



# Programming Microcontrollers

A COMPUTER SCIENTIST'S PERSPECTIVE

March 2022

André Pereira

# Agenda

- Goals & Pre-requisites
- Programming abstraction levels
- What's an Instruction Set Architecture?
- CISC vs RISC
- Microprocessors and microcontrollers
- Practical exercises

# Goals

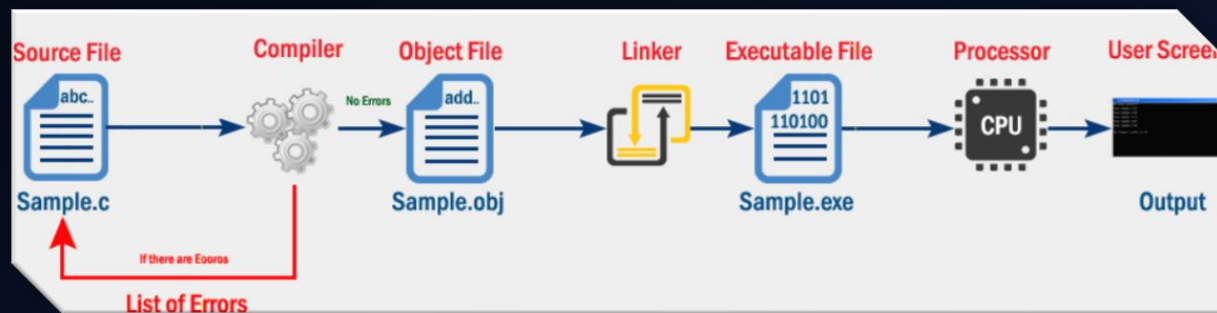
- Understand the architectural differences between microprocessors and microcontrollers
- Learn how to code and interact with a microcontroller
- Apply the fundamentals of microprocessor programming in assembly to microcontrollers

# Pre-requisites

- Previous enrollment in the *Sistemas de Computação* Curricular Unit
- A laptop with the AVR\_SIM v2.5 tool installed
  - Available for Linux and Windows
  - Download: [http://www.avr-asm-tutorial.net/avr\\_sim/index\\_en.html#download](http://www.avr-asm-tutorial.net/avr_sim/index_en.html#download)
  - Work with a colleague!
- Patience for my lack of knowledge in electronic engineering

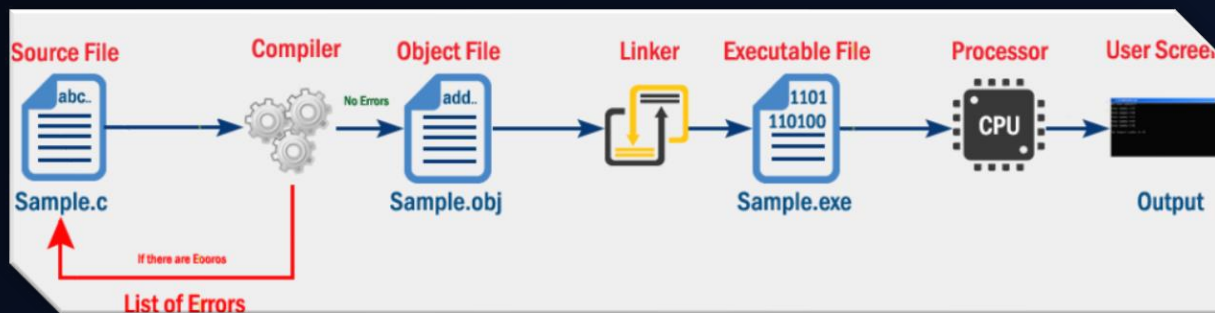
# Abstraction Levels

- High Level Languages (HLL)
  - Independent of the hardware
  - Various programming paradigms (functional, imperative, object-oriented, ...)
  - User usually codes at this abstraction level
    - Python, C/C++, Java, ...
- Assembly Languages
- Machine Languages



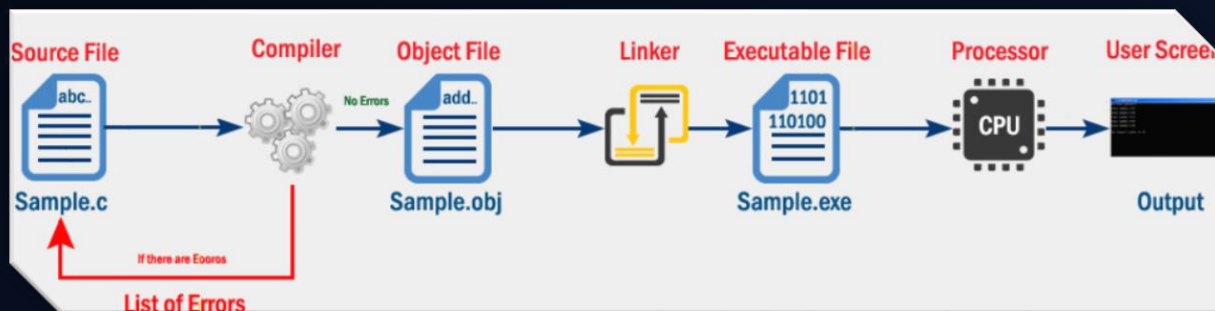
# Abstraction Levels

- High Level Languages (HLL)
- Assembly Languages
  - Hardware supports a given Instruction Set Architecture (ISA)
  - Availability of instructions is dependent on the hardware
  - Several ISAs available
    - IA-16/32, x86/x64, MIPS, ...
- Machine Languages



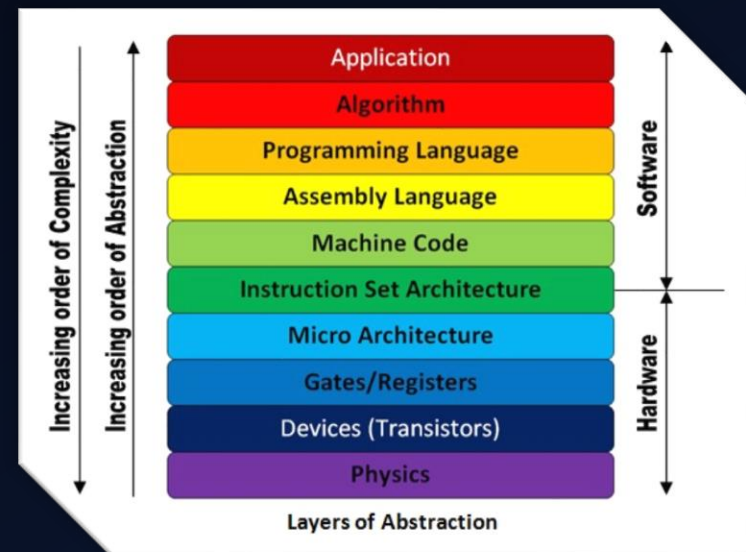
# Abstraction Levels

- High Level Languages (HLL)
- Assembly Languages
- Machine Languages
  - A list of commands supported by the hardware
  - Dependent on the specific chip
  - Chips from the same architecture may have different commands
  - Represented as pure binary
  - Two design philosophies, which impact the supported ISA
    - Complex Instruction Set Computers (CISC)
    - Reduced Instruction Set Computers (RISC)



