Rozwiązywanie konfliktów danych i sterowania w architekturze potokowej

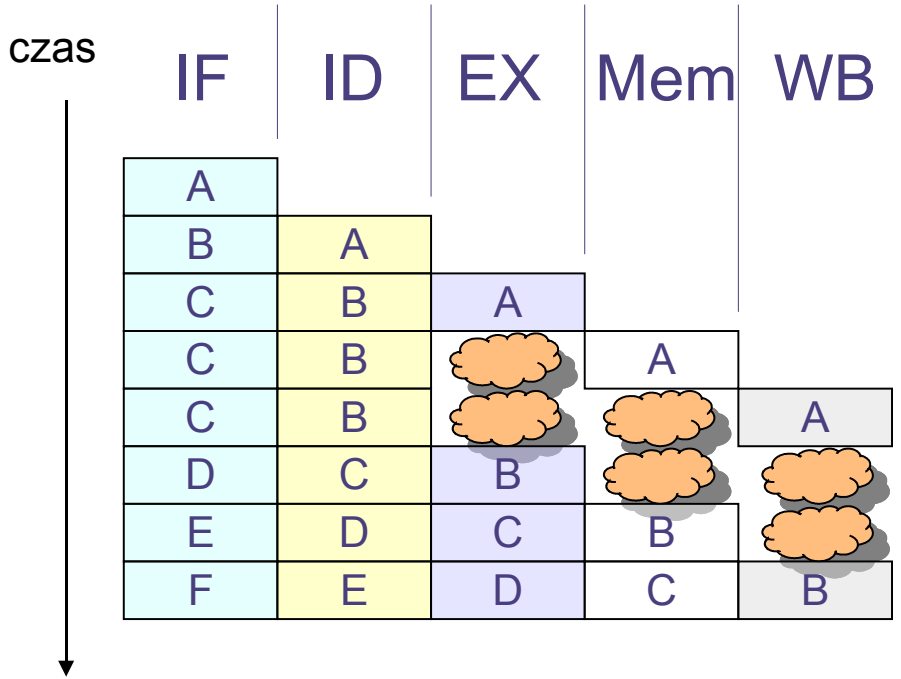
Konflikty w przetwarzaniu potokowym

- Konflikt danych – *Data Hazard*
  - Wstrzymywanie kolejki – *Pipeline Stall*
  - Optymalizacja kodu (metody programowe)
  - *Forwarding* (metoda sprzętowa)
- Konflikt sterowania – *Control Hazard*
  - Unieważnienie instrukcji – *Pipeline Flush*
  - Opóźnienie skoku – *Delayed Branching*
  - Wczesne wykrywanie skoku – *Early Branch Detection*
  - Tablica historii skoków – *Branch History Table*
- Konflikt zasobów – *Structural Hazard*
  - powielenie sprzętu

Wstrzymywanie kolejki

- Powtórzenie oraz unieważnienie niektórych faz instrukcji
  - faza ID może być wykonywana równocześnie z WB, przy zapewnieniu synchronizacji przepływu danych – wówczas wstrzymanie kolejki obejmuje tylko 2 cykle

```
A)  ADD  R1,R2,R3
B)  SUB  R4,R3,R5
C)  MUL  R4,R3,R1
D)  ...
E)  ...
F)  ...
```



Programowe „wstrzymywanie” kolejki

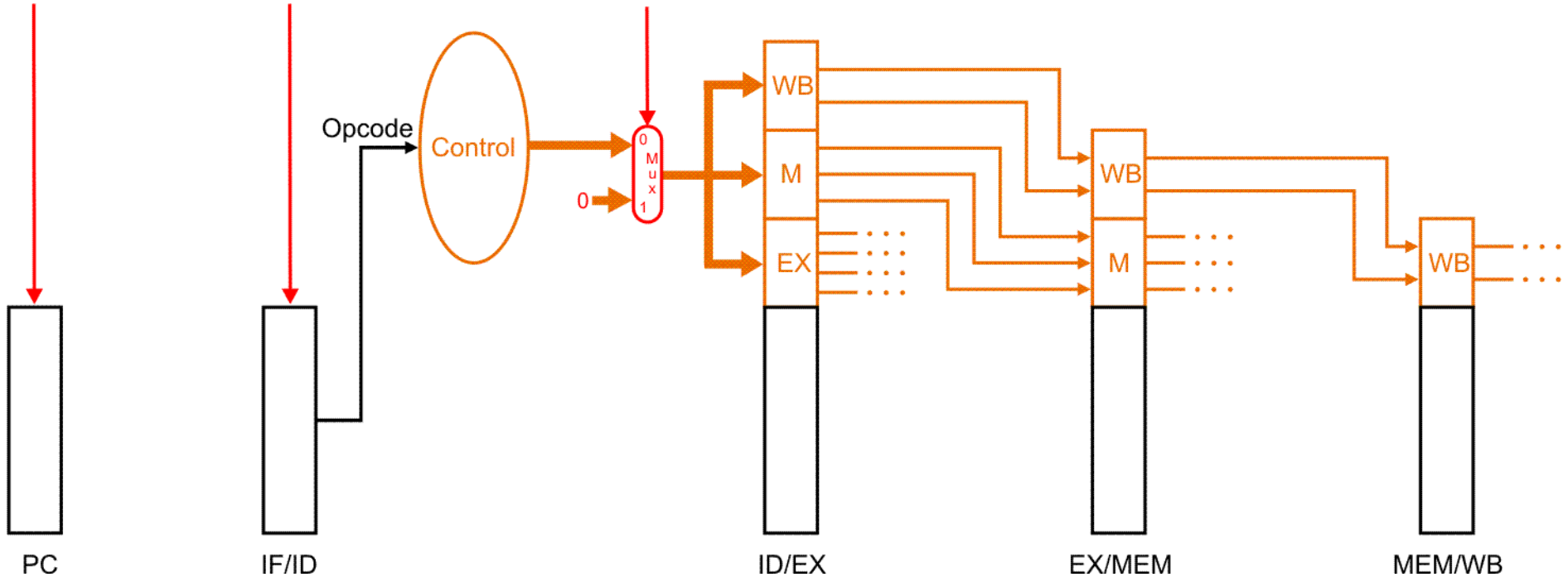
- Korekta kodu za pomocą instrukcji NOP (*No Operation*)
  - nie jest to "metoda optymalizacji", ale wstawianie NOP może okazać się niezbędne w niektórych przypadkach, gdy brak jest innych mechanizmów sprzętowych

```
next: LW   R1,0(R3)
      MUL  R1,R1,R1
      SW   R1,0(R3)
      SUBI R3,#4,R3
      BNE  R0,R3,next
      ...
```

```
next: LW   R1,0(R3)
      NOP
      MUL  R1,R1,R1
      NOP
      NOP
      SW   R1,0(R3)
      SUBI R3,#4,R3
      NOP
      NOP
      BNE  R0,R3,next
      ...
```

Sprzętowe wstrzymywanie kolejki

- Zamiana sygnałów na nieaktywne (0) dla Ex, Mem,WB
- Brak taktowania rejestrów PC i IF/ID

