Architektura komputerów

Wykład 12

Teoretyczny model komputera – DLW

Wojciech Kordecki

Collegium Witelona
Wydział Nauk Technicznych i Ekonomicznych
Zakład Informatyki

Semestr letni 2023/24

Maszyna von Neumanna – maszyna Turinga

Maszyna von Neumanna, opracowana w 1945 roku.
W pamięci komputera zarówno dane, jak i program.
Przetwarzanie w arytmometrze.
Jest kontynuacją maszyny Turinga:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$$

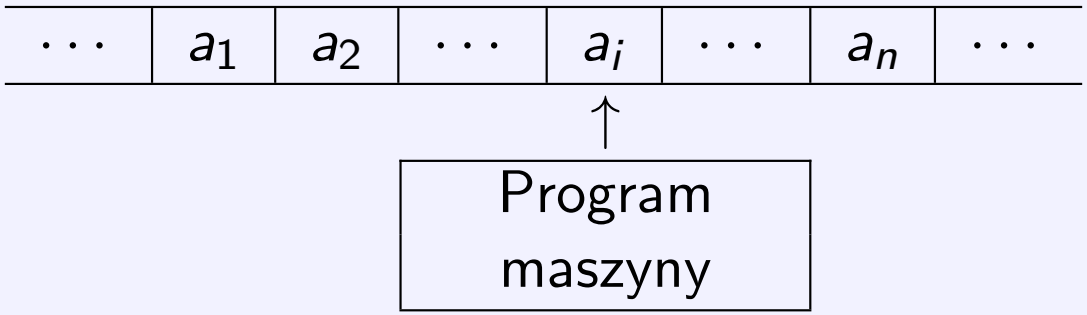
gdzie:

- Q – skończony zbiór stanów,
- $q_0 \in Q$ – stan początkowy,
- F – zbiór stanów końcowych,
- Γ – skończony zbiór dopuszczalnych symboli taśmowych,
- B – symbol pusty należący do Σ,
- Σ – podzbiór Γ nie zawierający B, zwany zbiorem symboli wejściowych,
- δ – funkcja następnego ruchu, odwzorowanie:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, -\}.$$

Schemat maszyny Turinga (1936)

- Taśma prawostronnie nieskończona z zaznaczonymi kwadratami.
- Ruchoma głowica odczytująco-zapisująca.



John von Neumann

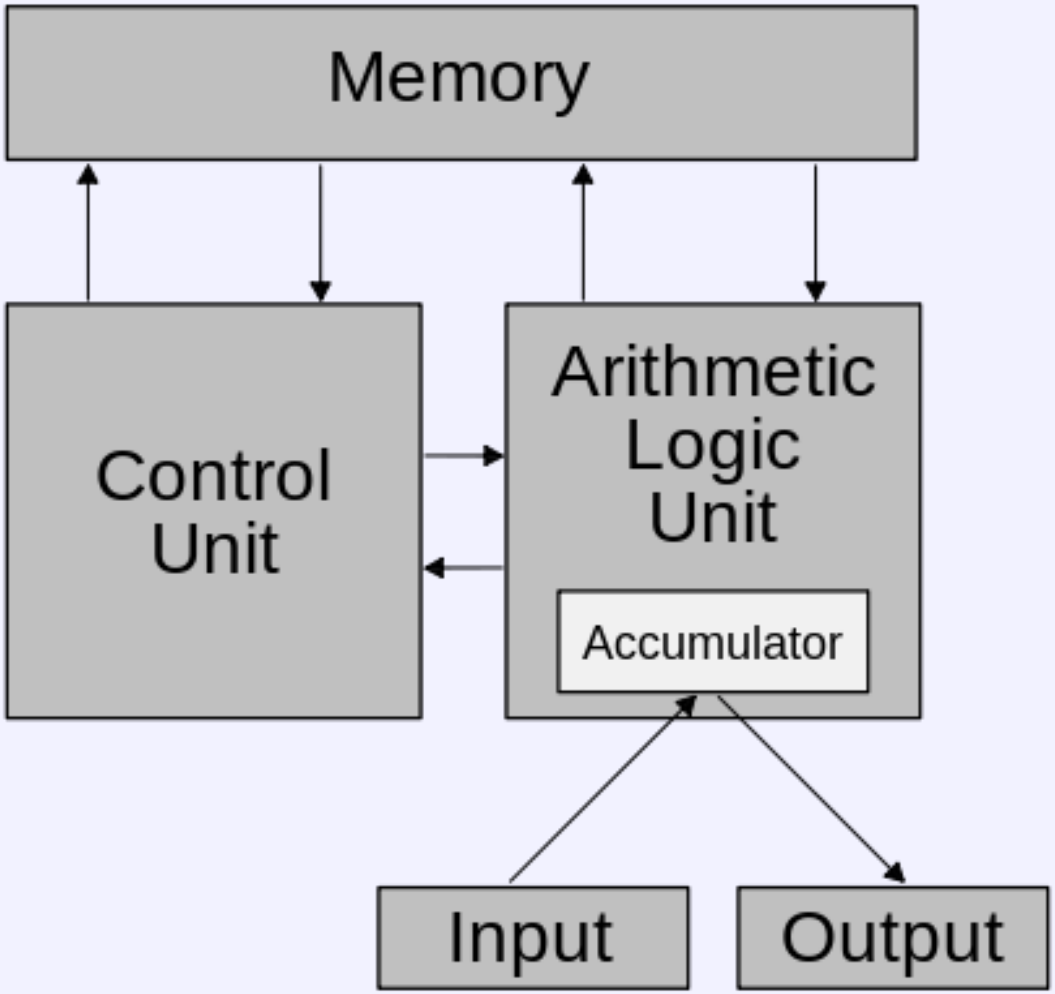
John von Neumann
(1903–1957)
– węgierski matematyk,
inżynier chemik, fizyk
i informatyk,
pracujący głównie w Stanach Zjednoczonych.

Architektura von Neumanna

Architektura opracowana przez Johna von Neumanna, Johna W. Mauchly'ego oraz Johna Prespera Eckerta w 1945 roku. Dane przechowywane są wspólnie z instrukcjami.
Komponenty.

- Pamięć komputerowa przechowująca dane programu oraz instrukcje programu; każda komórka pamięci ma adres.
- Jednostka sterująca odpowiedzialna za pobieranie danych i instrukcji z pamięci oraz ich sekwencyjne przetwarzanie.
- Jednostka arytmetyczno-logiczna odpowiedzialna za wykonywanie podstawowych operacji arytmetycznych.
- Urządzenia wejścia/wyjścia służące do interakcji z operatorem.

Architektura von Neumanna – schemat



Źródło – Wikipedia

System komputerowy

Control Unit + Arithmetic Logic Unit = Processor.

W architekturze von Neumanna system ma:

- skończony i pełny zbiór rozkazów,
- program wprowadzony do pamięci komputera tak jak danych,
- instrukcje i dane jednakowo dostępne dla procesora.
- przetwarzanie informacji przez sekwencyjne wykonywanie instrukcji.

Pamięć programu i danych jest wspólna.

Ograniczenia architektury von Neumanna

- Brak podziału między pamięcią programu i pamięcią danych powoduje możliwość popełnienia błędu polegającego na „wykonaniu" danych zamiast instrukcji.
- Przesyłanie dużej liczby danych między procesorem a pamięcią powoduje blokowanie dostępu do pamięci, a przesyłanie danych jest operacją wolną w stosunku do przetwarzania danych przez procesor.

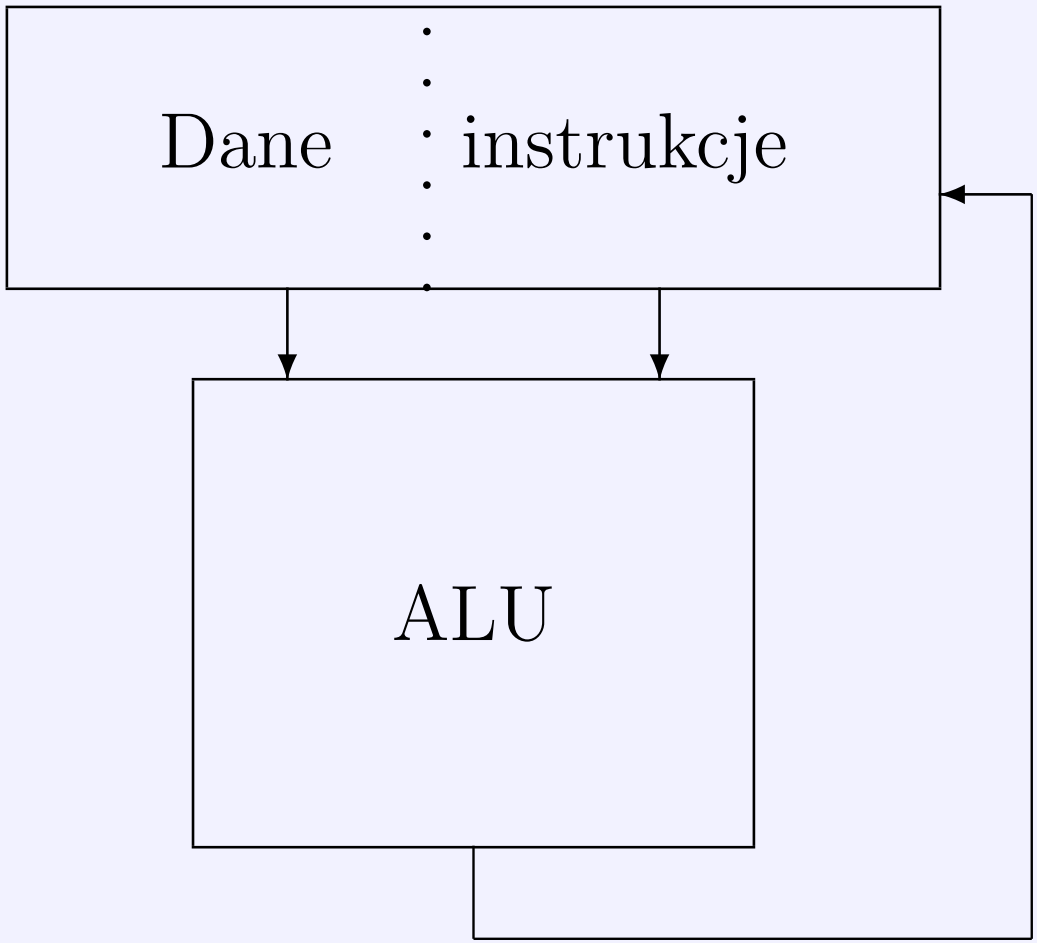
Przykład. Dodawanie dwóch liczb:

1. pobranie instrukcji dodawania,
2. pobranie pierwszej liczby,
3. pobranie drugiej liczby,
4. dodanie (w arytmometrze),
5. zapisanie wyniku w pamięci.