# Instruction Set Architecture (ISA)

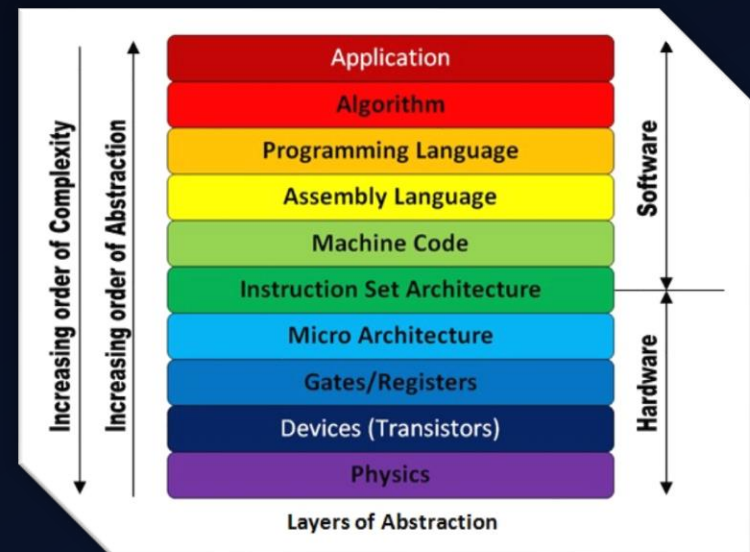
- Defines a set of instructions to control the behavior of a chip
  - Logic and arithmetic operations
  - Operand data access
  - Control of the execution flow
  - Input/output and others
- Imposes restrictions to the instruction representation in memory





# Instruction Set Architecture (ISA)

- Defines a set of instructions to control the behavior of a chip
- Imposes restrictions to the instruction representation in memory
  - Assembly reflects the instructions at HW level
  - Variable vs fixed instruction length



# ISA – Supported Operations

- Logic and arithmetic operations
  - And, or, xor, not, cmp, ...
  - Add, sub, mul, div, inc, dec, ...
- ISA defines the number of operands for each instruction
  - 3 operands – RISC for microprocessors (ex., ARM)
  - 2 operands – RISC/CISC for microprocessors (ex., IA-16)
  - 1 operand – RISC for microcontrollers
  - 0 operands – RISC/CISC

# ISA – Access and Types of Operands

- Operand types
  - Immediates
  - Scalars in registers
  - Complex data structures in memory (ex., C structs, arrays)
- Operand data access

## RISC

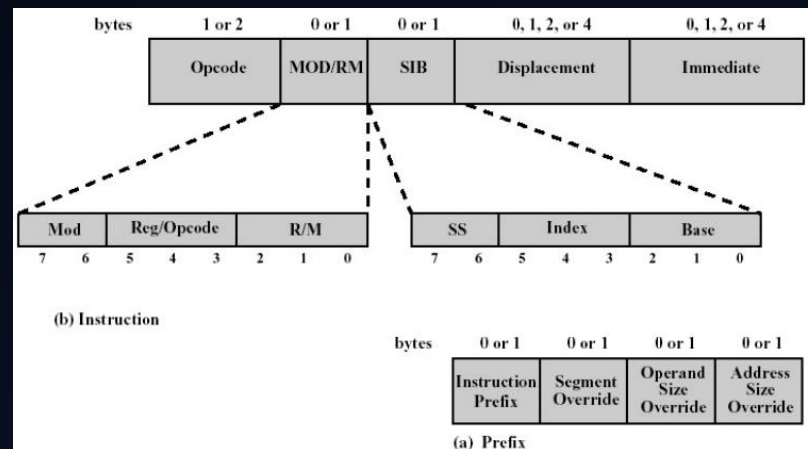
- L&A ops – all operands in registers
- Few address specification modes
- 32 general purpose registers

## CISC

- L&A ops – operands in registers or register/memory
- Several address specification modes
- 8 general purpose registers (IA-16)

# ISA – Instruction Length in Memory

- Variable instruction length
  - Instructions can range from 1 to several bytes
  - Representation organized in sections
    - The sections used may vary between instructions
  - Code requires less memory space (a big limitation in the past)
  - More complex Control Units and Decoder are required
    - Slow(er) fetch and decoding of a complex instruction