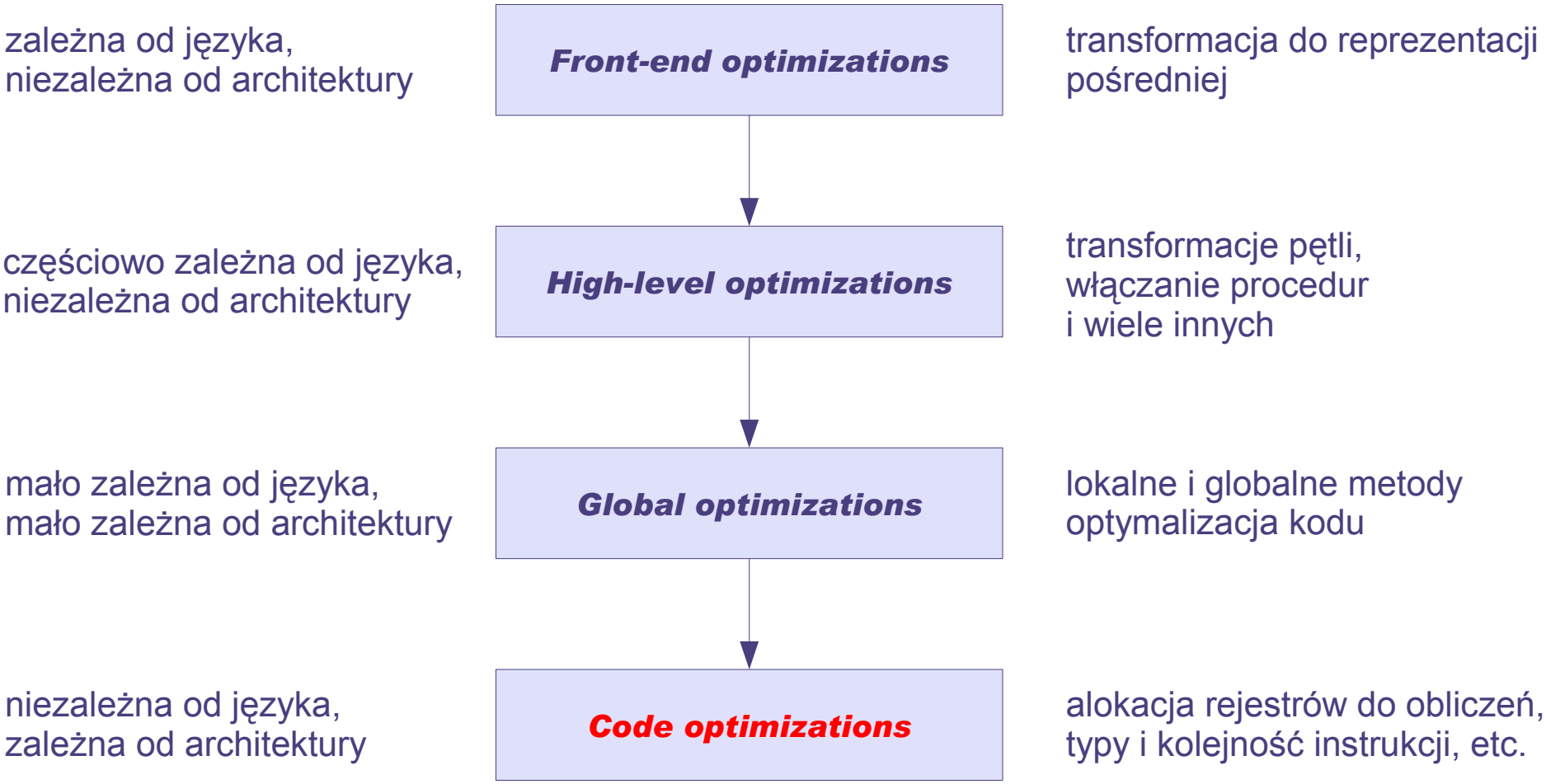


Optymalizacja kodu

- Metody statyczne: optymalizacja podczas kompilacji programu (*optimising compiler*)
  - np. gcc -On
- Metody dynamiczne: sprzętowe ustalanie optymalnej kolejności wykonywania instrukcji
  - dynamic scheduling
  - out of order execution
  - speculative execution

Poza zakresem tego wykładu

Statyczna optymalizacja kodu



Metody optymalizacji

gcc -o test test.c -O1

gcc -o test test.c -O2 -march=athlon

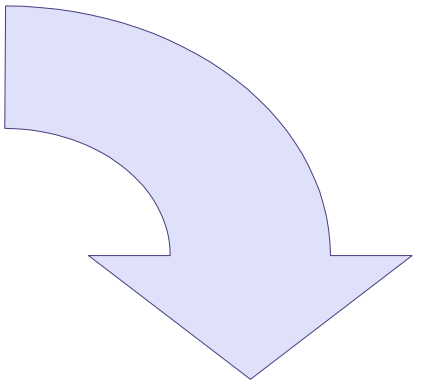
Target CPU Types	-march= Type
i386 DX/SX/CX/EX/SL	i386
i486 DX/SX/DX2/SL/SX2/DX4	i486
487	i486
Pentium	pentium
Pentium MMX	pentium-mmx
Pentium Pro	pentiumpro
Pentium II	pentium2
Celeron	pentium2
Pentium III	pentium3
Pentium 4	pentium4
Via C3	c3
Winchip 2	winchip2
Winchip C6-2	winchip-c6
AMD K5	i586
AMD K6	k6
AMD K6 II	k6-2
AMD K6 III	k6-3
AMD Athlon	athlon
AMD Athlon 4	athlon
AMD Athlon XP/MP	athlon
AMD Duron	athlon
AMD Tbird	athlon-tbird

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

Optymalizacja kodu - przykład

- Niskopoziomowa optymalizacja kodu dla architektury potokowej z *Single Delay Slot*

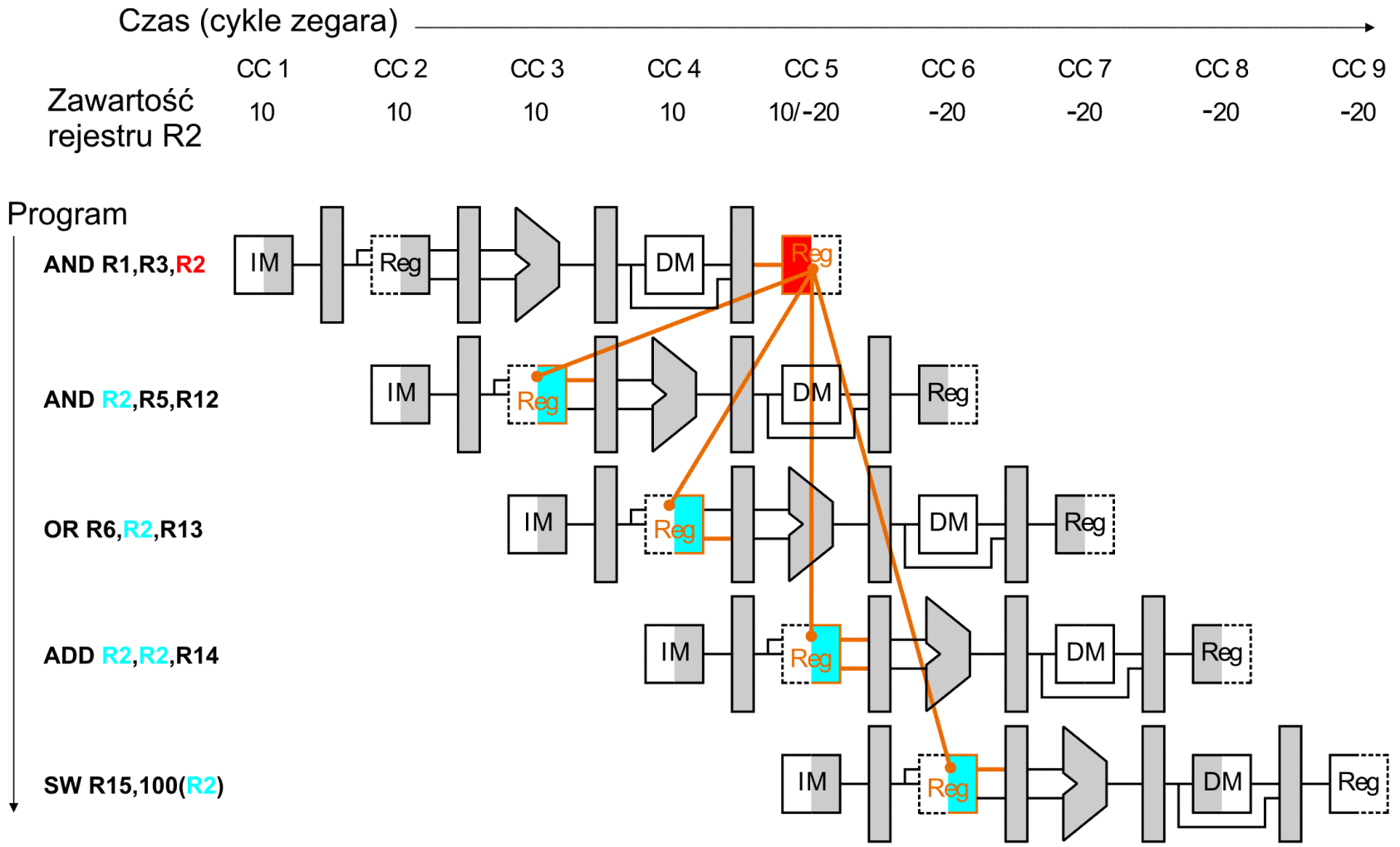
```
next: LW   R1,0(R3)
      MUL  R1,R1,R1
      SW   R1,0(R3)
      SUBI R3,#4,R3
      BNE  R0,R3,next
      ...
```



```
next: LW   R1,0(R3)
      SUBI R3,#4,R3
      MUL  R1,R1,R1
      BNE  R0,R3,next
      SW   R1,4(R3)
      ...
```

