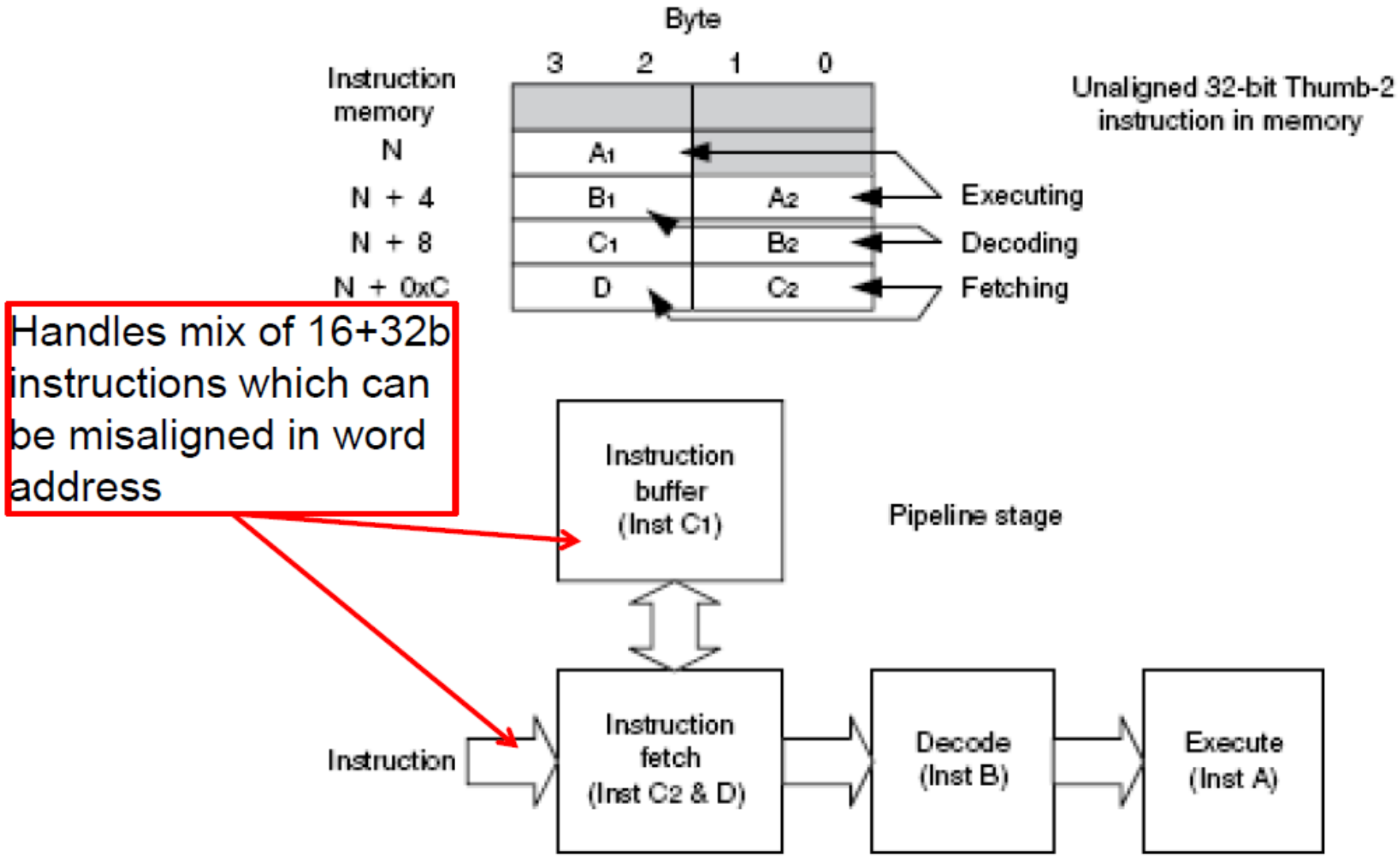
- **Pipelining** allows hardware resources to be fully utilized.



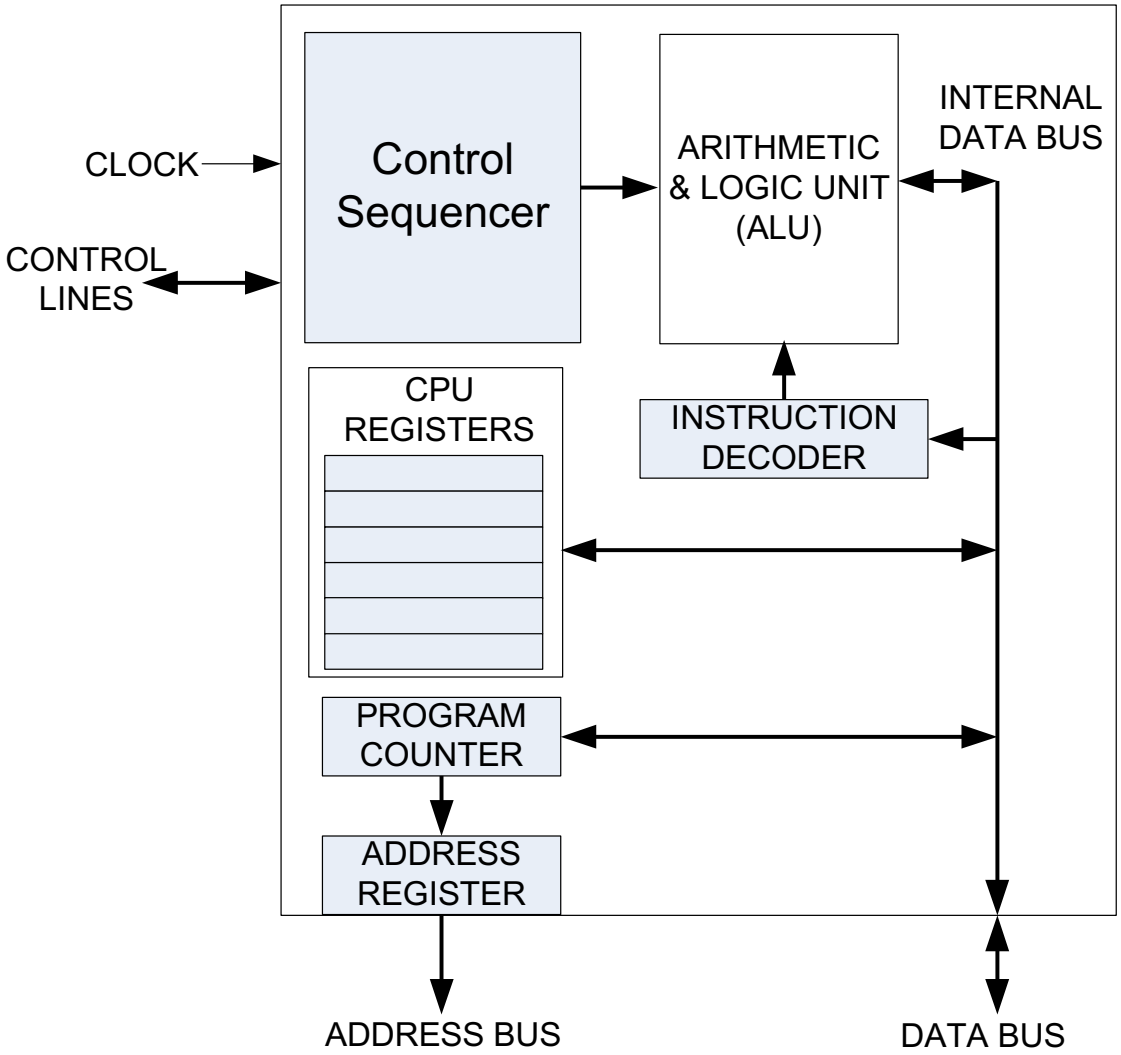
Cortex-M4 Processor - Pipeline

- Three-stage pipeline: fetch, decode, and execute
- Because of its pipeline architecture, an instruction fetch and a data access are performed at the same time.
- Since the buses are separated, the accesses are not interfered for each other.
- Some instructions may take multiple cycles to execute, in which case the pipeline will be stalled
- The pipeline will be flushed if a branch instruction is executed
- One 32 bit or Up to two 16-bit instructions can be fetched in one transfer




Byte
3
2
1
0
Instruction memory
N
N + 4
N + 8
N + 0xC
A1
B1
A2
C1
B2
D
C2
Executing
Decoding
Fetching
Unaligned 32-bit Thumb-2 instruction in memory
Handles mix of 16+32b instructions which can be misaligned in word address
Instruction buffer (Inst C1)
Pipeline stage
Instruction
Instruction fetch (Inst C2 & D)
Decode (Inst B)
Execute (Inst A)

Instruction Execution



The main functions of the CPU are:

- Data transfer
- Arithmetic and logic operations
- Decision making (instruction flow control)

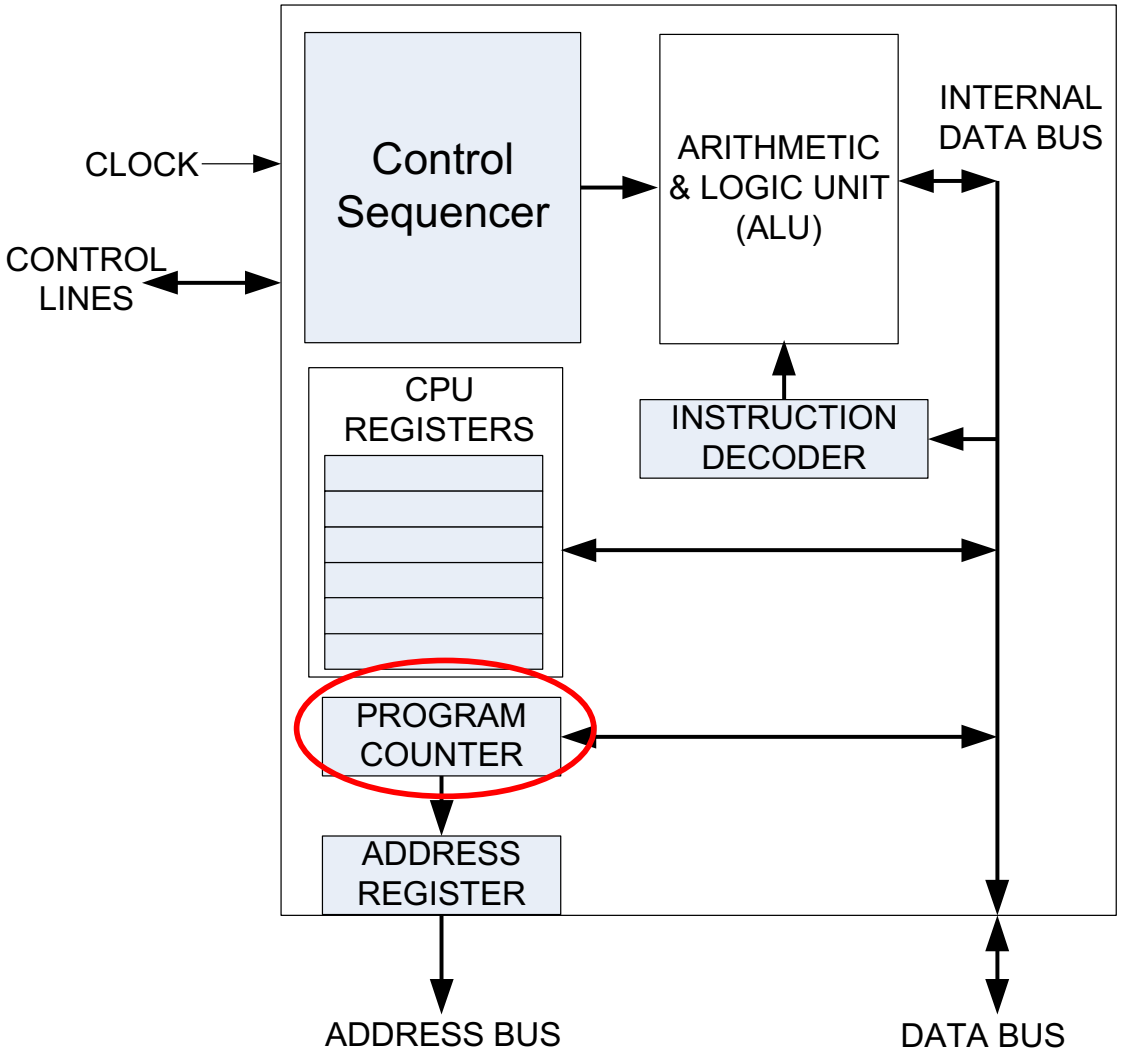
Control Sequencer:

- Controls the operations in the CPU

Register Array:

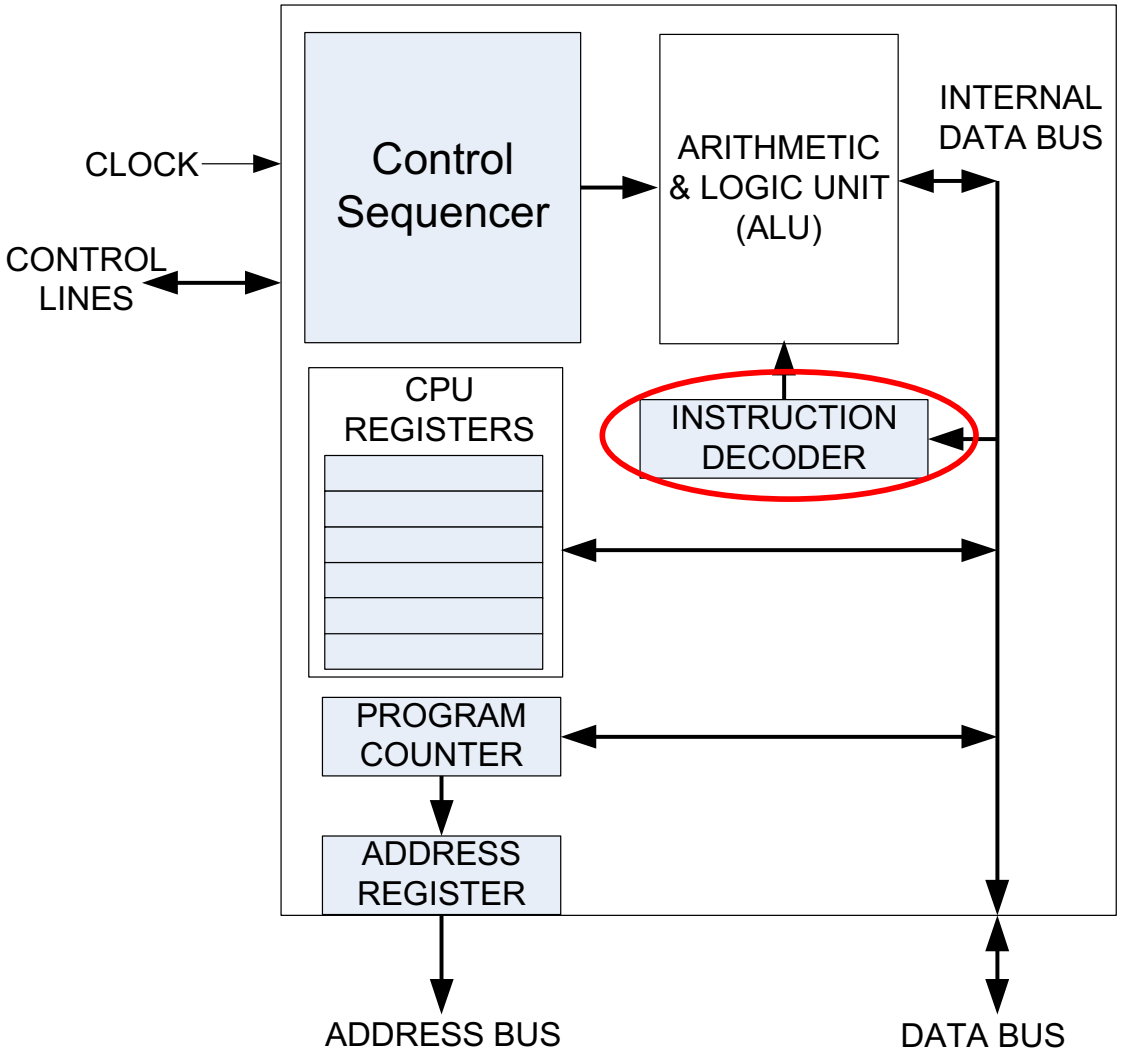
- Temporary storage (faster than memory)

Fetch/(Decode)/Execute Cycle



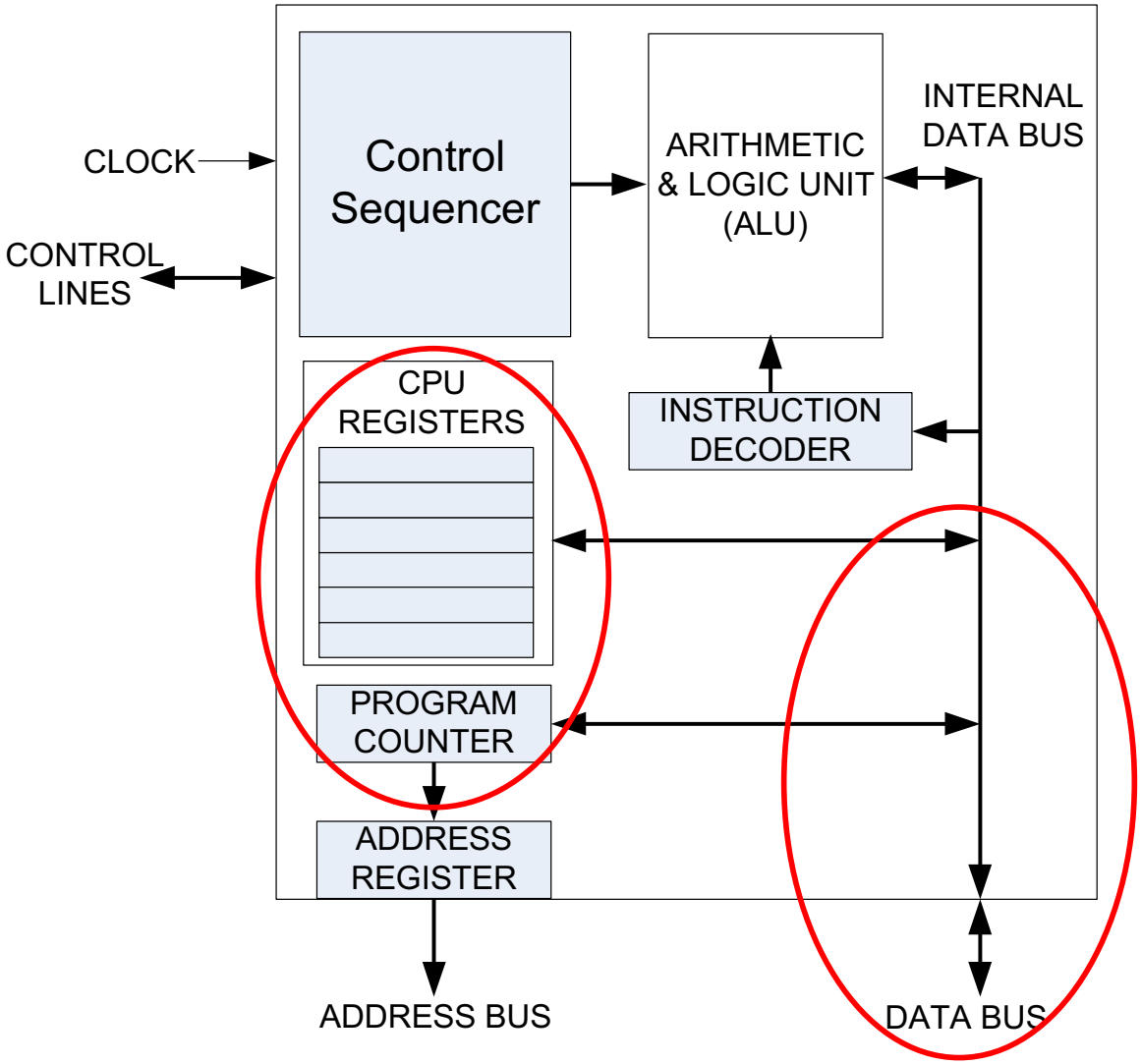
1. CPU keeps track of the location of the next instruction or data byte (to fetch) through the Program Counter (PC).

Fetch/(Decode)/Execute Cycle



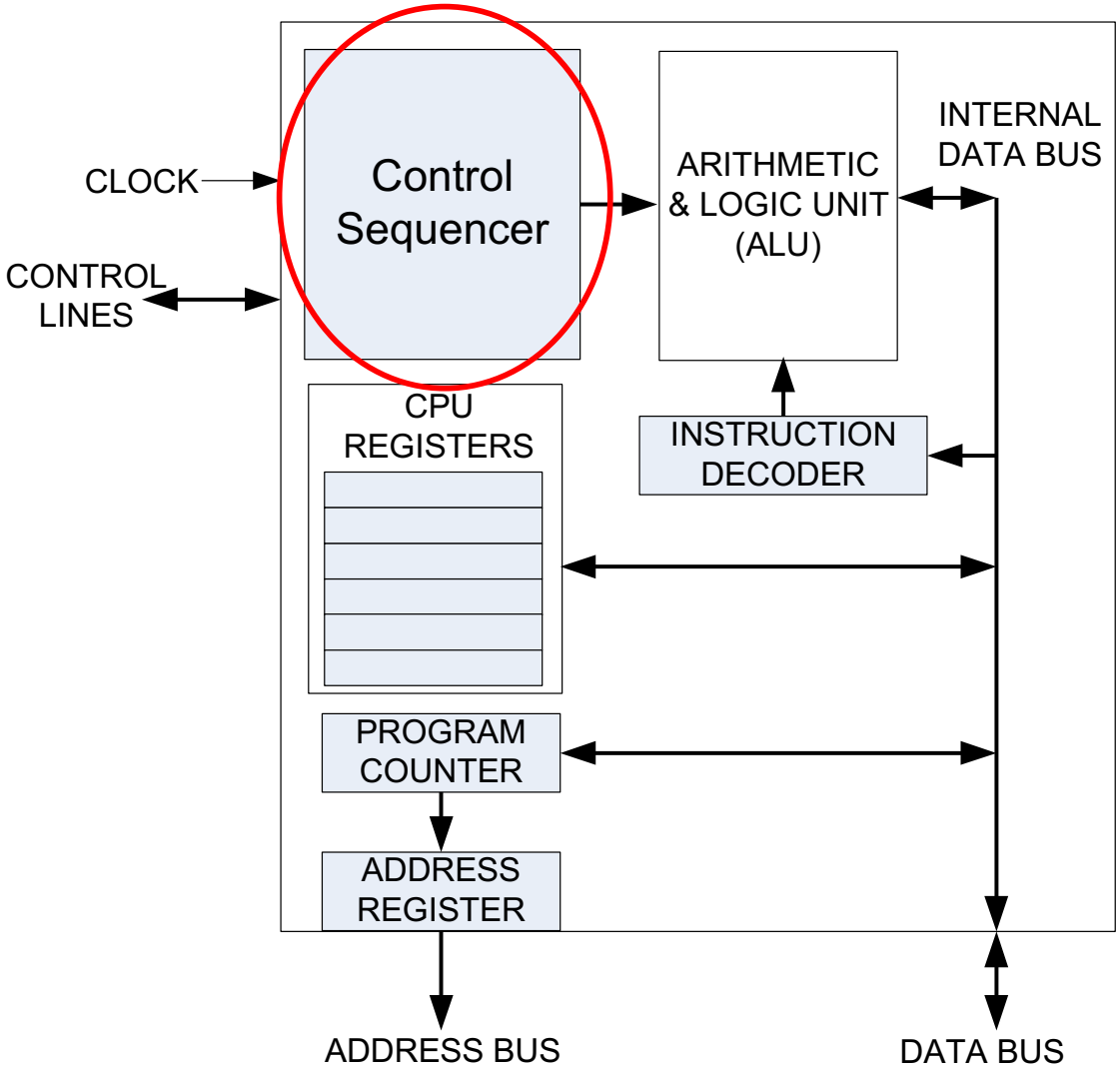
1. CPU keeps track of the location of the next instruction or data byte (to fetch) through the Program Counter (PC).
2. Instruction Decoder decodes the received data and configures the Arithmetic Logic Unit (ALU) to process it. Arithmetic or logic operations execute in the ALU.

Fetch/(Decode)/Execute Cycle



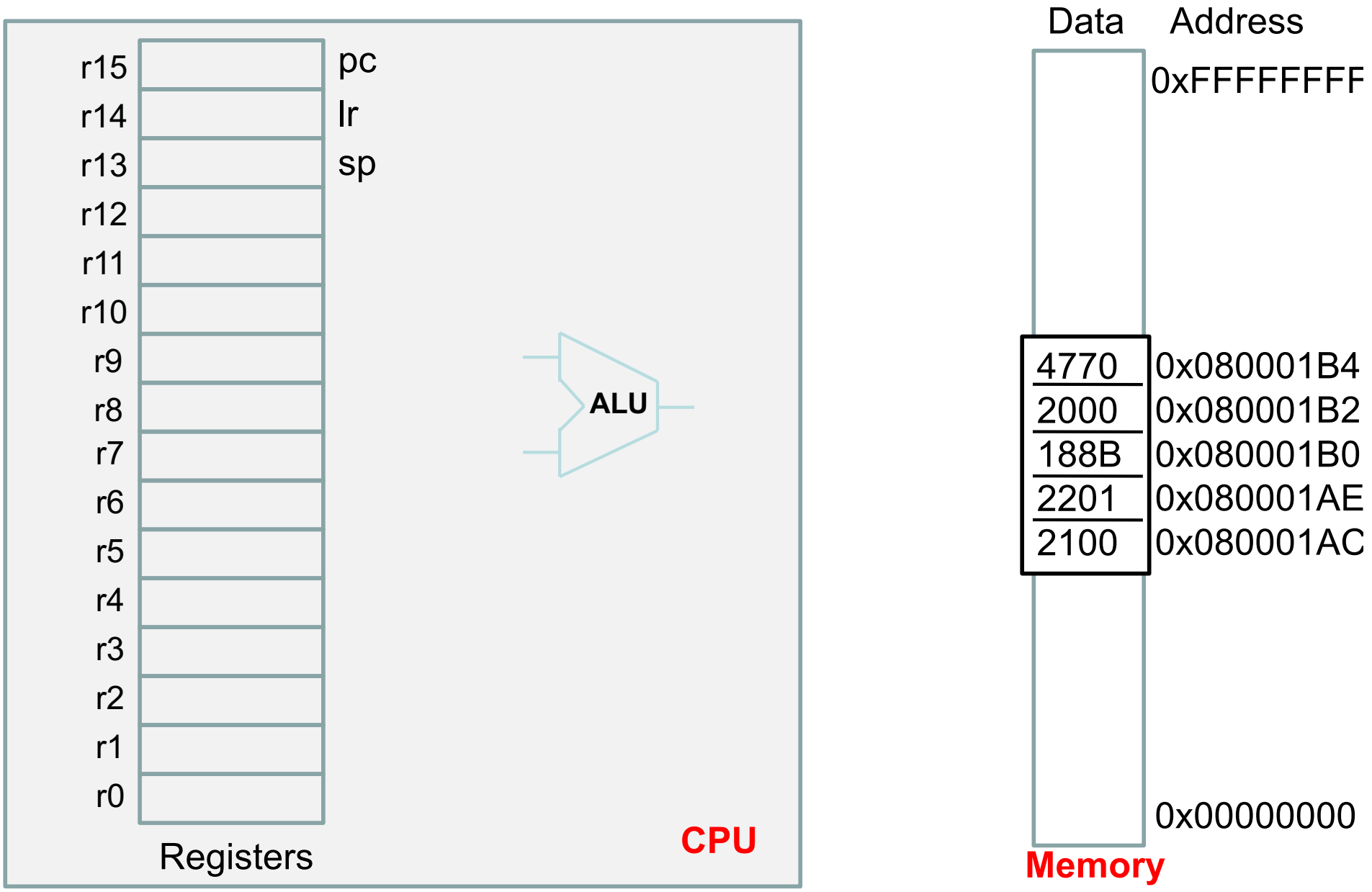
1. CPU keeps track of the location of the next instruction or data byte (to fetch) through the Program Counter (PC).
2. Instruction Decoder decodes the received data and configures the Arithmetic Logic Unit (ALU) to process it. Arithmetic or logic operations execute in the ALU.
3. After the execution, the relevant bits are set, and data gets stored to a register or memory location if applicable.

