What will we do?

Why do we have programming languages?

Is programming language a computer language?

What compilers do you know?

Compilers, interpreters.

Keep a translator nearby to translate a paper each time.

Why compiling, why interpreting?

Pros and cons

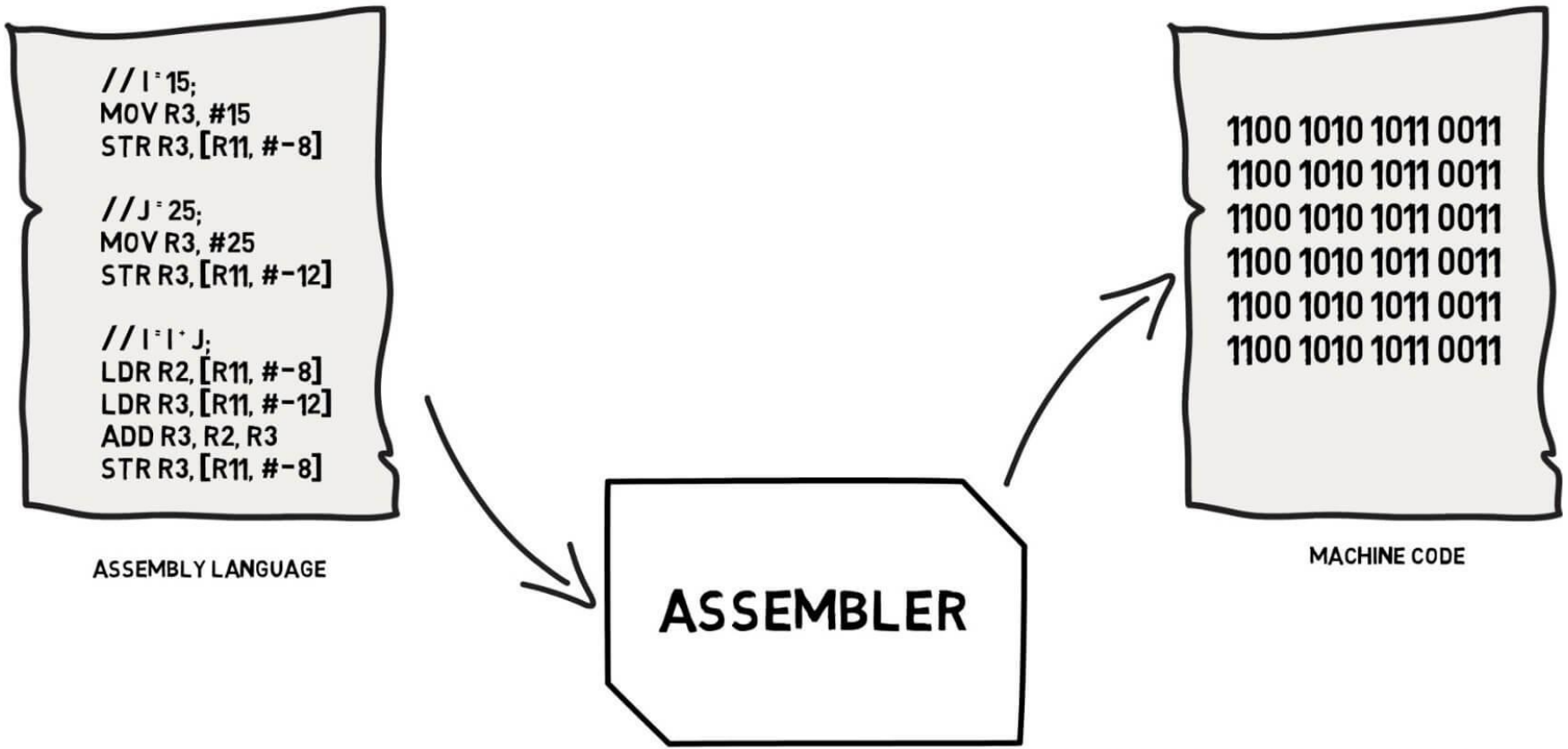
https://jaxenter.com/energy-efficient-programming-languages-137264.html

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Table 5. Pareto optimal sets for different combination of objectives.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

What is assembler?



What do we deal with in asm?
What do we deal with in PL?

Asm:

- registers

Asm:

- Registers
- Addresses

Asm:

- Registers
- Addresses
- CPU instructions (mov, add)

Asm:

- Registers
- Addresses
- CPU instructions (mov, add)

HL Programming languages

- Functions

HL Programming languages

- Functions
- Methods
-

HL Programming languages

- Functions
- Methods
- Variables
-

HL Programming languages

- Functions
- Methods
- Variables
- Data structures

HL Programming languages

- Functions
- Methods
- Variables
- Data structures
- loops

HL Programming languages

- Functions
- Methods
- Variables
- Data structures
- Loops
- modules

Why don’t we write in asm?
(or what do we write in asm?)

Std question:

Why learn other language if we can do everything in c++?

Are there languages, that cannot be compiled?

Who knows eval() function?

Cross compilers

JIT

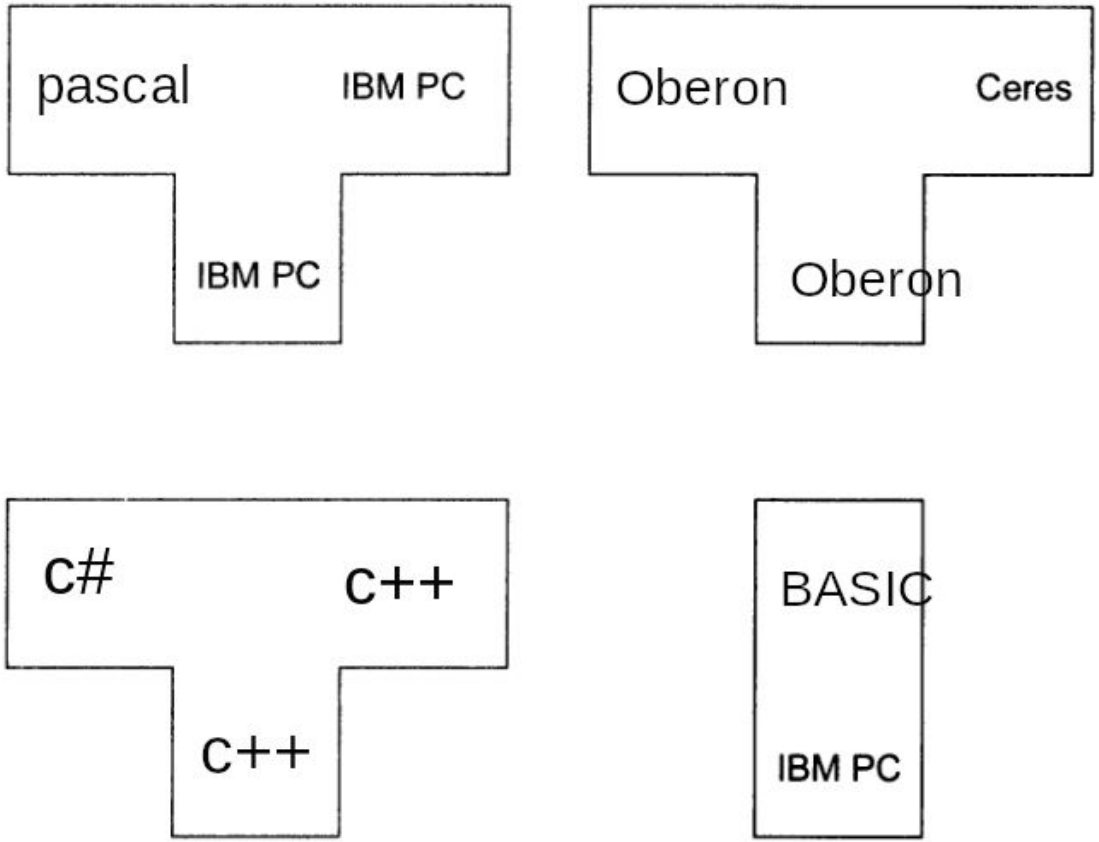
Binary compilers
(binary to binary translation)

X86 -> MIPS
6502 -> x86
68000 -> SPARC

Bratman’s T diagrams

Src lang Target lang

Instrumental lang


pascal
IBM PC
IBM PC
Oberon
Ceres
Oberon
C#
C++
C++
BASIC
IBM PC

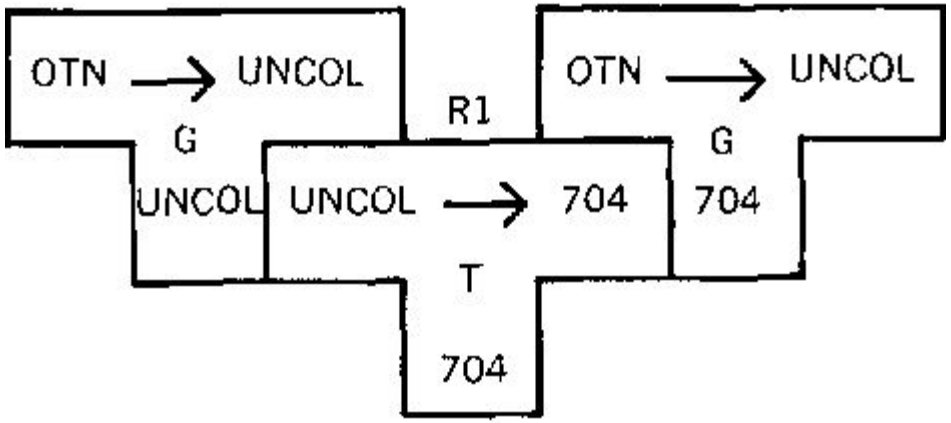
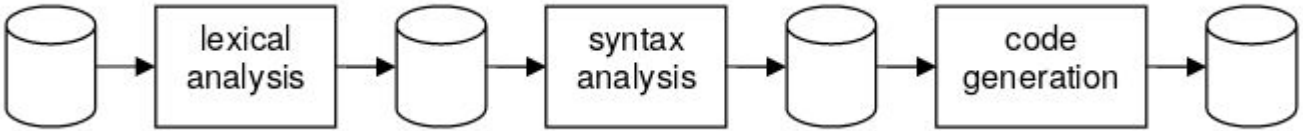


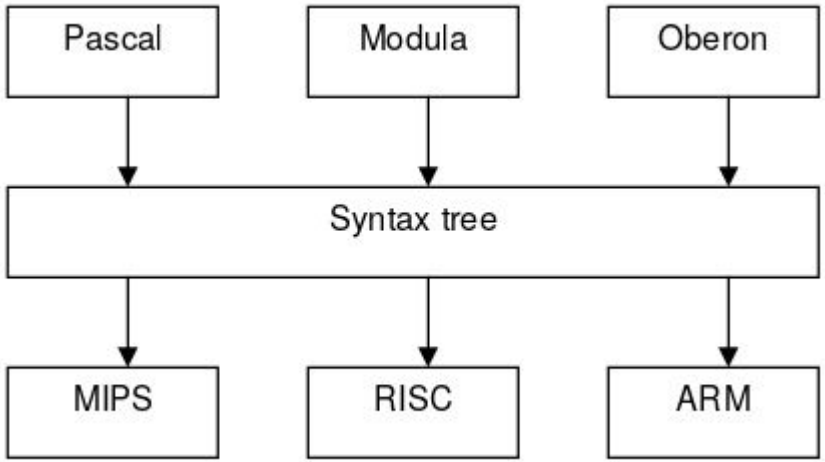
FIG. 2

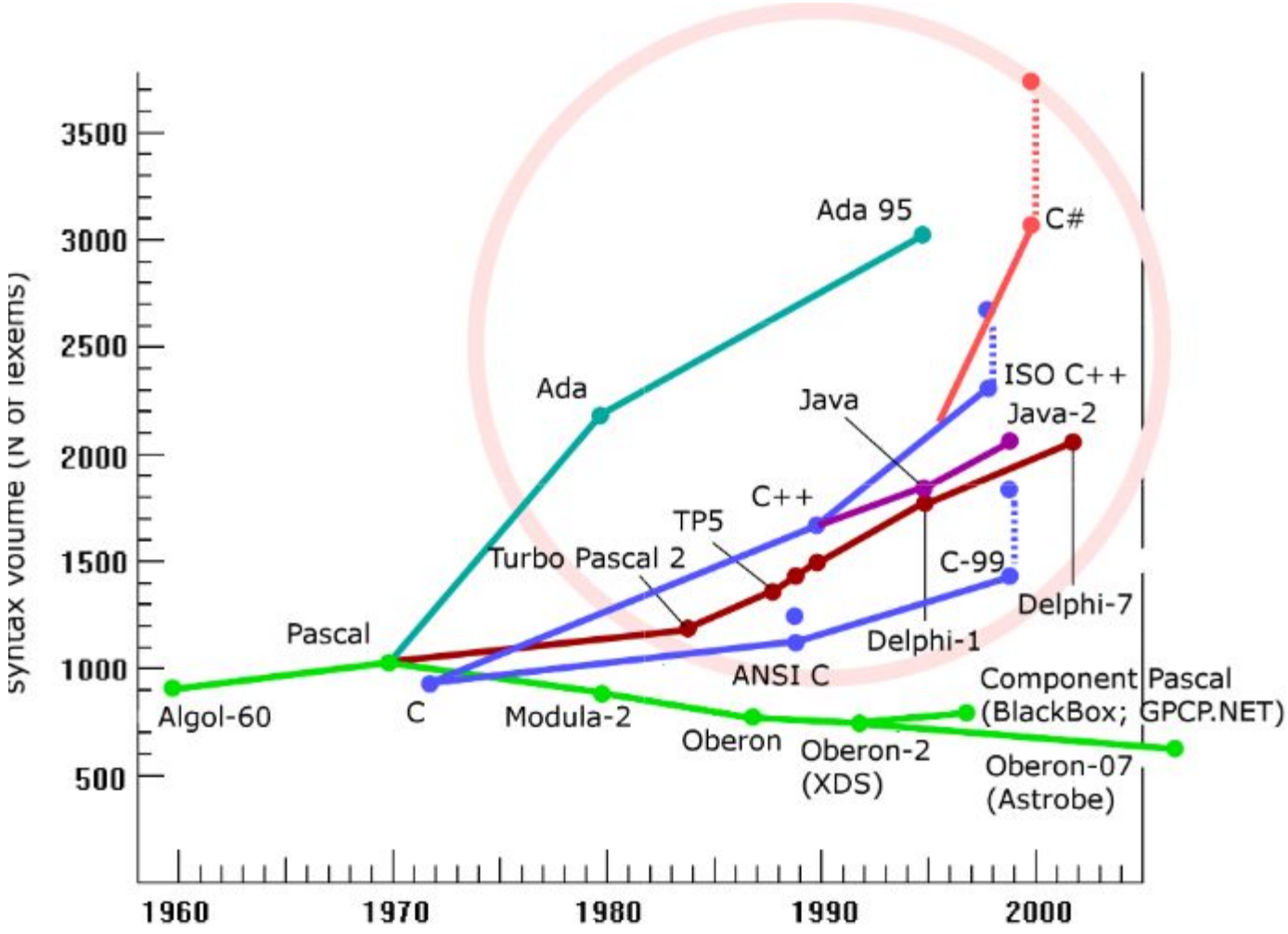
How first compilers were created?

