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

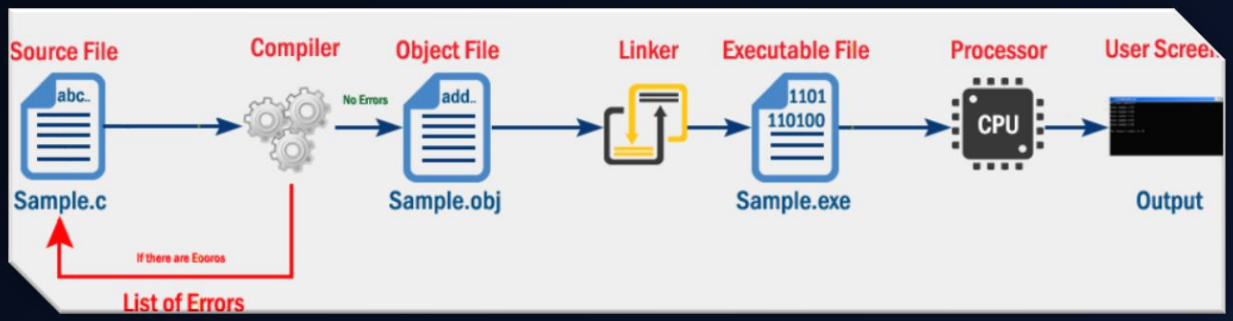
- Understand the architectural differences between microprocessors and microcontrollers
- Learn how to code and interact with a microcontroller
- Apply the fundamentals of microprocessor programming in assembly to microcontrollers

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
  - Work with a colleague!
- Patience for my lack of knowledge in electronic engineering

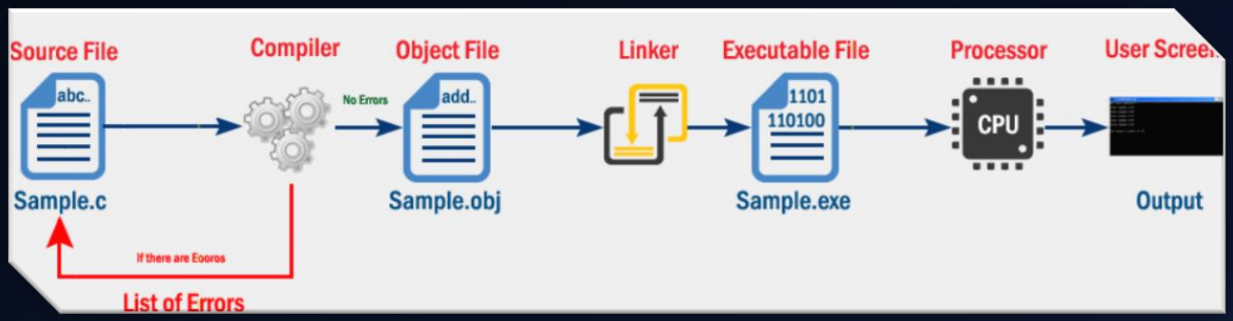
Abstraction Levels

- 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



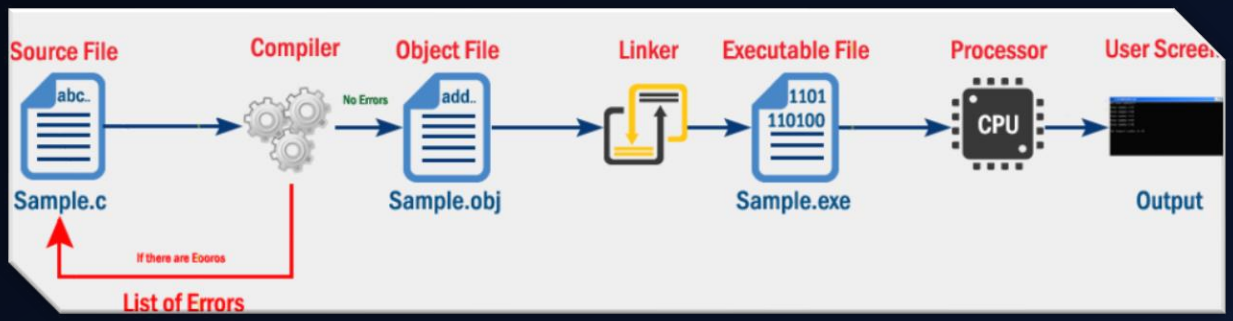
Abstraction Levels

- 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



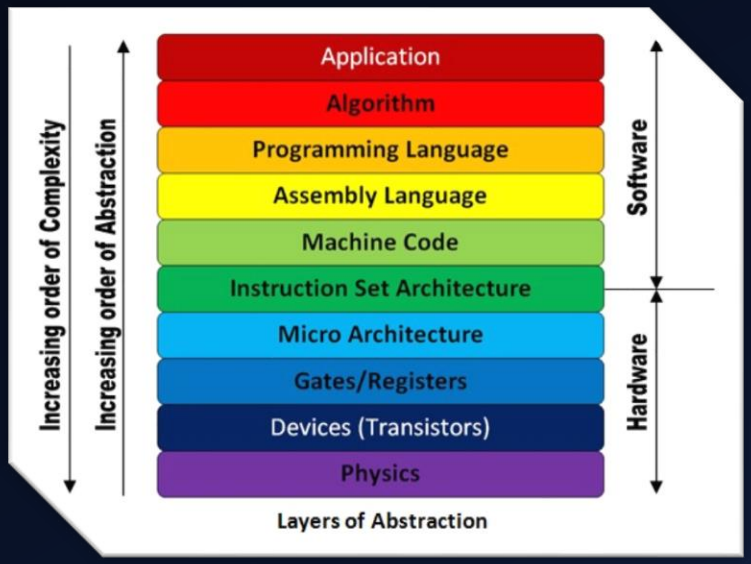
Abstraction Levels

- 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)