Fetch/(Decode)/Execute Cycle

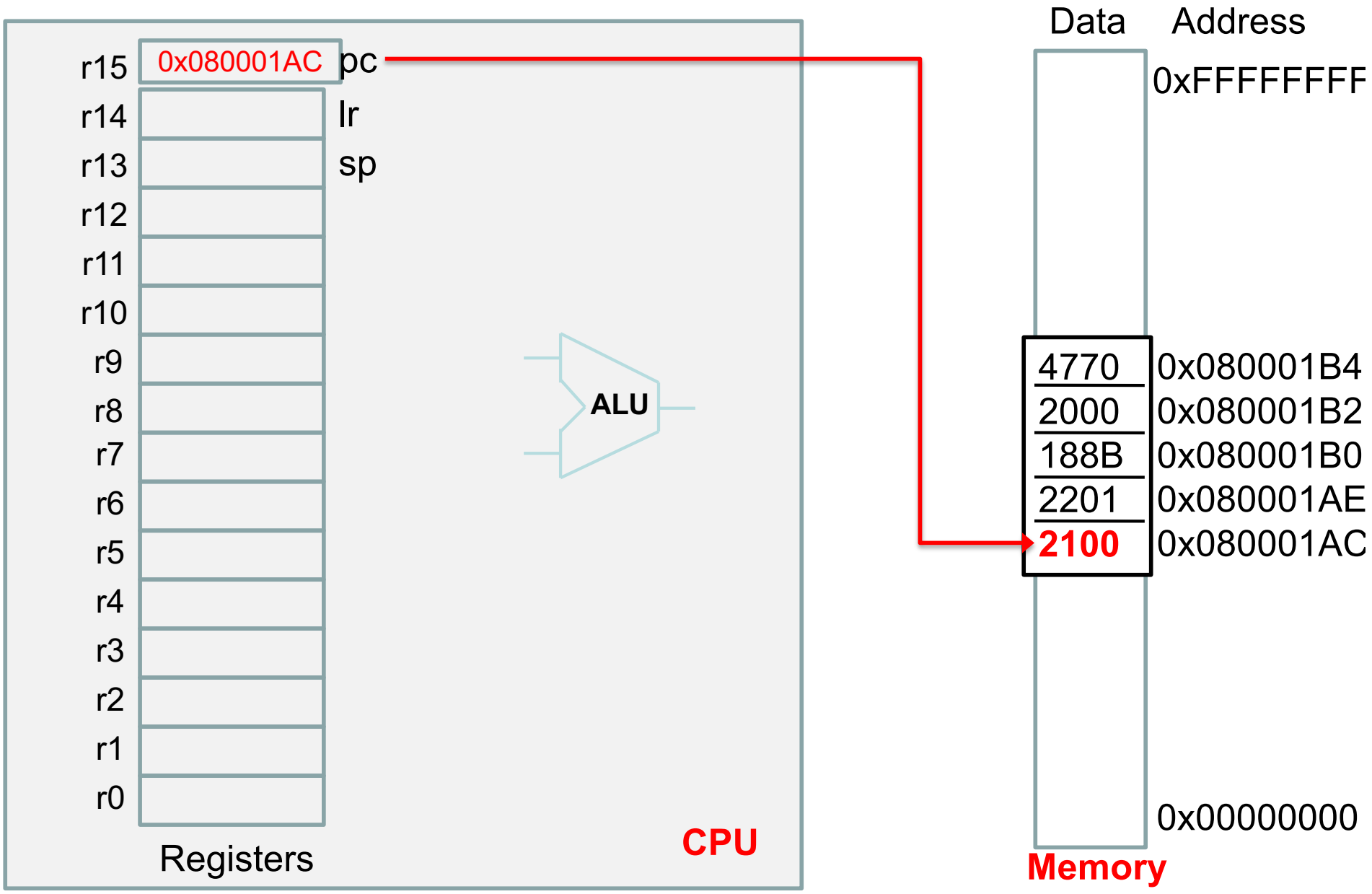


1. CPU keeps track of the location of the next instruction or data byte (to fetch) through the Program Counter (PC).
2. Instruction Decoder decodes the received data and configures the Arithmetic Logic Unit (ALU) to process it. Arithmetic or logic operations execute in the ALU.
3. After the execution, the relevant bits are set, and data gets stored to a register or memory location if applicable.
4. The Control Sequencer ensures the Fetch/Decode/Execute cycle repeats until the last instruction is detected.

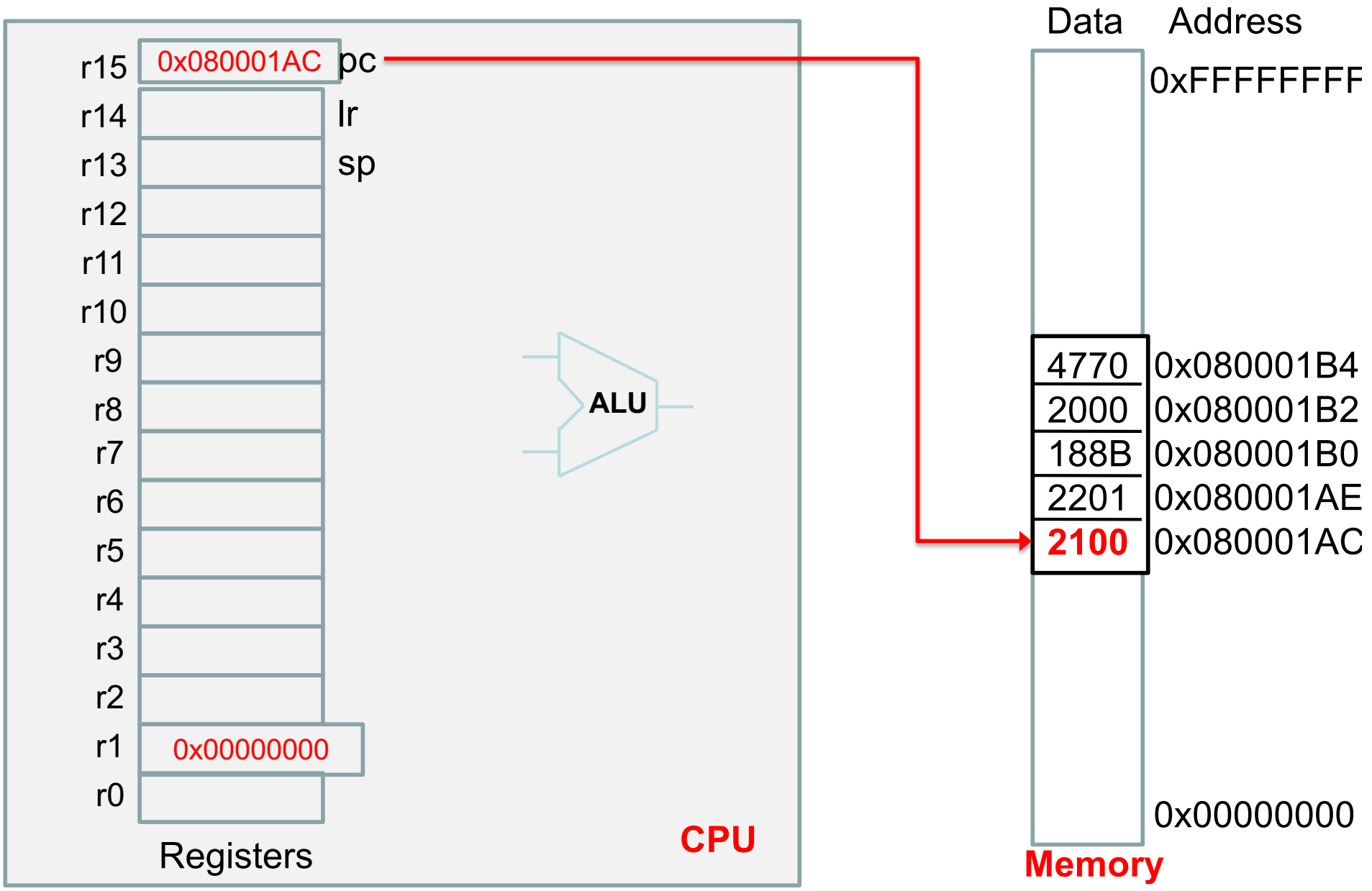
Instruction execution for ARM



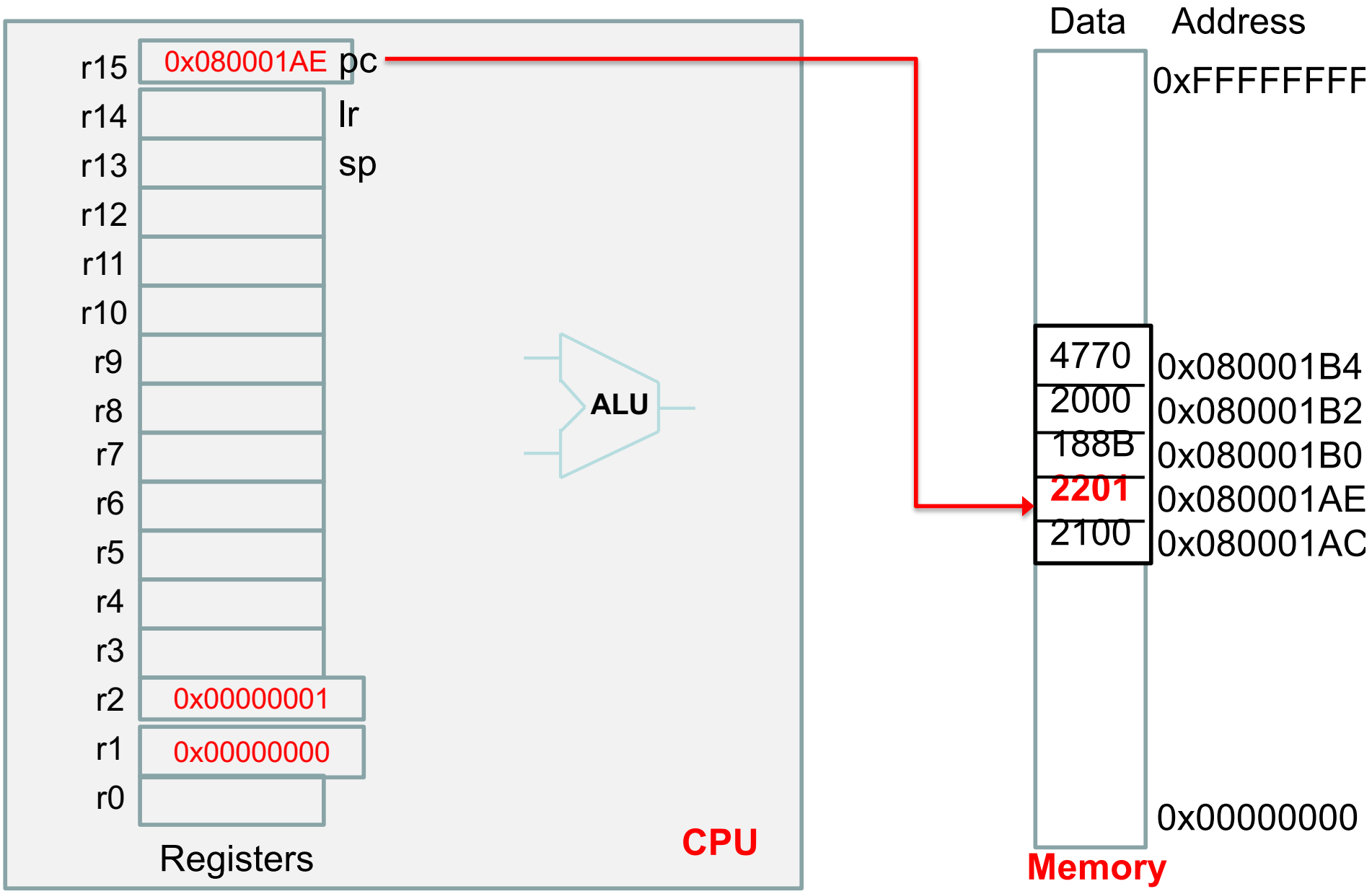
Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00



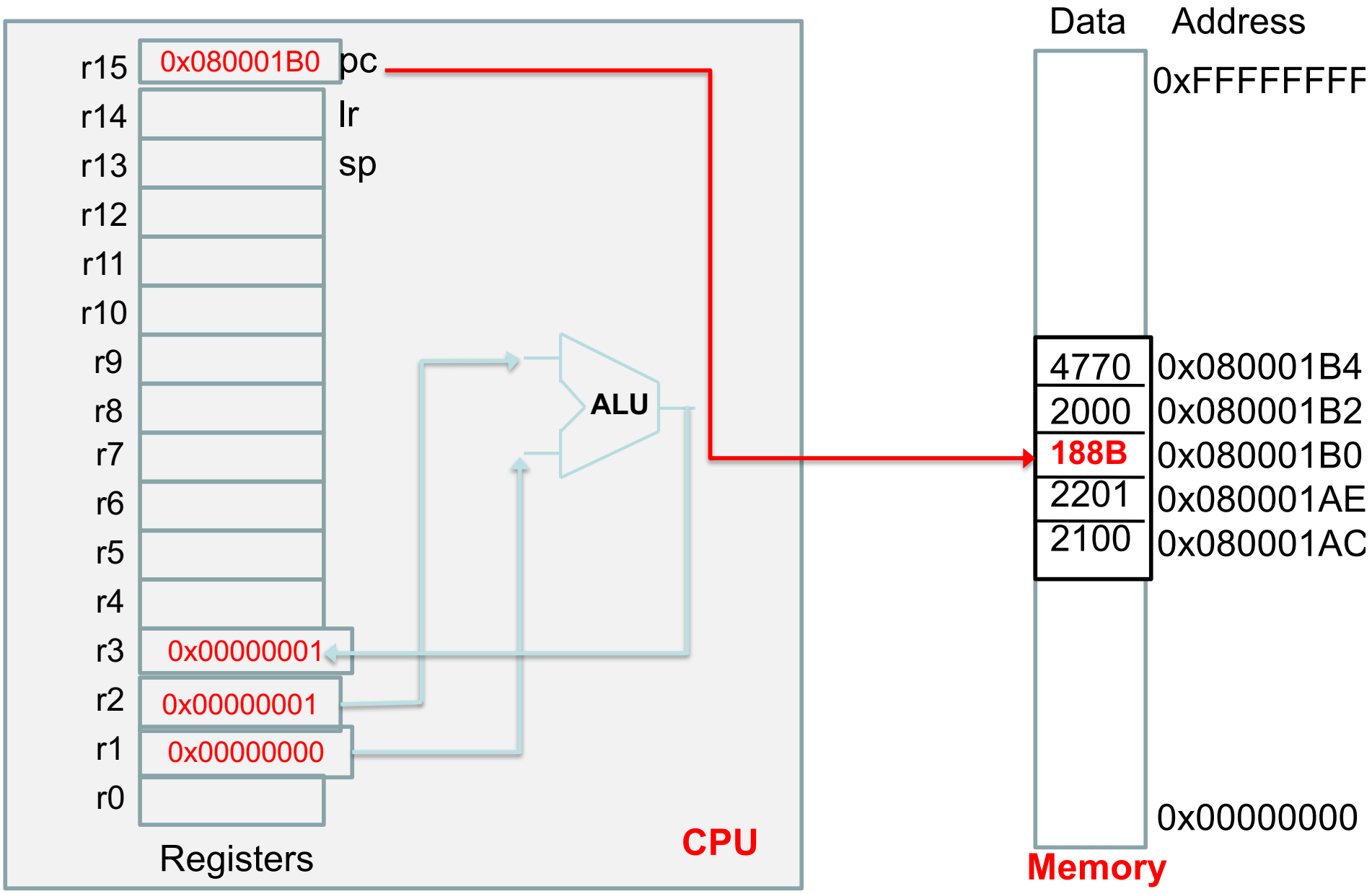
Execute Instruction:
MOVS r1, #0x00



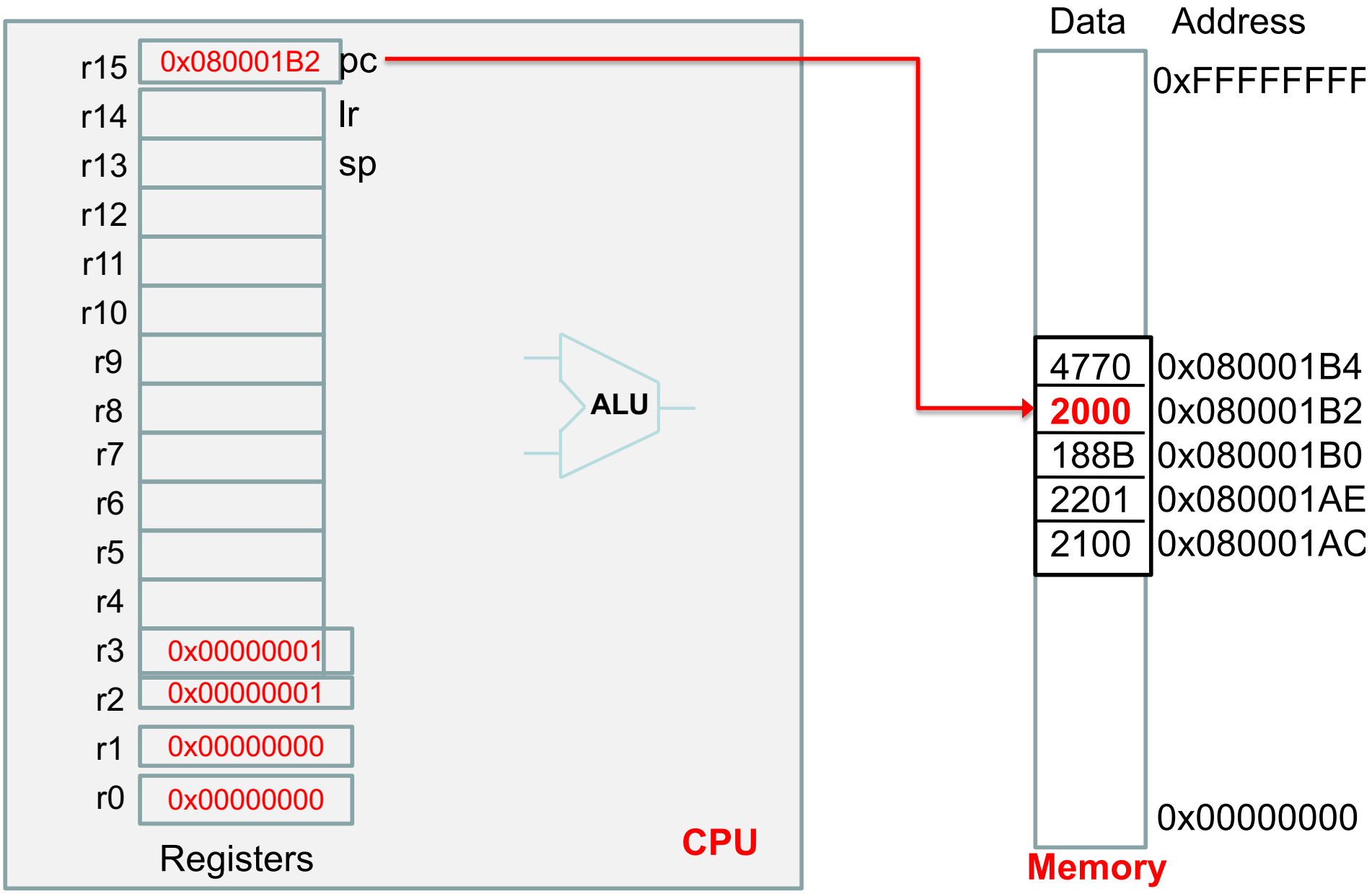
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



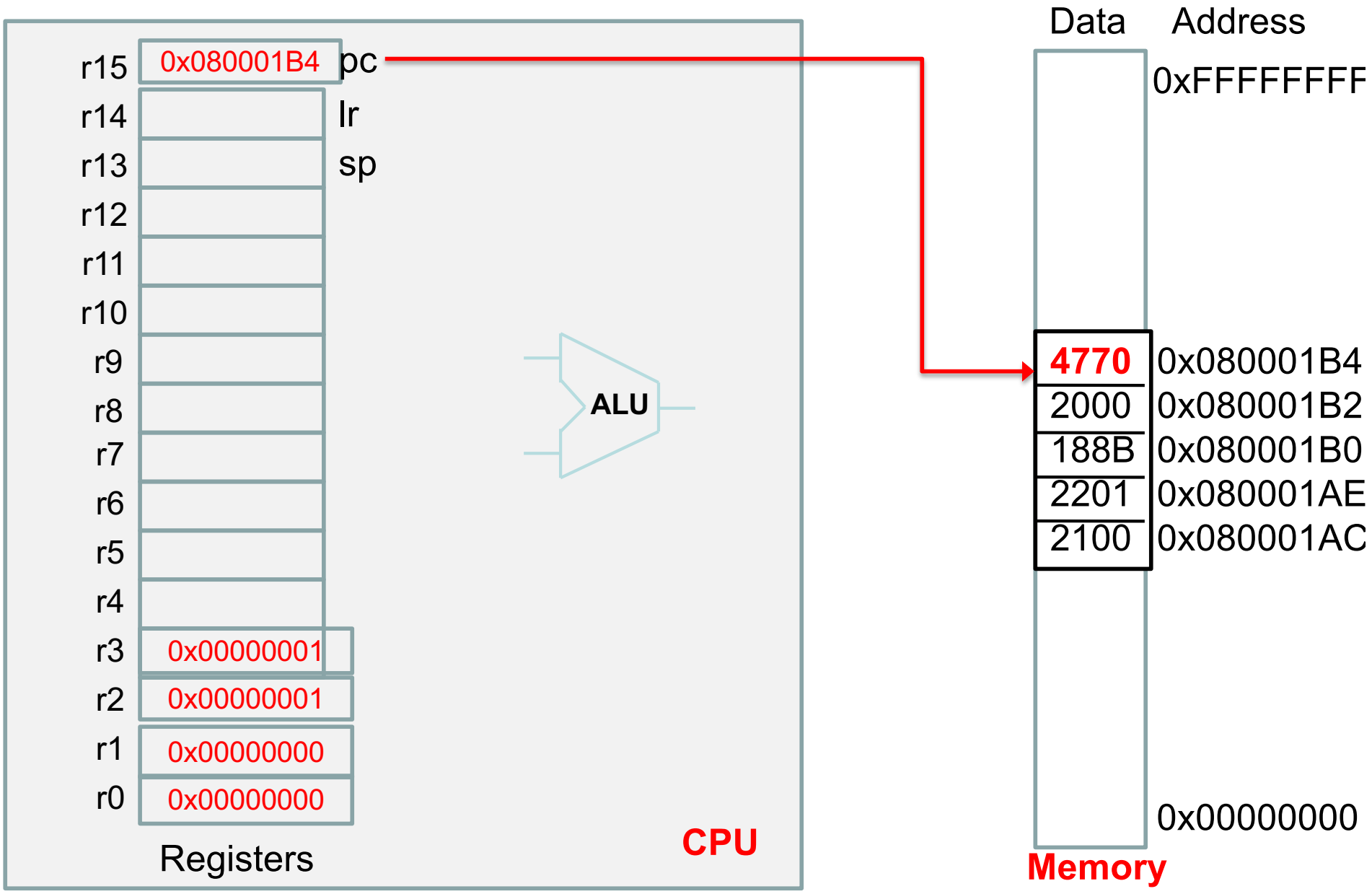
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2



Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, #0x00



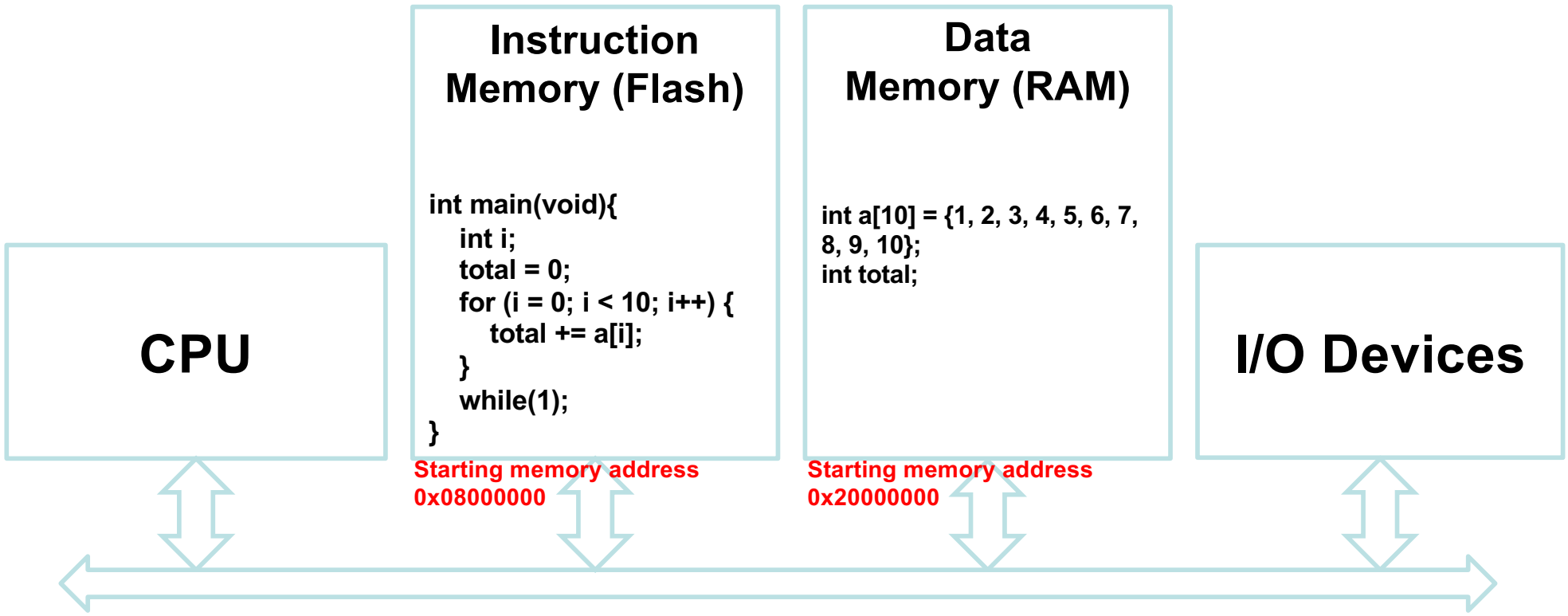
Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr



Example:
Calculate the Sum of an Array

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

int main(void){
        int i;
        total = 0;
        for (i = 0; i < 10; i++) {
                total += a[i];
        }
        while(1);
}
```

Example:
Calculate the Sum of an Array



Example:
Calculate the Sum of an Array

Instruction Memory (Flash)

```
int main(void){
    int i;
    total = 0;
    for (i = 0; i < 10; i++) {
        total += a[i];
    }
    while(1);
}
```

Starting memory address
0x08000000

```
0010 0001 0000 0000
0100 1010 0000 1000
0110 0000 0001 0001
0010 0000 0000 0000
1110 0000 0000 1000
0100 1001 0000 0111
1111 1000 0101 0001
0001 0000 0010 0000
0100 1010 0000 0100
0110 1000 0001 0010
0100 0100 0001 0001
0100 1010 0000 0011
0110 0000 0001 0001
0001 1100 0100 0000
0010 1000 0000 1010
1101 1011 1111 0100
1011 1111 0000 0000
1110 0111 1111 1110
```

```
        MOVS  r1, #0x00
        LDR   r2, = total_addr
        STR   r1, [r2, #0x00]
        MOVS  r0, #0x00
        B     Check
Loop: LDR     r1, = a_addr
        LDR   r1, [r1, r0, LSL #2]
        LDR   r2, = total_addr
        LDR   r2, [r2, #0x00]
        ADD   r1, r1, r2
        LDR   r2, = total_addr
        STR   r1, [r2,#0x00]
        ADDS  r0, r0, #1
Check: CMP    r0, #0x0A
       BLT    Loop
       NOP
Self:  B      Self
```

Example:
Calculate the Sum of an Array

Data Memory (RAM)

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;
```

Assume the starting memory address of the data memory is 0x20000000

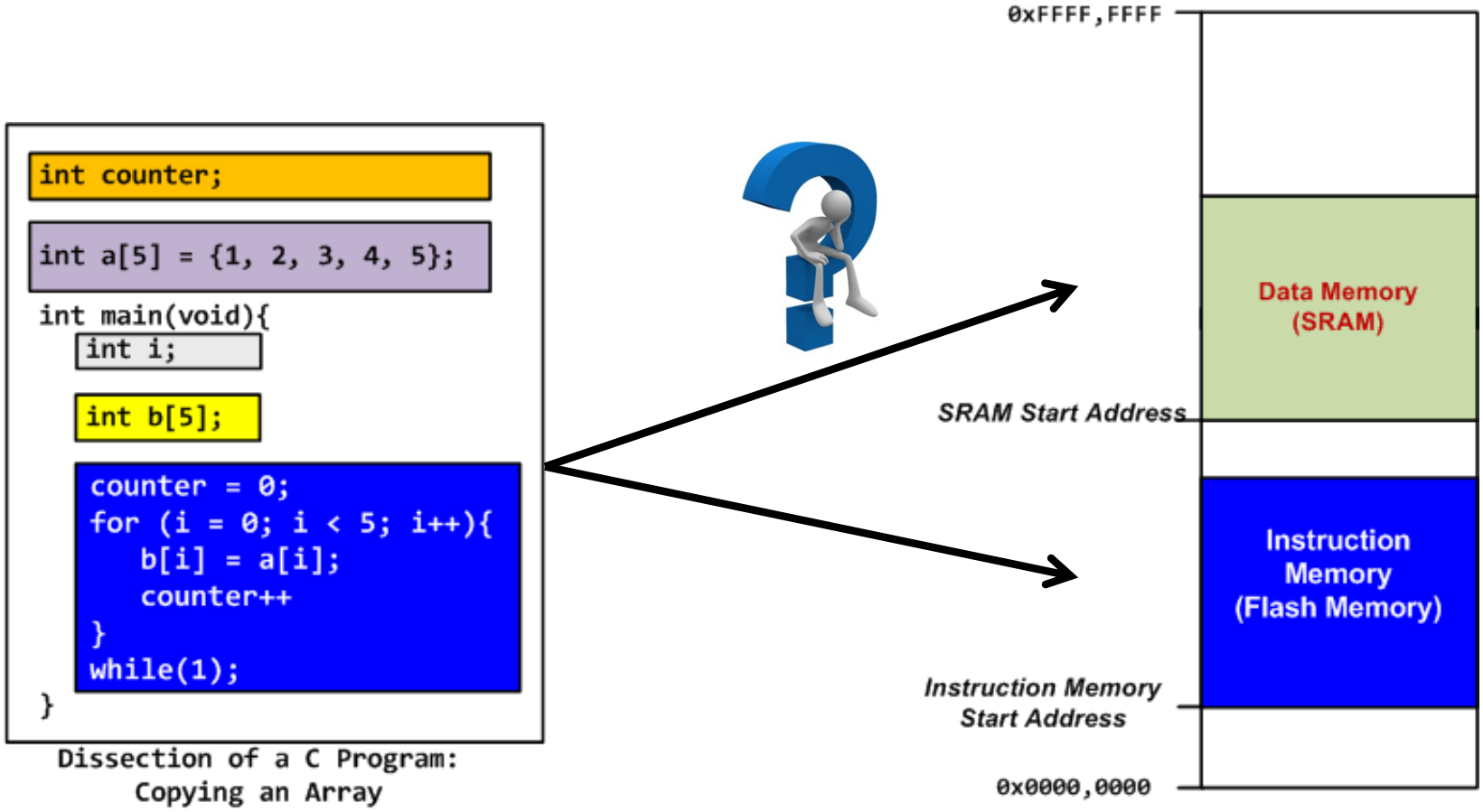
Memory address in bytes	Memory content	
0x20000000	0x0001	a[0] = 0x00000001
0x20000002	0x0000	
0x20000004	0x0002	a[1] = 0x00000002
0x20000006	0x0000	
0x20000008	0x0003	a[2] = 0x00000003
0x2000000A	0x0000	
0x2000000C	0x0004	a[3] = 0x00000004
0x2000000E	0x0000	
0x20000010	0x0005	a[4] = 0x00000005
0x20000012	0x0000	
0x20000014	0x0006	a[5] = 0x00000006
0x20000016	0x0000	
0x20000018	0x0007	a[6] = 0x00000007
0x2000001A	0x0000	
0x2000001C	0x0008	a[7] = 0x00000008
0x2000001E	0x0000	
0x20000020	0x0009	a[8] = 0x00000009
0x20000022	0x0000	
0x20000024	0x000A	a[9] = 0x0000000A
0x20000026	0x0000	
0x20000028	0x0000	total= 0x00000000
0x2000002A	0x0000	

Loading Code and Data into Memory

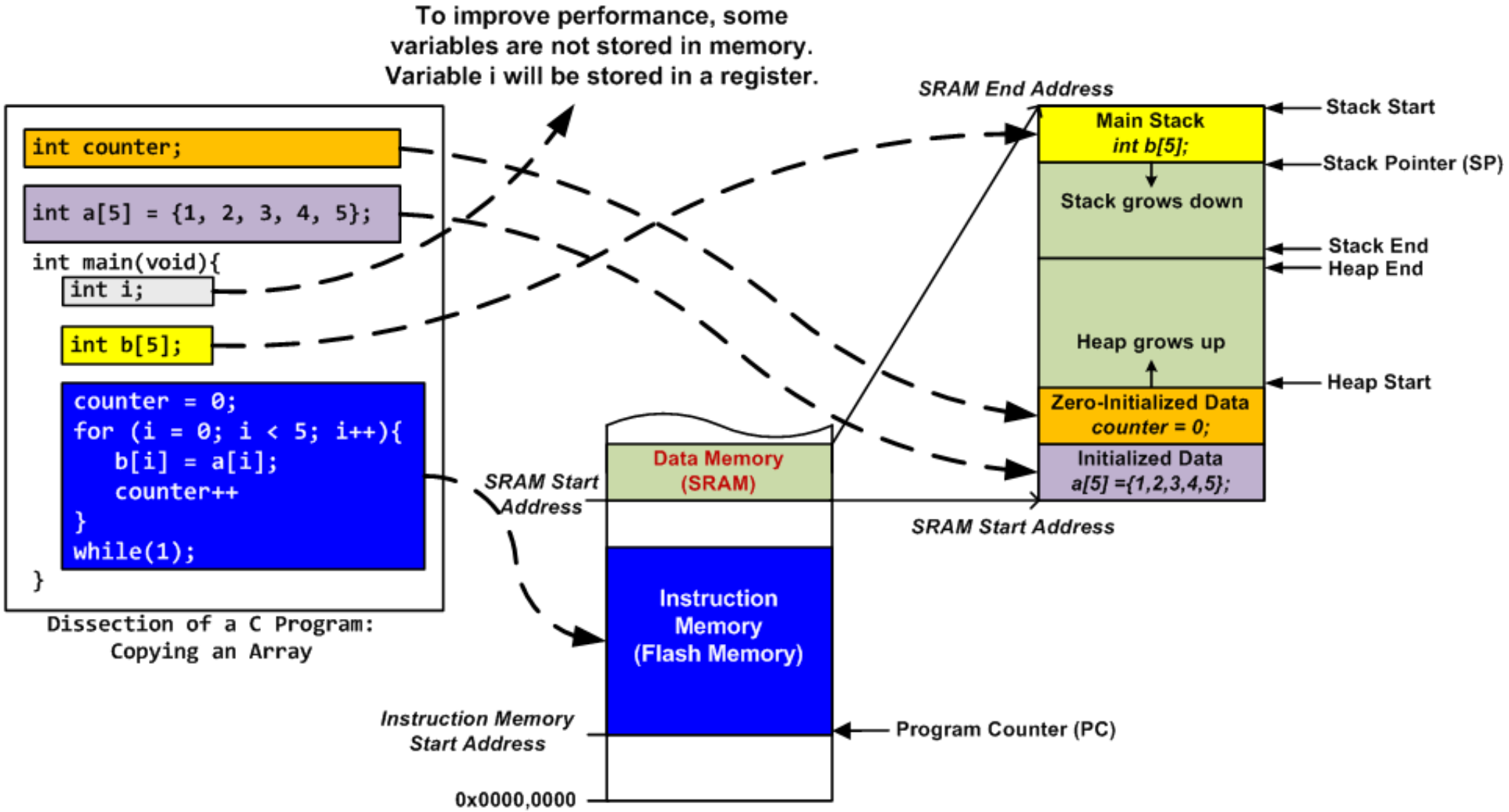
```
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
   int i;

   int b[5];

   counter = 0;
   for (i = 0; i < 5; i++){
      b[i] = a[i];
      counter++
   }
   while(1);
}
```

C Program:
Copying an Array

Loading Code and Data into Memory

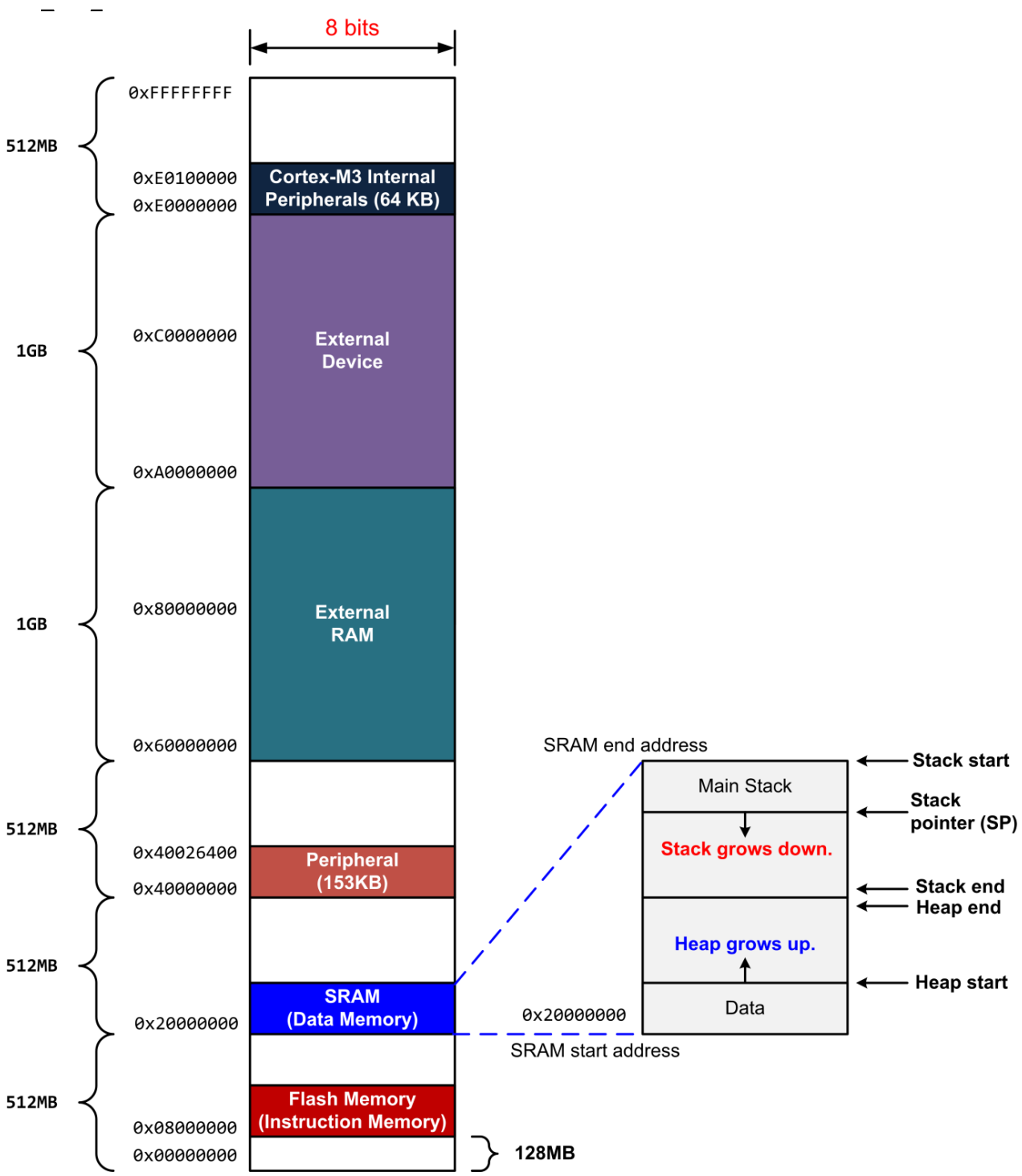


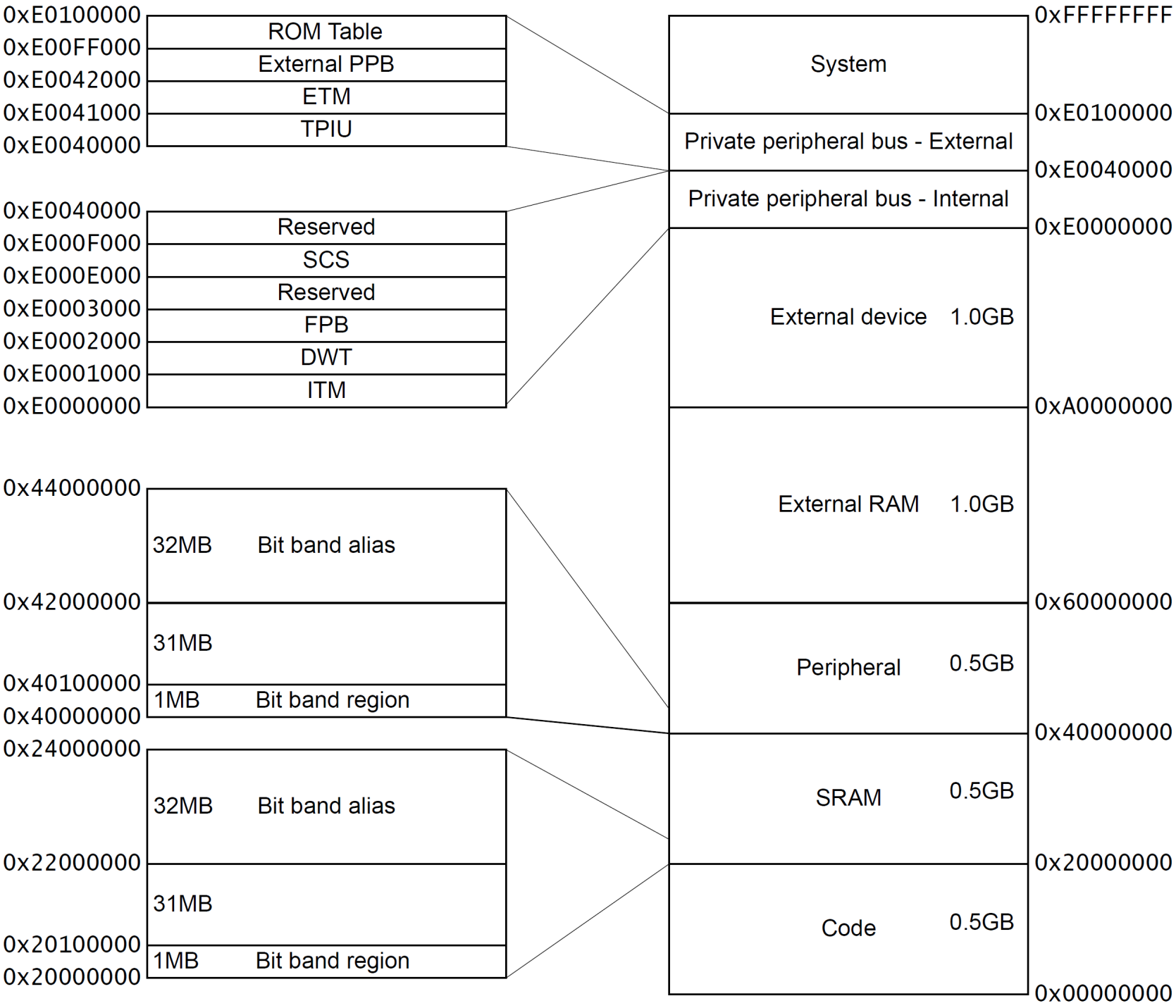
Loading Code and Data into Memory



Cortex-M4 processor has 32 bits for addressing and this means that it is supporting 4GB of memory space.

The memory space is used by the program code, data, peripherals, and some of the debug support components inside the processors.

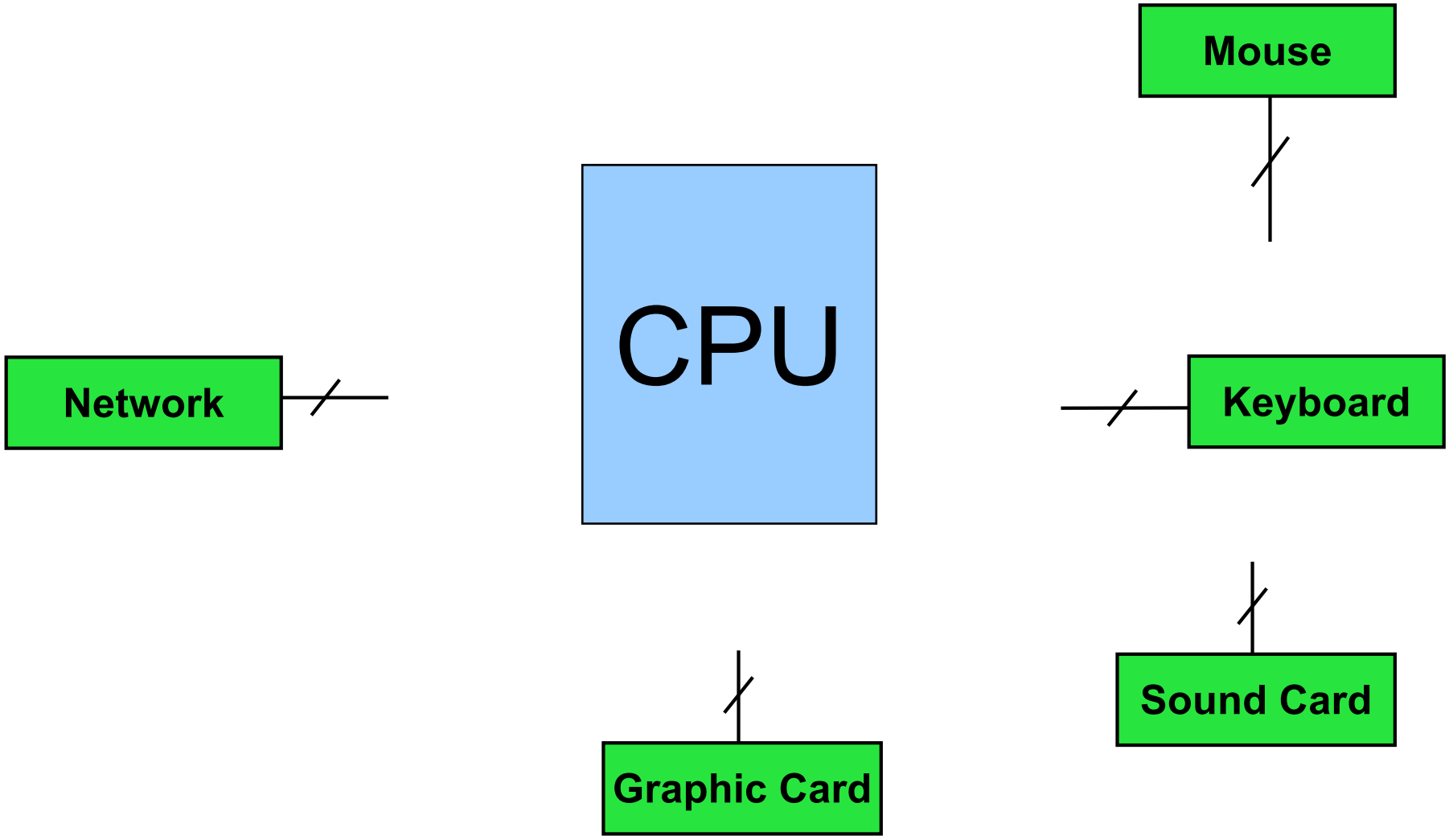




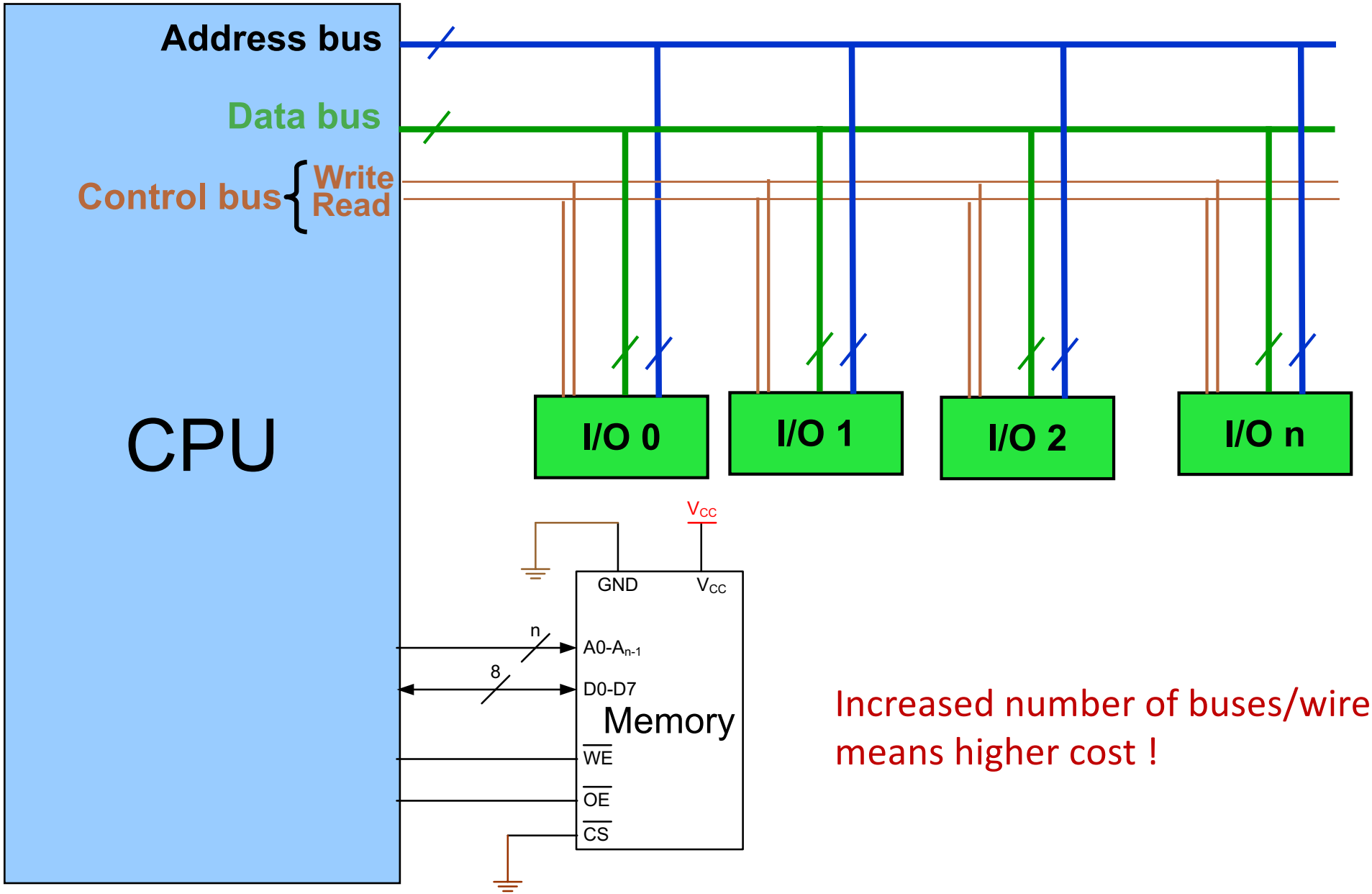
Cortex-M4 Fixed Memory Map

Connecting I/O devices to CPU

- CPU should have lots of pins!