What is bootstrapping?

Compilation process


lexical
analysis
syntax
analysis
code
generation


Pascal
Modula
Oberon
Syntax tree
MIPS
RISC
ARM


syntax volume (N of lexems)
3500
3000
2500
2000
1500
1000
500
1960
1970
1980
1990
2000
Ada 95
C#
Ada
ISO C++
Java
Java-2
C++
TP5
Turbo Pascal 2
C-99
Delphi-7
Pascal
Delphi-1
ANSI C
Component Pascal
(BlackBox; GPCP.NET)
Algol-60
C
Modula-2
Oberon
Oberon-2
(XDS)
Oberon-07
(Astrobe)

Metalanguages

Our spoken language is metalanguage.

Our spoken language is metalanguage.

Language and syntax

sentence = subject predicate.

```
sentence = subject predicate.

subject = "John" | "Mary".
predicate = "eats" | "talks".
```

sentence = subject predicate.

subject = "John" | "Mary".
predicate = "eats" | "talks".

John eats Mary eats
John talks Mary talks

$S = AB.$

S = AB.

A = “a” | “b”.
B = “c” | “d”.

S = AB.

A = “a” | “b”.
B = “c” | “d”.

L = {ac, ad, bc, bd}

S = AB.

A = “a” | “b”.
B = “c” | “d”.

L = {ac, ad, bc, bd}

Language contains four sentences.
Typically, language contains infinitely many sentences.

Infinite set may be defined with finite number of equations.
∅ - empty sequence.

$$S = A.$$

Infinite set may be defined with finite number of equations.
∅ - empty sequence.

$$S = A.$$

$$A = \text{“a”}\ A \mid \varnothing$$

Infinite set may be defined with finite number of equations.
∅ - empty sequence.

$$S = A.$$

$$A = \text{“a”}\ A \mid \varnothing$$

$$L = \{\varnothing, a, aa, aaa, aaaa...\}$$

Nested sequences.

S = A.

Nested sequences.

S = A.

A = “a” A “c” | “b”.

Nested sequences.

S = A.

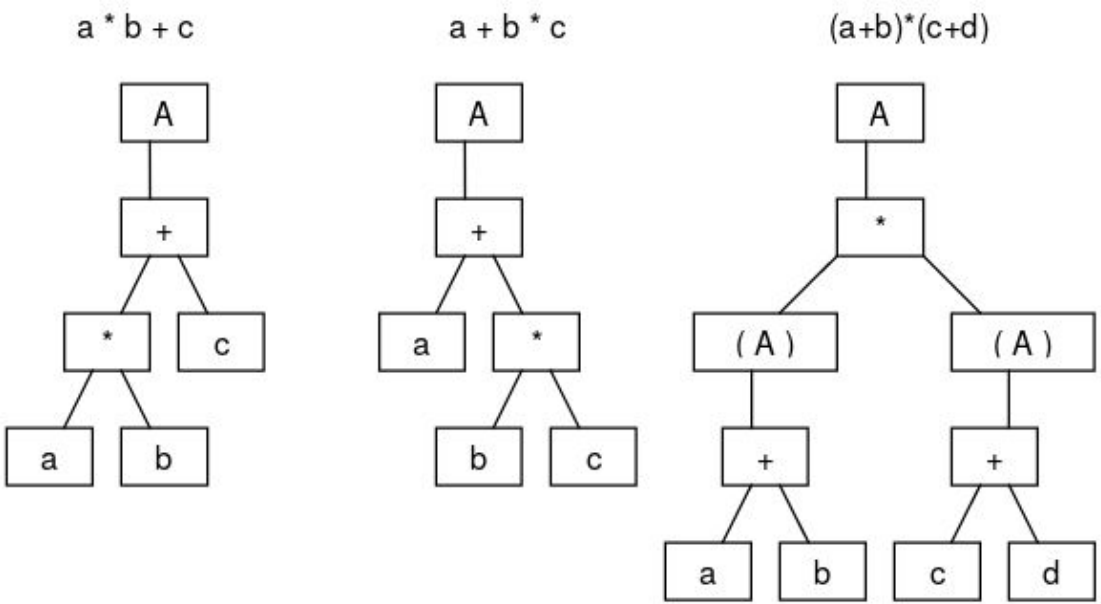
A = “a” A “c” | “b”.

L = {b, abc, aabcc, aaabccc, … }

Structure of expressions
E: expression; T: term; F: factor: V: variable;

E = T | A “+” T.
T = F | T “*” F.
F = V | “(“ E “)”.
V = “a” | “b” | “c” | “d”.

Syntax tree



Language is defined by:

Language is defined by:

- Set of terminal symbols.

Language is defined by:

- Set of terminal symbols.
- Set of nonterminal symbols.

Language is defined by:

- Set of terminal symbols.
- Set of nonterminal symbols.
- Set of syntactic equations (productions)

Language is defined by:

- Set of terminal symbols.
- Set of nonterminal symbols.
- Set of syntactic equations (productions)
- Start symbol

Backus Naur form, BNF, 1960.

Syntax = production syntax | ∅.
production = identifier "=" expression "." .
expression = term | expression "|" term.
Term = factor | term factor.
Factor = identifier | string.
Identifier = letter | identifier letter | identifier digit.
string = stringhead """.
stringhead = """ | stringhead character.
Letter = "A" | ... | "Z".
Digit = "0" | ... | "9".

Extended Backus Naur form, EBNF, 1977. By N. Wirth.

```
syntax = {production}.
production = identifier "=" expression "." .
expression = term {"|" term}.
term = factor {factor}.
Factor = identifier | string | "(" expression ")
" | "[" expression "]" | "{" expression "}".
Identifier = letter {letter | digit}.
String = """ {character} """.
Letter = "A" | ... | "Z".
Digit = "0" | ... | "9".
```

{x} is equivalent to an arbitrarily long sequence of x, therefore

{x} is equivalent to an arbitrarily long sequence of x, therefore

$$A = A\ B \mid \varnothing$$

{x} is equivalent to an arbitrarily long sequence of x, therefore

$$A = A\ B \mid \varnothing$$

Can be formulated as

$$A = \{B\}.$$

A factor of the form [x] is equivalent to “x or nothing”.

A factor of the form [x] is equivalent to “x or nothing”.

Need for the special symbol ∅ vanishes.

US postal address in BNF

<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
| <personal-part> <name-part>

<personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
<opt-apt-num> ::= <apt-num> | ""

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit                = "0" | digit excluding zero ;
```

Context free languages are languages that are describable by a context free grammar. So the question reduces to :
What is a context free grammar?

A language is a set of strings, where each string is built up from a base set of allowable symbols, the alphabet (Σ). Each such string is called a sentence of the language. This includes the empty sentence of no symbols at all.

For most languages, not just any sequences of symbols form a sentence, but there are certain structural rules that define the allowable sentences. These structural rules, taken together, form what is known as the grammar of the language.

For most languages, not just any sequences of symbols form a sentence, but there are certain structural rules that define the allowable sentences. These structural rules, taken together, form what is known as the grammar of the language.

It is important here to realize that, while a specific grammar, uniquely determines a language, the converse is not true in general. A given language, might have many different grammars that describe it.

For most languages, not just any sequences of symbols form a sentence, but there are certain structural rules that define the allowable sentences. These structural rules, taken together, form what is known as the grammar of the language.

It is important here to realize that, while a specific grammar, uniquely determines a language, the converse is not true in general. A given language, might have many different grammars that describe it.

Note also, that nowhere in the above have I mentioned any concept of meaning for the sentences of the language. Formal language theory does not address meaning, only structure.

Note also, that nowhere in the above have I mentioned any concept of meaning for the sentences of the language. Formal language theory does not address meaning, only structure.

For example, on this view, "*The invisible, pink unicorn ate wrathful balls, after sleeping diligently*" is a valid English sentence, although perhaps meaningless, because it respects the grammar of the English language.

Note also, that nowhere in the above have I mentioned any concept of meaning for the sentences of the language. Formal language theory does not address meaning, only structure.

For example, on this view, "*The invisible, pink unicorn ate wrathful balls, after sleeping diligently*" is a valid English sentence, although perhaps meaningless, because it respects the grammar of the English language.

<statement> = <subject> <action verb> <indirect object>
<object> | <subject> <linking verb> <adjective>
<subject> = Ann | Bob
<verb> = throws | gives | takes
<indirect object> = Ann | Bob | him | her
<object> = the ball | the sock
<linking verb> = is
<adjective> = female | male | tired | sleepy

<statement> = <subject> <action verb> <indirect object>
<object> | <subject> <linking verb> <adjective>
<subject> = Ann | Bob
<verb> = throws | gives | takes
<indirect object> = Ann | Bob | him | her
<object> = the ball | the sock
<linking verb> = is
<adjective> = female | male | tired | sleepy

This looks like a possible road to representing language, but it has a problem, which is expressed by the term “context-free”. That is, you can’t build a finite context-free language which can hold certain types of relations between things. We can’t build a language where, say, saying “Ann is female.” would imply that in order to speak of Ann as an indirect object later on, we have to use “her”.

CFG - context free grammar

a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements.

Recoursive descent parser

- Before the parsing starts, first symbol of the entity we parse is the current

Recoursive descent parser

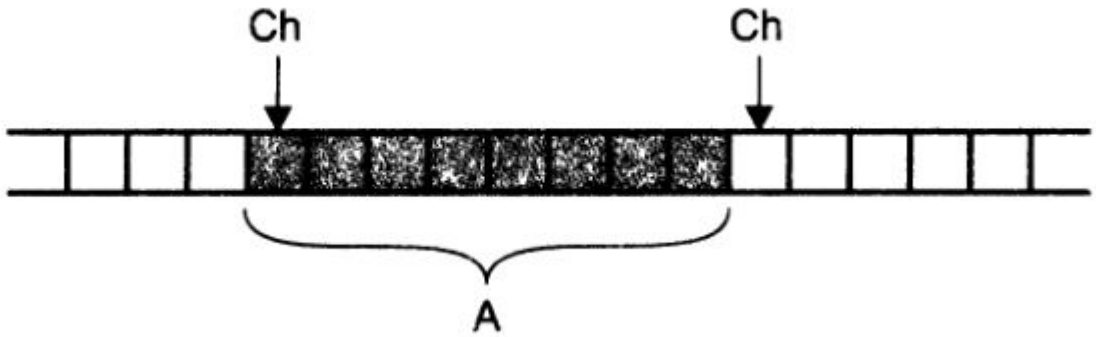
- Before the parsing starts, first symbol of the entity we parse is the current
- Function reads all symbols of the current non-terminal, or emits error

Recoursive descent parser

- Before the parsing starts, first symbol of the entity we parse is the current
- Function reads all symbols of the current non-terminal, or emits error
- If rules for this non-terminal contain other non-terminals in right side of equation, then function calls functions to call those non-terminals

Recoursive descent parser

- Before the parsing starts, first symbol of the entity we parse is the current
- Function reads all symbols of the current non-terminal, or emits error
- If rules for this non-terminal contain other non-terminals in right side of equation, then function calls functions to call those non-terminals
- Then first symbol of the next construct is current.



Current symbol at the start and the end of the parsing function of non-terminal A.

therefore

`NextCh(); - read the first symbol in to variable ch.`

therefore

NextCh(); - read the first symbol in to variable ch.

S(); - function of parsing a beginning of non-terminal;

therefore

NextCh(); - read the first symbol in to variable ch.

S(); - function of parsing a beginning of non-terminal;

If ch # eot then begin error() end;

Regular languages

Regular languages

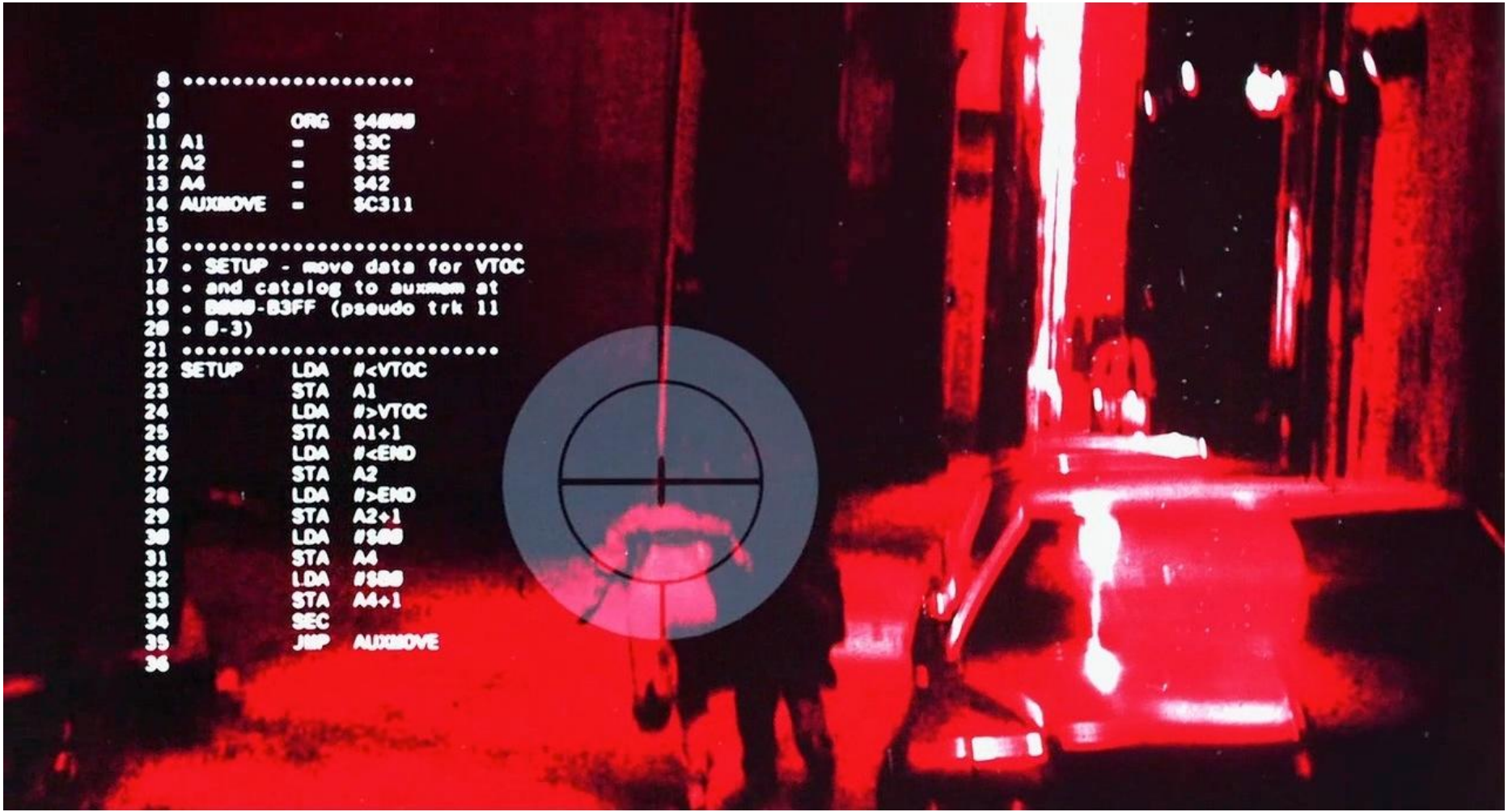
Syntactic equations of the form defined in EBNF generate context-free languages.

Regular languages

Syntactic equations of the form defined in EBNF generate context-free languages.

Substitution of the symbol left of = by a sequence derived from the expression to the right of = is always permitted, regardless of the context in which the symbol is embedded within the sentence.

- Machine language
- Assembly language
- High level language