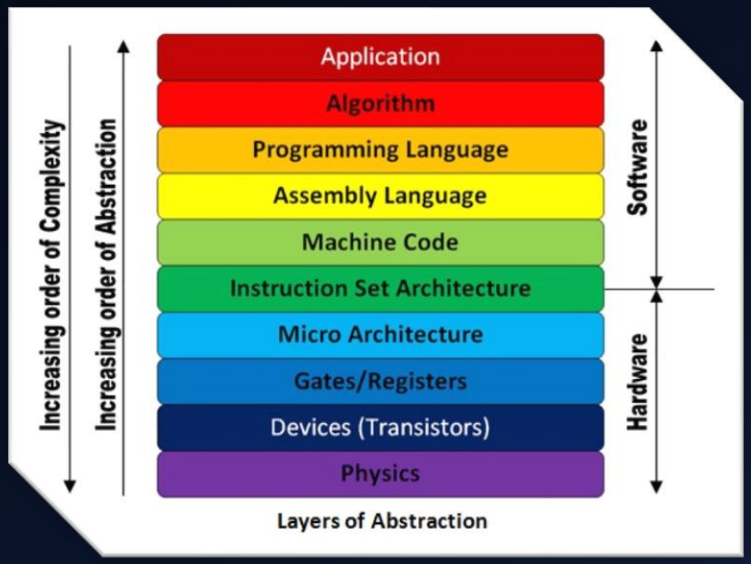
Instruction Set Architecture (ISA)

- 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

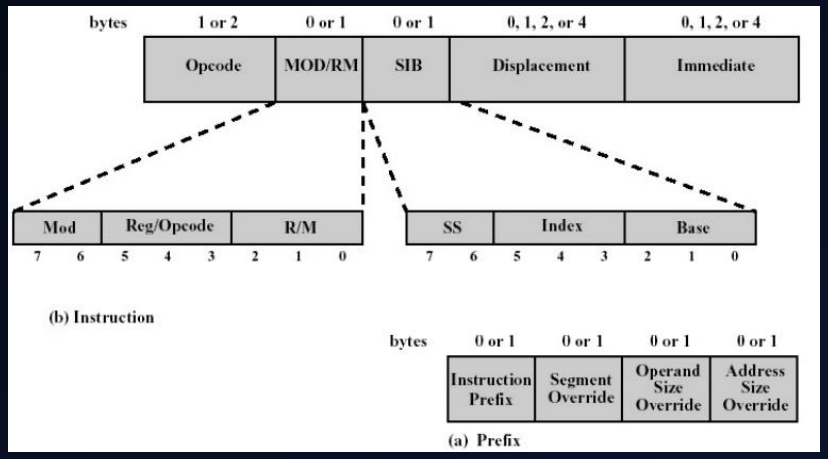
- Operand types
  - Immediates
  - Scalars in registers
  - Complex data structures in memory (ex., C structs, arrays)