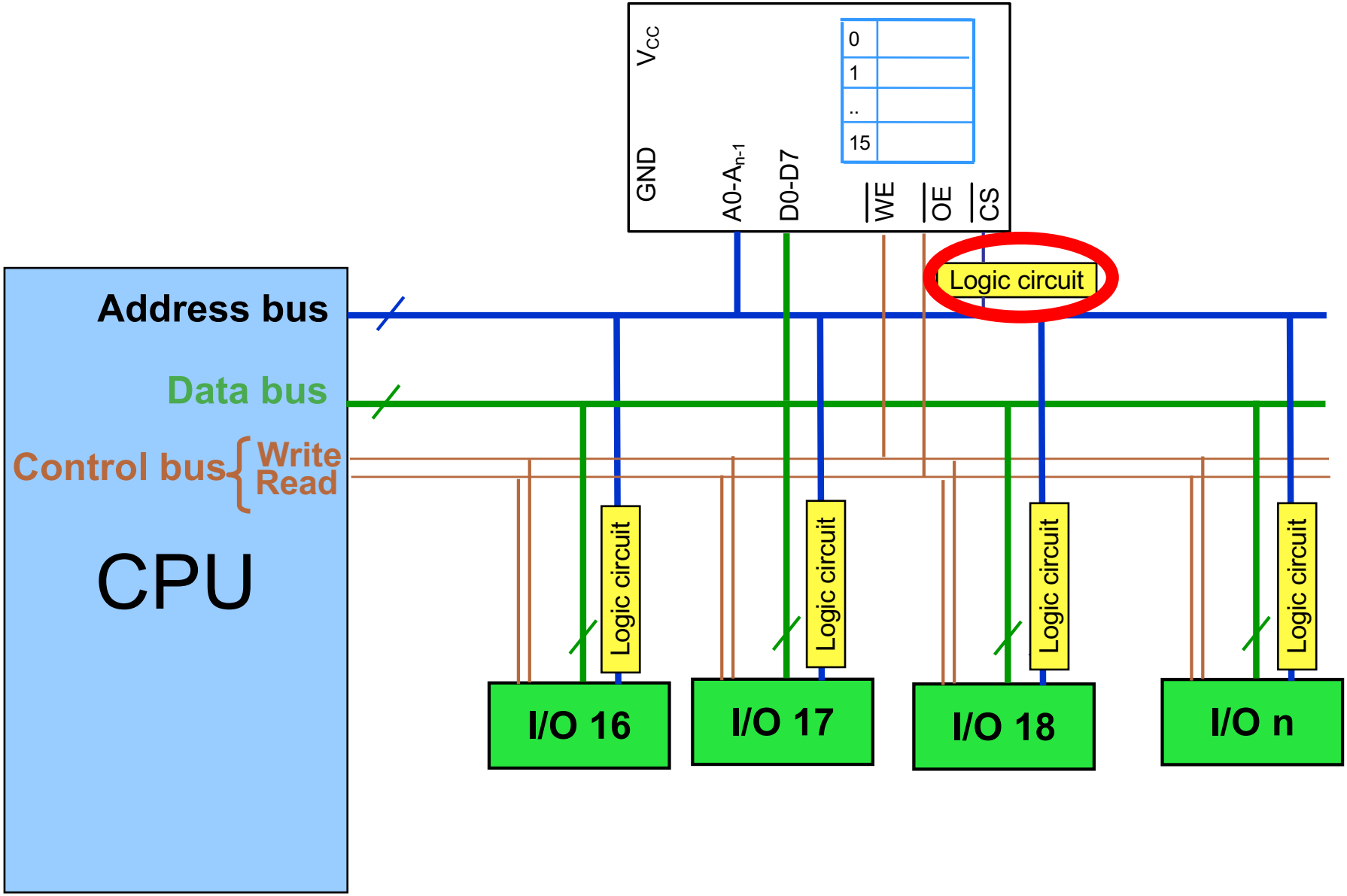


Connecting I/Os and Memory to CPU (Isolated I/0)



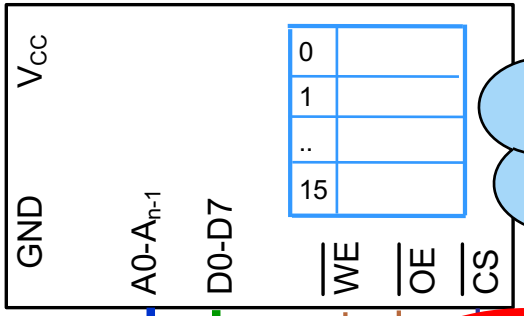
Increased number of buses/wires means higher cost !

Connecting I/Os and Memory to CPU
Using a Shared Bus (Memory Mapped I/O) – Cortex M4



Connecting I/Os and Memory to CPU Using a Shared Bus (Memory Mapped I/O scheme)

How to design the logic circuit?



The logic circuit enables CS when address is between 0 and 15

Logic circuit

8

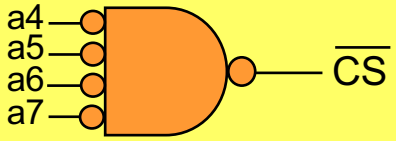
Address bus

Co

Solution

1. Write the address range in binary
2. Separate the fixed part of address
3. Using a NAND, design a logic circuit whose output activates when the fixed address is given to it.

	a7	a6	a5	a4	a3	a2	a1	a0
From address 0 ➔	0	0	0	0	0	0	0	0
To address15 ➔	0	0	0	0	1	1	1	1



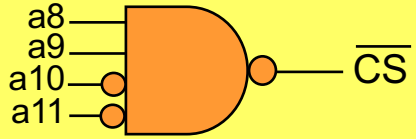
Address Decoding

- Design an address decoder for address of 300H to 3FFH (for a 12-bit address bus)

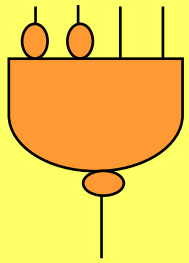
Solution

1. Write the address range in binary
2. Separate the fixed part of address
3. Design the logic circuit.

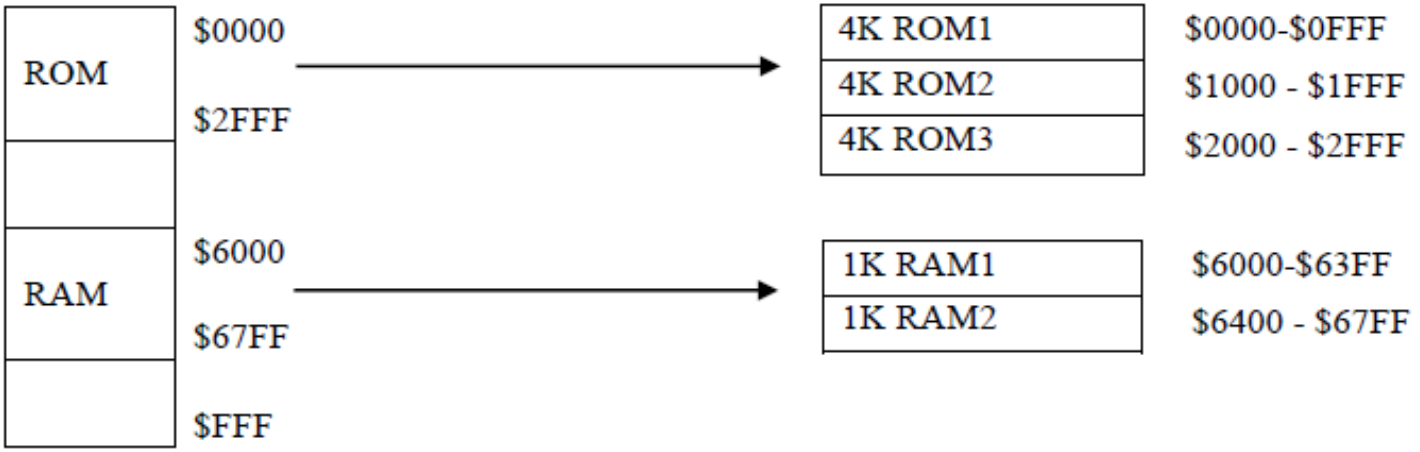
	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
From address 300H ➔	0	0	1	1	0	0	0	0	0	0	0	0
To address 3FFH ➔	0	0	1	1	1	1	1	1	1	1	1	1



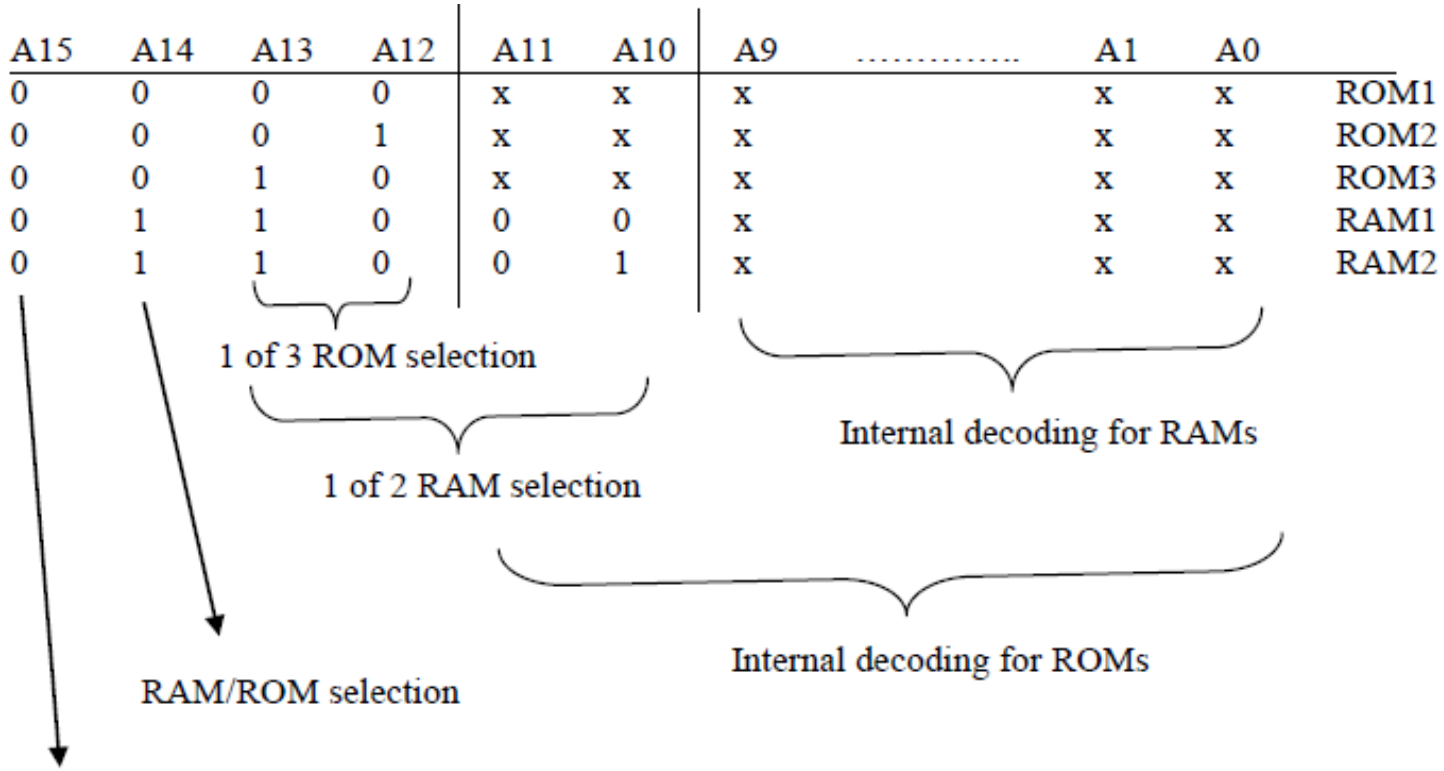
An easy way of designing



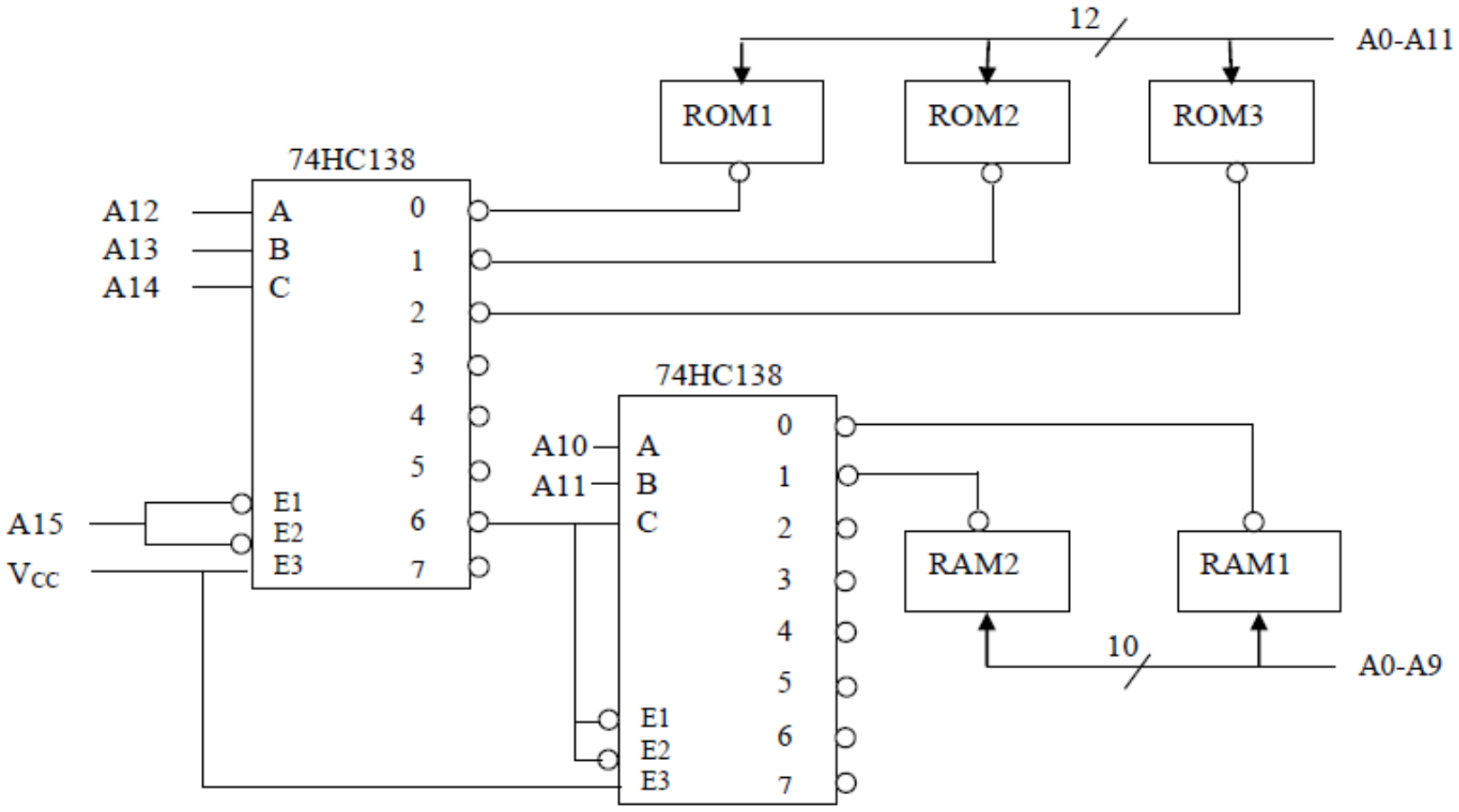
Example: The following memory map is given and it is required to be implemented using three 4K ROM and two 1K RAM chips available.



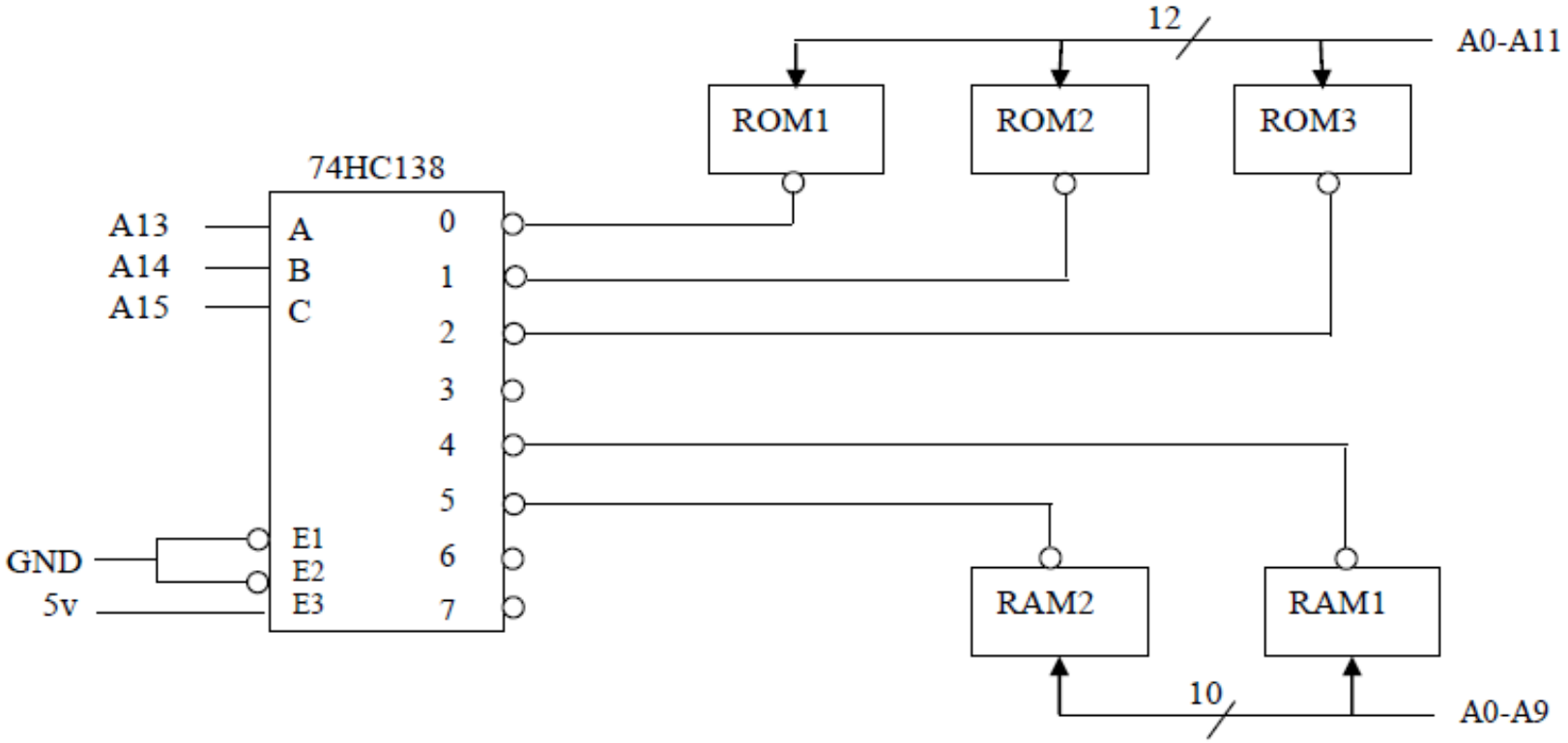
For RAM internal addressing we need 10 bits (A0-A9). A10-A15 are left for RAM selection.
For ROM internal addressing we need 12 bits (A0-A11). A12-A15 are left for ROM selection.


A15 A14 A13 A12 A11 A10 A9 ..... A1 A0
0 0 0 0 x x x x x ROM1
0 0 0 1 x x x x x ROM2
0 0 1 0 x x x x x ROM3
0 1 1 0 0 0 x x x RAM1
0 1 1 0 0 1 x x x RAM2
1 of 3 ROM selection
1 of 2 RAM selection
Internal decoding for RAMs
Internal decoding for ROMs
RAM/ROM selection
May be used for active enable input if a PROM decoder is used.

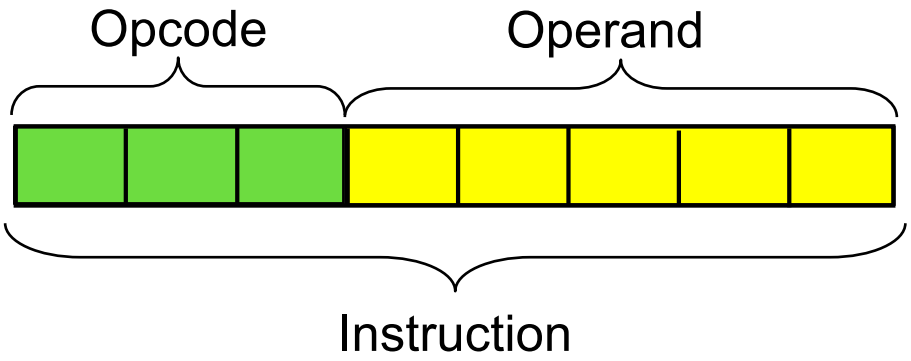
Using 74HC138 decoders

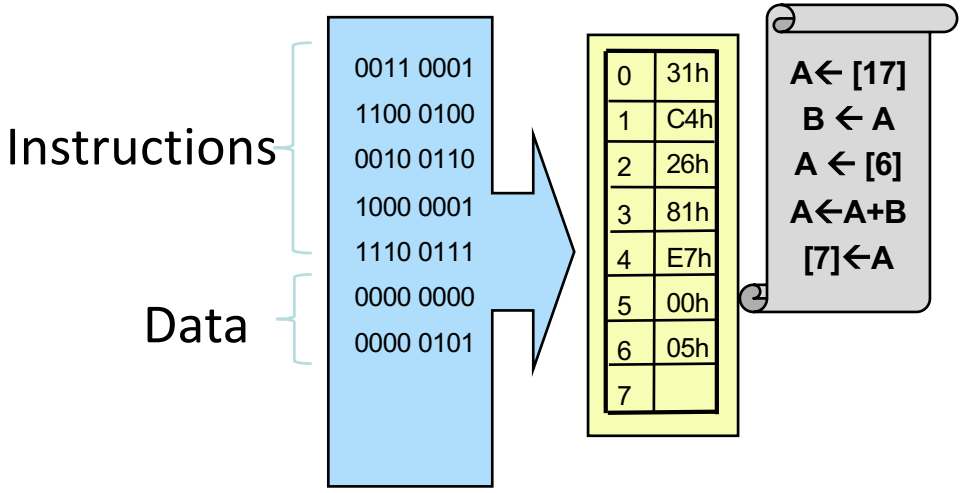


Memory overlaying



Another example for instruction execution:

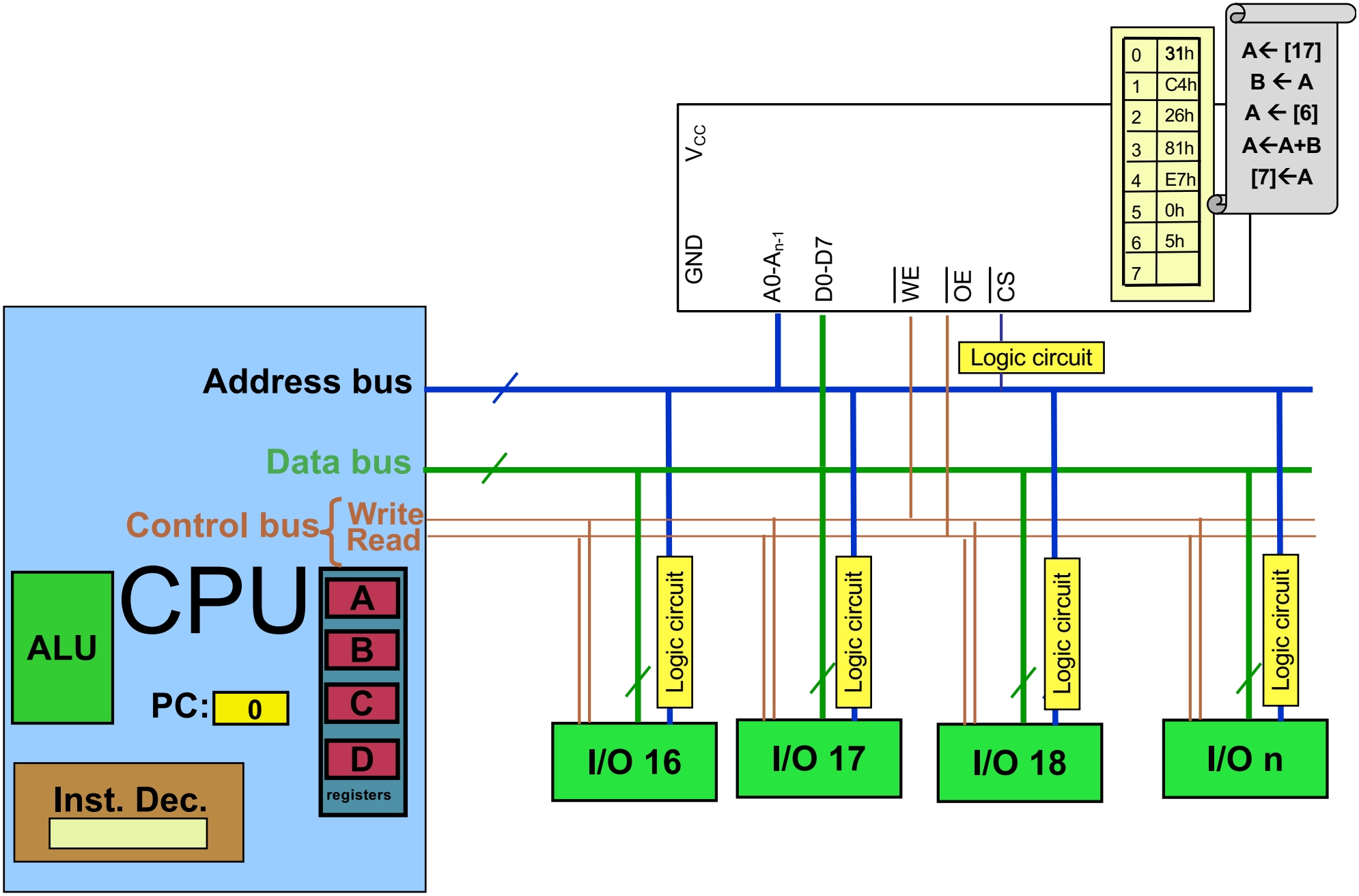




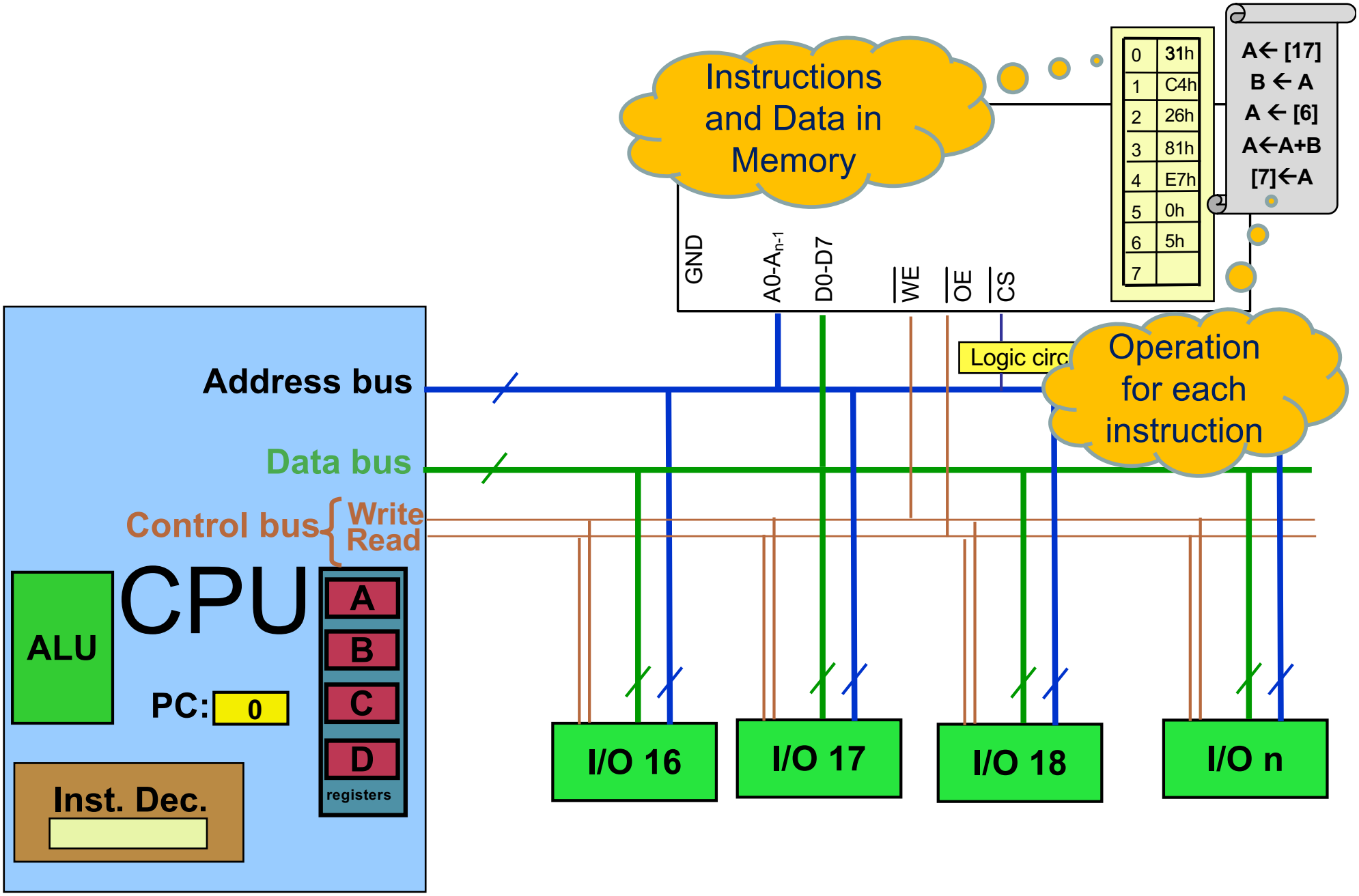
Operation Code	Meaning
000	A ← x
001	A ← [x]
010	A ← A – register (x)
011	A ← A + x
100	A ← A + register (x)
101	A ← A – x
110	Register (x_H) ← Register (x_L)
111	[x] ← A

Note: These are hypothetical instructions used for this example…

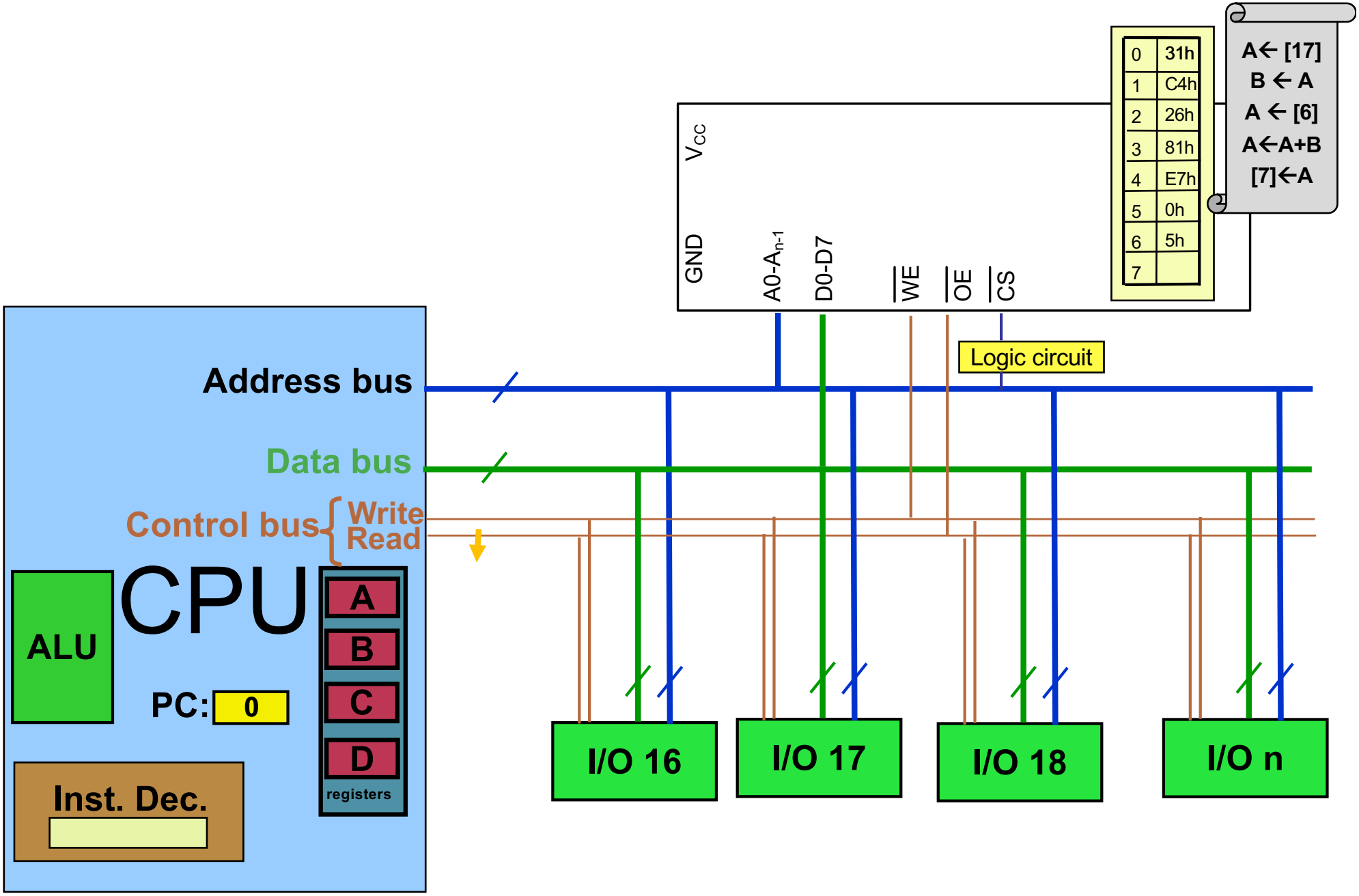
Instruction Fetch➔(Decode)➔Execute in CPU



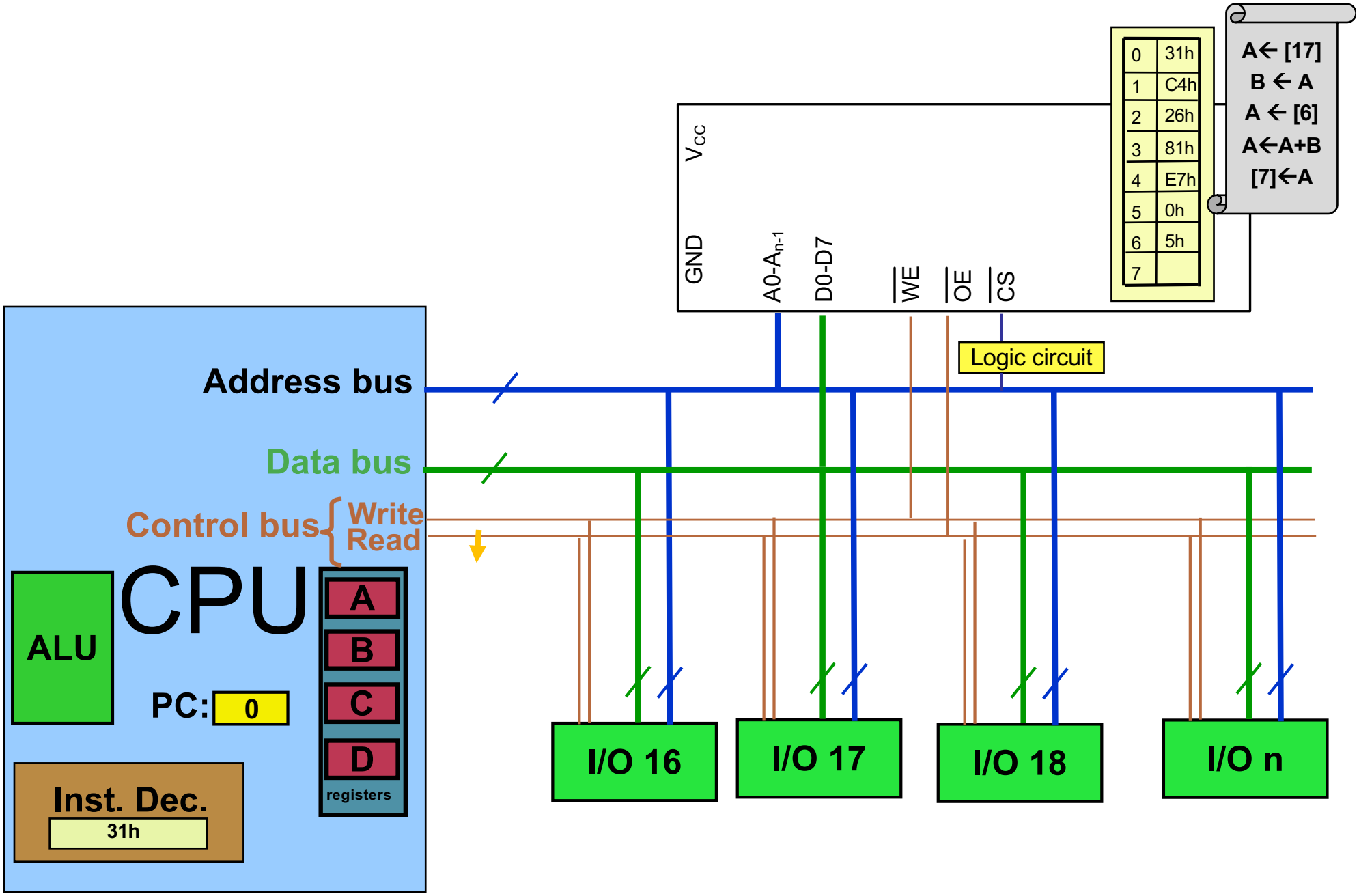
Instruction Fetch➔(Decode)➔Execute in CPU



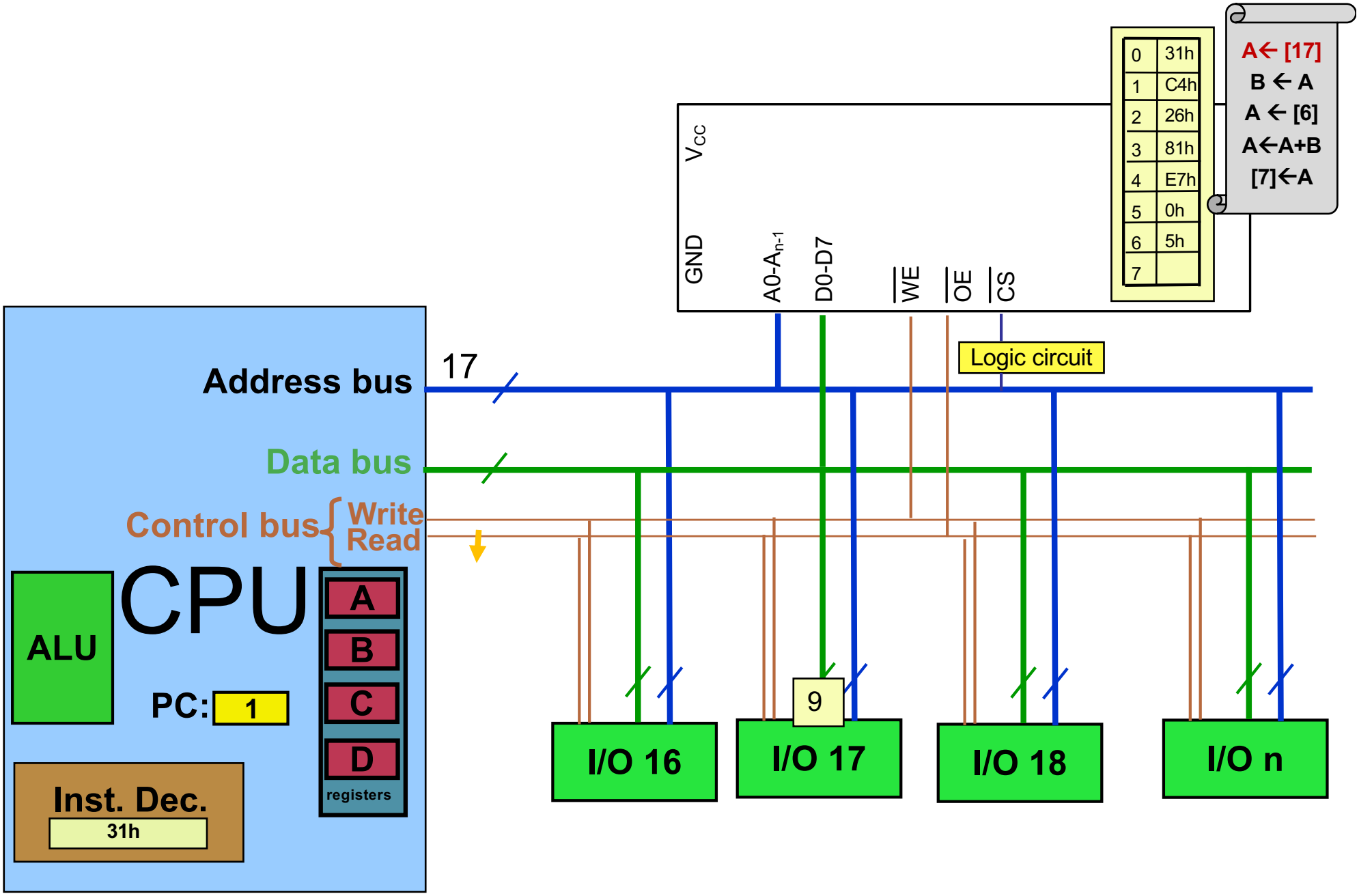
Instruction Fetch➔(Decode)➔Execute in CPU



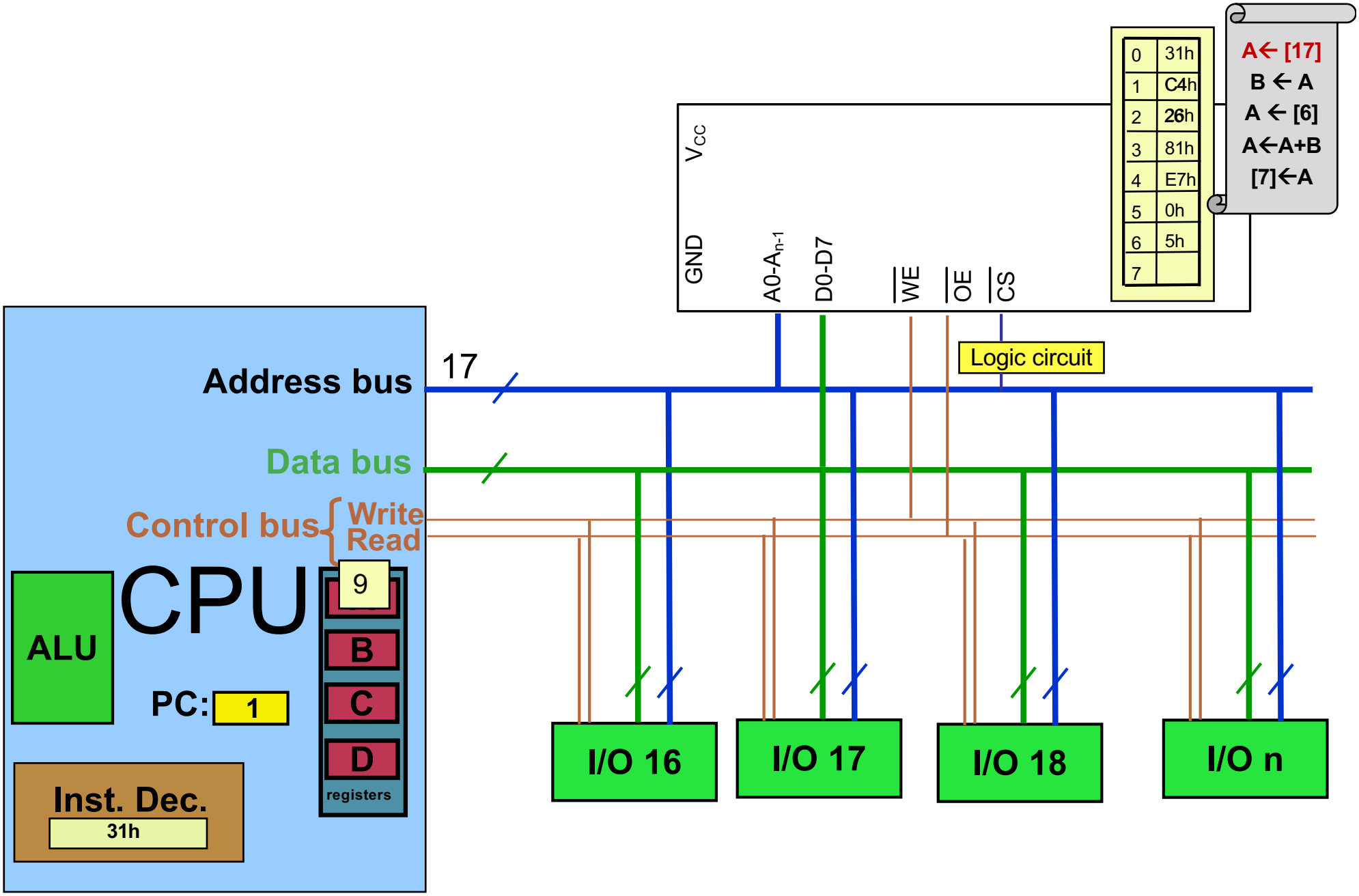
Instruction Fetch➔(Decode)➔Execute in CPU



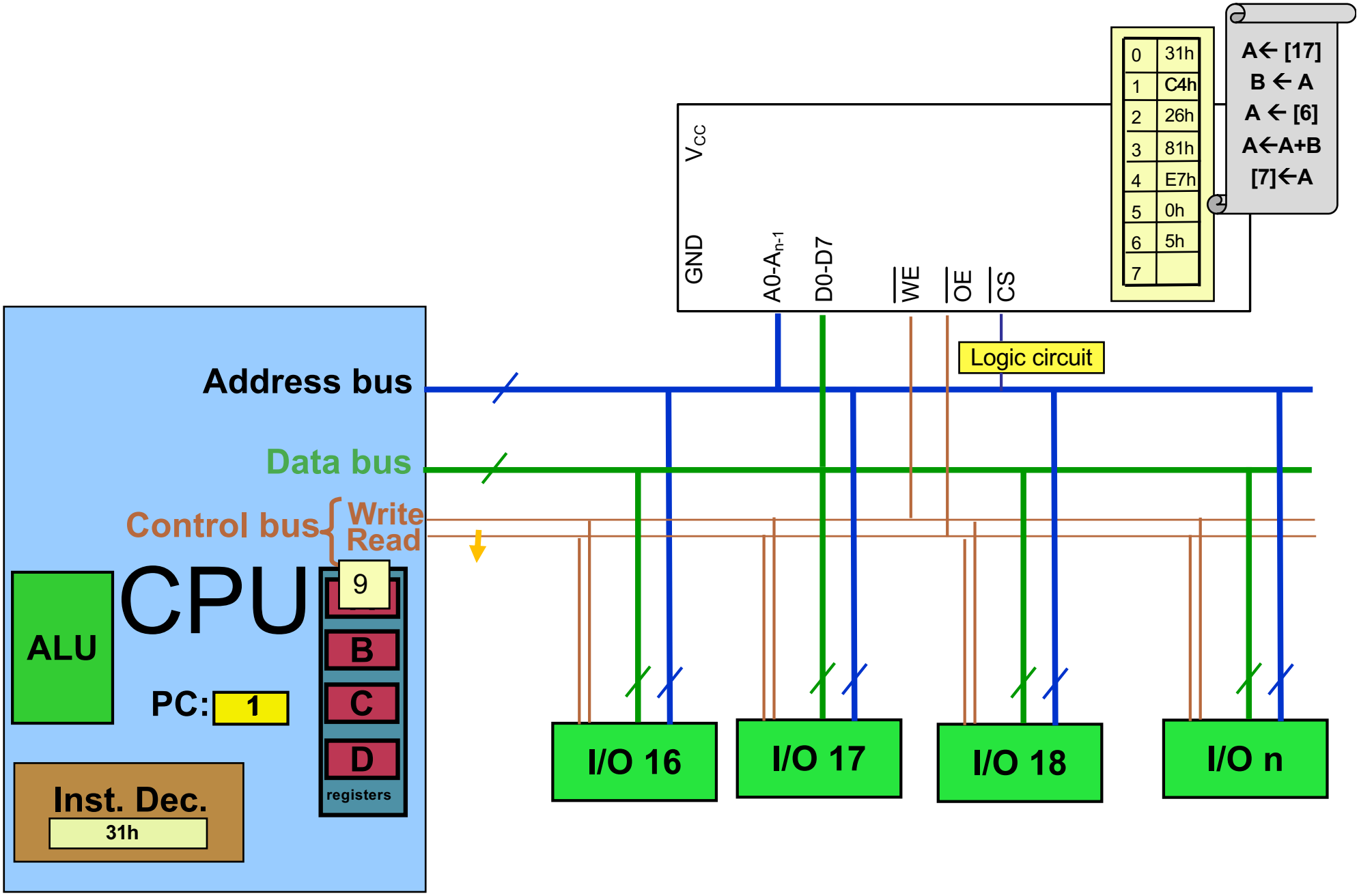
Instruction Fetch➔(Decode)➔Execute in CPU



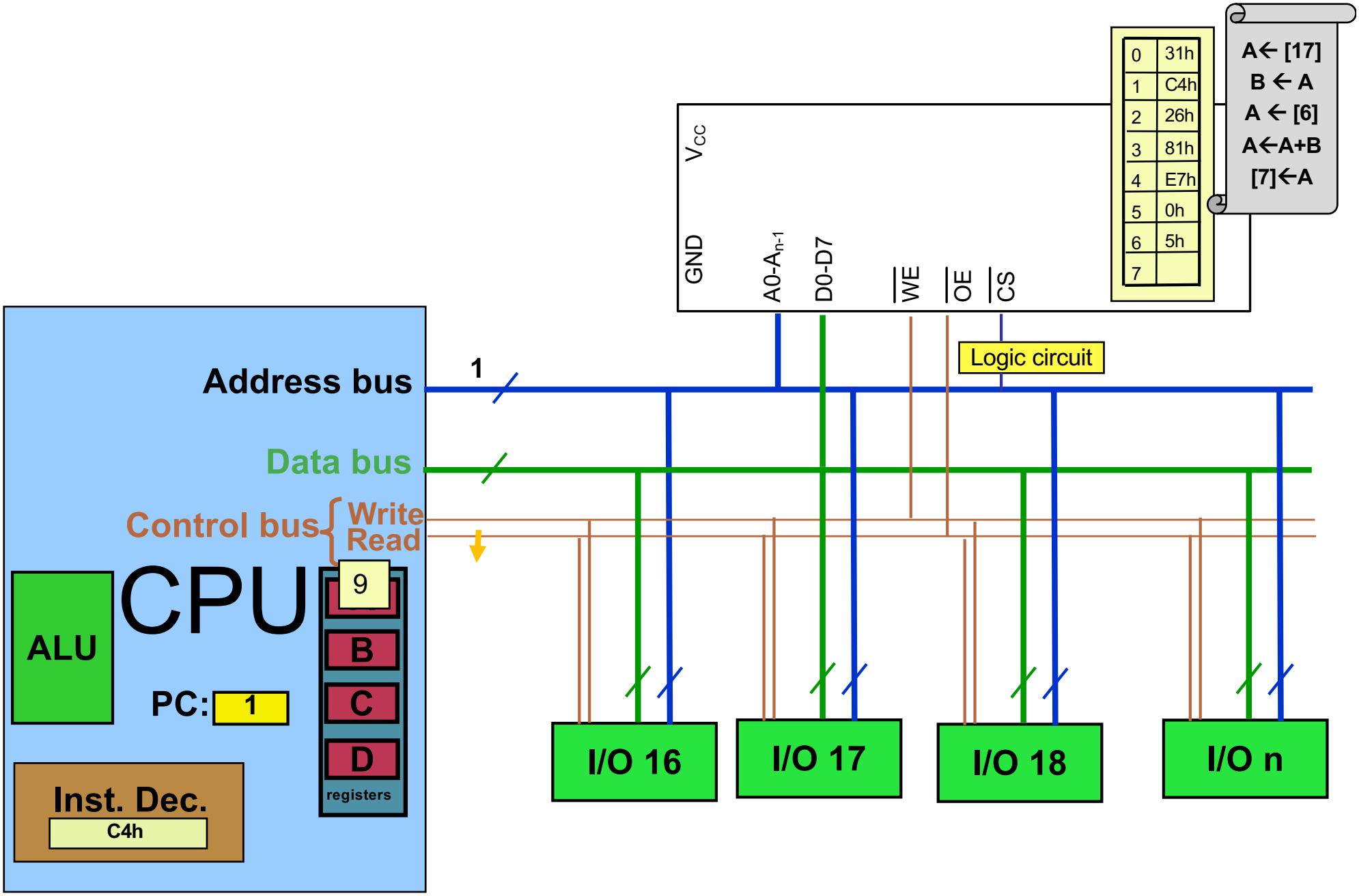
Instruction Fetch➔(Decode)➔Execute in CPU