```
 8 ....................
 9
10            ORG   $4000
11 A1         =     $3C
12 A2         =     $3E
13 A4         =     $42
14 AUXMOVE    =     $C311
15
16 ............................
17 * SETUP - move data for VTOC
18 * and catalog to auxmem at
19 * D000-D3FF (pseudo trk 11
20 * 0-3)
21 ............................
22 SETUP      LDA   #<VTOC
23            STA   A1
24            LDA   #>VTOC
25            STA   A1+1
26            LDA   #<END
27            STA   A2
28            LDA   #>END
29            STA   A2+1
30            LDA   #$D0
31            STA   A4
32            LDA   #$D0
33            STA   A4+1
34            SEC
35            JMP   AUXMOVE
36
```

https://www.masswerk.at/6502/6502_instruction_set.html

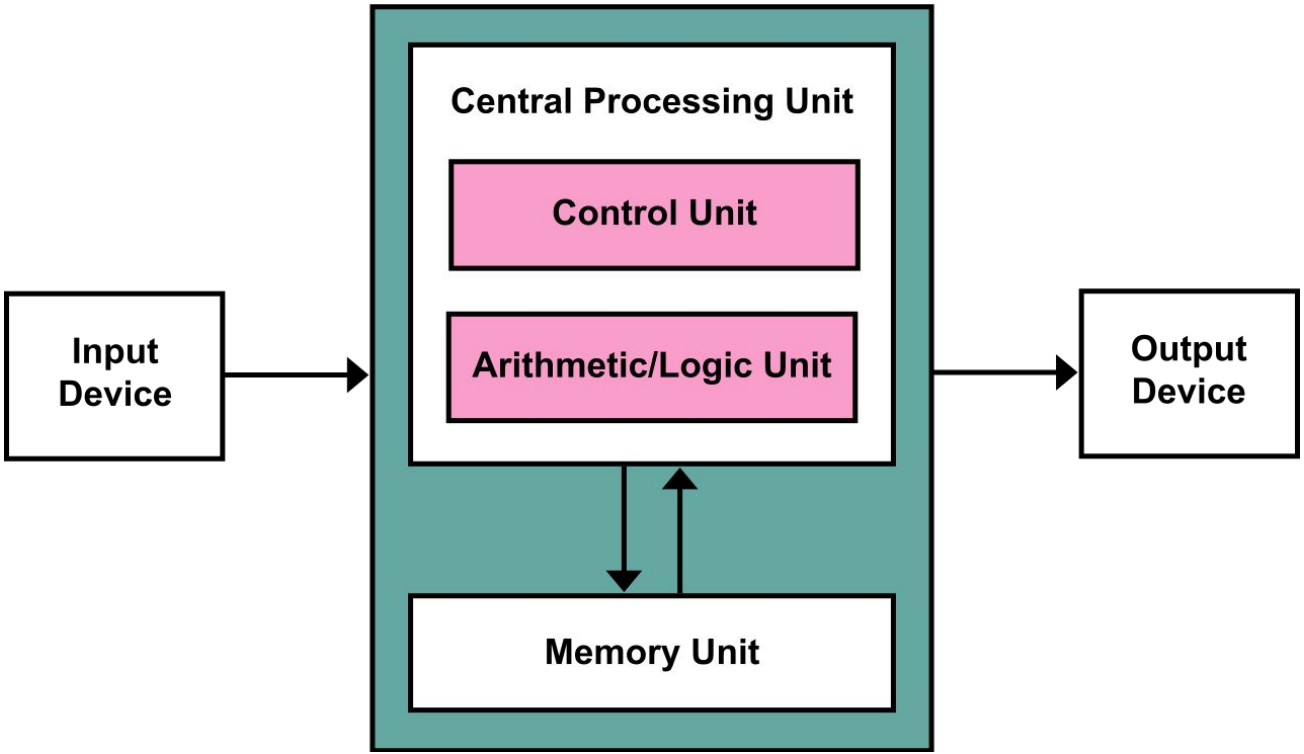
Machine Language vs. Assembly Language

Objective: Multiply the value stored in R4 by 12

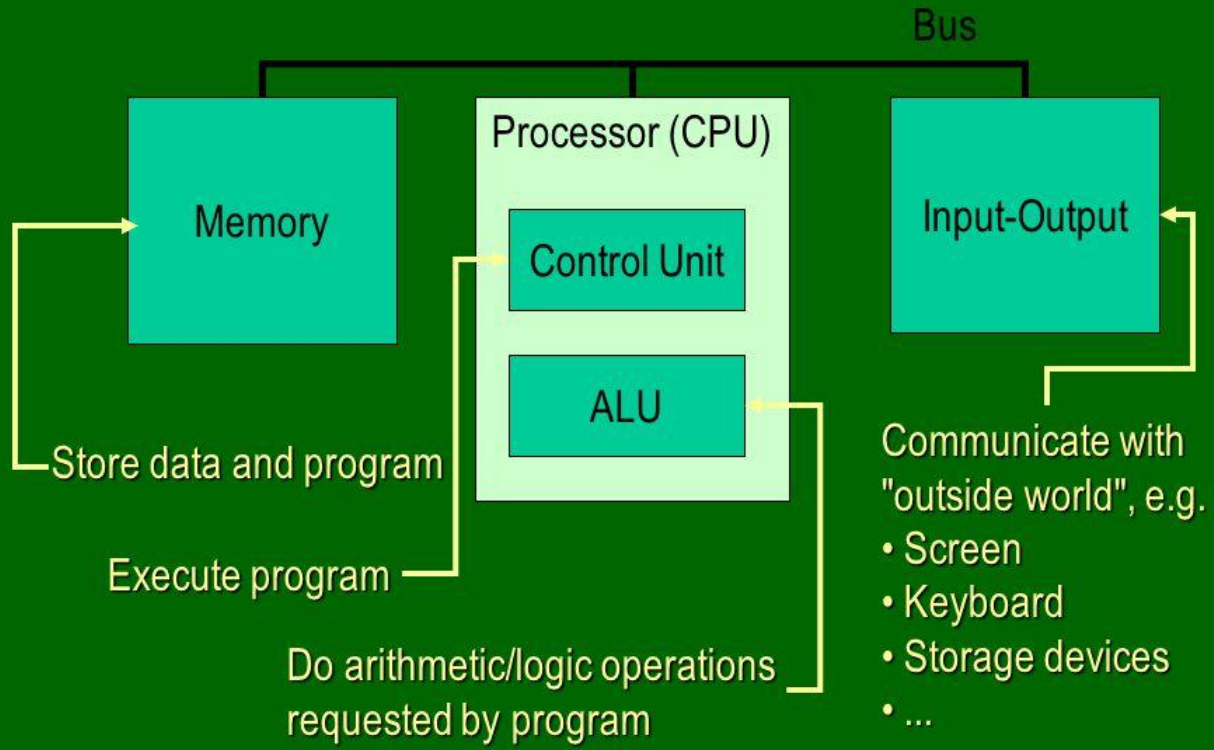
Machine Language	Assembly Language
0011 0000 0000 0000	.ORIG x3000
0101 000 000 1 00000	AND R0, R0, #0
0101 111 111 1 00000	AND R7, R7, #0
0001 000 000 1 01100	ADD R0, R0, #12
0000 110 000000011	TEST BRnz DONE
0001 111 111 000 100	ADD R7, R7, R4
0001 000 000 1 11111	ADD R0, R0, #-1
0000 111 111111100	BRnzp TEST
1111 0000 0010 0101	DONE HALT
	.END

Von Neumann architecture

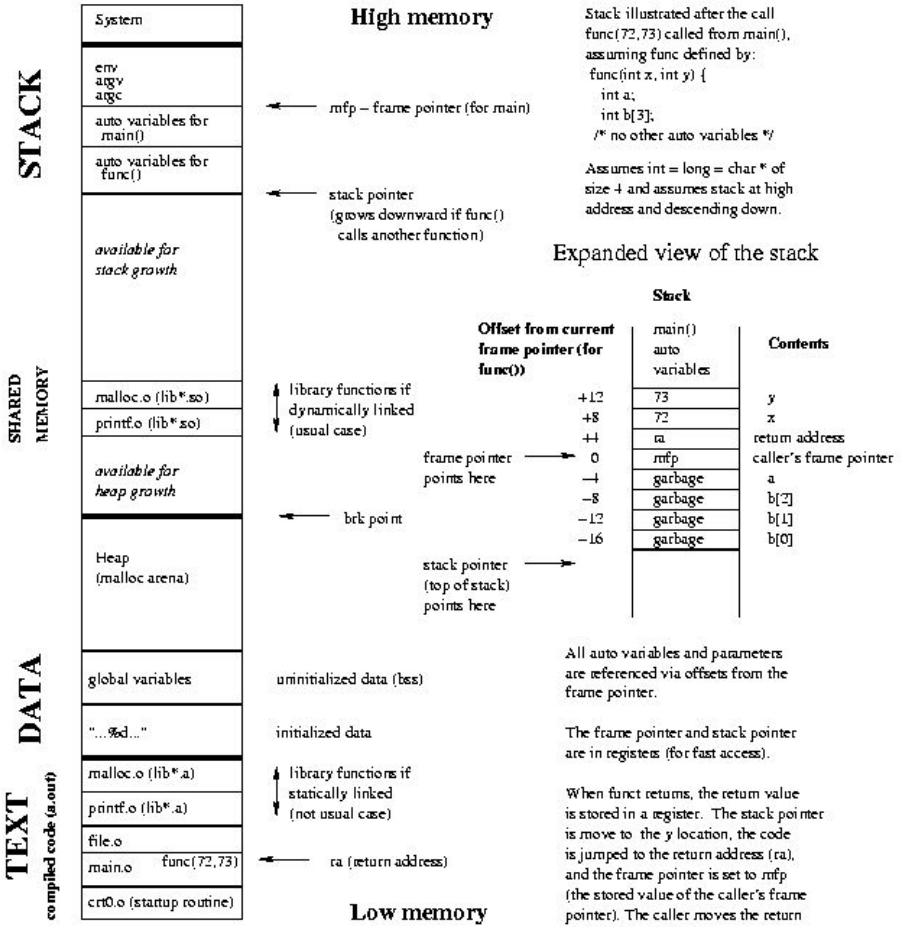
Von Neumann architecture

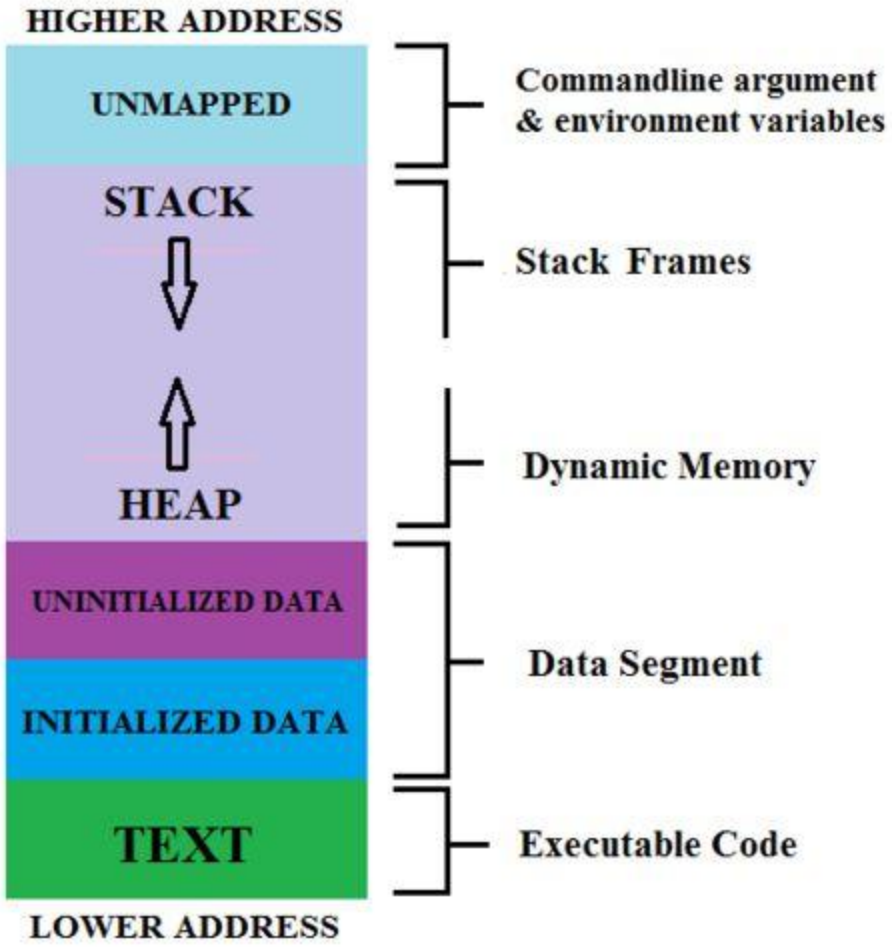


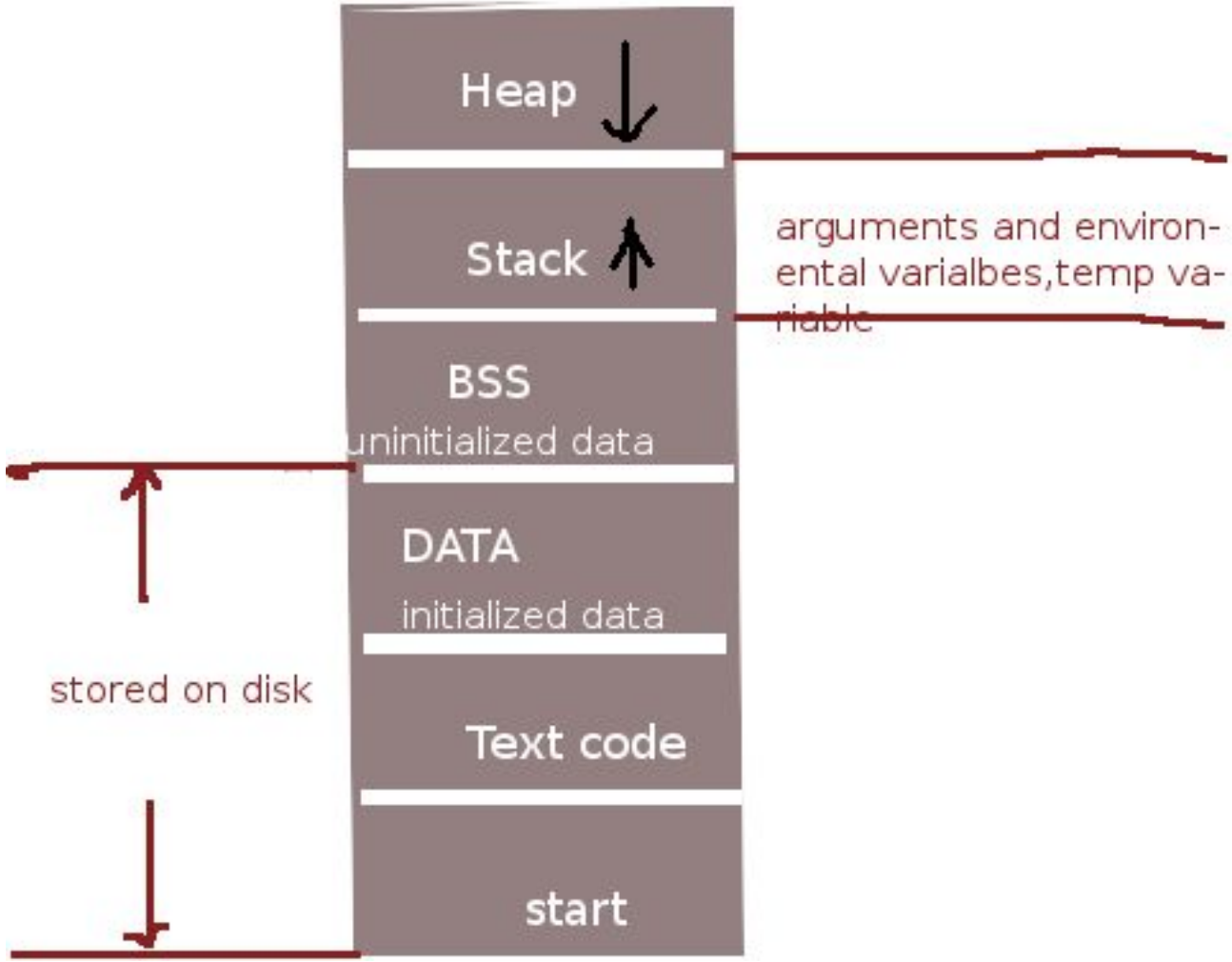
The Von Neumann Architecture



Memory Layout (Virtual address space of a C process)




HIGHER ADDRESS
UNMAPPED
STACK
HEAP
UNINITIALIZED DATA
INITIALIZED DATA
TEXT
LOWER ADDRESS
Commandline argument
& environment variables
Stack Frames
Dynamic Memory
Data Segment
Executable Code


Heap
Stack
arguments and environ-
ental varialbes,temp va-
riable
BSS
uninitialized data
DATA
initialized data
stored on disk
Text code
start

Von Neumann architecture

- In case of von Neumann architecture, instructions and data are stored in the same memory, so instructions are fetched over the same data path used to fetch data.

Von Neumann architecture

- In case of von Neumann architecture, instructions and data are stored in the same memory, so instructions are fetched over the same data path used to fetch data.