- Operand data access

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



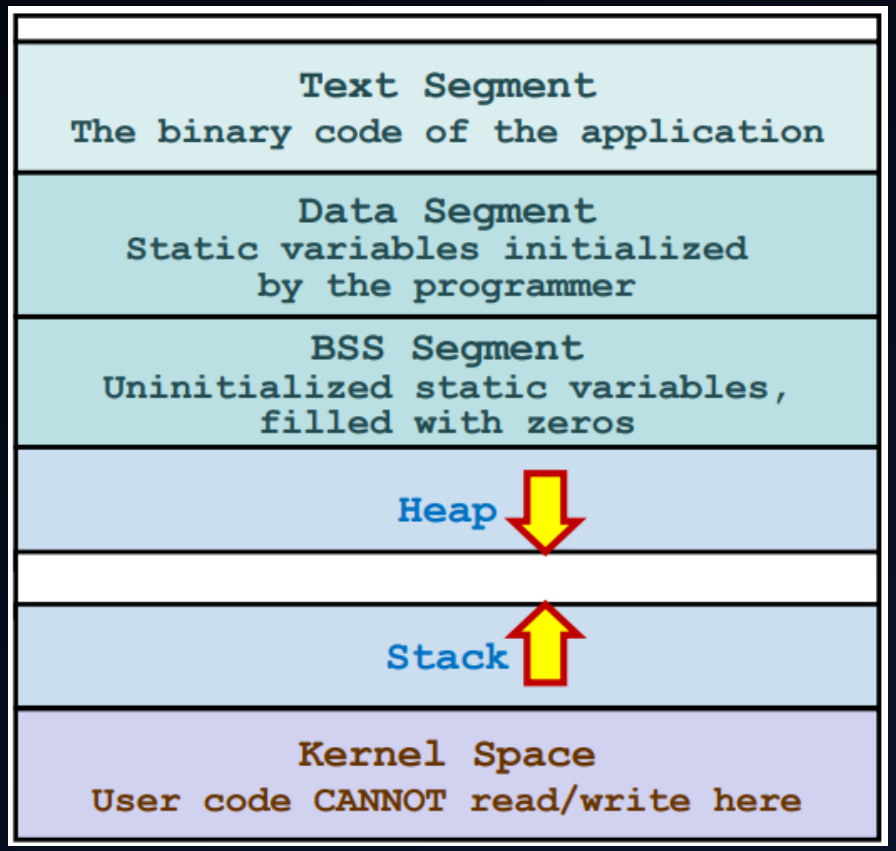
ISA – Instruction Length in Memory

- 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

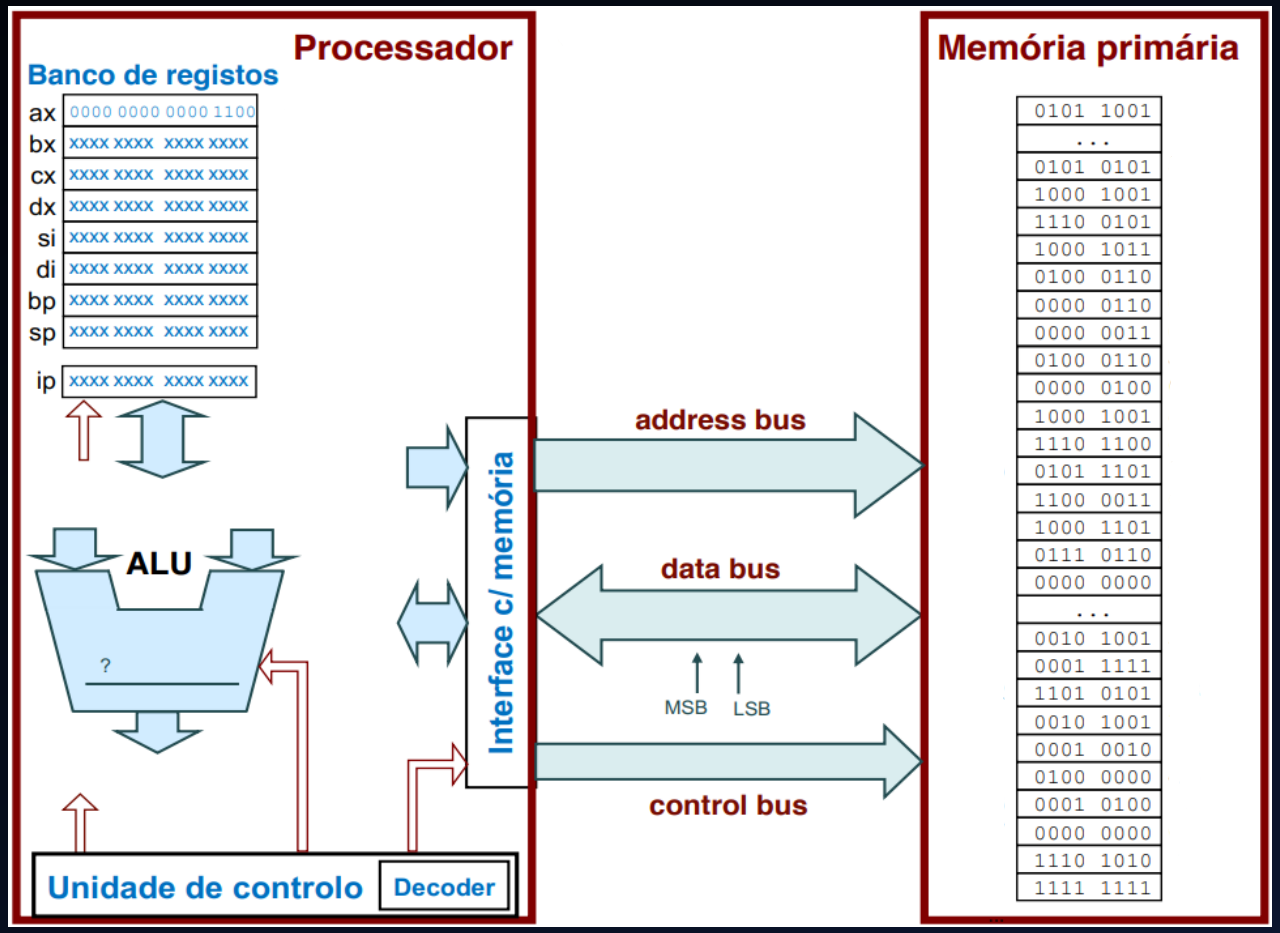
	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
data processing immediate shift	cond	0 0 0	opcode				S	Rn	Rd	shift amount		shift	0	Rm
data processing register shift	cond	0 0 0	opcode				S	Rn	Rd	Rs	0	shift	1	Rm
data processing immediate	cond	0 0 1	opcode				S	Rn	Rd	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										

ARM

Program Organization in Memory



Instruction Execution on IA-16 (CISC)



A *CISC* vs *RISC* Summary

CISC (IA-16)

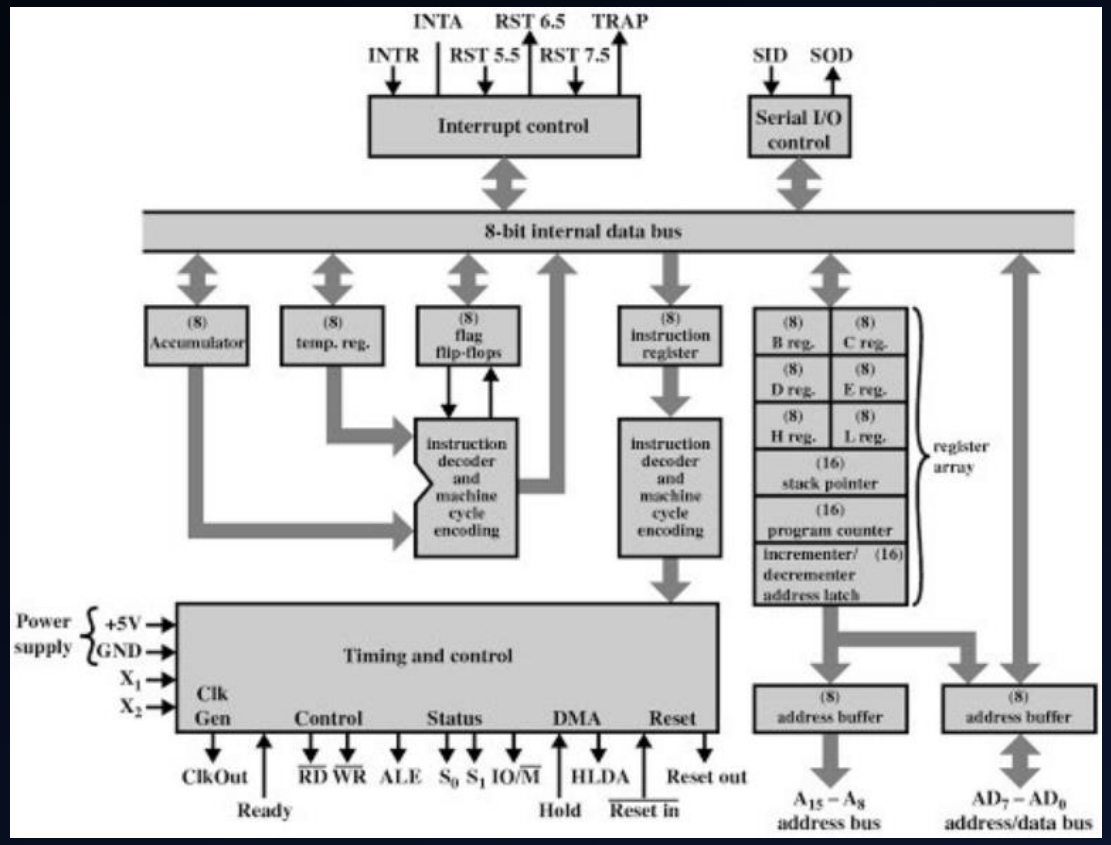
- 6+2 generic registers
  - Intensive use of the stack
    - Local variables
    - Function argument passing
    - Memory accesses are not efficient!
- Smaller codebases in memory
  - Variable length instructions
- Complex operations
  - Ex. push