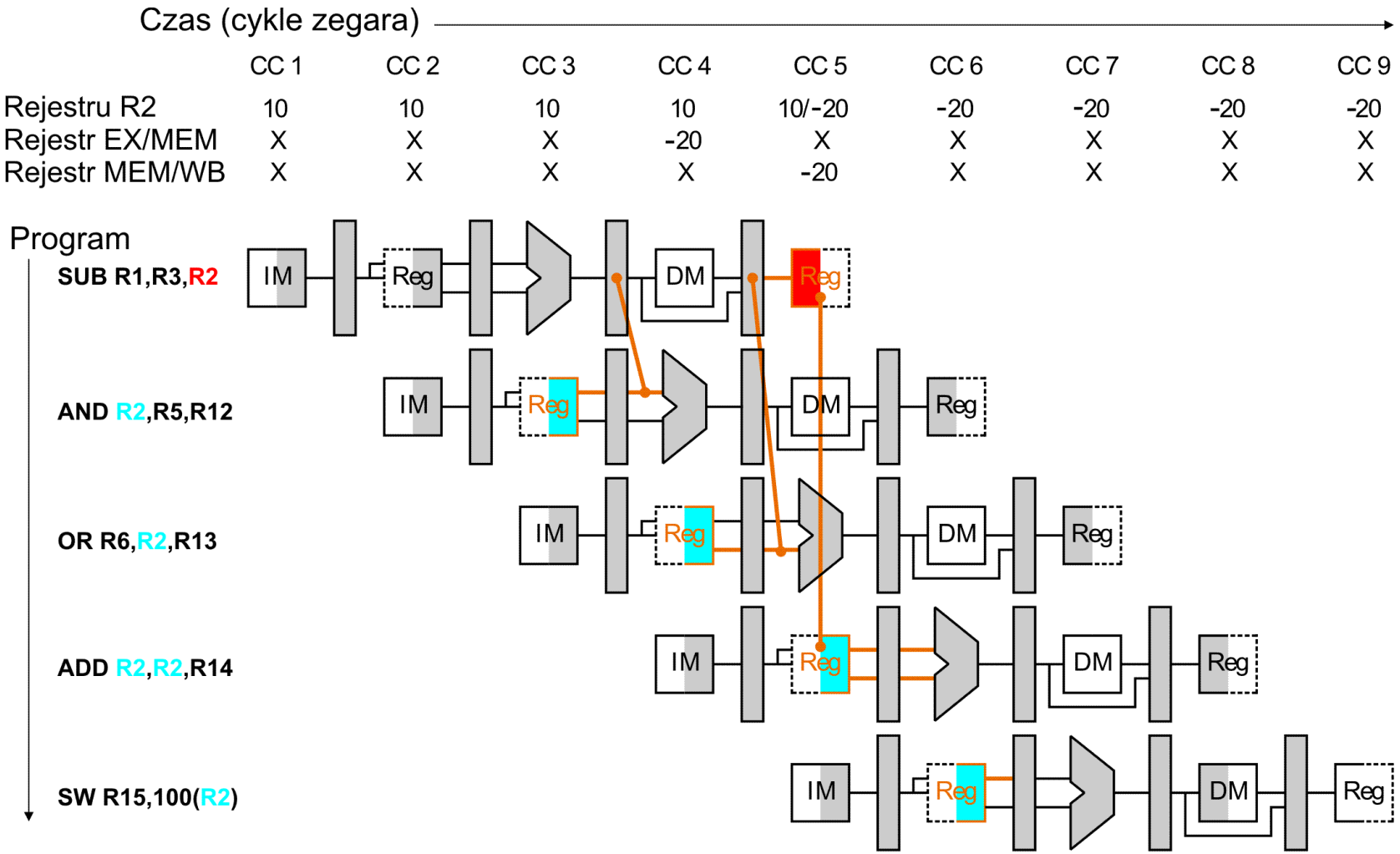
Forwarding

- Sprzętowa metoda rozwiązywania konfliktu danych



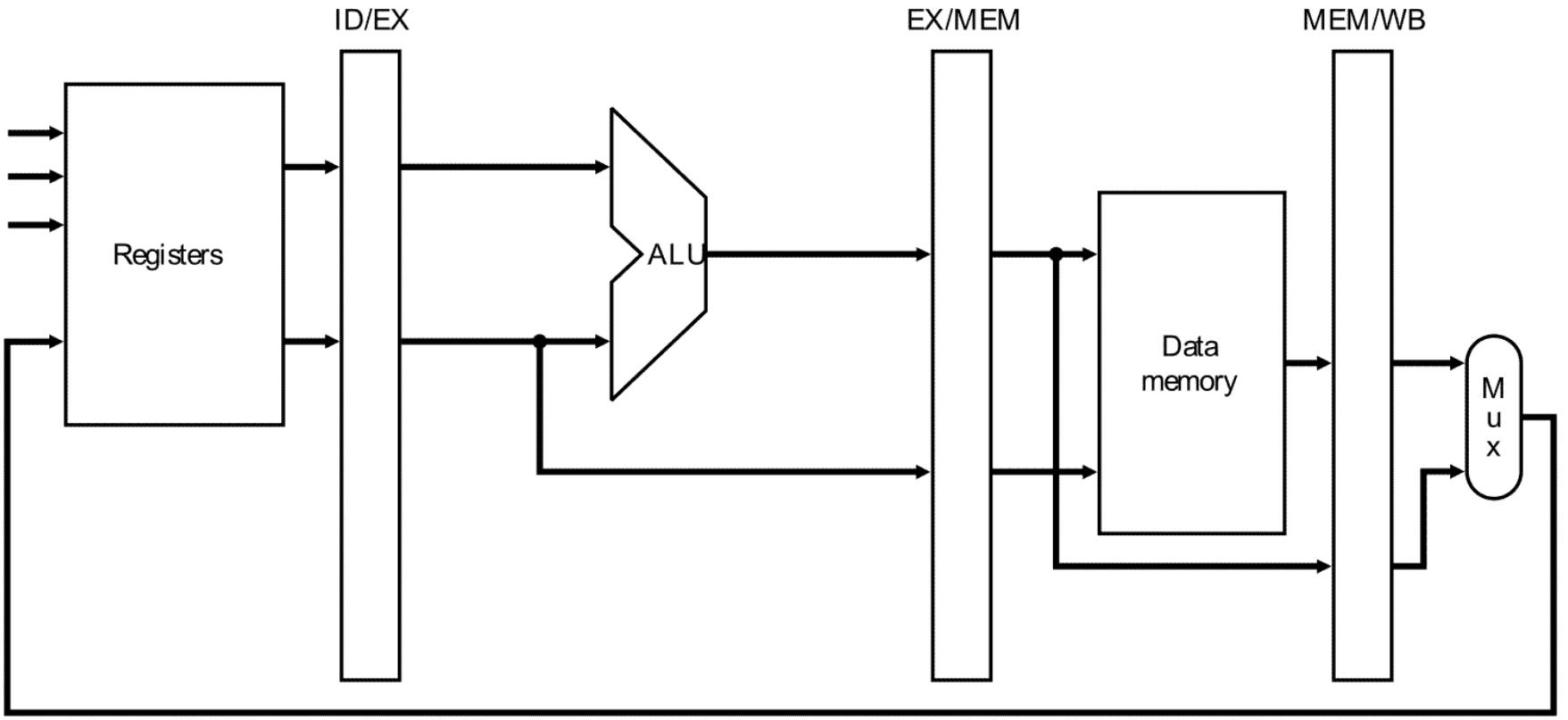
Forwarding

- Idea: transfery pomiędzy rejestrami pośrednimi



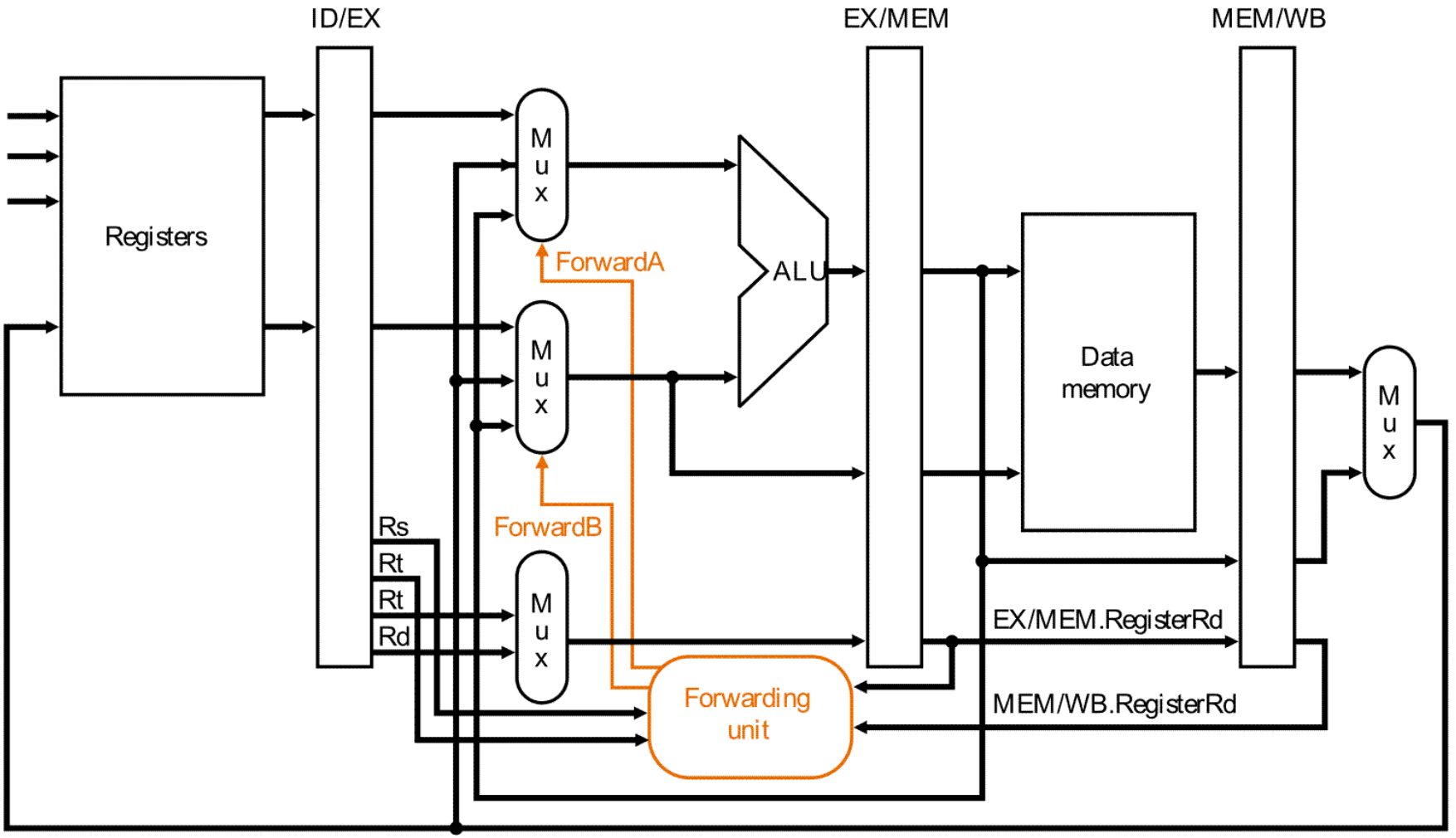
Transfery (bez forwarding'u)

- Tylko "w przód" oraz końcowa modyfikacja rejestru



Transfery (z forwarding'iem)

- EX/MEM → ALU
- MEM/WB → ALU



Forwarding

- Sprzętowa eliminacja prawie wszystkich konfliktów danych (pomiędzy wybranymi fazami)
- Transfery najbardziej aktualnych wartości rejestrów z faz Ex i Mem na wejście ALU
- Blok logiki kombinacyjnej porównujący (komparatory):