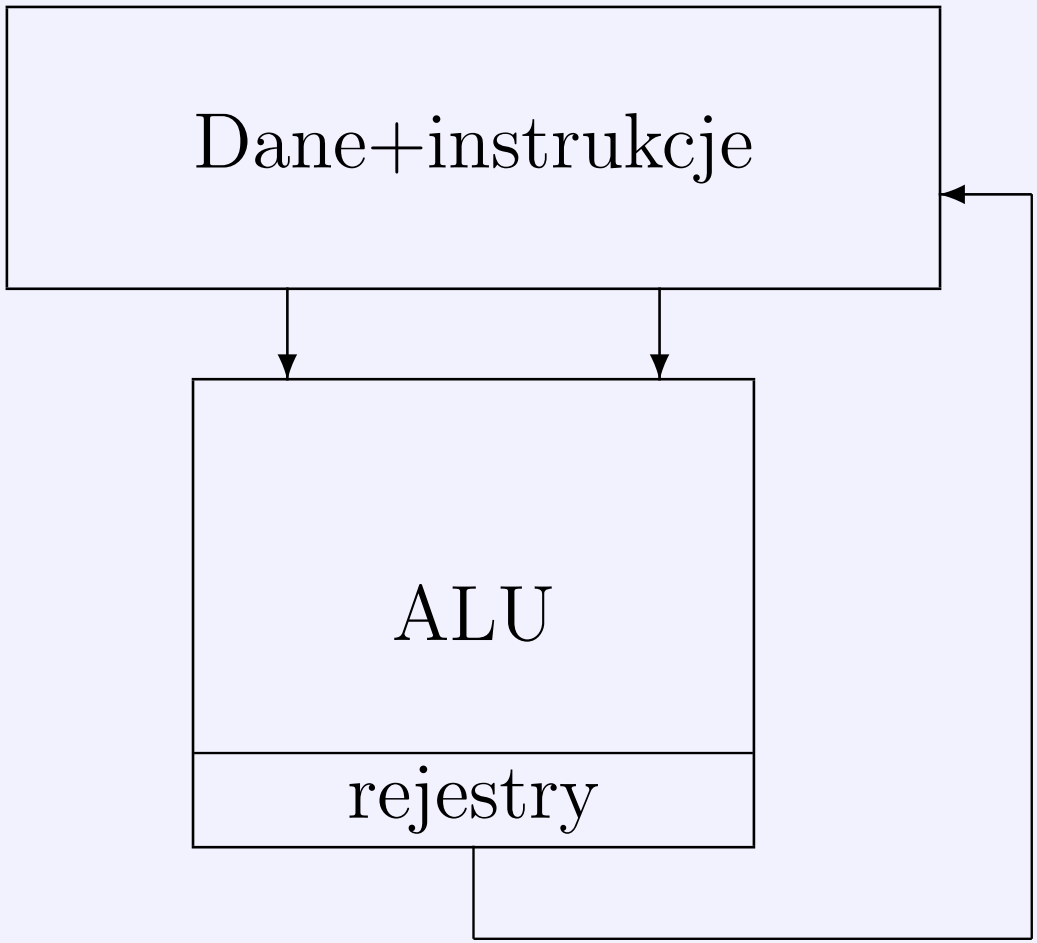
Przeciwdziałanie ograniczeniom

- Brak podziału między pamięcią programu i pamięcią danych: mechanizmy ochrony pamięci – przy próbie pobrania instrukcji z bloku danych – odmowa wykonania.
- Przesyłanie dużej liczby danych między procesorem a pamięcią: stosowanie pamięci podręcznej (cache).

Architektura von Neumanna – schemat uproszczony



Architektura von Neumanna – rejestry



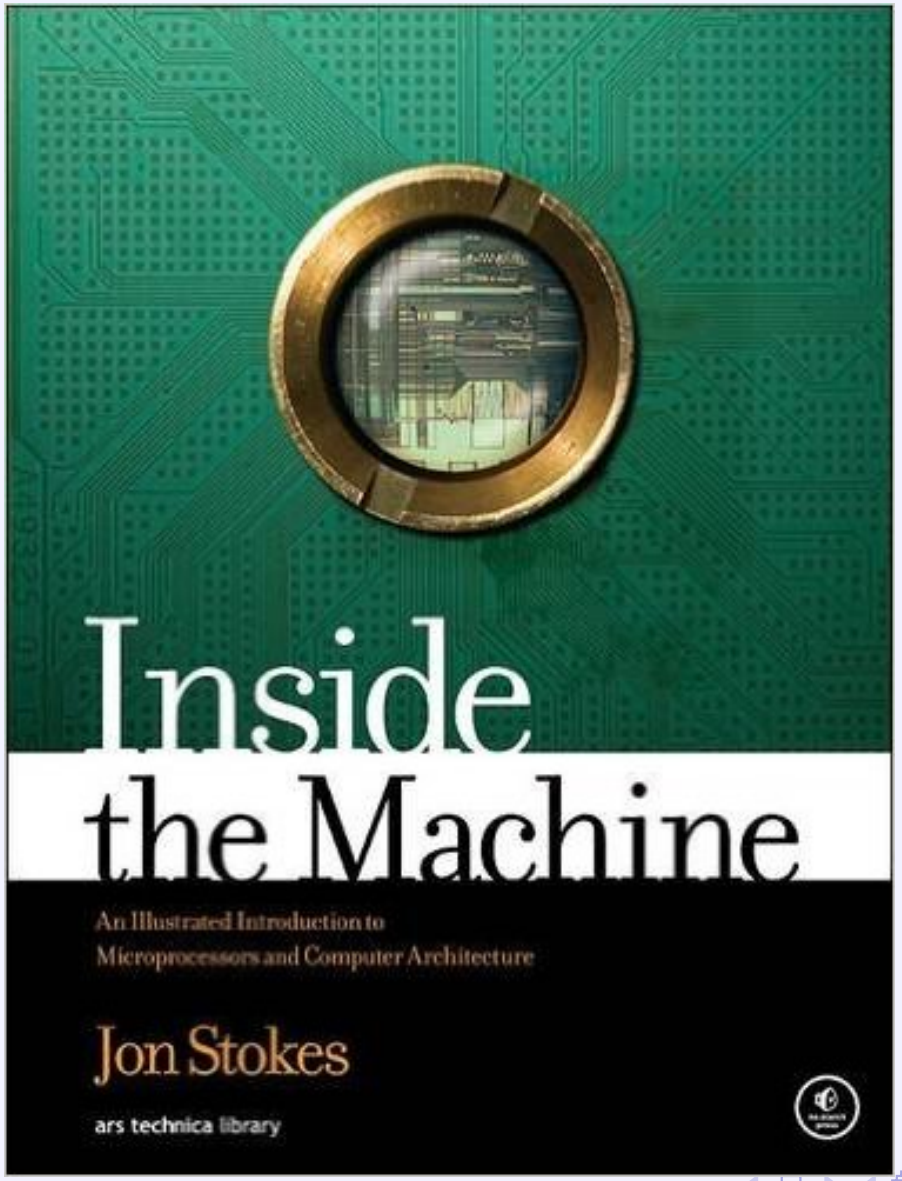
INSIDE THE MACHINE

Jon Stokes, INSIDE THE MACHINE, No Starch Press 2007.

https://www.nostarch.com/insidemachine.htm

Jon M. Stokes is co-founder of and Senior CPU Editor for Ars Technica. He has written extensively on microprocessor architecture and the technical aspects of personal computing for a variety of publications. Stokes holds a degree in computer engineering from Louisiana State University and two advanced degrees in the humanities from Harvard University. He is currently pursuing a Ph.D. at the University of Chicago