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

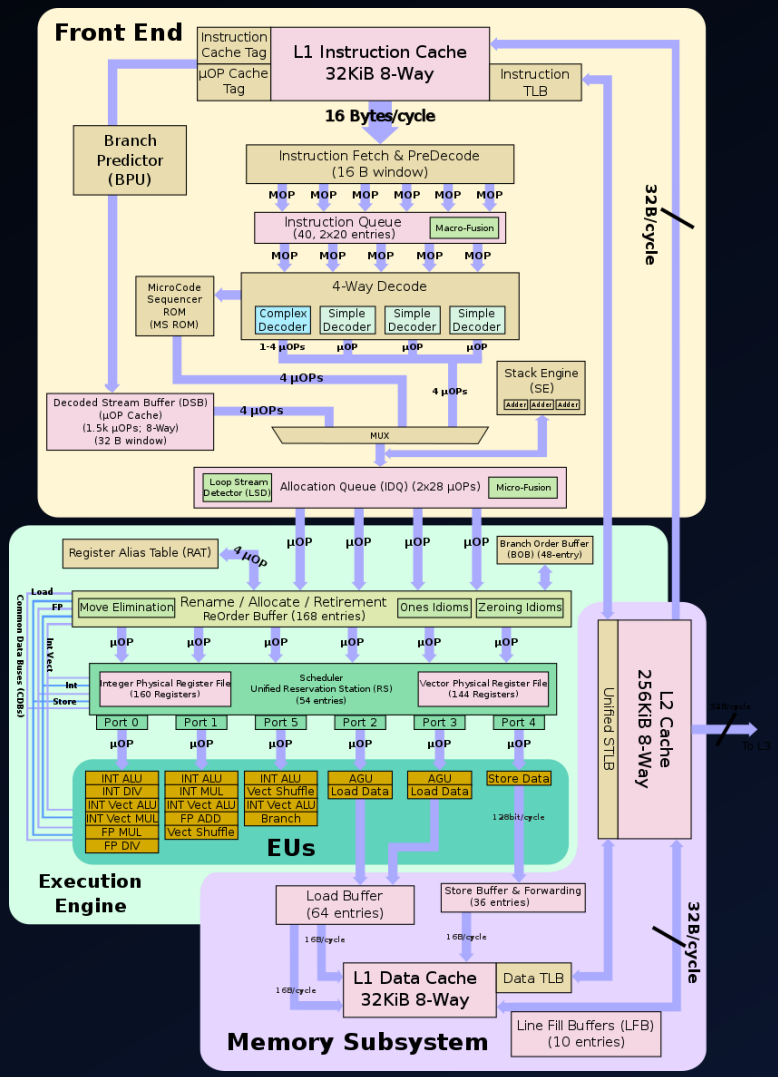
Microprocessor Architecture

Intel 8085 (1976)



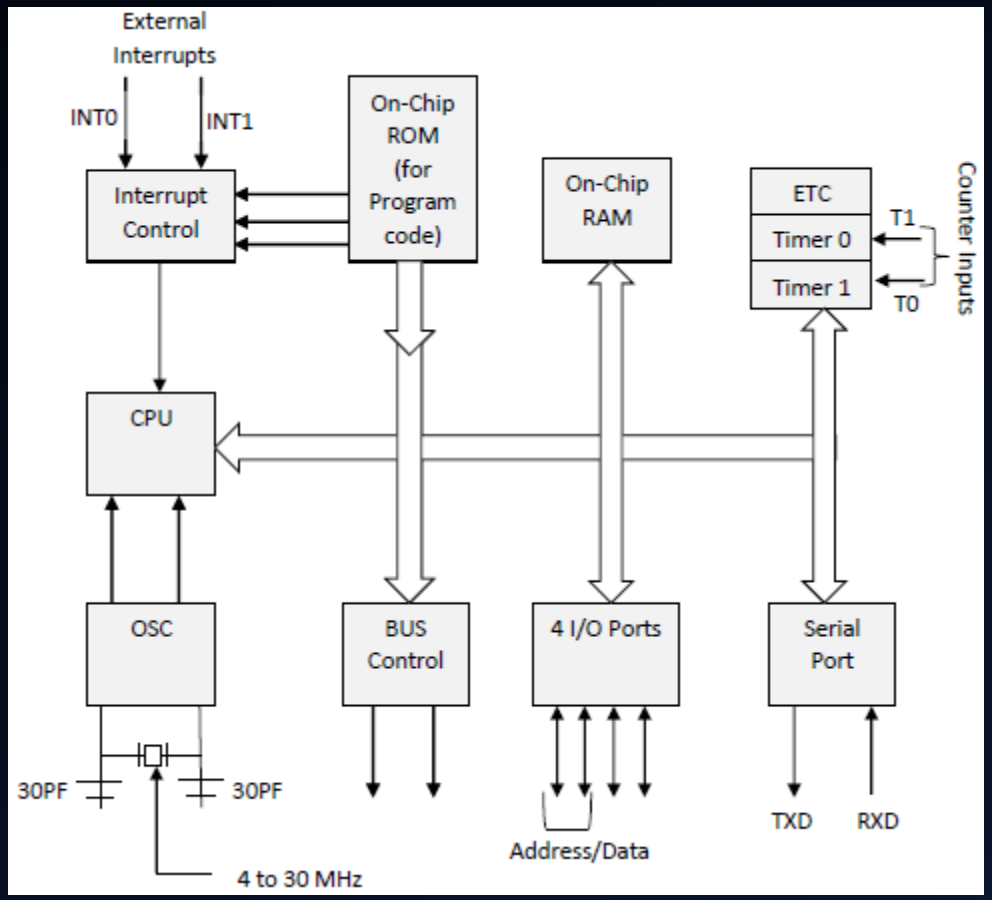
Microprocessor Architecture

Intel Ivy Bridge (2012)