  - numery rejestrów które będą modyfikowane (Ex/Mem.Rd lub Mem/WB.Rd)
  - numery rejestrów, które są operandami dla ALU (ID/Ex.Rs lub ID/Ex.Rt)
- Plik rejestrów jest modyfikowany zgodnie z kolejnością instrukcji w programie

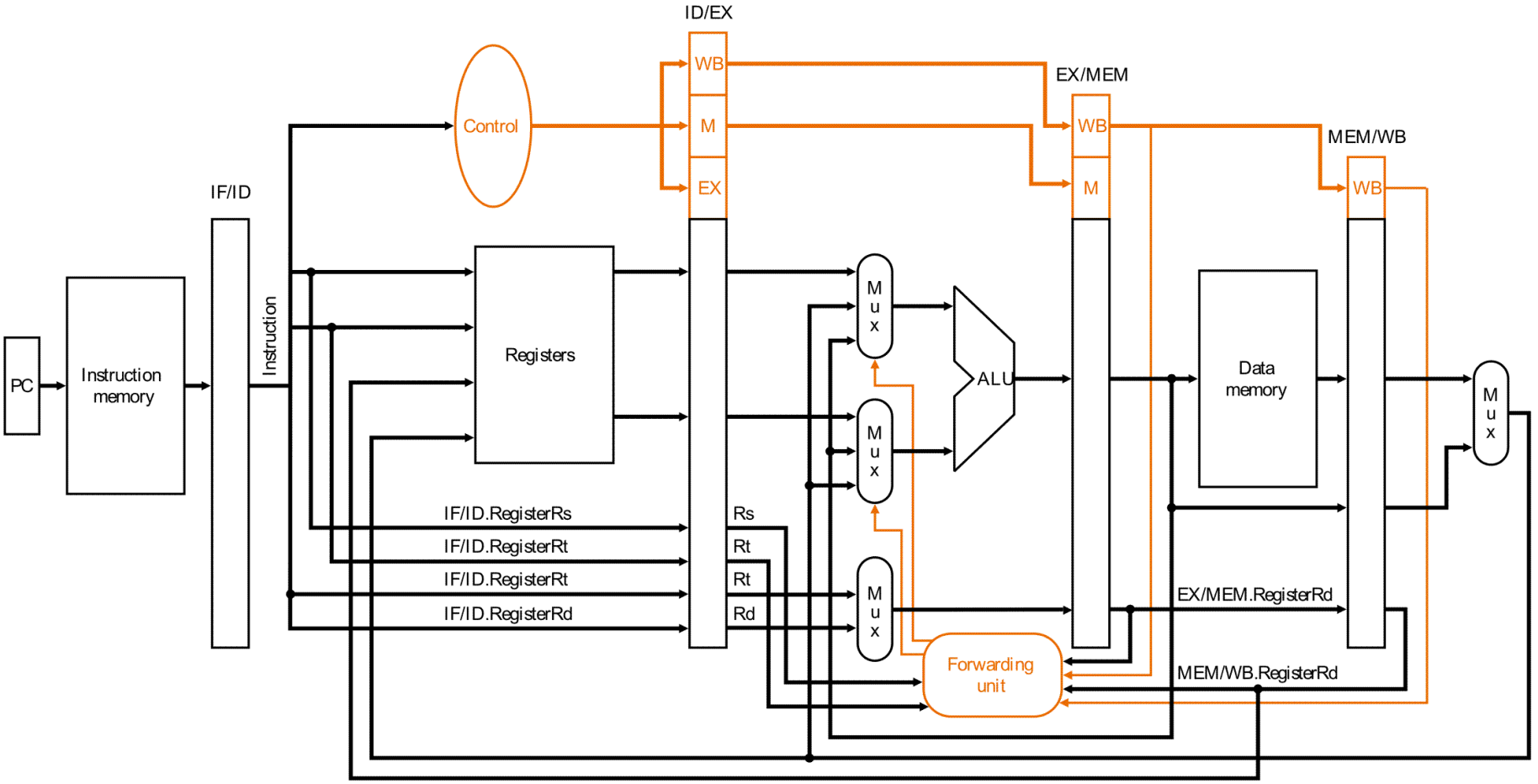
Forwarding

- Multipleksery na wejściu ALU są sterowane wyłącznie przed układ forwarding'u

Multiplexer	*Dane*	*Opis*
For_A=00	ID/EX	Register File to ALU
For_A=10	EX/MEM	ALU to ALU
For_A=01	MEM/WB	MemData or ALU toALU
For_B=00	ID/EX	Register File to ALU
For_B=10	EX/MEM	ALU to ALU
For_B=01	MEM/WB	MemData or ALU toALU