Inside the Machine



Opinia: Dave Walz-Burkett

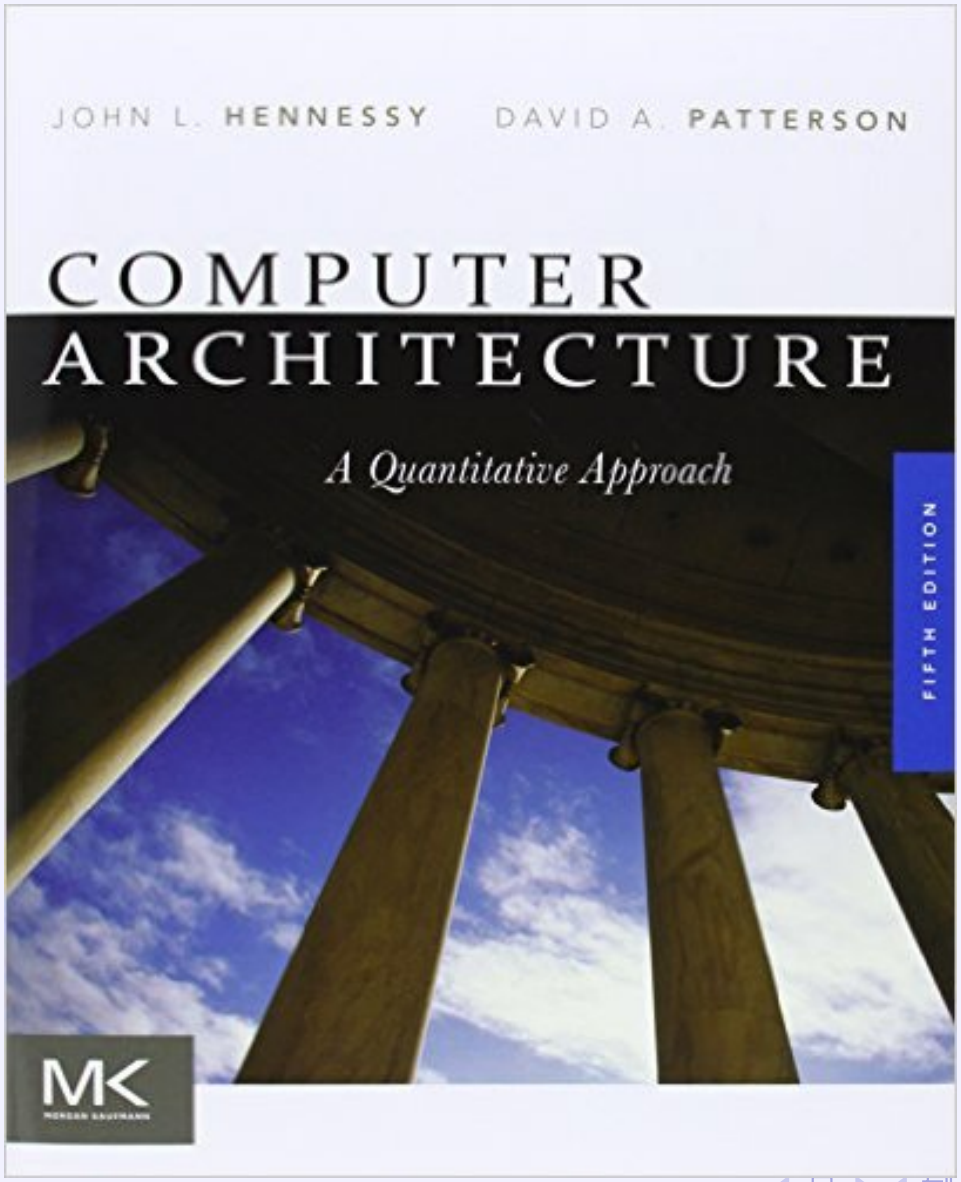
Topics covered include 'basic computing concepts', 'mechanics of program execution', 'pipelined execution', 'superscalar execution', '64-bit computing and X86-64' and 'caching and performance'. Processors covered include Intel (Pentium, Pentium Pro, Pentium 4, Pentium M, Core Duo, and Core 2 Duo) and Motorola (PowerPC 601, 603, 604, 604e, 750 aka G3, 7400 aka G4, G4E and 970 aka G5).

Komputer DLW

"DLW" in honor of the DLX architecture used by Hennessy and Patterson in their books on computer architecture.

John L. Hennessy, David A. Patterson. *Computer Architecture. A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier 2012.

Computer Architecture



Instrukcje

Cztery rejestry A,B,C,D.
256 komórek pamięci #0 . . . #255.

Format instrukcji arytmetycznej:

```
instruction source1, source2, destination
```

Przykład:

Kod	*Komentarz*
`add A, B, C`	Dodaj zawartość A do zawartości B, wynik do C, zamazując poprzednią zawartość

Dygresja: różne asemblery (1)

Zwykle w asemblerach:
`instruction destination, [source,] source`

Przykład. MCS-51:

```
MOV A, #1    ; A=1
MOV 10h, A   ; adres(10h)=1
ADD A,10h    ; A=2
```

AVR

```
ldi R16,1    ; R16=1
mov R1,R16   ; R1=1
add R1,R16   ; R1=2
```

Dygresja: różne asemblery (2)

ARM

```
MOV r0, #0x11   ; załadowanie wartosci poczatkowej
MOV r1, r0, LSL #1 ; przesuniecie o jeden bit w lewo
MOV r2, r1, LSL #1 ; przesuniecie o jeden bit w lewo
```

DLW

```
instruction source, [source,] destination
```

Dostęp do pamięci

Komórki pamięci #11 – #14: 12, 6, 2, 3.
Format instrukcji dostępu do pamięci:

```
instruction source, destination
```

Program 1.

Wiersz	*Kod*	*Komentarz*
1	`load #12, A`	zawartość komórki #12 do A
2	`load #13, B`	zawartość komórki #13 do B
3	`add A, B, C`	Dodaj zawartość A do zawartości B, wynik do C,
4	`store C, #14`	Wynik dodawania z C do komórki #14

Komórki pamięci:

#11	#12	#13	#14
12	6	2	8

Wartości bezpośrednie

Kod	*Komentarz*
`add A, 2, A`	wartość 2 dodana do zawartości A

Adres w rejestrze

`#register` – adres jest zawarty w rejestrze.

Program 2.

Wiersz	*Kod*	*Komentarz*
1	`load #D, A`	zawartość komórki o adresie w D (D = 12) do A
2	`load #13, B`	zawartość komórki #13 do B
3	`add A, B, C`	Dodaj zawartość A do zawartości B, wynik do C,
4	`store C, #14`	Wynik dodawania z C do komórki #14

Programy 1 i 2 – porównanie

Programy 1 i 2 różnią się tylko pierwszymi wierszami, ale wykonują dokładnie to samo.

Program 1: `load #12, A`
Program 2: `load #D, A`

Tak samo dla instrukcji `store`.

Przykład

Program 3.

Wiersz	*Kod*	*Komentarz*
1	`#11, D`	zawartość #11 do D
2	`load #D, A`	zawartość komórki o adresie w D (D = 12) do A
3	`load #13, B`	zawartość komórki #13 do B
4	`add A, B, C`	Dodaj zawartość A do zawartości B, wynik do C,
5	`store C, #14`	Wynik dodawania z C do komórki #14

Segment i offset

Adresowanie względne.
Segment danych, to ciągły blok pamięci.
`base address + offset` gdzie `base address` jest adresem początku bloku, a `offset` odległością od początku bloku.

```
load #(D + 108), A  ; zawartość z adresu #(D + 108) do A
store B, #(D + 108) ; zawartość B pod adres  #(D + 108)
```

Rejestry A, B, C, D – ogólnego użytku.

Język maszynowy komputera DLW-1

Mnemoniki `add`, `load`, `store`, `...` → kody, podobnie rejestry,

Mnemonics → *opcodes*

Mnemonic	Opcode
add	000
sub	001
load	010
store	011

Registers → *Binary Code*

Register	Binary Code
A	00
B	01
C	10
D	11

Dekodowanie instrukcji

Operacje arytmetyczne typu rejestrowego: mode=0

Byte 1

0	1	2	3	4	5	6	7
mode	opcode			source1		source2	

Byte 2

0	1	2	3	4	5	6	7
destination		000000					

`add A, B, C` 00000001 10000000

Dekodowanie instrukcji

Operacje arytmetyczne z adresem bezpośrednim: mode=1

Byte 1

1	1	2	3	4	5	6	7
mode	opcode			source		destination	

Byte 2

0	1	2	3	4	5	6	7
8-bit immediate value							

```
add C, 8, A    10001000 00001000
add 5, A, C    10000010 00000101
sub 25, D, C   10011110 00011001
```

Dostęp do pamięci

`load`

Byte 1

1	1	2	3	4	5	6	7
mode	opcode			00		destination	

Byte 2

0	1	2	3	4	5	6	7
8-bit immediate value							

`load #12, A` 10100000 00001100

Dostęp do pamięci

`store`

Byte 1

1	1	2	3	4	5	6	7
mode	opcode			source		00	

Byte 2

0	1	2	3	4	5	6	7
8-bit immediate value							

`load #12, A` 10100000 00001100

Przykład

wiersz	asembler	język maszynowy
1	`load #12, A`	10100000 00001100
2	`load #13, B`	10100001 00001101
3	`add A, B, C`	00000001 10000000
4	`store C, #14`	10111000 00001110