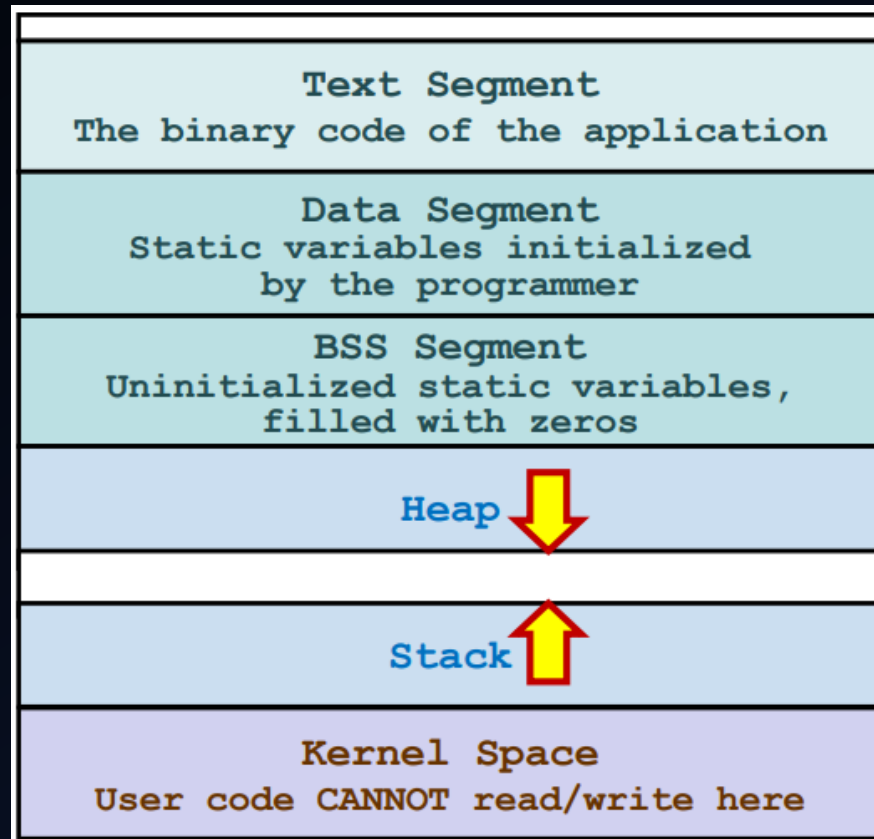


# ISA – Instruction Length in Memory

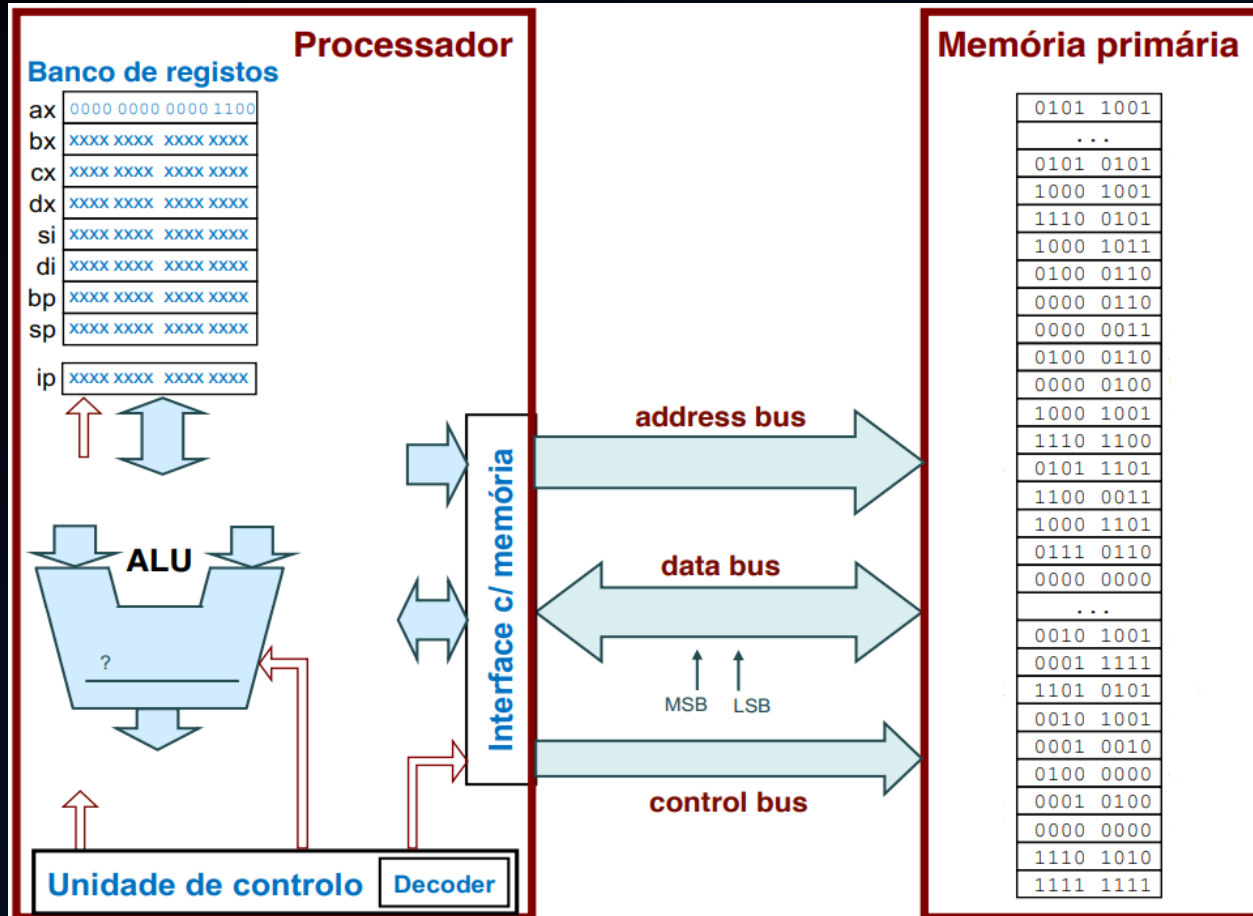
- Fixed instruction length
  - Instructions often coded using 32 bits
  - Representation organized in sections
    - Unused sections are offset to add up to 32 bits
  - Larger in-memory code bases (still a few KiBs to MeBs)
  - Simpler and faster decoding

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
data processing immediate shift	cond		0			0			0			opcode			S	Rn				Rd				shift amount				shift		0	Rm							
data processing register shift	cond		0			0			0			opcode			S	Rn				Rd				Rs				0	shift		1	Rm						
data processing immediate	cond		0			0			1			opcode			S	Rn				Rd				rotate				immediate										
load/store immediate offset	cond		0			1			0			P	U	B	W	L	Rn				Rd				immediate													
load/store register offset	cond		0			1			1			P	U	B	W	L	Rn				Rd				shift amount				shift		0	Rm						
load/store multiple	cond		1			0			0			P	U	S	W	L	Rn				register list																	
branch/branch with link	cond		1			0			1			L	24-bit offset																									

# Program Organization in Memory



# Instruction Execution on IA-16 (CISC)



# A CISC vs RISC Summary

## CISC (IA-16)

- 6+2 generic registers
  - Intensive use of the stack
    - Local variables
    - Function argument passing
    - Memory accesses are not efficient!
- Smaller codebases in memory
  - Variable length instructions
- Complex operations
  - Ex. push

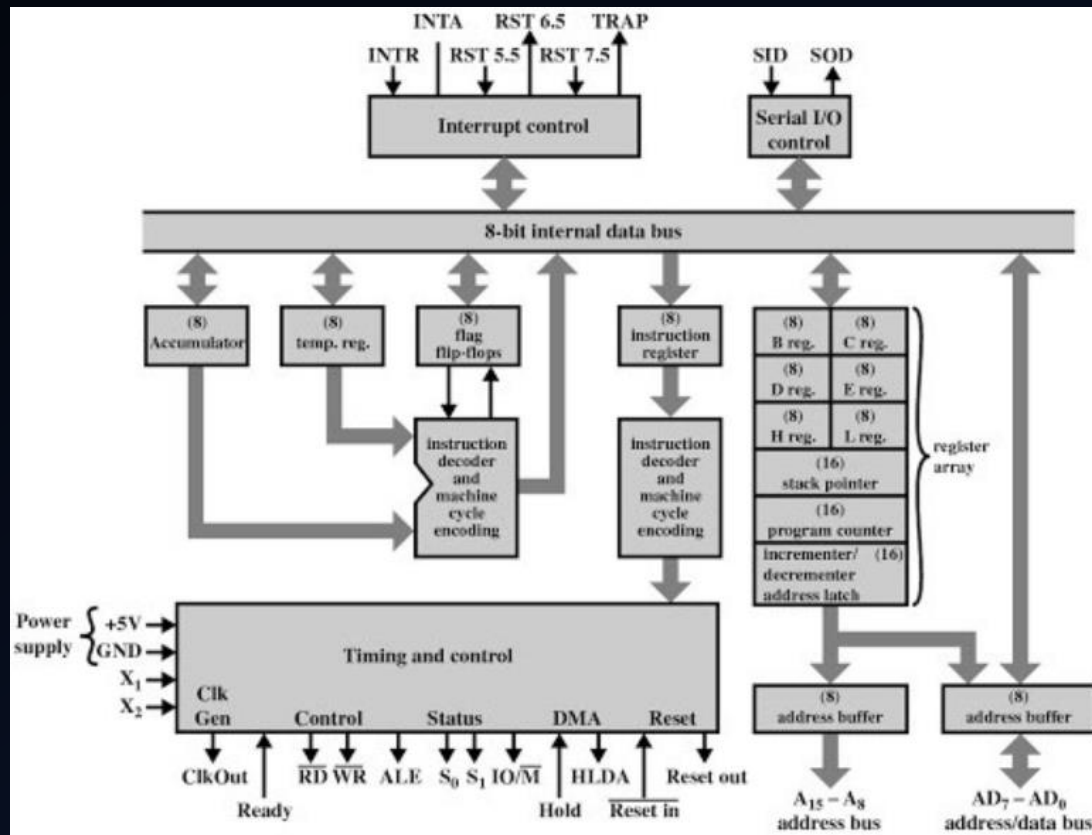
## RISC

- 16 to 32 generic registers
  - Stack used much less
    - Fewer variables in memory
    - Registers used to pass arguments to functions
    - Function return address in register
- Larger codebases in memory
  - Fixed length instructions
- Simpler operations
  - Ex. push -> sub + store