- Forwarding jest niewidoczny dla jednostki sterującej i nie zwiększa jej komplikacji
- W przypadku "głębokiej kolejki" komplikacja forwarding'u ogranicza jego stosowalność

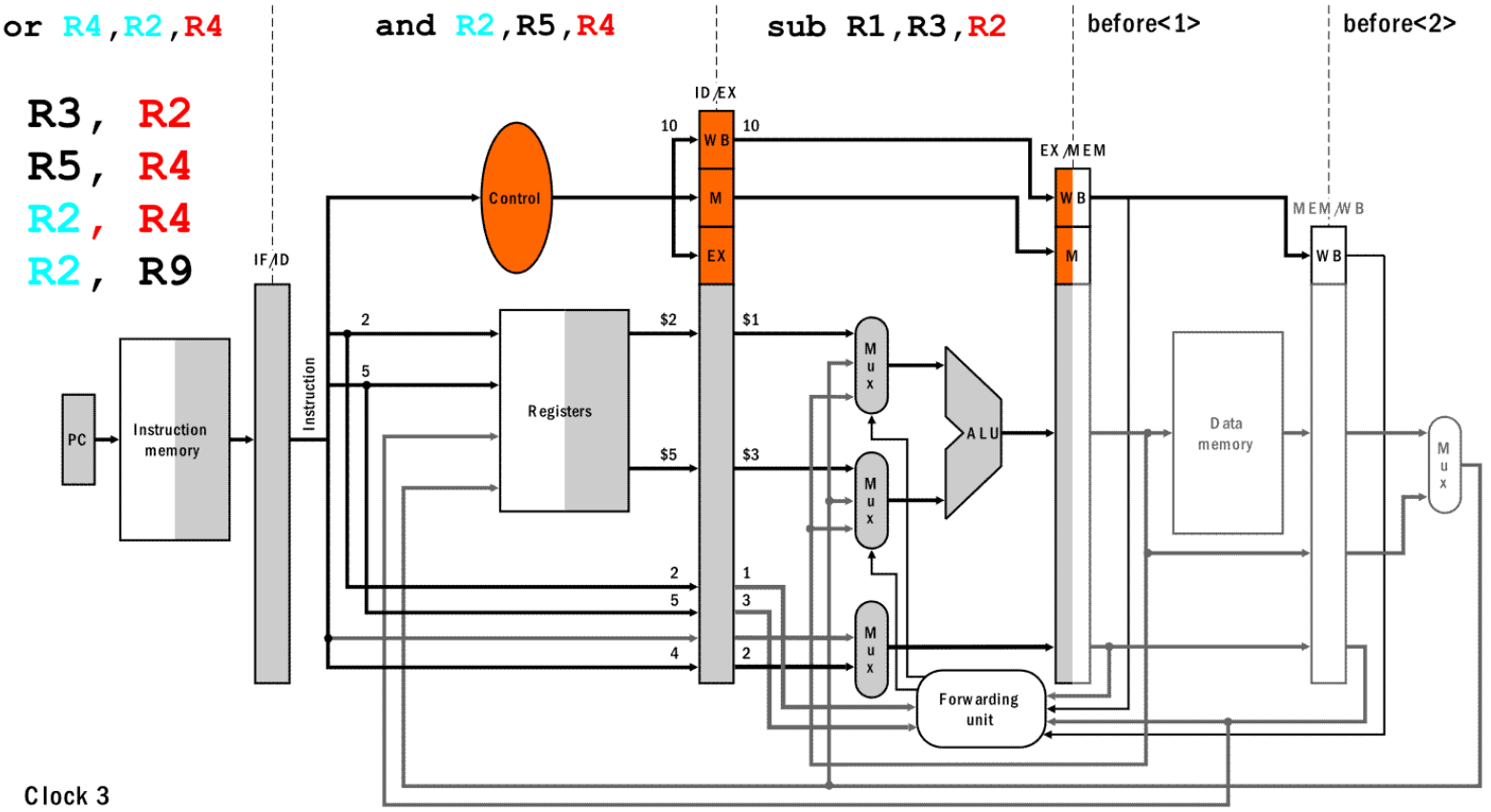
Architektura z forwarding'iem



(tutaj jeszcze bez instrukcji skoków)

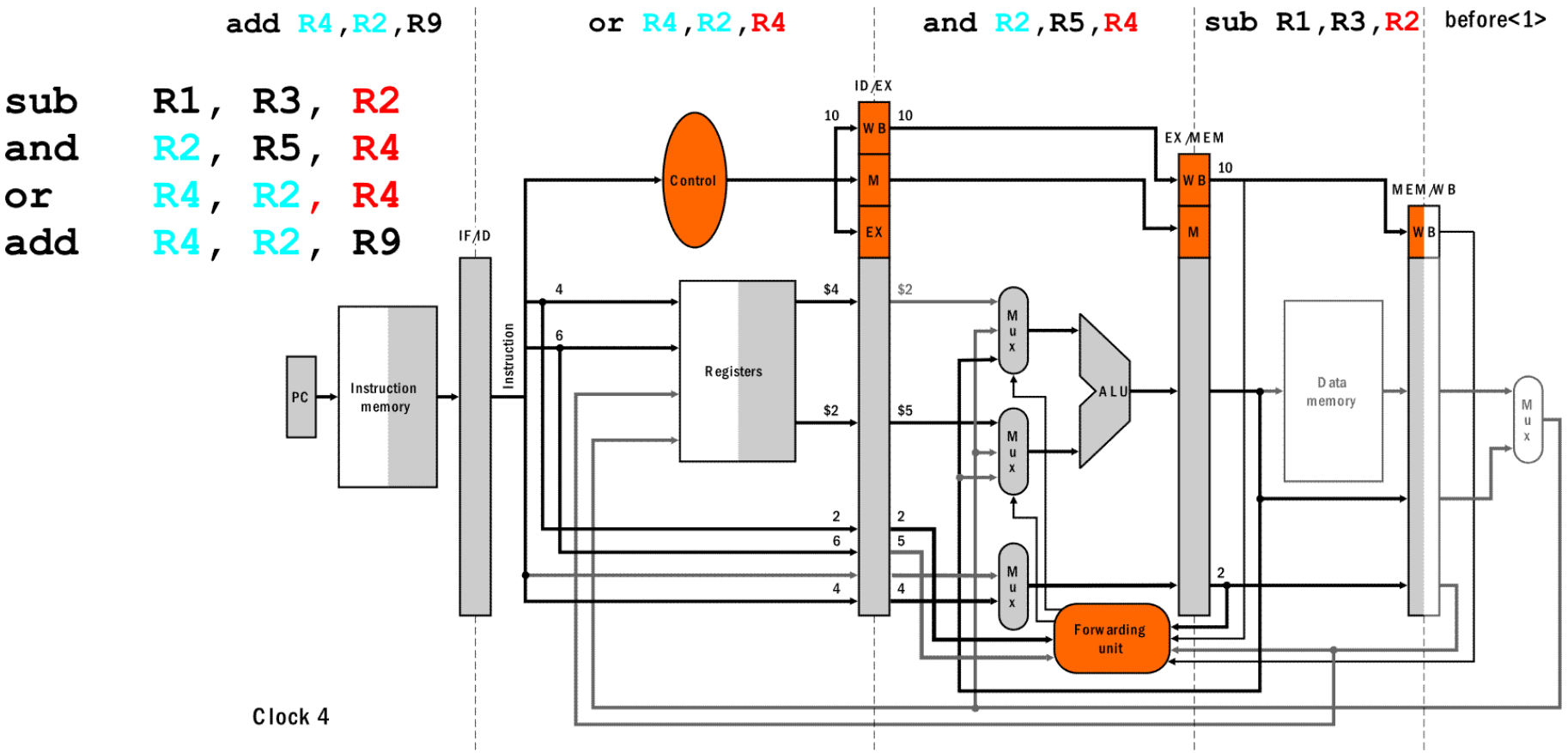
Forwarding w działaniu (1)

```
sub   R1, R3, R2
and   R2, R5, R4
or    R4, R2, R4
add   R4, R2, R9
```