Wykonanie programu

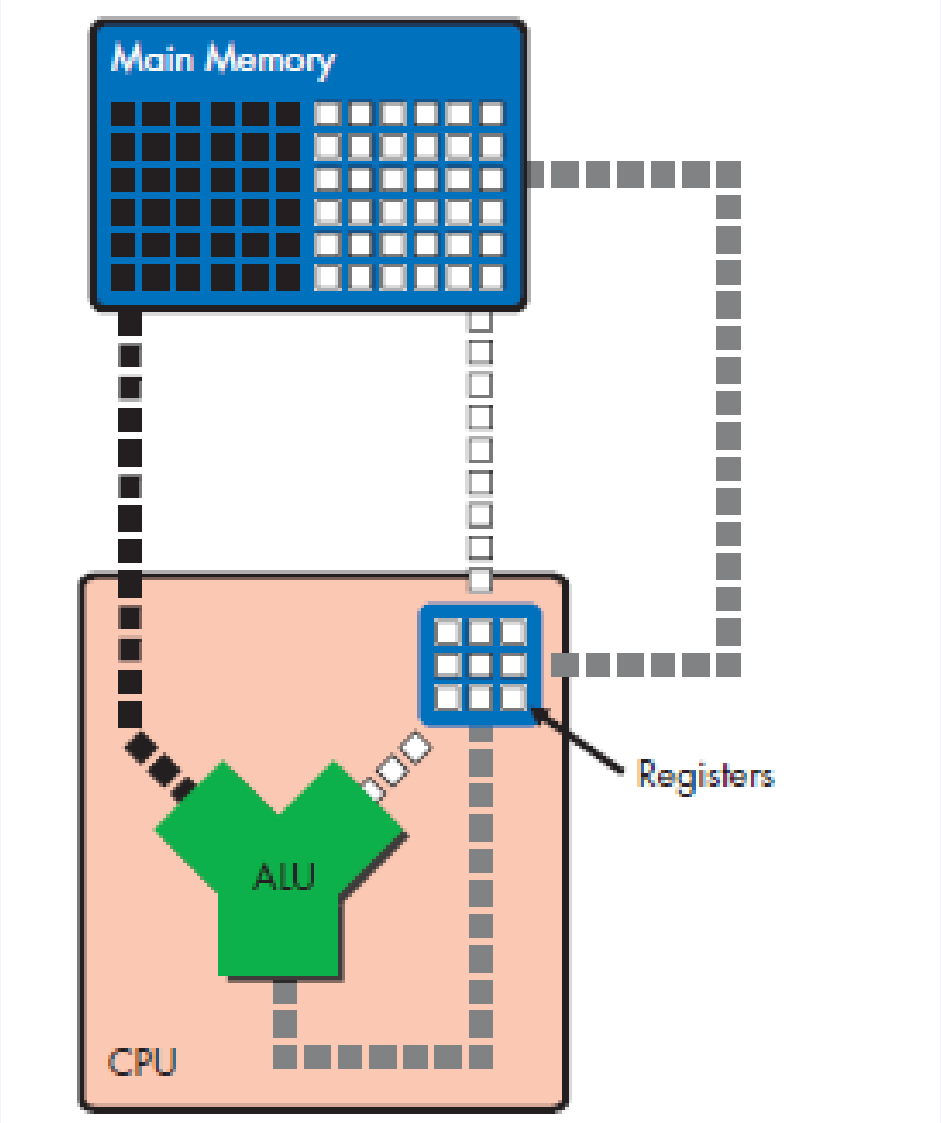
Program 1.

Program umieszczony w pamięci począwszy od adresu #500

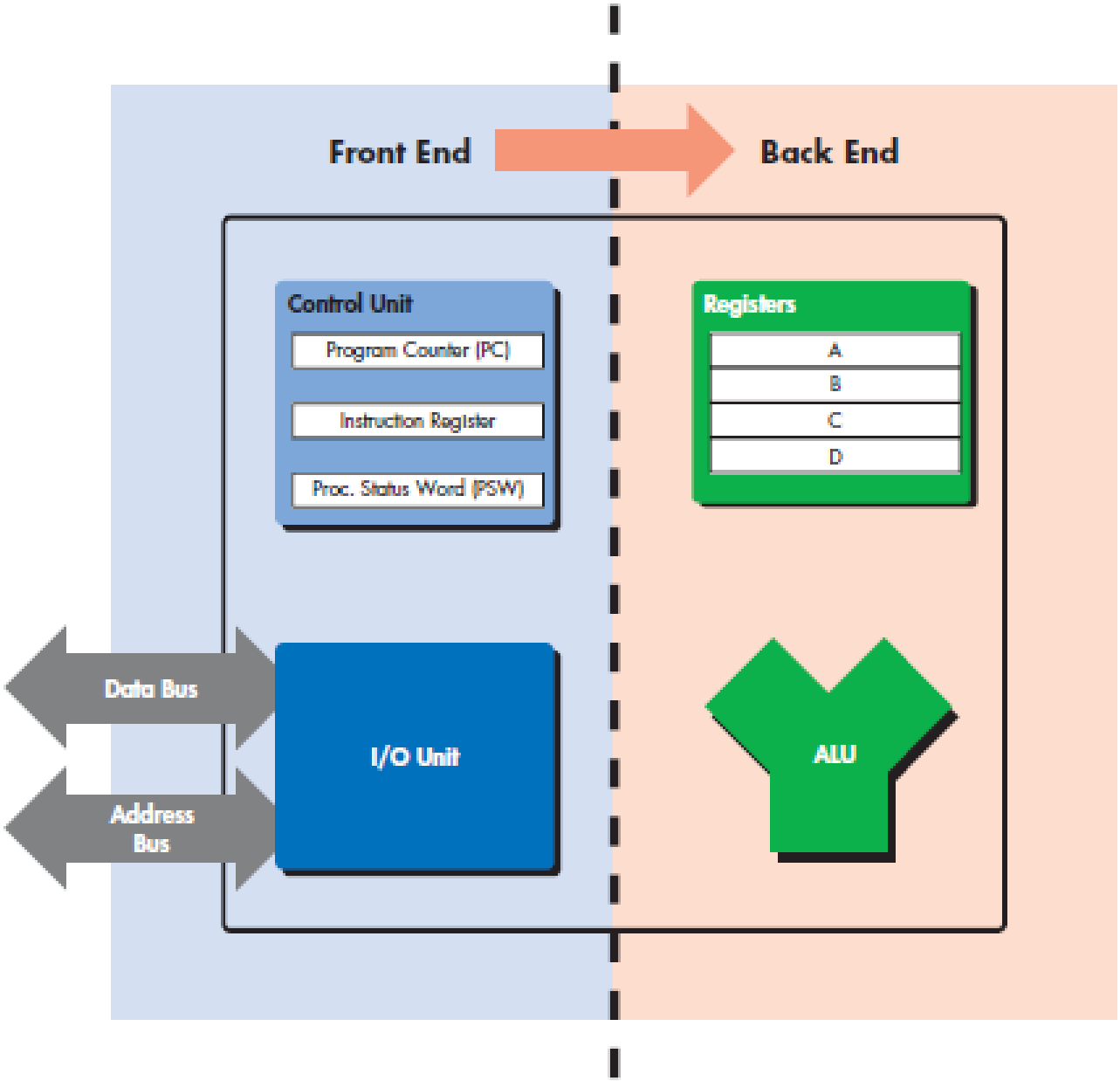
```
load #12, A
load #13, B
add A, B, C
store C, #14
```

#500	#501	#502	#503	#504	#505	#506	#507
load #12,A		load #13, B		add A, B, C		store C, #14	

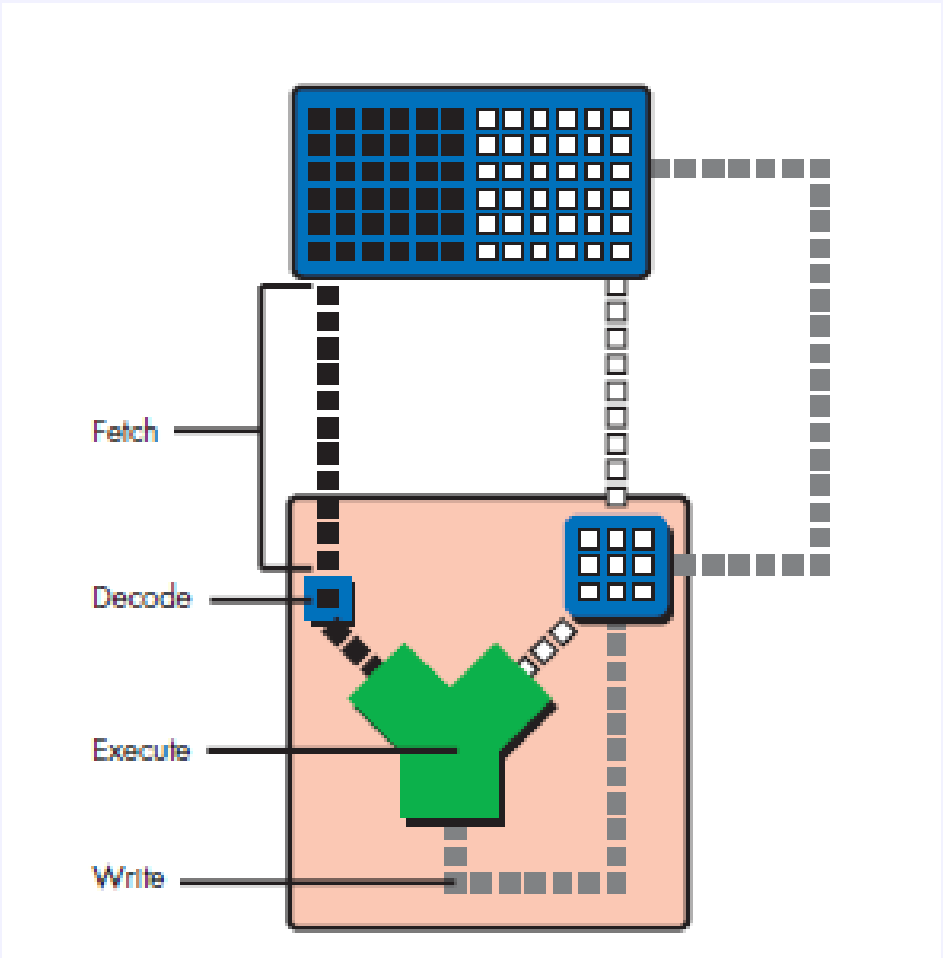
DLW-1



Cztery fazy


Front End
Back End
Control Unit
Program Counter (PC)
Instruction Register
Proc. Status Word (PSW)
Registers
A
B
C
D
Data Bus
Address Bus
I/O Unit
ALU

Cztery fazy c.d.