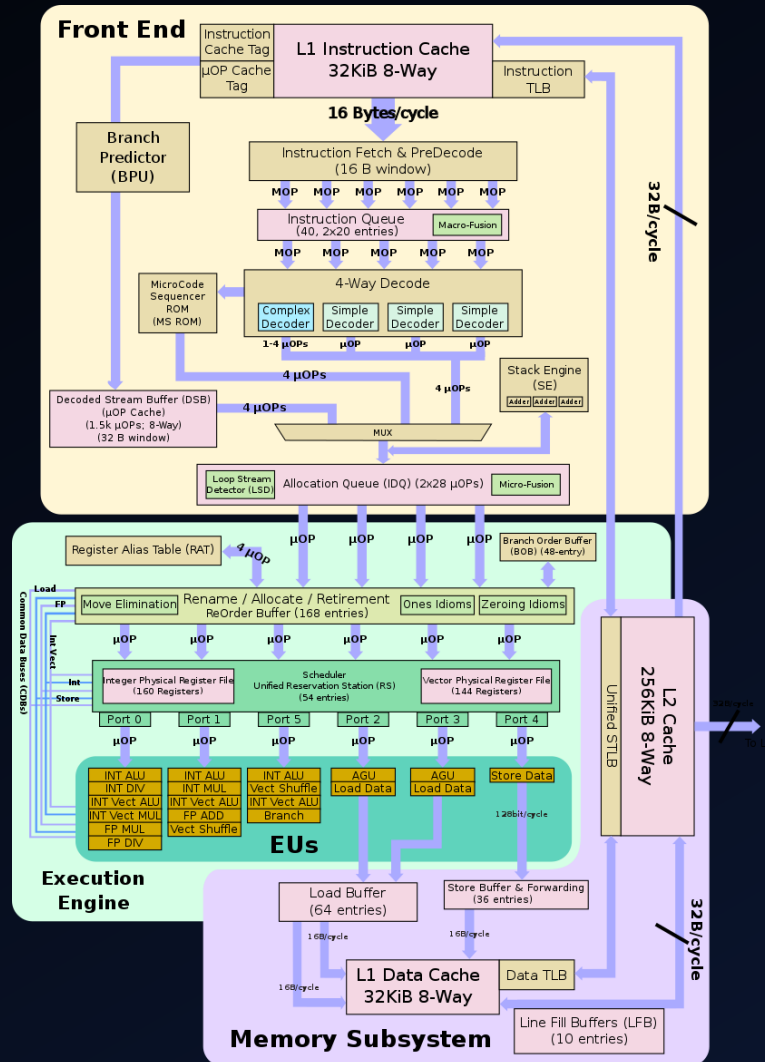
# Microprocessor Architecture

## Intel 8085 (1976)



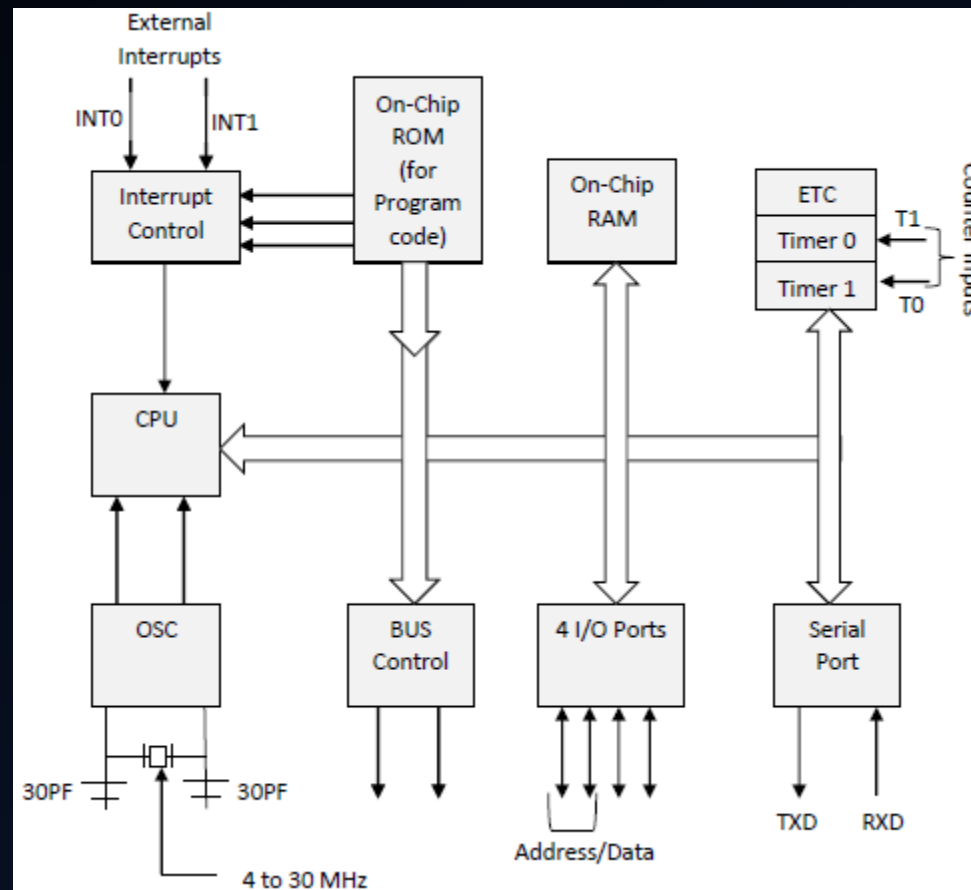
# Microprocessor Architecture

## Intel Ivy Bridge (2012)

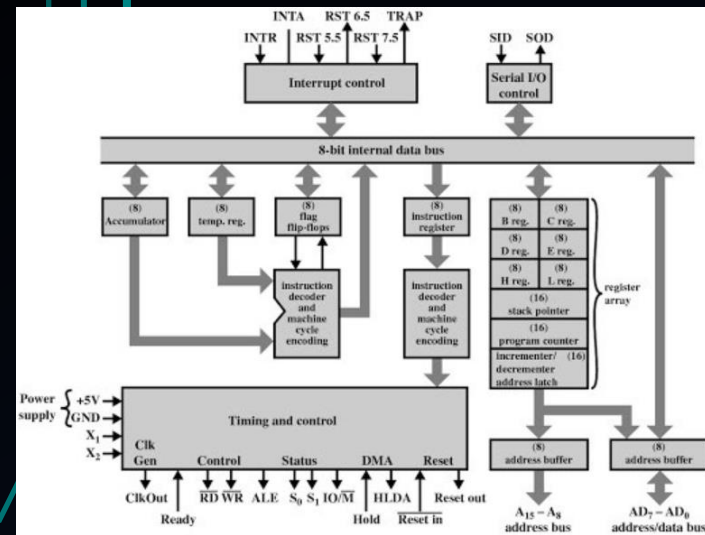


# Microcontroller Architecture

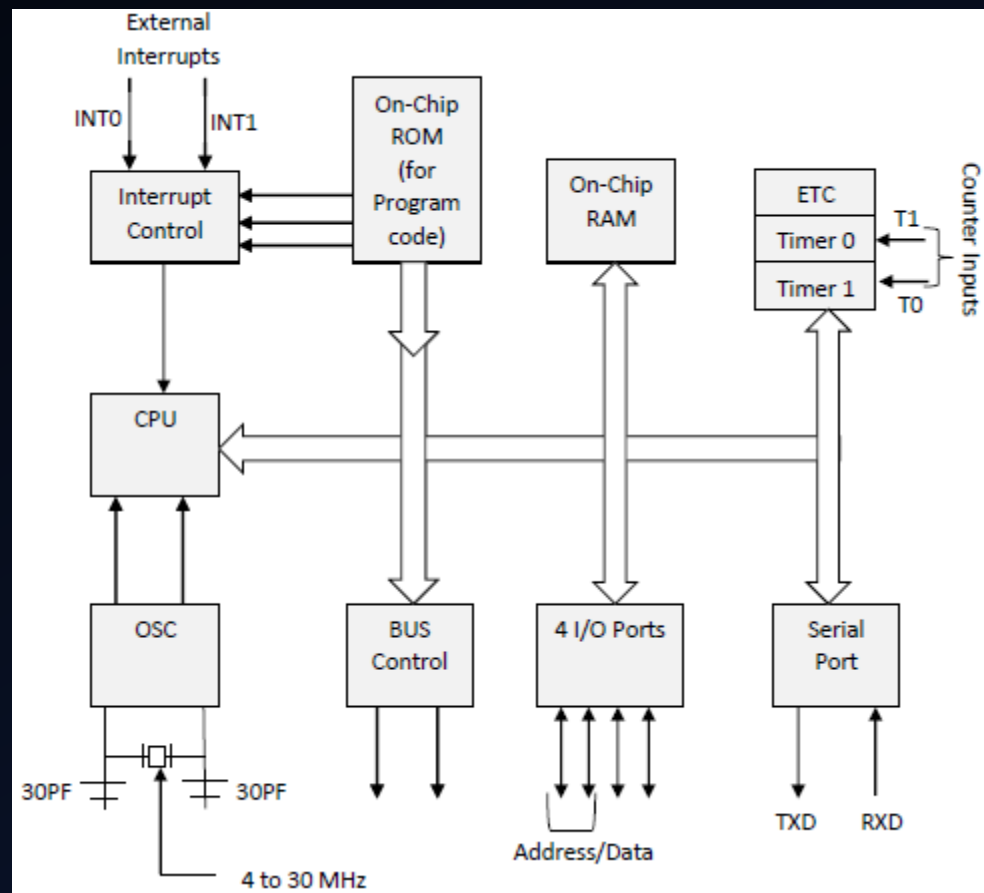
## Intel 8051 (1980)



# Microcontroller Architecture



## Intel 8051 (1980)



# The Differences

## MICROPROCESSOR

- Chips designed for computation
  - Generic operations (add, sub, ...)
  - Complex operations (sin, cos, ...)
  - Different designs for different computations
  - Usually computational efficiency is key
  - Fast clock speeds (up to 5.4 GHz)
  - Expensive and complicated
  - Must be integrated into a system
  - Generic use

## MICROCONTROLLER

- Chips designed for electronic device control
  - Integrates a simple microprocessor
  - Embeds memory (volatile and non-volatile) and I/O
  - Energy efficiency and thermal dissipation are key
  - Slow clock speeds (< 200 MHz)
  - Cheap and simple
  - SoC-like
  - Application specific

# Microcontroller Programming Models

## LINEAR/IMPERATIVE

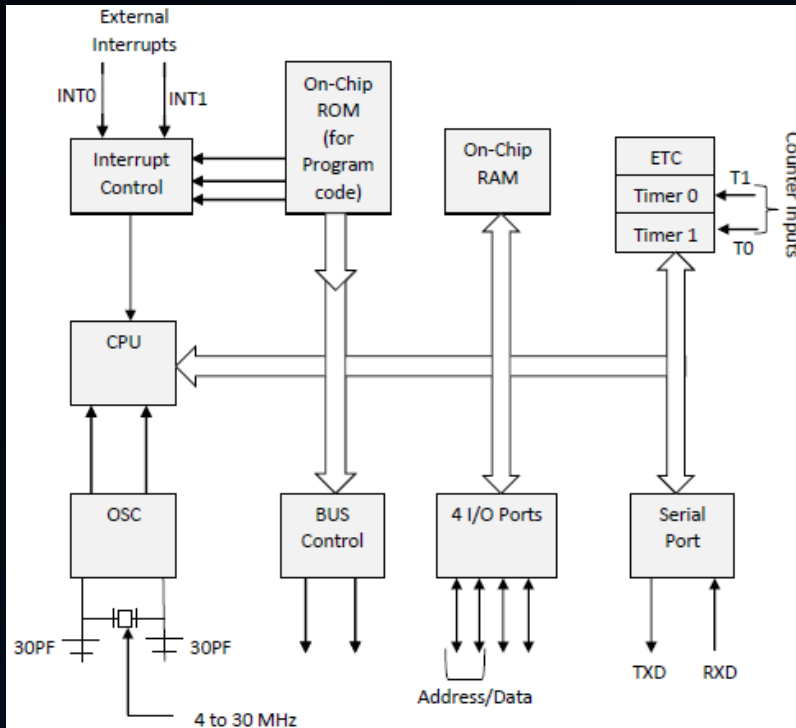
- Program based on statements