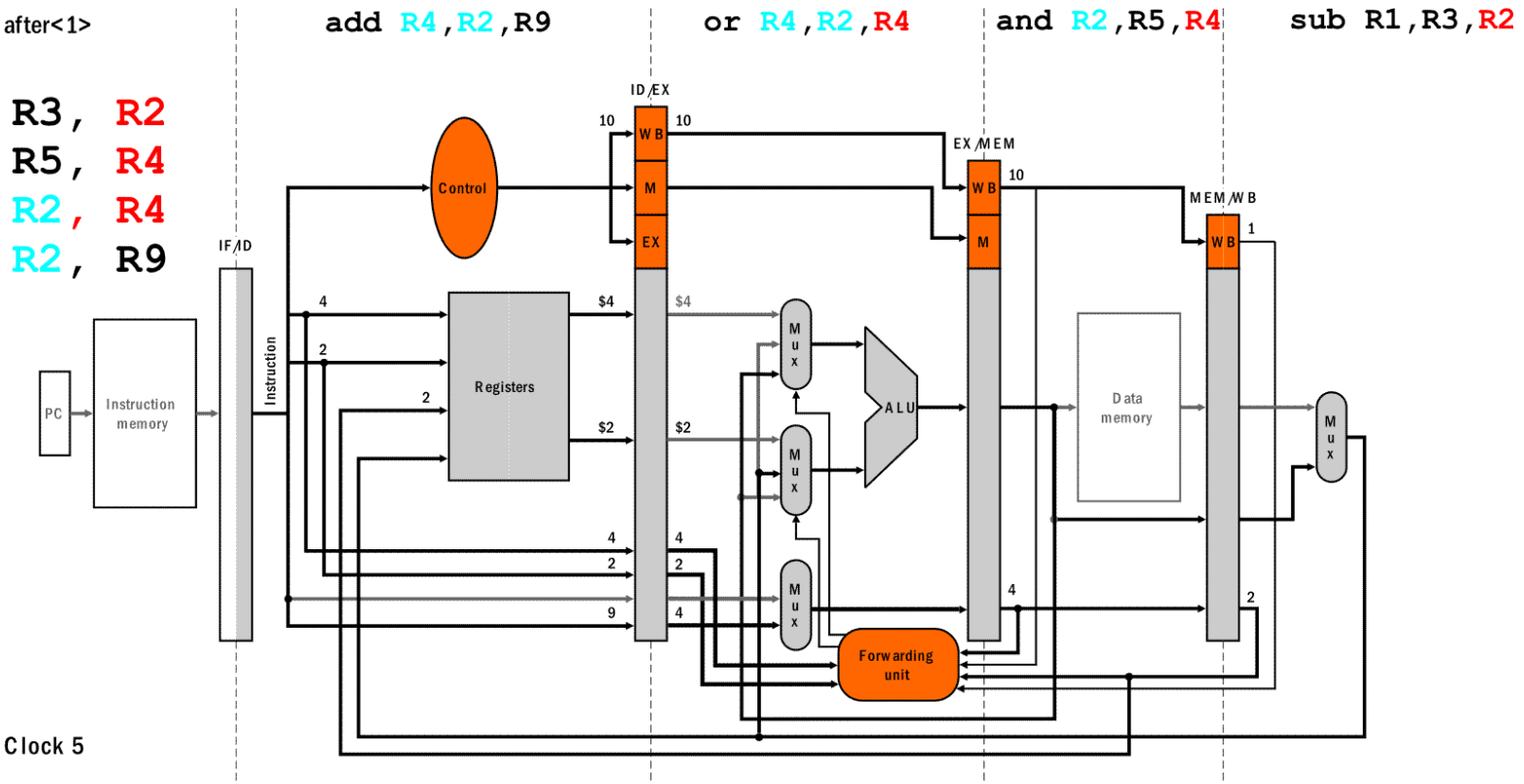
Forwarding w działaniu (2)

```
sub   R1, R3, R2
and   R2, R5, R4
or    R4, R2, R4
add   R4, R2, R9
```



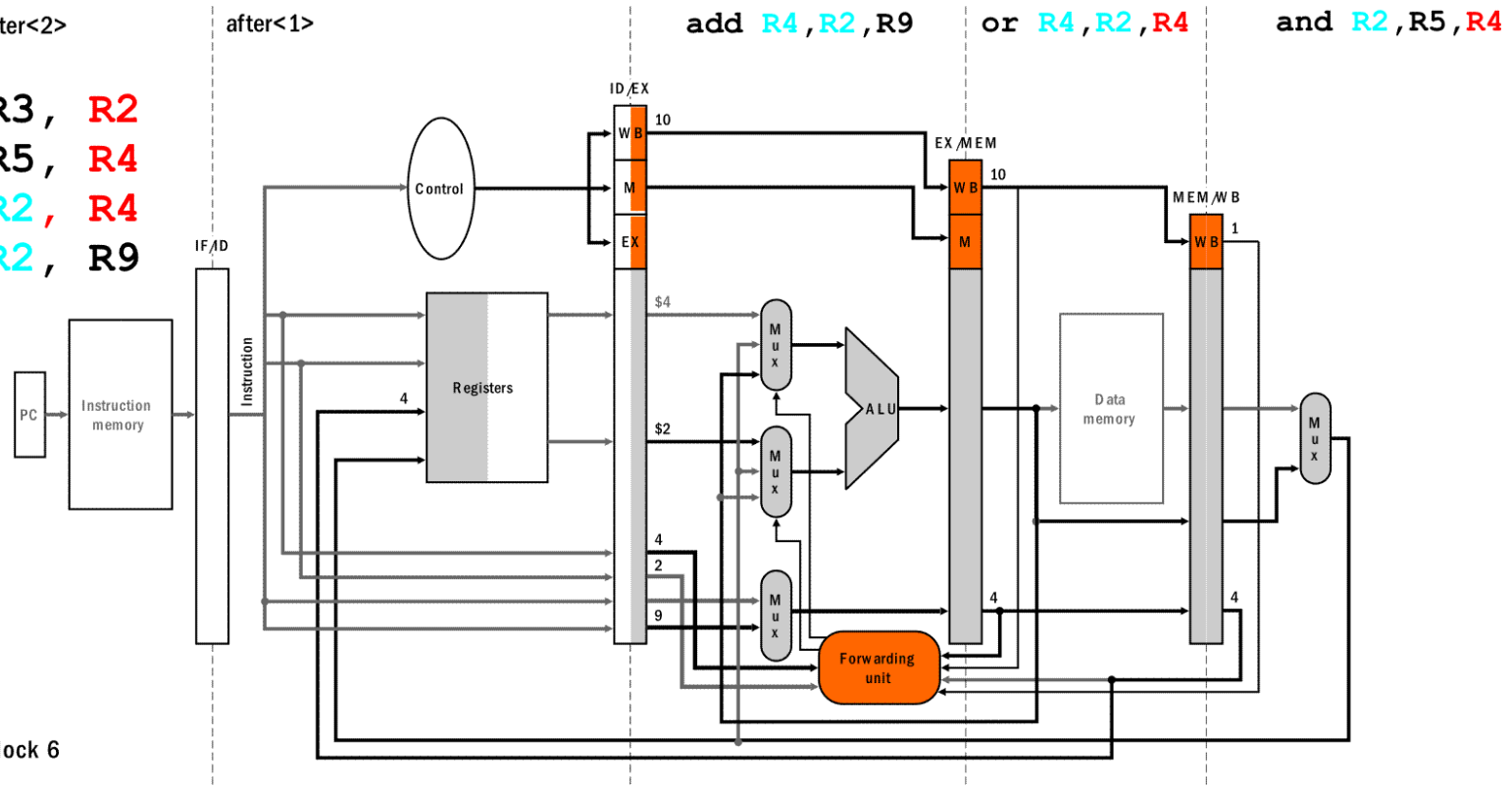
Forwarding w działaniu (3)

```
sub   R1, R3, R2
and   R2, R5, R4
or    R4, R2, R4
add   R4, R2, R9
```



