Systemy komputerowe: architektura i programowanie (SYKT/SYKOM)

Wykład: Procesor

Aleksander Pruszkowski

Instytut Telekomunikacji Politechniki Warszawskiej

- Plan wykładu
  - Budowa procesorów i łącznie ich z otoczeniem
  - Procesor RISC-V
  - Procesor x86
  - Inne procesory i mikrokontrolery

Budowa procesorów i łącznie ich z otoczeniem

Budowa procesorów i łącznie ich z otoczeniem

- Co to jest procesor?
  - Generalnie: element przetwarzający informacje
    - scenariusz przetwarzania:
      - kod
    - informacje:
      - dane:
        - stałe w kodzie i w pamięci operacyjnej
        - nośniki danych
        - urządzenia we/wy
  - A rezultat pracy procesora?
    - przekazywane do:
      - dane wpływające na otoczenie (urządzenia we/wy)
      - dane zapamiętywane na nośnikach

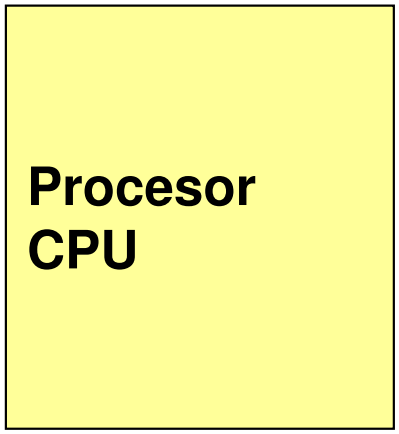
Budowa procesorów i łącznie ich z otoczeniem

- Jak procesor widzi kod i dane
  - Mamy dwie architektury
    - Von Neumana
      - kod i dane - dostępne w ten sam sposób
    - Harwardzka
      - kod i dane - mają osobne przestrzenie

- A jakiej informacje potrafi przetwarzać procesor?
  - „Na raz” - liczba bitów: im więcej tym lepiej?
    - konsekwencje
      - większa liczba tranzystorów
        - wyższy koszt wytworzenia i użytkowania
      - szerokość doprowadzania informacji - tzw. magistral
      - niedopasowanie - zapamiętanie wartości z zakresu 0...255 a zajmuje tyle „krzemu” ile dla zapisania wartości z zakresu 0...4294967295 (!)
      - ...
  - Przyjmuje się że elementarna operacja dodawania realizowana przez procesor wykonuje się na określonej liczbie bitów „na raz” - to potoczna liczba bitów procesora

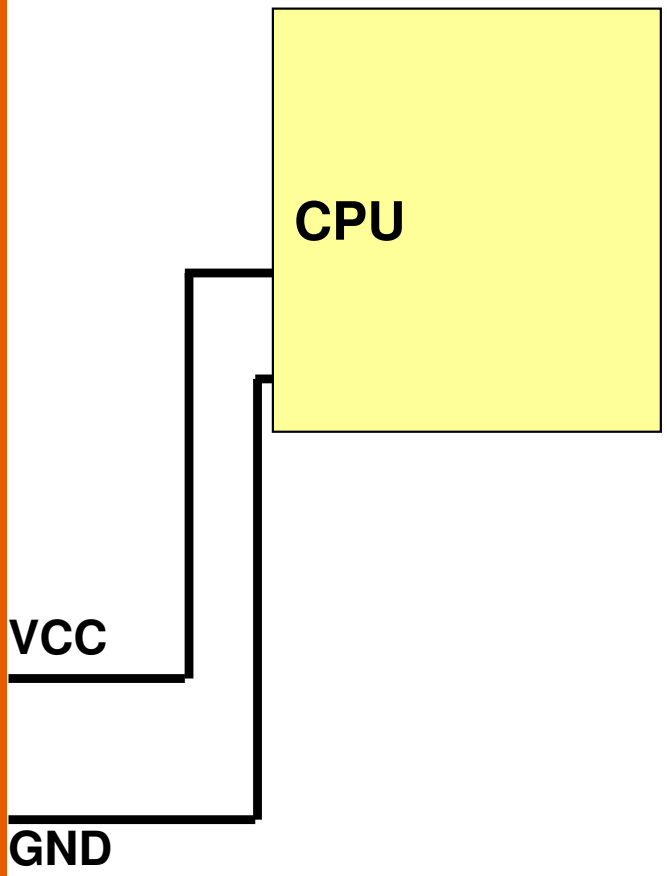
Budowa procesorów i łącznie ich z otoczeniem

- Jak łączyć procesor z resztą elementów komputera?
  - Jaka obudowa - fizyczne parametry
    - Liczba wyprowadzeń, kształt, montaż radiatora, ...
    - DIP/..../BGA - trudność montażu

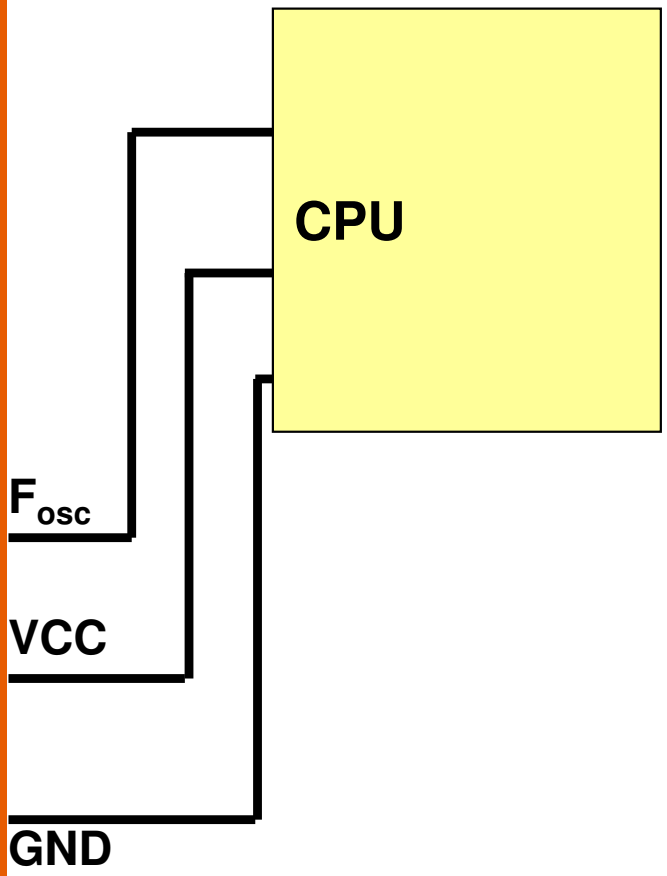


Budowa procesorów i łącznie ich z otoczeniem

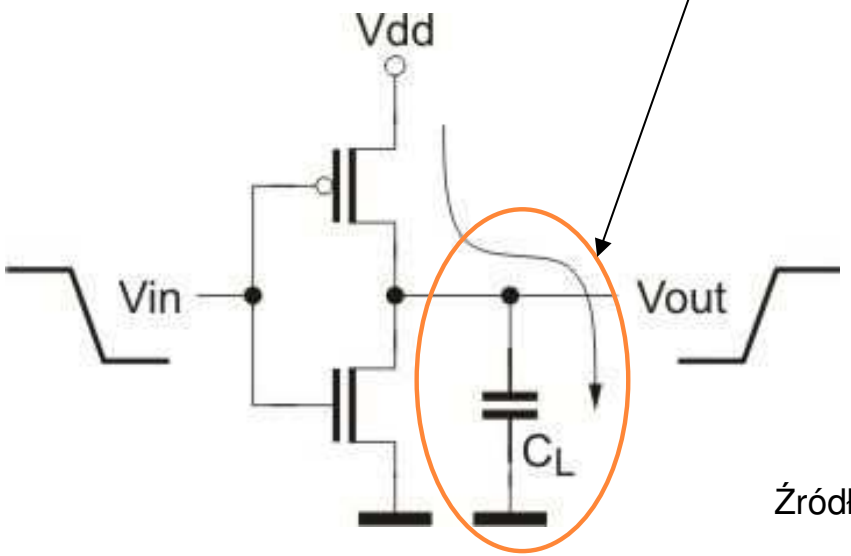
- Jak łączyć procesor z resztą elementów komputera? Cd.
  - Jakie napięcie i jakie natężenie prądu?
    - zjawiska statyczne (np.: upływ prądu przez izolatory, ...)



- Jak łączyć procesor z resztą elementów komputera? Cd.
  - Zegar (Fosc) - czy najszybszy „zegar” to najlepszy wybór?
    - zjawiska dynamiczne
      - problem przełączania bramek CMOS (pojemności, nie symetria wykonania tranzystorów)



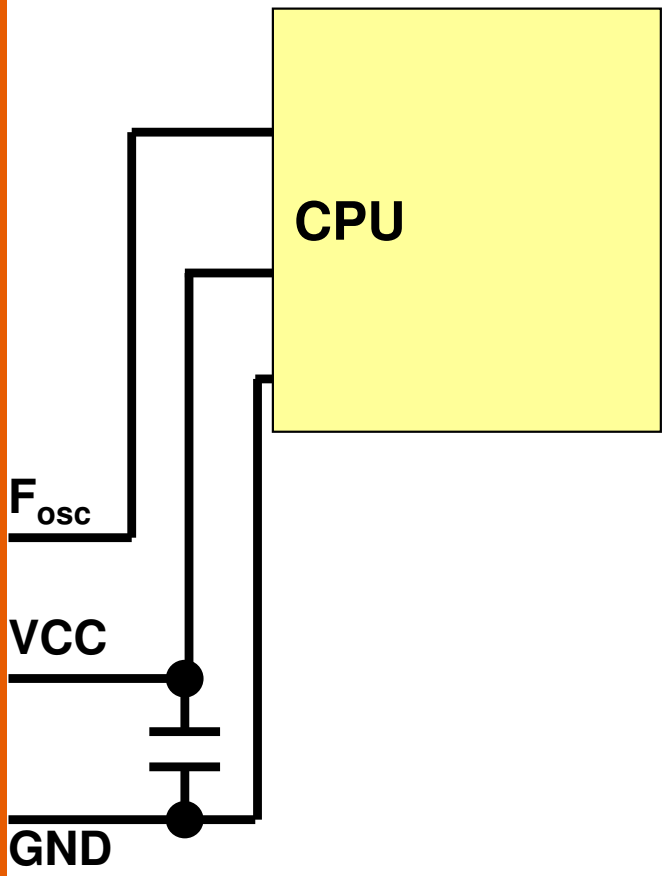
Pojemność obciążenia
(np.: pojemność bramki
tranzystora MOS dołączonej bramki)



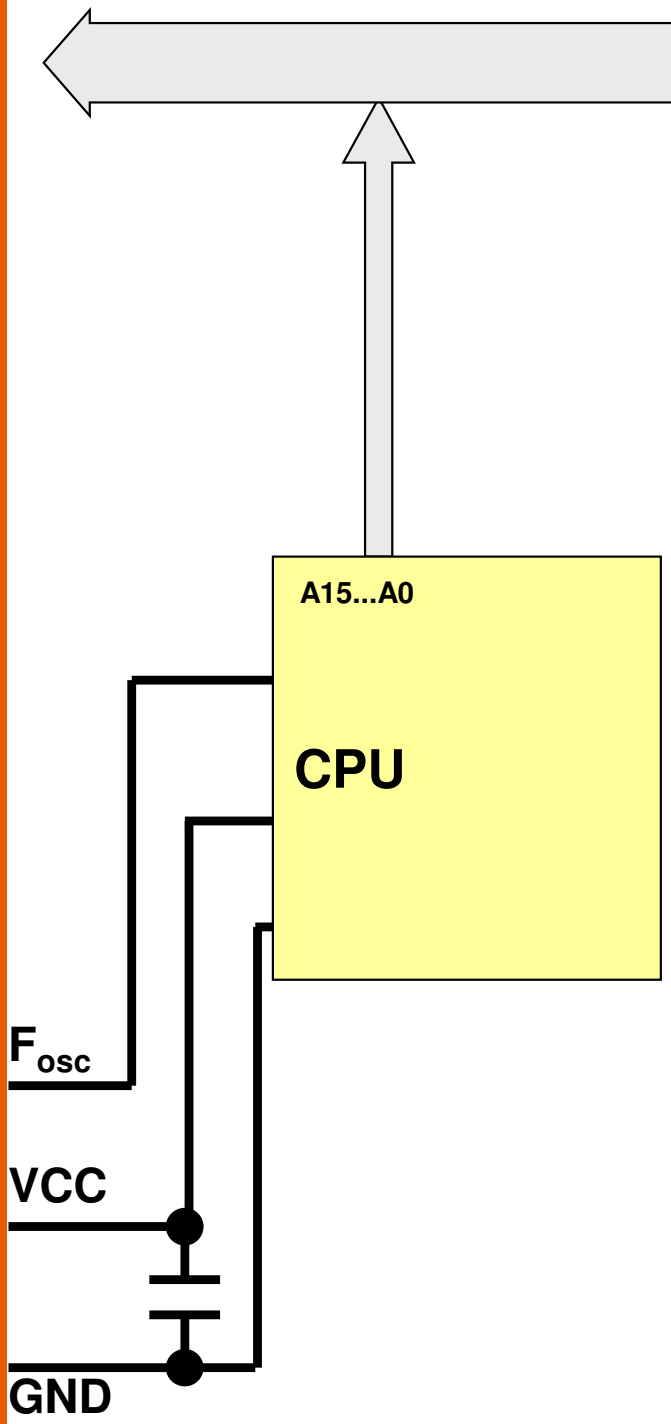
Źródło: elektronikab2b.pl

- Jak łączyć procesor z resztą elementów komputera? Cd.
  - A co z zakłóceniami i dziwnym zachowaniem procesora
    - błędy w kodzie, ...
    - „piki” prądowe - a co to?

- remedium kondensatory filtrujące!
  - pojemność: 100nF, ...
- montaż: jak najbliżej wyprowadzeń



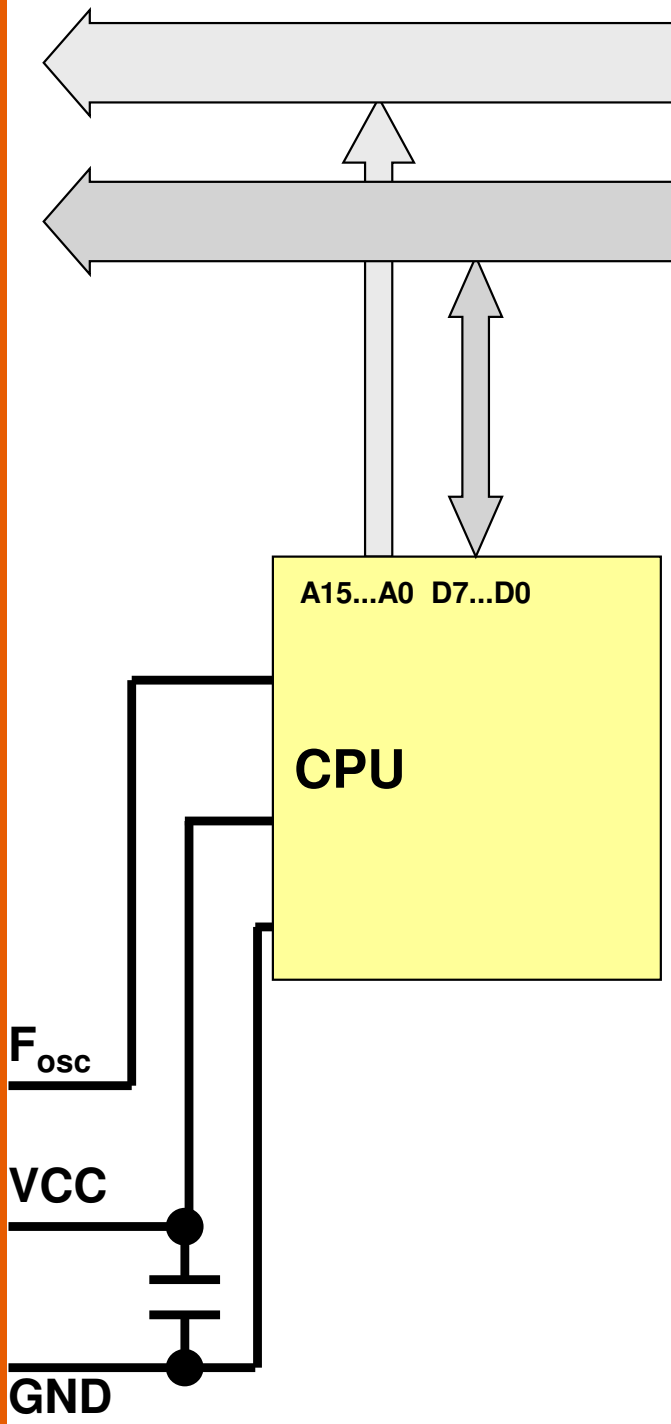
Budowa procesorów i łącznie ich z otoczeniem