  - Change the program state
- Statements define commands for the system to execute
- Implements *how* a program performs an algorithm
- Statements executed in sequence by default
- C, Fortran, IA-32 ISA, ...

## INTERRUPT BASED

- Hardware interrupts trigger function calls
  - Software interrupts not usually supported
  - Can happen in the middle of a multi-cycle instruction execution
- Interrupts must be dealt with immediately
- Functions often programmed linearly
- ATMEL ISA, ...

# Microcontroller Programming Models

## INTERRUPT BASED



- Hardware interrupts trigger function calls
  - Software interrupts not usually supported
  - Can happen in the middle of a multi-cycle instruction execution
- Interrupts must be dealt with immediately
- Functions often programmed linearly
- ATMEL ISA, ...

# ATMEL Microcontrollers

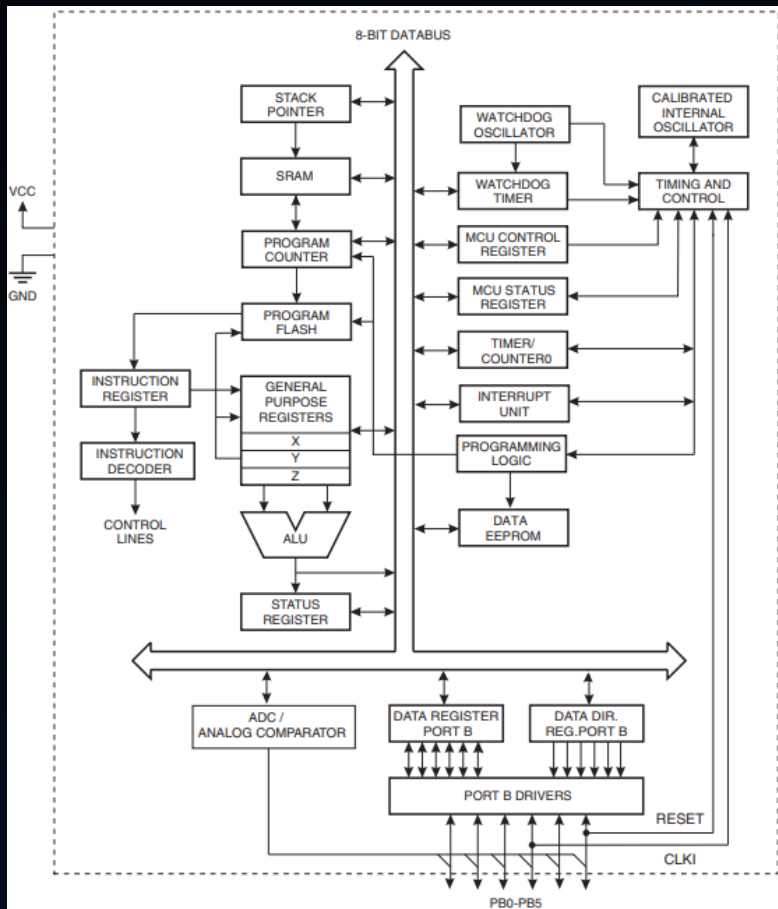
- ATTiny13
  - RISC (120 ins)
  - 8 bit
  - 32 8bit general purpose registers
  - 1 – 10 MHz
  - 1 KiBytes flash storage
  - 1x 8bit timer
  - 6 I/O lines (ports)
  - ...
- ATMega32
  - Advanced RISC Arch (131 ins)
  - 32 8bit general purpose registers
  - 32 KiBytes flash storage
  - 1 – 16 MHz clock
  - 2x 8bit + 1x 16bit timers
  - 32 I/O lines