- This means that a CPU cannot simultaneously read an instruction and read or write data from or to the memory.

Von Neumann architecture

- In case of von Neumann architecture, instructions and data are stored in the same memory, so instructions are fetched over the same data path used to fetch data.
- This means that a CPU cannot simultaneously read an instruction and read or write data from or to the memory.

Harvard architecture

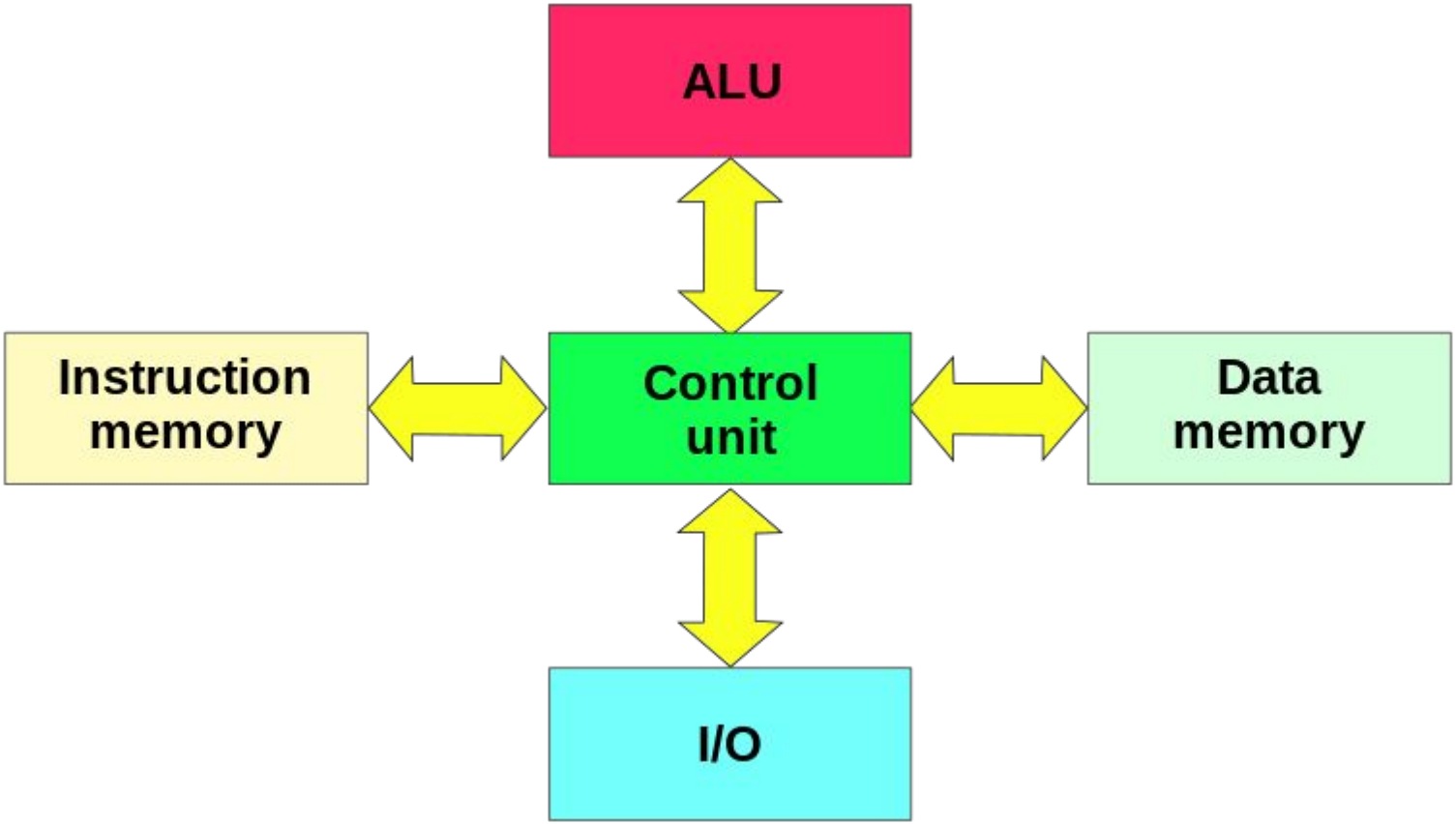
- In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache.

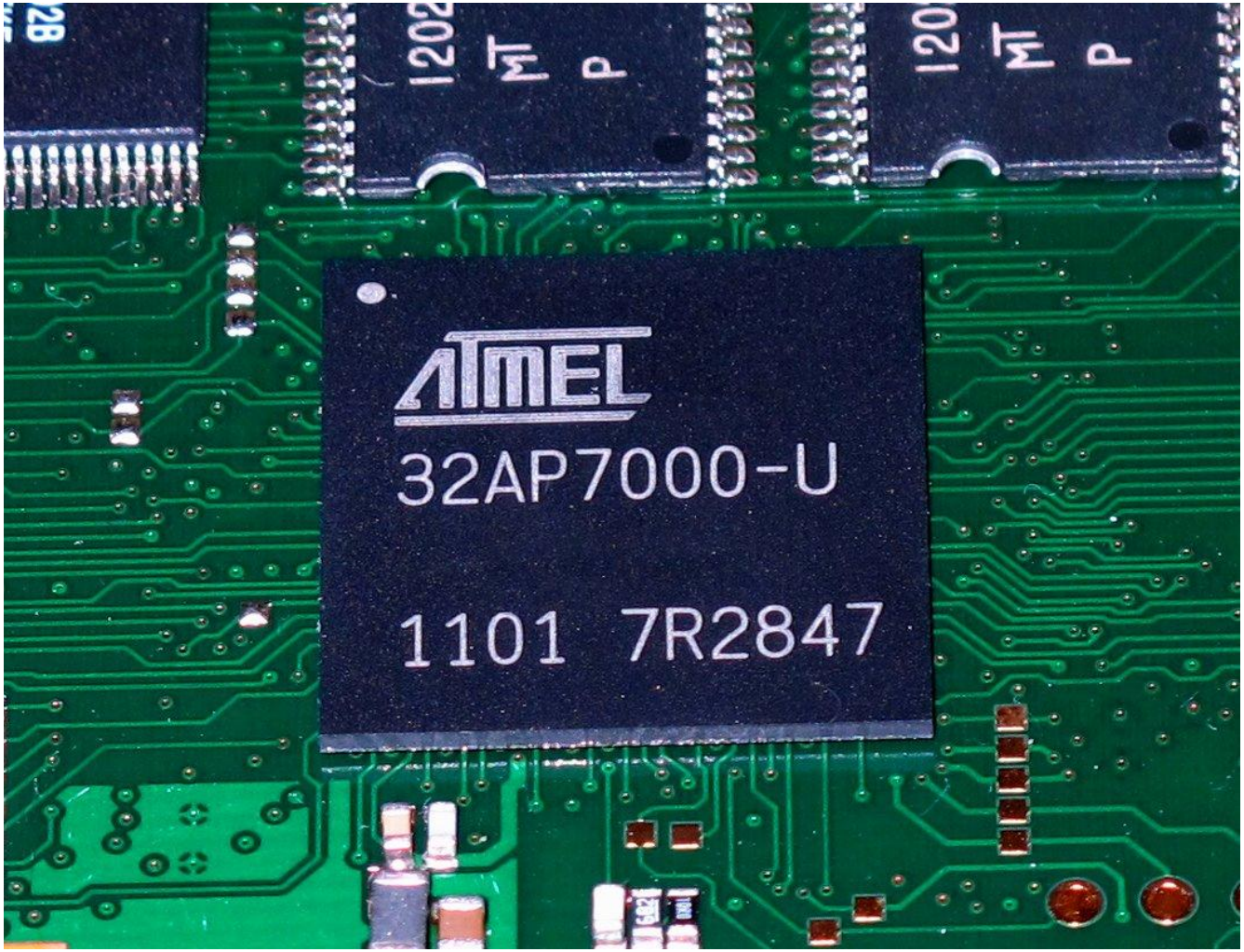
Harvard architecture

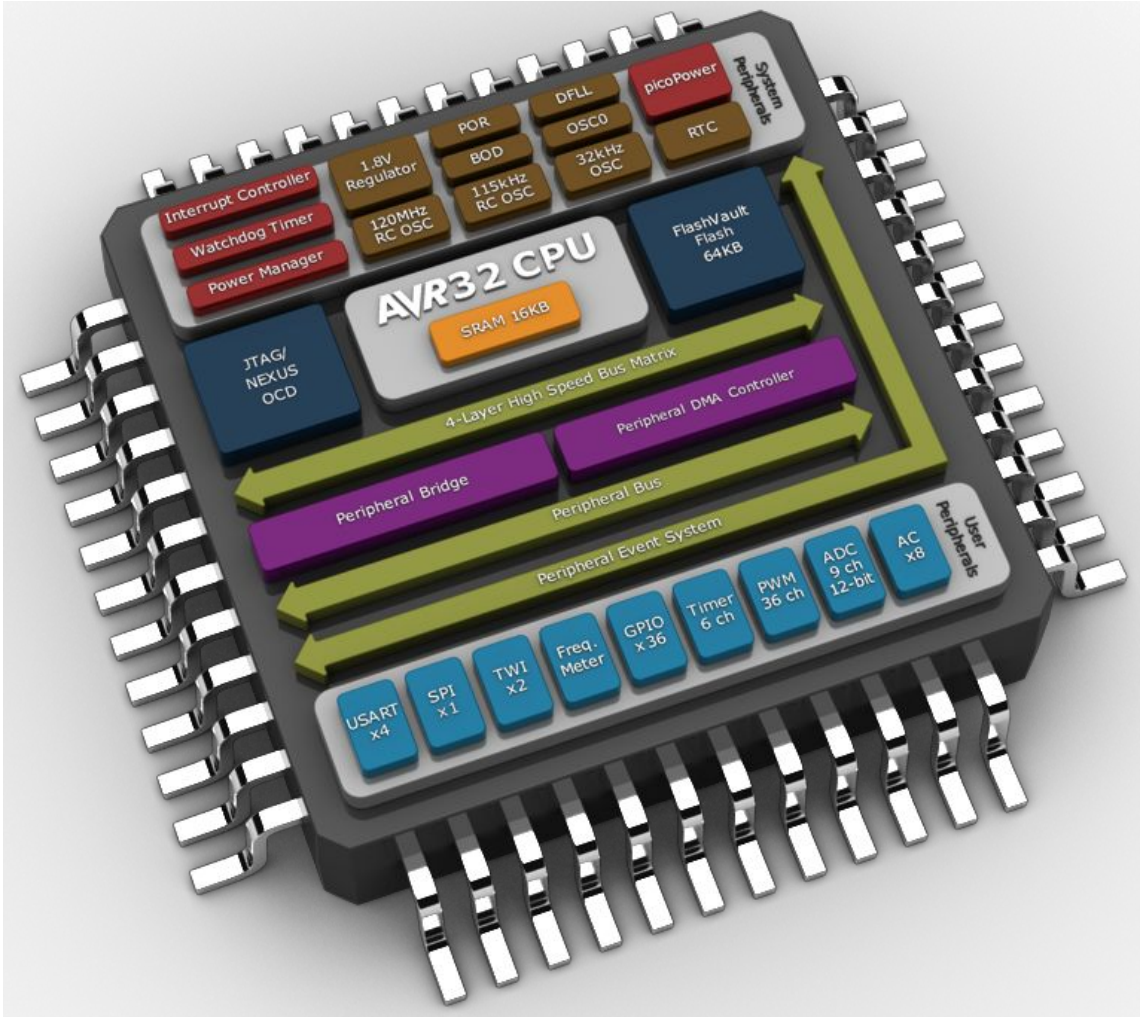
- In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache.
- A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

Harvard architecture

- In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache.
- A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.
- has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight-bit byte


ALU
Instruction memory
Control unit
Data memory
I/O


ATMEL
32AP7000-U
1101 7R2847


System Peripherals
picoPower
RTC
DFLL
OSC0
32kHz OSC
POR
BOD
115kHz RC OSC
1.8V Regulator
120MHz RC OSC
Interrupt Controller
Watchdog Timer
Power Manager
AVR32 CPU
SRAM 16KB
FlashVault Flash 64KB
JTAG/ NEXUS OCD
4-Layer High Speed Bus Matrix
Peripheral DMA Controller
Peripheral Bridge
Peripheral Bus
Peripheral Event System
User Peripherals
AC x8
ADC 9 ch 12-bit
PWM 36 ch
Timer 6 ch
GPIO x36
Freq. Meter
TWI x2
SPI x1
USART x4

Von Neumann architecture

Two main parts

- CPU

Von Neumann architecture

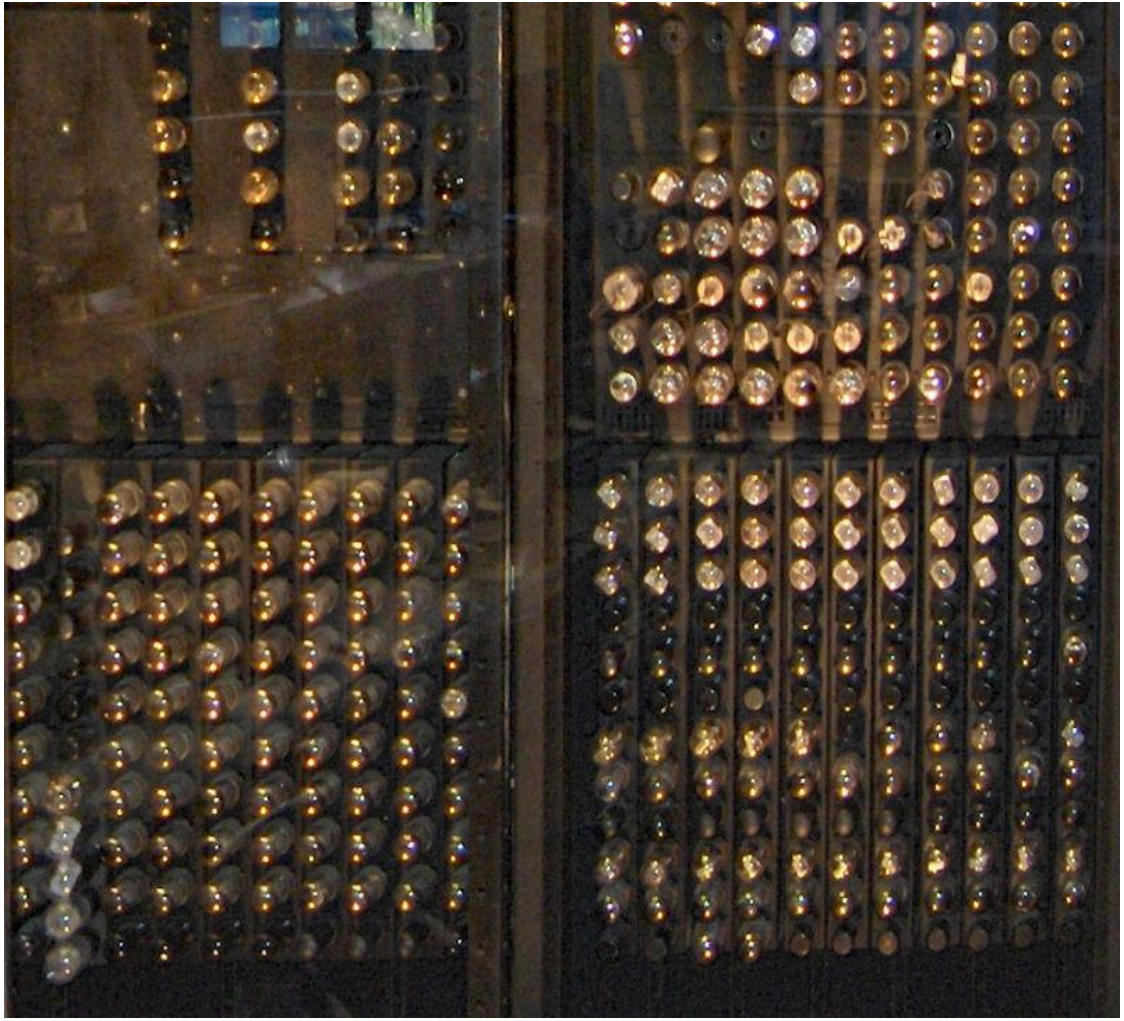
Two main parts

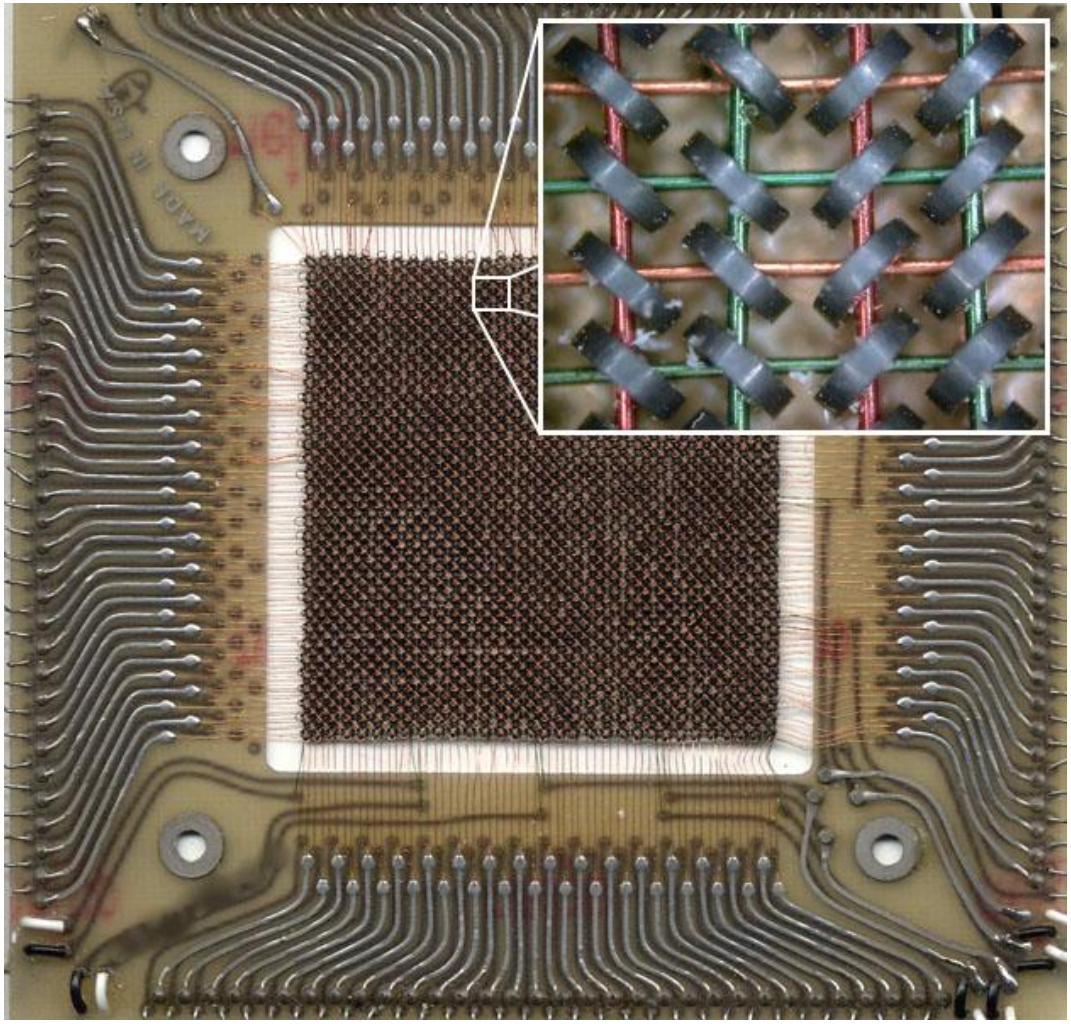
- CPU
- Memory

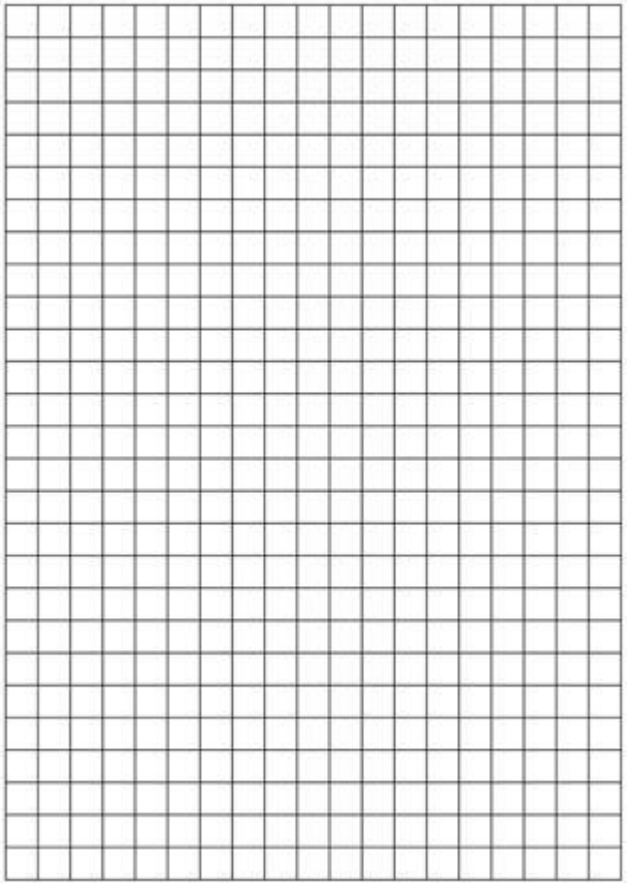
Von Neumann architecture

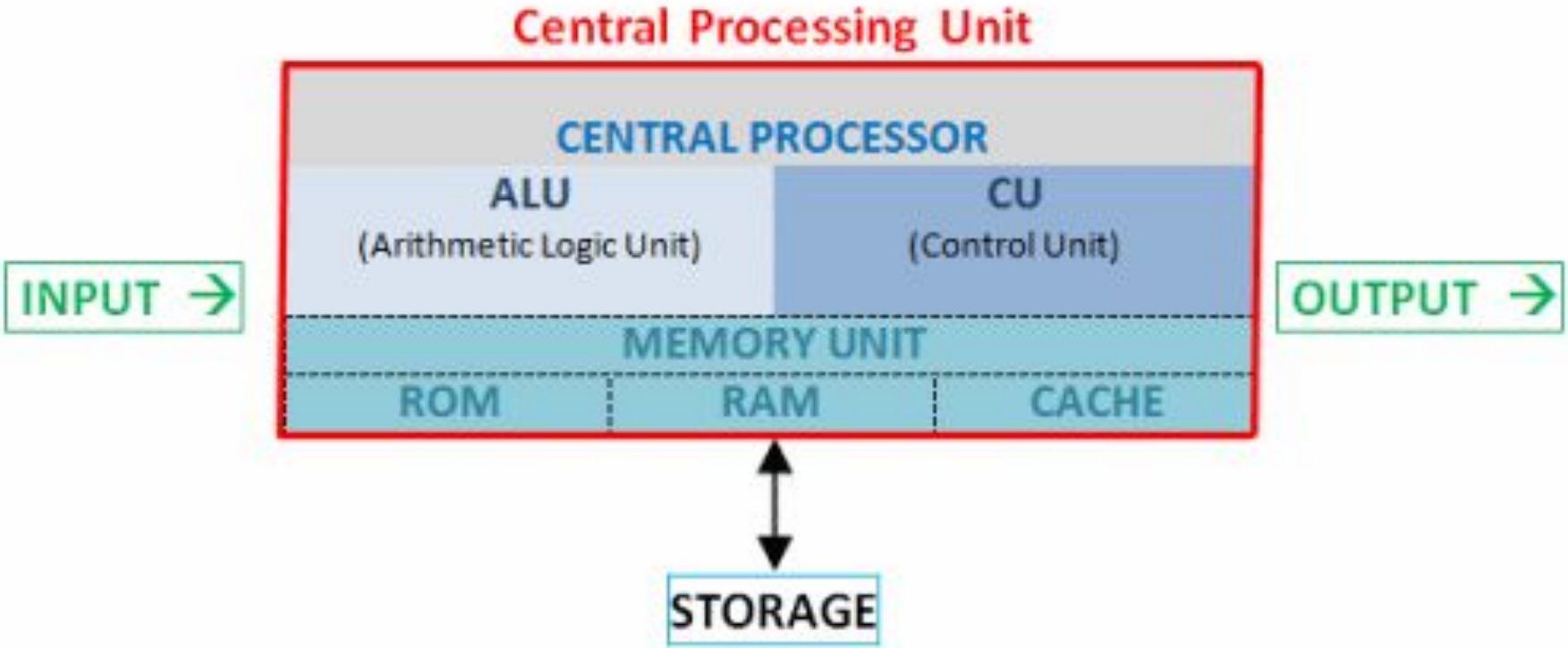
Two main parts

- CPU
- Memory








Central Processing Unit
CENTRAL PROCESSOR
ALU
(Arithmetic Logic Unit)
CU
(Control Unit)
INPUT
OUTPUT
MEMORY UNIT
ROM
RAM
CACHE
STORAGE

- Instruction Set Architecture.
- An agreement between hardware and human for making interaction.
- Example : ADD R1, R2, R3
  - Can be represented as :

00101111100001111001010101010101