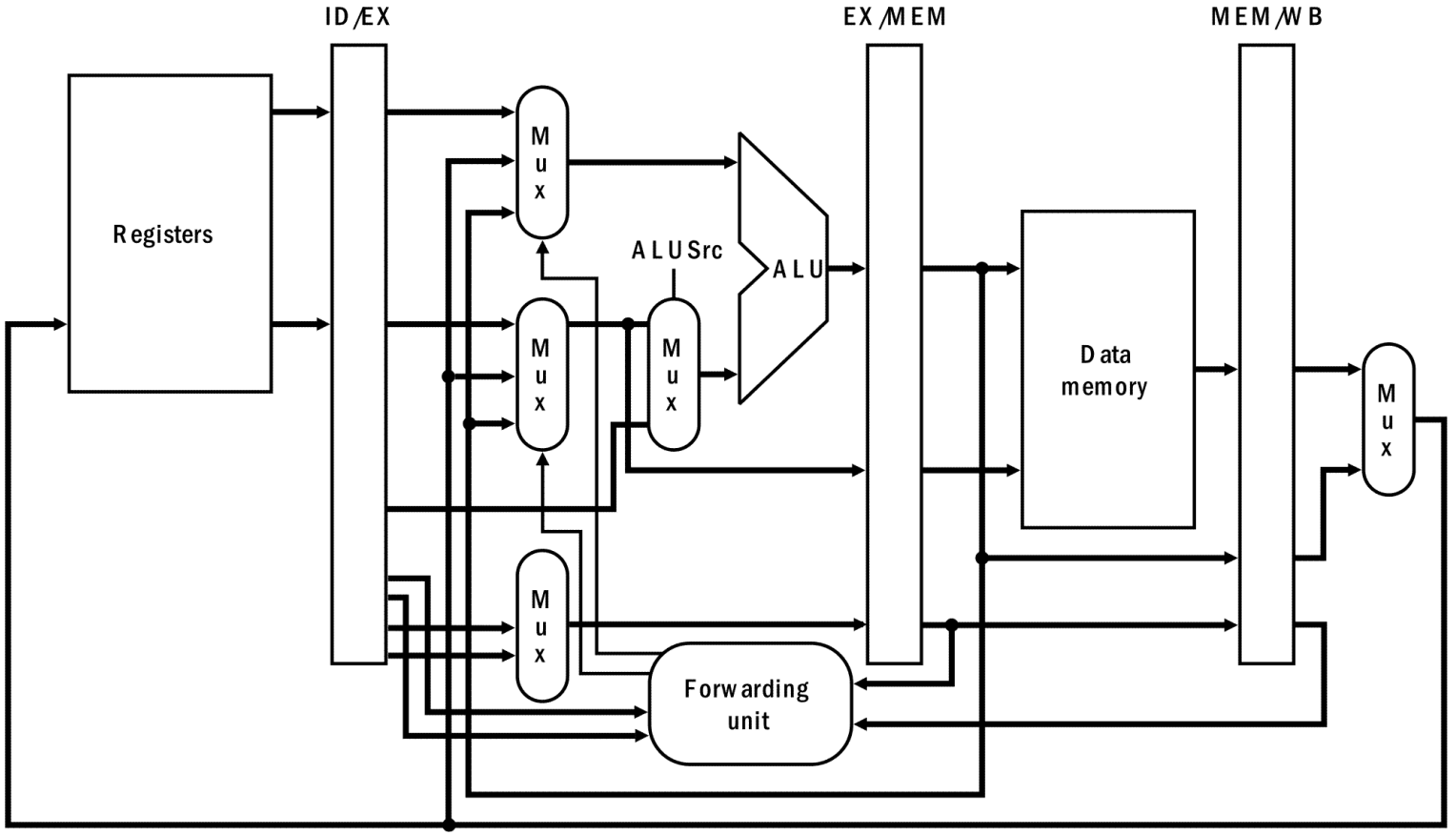
Forwarding w działaniu (4)

```
sub   R1, R3, R2
and   R2, R5, R4
or    R4, R2, R4
add   R4, R2, R9
```



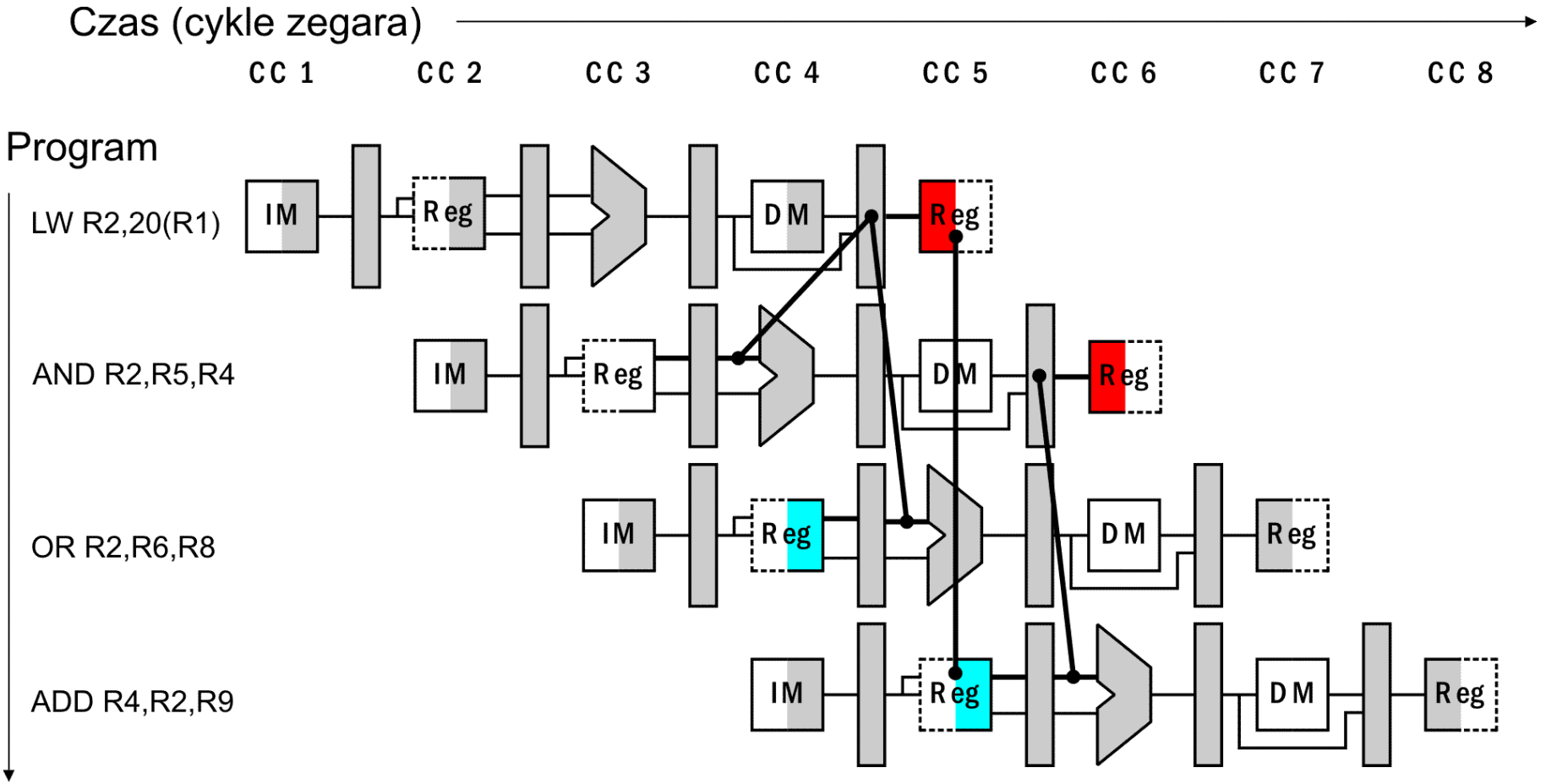
Forwarding – korekta dla ALUsrc

- Sygnałem ALUSrc steruje jednostka centralna
- Niezależność sterowania i forwarding'u wymaga niezależnych multiplekserów