- Szerokość magistrali adresowej:

  zdolność do adresowania

Budowa procesorów i łącznie ich z otoczeniem

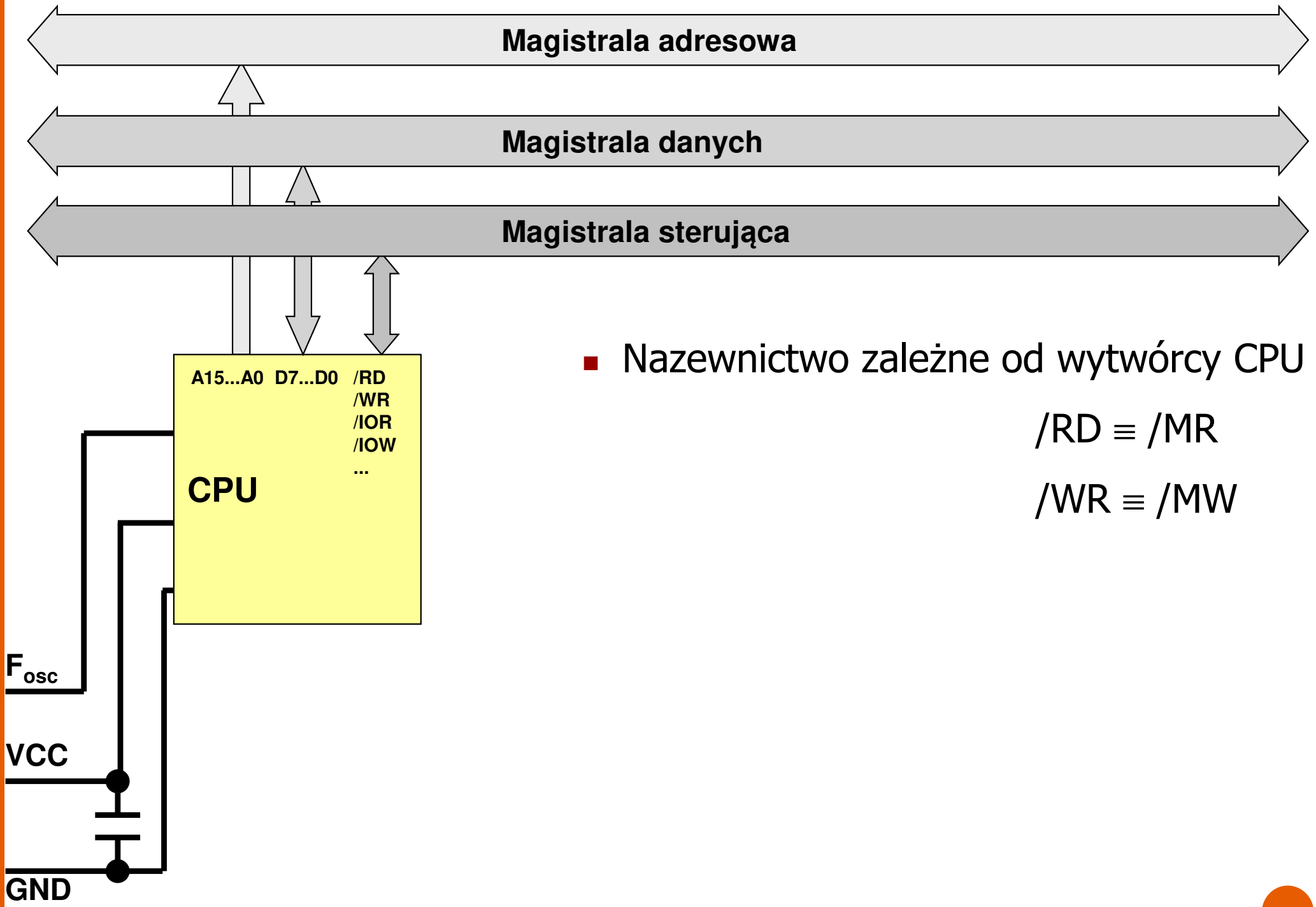


- Szerokość magistrali danych:

  wpływa na szybkość

  - Związek z liczbą bitów CPU

Budowa procesorów i łącznie ich z otoczeniem

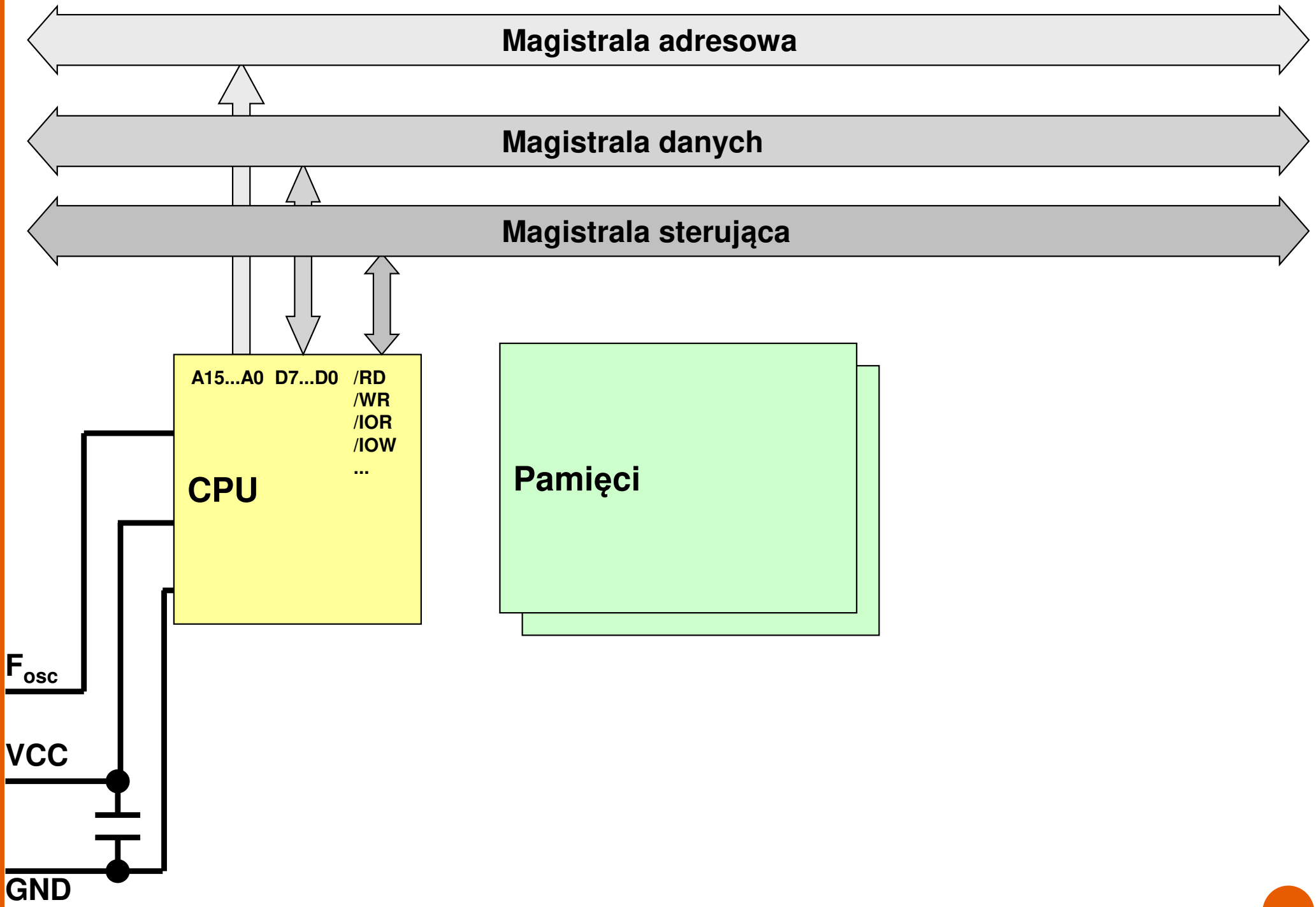


- Nazewnictwo zależne od wytwórcy CPU

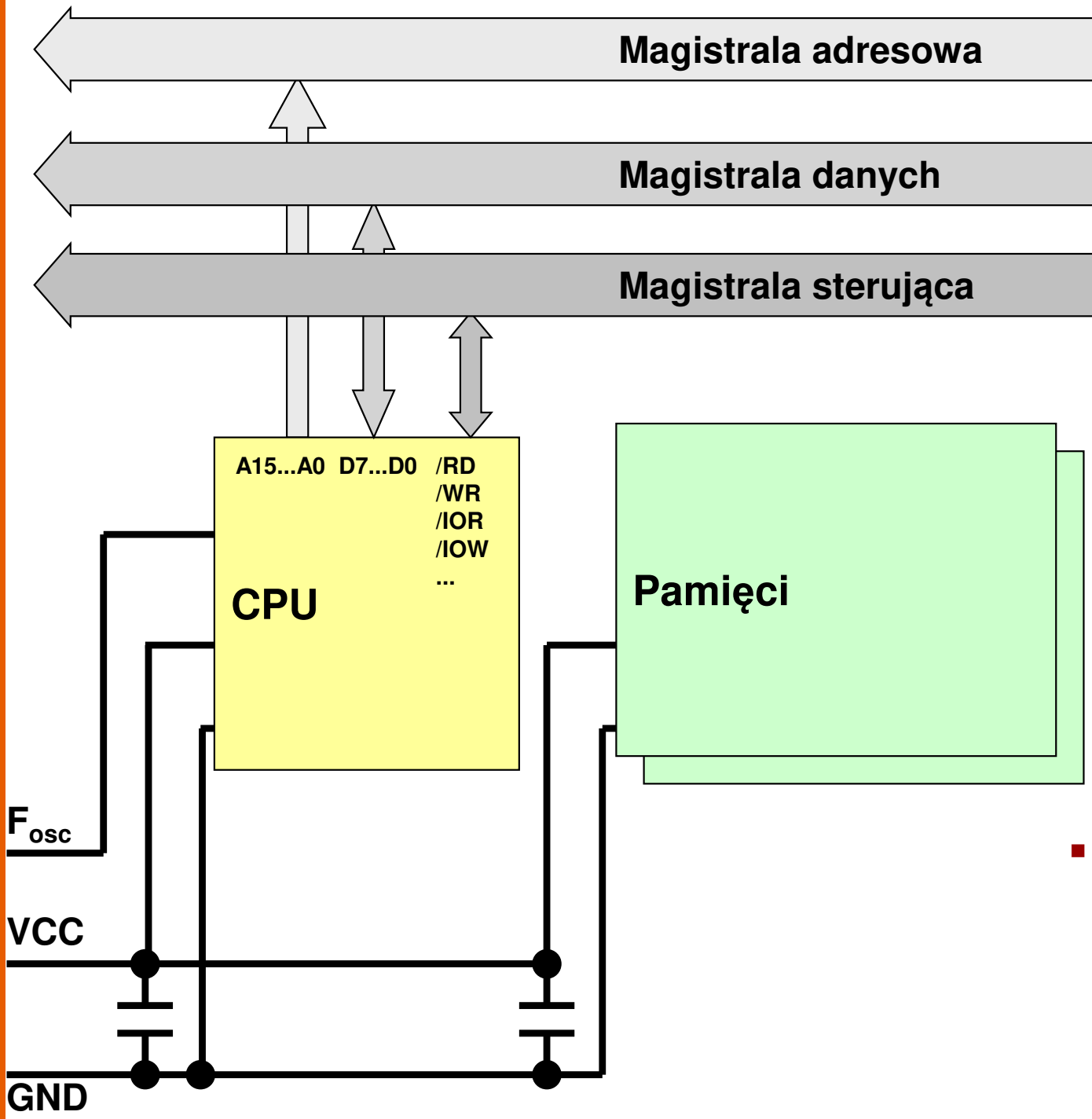
  /RD ≡ /MR

  /WR ≡ /MW

Budowa procesorów i łącznie ich z otoczeniem



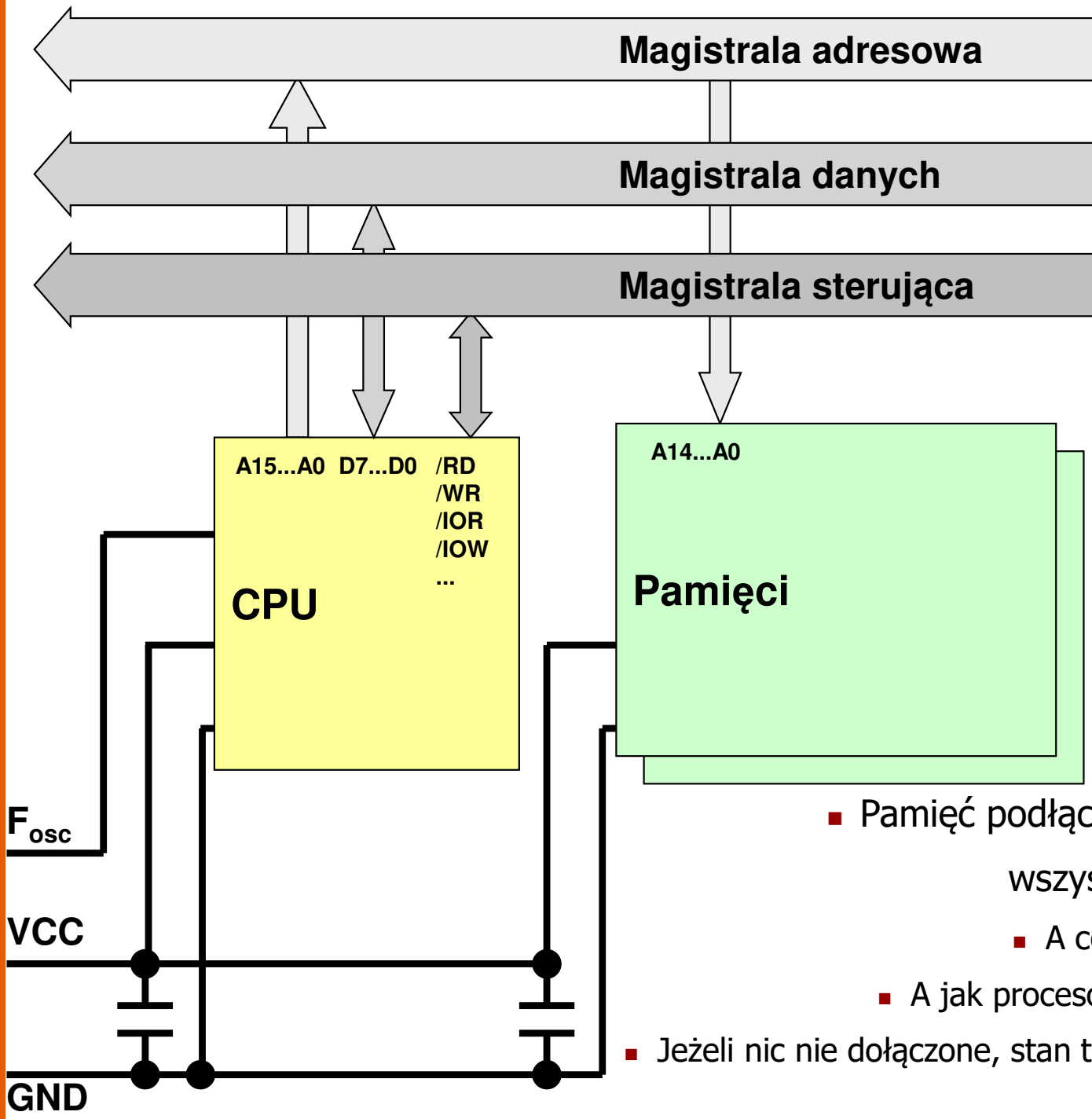
Budowa procesorów i łącznie ich z otoczeniem



GENERALNE ZASADY:

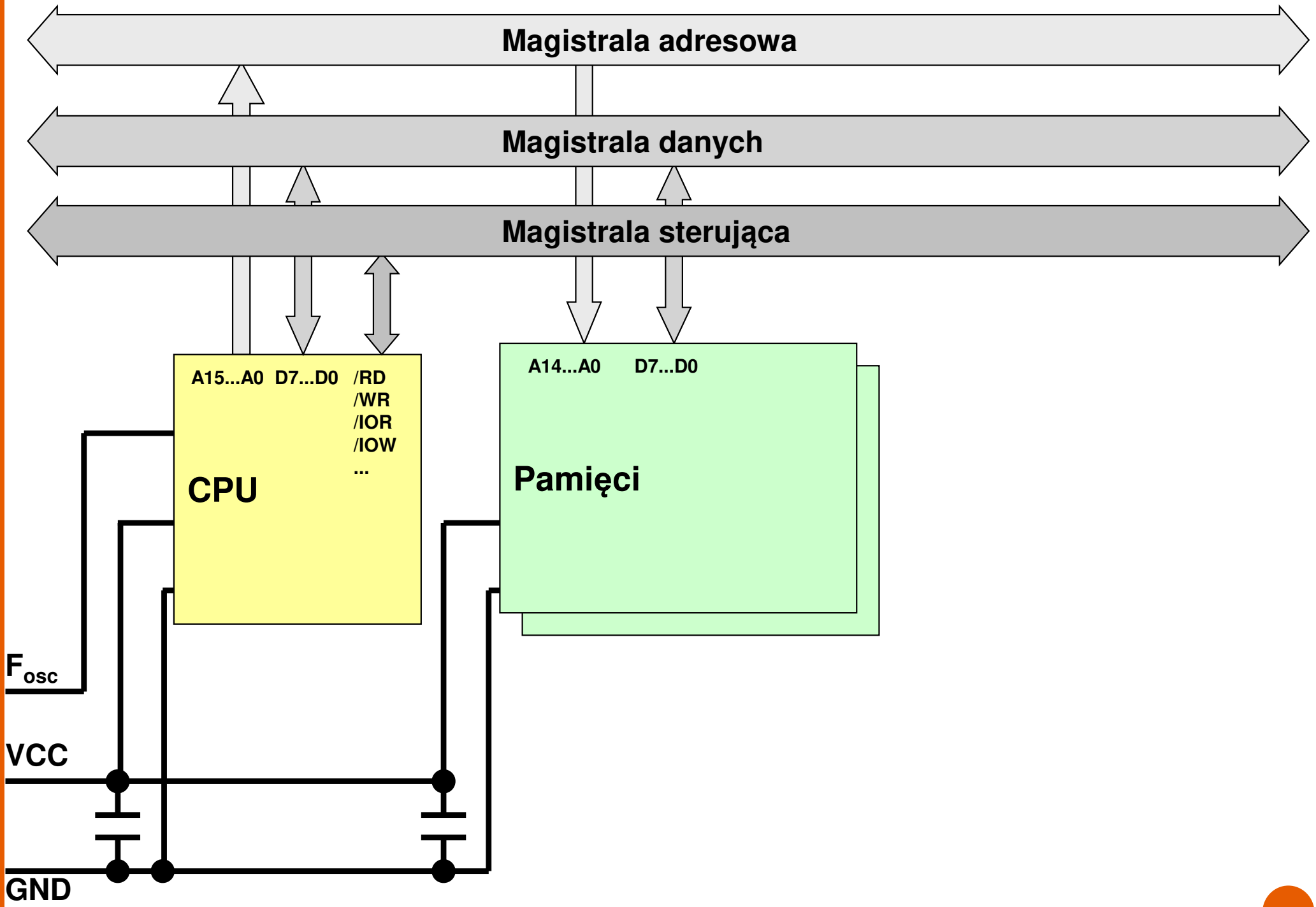
- wspólny potencjał GND!!!
- zgodność poziomów logicznych