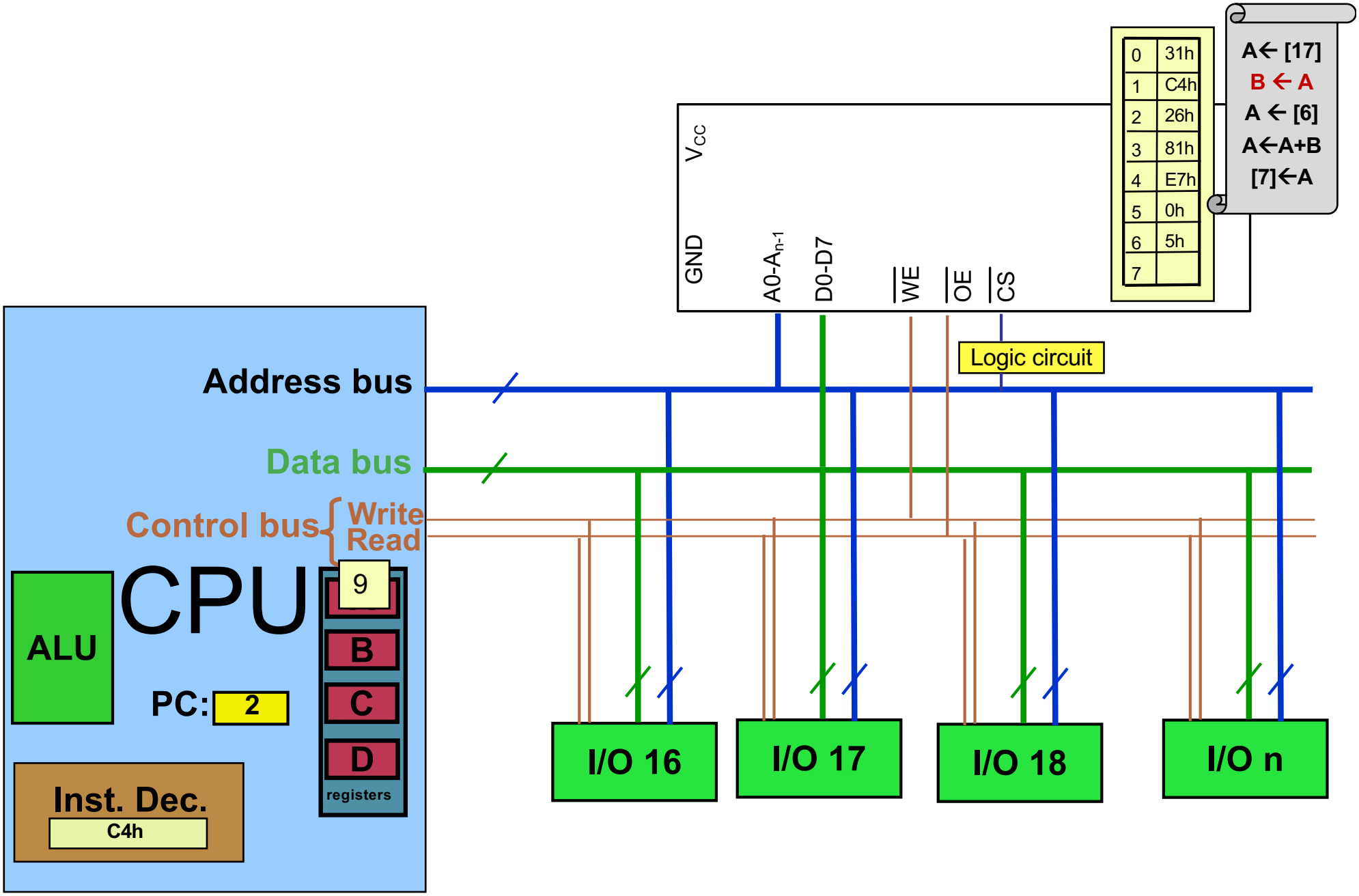
Instruction Fetch➔(Decode)➔Execute in CPU



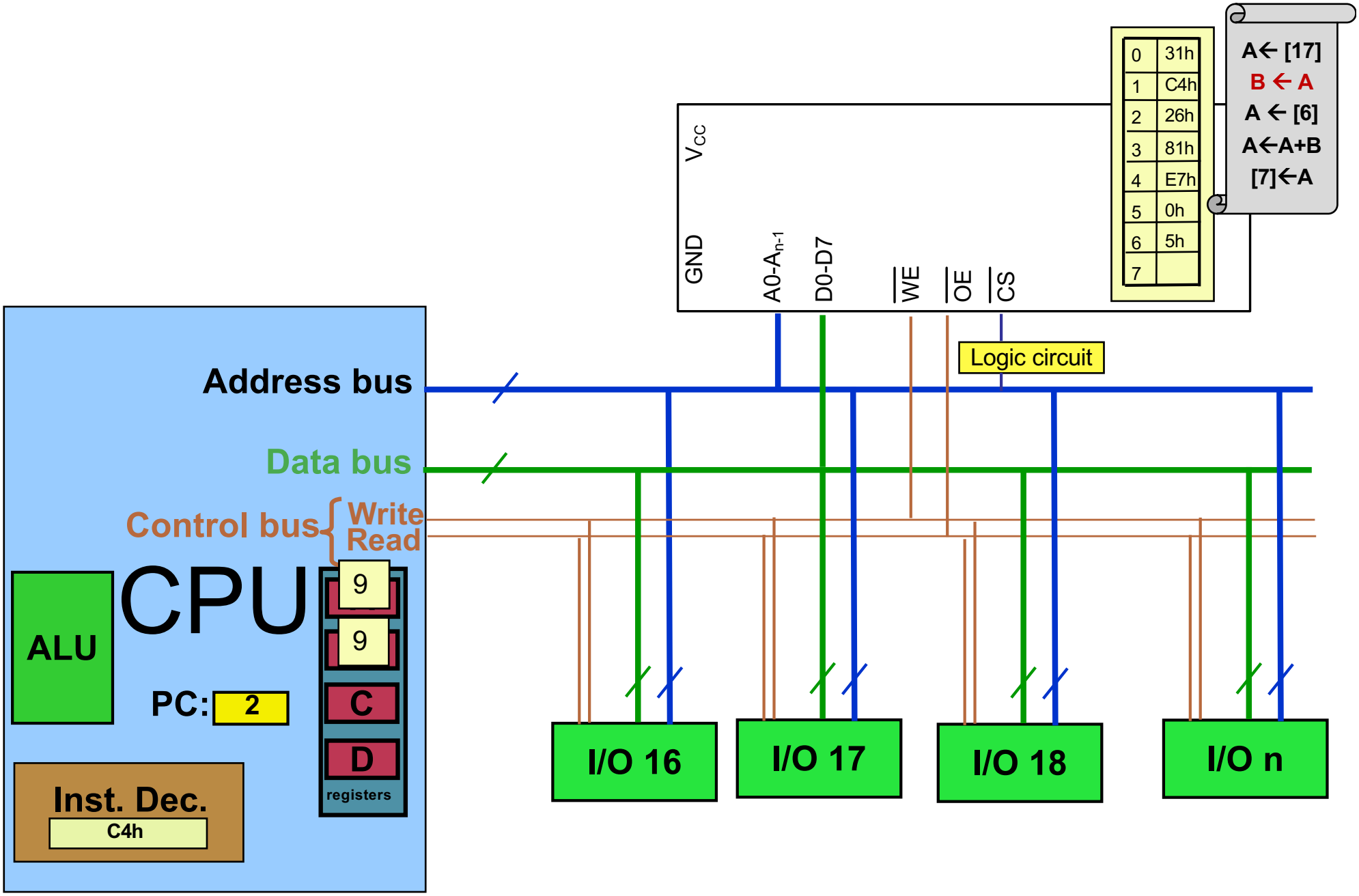
Instruction Fetch➔(Decode)➔Execute in CPU



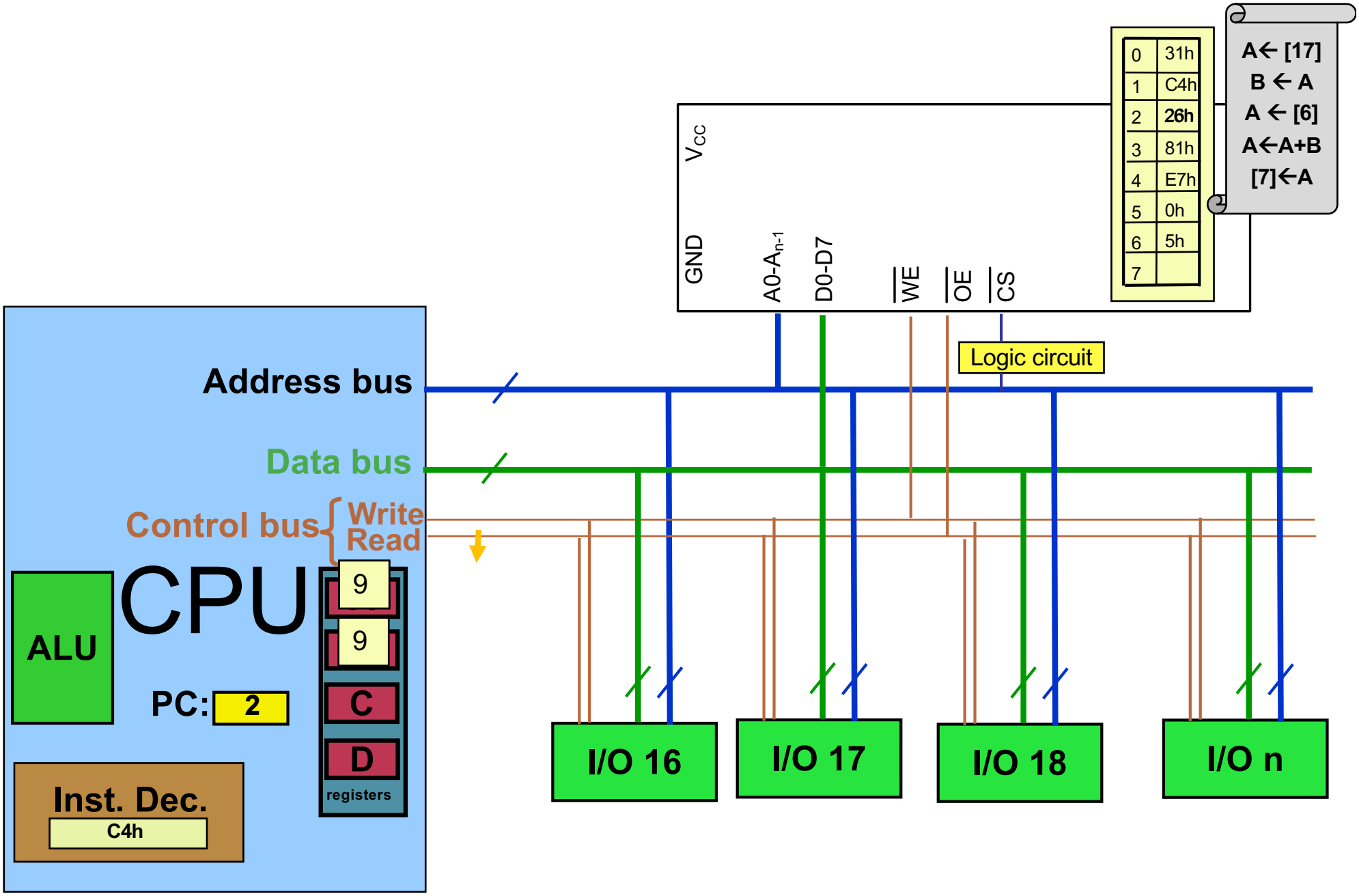
Instruction Fetch➔(Decode)➔Execute in CPU



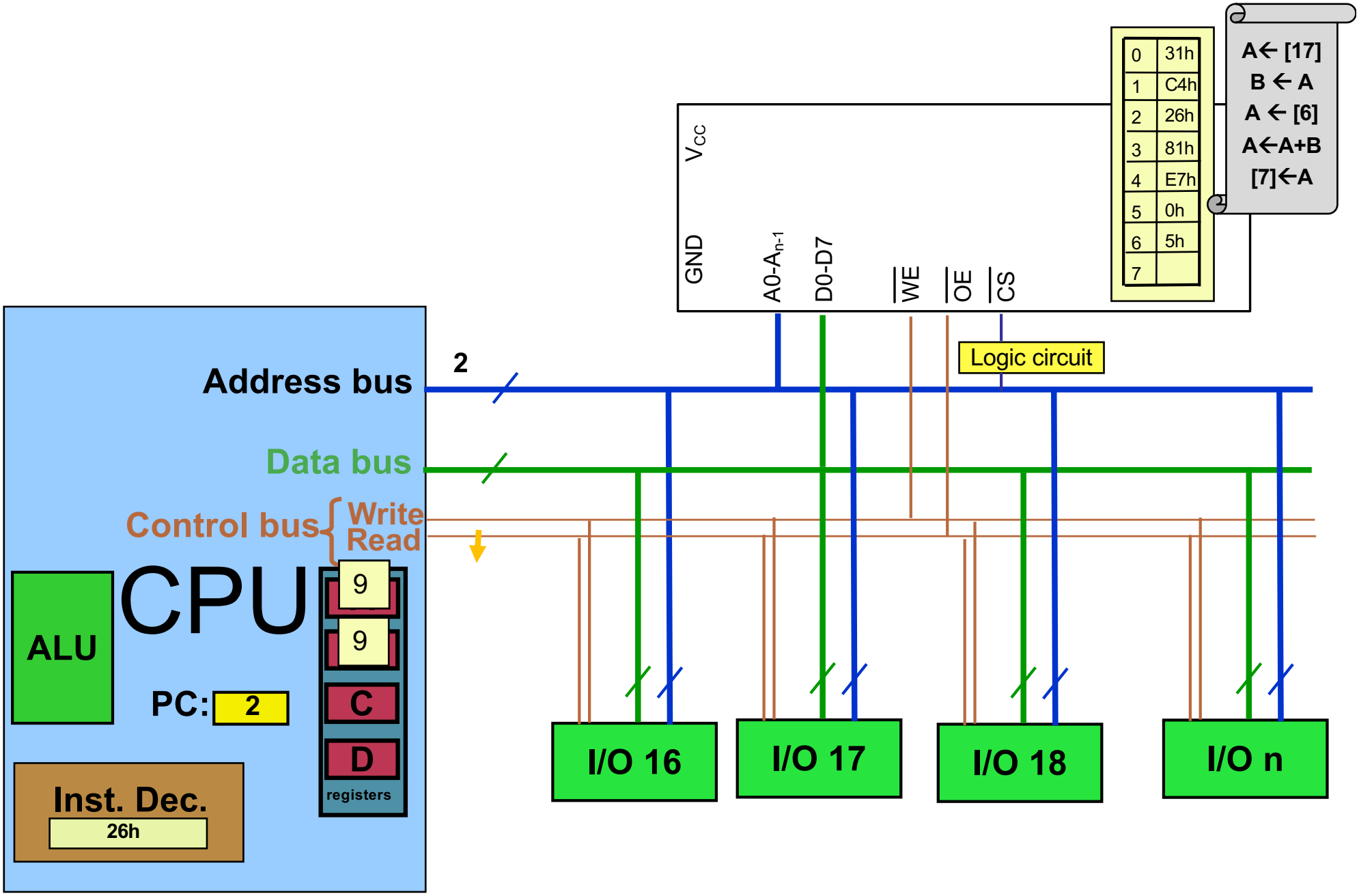
Instruction Fetch➔(Decode)➔Execute in CPU



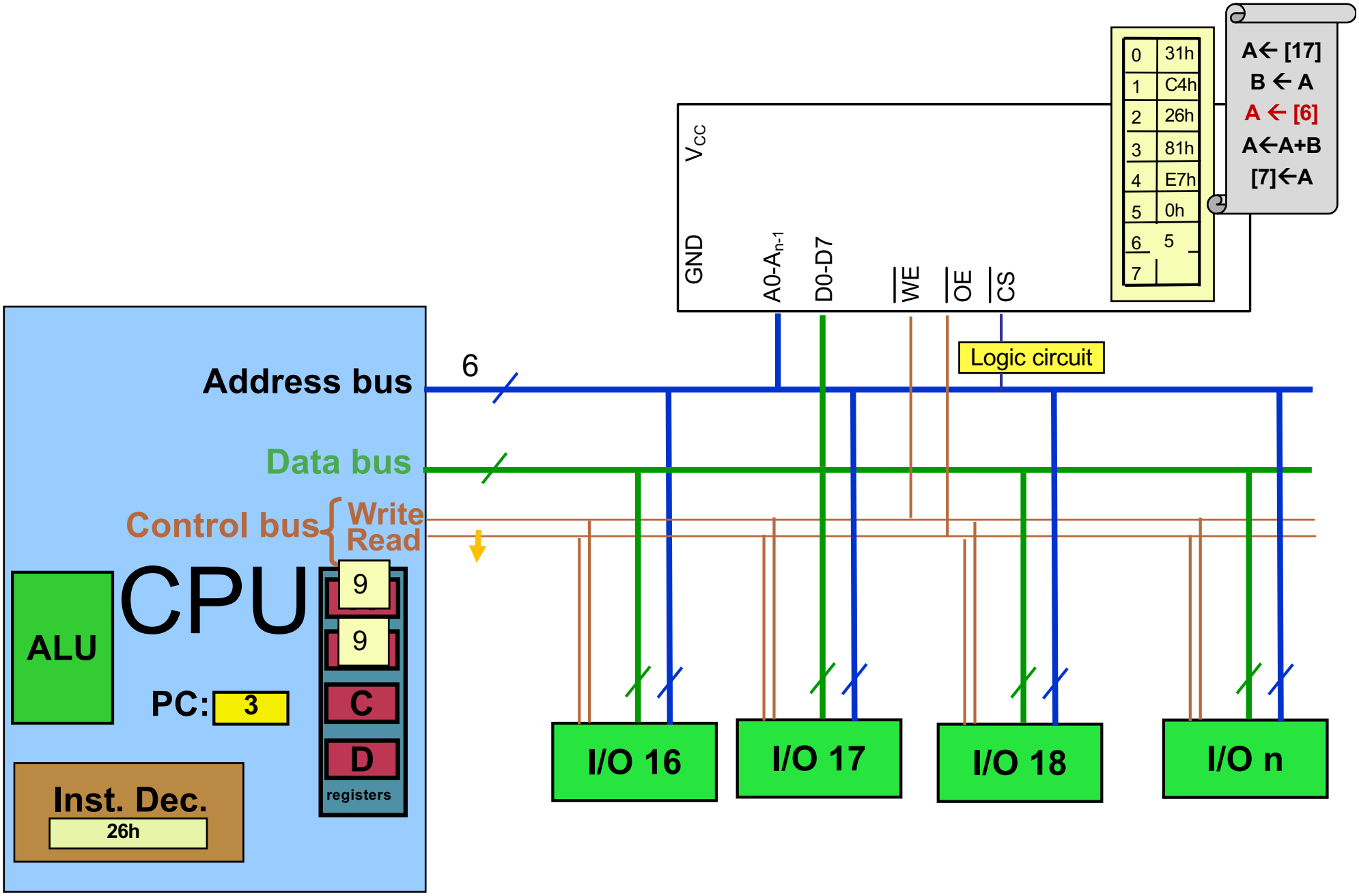
Instruction Fetch➔(Decode)➔Execute in CPU



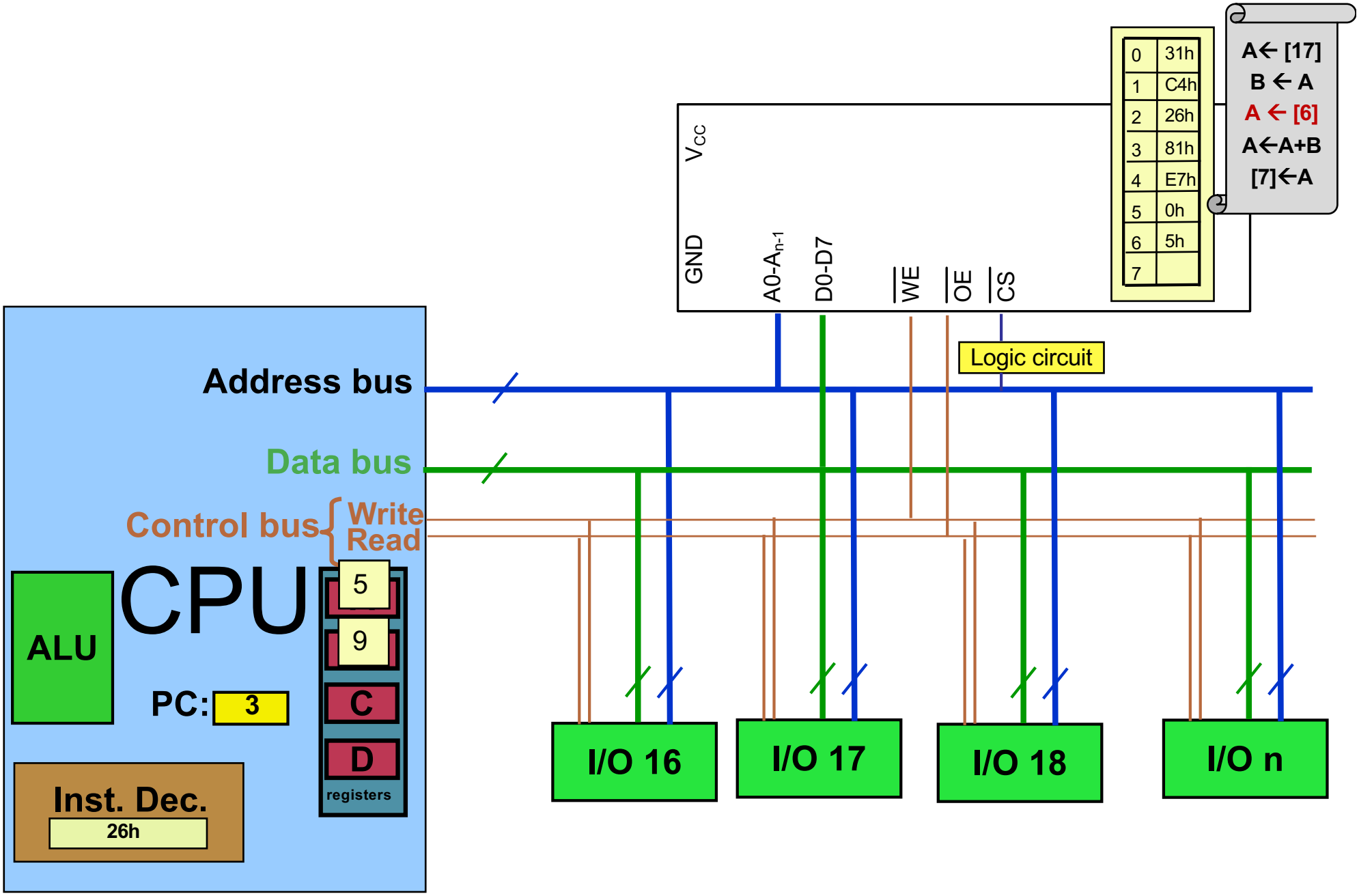
Instruction Fetch➔(Decode)➔Execute in CPU



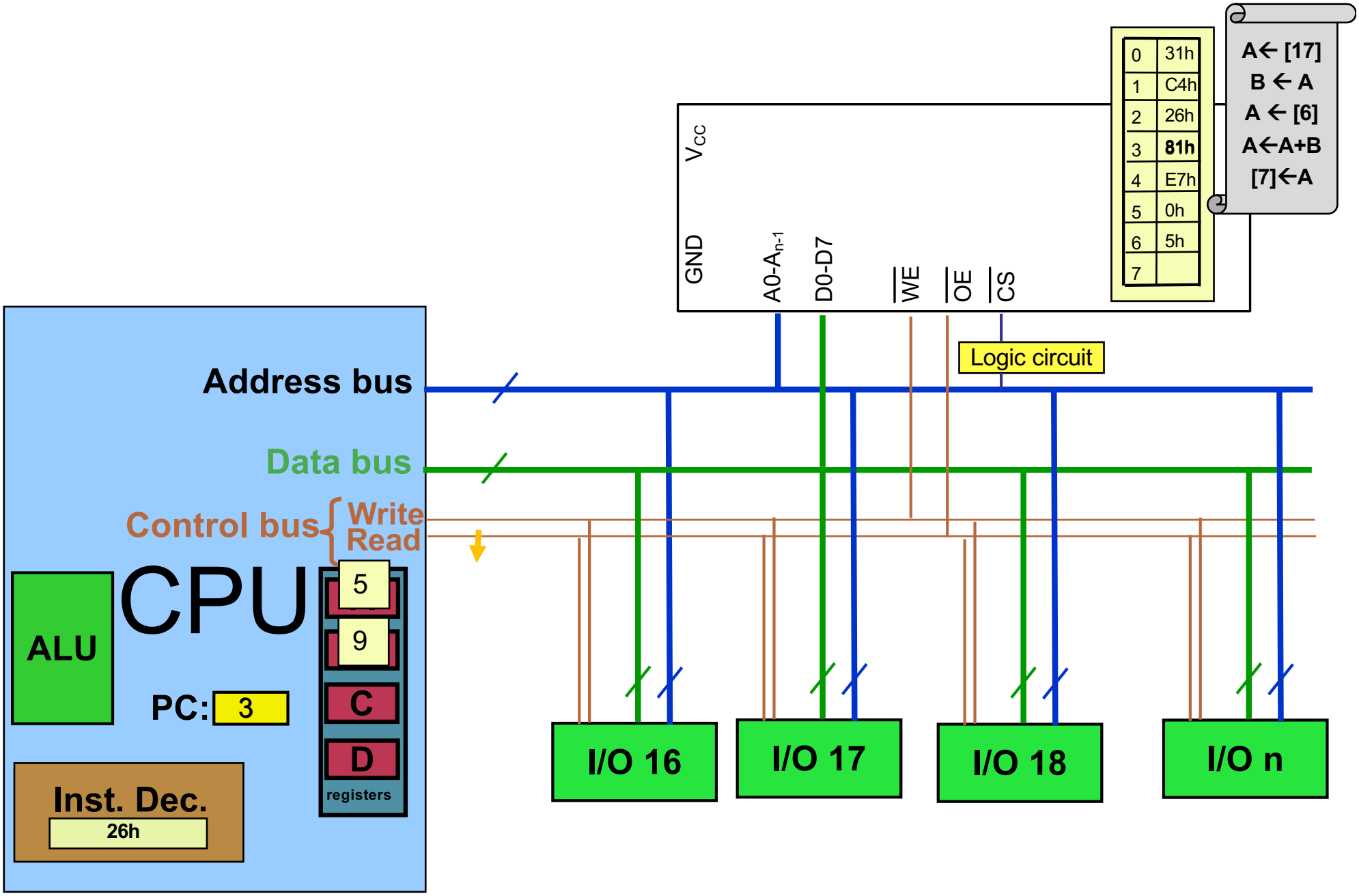
Instruction Fetch➔(Decode)➔Execute in CPU