10111010100011110101001011011010

ISA Classification

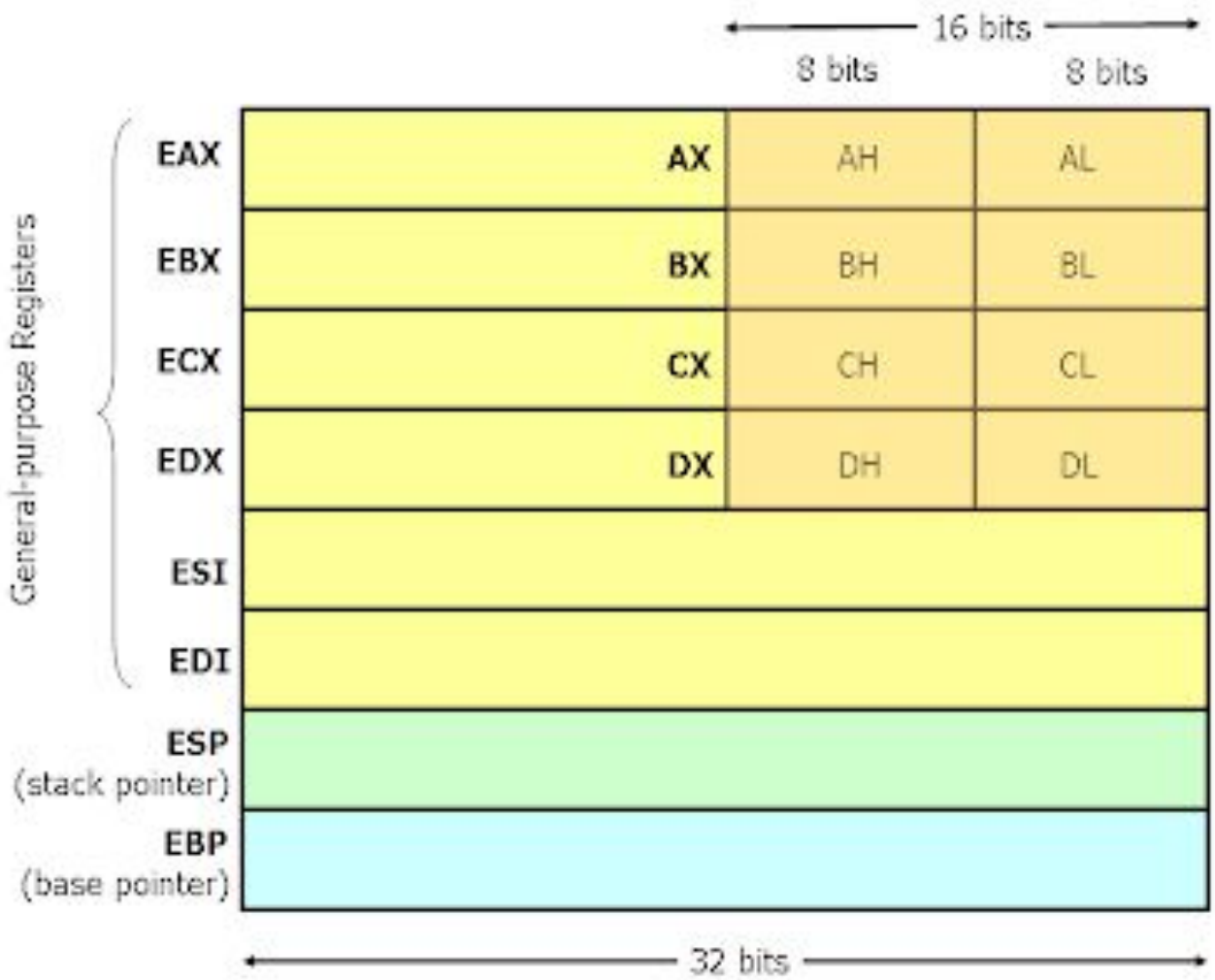
- Two major schools of ISA
- CISC
  - Complex Instruction Set Computer
- RISC
  - Reduced Instruction Set Computer

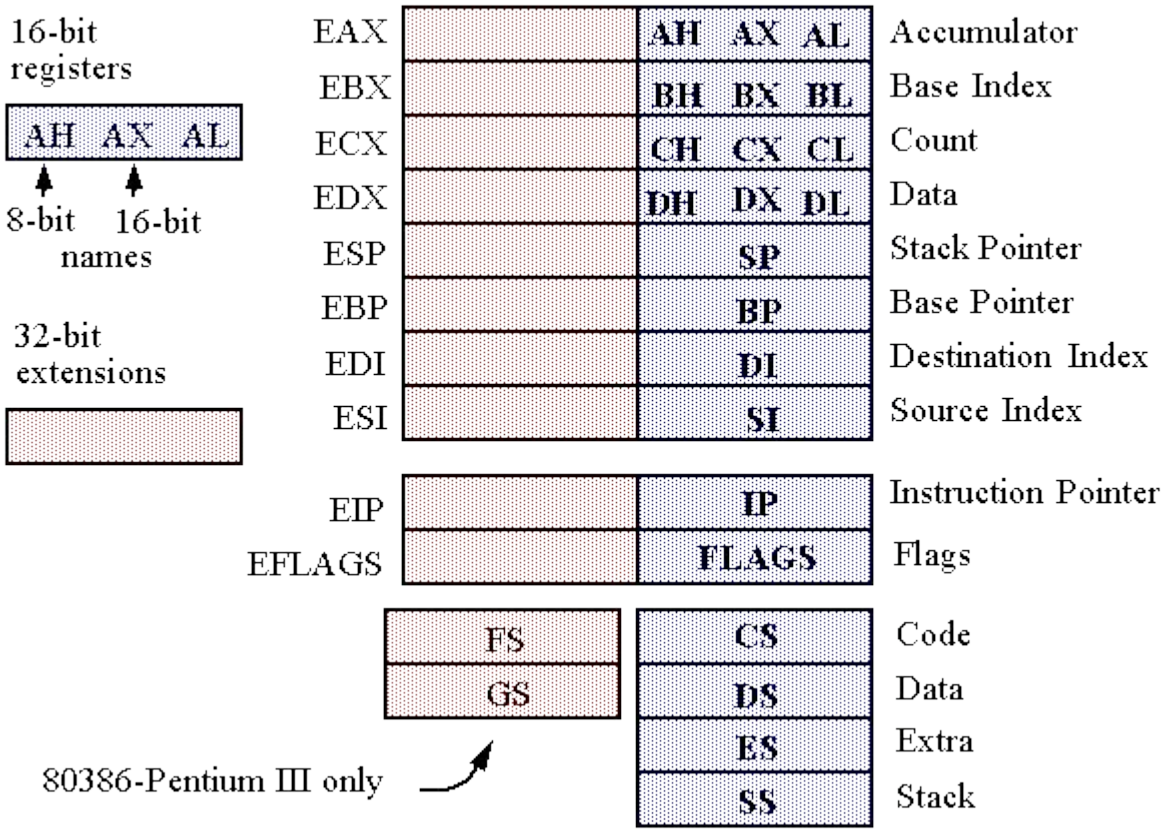
CISC Philosophy

- Reduce amount of storage used and accessed - reduce load/store.
- Give support for compatibility.
- Make compiler’s job easier.
- Support complex assembly level programming.

RISC Philosophy

- Execute one instruction per clock.
- Keep all instructions of same size.
- Allow only load / store instruction to access the memory.
- Give support for high level languages (like C, C++, Java).


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits


16-bit registers
AH AX AL
8-bit
16-bit names
32-bit extensions
EAX
EBX
ECX
EDX
ESP
EBP
EDI
ESI
AH AX AL
BH BX BL
CH CX CL
DH DX DL
SP
BP
DI
SI
Accumulator
Base Index
Count
Data
Stack Pointer
Base Pointer
Destination Index
Source Index
EIP
EFLAGS
IP
FLAGS
Instruction Pointer
Flags
FS
GS
CS
DS
ES
SS
Code
Data
Extra
Stack
80386-Pentium III only


MC68000FN12
4C91E
QQVZ9431

CPU

- Program counter

CPU

- Program counter
- Instruction decoder

CPU

- Program counter
- Instruction decoder
- Data bus

CPU

- Program counter
- Instruction decoder
- Data bus
- General purpose registers

CPU

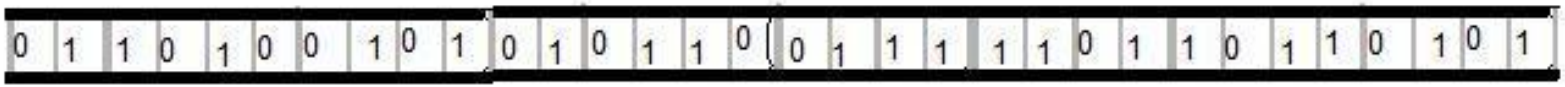
- Program counter
- Instruction decoder
- Data bus
- General purpose registers
- Arithmetic and logic unit

A pictorial representation of some of the registers in a 32-bit processor. Note that each register is composed of 32 chambers containing either zero or one. Since there are only two possible values a chamber can have and since there are 32 chambers in a register, a single register can have (2 raised to the power 32) different combinations of 0s and 1s (i.e. =2^32 different values) .

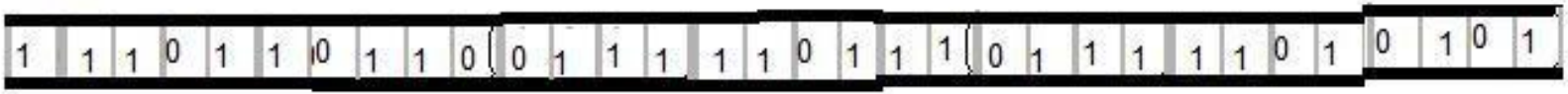
EAX

EBX

ECX

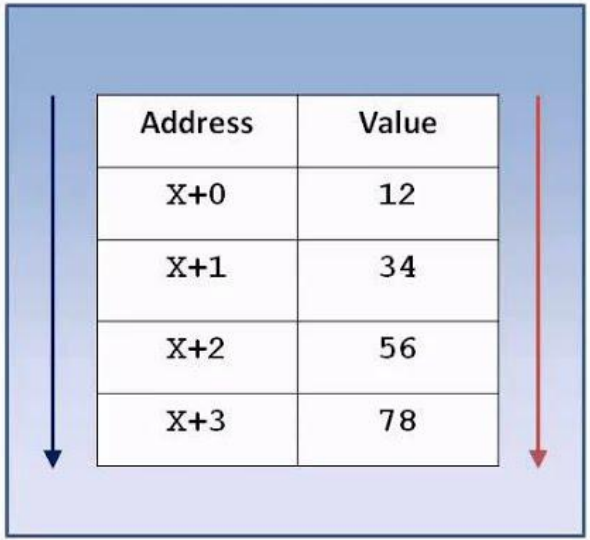


EDX



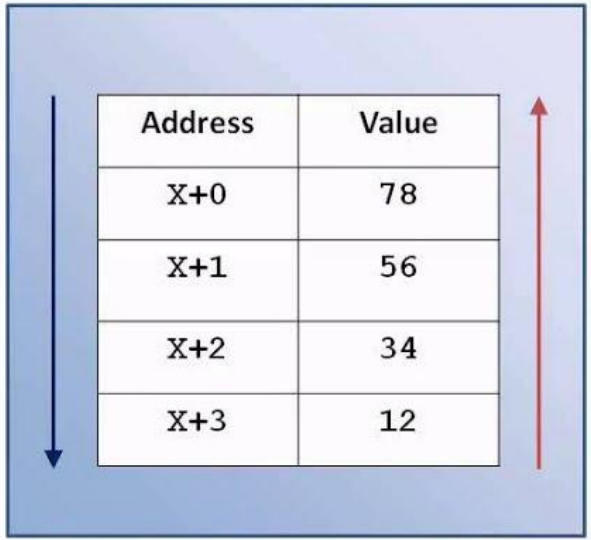
Endianness
0x12345678

Big Endian

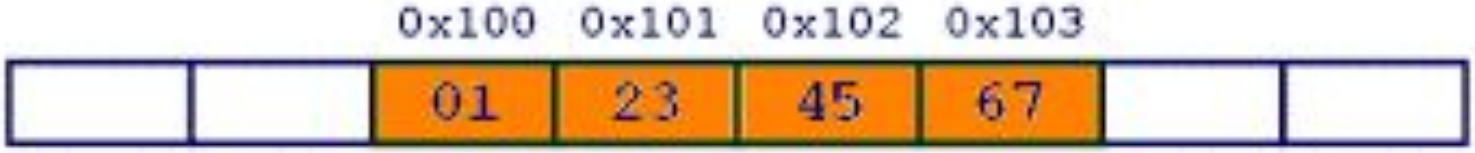


Address	Value
X+0	12
X+1	34
X+2	56
X+3	78

Little Endian



Address	Value
X+0	78
X+1	56
X+2	34
X+3	12



Big Endian

Little Endian

MIPS vs. Intel