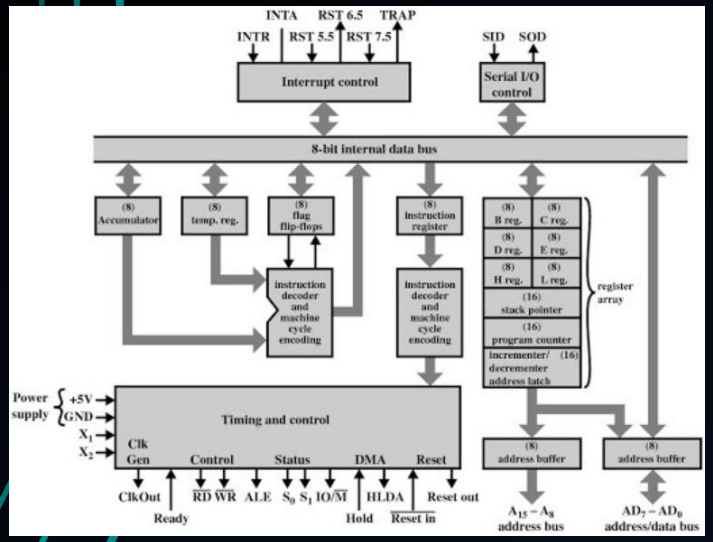


Microcontroller Architecture

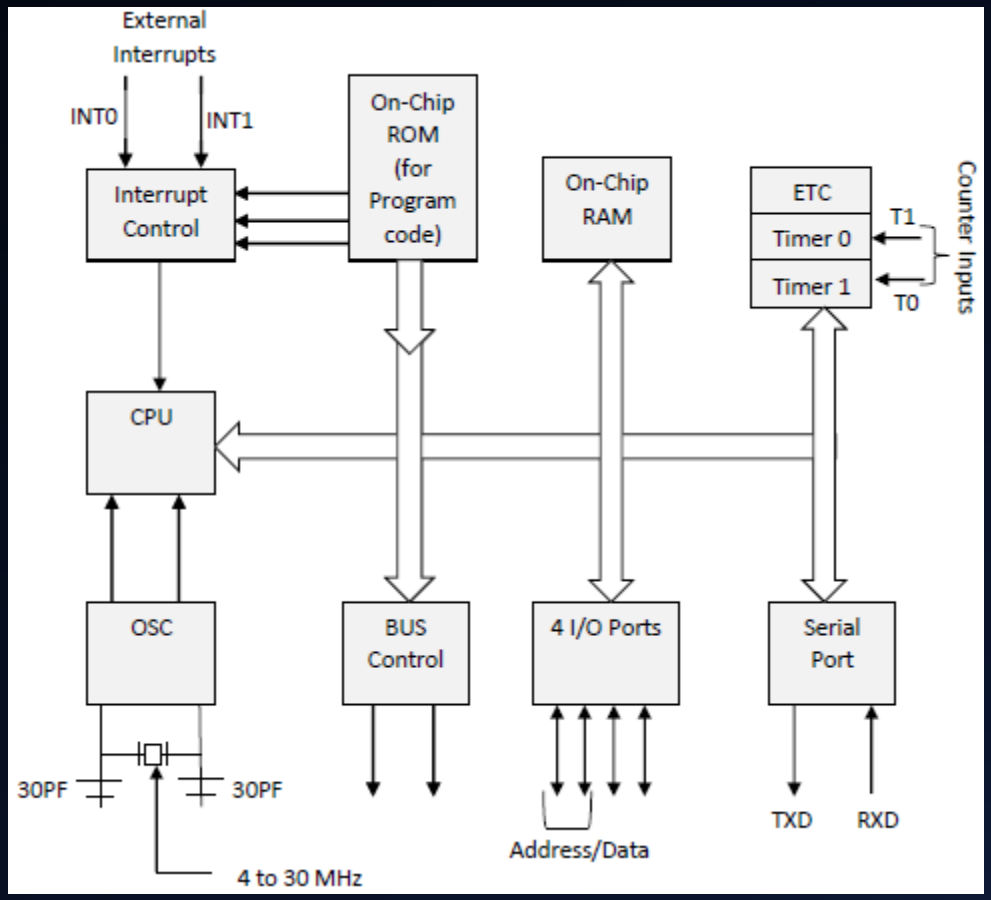
Intel 8051 (1980)