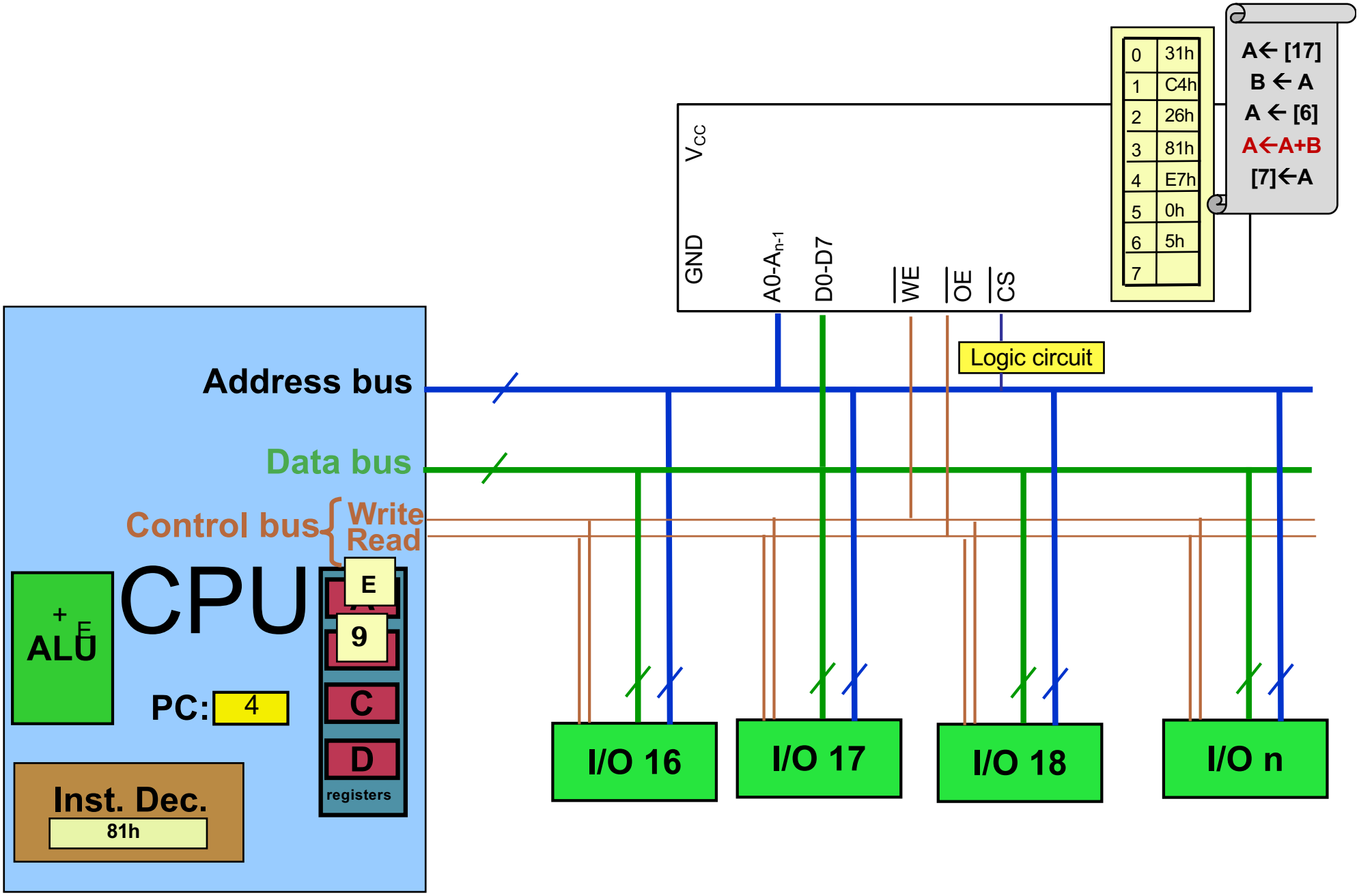
Instruction Fetch➔(Decode)➔Execute in CPU



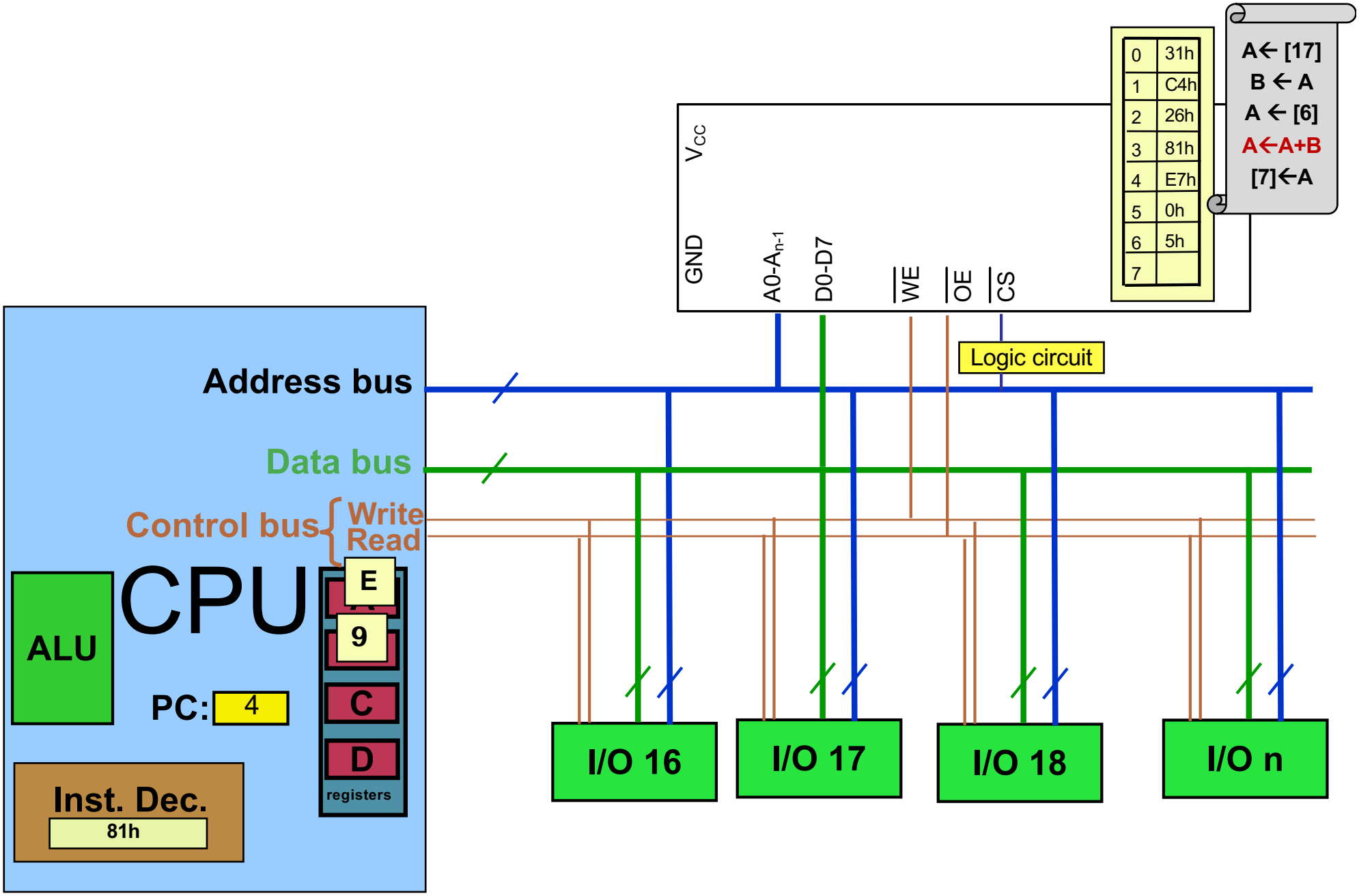
Instruction Fetch➔(Decode)➔Execute in CPU



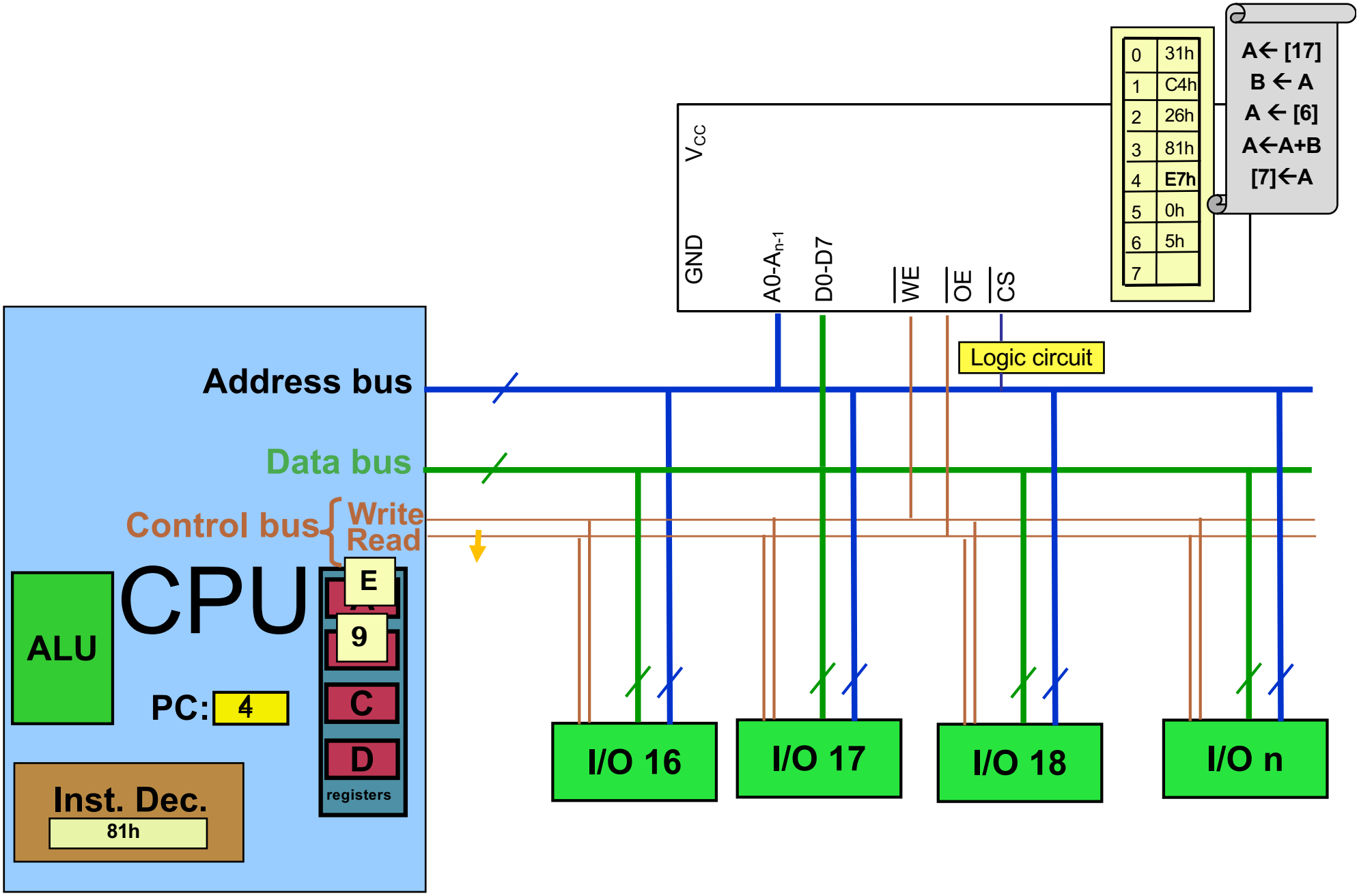
Instruction Fetch➔(Decode)➔Execute in CPU



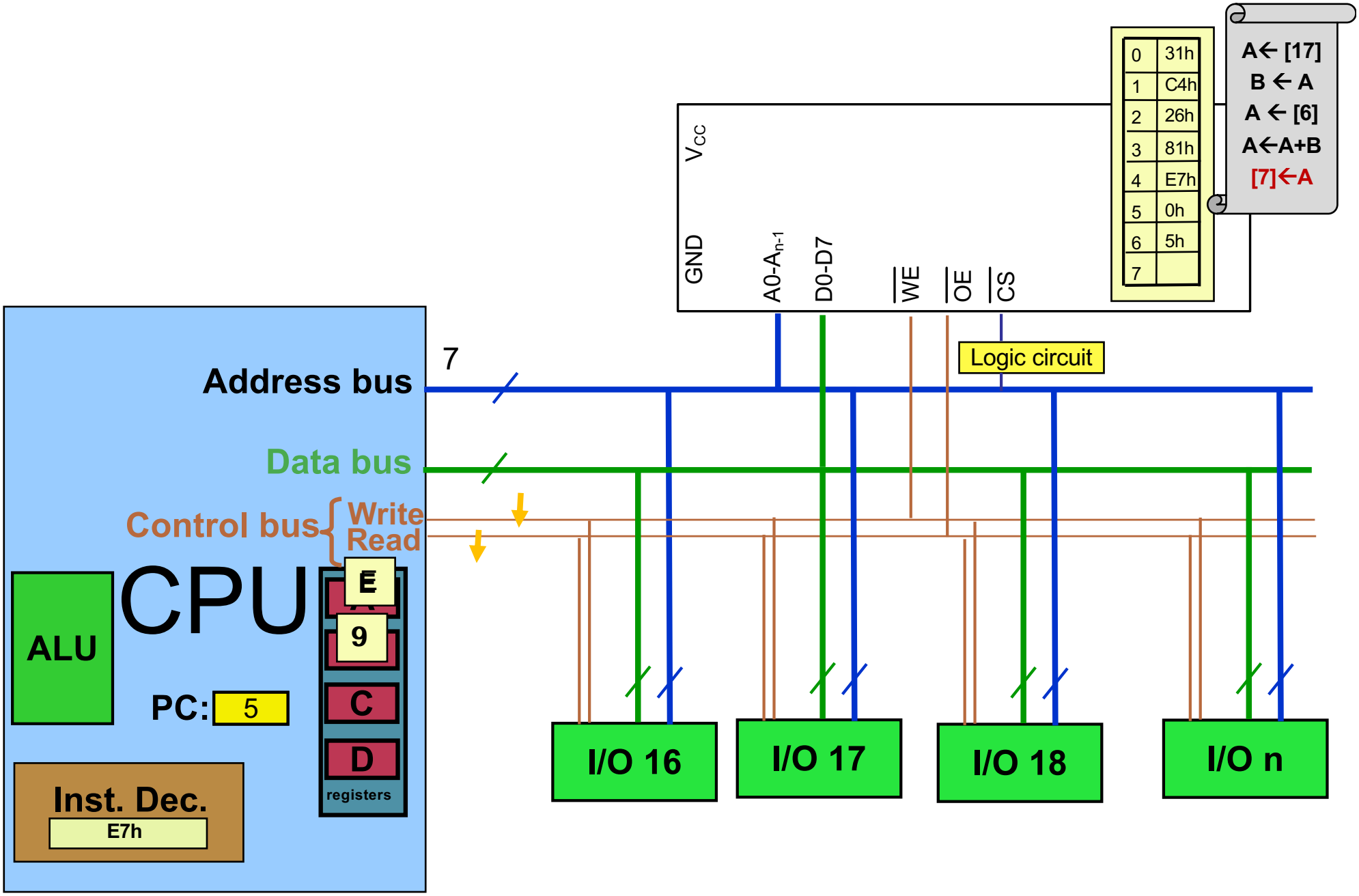
Instruction Fetch➔(Decode)➔Execute in CPU



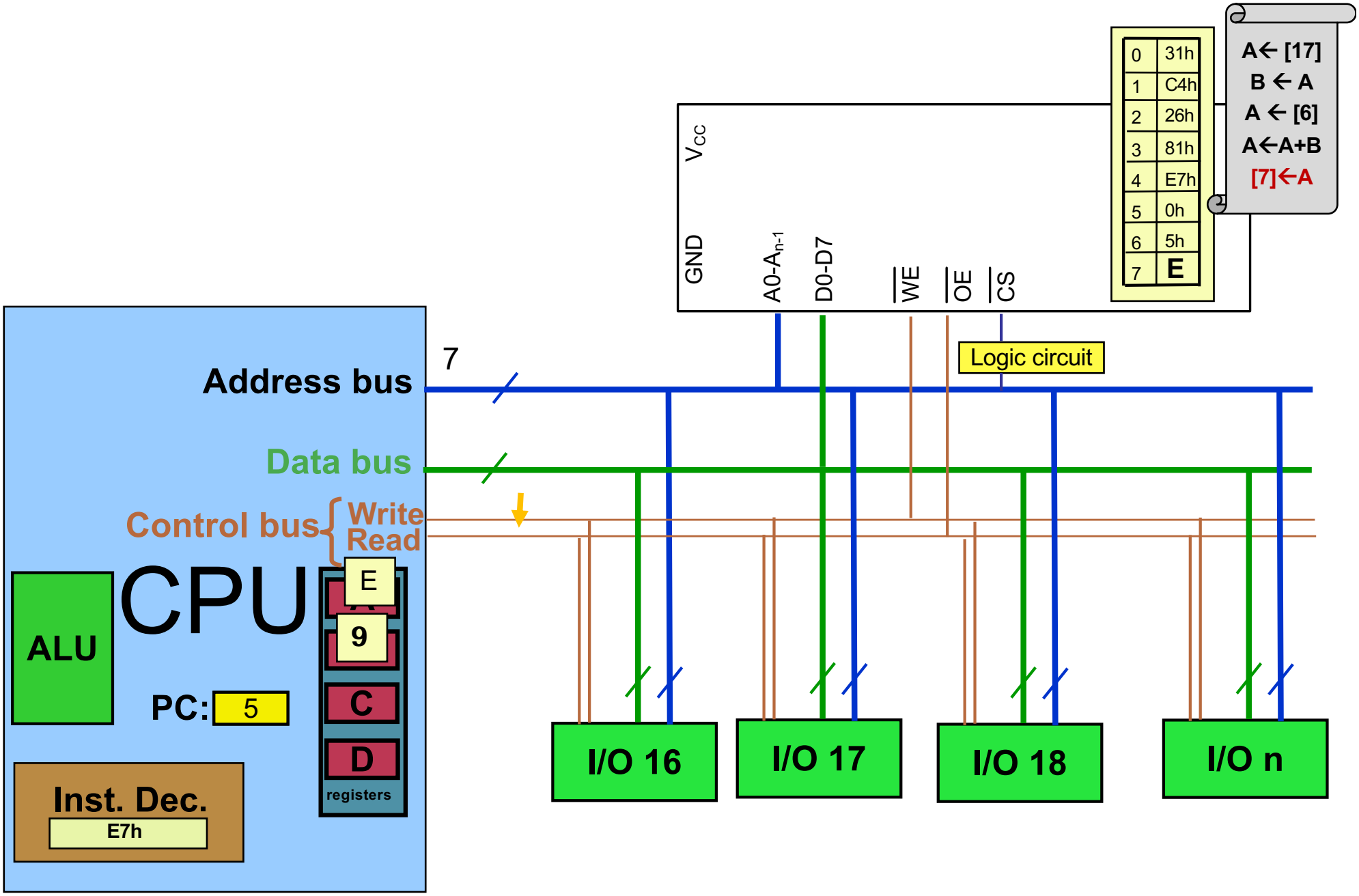
Instruction Fetch➔(Decode)➔Execute in CPU



Instruction Fetch➔(Decode)➔Execute in CPU



Instruction Fetch➔(Decode)➔Execute in CPU



Instruction Set architecture
RISC vs. CISC

- CISC (Complex Instruction Set Computer)
  - Put as many instruction as you can into the CPU

- RISC (Reduced Instruction Set Computer)
  - Reduce the number of instructions, and use your facilities in a more proper way.

CISC

Features of the Complex Instruction Set Computing (CISC):

- many instructions
- complex instructions
  - each instruction can execute several low level operations
- complex addressing modes
  - smaller number of registers needed

A semantically rich instruction set is accommodated by allowing instructions of variable length

RISC

RISC –Reduced Instruction Set Computing

- Small instruction sets
- Simpler instructions
- Fixed length instructions
- Large number of registers
- Simpler addressing mode with the Load/Store instructions for accessing memory

Adv/Disadv of CISC

ADV

- As each instruction can execute several low level operations,
  - the code size is reduced to save on memory requirements
  - less main memory access is required and hence processing time is reduced (faster)
- Backward code compatibility is maintained
  - can add new (and more powerful) instructions while retaining the ‘old’ instruction set for code compatibility (i.e. legacy programs can still run)
- Easy to program
  - complex instructions that fit well with high-level language expressions

DISADV

- A highly encoded instruction set needs to be decoded by hardwired microcode electronic circuitry
  - more complex hardware design
  - slower instruction decoding/execution
- Variable length instructions
  - different execution time among instructions
  - affects pipelined operations

Adv/Disadv of RISC

ADV

- Simpler instructions
  - one clock per instruction gives faster execution than on a CISC processor with the same clock speed
- Simpler addressing mode
  - faster decoding
- Fixed length instructions
  - faster decoding and better pipeline performance
- Simpler hardware
  - less silicon area
  - less power consumption

M4 adds FPU and DSP-like instructions

DISADV

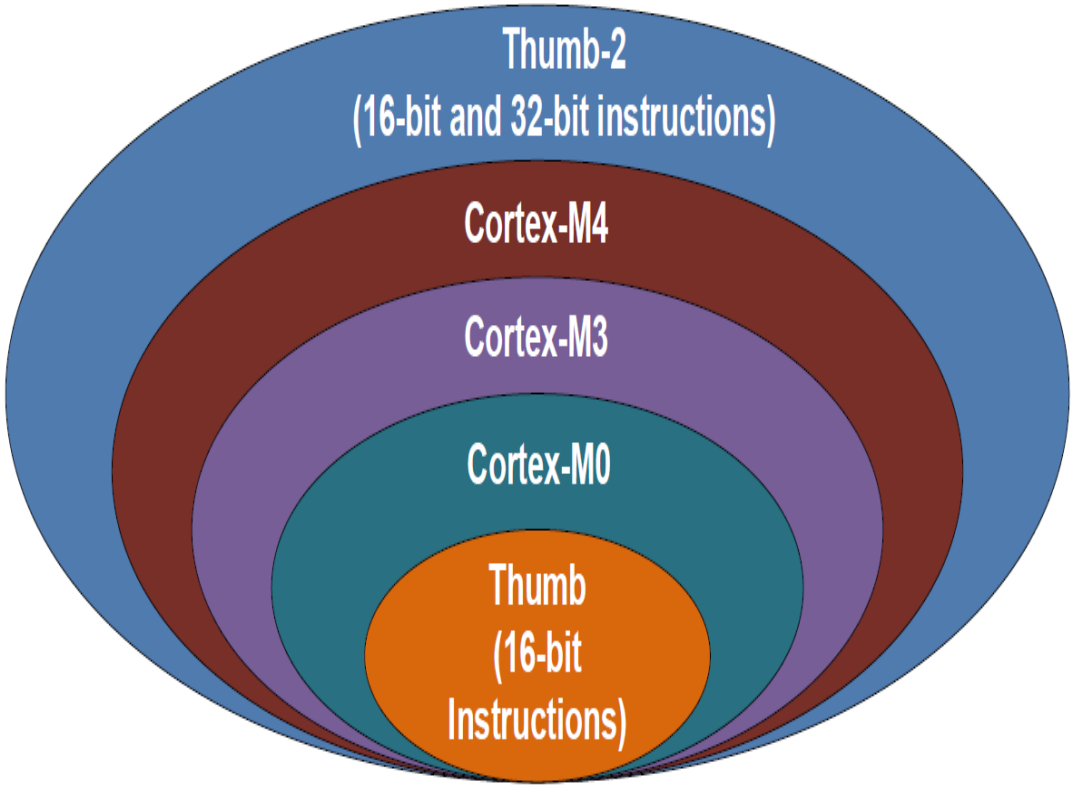
- Fewer instructions than CISC
  - as compared to CISC, RISC needs more instructions to execute one task
  - code density is less
  - needs more memory
- No complex instructions
  - no hardware support for division or floating-point arithmetic operations
  - needs a more complex compiler and longer compiling time

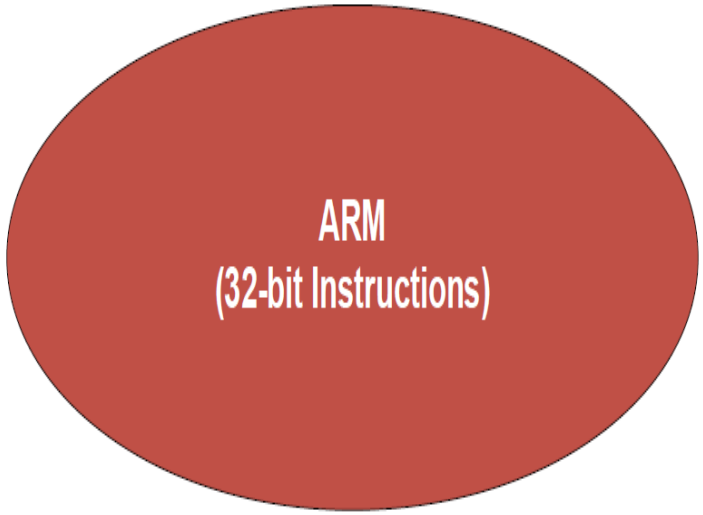
To reduce memory requirements and cost ARM provides the 16-bit Thumb instruction set and 16/32-bir Thumb2 instruction set as an option for its RISC processor cores