Budowa procesorów i łącznie ich z otoczeniem

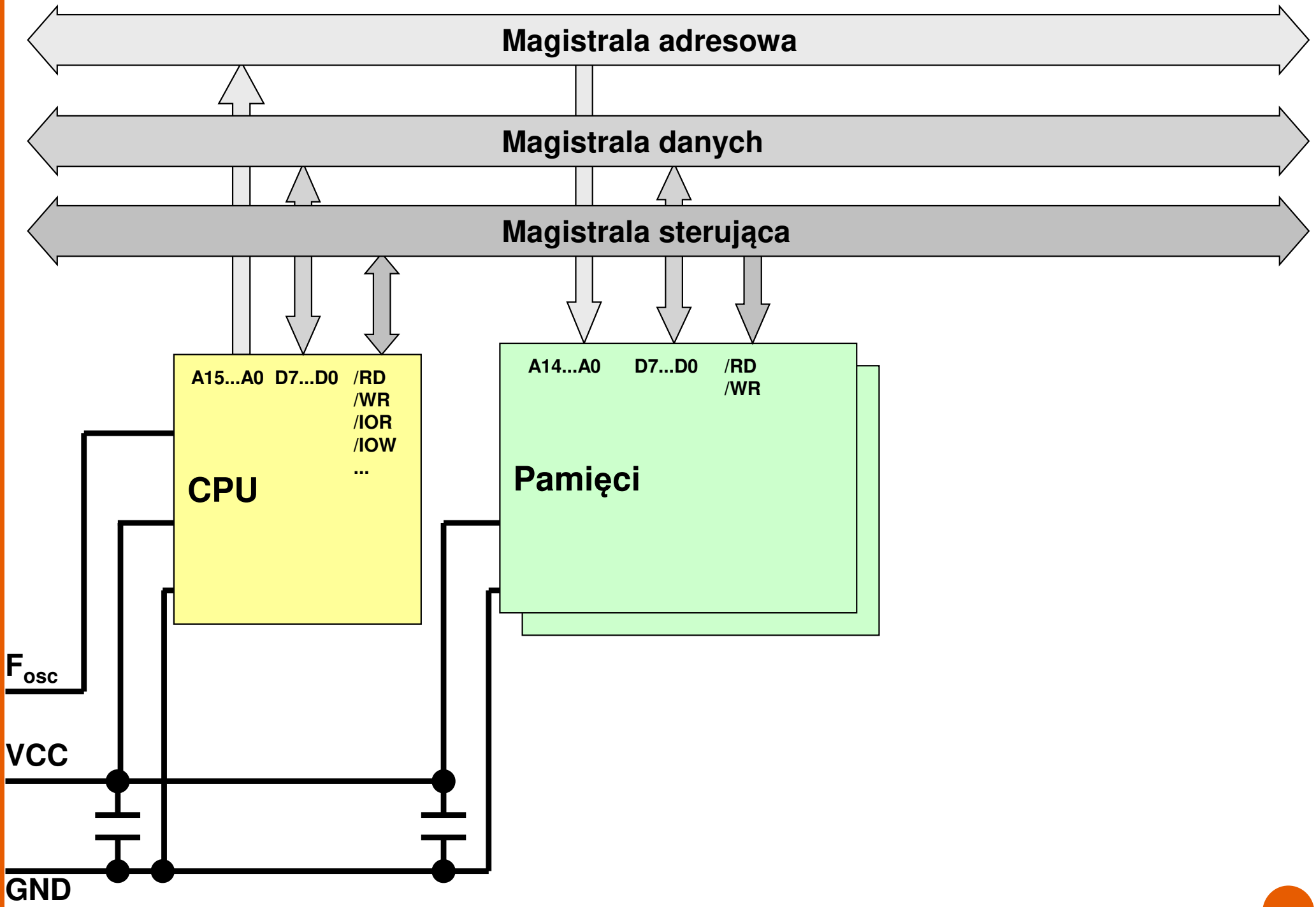


- Pamięć podłączamy do CPU nie wykorzystując wszystkich linii magistrali adresowej?
  - A co z pozostałymi liniami adresowymi?
  - A jak procesor widzi nie obsługiwaną przestrzeń?
- Jeżeli nic nie dołączone, stan tej przestrzeni może być nie ustalony

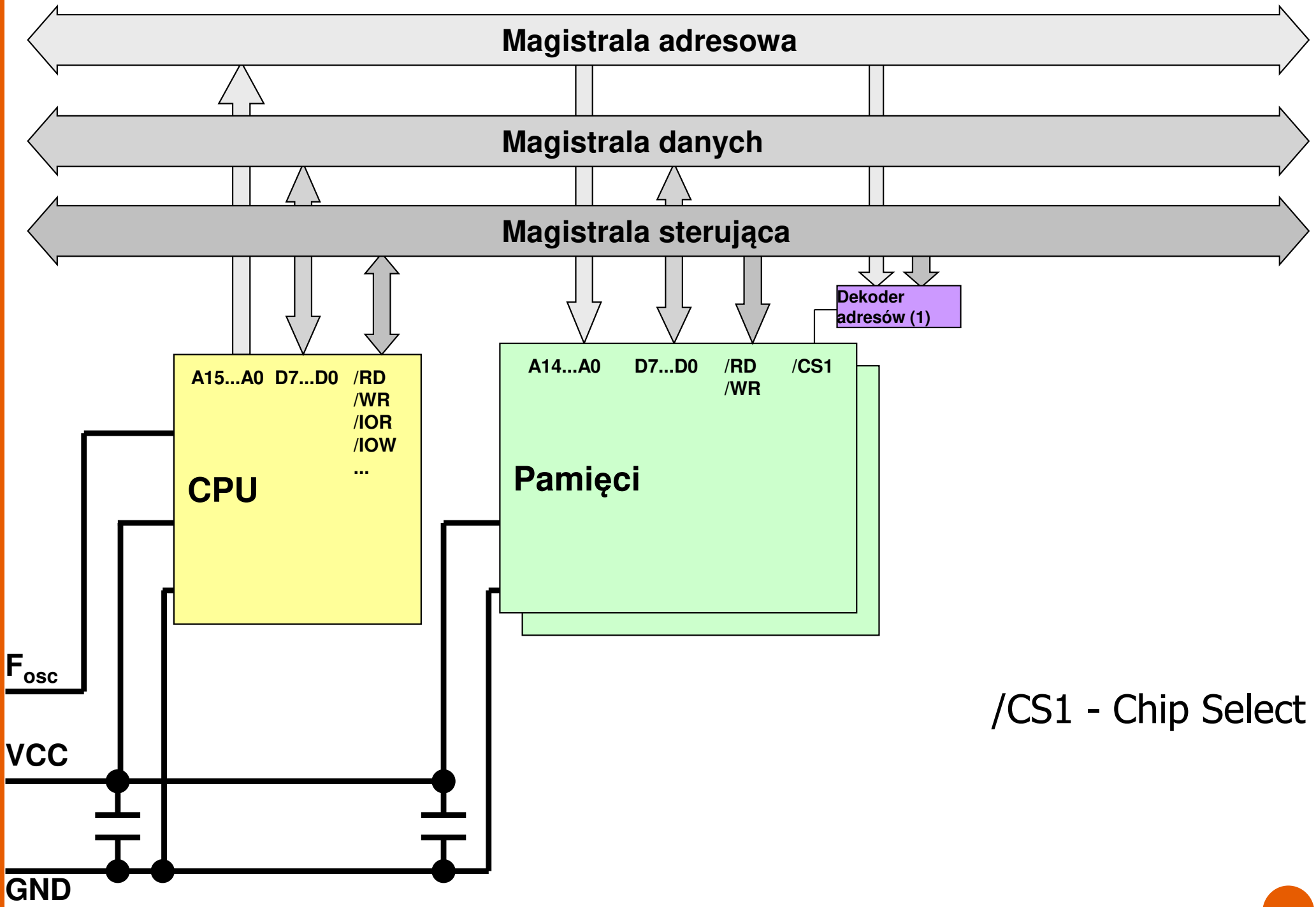
Budowa procesorów i łącznie ich z otoczeniem



Budowa procesorów i łącznie ich z otoczeniem

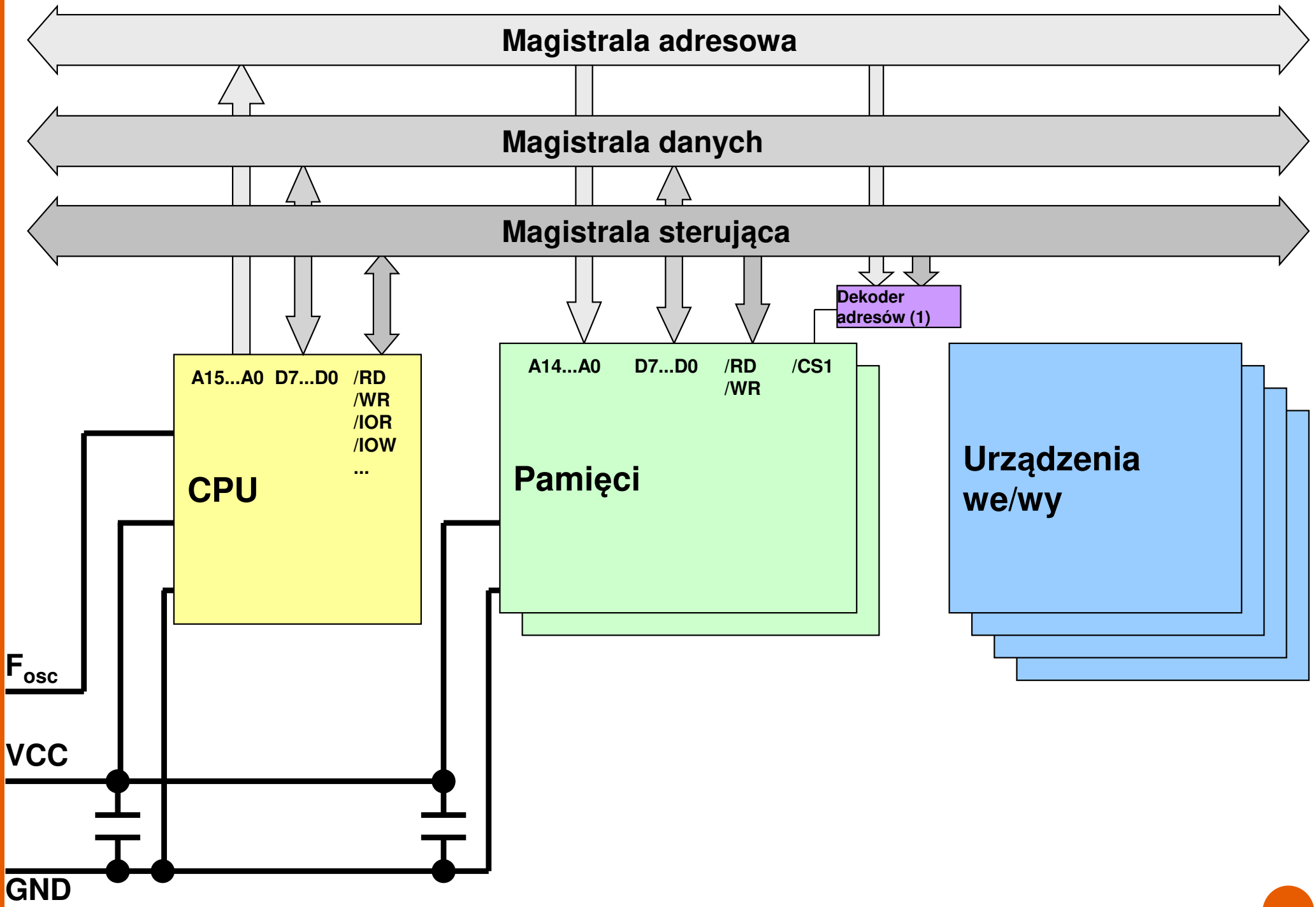


Budowa procesorów i łącznie ich z otoczeniem

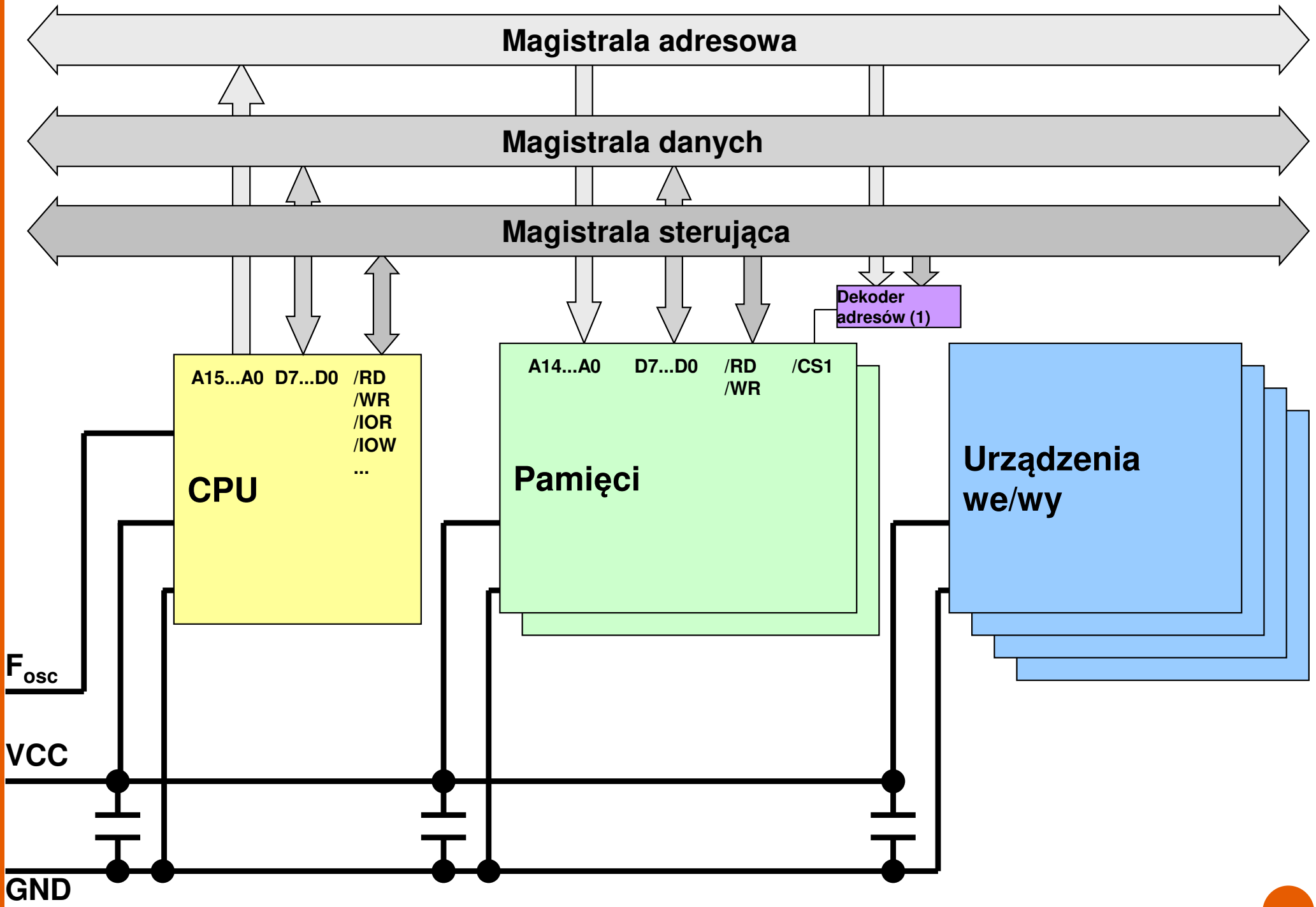


/CS1 - Chip Select

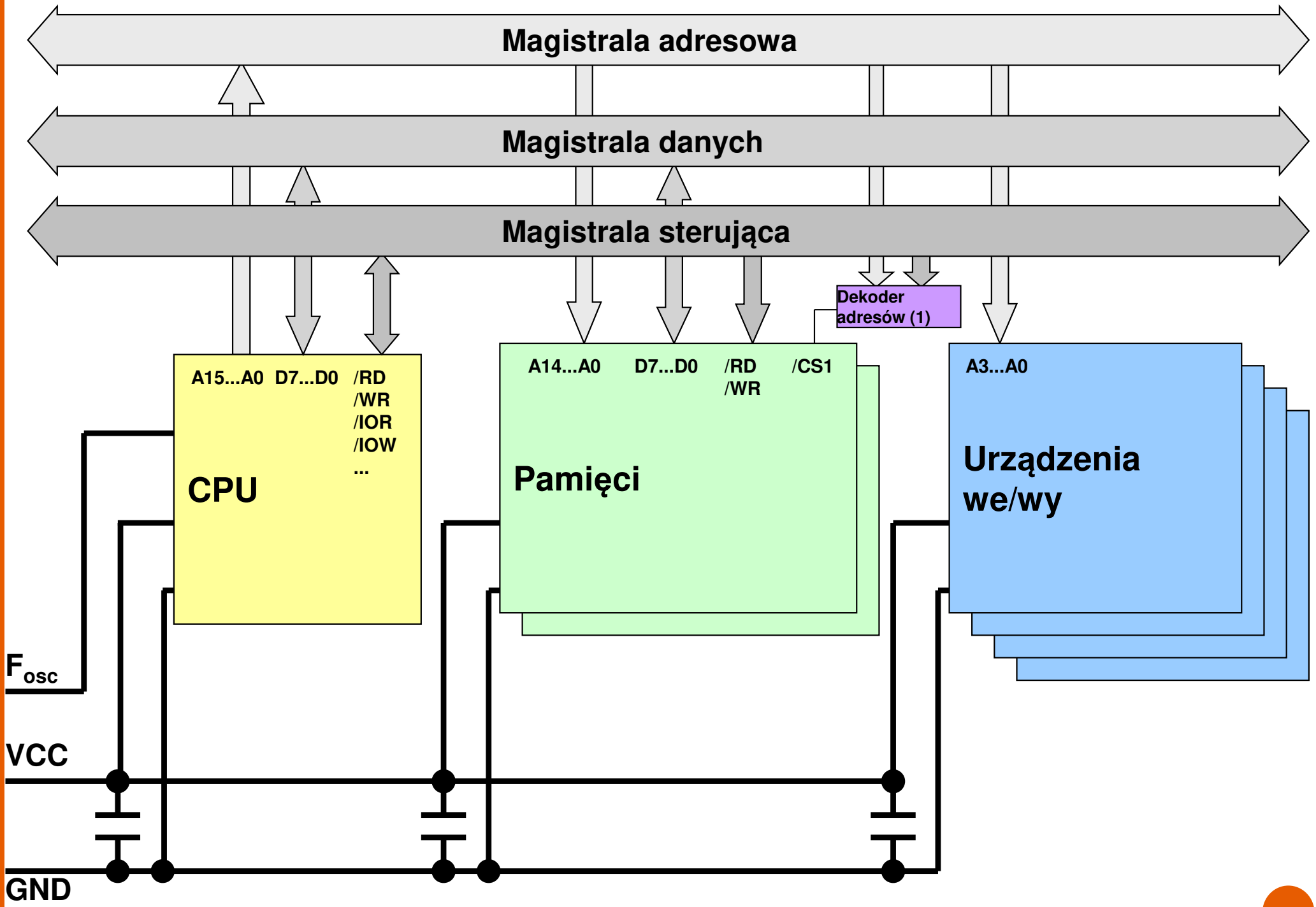
Budowa procesorów i łącznie ich z otoczeniem