Peak Instruction throughput: 1MIPS @ 1 MHz

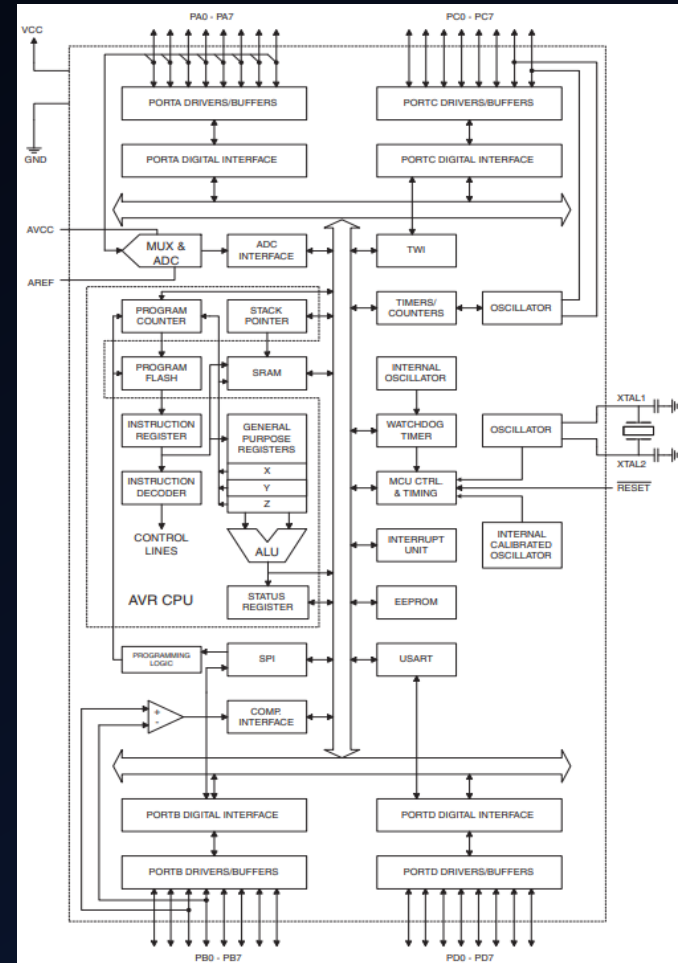


# ATMEL Microcontrollers

## ATTiny13

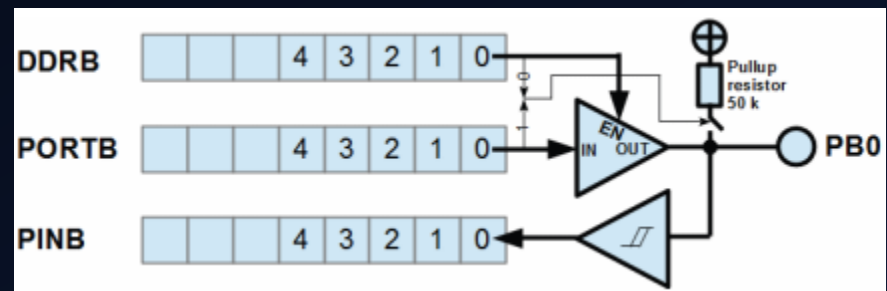
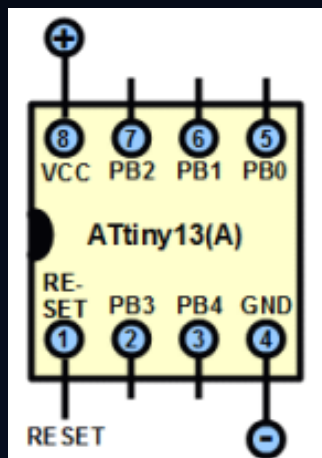


## ATMega32



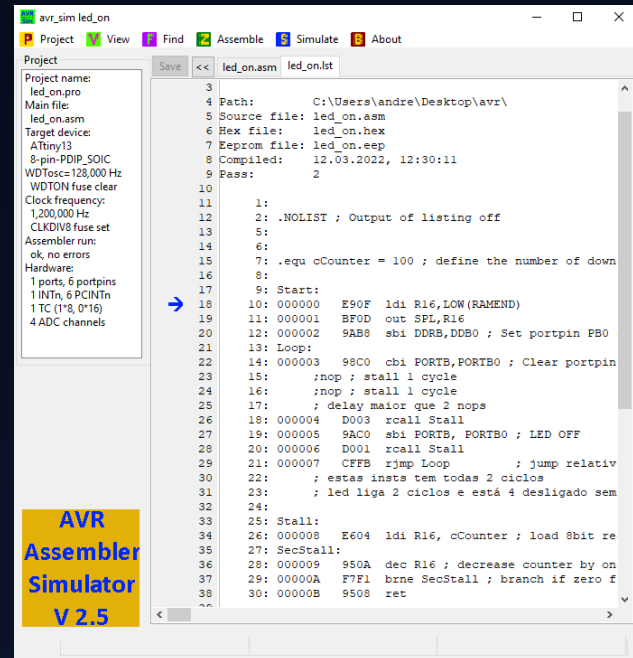
# ATTiny13

- 5 I/O addressable port pins: PB0 to PB4
  - Each controlled by a bit in DDRB and PORTB “registers”
  - DDRB (Data Direction Register port B): 1/0 -> output/input (extremely simplified, depends on breadboard setup)
  - PORTB (data output register port B): 1/0 -> high/low voltage
  - PINB (input register B): stores the logical state of the port pin
- Processing unit programmed similarly to RISC microprocessors



# AVR Simulator

- Tool for the simulation of code execution in microcontrollers
  - Assembles the ISA to produce machine code
  - Simulates ATMEL microcontrollers
- Allows full control of
  - View and change bits in memory and registers
  - Timer and interrupt manipulation
  - Microcontroller clock frequency
- Looks are inversely proportional to the tool potential/usefulness



The screenshot displays the AVR Simulator V2.5 interface. On the left, a 'Project' panel lists settings: Project name (led\_on.pro), Main file (led\_on.asm), Target device (ATtiny13), 8-pin PDIP\_SOIC, WDTosc=128,000 Hz, WDTON fuse clear, Clock frequency: 1,200,000 Hz, CLKDIV8 fuse set, Assembler run: ok, no errors, Hardware: 1 ports, 6 portpins, 1 INTn, 6 PCINTn, 1 TC (1\*8, 0\*16), 4 ADC channels. The main window shows assembly code for 'led\_on.asm' with a list of instructions and their addresses. A blue arrow points to line 17, which is the start of the program. The code includes comments in Portuguese and assembly instructions like 'ldi R16, LOW(RAMEND)', 'out SFL, R16', 'sbi DDRB, DDB0', 'cbi PORTB, PORTB0', 'rcall Stall', 'rjmp Loop', 'dec R16', and 'brne SecStall'.