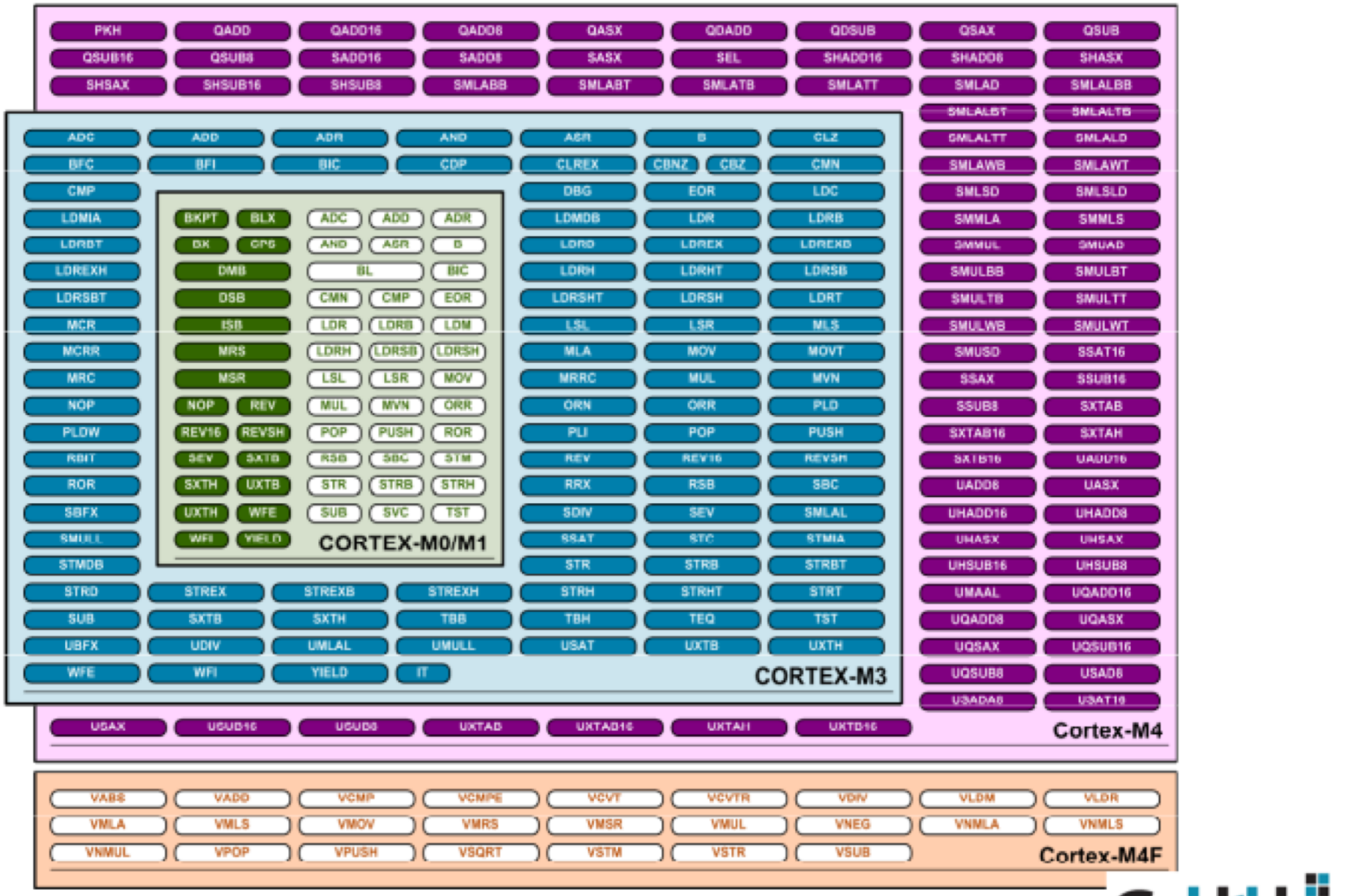
Cortex-M4 Processor Features

- 32-bit Reduced Instruction Set Computing (RISC) processor
  - fixed instruction size.
    - It makes the task of instruction decoder easier.
    - In Cortex M4 the instructions are 2 or 4 bytes.
  - reduced number of *simple* instructions
    - Pros: Reduces the number of used transistors
    - Cons: Can make the assembly programming more difficult , Can lead to using more memory
  - Limited number of addressing modes
    - Advantage: Hardwiring from instruction register to datapath
    - Disadvantage: Can make the assembly programming more difficult

Instruction Sets of ARM

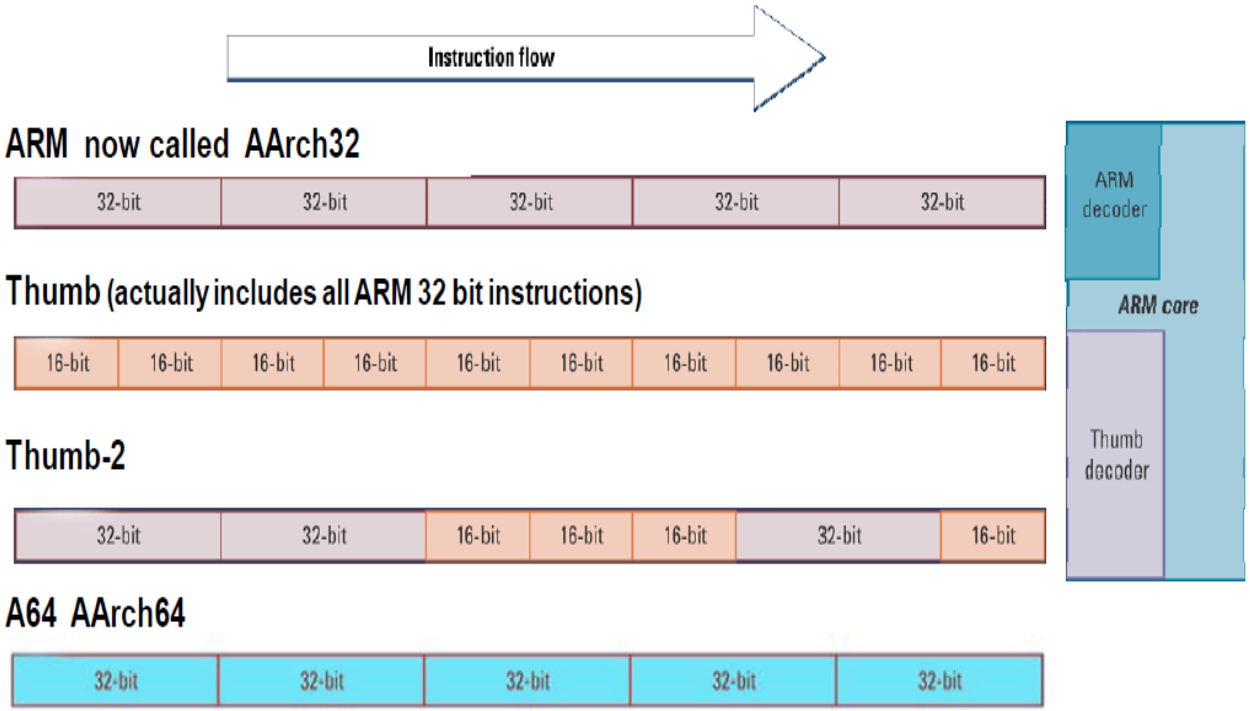


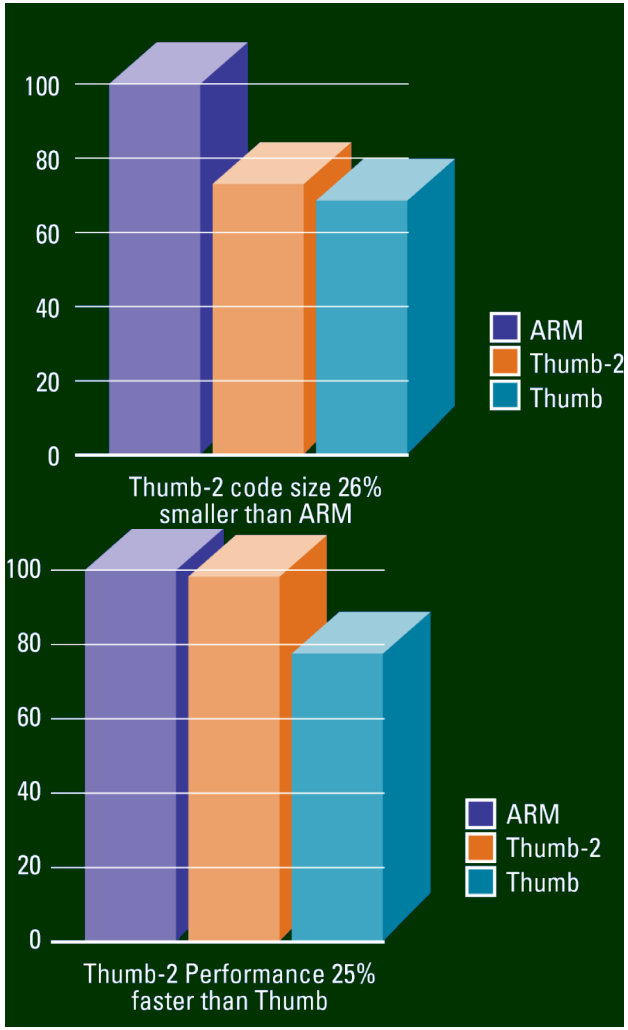



PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX QSUB
QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16 SHADD8 SHASX
SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB SMLATT SMLAD SMLALBB
SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
SMUSD SSAT16
SSAX SSUB16
SSUB8 SXTAB
SXTAB16 SXTAH
SXTB16 UADD16
UADD8 UASX
UHADD16 UHADD8
UHASX UHSAX
UHSUB16 UHSUB8
UMAAL UQADD16
UQADD8 UQASX
UQSAX UQSUB16
UQSUB8 USAD8
USADA8 USAT16
ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB
LDRDT LDRD LDREX LDREXB
LDREXH LDRH LDRHT LDRSB
LDRSBT LDRSHT LDRSH LDRT
MCR LSL LSR MLS
MCRR MLA MOV MOVT
MRC MRRC MUL MVN
NOP ORN ORR PLD
PLDW PLI POP PUSH
RBIT REV REV16 REVSH
ROR RRX RSB SBC
SBFX SDIV SEV SMLAL
SMULL SSAT STC STMIA
STMDB STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT
BKPT BLX ADC ADD ADR
DK DPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD
CORTEX-M0/M1
CORTEX-M3
USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16
Cortex-M4
VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM VLDR
VMLA VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB
Cortex-M4F
Cortex
Low-Power Leadership from ARM

Cortex-M4

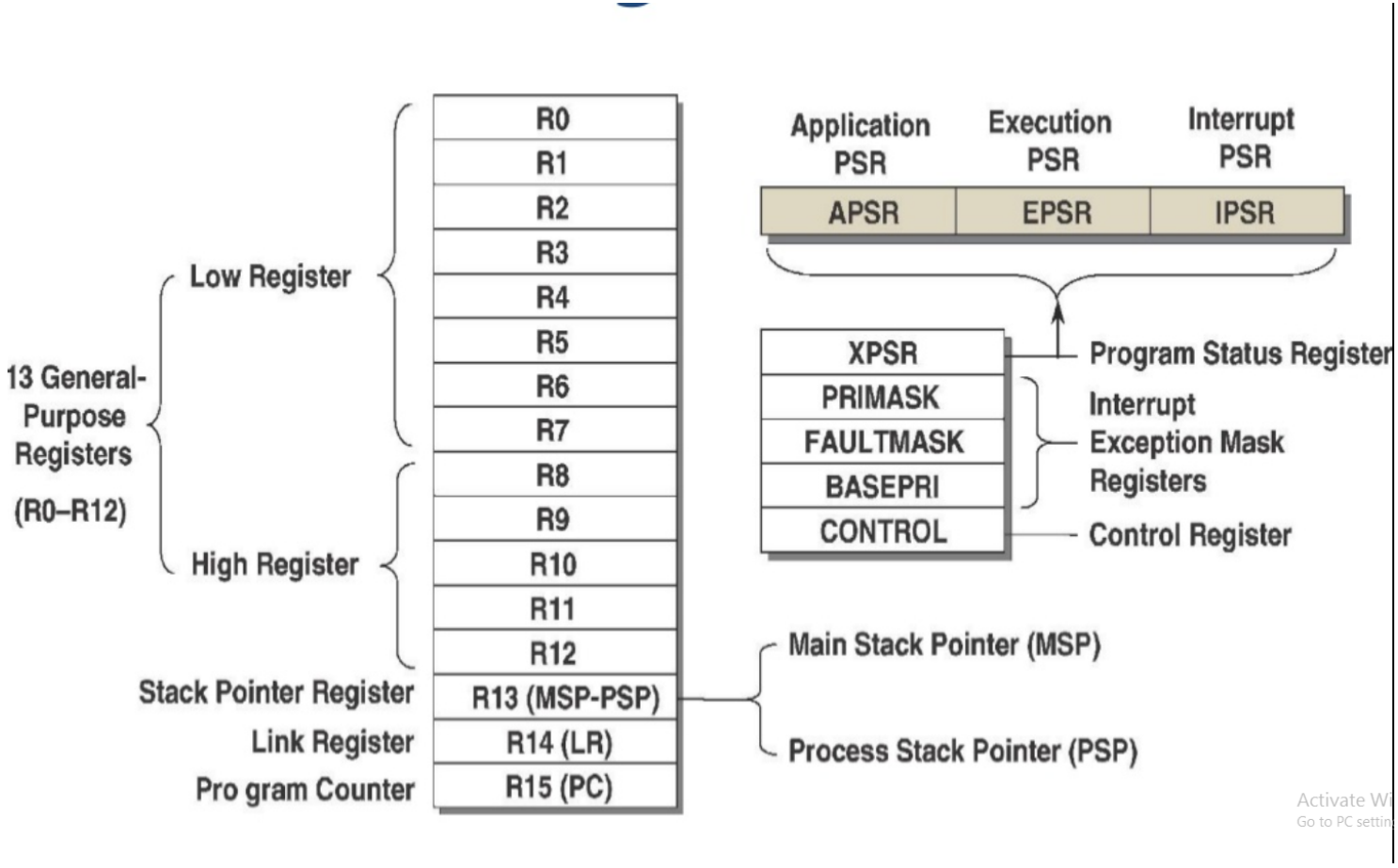
- Instruction set
  - Include the entire Thumb®-1 (16-bit) and Thumb®-2 (16/ 32-bit) instruction sets



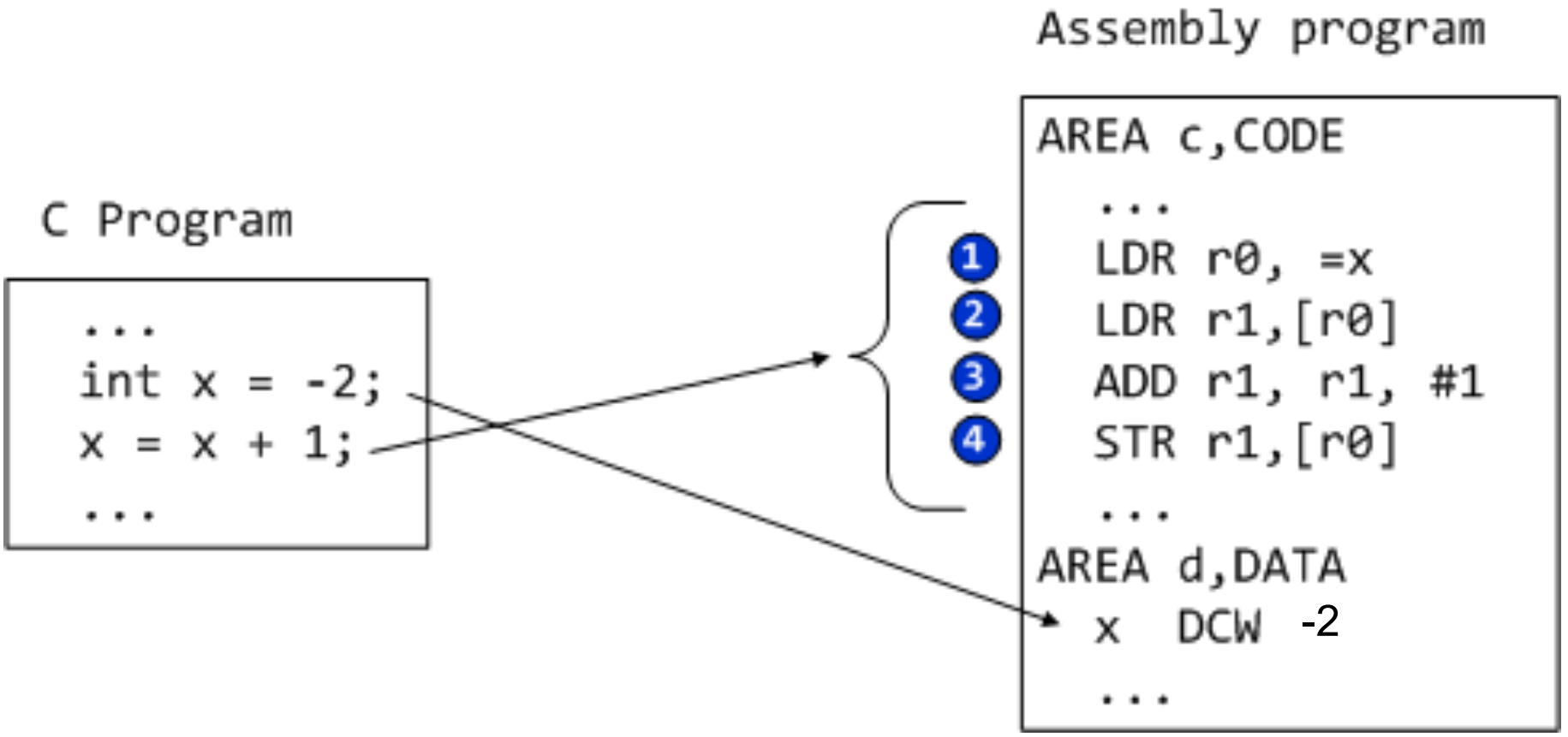


Cortex-M4 Processor Registers

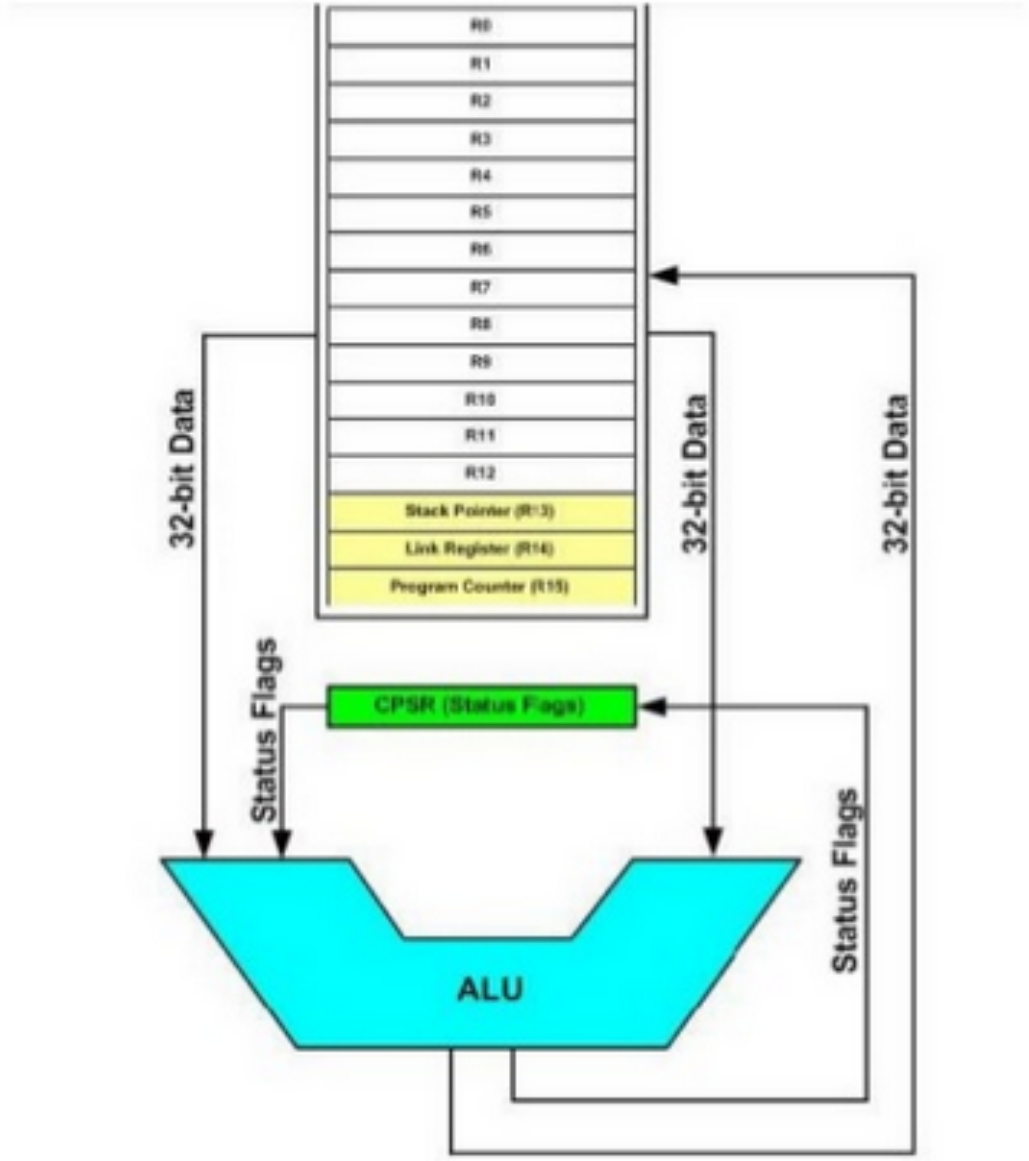
- High user accessible register count
  - RISC processors have at least 32 registers.
  - Decreases the need for stack and memory usages.



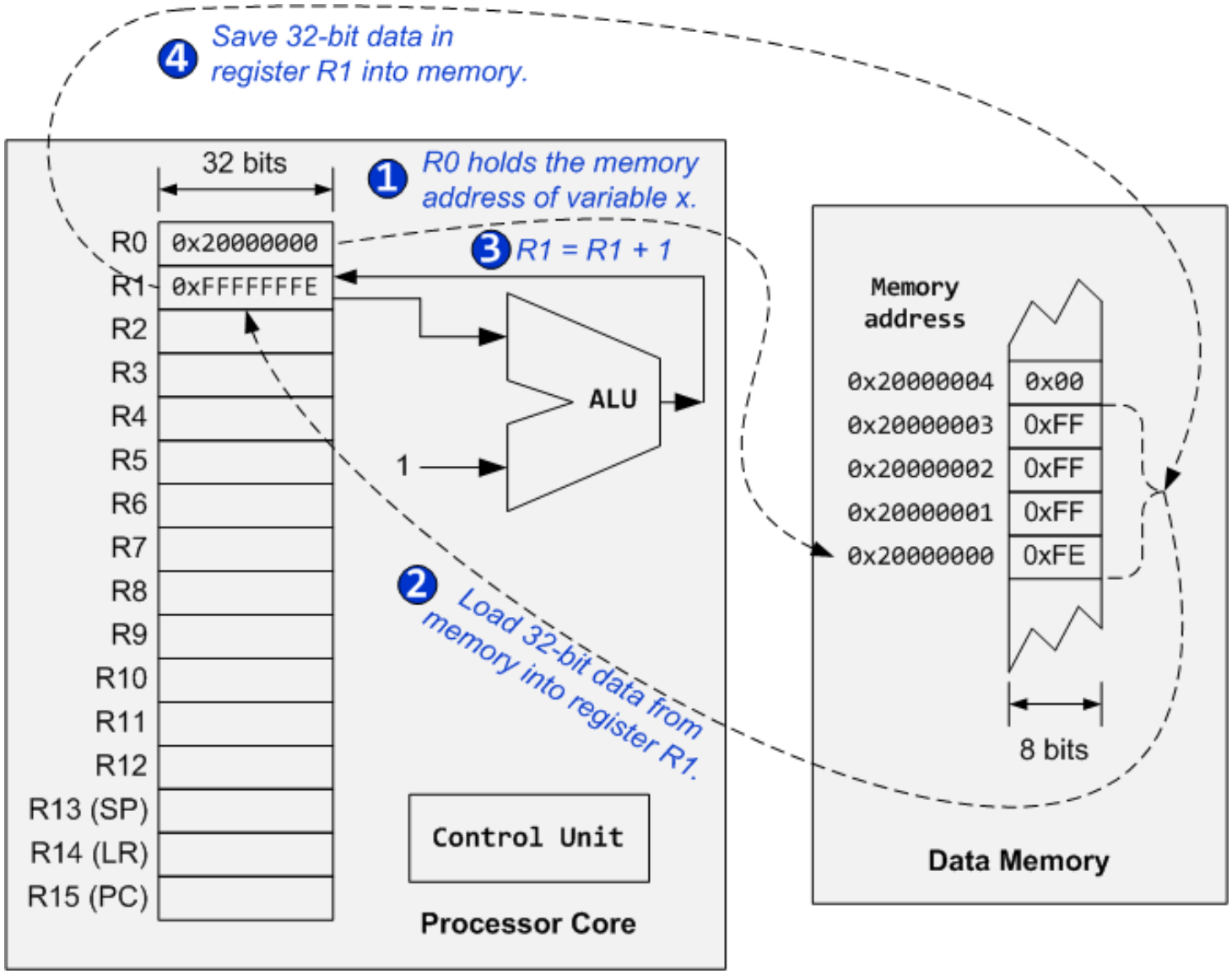
Load-Modify-Store Architecture



– Load/Store architecture
  - ALU operations are simple and are always between registers
  - Use simple dedicated load/store instructions for register-memory transfers (which should be less frequent than ALU operations)



Load-Modify-Store



Operating Modes

- Thread Mode for User tasks – Main program – Foreground
- Handler Mode for OS tasks and exceptions - Interrupt Service - Background