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

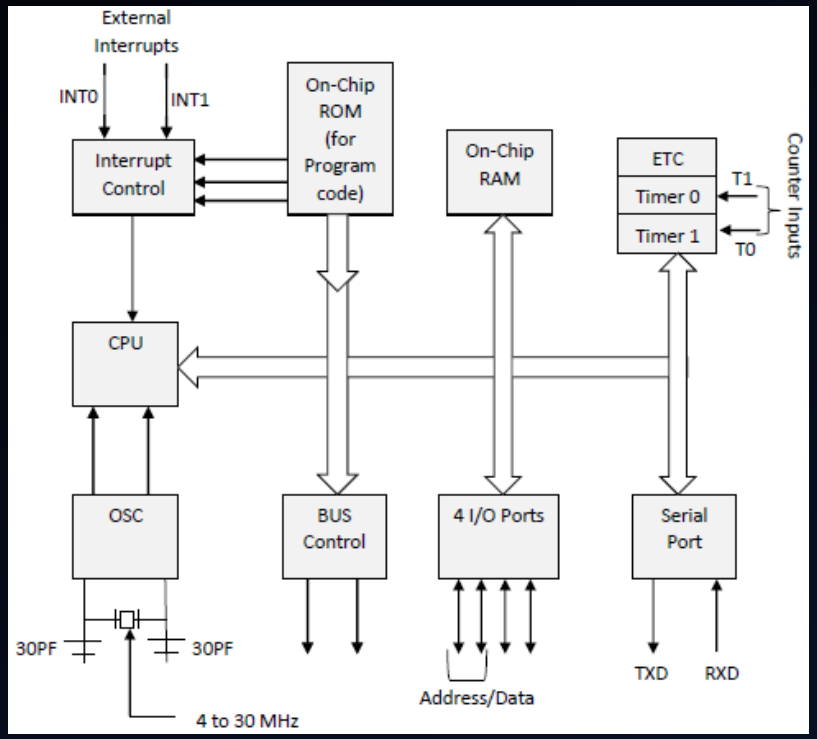
LINEAR/IMPERATIVE

- 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, …

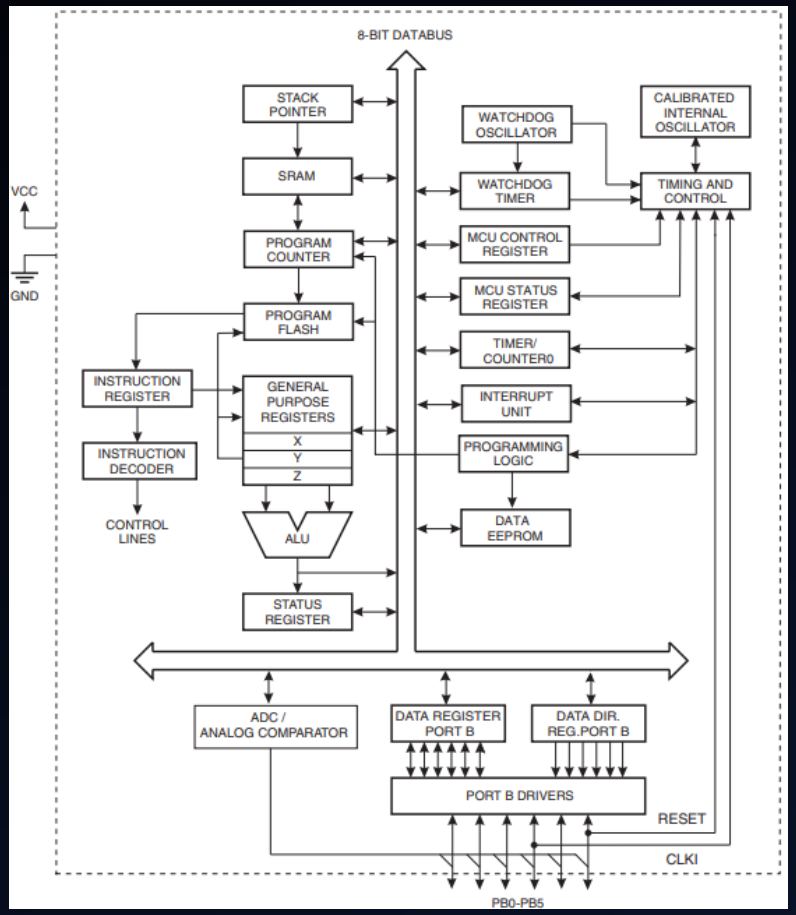
ATMEL Microcontrollers

- 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

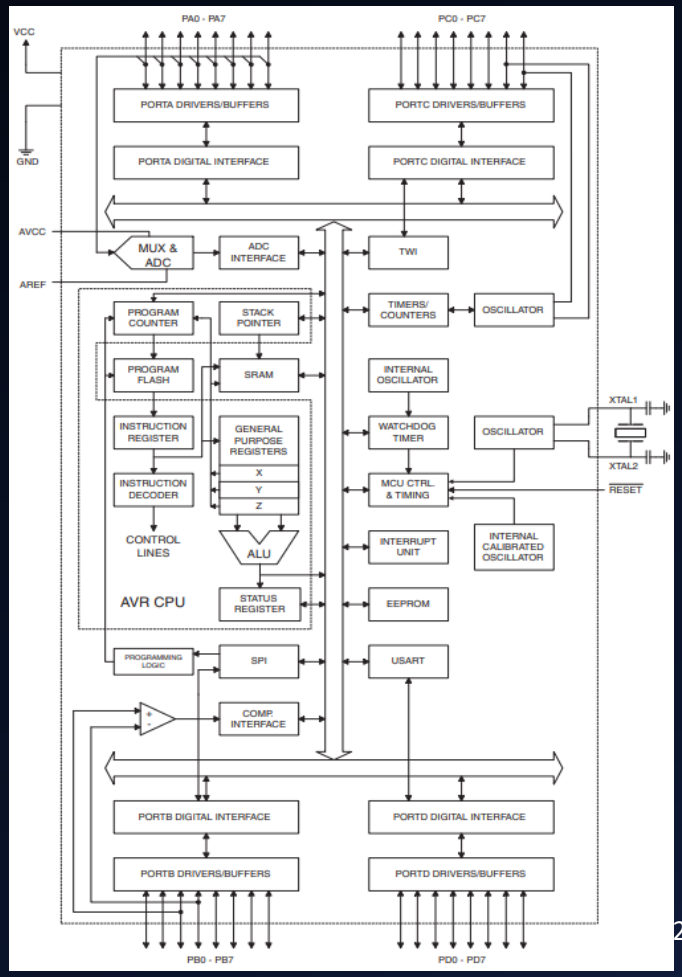
Peak Instruction throughput: 1MIPS @ 1 MHz

ATMEL Microcontrollers

ATTiny13

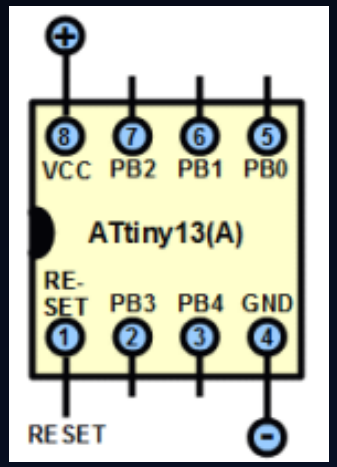


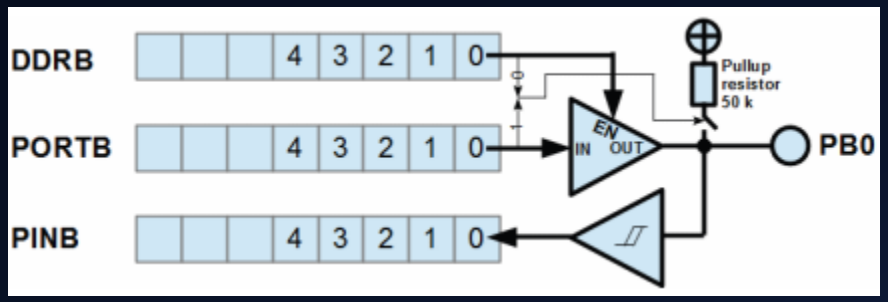
ATMega32



ATTiny13

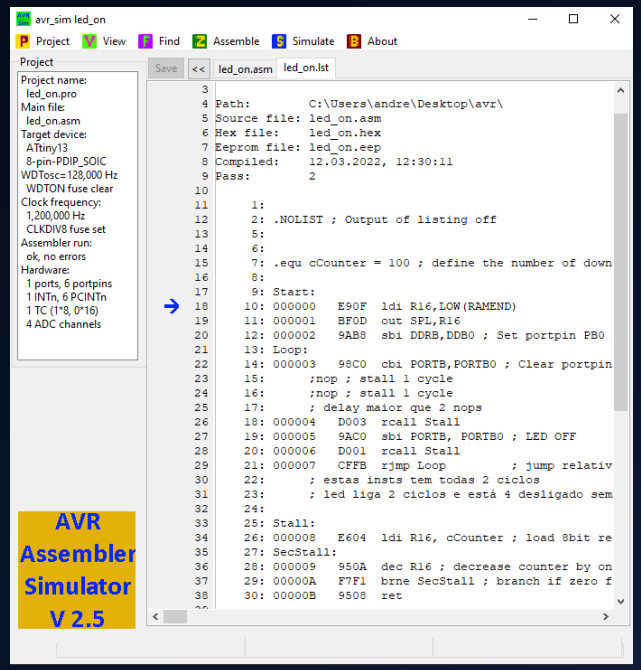
- 5 I/O addressable port pins: PB0 to PB4
  - Each controlled by a bit in DDRB and PORTB “registers”
  - DDRB (Data Direction Register port B): 1/0 -> output/input (extremely simplified, depends on breadboard setup)
  - PORTB (data output register port B): 1/0 -> high/low voltage
  - PINB (input register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