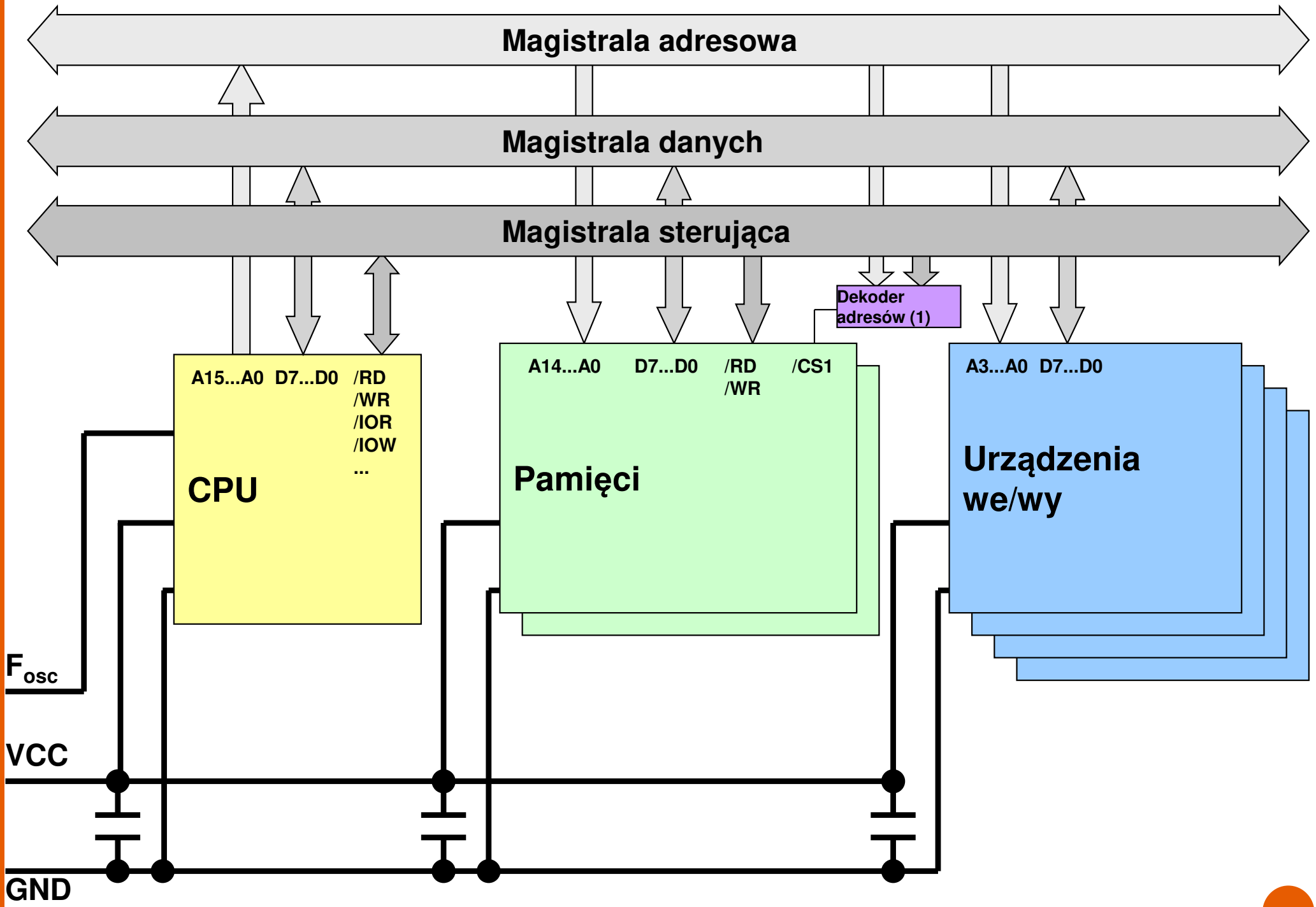
Budowa procesorów i łącznie ich z otoczeniem



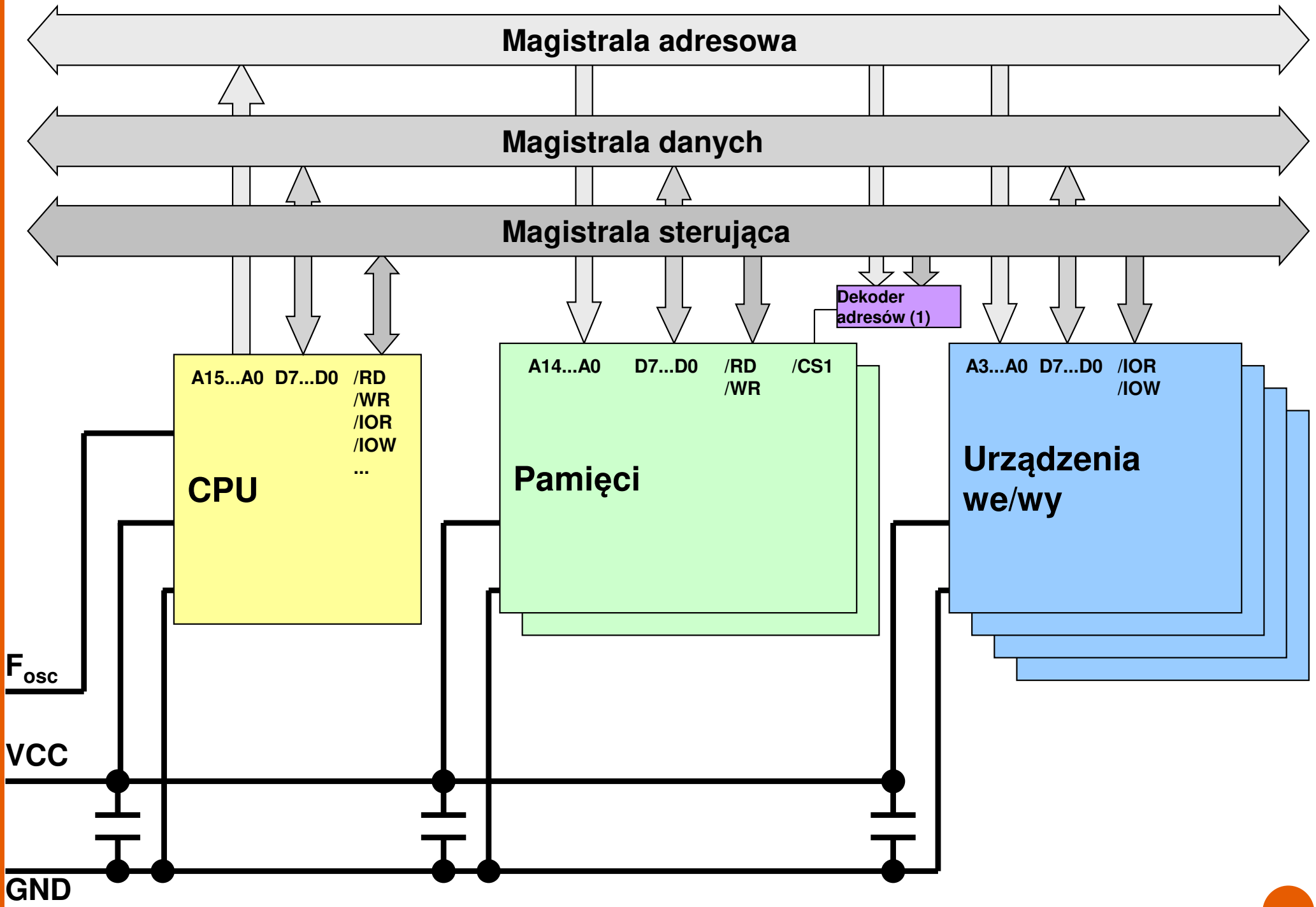
Budowa procesorów i łącznie ich z otoczeniem



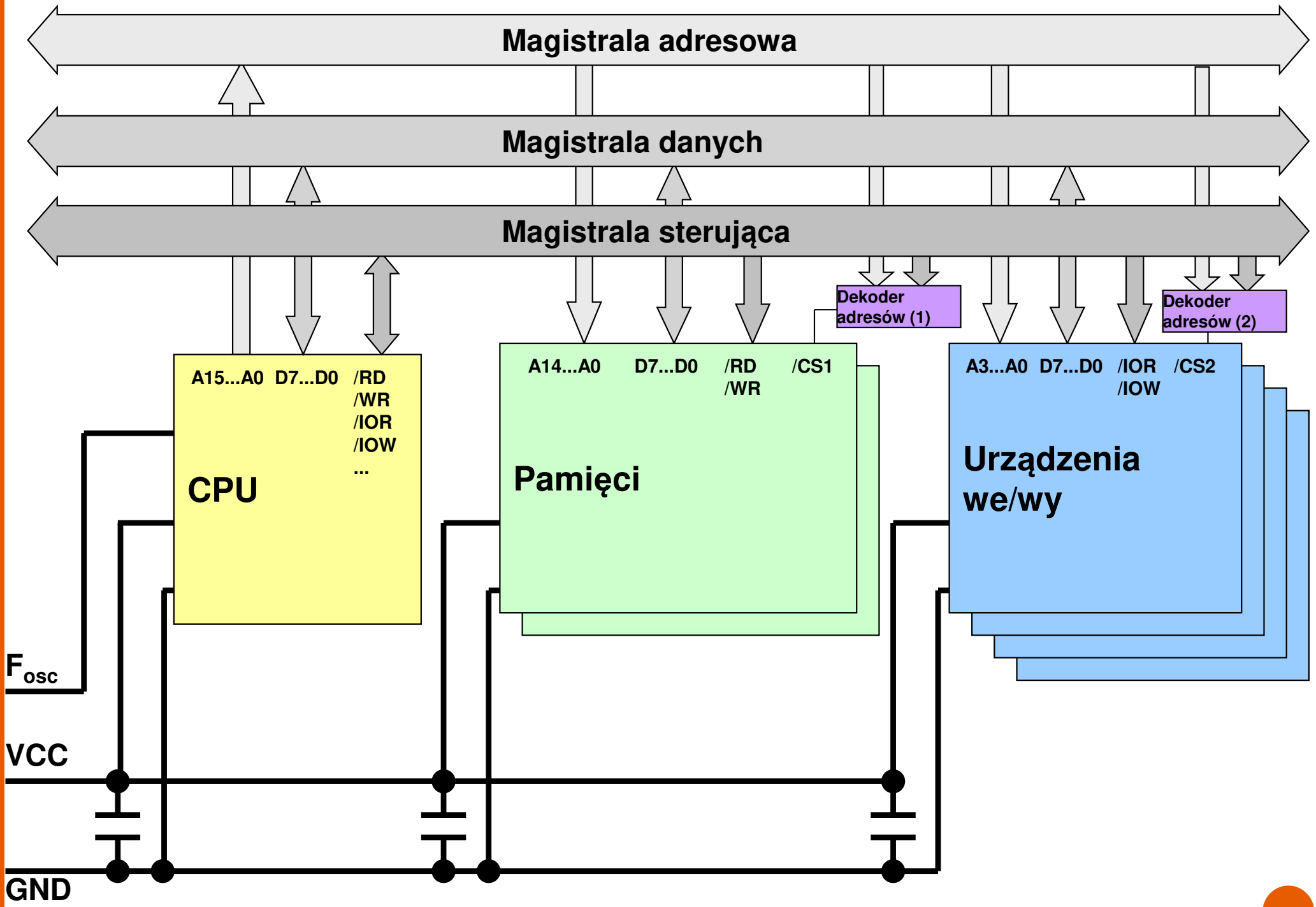
Budowa procesorów i łącznie ich z otoczeniem



Budowa procesorów i łącznie ich z otoczeniem



Budowa procesorów i łącznie ich z otoczeniem



- A jak właściwie działa procesor?
  1. **Pobierz** kod operacji
     - Kod operacji to elementarna jednostka kodu
       - Odpowiadająca pojedynczej instrukcji procesora
       - Może posiadać tzw. argumenty wbudowane
  2. **Zdekoduj**
     - Dla niektórych instrukcji wymagane jest pobranie tzw. argumentu zewnętrznego
  3. **Wykonaj** (i zapisz wynik)
  4. **Analizuj** stan peryferii (np.: stan przerwań)
     - Operacja opcjonalna
  5. **Skacz** do następnej instrukcji

Skąd pobiera kod operacji po RESET – ze zdefiniowanego przez producenta adresu, np.: 0 (AVR, ARM, RISC-V), 0xFFFF0 (X86)

Procesor RISC-V

Procesor RISC-V

- RISC-V
  - Nowość na rynku
  - Otwarta ISA (Instruction Set Architecture) z typowy RISC
    - brak opłat licencyjnych za tworzenie własnych mikroprocesorów czy mikrokontrolerów
    - specyfikacja żyje (!)
      - ostatnia zatwierdzona wersja 2019.12.13
      - [https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf]
      - specyfikacja wprowadza pojęcie „hart” będącym odpowiednikiem sprzętowych wątków, których może być wiele w CPU, i każdy z nich może wykonywać swój własny program

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	**Version**	**Status**
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

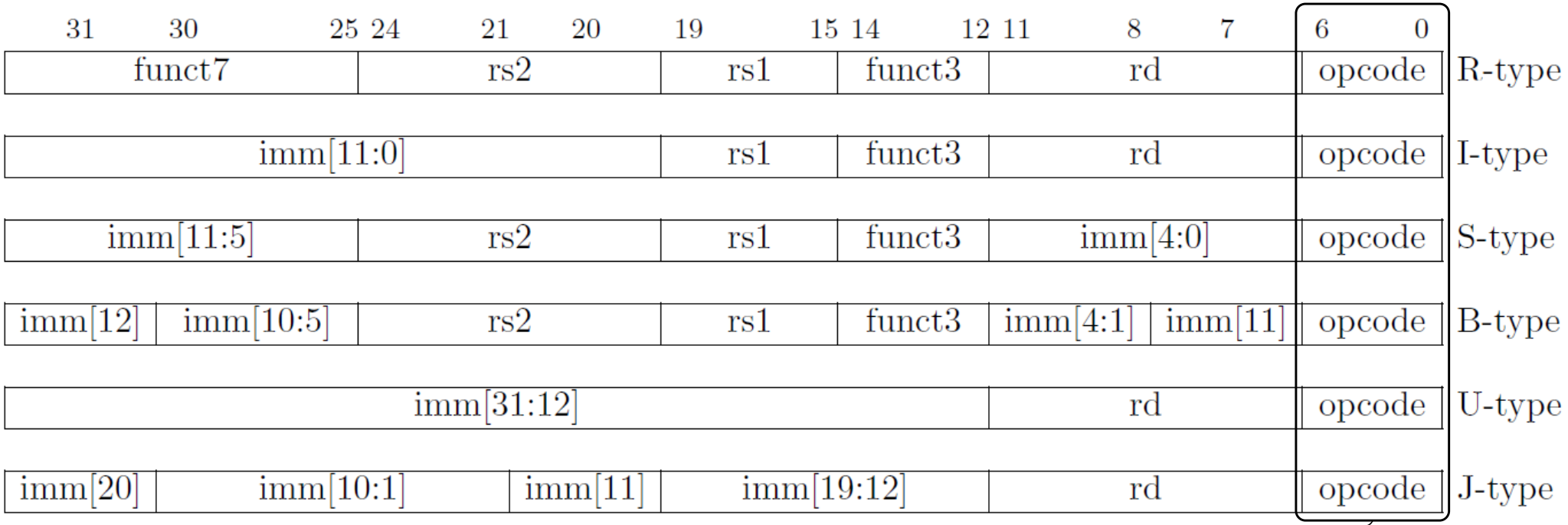
Źródło: riscv.org

- RISC-V
  - Otwarta ISA (Instruction Set Architecture) z typowy RISC, cd.
    - specyfikacja ustala wiele wersji CPU - każda ma własną „ISA”
  - RV32I
    - RV32M (dodane operacje mnożenia), RV32A (operacje atomowe), RV32F (operacje zmienno-przecinkowe), RV32D (operacje zmienno-przecinkowe rozszerzone dwukrotnie), RV32Q (operacje zmienno-przecinkowe rozszerzone czterokrotnie)
  - RV64I
    - RV64M, RV64A, RV64F, RV64D, RV64Q
  - RV128I
  - RV32E - tzw. „embedded microcontrollers”, jak RV32I ale tylko z rejestrami x0...x15
  - RV32C, RV64C, RV128C - skompresowane instrukcje (większość ma 16bitów)

- RISC-V - rejestry
  - w specyfikacji RISC-V zdefiniowano stałą XLEN która określa liczbę bitów rejestrów stało przecinkowych
  - 32 rejestry 32 bitowe (lub 64bitowe w CPU wersji 64bitowej, ...) stało przecinkowe
    - x0 (alias: Zero) - zawsze wartość zero, nie można zmienić jego treści
    - x1 (ra) - adres powrotu (Return Address)
    - x2 (sp) - wskaźnik stosu (Stack Pointer)
    - x3 (gp) - globalny wskaźnik (Global pointer)
    - x4 (tp) - wskaźnik wątku (Thread pointer)
    - x5...x7 (t0...t2) - rej. tymczasowe
    - x8 (s0/fp) - „zachowany rejestr” (ang. saved register) / wskaźnik ramki (Frame Pointer)
    - x9 (s1) - „zachowany rejestr”
    - x10...x11 (a0...a1) - argumenty funkcji / rezultat wykonania funkcji
    - x12...x17 (a2...a7) - kolejne argumenty funkcji
    - x18...x27 (s2...s11) - „zachowane rejestry”
    - x28...x31 (t3...t6) - rej. tymczasowe

Saved register – są używane do przechowywania tymczasowych wartości zachowywanych podczas wywołania

- RISC-V - instrukcje
  - Bazowy formaty instrukcji (typy i metoda kodowania)



  - Instrukcje procesora mają postać

`„kod operacji”` `rd, rs1, rs2`

  - `rd` - argument wynikowy (gdzie wynik zostanie zapisany)
  - `rs1` - argument źródłowy 1
  - `rs2` - argument źródłowy 2
  - `imm` - argument wbudowany

Procesor RISC-V

- RISC-V - instrukcje
  - Tzw. „Reference card” - baza opisująca ISA wszystkich architektur bliźniaczych