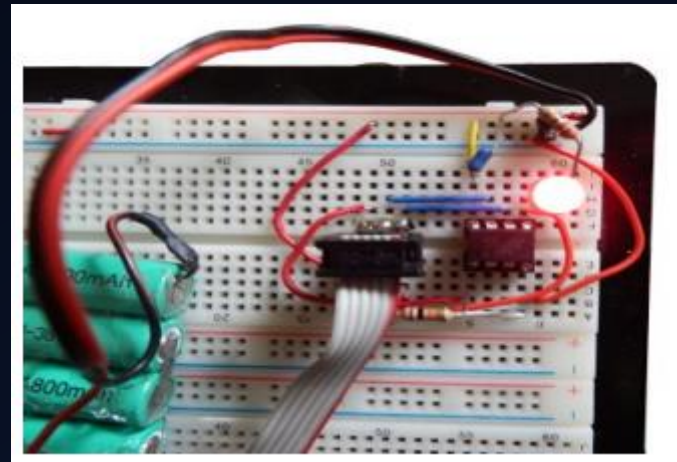
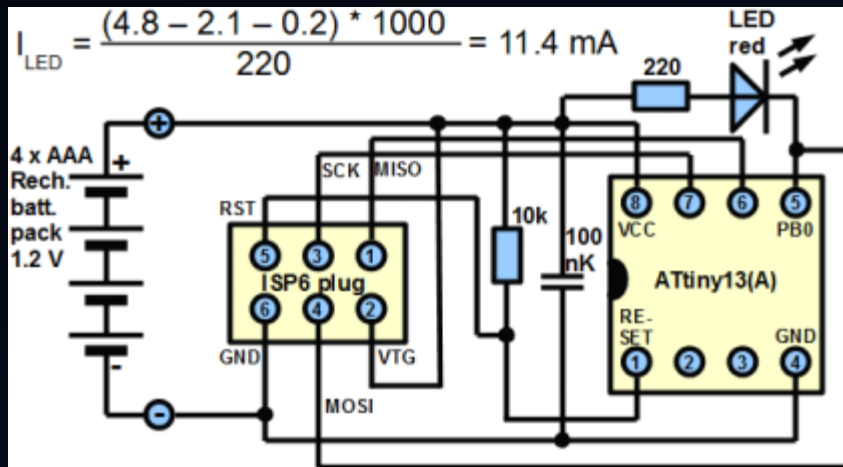
```
3
4 Path: C:\Users\andre\Desktop\avr\
5 Source file: led_on.asm
6 Hex file: led_on.hex
7 Eeprom file: led_on.eep
8 Compiled: 12.03.2022, 12:30:11
9 Pass: 2
10
11 1:
12 2: .NOLIST ; Output of listing off
13 5:
14 6:
15 7: .equ cCounter = 100 ; define the number of down
16 8:
17 9: Start:
18 10: 000000 E90F ldi R16, LOW(RAMEND)
19 11: 000001 BF0D out SFL, R16
20 12: 000002 9AB8 sbi DDRB, DDB0 ; Set portpin PB0
21 13: Loop:
22 14: 000003 98C0 cbi PORTB, PORTB0 ; Clear portpin
23 15: ;nop ; stall 1 cycle
24 16: ;nop ; stall 1 cycle
25 17: ; delay maior que 2 nops
26 18: 000004 D003 rcall Stall
27 19: 000005 9AC0 sbi PORTB, PORTB0 ; LED OFF
28 20: 000006 D001 rcall Stall
29 21: 000007 CFFB rjmp Loop ; jump relativ
30 22: ; estas insts tem todas 2 ciclos
31 23: ; led liga 2 ciclos e está 4 desligado sem
32 24:
33 25: Stall:
34 26: 000008 E604 ldi R16, cCounter ; load 8bit re
35 27: SecStall:
36 28: 000009 950A dec R16 ; decrease counter by on
37 29: 00000A F7F1 brne SecStall ; branch if zero f
38 30: 00000B 9508 ret
```

# Practical Session

[HTTPS://GITHUB.COM/AMPEREIRA90/MICROCONTROLLERS](https://github.com/AMPEREIRA90/MICROCONTROLLERS)

# A Practical Example (1)

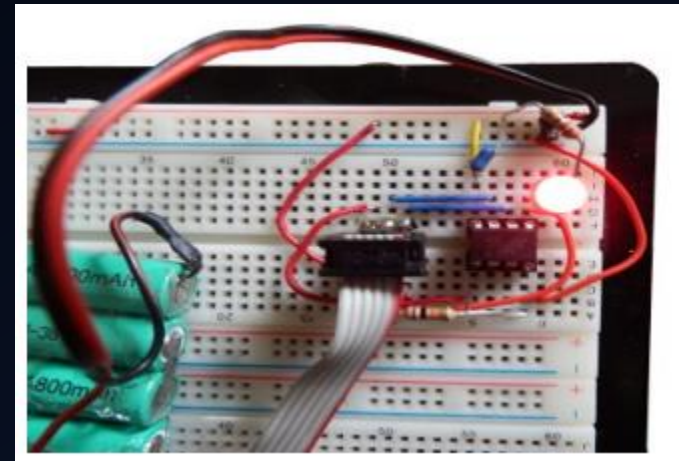
- Switch a LED on/off each couple of cycles
- In a “sink” configuration, output bit turns on the LED



# A Practical Example (1)

> led\_on\_off.asm file

```
1  .NOLIST ; Output of listing off
2  .INCLUDE "tn13def.inc" ; Read port definitions
3  .LIST ; Output of listing on
4
5  Start:
6      ldi R16,LOW(RAMEND)
7      out SPL,R16
8      sbi DDRB,DDB0 ; Set portpin PB0 as output
9  Loop:
10     cbi PORTB,PORTB0 ; Clear portpin PB0 LED ON - 2 cycles
11     nop ; stall 1 cycle
12     nop ; stall 1 cycle
13     sbi PORTB, PORTB0 ; LED OFF - 2 cycles
14     rjmp Loop ; jump relative back to label
```