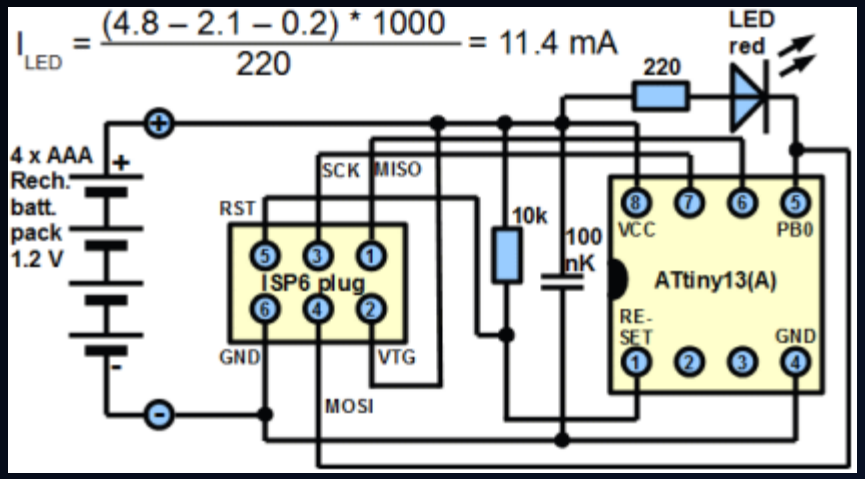


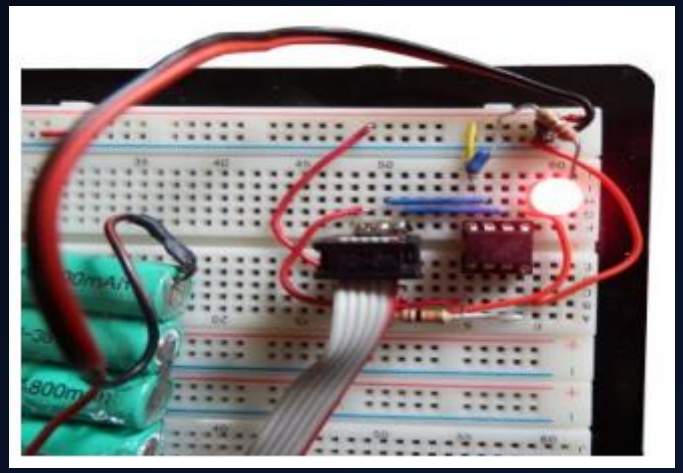
Practical Session

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



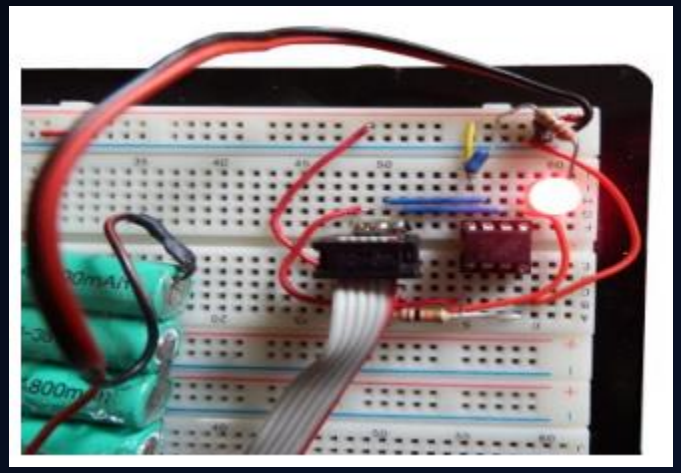


A Practical Example (1)

> `led_on_off.asm` file

```
.NOLIST ; Output of listing off
.INCLUDE "tn13def.inc" ; Read port definitions
.LIST ; Output of listing on

Start:
        ldi R16,LOW(RAMEND)
        out SPL,R16
        sbi DDRB,DDB0 ; Set portpin PB0 as output
Loop:
       cbi PORTB,PORTB0 ; Clear portpin PB0  LED ON - 2 cycles
       nop ; stall 1 cycle
       nop ; stall 1 cycle
       sbi PORTB, PORTB0 ; LED OFF - 2 cycles
       rjmp Loop         ; jump relative back to label
```

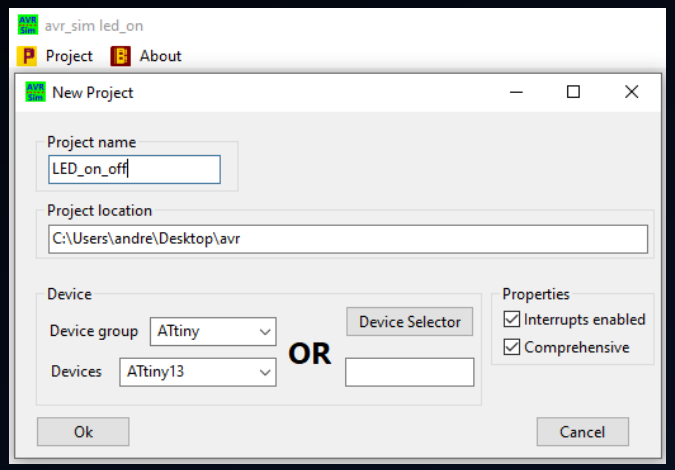


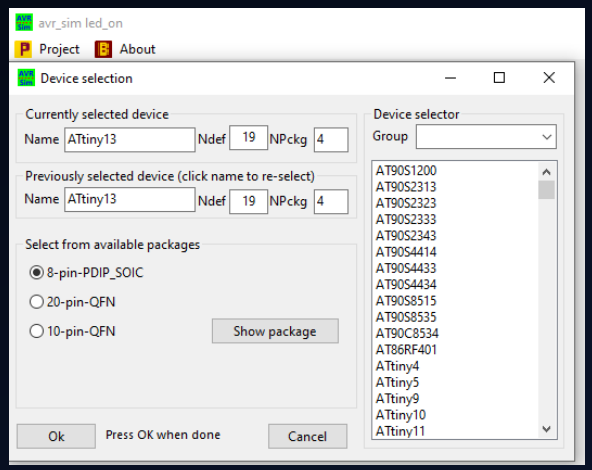
- How long does the LED stays on and off?
- Let’s simulate this!