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

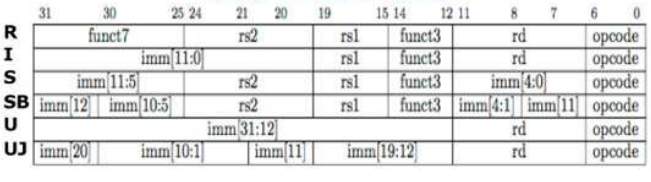
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

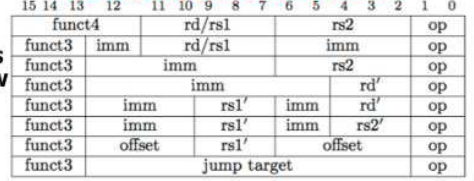
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Źródło: linuxgizmos.com

- RISC-V - instrukcje RV32I
  - Instrukcje typu U
    - działania opierają się na liczbach całkowitych zapisanych w OPCODE (stałe 20 bitowe)
    - LUI
      - załadowanie do górnych 20 bitów rejestru 'rd' zawartości z 'imm'* (dolne bity ustawione na 0)
      - rd = imm[32...12] | zero[11...0]

```
lui a1,0x21
```

    - AUIPC
      - tworzy 32bitowy offset sumując PC z górnymi 20bitami z 'imm'* (przyjmuje dolne 12 bitów jako równe zero)
      - rd = PC + (imm[32...12] | zero[11...0])

```
auipc ra,0x0
```

*) imm jest tu rozszerzane ze znakiem z 12/20 bitowego do pełnej długości (32bitów)

- RISC-V - instrukcje RV32I
  - Instrukcje typu I
    - działania opierają się na liczbach całkowitych zapisanych w rejestrach i w OPCODE (stałe 12 bitowe)
    - ADDI
      - dodawanie imm*) do rs1 i zapisanie w rd

        ```
        addi sp,sp,-16
        ```

      - pseudo instrukcje: `mv rd,rs1` może być emulowana przez: `addi rd,rs1,0`
        - rozdział „RISC-V Assembly Programmer's Handbook" specyfikacji zawiera „Table 25.2: RISC-V pseudo instructions" opisująca wiele możliwości emulowania nie istniejących instrukcji
    - SLTI | SLTIU
      - wstawia wartość 1 do rd jeżeli rs1 jest większy od imm*, w przeciwnym przypadku wstawia 0 do rd | jak w SLTI ale porównanie wykonywane jest bez znaku
    - ANDI | ORI | XORI
      - operacje logiczne iloczynu | sumy | alternatywy rozłącznej na rs1 oraz imm* z zapisaniem wyniku w rd
      - pseudo instrukcja `NOT` może być zrealizowana przez: `XORI rd, rs1, -1`
    - SRLI | SRAI
      - logiczne (starsze bity zapełniane zerami) | arytmetyczne (starsze bity zapełniane bitem znaku) przesunięcie w prawo

- RISC-V - instrukcje RV32I
  - Instrukcje typu R
    - działania opierają się na liczbach całkowitych zapisanych w rejestrach
    - ADD | SUB
      - jak w instrukcjach typu I
      - rd = rs1 + rs2 | rd = rs1 - rs2
    - AND | OR | XOR
      - jak ANDI | ORI | XORI ale na rejestrach
    - SLT | SLTU
      - wstawia wartość 1 do rd jeżeli rs1 jest większy od rs2, w przeciwnym przypadku wstawia 0 do rd | jak SLT ale bez znaku
    - SLL
      - logiczne przesunięcie w lewo, młodsze bity zapełniane zerami
      - rd = rs1 << rs2[5..0]
    - SRL | SRA
      - logiczne | arytmetyczne przesunięcia w prawo
      - rd = rs1 >> rs2[5..0]

- RISC-V - instrukcje RV32I
  - Instrukcje typu J
    - instrukcje sterowania transferem (skoki)
    - JAL
      - bezwarunkowy skok relatywny (ang. relative jump)
      - adres skoku wyznacza: PC+imm2
        - imm2 to imm[19...0] rozszerzone ze znakiem i pomnożone przez 2 (tzw. word alignment)
      - PC użyte do operacji wyznaczania adresu wskazuje na aktualna instrukcje, ta wartość powiększona o 4 (następna instrukcja) trafia też do rd (typowo używane jest x1 lub x5)
      - zasięg skoku +/- 1MiB
    - JALR
      - bezwarunkowy skok niebezpośredni (ang. indirect jump)
      - imm jest tylko 12 bitową wartością ze znakiem
      - adres skoku wyznacza: rs1+imm2

- RISC-V - instrukcje RV32I
  - Instrukcje typu B
    - w architekturze ***nie przewidziano*** rejestru flag przechowującego status po wykonanych operacjach(!), uznano że instrukcje rozgałęzienia z wbudowanym porównaniem rozwiążą problem
    - BEQ | BNE
      - skok gdy rs1 i rs2 są sobie równe | są różne
      - gdy warunek nie jest spełniony - przejście do następnej instrukcji a gdy ma być wykonany skok - nowe miejsce w kodzie wyznacza: PC+imm2*2
      - imm2 to imm[12...0] rozszerzone ze znakiem i pomnożone przez 2 (tzw. word alignment)
    - BLT | BLTU
      - jak powyższe tyle, że skok gdy rs1 mniejsze od rs2 traktowane ze znakiem | bez znaku
    - BGE | BGEU
      - jak powyższe tyle, że skok gdy rs1 większe od rs2 traktowane ze znakiem | bez znaku
    - Operacje: BGT, BGTU, BLE oraz BLEU mogą być łatwo zrealizowane przez zamianę w kodzie miejscami rs1 z rs2 i użycie BLT, BLTU, BGE oraz BGEU
      - prostszy „krzem” bo mniej mnemoników, ale utrudnienie dla twórców kodu
    - Zasięg ewentualnego skoku to +/- 4KiB

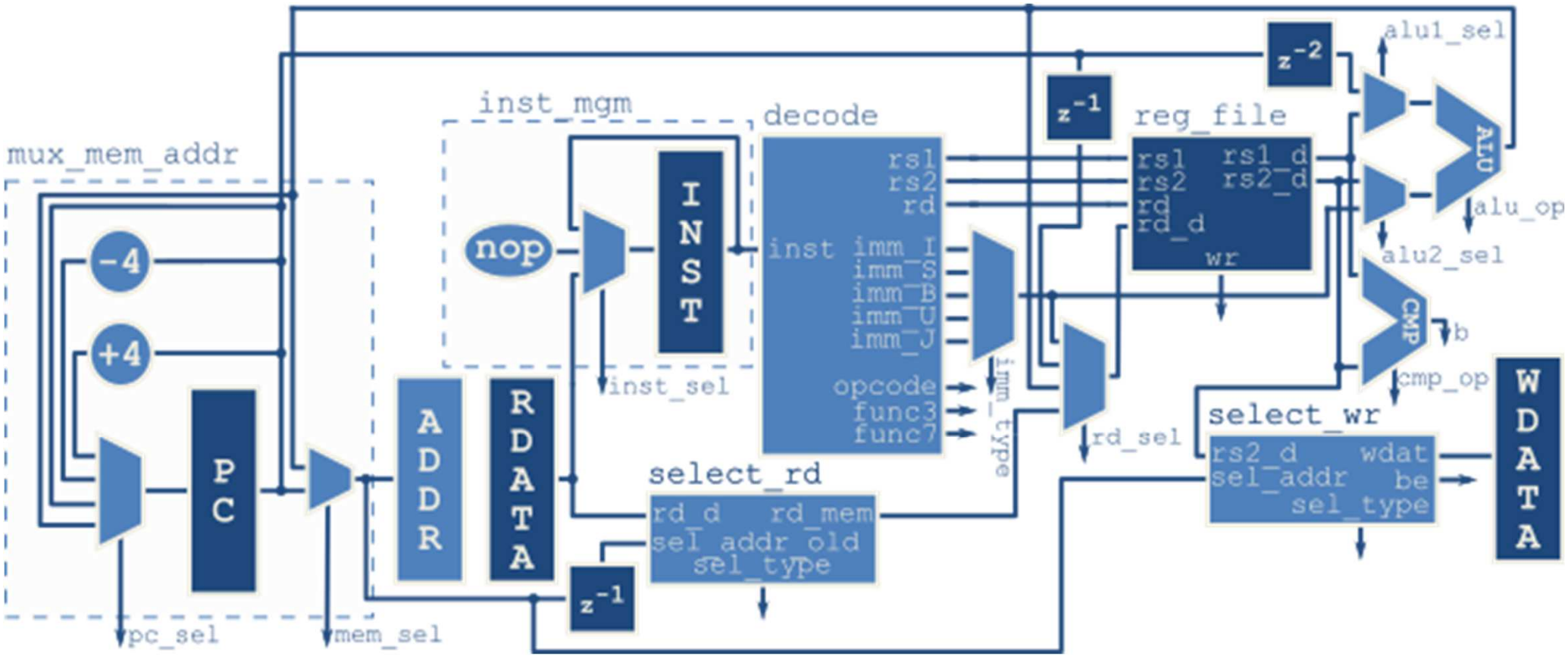
- RISC-V - instrukcje RV32I
  - Architektura typu Load - Store
    - ograniczone tryby adresowania (typowe dla arch. Risc)
  - Dostęp do pamięci możliwy jedynie za pomocą dwóch instrukcji
    - efektywny adres odwołania do pamięci to rs1+imm[11...0]
    - LOAD - typ I
      - lw - załaduj do rd z pamięci spod efektywnego adresu słowo 32bitowe

```
lw ra,12(sp)
```

      - lh | lhu - załaduj do rd z pamięci spod efektywnego adresu 16bitów, pozostałe bity w rd zapełnij bitem znaku | zerami
      - lb | lbu - załaduj do rd z pamięci spod efektywnego adresu 8bitów, pozostałe bity w rd zapełnij bitem znaku | zerami
    - STORE - typ S
      - sw | sh | sb - zapisz w pamięci pod efektywny adres wartość 32bitową | 16bitową | 8bitową z rs2
      - dla sh i sb używaj młodszych bitów z rs2

- RISC-V - instrukcje RV64I
  - Wspiera te same instrukcje co RV32I
    - operuje na 64bitach - gdzie potrzeba rozszerza argumenty i wynik do takiej liczby bitów
    - dostęp do pamięci
      - lw | lwu | lh | lhu | lb | lbu | sw | sh | sb - operują na przestrzeni 64bitowej
      - ld | sd - zapewniają odczyt | zapis 64bitów z | w pamięci z przestrzeni 64bitowej
- RISC-V - instrukcje RV128I
    - operuje na 128bitach
    - przestrzeń pamięciowa 128bitowa
    - dodano instrukcje odwołań do pamięci
      - LQ i SQ do obsługi pamięci za pomocą „quadword” (poczwórnych słów)
- RISC-V - instrukcje RV32C
    - OPCODE 16 bitowe
    - oznaczenia mnemoników zaczynają się od C np.: C.ANDI, C.ADD, C.BEQZ, ...
    - Posiada instrukcję C.NOP

- RISC-V - przykład architektury
  - Rysy Core (polski wkład)
    - open source (BSD), wspiera ISA: RV32I
    - obok projektu jest symulator: http://rysy_core.gitlab.io/venus/



Źródło: https://gitlab.com/rysy_core/rysy_core

Procesor x86

- X86 - Rejestry
  - Rejestry segmentowe (16bitowe)
    - CS - segment kodu
    - DS – segment danych
    - SS – segment stosu
    - ES | FS | GS - segmenty dodatkowe

```
add al, CS:EBX
add DS:EBX, 12
```

  - Rejestry ogólnego przeznaczenia (32bitowe)
    - EAX - akumulator
    - EBX - adres pośredni/przemieszczenie
    - ECX - licznik pętli
    - EDX - wskaźnik portu I/O

- X86 - Rejestry
  - Rejestry wskaźnikowe i indeksowe (32bitowe)
    - ESI - indeks źródłowy
    - EDI - indeks docelowy
    - EBP - wskaźnik bazowy segmentu stosu np.: SS:EBP
    - ESP - wskaźnik stosu

```
add EAX, DS:ESI
add DS:EDI, EAX
```

  - Dla zgodności z kodem dla poprzednich generacji komputerów można używać:
    - rejestry 16 bitowe: AX, BX, CX, DX, BP, DI, SI, SP
    - rejestry 8 bitowe rejestry AL, BL, CL, DL, AH, BH, CH, DH
  - Wskaźnik rozkazów EIP = PC
    - element 32bitowy
  - X86 w wersji 64bitowej
    - posiada 64bitowe rejestry z przedrostkiem „R” zamiast „E” np.:
      - RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, ... oraz dodatkowe rej. 64bitowe: r8...r15
      - EIP nazywa się RIP (;->)

- X86 - Rejestry
  - Rejestr stanu EFLAGS (32bitowy) - zawiera bity
    - CF (carry flag) - przeniesienie z najbardziej znaczącego bitu
    - PF (parity flag) - „1” gdy liczba jedynek wyniku ostatniej operacji jest parzysta
    - AF (auxiliary flag) - „1” gdy występuje przeniesienie z bitu numer 3 (kod BCD)
    - ZF (zero flag) - „1” gdy wynik ostatniej operacji jest zero
    - SF (sign flag) - „1” gdy wynik operacji jest ujemny (kod U2)
    - OF (overflow) - „1” gdy wynik operacji przekracza największą dodatnią lub najmniejszą ujemną liczbę reprezentowalną w kodzie U2 (nadmiar stałoprzecinkowy)
    - TF (trap flag) - wymuszenie pracy krokowej (po rozkazie skok do przerwania INT1)
    - IF (interrupt flag) - blokada przerwań maskowalnych
    - DF (direction flag) - zmiana SI/DI w kierunku adresów rosnących czy malejących
    - VM (virtual mode) - flaga ustawienia trybu wirtualnego 8086

- X86 - Rejestry
  - Rejestry sterujące:
    - CR0...CR7
      - 32bitowe rejestry sterujące pracą procesora (control registers): tryb pracy, tryb pracy pamięci podręcznej "cache"
        - Np.: CR3 - używany w translacji pamięci związanej ze stronicowaniem
    - DR0...DR7
      - 32bitowe rejestry uruchomieniowe (debug registers): adresy pułapek sprzętowych, status, …
    - TR3...TR7,TR12
      - 32bitowe rejestry przeznaczone do testowanie procesora (test registers)
    - GDTR (Global Descriptor Table Reregister), LDTR (Local Descriptor Table Register)
      - 48bitowe rejestry sterujące segmentacją pamięci
    - IDTR (Interrupt Descriptor Table Register), TR (Task Register)
      - 16bitowe rejestry związane z działaniem programów/procesów podczas gdy włączono segmentację pamięci

- X86 - Instrukcje
  - Instrukcje - typowo dwu argumentowe

```
„kod operacji”  arg.d, arg.z
```

  - Rozkazy przesłania
    - zwykłe
      - MOV - prześlij zawartość
      - XCHG - wymień wzajemnie zawartości
      - SETC - ustaw argument docelowy na 1 gdy CF=1
      - SETZ - ustaw argument docelowy na 1 gdy ZF=1
      - SET...
    - stosowe
      - PUSH | POP - włóż na | zdejmij ze stosu rejestr
        - stos - specjalna przestrzeń pamięci danych, zarządzana przez rejestr SP/ESP
      - PUSHA | POPA - włóż na | zdejmij ze stosu cały zestaw rejestrów
        - z zachowaniem kolejności: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
      - PUSHF | POPF - włóż na | zdejmij ze stosu rejestr EFLAGS
      - ENTER | LEAVE - rezerwacja | zwolnienie na stosie ramki dla zmiennych lokalnych

Uwaga!
Dwie notacje:

Intel:
```
mov eax, 5
```

AT&T:
```
movl $5, %eax
```

- X86 - Instrukcje
  - Rozkazy przesłania
    - specjalne
      - LAHF | SAHF- załaduj do AH zawartość EFLAGS | zapisz AH w EFLAGS
      - LDS,LES,LFS,LGS,LSS - załaduj 48bitowy daleki adres (segment : przemieszczenie)
      - LEA - zapisz efektywny adresu argumentu źródłowego

        ```
        lea eax, [ebp-4]
        ```
      - ARPL - zmiana pola RPL selektora
      - LGDT | SGDT - załaduj do | zapisz z rejestru Globalnej Tablicy Deskryptorów
      - LIDT | SIDT - załaduj do | zapisz z rejestru Tablicy Deskryptorów Przerwań
      - LLDT | SLDT - załaduj do | zapisz z rejestru Lokalnej Tablicy Deskryptorów
      - LMSW - załaduj CR0 stanem maszyny
      - LSL - załaduj wielkość wskazanego segmentu do rejestru (przy włączonej segmentacji)
      - LTR | STR - załaduj do | zapisz z rejestru zadania
      - CPUID - identyfikacja jednostki centralnej (CPU), zwraca np.:
        - Later Intel 486 and Pentium 0x01
        - Xeon 0x02
        - Atom 0x0A
        - IvyBridge-based processors 0x0D
        - ...