Skoki

Bezwarunkowy: `jump #target`

Warunkowy:

```
sub A, B, C
jumpz #106
add A, B, C
```

Skoki c.d.

Warunkowy:

```
sub A, B, C
jumpz #C
add A, 15, A
```

`jump #(C  30)+` – skok o 15 instrukcji

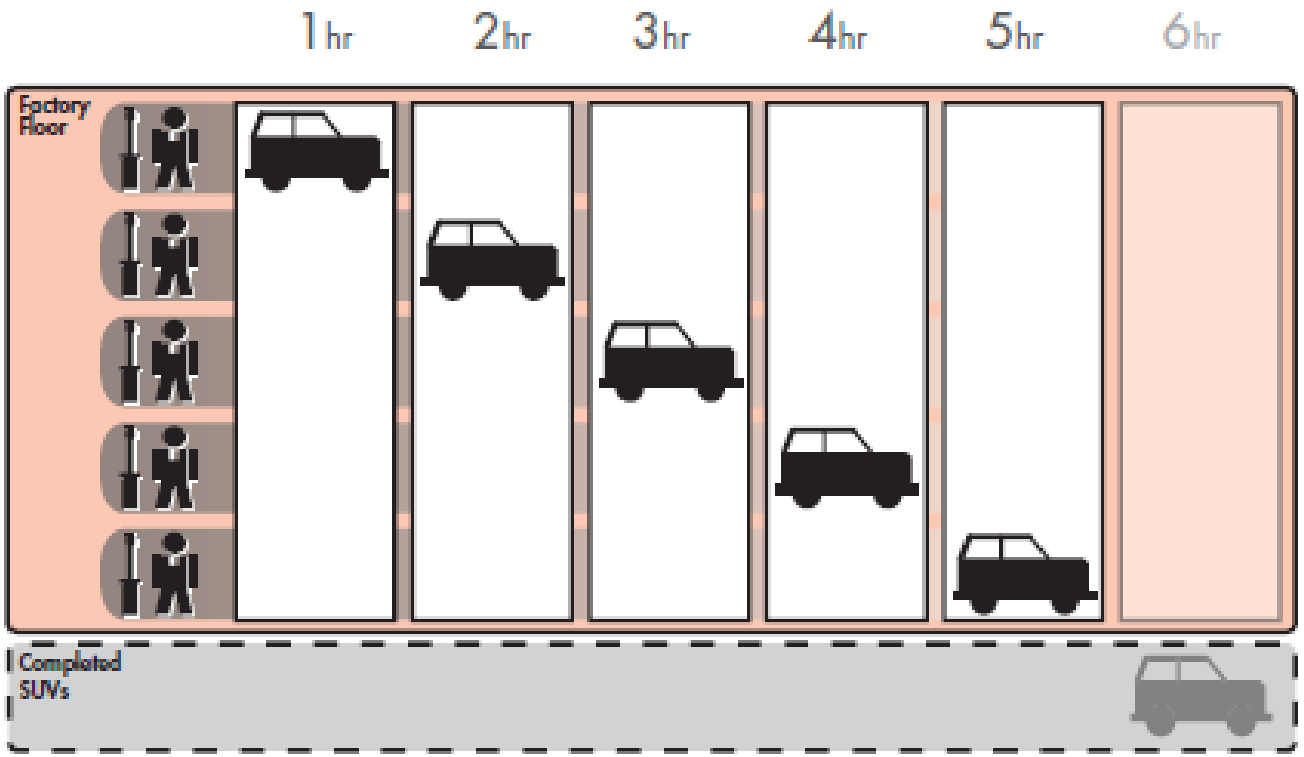
Skoki z użyciem etykiet

```
        sub A, B, A
        jumpz LBL1
        add A, 15, A
        store A, #(D + 16)
LBL1:   add A, B, B
        store B, #(D + 16)
```