A Practical Example (1)

> `led_on_off.asm` file

- Create a “New Project” in AVR_SIM



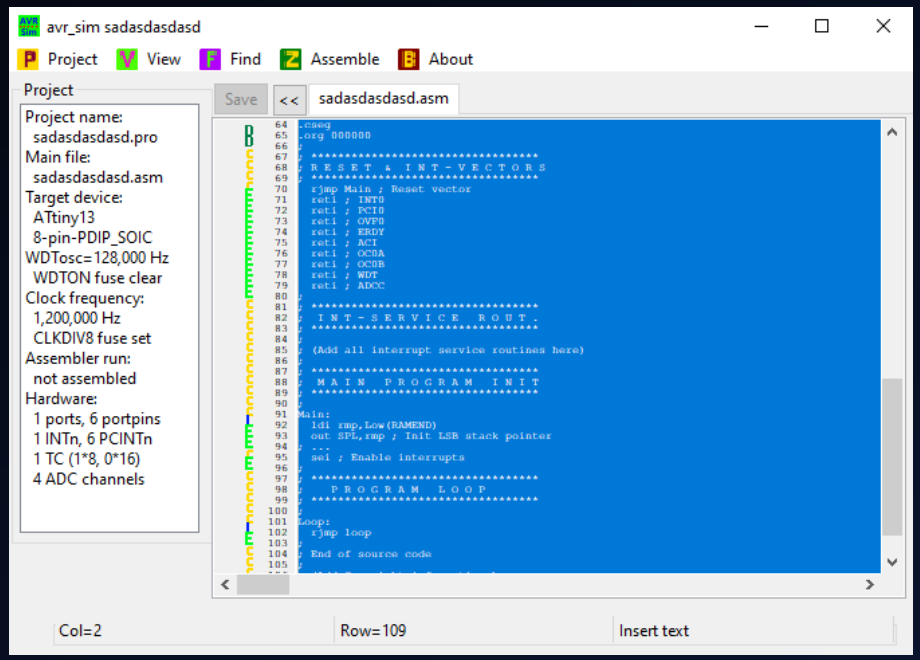


- Delete all default code and implement your own
- “Assemble” and then “Simulate”

A Practical Example (1)

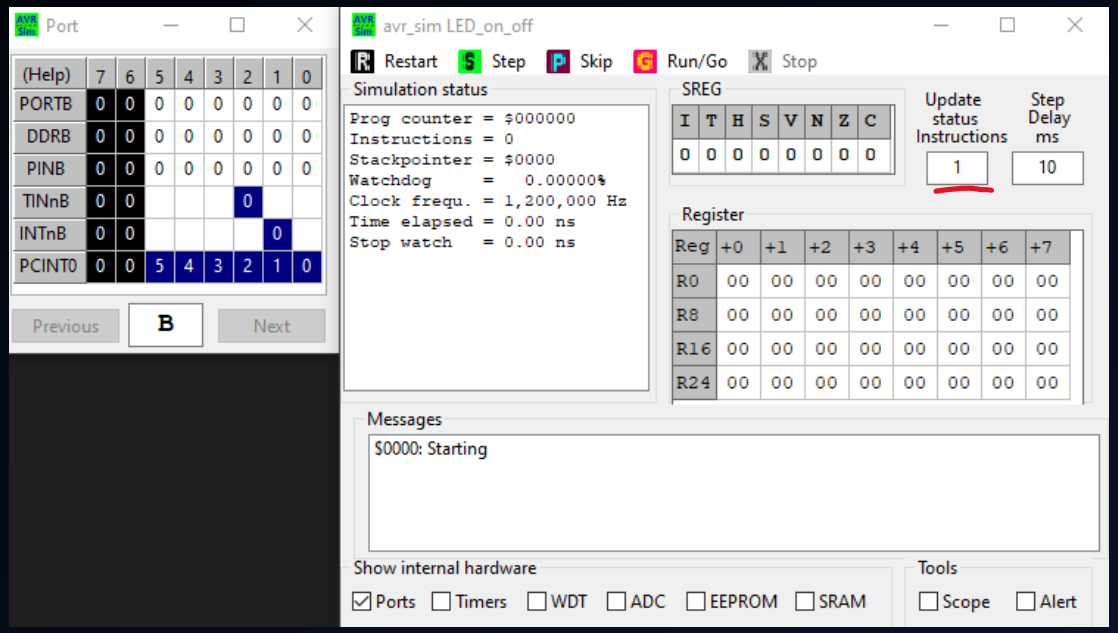
> `led_on_off.asm` file

- 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



- “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

```
Start:
        ldi R16,LOW(RAMEND)
        out SPL,R16
        sbi DDRB,DDB0 ; Set portpin PB0 as output
Loop:
       cbi PORTB,PORTB0 ; Clear portpin PB0  LED ON
       rcall Stall
       sbi PORTB, PORTB0 ; LED OFF
       rcall Stall
       rjmp Loop         ; jump relative back to label

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!

```
.equ smallStall = 20
.equ firstVal = 5
.equ secVal = 2

Start:
        ldi R16,LOW(RAMEND) ; fetch end of SRAM
        out SPL,R16         ; set stack pointer to that
        sbi DDRB,DDB0 ; Set portpin PB0 as output
        cbi DDRB,DDB1 ; Set portpin PB1 as intput
        sbi PORTB,PORTB0 ; LED OFF

        ldi R24,firstVal  ; load firstVal into the R24 register
        ldi R25,secVal

        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 & immediate

Programming Microcontrollers

THE END

March 2022

André Pereira