- X86 - Instrukcje
  - Rozkazy arytmetyki całkowitoliczbowej
    - ADD | ADC - dodanie | dodanie z uwzględnieniem flagi CF
    - SUB | SBB - odjęcie | odjęcie z uwzględnieniem flagi CF
    - INC | DEC - zwiększ | zmniejsz o 1 (lub inną wartość!)
    - CMP - porównaj
    - NEG - zaneguj (dst=0-src)
    - MUL | IMUL - mnożenie całkowitoliczbowe argumentu bez znaku | ze znakiem
    - DIV | IDIV - dzielenie całkowitoliczbowe argumentu bez znaku | ze znakiem
      - generuje przerwanie „dzielenia przez zero”
    - XADD - jak XCHG i dodawanie
      - T=src+dst; src=dst; dst=T, gdzie T rej. Tymczasowy, żaden z dostępnych dla programisty

- X86 - Instrukcje
  - Rozkazy arytmetyki BCD
    - DAA | DAS - korekcja dodawania | odejmowania w upakowanym kodzie BCD
    - AAA | AAS - korekcja dodawania | odejmowania w nie upakowanym kodzie BCD
    - AAD - korekcja dzielnej w nie upakowanym kodzie BCD przed dzieleniem
    - AAM - korekcja wyniku w nie upakowanym kodzie BCD po mnożeniu

- X86 - Instrukcje
  - Operacje bitowe
    - Operacje logiczne
      - NOT - negacja boolowska
      - AND - iloczyn boolowski
      - OR - suma boolowska
      - XOR - alternatywa wykluczająca
    - Operacje na bitach
      - TEST - porównanie logiczne dwóch argumentów (jak CMP tyle że operacją AND)
      - BSF | BSR - przeszukanie w przód | w tył, argumentu w celu znalezienia pozycji bitu równego „1”
      - BT - skopiowanie do CF wartości bitu o numerze w arg2, z arg1
      - BTS | BTR | BTC - skopiowanie do CF wartości bitu o numerze w arg2, a w arg1 w jego miejsce ustawienie | wyzerowanie | zanegowanie bitu wskazanego w arg2

- X86 - Instrukcje
  - Rozkazy przemieszczeń i obrotów
    - SAL | SAR - przesunięcie arytmetyczne w lewo | prawo z powielaniem najstarszego | najmłodszego bitu, nie mieszczące się bity kopiowane są do CF
    - SHL | SHR - przesunięcie arytmetyczne w lewo | prawo z zerowaniem najstarszego | najmłodszego bitu, nie mieszczące się bity kopiowane są do CF
    - SHLD | SHRD - jak SHL | SHR ale operacje wykonywane są na podwójnej precyzji argumencie
    - RCL | RCR - rotacja w lewo | prawo, nie mieszczące wstawiane są do CF a wcześniejsza wartość CF kopiowana na nastraszy | najmłodszy bit
    - ROL | ROR - rotacja w lewo | prawo, nie mieszczące wstawiane są do CF i kopiowane na nastraszy | najmłodszy bit

- X86 - Instrukcje
  - Rozkazy sterowania
    - Sterowanie programem
      - JMP - skok bezwarunkowy
      - JA | JNBE | JAE | JNB | JB | JNAE | JBE | JNA | JC | JCXZ | JE | JECXZ | JZ | JNC | JNE | JNZ | JNP | JPO | JP | JPE | JG | JNLE | JGE | JNL | JL | JNGE | JLE | JNG | JNO | JNS | JNZ | JO | JS - skoki warunkowe jeżeli ponad | nie poniżej lub równe | ...
      - LOOP | LOOPZ | LOOPNE | LOOPNZ - rozkaz pętli wykonywanej aż warunek w którym licznik ECX=0 | ECX=0 i ZF=1 | ECX=0 i ZF=0 będą spełnione
        - LOOPZ = LOOPE, LOOPNZ = LOOPNE
    - REP | REPE(Z) | REPNE(Z) - powtarzaj następną instrukcję dopóki warunek ECX=0 | ZF=0 | ZF=1 będzie spełniony
      - REPE = REPZ, REPNE = REPNEZ
    - BOUND - sprawdzenie czy indeks podany jako arg1. leży w tablicy wskazanej przez arg2, i ograniczonej przez jej ostatni element wskazany przez arg3
    - LOCK - wygenerowanie przez CPU sygnału LOCK#, w systemach wieloprocesorowych zapewnia to wykluczający dostęp do pamięci dzielonej z innymi procesorami

- X86 - Instrukcje
  - Rozkazy sterowania
    - Podprogramy i przerwania
      - CALL - skok do podprogramu
        - dziejesz języki programowania C/C++/Java/Python bardzo mocno korzystają z funkcji/procedur
        - typowo programista w ciele funkcji/procedury na jej początku zachowuje na stosie wszystkie używane przez siebie w funkcji/procedurze rejestry
      - RET - powrót z podprogramu
        - typowo programista przed tą instrukcją odtwarza wszystkie używane przez siebie i zachowane w prologu funkcji/procedurze rejestry ze stosu w kolejności odwrotnej do zapamiętanych w CALL
      - IRET - powrót z podprogramu obsługi przerwania (przywraca rejestr EFLAGS)
        - rejestr EFLAGS w tym procesorze zapamiętywany jest automatycznie podczas wywołania podprogramu obsługi przerwań - celem transparentnego wykonania takiego kodu
      - INT - przerwanie programowe
        - można je wywołać jawnie w kodzie
      - INTO - wywołanie podprogramu obsługi wyjątku nadmiaru

- X86 - Instrukcje
  - Rozkazy sterowania
    - Sterowanie systemem
      - HLT - zatrzymanie CPU aż do
        - przejścia w stan RESET
        - wywołana przerwania sprzętowego
      - WAIT - wprowadzenie CPU w stan oczekiwania aż pojawi się sygnał na końcówce WAIT#
        - synchronizowanie pracy z koprocesorem
      - NOP - rozkaz pusty
      - RSM - wznowienie pracy z trybu zarządzania systemem (System Management Mode)
      - INVD | WBINVD - opróżnienie pamięci podręcznej | opróżnienie i skopiowanie treści do głównej pamięci
      - INVLPG - opróżnienie wpisów w pamięci TLB (Translation Lookaside Buffer) - część MMU
      - VERR | VERW - sprawdzenie możliwości odczytu z | zapisu do segmentu bez generowania błędu ochrony

- X86 - Instrukcje
  - Rozkazy sterowania
    - Operacje na znacznikach
      - STC | CLC | CMC - ustawianie | zerowanie | negowanie flagi CF
      - STD | CLD - ustawianie | skasowanie znacznika DF
      - STI | CLI - ustawienie | skasowanie znacznika odblokowania przerwań IF
      - CLTS - ustawienie flagi TS (Task-Switched), wymagany poziomo ochrony CPL=0

- X86 - Instrukcje
  - Rozkazy łańcuchowe
    - MOVSB | MOVSW | MOVSD - przesłanie bajtu | słowa | podwójnego słowa z miejsca wskazanego przez ESI na pozycję wskazywaną przez EDI, równocześnie zwiększenie lub zmniejszenie (zależnie od flagi DF) ESI i EDI

```
:[EDI] = :[ESI];
ESI++/--;
EDI ++/--
```

    - CMPSB | CMPSW | CMPSD - porównanie łańcuchowe

```
:[EDI] ?= :[ESI];
ESI++/--;
EDI ++/--
```

    - SCASB | SCASW | SCASD - skanowanie łańcuchowe

```
:[EDI] ?= EAX;
EDI ++/--
```

    - LODSB | LODSW | LODSD - ładowanie łańcuchowe

```
EAX = :[ESI];
ESI ++/--
```

    - STOSB | STOSW | STOSD - zapamiętanie łańcuchowe

```
:[EDI] = EAX;
EDI ++/--
```

- X86 - Instrukcje
  - Rozkazy łańcuchowe, cd.
    - Instrukcja REP wspiera operacje łańcuchowe, wykonywane są one automatycznie aż ZF stanie się 1 (zakładane jest że operacja łańcuchowa po każdym kroku modyfikuje ZF), np.:

```
mov ESI, 0x12345678 //adres miejsca w pamięci
mov AL, 0x00        //szukany znak
rep scasb           //skanuj w poszukiwaniu znaku identycznego jak w AL
...     //tutaj ESI wskaże miejsce w pamięci gdzie jest znaleziony znak
```

- X86 - Instrukcje
  - Rozkazy wejścia-wyjścia
    - IN - odczytanie bajtu lub słowa z portu wejściowego

      ```
      in AL, DX
      ```

    - OUT - zapisano do porty wyjściowego bajtu lub słowa

      ```
      out  DX, AL
      ```

    - INSB | INSW | INSD - łańcuchowe odczytanie bajtu | słowa | podwójnego słowa z portu wejściowego

      ```
      insb ES:[EDI], DX
      ```

      - ES:[EDI] - miejsce gdzie odczytany bajt będzie umieszczony, DX-numer portu
    - OUTSB | OUTSW | OUTSD - łańcuchowe przeniesienie bajta | słowa | podwójnego słowa do portu wyjściowego

      ```
      outsb DX, ES:[ESI]
      ```

      - DX-numer portu, ES:[ESI] - miejsce gdzie odczytany bajt będzie umieszczony

- X86 - Instrukcje
  - Rozkazy konwersji i translacji
    - CBW - konwersja bajtu w słowo, z uwzględnieniem znaku
      - AX <= AL
    - CWD - konwersja słowa w podwójne słowo, z uwzględnieniem znaku
      - DX:AX <= AX
    - CWDE - konwersja słowa w podwójne słowo, z uwzględnieniem znaku
      - EAX <= AX
    - CDQ - konwersja podwójnego słowa w poczwórne słowo
      - EDX:EAX <= EAX
    - BSWAP - konwersja wartości 32bitowej zapisanej z użyciem reprezentacji grubo końcowej w wartość z użyciem cienko końcowej reprezentacji
      - t=dst; dst[0..7]=t[24..31]; dst[8..15]=t[16..23]; dst[16..23]=t[8..15]; dst[24..31]=t[0..7];
    - XLATB - tłumaczenie indeksu (AL) do elementu w tablicy (wskazana przez BX) na wartość elementu (wynik w AL)
      - AL = :[BX+AL]

- X86 - Tryby adresowania
  - Tryb adresowania - sposób odwoływania się w rozkazie do miejsca przechowywania argumentów lub sposób zapisania wyniku
    - Tryb rejestrowy

      `ADD EAX, EBX`

      - informacje które rejestry (tu EAX i EBX) użyć zapisana w OPCODE
    - Tryb natychmiastowy - argument znajduje się w OPCODE

      `ADD EAX, 5`

      - tutaj argument 5 jest przechowywany w OPCODE jako pewna jego liczba bitów (tzw. argument wbudowany)
    - Tryb bazowy (zwany pośrednim rejestrowym)

      `ADD EAX, [EBX]`

      - wskazanie [EBX] to odwołanie do pamięci zgodnie z wartością w EBX
    - Tryb przemieszczeniowy

      `ADD EAX, [245AH]`

      - odwołanie do pamięci z przemieszczeniem równym 245AH, wartość ta z reguły zapisana jako tzw. argument zewnętrzny - czyli jako zestaw bitów zapisanych zaraz po OPCODE o liczbie identycznej jak szerokość OPCODE

- X86 - Tryby adresowania
  - Trybem adresowania - sposób odwoływania się w rozkazie do miejsca przechowywania argumentu, cd.
    - Tryb baza-przemieszczenie

      ```
      ADD EAX, [EBX+4]
      ```

      - adres efektywny gdzie trzeba sięgnąć do pamięci to suma EBX i stałej 4
    - Tryb indeks-skala-przemieszczenie

      ```
      ADD EAX, [8*EBX + 4]
      ```

      - adres efektywny gdzie trzeba sięgnąć do pamięci to iloczyn EBX i wartości 8 oraz dodany do stałej 4
      - z reguły wartość używana w iloczynie (tu: 8) to 2^N gdzie N to mała liczba 1...6, liczba N mieści się w paru bitach OPCODE
    - Tryb baza-indeks-skala-przemieszczenie

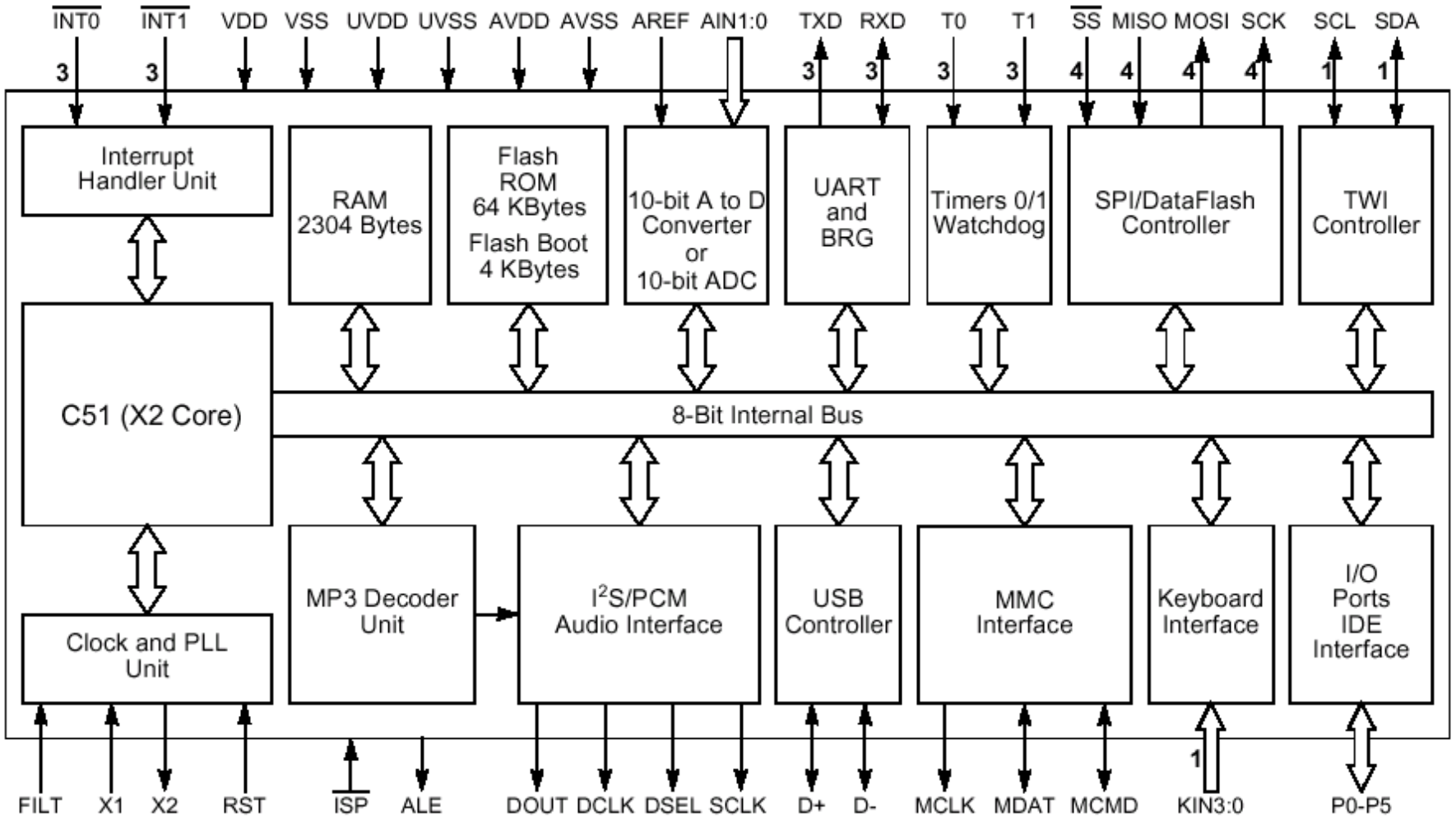
      ```
      ADD EAX, [EBP + 8*EBX + 4]
      ```

      - jak powyżej ale z dodatkowo dodaną zawartością EBP

Inne procesory -

- mikrokontrolery

- C51 (AT83C51SND1C) - ogólny widok wnętrza (głównie peryferii)



Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C) - wycinek listy rozkazów

	0	1	2	3	4	5	6	7
0	NOP	JBC bit,rel [3B, 2C]	JB bit, rel [3B, 2C]	JNB bit, rel [3B, 2C]	JC rel [2B, 2C]	JNC rel [2B, 2C]	JZ rel [2B, 2C]	JNZ rel [2B, 2C]
1	AJMP (P0) [2B, 2C]	ACALL (P0) [2B, 2C]	AJMP (P1) [2B, 2C]	ACALL (P1) [2B, 2C]	AJMP (P2) [2B, 2C]	ACALL (P2) [2B, 2C]	AJMP (P3) [2B, 2C]	ACALL (P3) [2B, 2C]
2	LJMP addr16 [3B, 2C]	LCALL addr16 [3B, 2C]	RET [2C]	RETI [2C]	ORL dir, A [2B]	ANL dir, A [2B]	XRL dir, a [2B]	ORL C, bit [2B, 2C]
3	RR A	RRC A	RL A	RLC A	ORL dir, #data [3B, 2C]	ANL dir, #data [3B, 2C]	XRL dir, #data [3B, 2C]	JMP @A + DPTR [2C]
4	INC A	DEC A	ADD A, #data [2B]	ADDC A, #data [2B]	ORL A, #data [2B]	ANL A, #data [2B]	XRL A, #data [2B]	MOV A, #data [2B]
5	INC dir [2B]	DEC dir [2B]	ADD A, dir [2B]	ADDC A, dir [2B]	ORL A, dir [2B]	ANL A, dir [2B]	XRL A, dir [2B]	MOV dir, #data [3B, 2C]
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	MOV @R0, @data [2B]
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, @R1	MOV @R1, #data [2B]

Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C) - wycinek opisu jednej instrukcji

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

ADDC A,R0

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	1	r	r	r

Operation: ADDC
(A) ← (A) + (C) + (R_n)

ADDC A,direct

Bytes: 2

Cycles: 1

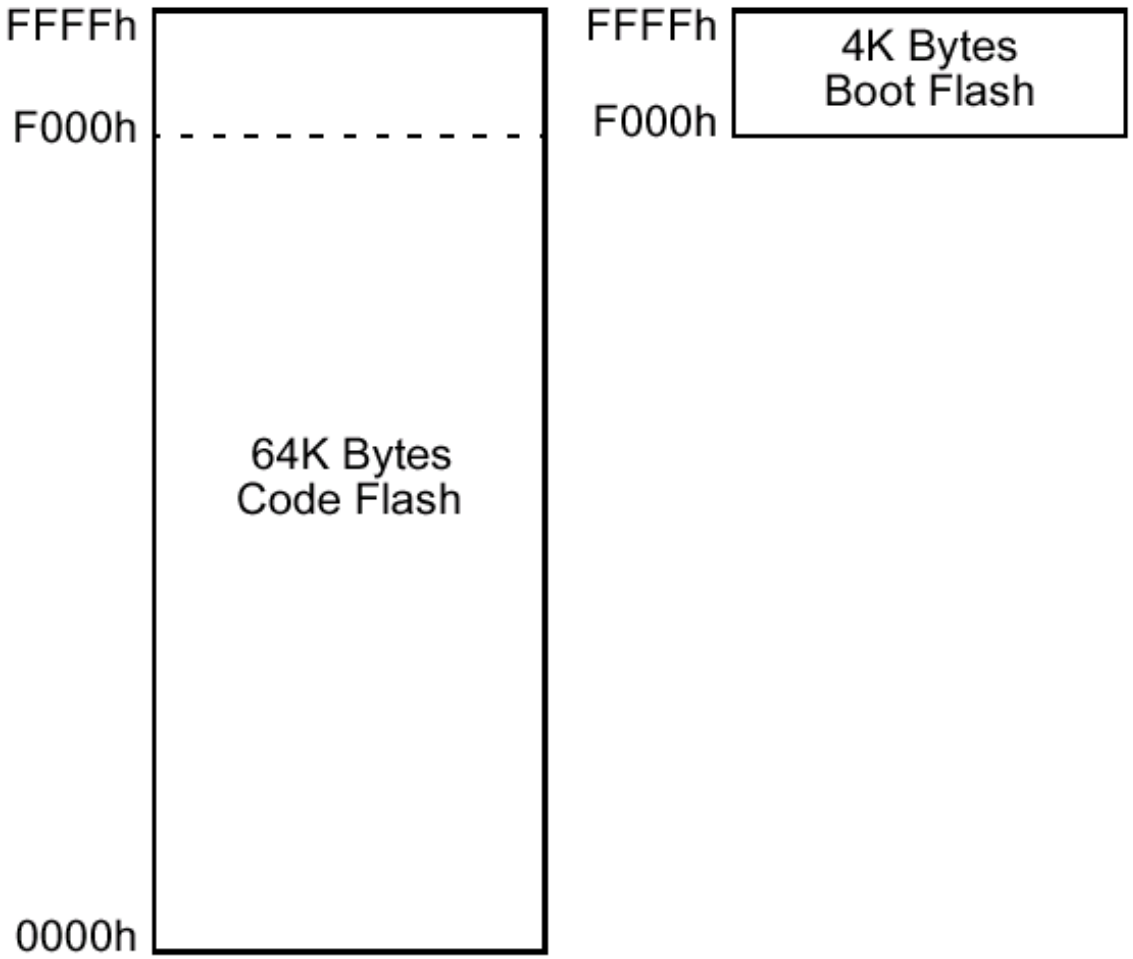
Encoding:

0	0	1	1	0	1	0	1	direct address

Operation: ADDC
(A) ← (A) + (C) + (direct)

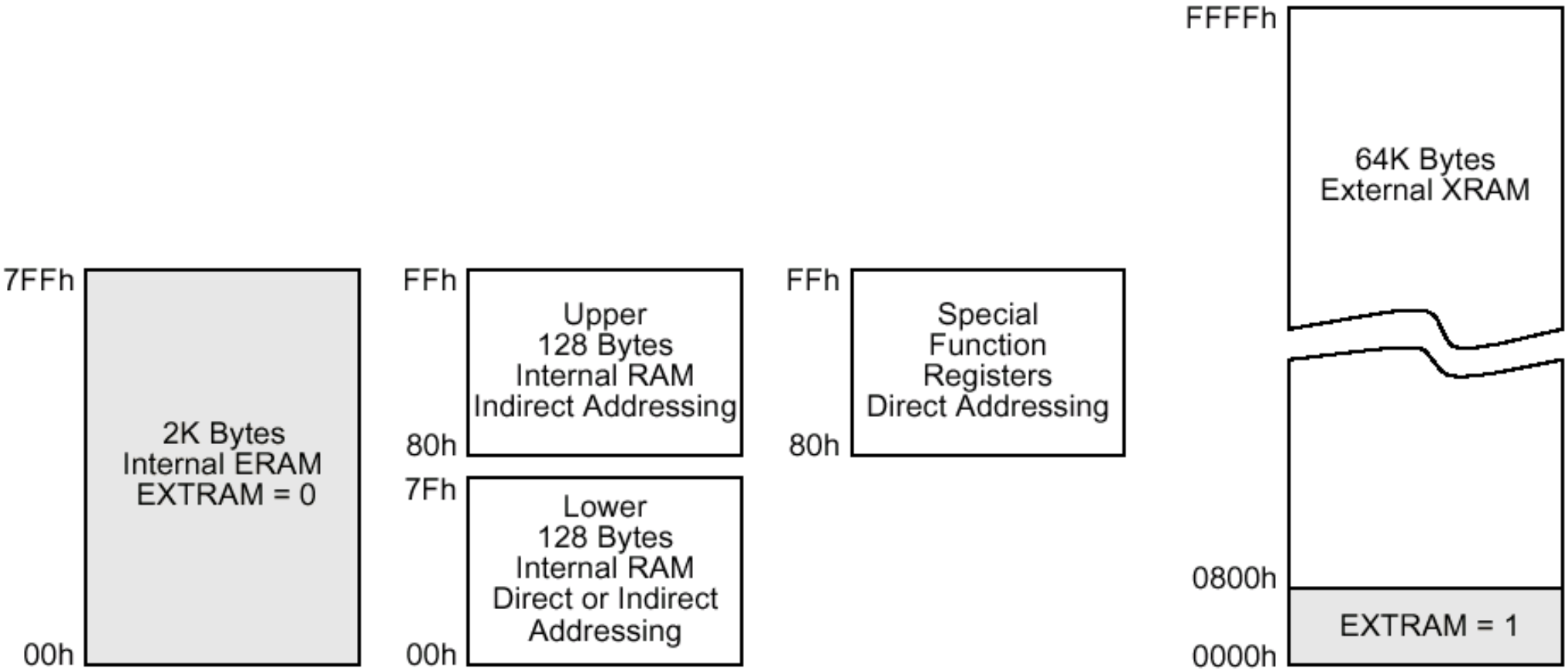
Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C) - przestrzeń pamięci kodu



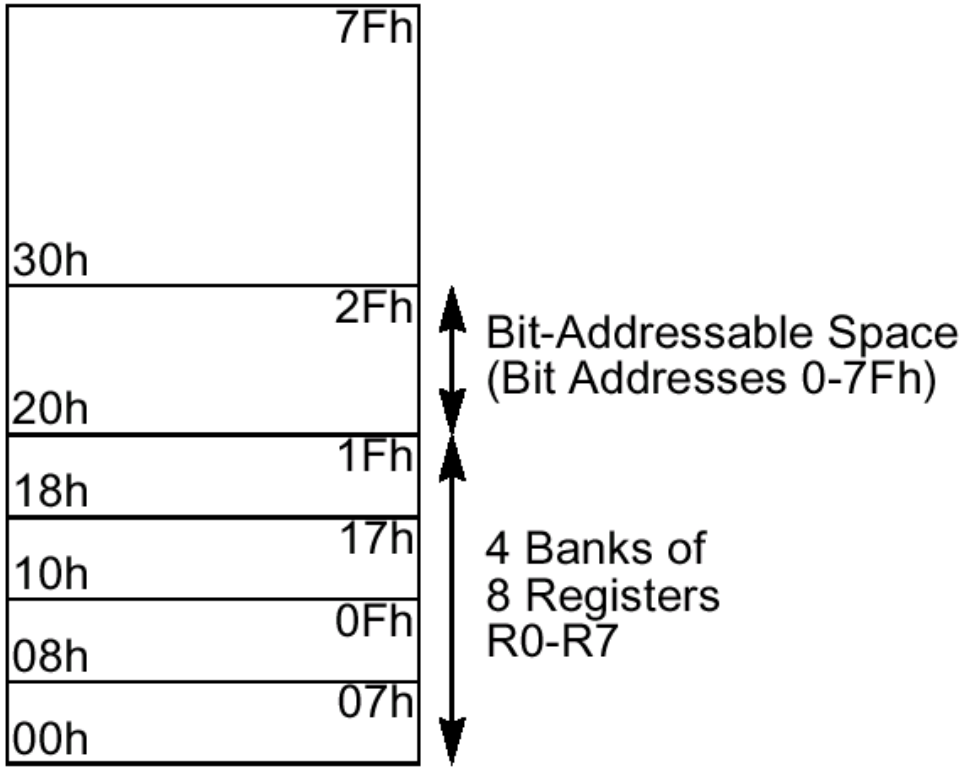
Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C) - przestrzenie pamięci danych



Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C): DATA - pamięć danych (adresy 0x00...0x7f)
  - Pamięć zwana jako „Internal RAM” może działać z
    - adresowaniem bezpośrednim („Direct Addressing”)
    - adresowaniem pośrednim („Indirect Addressing”)



Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C): SFR (Special Function Register)
  - Obszar rejestrów specjalnego przeznaczenia (adresy 0x80...0xFF)
  - Przykładowa mapa pamięci jednej z odmian C51

0F8H								0FFH
0F0H	B 00000000							0F7H
0E8H								0EFH
0E0H	ACC 00000000							0E7H
0D8H								0DFH
0D0H	PSW 00000000							0D7H
0C8H	T2CON 00000000	T2MOD XXXXXX00	RCAP2L 00000000	RCAP2H 00000000	TL2 00000000	TH2 00000000		0CFH
0C0H								0C7H
0B8H	IP XX000000							0BFH
0B0H	P3 11111111							0B7H
0A8H	IE 0X000000							0AFH
0A0H	P2 11111111		AUXR1 XXXXXXX0				WDTRST XXXXXXXX	0A7H
98H	SCON 00000000	SBUF XXXXXXXX						9FH
90H	P1 11111111							97H
88H	TCON 00000000	TMOD 00000000	TL0 00000000	TL1 00000000	TH0 00000000	TH1 00000000	AUXR XXX00XX0	8FH
80H	P0 11111111	SP 00000111	DP0L 00000000	DP0H 00000000	DP1L 00000000	DP1H 00000000	PCON 0XXX0000	87H

Źródło: Materiały informacyjne firmy Atmel

- C51 (AT83C51SND1C): SFR - konsekwencje mapowania rejestrów
  - Akumulator

    `MOV A, #5`
    - działa identycznie jak

    `MOV ACC, #5`
    - gdzie „A” to nazwa rejestru, gdy „ACC” to rejestr „SFR” pod adresem 0xE0
  - Wskaźnik stosu
    - SP - mapowany pod adresem 0x81
  - Rejestr flag
    - PSW - mapowany pod adresem 0xD0
  - Wskaźnik DPTR
    - dzielony na dwie części: DPL i DPH - mapowane pod adresami 0x82 i 0x83
  - Licznik rozkazów
    - IP (interrupt priority) to nie PC!!!, nie jest mapowany w SFR

- C51 (AT83C51SND1C): SFR - obszar rejestrów specjalnego rejestr PSW (flagi)
  - PSW - mapowany pod adresem 0xD0
    - PSW.0 (P) - parzystość
    - PSW.1 (F1) - w 80C52 flaga kontrolowana przez aplikacje użytkownika
    - PSW.2 (Overflow) - przepełnienie
    - PSW.3 (RS0) - młodszy bit aktualnie używanego numeru banku rejestrów (Register Bank Switch)
    - PSW.4 (RS1) - starszy bit aktualnie używanego numeru banku rejestrów (Register Bank Switch)
    - PSW.5 (F0) - flaga ogólnego zastosowania
    - PSW.6 (AC) - przeniesienie z młodszych 4 bitów do starszego 4 bitów (Auxiliary Carry)
    - PSW.7 (CY) - przeniesienie z najstarszego bitu

- C51 (AT83C51SND1C): IDATA - pamięć dostępna pośrednio
  - Dostęp bezpośredni

```
MOV direct, #5
```

    - dla 0<„direct”<0x7f trafi do „DATA”
    - dla „direct”>=0x80 trafi do „SFR” - co oznacza „Direct addressing”
      - dla direct=0xE0, będzie to równoważne z MOV ACC, #5
  - Dostęp pośredni

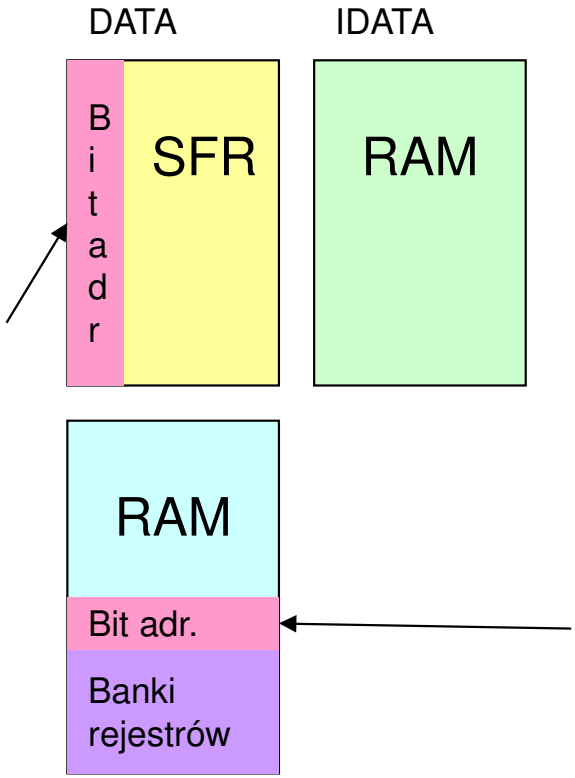
```
MOV @R0, #5
```

    - gdyby jednak R0 było równe 0xE0, wartość 5 nie trafi do ACC, ale do pamięci IDATA („Internal RAM Indirect Addressing”)
  - Stos (jako przykład adresowania pośredniego) może być umieszczony w pamięci IDATA
    - decyzje gdzie - podejmuje programista z reguły pierwszych instrukcjach kodu aplikacji

```
MOV SP,#0xF0
```

- C51 (AT83C51SND1C) - adresowanie bitowe
  - specyficzny sposób odwołań do pamięci
    - do określonych lokacji można odwoływać się tradycyjnie i bitowo

Tutaj bitowo adresowalne są wyłącznie rejestry o adresach podzielnych bez reszty przez 8, np.:
PSW ma adres „bajtowy” 0xD0, ale jego bit PSW.7 ma adres „bitowy” 0xD7

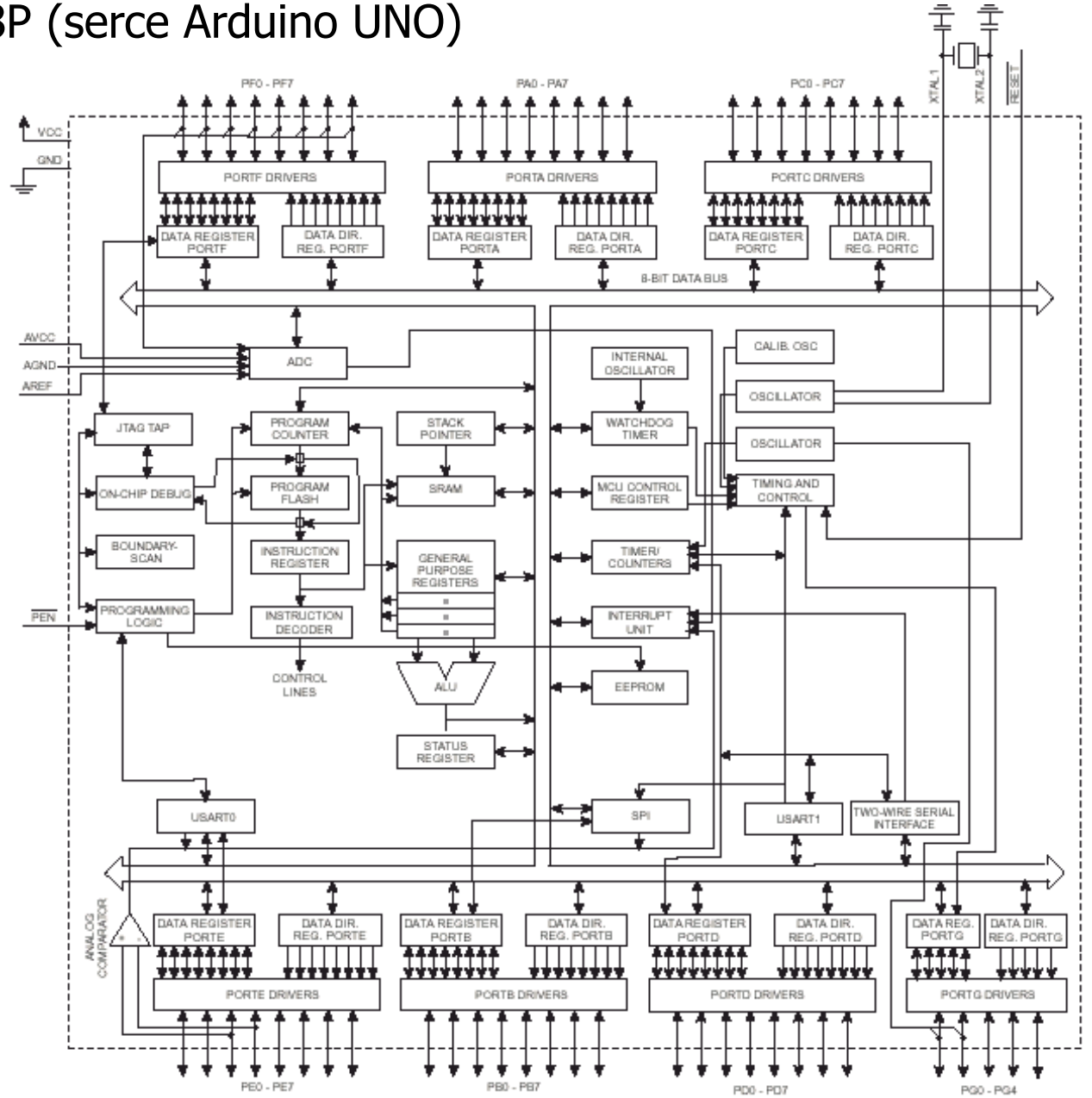


- Obszar o specjalnym adresowaniu bitowym:
  - Pierwszy bit adresowalny bitowo - leży w „bajcie” DATA pod adresem „bajtowym” 0x20 ale ma adres „bitowy” 0x00
  - Ostatni bit adresowalny bitowo leży w „bajcie” DATA pod adresem „bajtowym” 0x2F i ma adres „bitowy” 0x7F
  - Równoważne zapisy (tylko niektóre instrukcje tak działają)