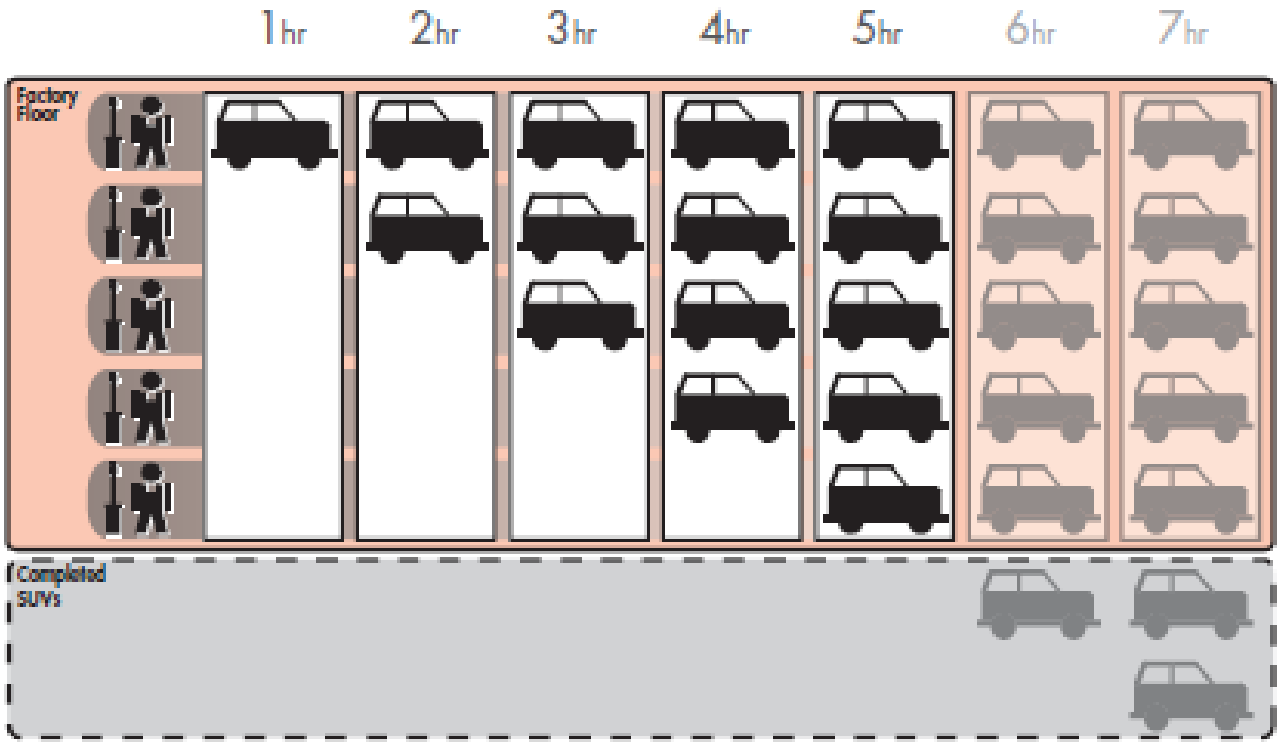
Potok

- Fetch
- Decode
- Execute

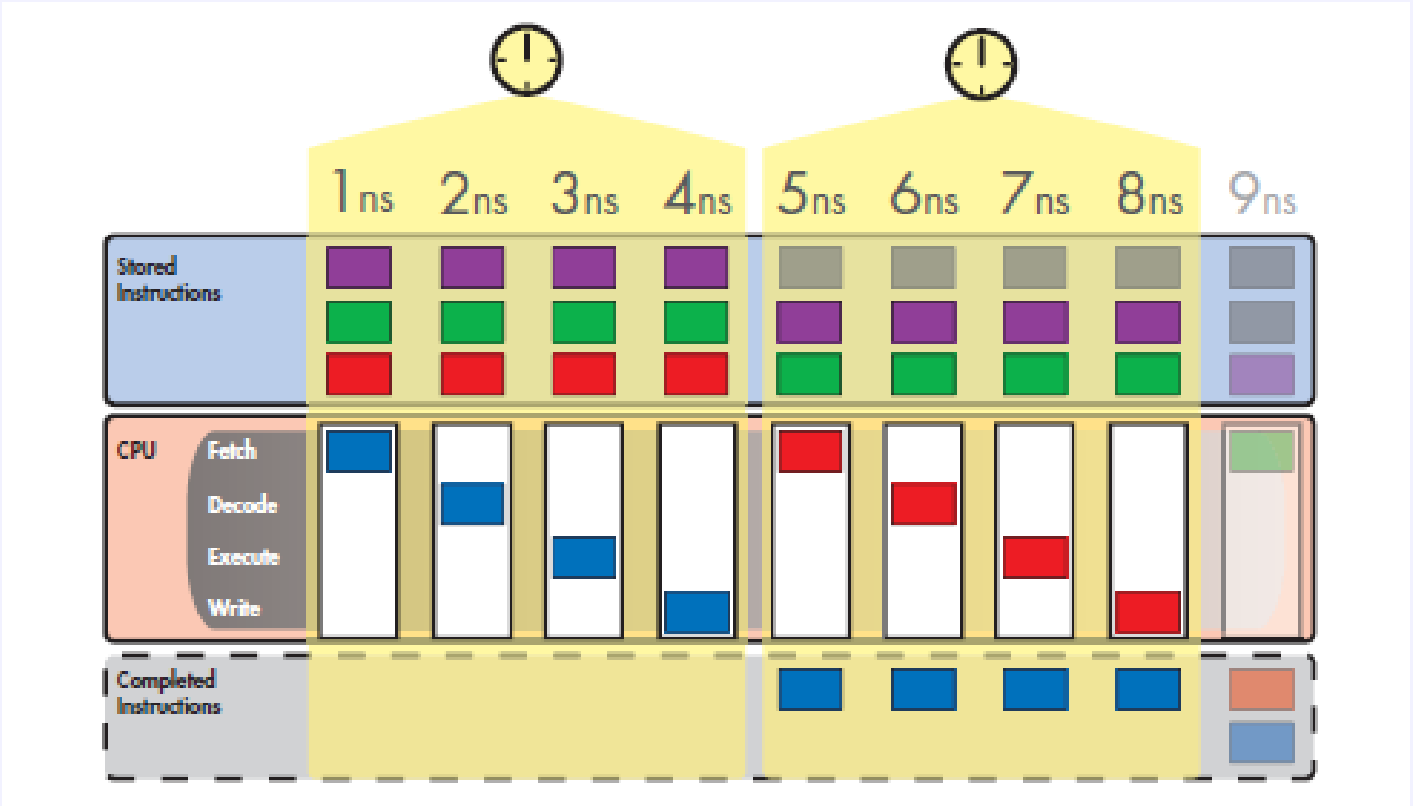
Przetwarzanie bezpotokowe



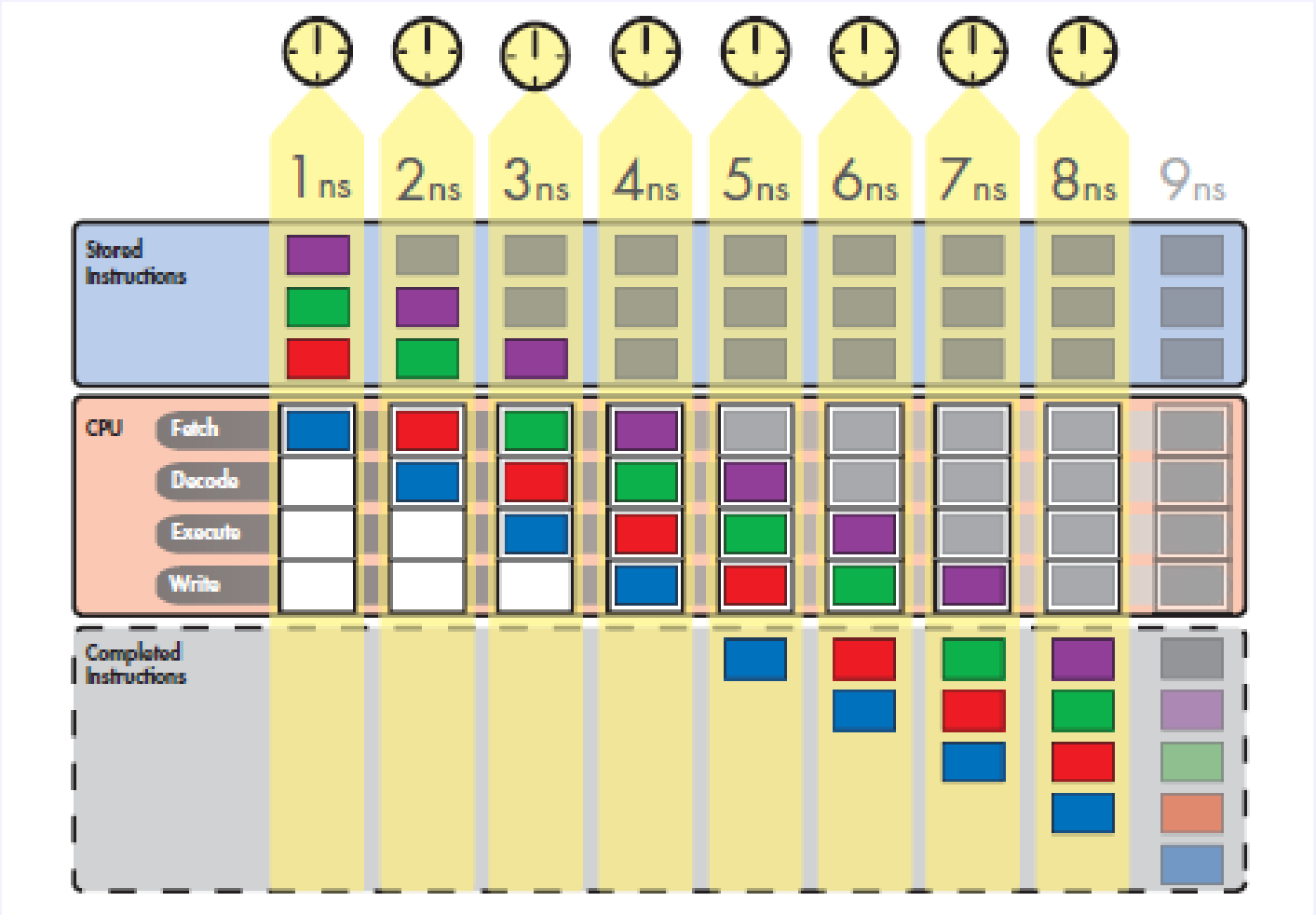
Przetwarzanie potokowe



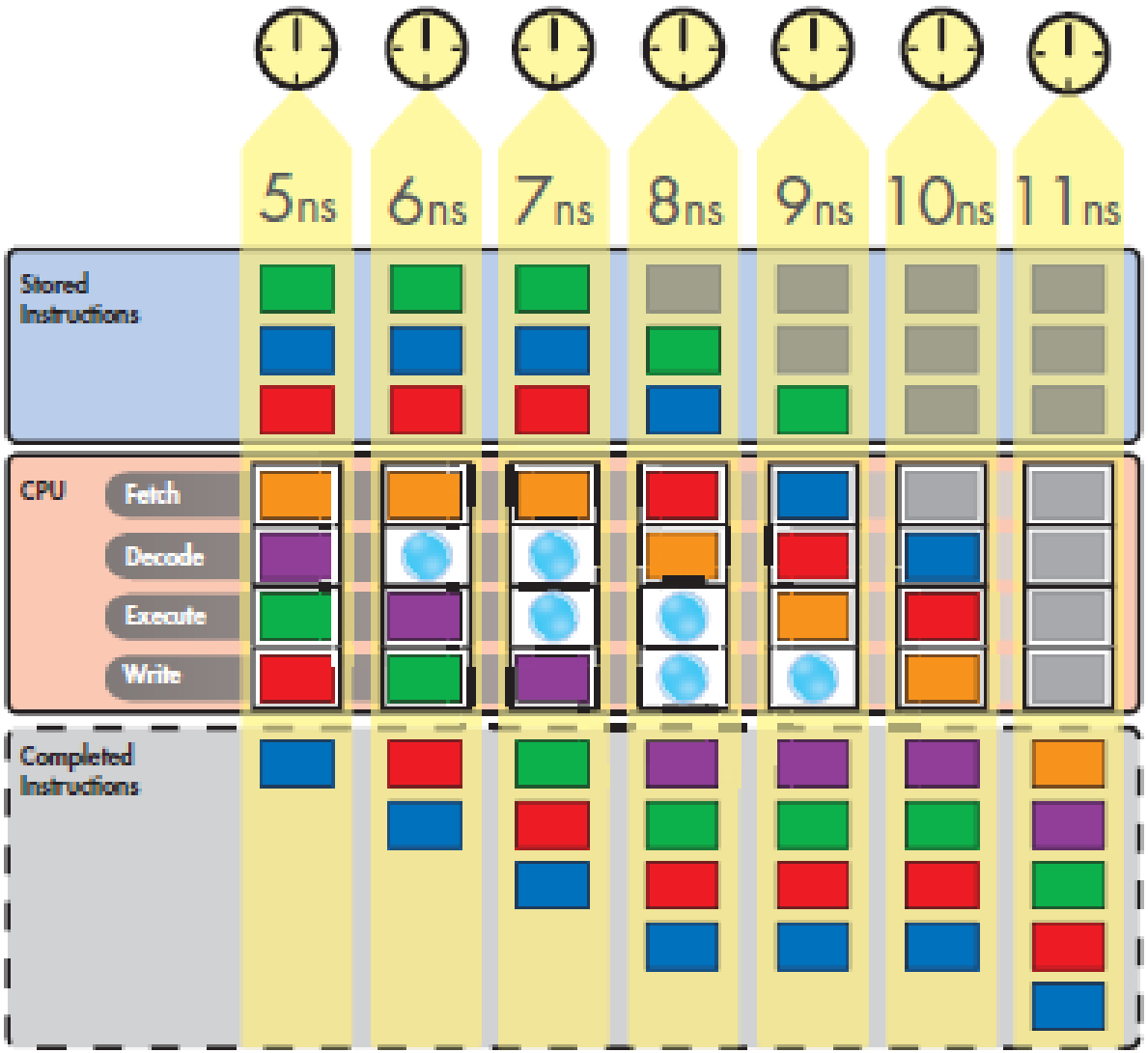
Procesor jednocyklowy



Procesor wielocyklowy



Zatrzymania „bąble”



Historia

Komputer superskalarny przetwarza więcej niż jeden skalar (liczbę, tutaj całkowitą) jednocześnie.