```
bitcount:
        ADD     r2, r0, r0
        ADD     r3, r0, r0
loop:
        ADDI    r4, r0, #1
        SHLL    r4, r4, r2
        AND     r4, r4, r1
        BEQZ    noadd
        ADDI    r2, r0, #1
noadd:
        ADDI    r2, r0, #1
        SUBI    r0, r2, #32
        BNEZ    loop
        RET
```

```
bitcount: # EAX->EBX
        XOR     EBX, EBX
        XOR     CL, CL
loop:
        MOV     EDX, #1
        SHL     EDX, CL
        AND     EDX, EAX
        BEQ     noadd
        INC     EBX
noadd:
        INC     CL
        CMP     CL, #32
        BNE     loop
        RET
```

CISC (it is a retronym)

CISC

- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.

CISC

- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.
- Small number of general purpose registers. Instructions which operate directly on memory, and only the limited amount of chip space is dedicated for general purpose registers.

CISC

- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.
- Small number of general purpose registers. Instructions which operate directly on memory, and only the limited amount of chip space is dedicated for general purpose registers.
- Several special purpose registers. Many CISC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design somewhat, at the expense of making the instruction set more complex.

CISC

- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.
- Small number of general purpose registers. Instructions which operate directly on memory, and only the limited amount of chip space is dedicated for general purpose registers.
- Several special purpose registers. Many CISC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design somewhat, at the expense of making the instruction set more complex.
- 'Condition code" register. This register reflects whether the result of the last operation is less than, equal to, or greater than zero and records if certain error conditions occur.

RISC

- One Cycle Execution Time: RISC processors have a CPI (clock per instruction) of one cycle.

RISC

- One Cycle Execution Time: RISC processors have a CPI (clock per instruction) of one cycle.
- Pipelining: A technique that allows simultaneous execution of parts, or stages, of instructions to more efficiently process instructions.

RISC

- One Cycle Execution Time: RISC processors have a CPI (clock per instruction) of one cycle.
- Pipelining: A technique that allows simultaneous execution of parts, or stages, of instructions to more efficiently process instructions.
- Large Number of Registers. The RISC design philosophy generally incorporates a larger number of registers to prevent large amounts of interactions with memory.

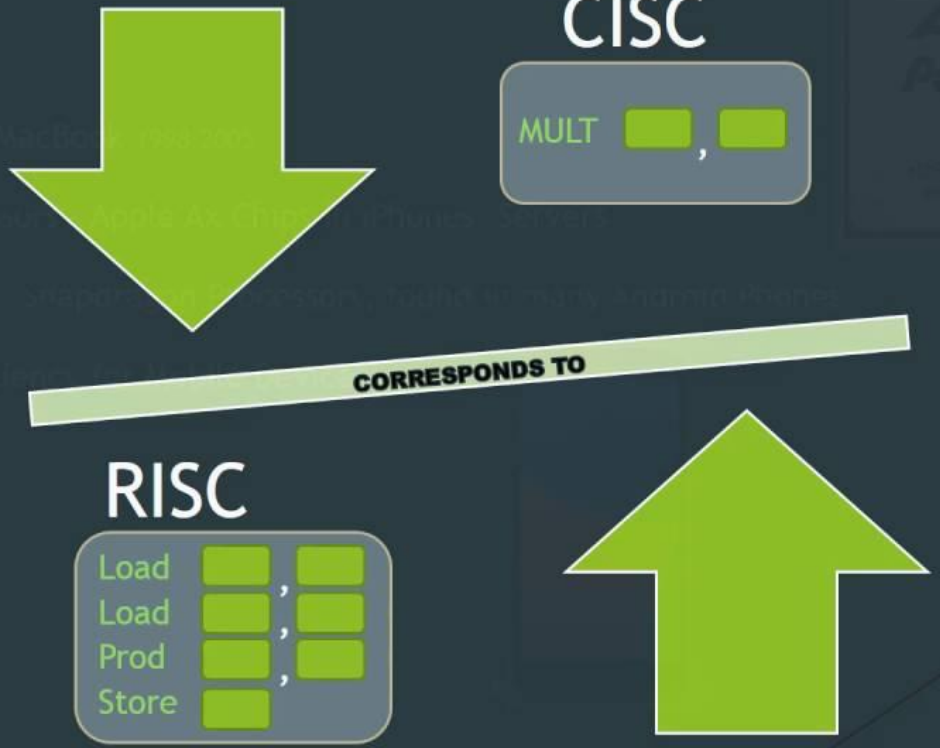
Architectural Characterstics	Complex Instruction Set Computer(CISC)	Reduced Instruction Set Computer(RISC)
Instruction size and format	Large set of instructions with variable formats (16-64 bits per instruction).	Small set of instructions with fixed format (32 bit).
Data transfer	Memory to memory.	Register to register.
CPU control	Most micro coded using control memory (ROM) but modern CISC use hardwired control.	Mostly hardwired without control memory.
Instruction type	Not register based instructions.	Register based instructions.
Memory access	More memory access.	Less memory access.
Clocks	Includes multi-clocks.	Includes single clock.
Instruction nature	Instructions are complex.	Instructions are reduced and simple.

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

RISC vs. CISC

Parameter	RISC	CISC
Instruction types	Simple	Complex
Number of instructions	Reduced (30-40)	Extended (100-200)
Duration of an instruction	One cycle	More cycles (4-120)
Instruction format	Fixed	Variable
Instruction execution	In parallel (pipeline)	Sequential
Addressing modes	Simple	Complex
Instructions accessing the memory	Two: Load and Store	Almost all from the set
Register set	multiple	unique
Complexity	In compiler	In CPU (micro-program)

Comparison of Instruction Sets



Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Direct addressing mode:

```
movl ADDRESS, %eax
```

This loads %eax with the value at memory address ADDRESS .

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Direct addressing mode:

```
movl ADDRESS, %eax
```

This loads %eax with the value at memory address ADDRESS.

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Direct addressing mode:

```
movl ADDRESS, %eax
```

This loads %eax with the value at memory address ADDRESS.

Indirect addressing mode:

```
movl (%eax), %ebx
```

Eax helds an address, and we move the value at that address to ebx.

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Direct addressing mode:

```
movl ADDRESS, %eax
```

This loads %eax with the value at memory address ADDRESS.

Indirect addressing mode:

```
movl (%eax), %ebx
```

%eax helds an address, and we move the value at that address to %ebx.

Base pointer addressing mode:

```
movl  4(%eax), %ebx
```

As indirect, but adds a constant value to the address in the register. Useful for record fields.

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Immediate addressing mode:

```
movl $12, %eax
```

If we did not use $ sign, then the value located at memory location 12 would be used.

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Immediate addressing mode:

```
movl $12, %eax
```

If we did not use $ sign, then the value located at memory location 12 would be used.

Register addressing mode:

simply moves data in or out of a register. In all of our examples, register addressing mode was used for the other operand.

Addresing modes

https://en.wikipedia.org/wiki/Addressing_mode

Immediate addressing mode:

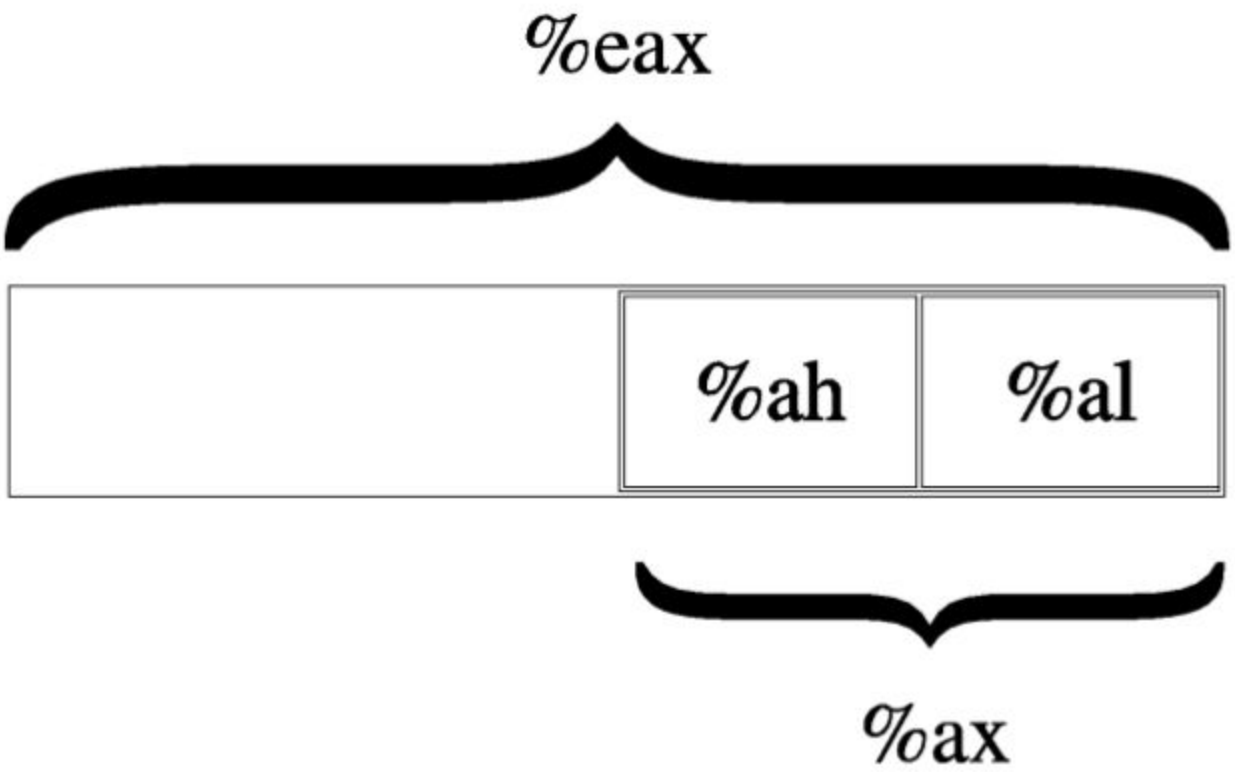
```
movl $12, %eax
```

If we did not use $ sign, then the value located at memory location 12 would be used.

Register addressing mode:

simply moves data in or out of a register. In all of our examples, register addressing mode was used for the other operand.

Every mode except immediate mode can be used as either the source or destination operand.
Immediate mode can only be a source operand.



Layout of the `%eax` register

How functions work.

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

local variables

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

local variables

static variables

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

local variables

static variables

global variables

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

local variables

static variables

global variables

return address

How functions work.

function name - represents the address where the function’s code starts. (in asm - label)

function parameters.

local variables

static variables

global variables

return address

return value

Calling conventions.

overflows

```
byte byte1 = 150; // 10010110
byte byte2 = 199; // 11000111
```

overflows

```
byte byte1 = 150; // 10010110
byte byte2 = 199; // 11000111

        byte byte3 = byte1 + byte2;
```

overflows

```
byte byte1 = 150; // 10010110
byte byte2 = 199; // 11000111

        byte byte3 = byte1 + byte2;
                                        byte3 = 94
```

overflows