    CLR 0x2F.6

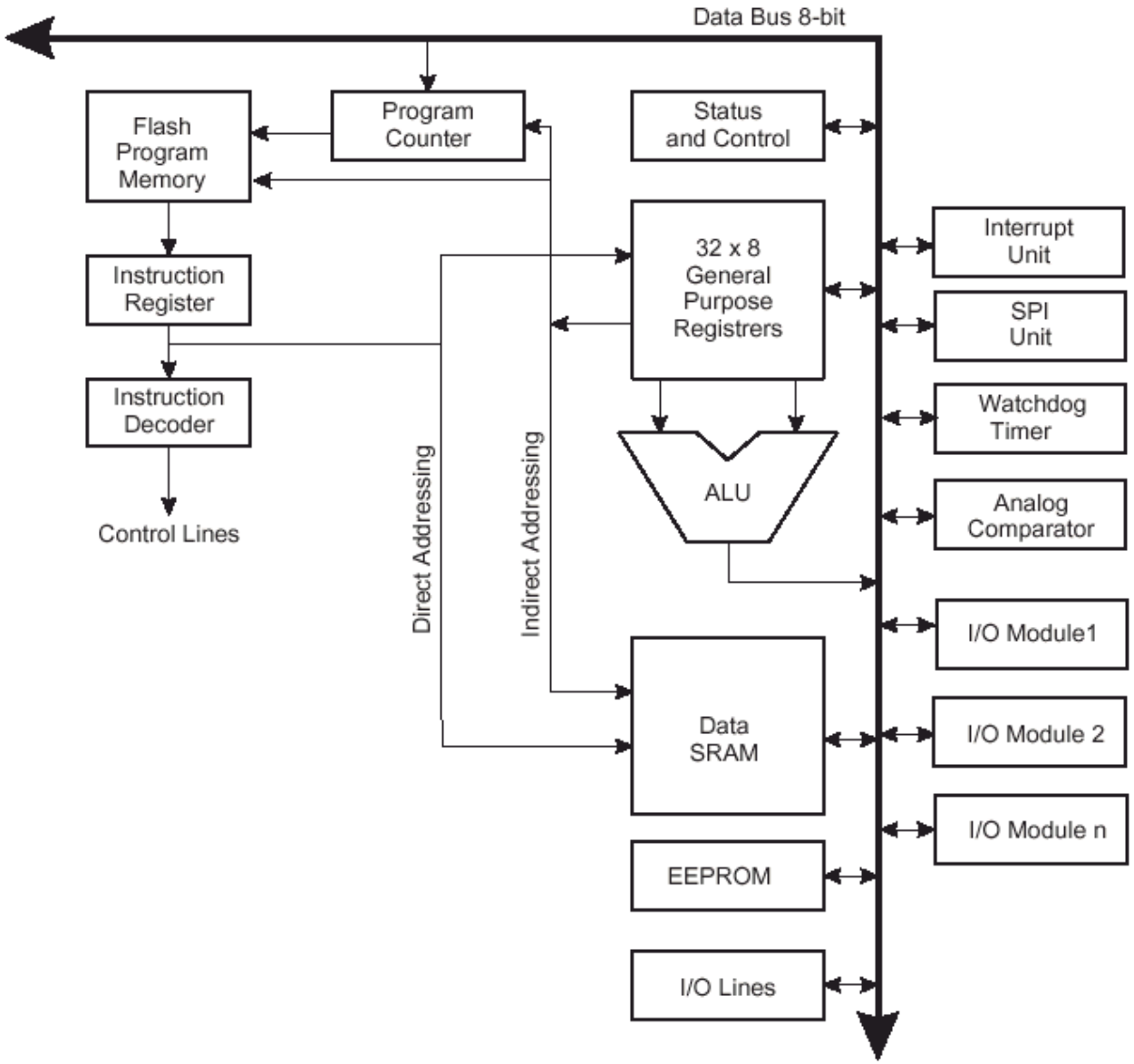
    CLR 0x7E

- AVR (ATmega128) - ogólny widok wnętrza
  - bogatszy brat ATMega328P (serce Arduino UNO)



Źródło: Materiały informacyjne firmy Atmel

- AVR (ATmega128) - uproszczony widok wnętrza



Źródło: Materiały informacyjne firmy Atmel

- AVR (ATmega128) - wycinek listy rozkazów

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
Arithmetic and Logic Instructions					
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,S,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,S,H	1
ADIW	Rd, K	Add Immediate to Word	Rd+1:Rd ← Rd+1:Rd + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,S,H	1
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,S,H	1
SBC	Rd, Rr	Subtract with Carry	Rd ← Rd - Rr - C	Z,C,N,V,S,H	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd ← Rd - K - C	Z,C,N,V,S,H	1
SBIW	Rd, K	Subtract Immediate from Word	Rd+1:Rd ← Rd+1:Rd - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND	Rd ← Rd • Rr	Z,N,V,S	1
ANDI	Rd, K	Logical AND with Immediate	Rd ← Rd • K	Z,N,V,S	1
OR	Rd, Rr	Logical OR	Rd ← Rd v Rr	Z,N,V,S	1
ORI	Rd, K	Logical OR with Immediate	Rd ← Rd v K	Z,N,V,S	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr	Z,N,V,S	1
COM	Rd	One's Complement	Rd ← $FF - Rd	Z,C,N,V,S	1
NEG	Rd	Two's Complement	Rd ← $00 - Rd	Z,C,N,V,S,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V,S	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FFh - K)	Z,N,V,S	1

Źródło: Materiały informacyjne firmy Atmel

- AVR (ATmega128) - wycinek opisu jednej instrukcji

ADC - Add with Carry

Description:

Adds two registers and the contents of the C flag and places the result in the destination register Rd.

Operation:

(i) Rd ← Rd + Rr + C

	Syntax:	Operands:	Program Counter:
(i)	ADC Rd,Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1

16-bit Opcode:

0001	11rd	dddd	rrrr

Status Register (SREG) Boolean Formulae:

I	T	H	S	V	N	Z	C
-	-	⇔	⇔	⇔	⇔	⇔	⇔

H: $Rd3 \bullet Rr3 + Rr3 \bullet \overline{R3} + \overline{R3} \bullet Rd3$
Set if there was a carry from bit 3; cleared otherwise

S: $N \oplus V$, For signed tests.

V: $Rd7 \bullet Rr7 \bullet \overline{R7} + \overline{Rd7} \bullet \overline{Rr7} \bullet R7$
Set if two's complement overflow resulted from the operation; cleared otherwise.

N: R7
Set if MSB of the result is set; cleared otherwise.

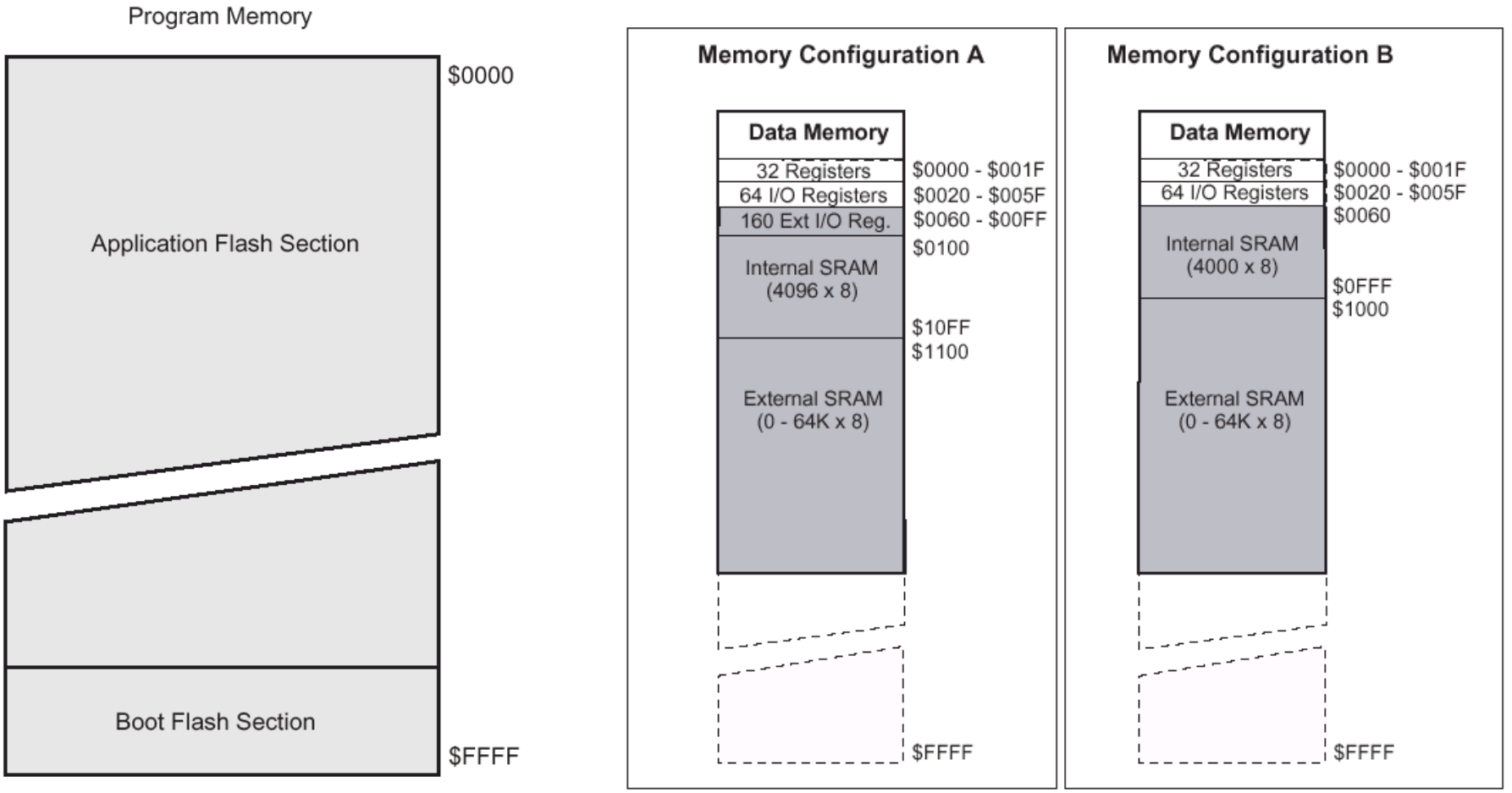
Z: $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if the result is $00; cleared otherwise.

C: $Rd7 \bullet Rr7 + Rr7 \bullet \overline{R7} + \overline{R7} \bullet Rd7$
Set if there was carry from the MSB of the result; cleared otherwise.

R (Result) equals Rd after the operation.

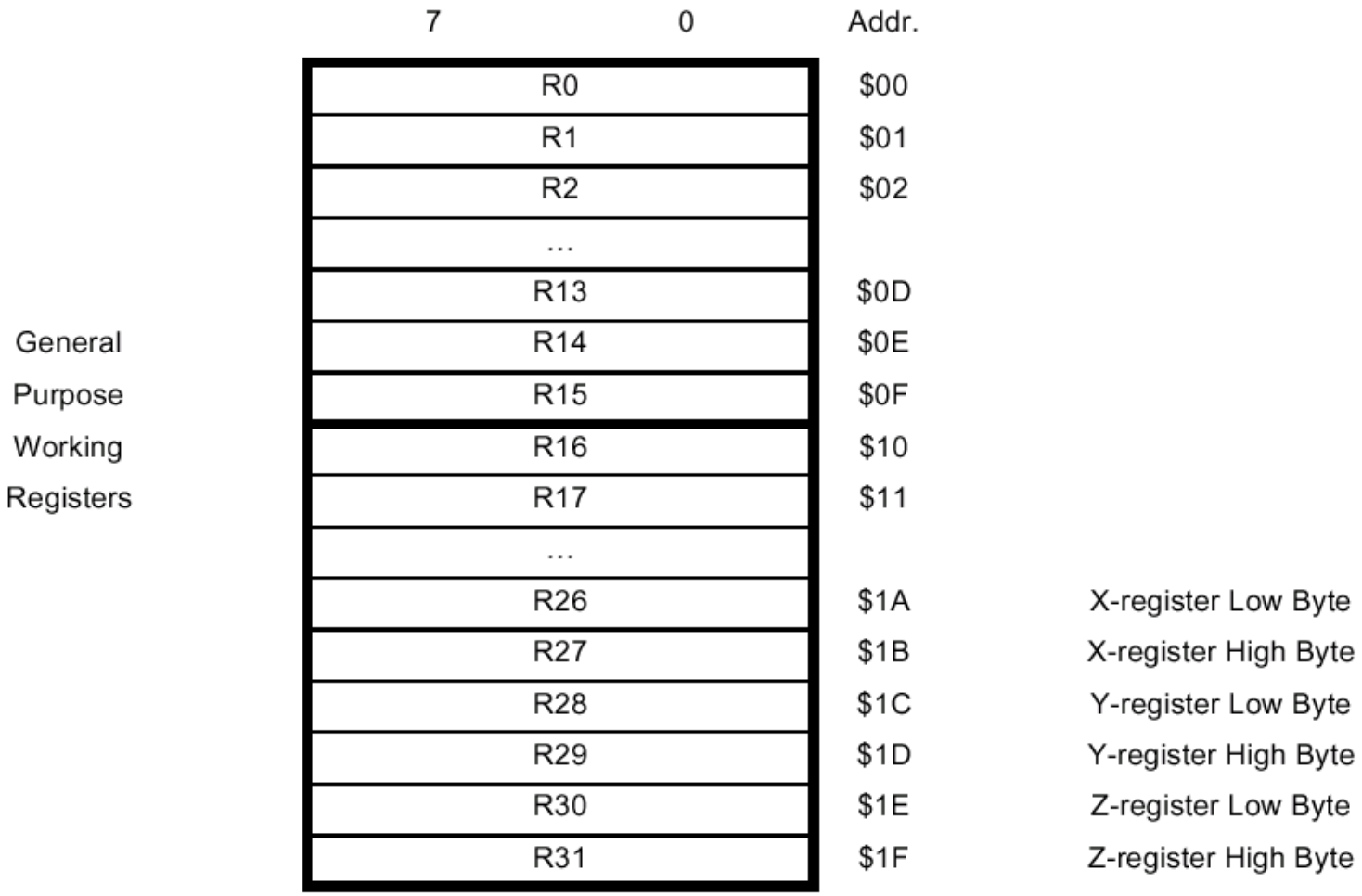
Źródło: Materiały informacyjne firmy Atmel

- AVR (ATmega128) - przestrzenie pamięci



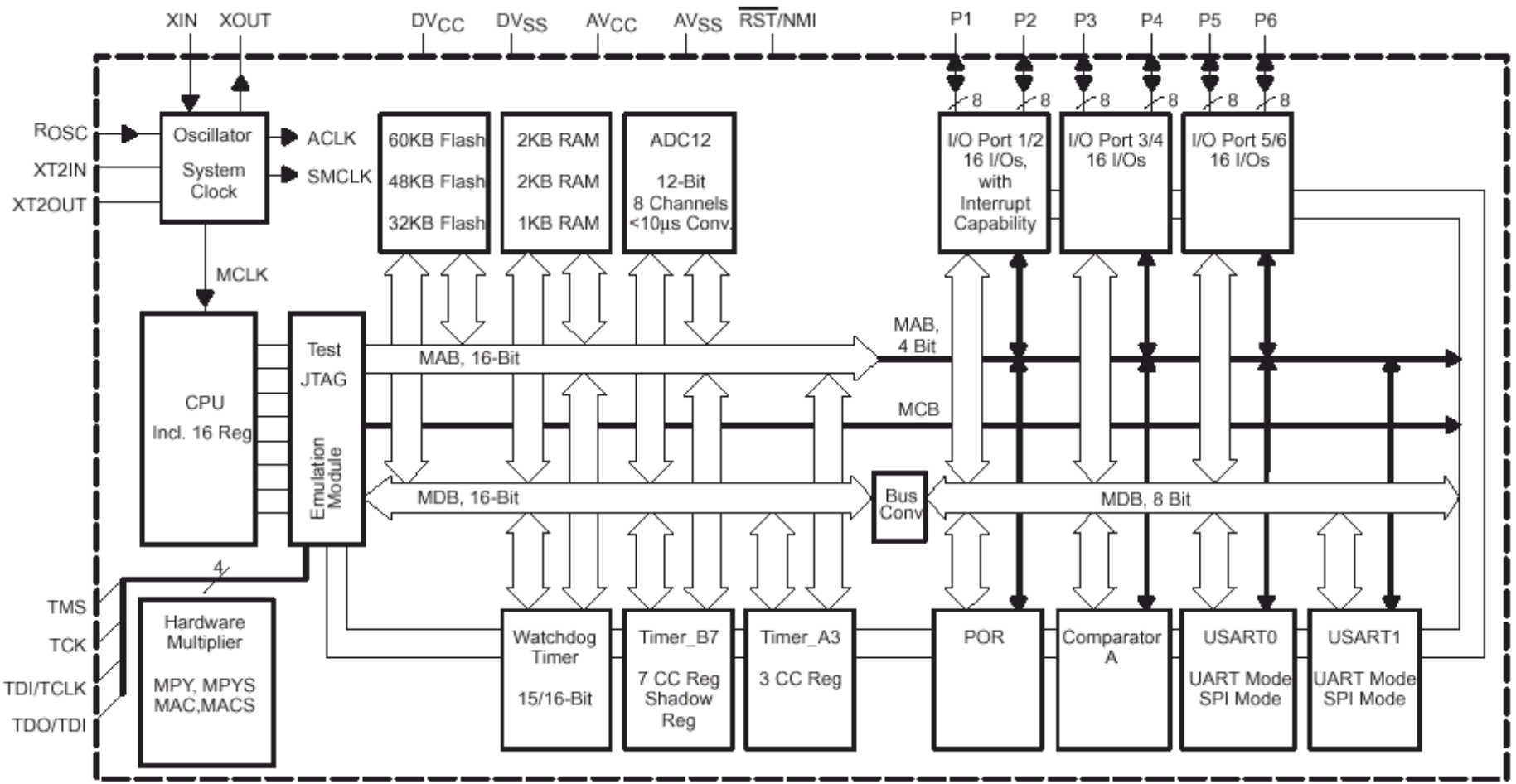
Źródło: Materiały informacyjne firmy Atmel

- AVR (ATmega128) - rejestry



Źródło: Materiały informacyjne firmy Atmel

- MSP430 (MSP430F1xx) - ogólny widok wnętrza



Źródło: Materiały informacyjne firmy Texas Instruments

- MSP430 (MSP430F1xx) - lista rozkazów

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Źródło: Materiały informacyjne firmy Texas Instruments

- MSP430 (MSP430F1xx) - wycinek opisu jednej instrukcji

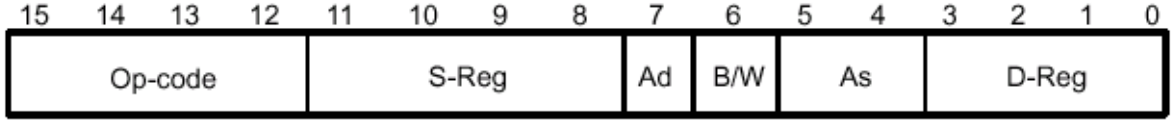


Table 3−11 lists and describes the double operand instructions.

Table 3−11. Double Operand Instructions

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	–	–	–	–
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst − src	*	*	*	*
DADD(.B)	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src,dst	src .and. dst	0	*	*	*
BIC(.B)	src,dst	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	src .or. dst → dst	–	–	–	–
XOR(.B)	src,dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	src .and. dst → dst	0	*	*	*

Źródło: Materiały informacyjne firmy Texas Instruments

- MSP430 (MSP430F1xx) - rejestry

Źródło: Materiały informacyjne firmy Texas Instruments

Dziękujemy za uwagę!