Pierwszy komercyjny komputer superskalarny: RS6000, IBM w roku 1999.

Intel w roku 1993 – dwa ALU.

DLW-2

DLW-2 – wersja DLW-1 z dwoma ALU.

Wspólne rejestry.

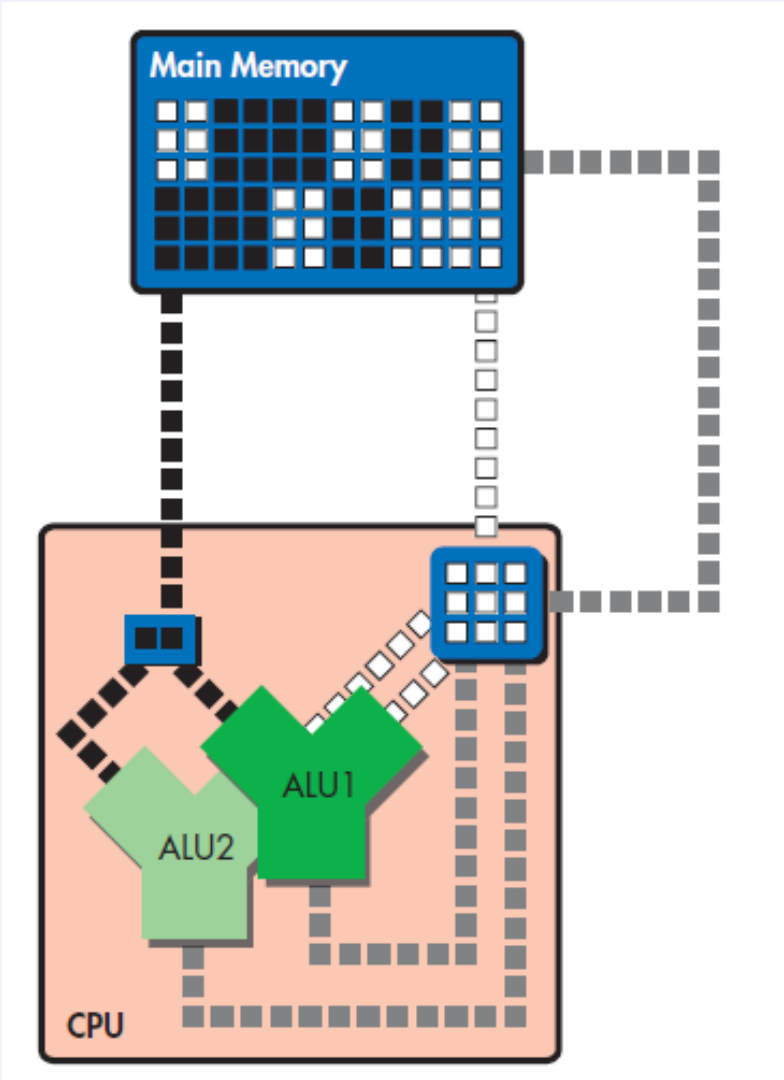
DLW-2

DLW-2 – wersja DLW-1 z dwoma ALU.

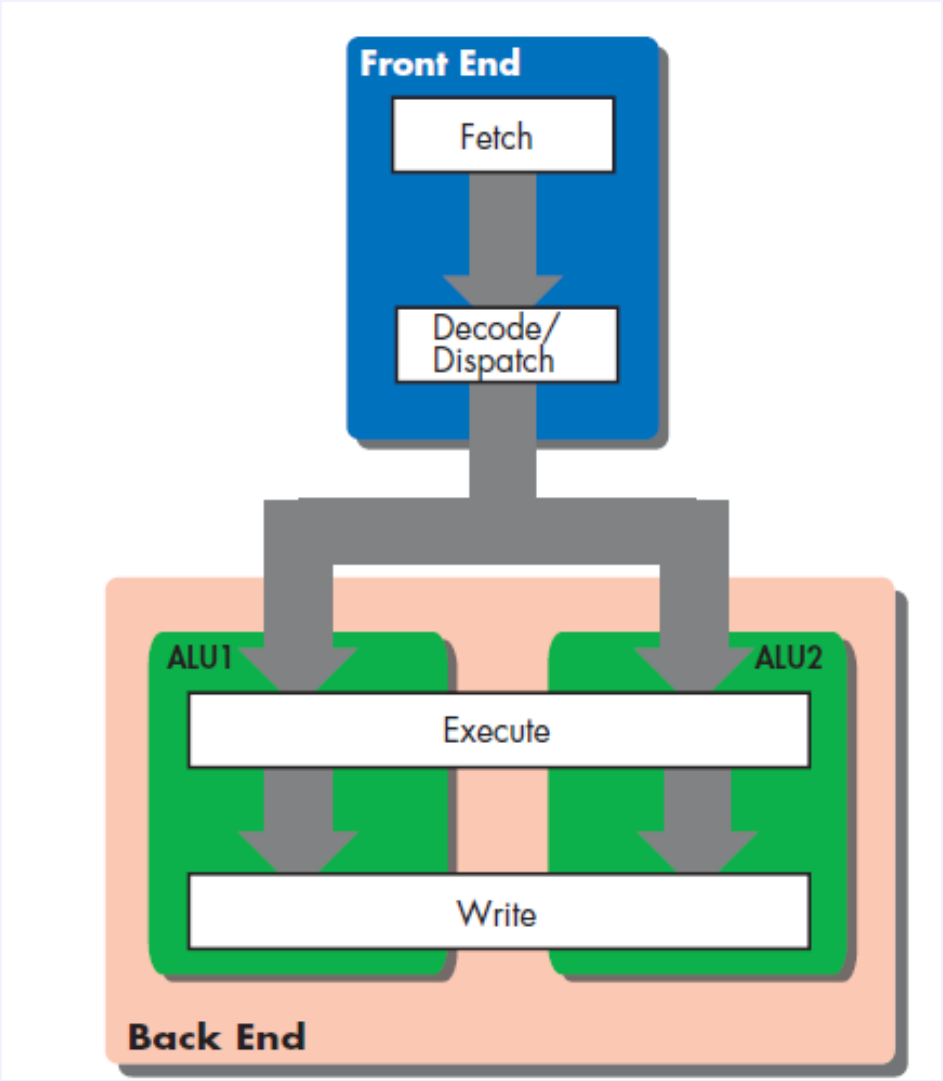
Wspólne rejestry.

Analogia biurowa:
do obsługi klientów umieszczamy drugiego urzędnika w tym samym pokoju biurowym, z tymi samymi zasobami i zbiorami dokumentów bieżących.