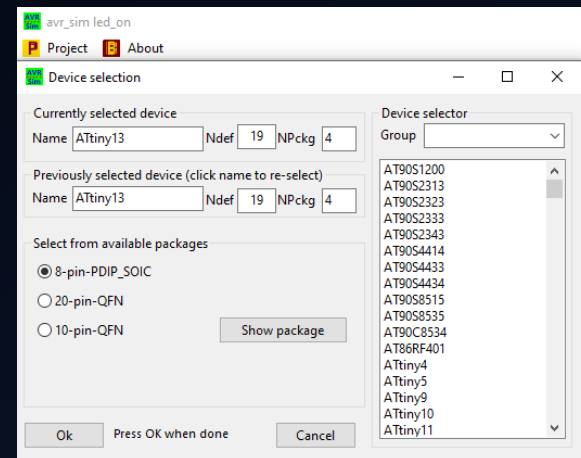
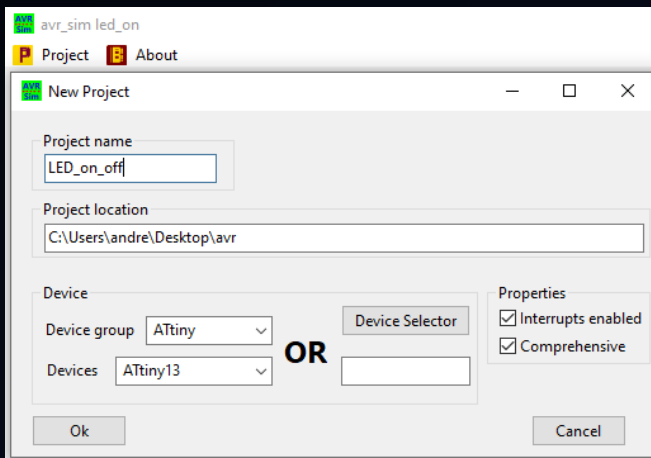


- How long does the LED stays on and off?
- Let's simulate this!

# A Practical Example (1)

> `led_on_off.asm` file

- Create a “New Project” in AVR\_SIM

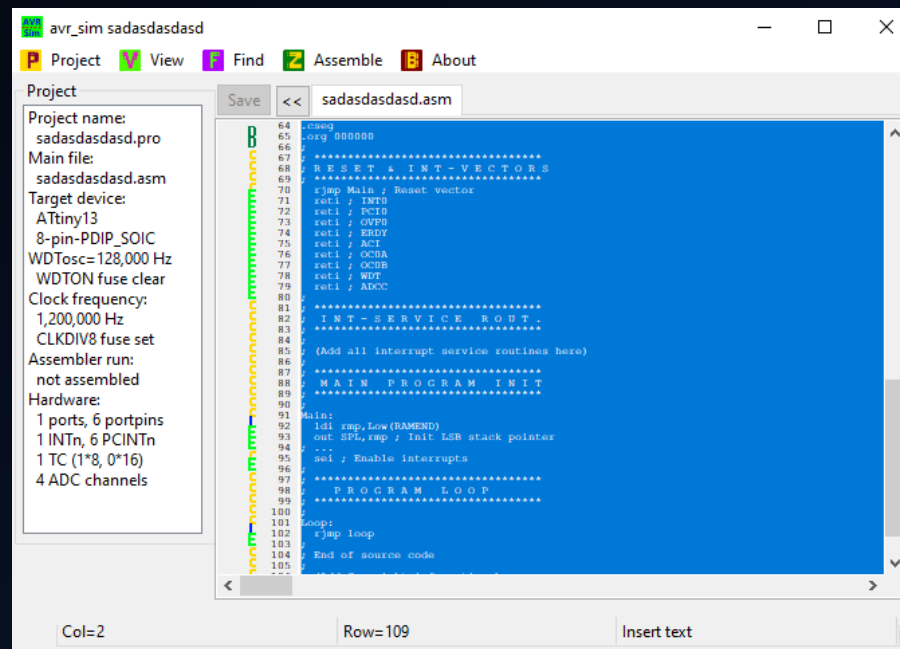


- Delete all default code and implement your own
- “Assemble” and then “Simulate”

# A Practical Example (1)

> led\_on\_off.asm file

- Create a “New Project” in AVR\_SIM
- Delete all default code and implement your own
- “Assemble” and then “Simulate”





# A Practical Example (1)

> `led_on_off.asm` file

The screenshot displays the AVR simulator interface for the `avr_sim LED_on_off` project. The **Port** window shows the state of various I/O registers:

(Help)	7	6	5	4	3	2	1	0
PORTB	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	0
PINB	0	0	0	0	0	0	0	0
TINnB	0	0				0		
INTnB	0	0					0	
PCINT0	0	0	5	4	3	2	1	0

The **Simulation status** window shows the following information:

- Prog counter = \$000000
- Instructions = 0
- Stackpointer = \$0000
- Watchdog = 0.00000%
- Clock frequ. = 1,200,000 Hz
- Time elapsed = 0.00 ns
- Stop watch = 0.00 ns

The **SREG** window shows the status register flags:

I	T	H	S	V	N	Z	C
0	0	0	0	0	0	0	0

The **Register** window shows the state of registers R0 through R24:

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	00	00

The **Messages** window shows the following message:

```
$0000: Starting
```

The **Tools** window shows the following options:

- ☒ Ports
- ☐ Timers
- ☐ WDT
- ☐ ADC
- ☐ EEPROM
- ☐ SRAM
- ☐ Scope
- ☐ Alert

- “Step” executes the next instruction
- “Run” executes the whole program

## A Practical Example (2)

> led\_on\_off\_stall.asm file

- Let's complicate it
  - Replace the “nops” by a function call to “Stall”
  - “Stall” function should delay the execution by 100 cycles

```
9  Start:
10      ldi R16,LOW(RAMEND)
11      out SPL,R16
12      sbi DDRB,DDB0 ; Set portpin PB0 as output
13  Loop:
14      cbi PORTB,PORTB0 ; Clear portpin PB0 LED ON
15      rcall Stall
16      sbi PORTB, PORTB0 ; LED OFF
17      rcall Stall
18      rjmp Loop          ; jump relative back to label
19
20  Stall:
```

# A Practical Example (3)

> led\_on\_off\_counter.asm file

- Implement a program that
  - Reads the PINB1 bit
  - If the bit is 0 add R24 to R25
  - If the bit is 1 subtract R24 to R25
  - Flash the LED x times with an interval of 20 cycles
    - Where x is the result of the addition/subtraction
- Draw a diagram of the execution flow and tasks before coding!

```
7  .equ smallStall = 20
8  .equ firstVal = 5
9  .equ secVal = 2
10
11  Start:
12      ldi R16,LOW(RAMEND) ; fetch end of SRAM
13      out SPL,R16         ; set stack pointer to that
14      sbi DDRB,DDB0 ; Set portpin PB0 as output
15      cbi DDRB,DDB1 ; Set portpin PB1 as input
16      sbi PORTB,PORTB0 ; LED OFF
17
18      ldi R24,firstVal ; load firstVal into the R24 register
19      ldi R25,secVal
20
21      in R16,PINB ; read PINB into the R16 register
```

# Relevant Instructions – ATMEL ISA

IA-16 ISA	ATMEL ISA	Description
add r1, r2	add r2, r1	$r2 = r2 + r1$
sub r1, r2	sub r2, r1	$r2 = r2 - r1$
inc r1	inc r1	$r1 = r1 + 1$
dec r1	dec r1	$r1 = r1 - 1$
mov v1, r1	ldi r1, v1	$r1 = v1$
jmp label	rjmp label	ProgramCounter = label
call func	rcall func	ProgramCounter = func
jne	brne	Jump if ZF not set
-	in r1, PINB	Loads from I/O space to r1
-	out PINB, r1	Stores r1 on I/O space
-	sbi PORTB, PORTB0	PORTB0 bit set to 1 in PORTB
-	cbi PORTB, PORTB0	PORTB0 bit set to 0 in PORTB
and r1, r2	and r2, r1	$r2 = r1 \& r2$
and \$v, r1	andi r1, 0x00	$r1 = r1 \& \text{immediate}$



# Programming Microcontrollers

THE END