```
byte byte1 = 150; // 10010110
byte byte2 = 199; // 11000111

        byte byte3 = byte1 + byte2;
                                                byte3 = 94
```

150+199=349, binary 1 0101 1101,
the upper 1 bit is dropped and the byte becomes
0101 1101;

Why computers use ‘two’s complement’ to represent signed numbers?

Why computers use ‘two’s complement’ to represent signed numbers?

Clock arithmetic

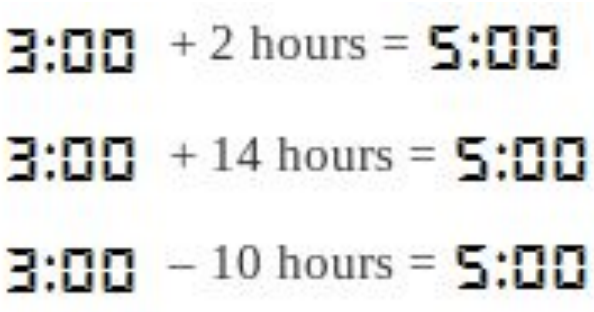
3:00 + 2 hours = 5:00

3:00 + 14 hours = 5:00

3:00 – 10 hours = 5:00

Why computers use ‘two’s complement’ to represent signed numbers?

Clock arithmetic



In the “clock arithmetic”, 2, 14 and –10 are just three different ways to write down the same number.

Why computers use ‘two’s complement’ to represent signed numbers?

Clock arithmetic

3:00 + 2 hours = 5:00

3:00 + 14 hours = 5:00

3:00 – 10 hours = 5:00

In the “clock arithmetic”, 2, 14 and –10 are just three different ways to write down the same number.

They are interchangeable in multiplications too:

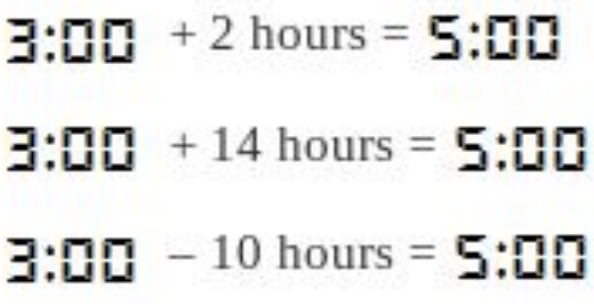
12:00 + (3 * 2) hours = 6:00

12:00 + (3 * 14) hours = 6:00

12:00 + (3 * -10) hours = 6:00

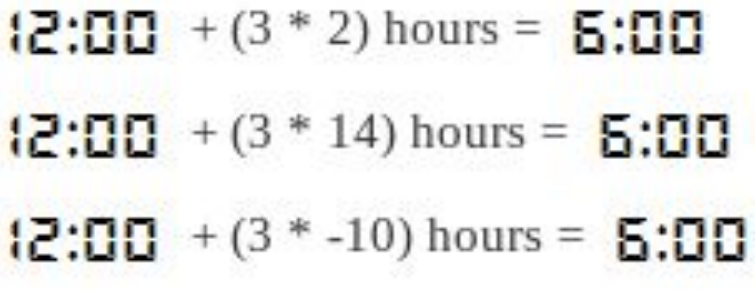
Why computers use ‘two’s complement’ to represent signed numbers?

Clock arithmetic



In the “clock arithmetic”, 2, 14 and –10 are just three different ways to write down the same number.

They are interchangeable in multiplications too:



formal term for “clock arithmetic” is “modular arithmetic”. In modular arithmetic, two numbers are equivalent if they leave the same non-negative remainder when divided by a particular number.

Why computers use ‘two’s complement’ to represent signed numbers?

consider 3-bit integers, which can represent integers from 0 to 7.
If you add or multiply two of these 3-bit numbers in fixed-width binary arithmetic, you’ll get the “modular arithmetic” answer:

1 + 2 –> 3

4 + 5 -> 1

Why computers use ‘two’s complement’ to represent signed numbers?

consider 3-bit integers, which can represent integers from 0 to 7.
If you add or multiply two of these 3-bit numbers in fixed-width binary arithmetic, you’ll get the “modular arithmetic” answer:

1 + 2 –> 3
4 + 5 -> 1

The calculations wrap around, because any answer larger than 7 cannot be represented with 3 bits. The wrapped-around answer is still meaningful:

Why computers use ‘two’s complement’ to represent signed numbers?

consider 3-bit integers, which can represent integers from 0 to 7.
If you add or multiply two of these 3-bit numbers in fixed-width binary arithmetic, you’ll get the “modular arithmetic” answer:

1 + 2 –> 3
4 + 5 -> 1

The calculations wrap around, because any answer larger than 7 cannot be represented with 3 bits. The wrapped-around answer is still meaningful:

- **The answer we got is congruent (i.e., equivalent) to the real answer, modulo 8.**
  This is the modular arithmetic! The real answer was 9, but we got 1.
  And, both 9 and 1 leave remainder 1 when divided by 8.

Why computers use ‘two’s complement’ to represent signed numbers?

consider 3-bit integers, which can represent integers from 0 to 7.
If you add or multiply two of these 3-bit numbers in fixed-width binary arithmetic, you’ll get the “modular arithmetic” answer:

1 + 2 –> 3
4 + 5 -> 1

The calculations wrap around, because any answer larger than 7 cannot be represented with 3 bits. The wrapped-around answer is still meaningful:

- **The answer we got is congruent (i.e., equivalent) to the real answer, modulo 8.**
  This is the modular arithmetic! The real answer was 9, but we got 1.
  And, both 9 and 1 leave remainder 1 when divided by 8.

- **The answer we got represents the lowest 3 bits of the correct answer**
  For 4+5, we got 001, while the correct answer is 1001.

Why computers use ‘two’s complement’ to represent signed numbers?

consider 3-bit integers, which can represent integers from 0 to 7.
If you add or multiply two of these 3-bit numbers in fixed-width binary arithmetic, you’ll get the “modular arithmetic” answer:

1 + 2 –> 3
4 + 5 -> 1

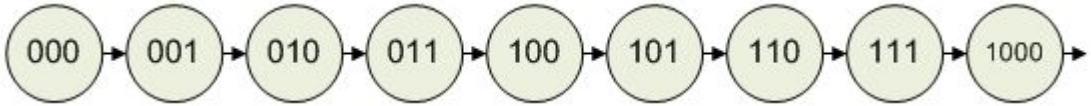
The calculations wrap around, because any answer larger than 7 cannot be represented with 3 bits. The wrapped-around answer is still meaningful:

- **The answer we got is congruent (i.e., equivalent) to the real answer, modulo 8.**
  This is the modular arithmetic! The real answer was 9, but we got 1.
  And, both 9 and 1 leave remainder 1 when divided by 8.

- **The answer we got represents the lowest 3 bits of the correct answer**
  For 4+5, we got 001, while the correct answer is 1001.

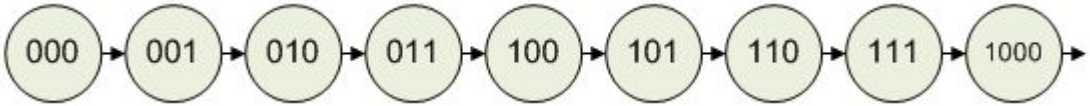
Why computers use ‘two’s complement’ to represent signed numbers?

Imagine infinite number line:

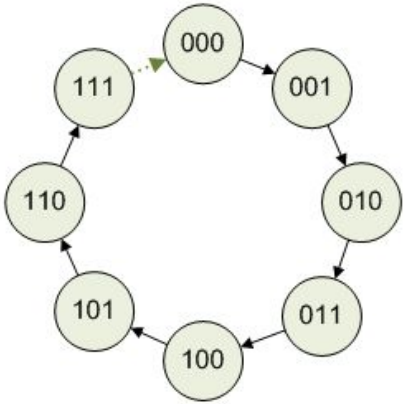


Why computers use ‘two’s complement’ to represent signed numbers?

Imagine infinite number line:



Then curl the number line into a circle so that 1000 overlaps the 000:



Why computers use ‘two’s complement’ to represent signed numbers?

the adder for unsigned integers can be used for signed integers too, exactly as it is!

Binary	Unsigned value	Signed value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

You can interpret those eight values as signed or unsigned.

Why computers use ‘two’s complement’ to represent signed numbers?

It is called “two’s complement” because to negate an integer, you subtract it from 2^N.

For example, to get the representation of –2 in 3-bit arithmetic, you can compute 8 – 2 = 6, and so –2 is represented in two’s complement as 6 in binary: 110.

Why computers use ‘two’s complement’ to represent signed numbers?

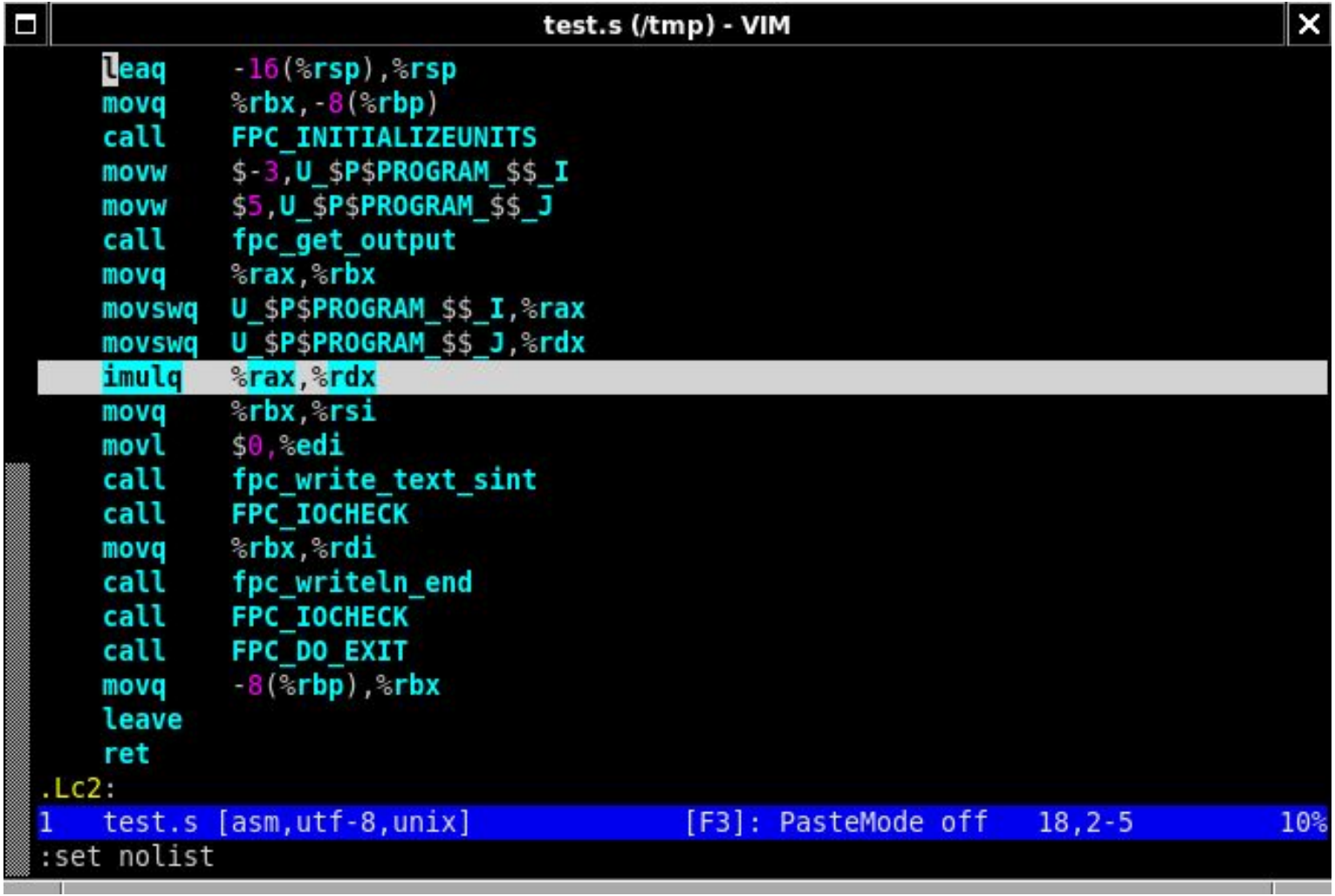
It is called “two’s complement” because to negate an integer, you subtract it from 2^N.

For example, to get the representation of –2 in 3-bit arithmetic, you can compute 8 – 2 = 6, and so –2 is represented in two’s complement as 6 in binary: 110.

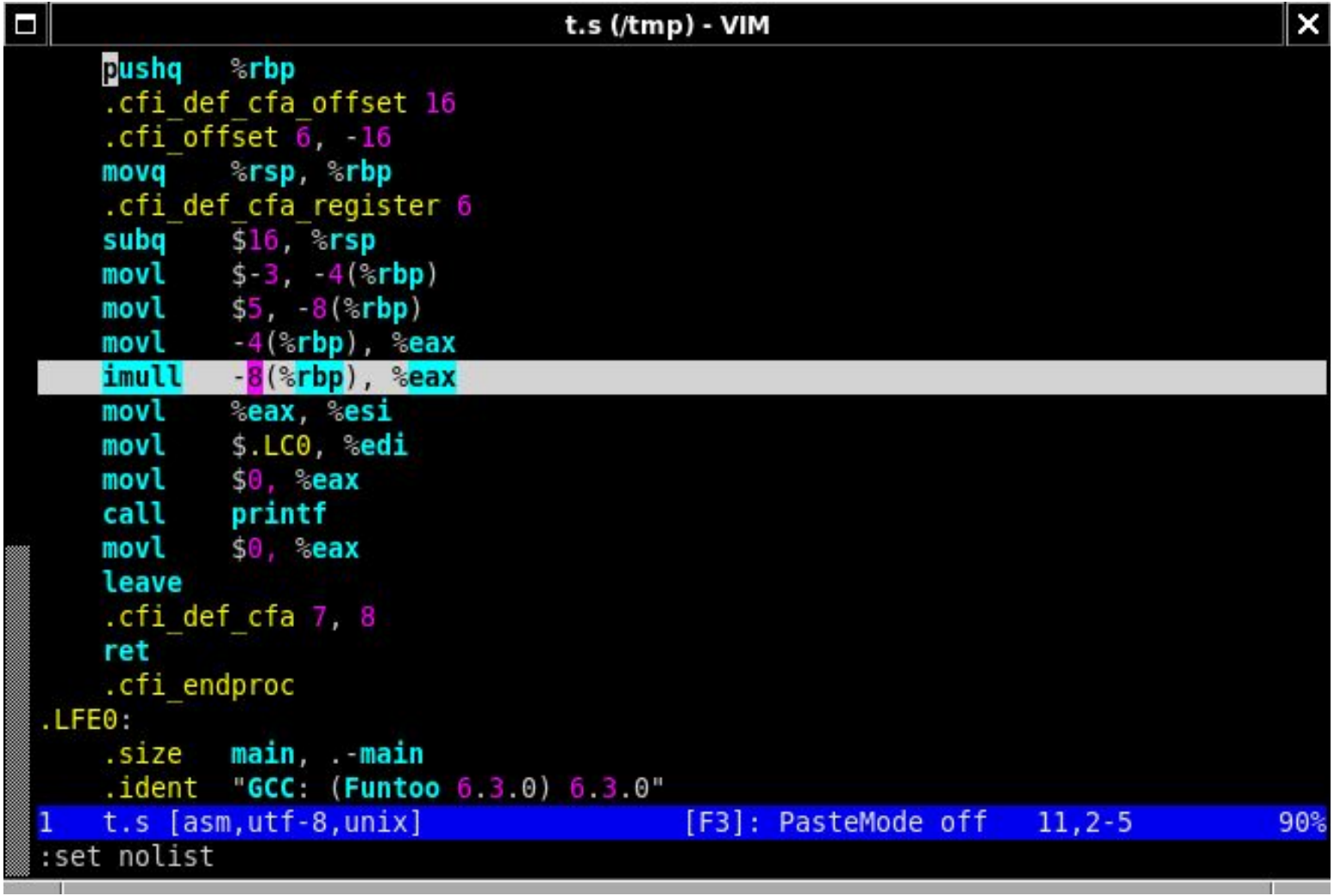
another way to compute two’s complement, which is easier to imagine implemented in hardware:

1. Start with a binary representation of the number you need to negate
2. Flip all bits
3. Add one


test.pas (/tmp) - VIM
var i, j: integer;
begin
i := -3;
j := 5;
writeln (i * j)
end.
1 test.pas [pascal,utf-8,unix] [F3]: PasteMode off 8,3 All
"test.pas" 10L, 73C


test.s (/tmp) - VIM
leaq -16(%rsp),%rsp
movq %rbx,-8(%rbp)
call FPC_INITIALIZEUNITS
movw $-3,U_$P$PROGRAM_$$_I
movw $5,U_$P$PROGRAM_$$_J
call fpc_get_output
movq %rax,%rbx
movswq U_PPROGRAM_$$_I,%rax
movswq U_PPROGRAM_$$_J,%rdx
imulq %rax,%rdx
movq %rbx,%rsi
movl $0,%edi
call fpc_write_text_sint
call FPC_IOCHECK
movq %rbx,%rdi
call fpc_writeln_end
call FPC_IOCHECK
call FPC_DO_EXIT
movq -8(%rbp),%rbx
leave
ret
.Lc2:
1 test.s [asm,utf-8,unix] [F3]: PasteMode off 18,2-5 10%
:set nolist


t.c (/tmp) - VIM
#include <stdio.h>
int main()
{
Unsigned int a, b;
a = -3;
b = 5;
printf ("%d\n", a * b);
}
1 t.c [c,utf-8,unix] [F3]: PasteMode off 6,3 All
"t.c" 13L, 106C


t.s (/tmp) - VIM
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $-3, -4(%rbp)
movl $5, -8(%rbp)
movl -4(%rbp), %eax
imull -8(%rbp), %eax
movl %eax, %esi
movl $.LC0, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Funtoo 6.3.0) 6.3.0"
1 t.s [asm,utf-8,unix] [F3]: PasteMode off 11,2-5 90%
:set nolist

Why computers use ‘two’s complement’ to represent signed numbers?

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
Console.WriteLine(a * b);
```

```
0000004f  call    79084EA0
00000054  mov         ecx,eax
00000056  imul        esi,edi
00000059  mov         edx,esi
0000005b  mov         eax,dword ptr [ecx]
```

```
uint a = uint.Parse(Console.ReadLine());
uint b = uint.Parse(Console.ReadLine());
Console.WriteLine(a * b);
```

```
0000004f  call    79084EA0
00000054  mov         ecx,eax
00000056  imul        esi,edi
00000059  mov         edx,esi
0000005b  mov         eax,dword ptr [ecx]
```

The IMUL instruction does not know whether its arguments are signed or unsigned, and it can still multiply them correctly!

Why computers use ‘two’s complement’ to represent signed numbers?

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
Console.WriteLine(a * b);
```

```
0000004f  call    79084EA0
00000054  mov         ecx,eax
00000056  imul        esi,edi
00000059  mov         edx,esi
0000005b  mov         eax,dword ptr [ecx]
```

```
uint a = uint.Parse(Console.ReadLine());
uint b = uint.Parse(Console.ReadLine());
Console.WriteLine(a * b);
```

```
0000004f  call    79084EA0
00000054  mov         ecx,eax
00000056  imul        esi,edi
00000059  mov         edx,esi
0000005b  mov         eax,dword ptr [ecx]
```

IMUL is for signed multiplications. MUL is for unsigned. IMUL checks for overflows and sets the overflow processor flag.

Why computers use ‘two’s complement’ to represent signed numbers?

127	0111 1111	0111 1111	0111 1111
1	0000 0001	0000 0001	0000 0001
0	0000 0000	0000 0000	0000 0000
-0	1000 0000	1111 1111	---
-1	1000 0001	1111 1110	1111 1111
-2	1000 0010	1111 1101	1111 1110
-3	1000 0011	1111 1100	1111 1101
-4	1000 0100	1111 1011	1111 1100
-5	1000 0101	1111 1010	1111 1011
-6	1000 0110	1111 1001	1111 1010
-7	1000 0111	1111 1000	1111 1001
-8	1000 1000	1111 0111	1111 1000
-9	1000 1001	1111 0110	1111 0111
-10	1000 1010	1111 0101	1111 0110
-11	1000 1011	1111 0100	1111 0101
-127	1111 1111	1000 0000	1000 0001
-128	---	---	1000 0000

Why computers use ‘two’s complement’ to represent signed numbers?

...	...
13	0013
12	0012
11	0011
10	0010
9	0009
8	0008
...	...
2	0002
1	0001
0	0000
-1	9999
-2	9998
-3	9997
-4	9996
...	...
-9	9991
-10	9990
-11	9989
-12	9988
...	...

Why computers use ‘two’s complement’ to represent signed numbers?

Pascal:

```
if (a < 0) then
    a := ((not a) or 128) + 1;
```

C:

```
int convert(int a) {
 if (a < 0)
   a = ~(-a) + 1;
 return a;
}
```

Why computers use ‘two’s complement’ to represent signed numbers?

Pros:

- Same cpu instructions for addition, substruction, multiplication (check overflow flags)
- No -0

Cons:

- Unusual for humans representation
- if you try to negate the lowest representable value, you get an overflow

Why computers use ‘two’s complement’ to represent signed numbers?

Sign extention

10	0000 1010	0000 0000 0000 1010
-15	1111 0001	1111 1111 1111 0001

Why computers use ‘two’s complement’ to represent signed numbers?