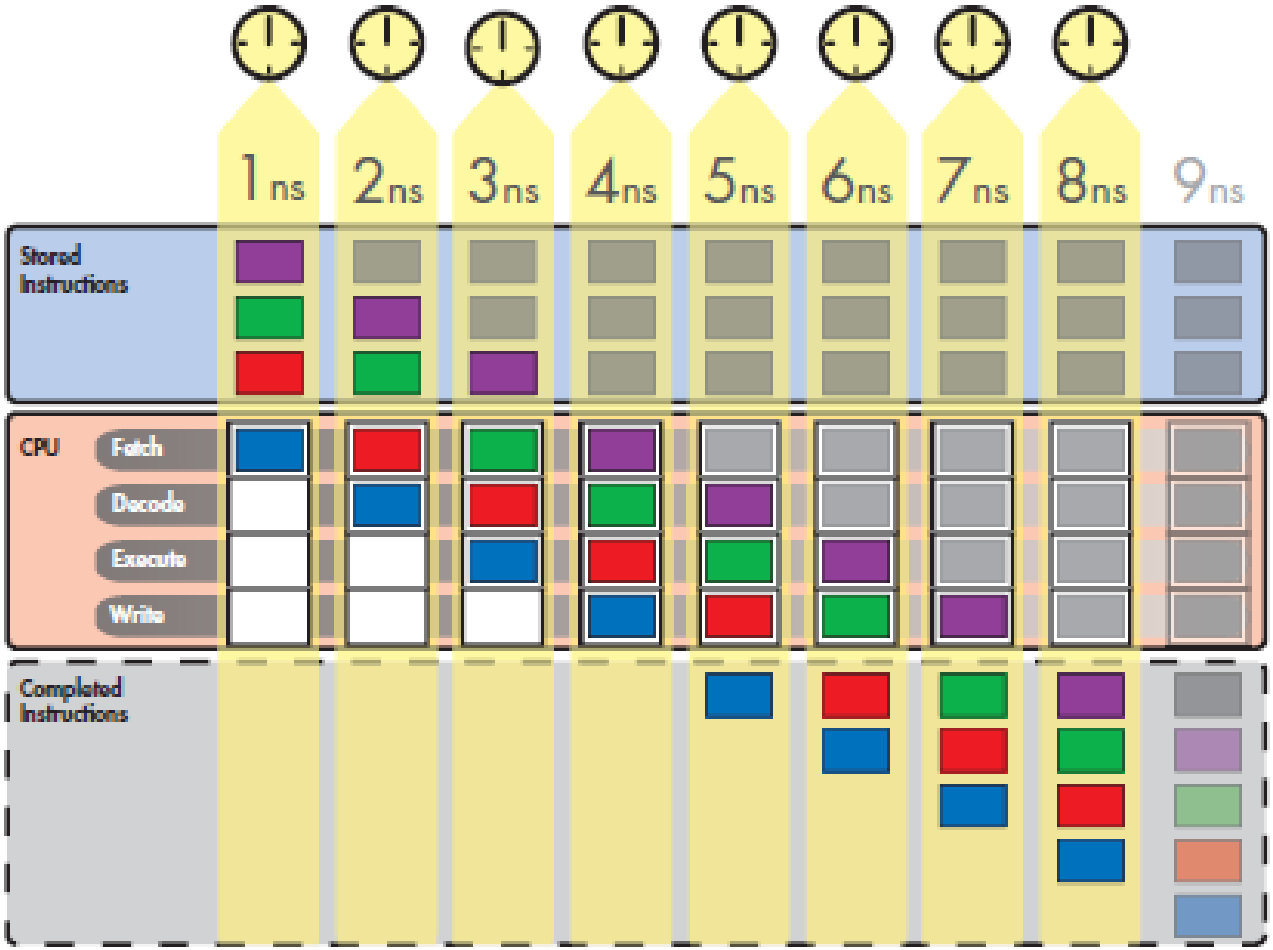
DLW-2 – schemat



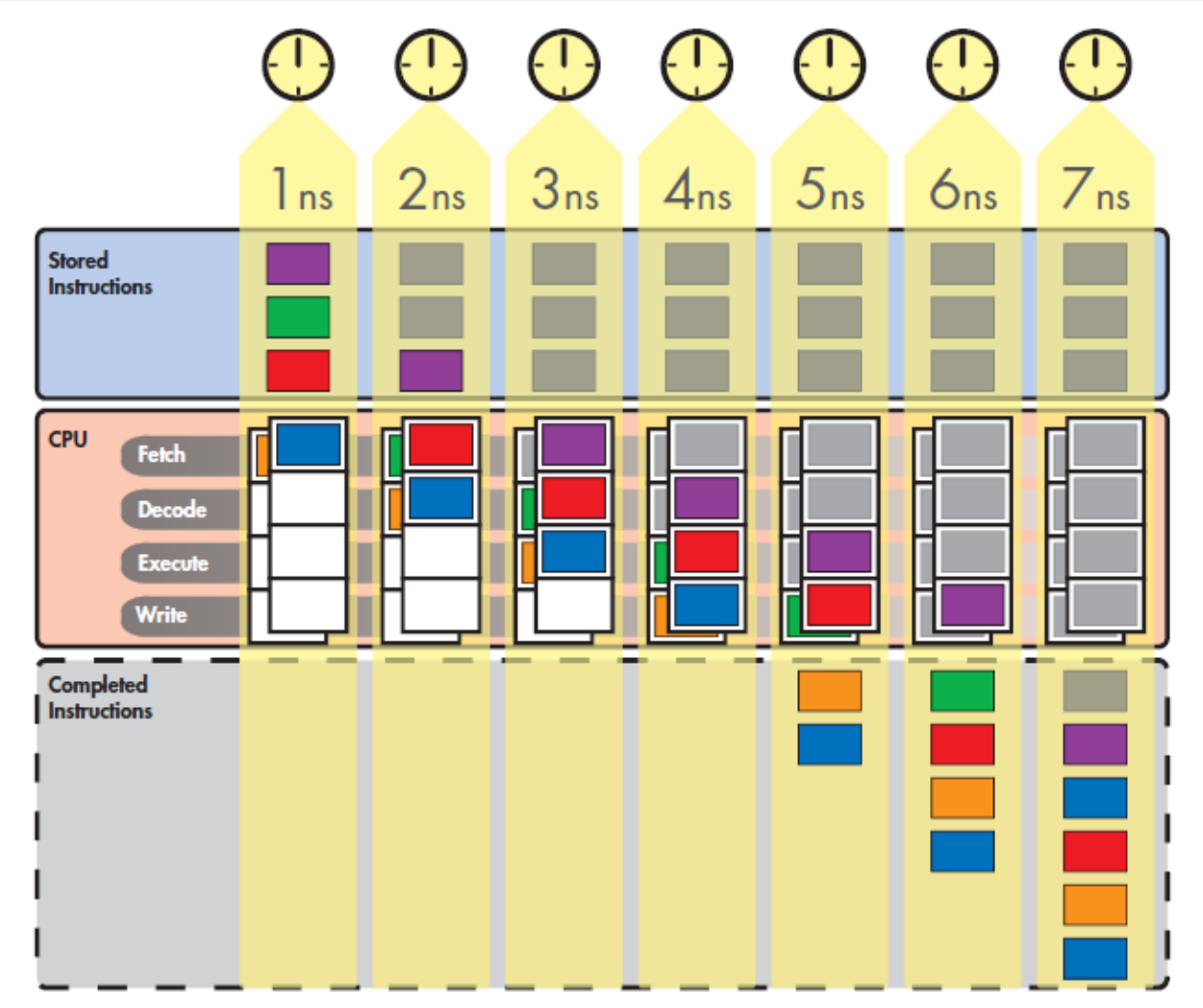
Potok w DLW-2


Front End
Fetch
Decode/
Dispatch
ALU1
ALU2
Execute
Write
Back End

Przetwarzanie potokowe



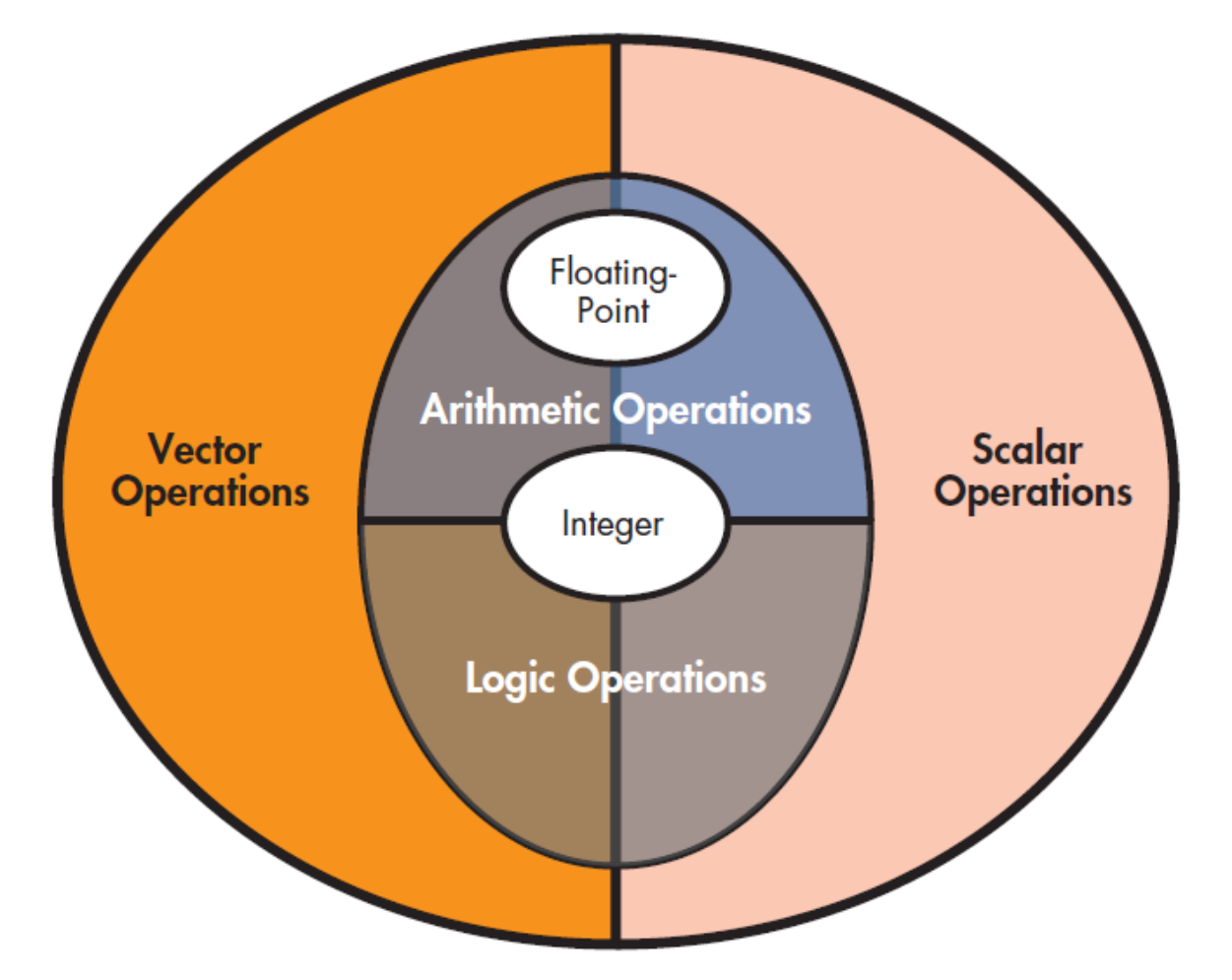
Przetwarzanie potokowe – superskalarne



Przetwarzanie skalarne i wektorowe

	Integer	Floting-Point
Scalar	14	1.01
	-500	12.34
	10	-0.0001
Vector	$\{5, -6, 7, 8\}$	$\{1.1, 3.4, 5.0, 11, 23\}$

Formaty


Floating-
Point
Arithmetic Operations
Vector
Operations
Scalar
Operations
Integer
Logic Operations

Intel Pentium