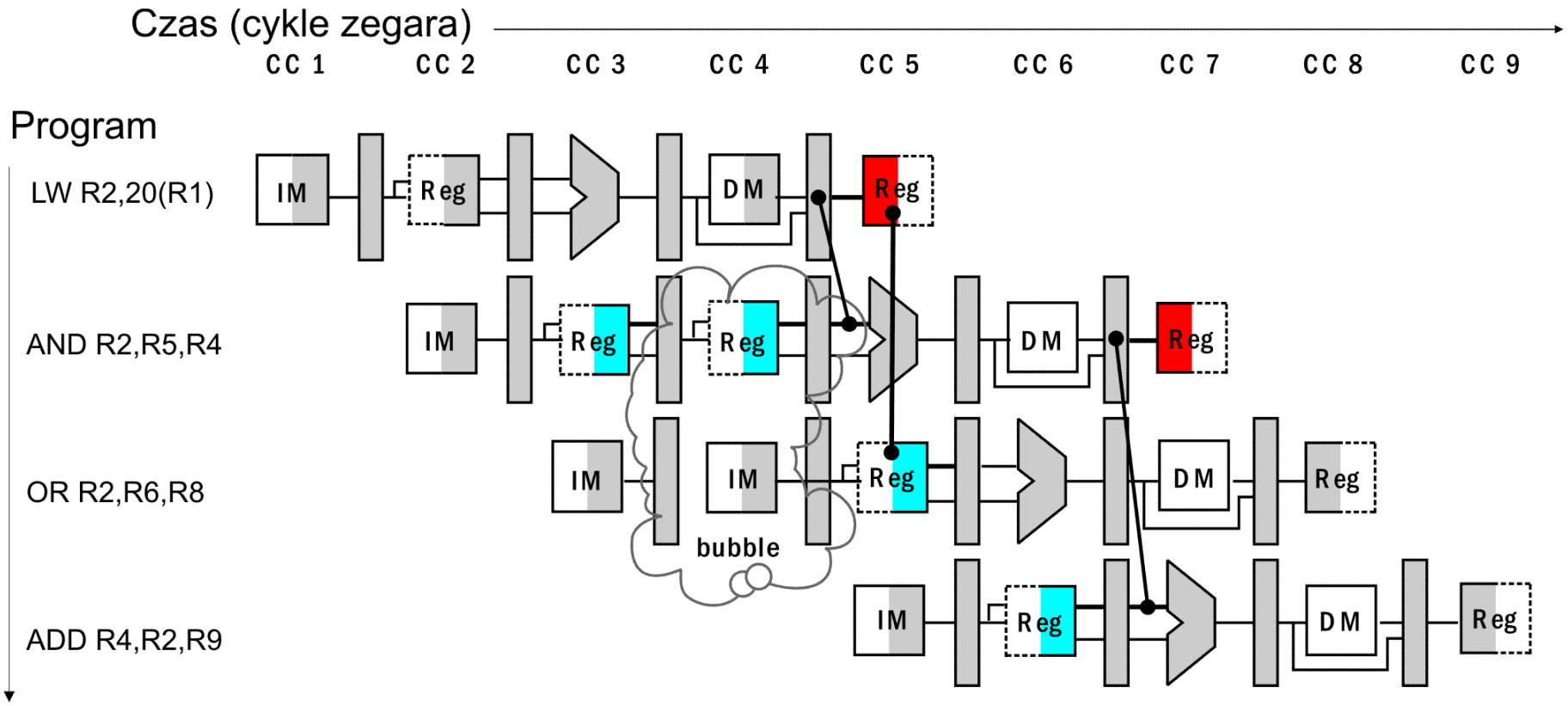
"Trudne" konflikty danych

- Forwarding nie rozwiązuje wszystkich konfliktów danych
- *Read After Write* (RAW) – tutaj: LW & ADD



"Trudne" konflikty danych

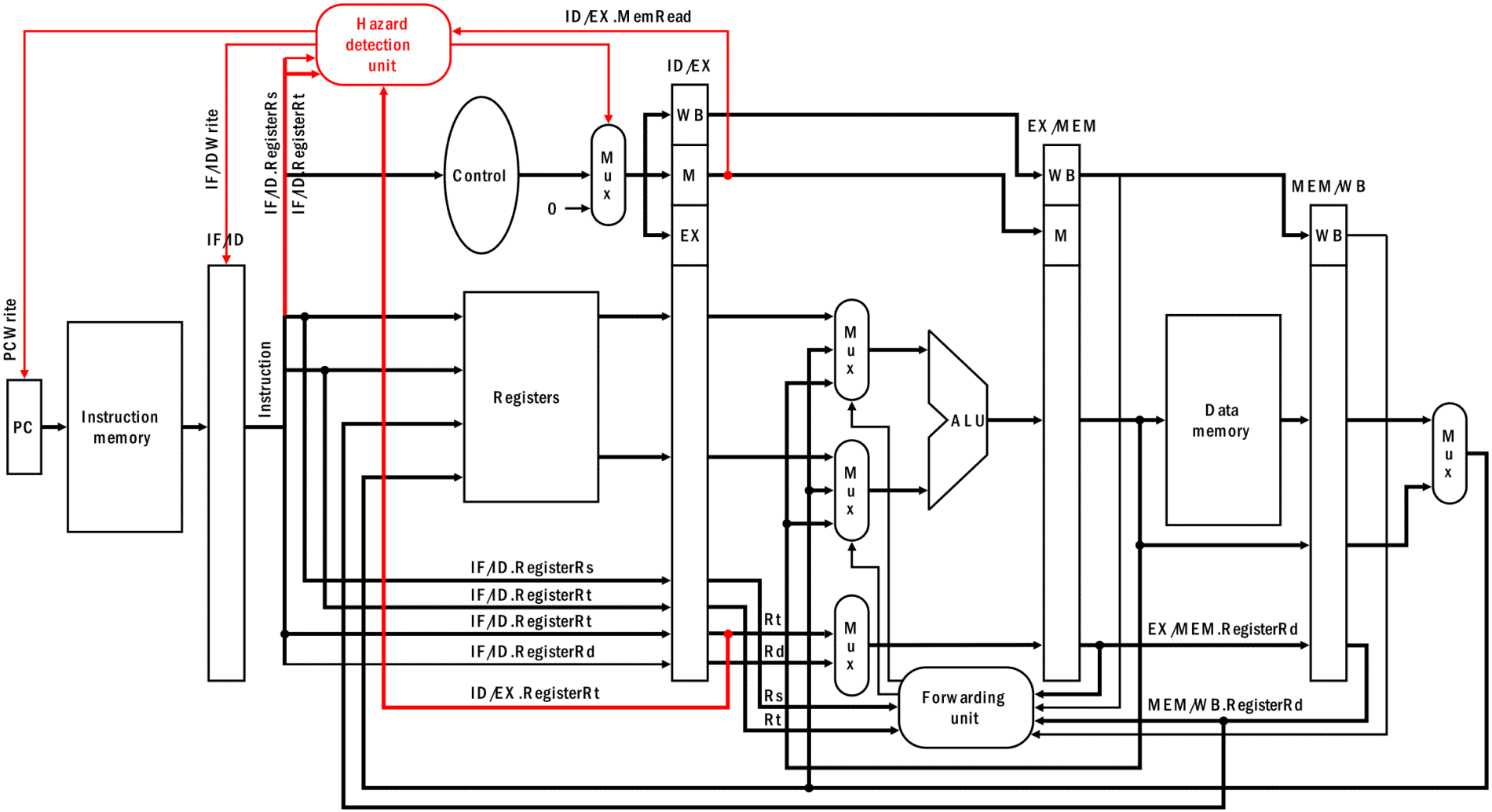
- Konieczne jest sprzętowe wstrzymanie kolejki



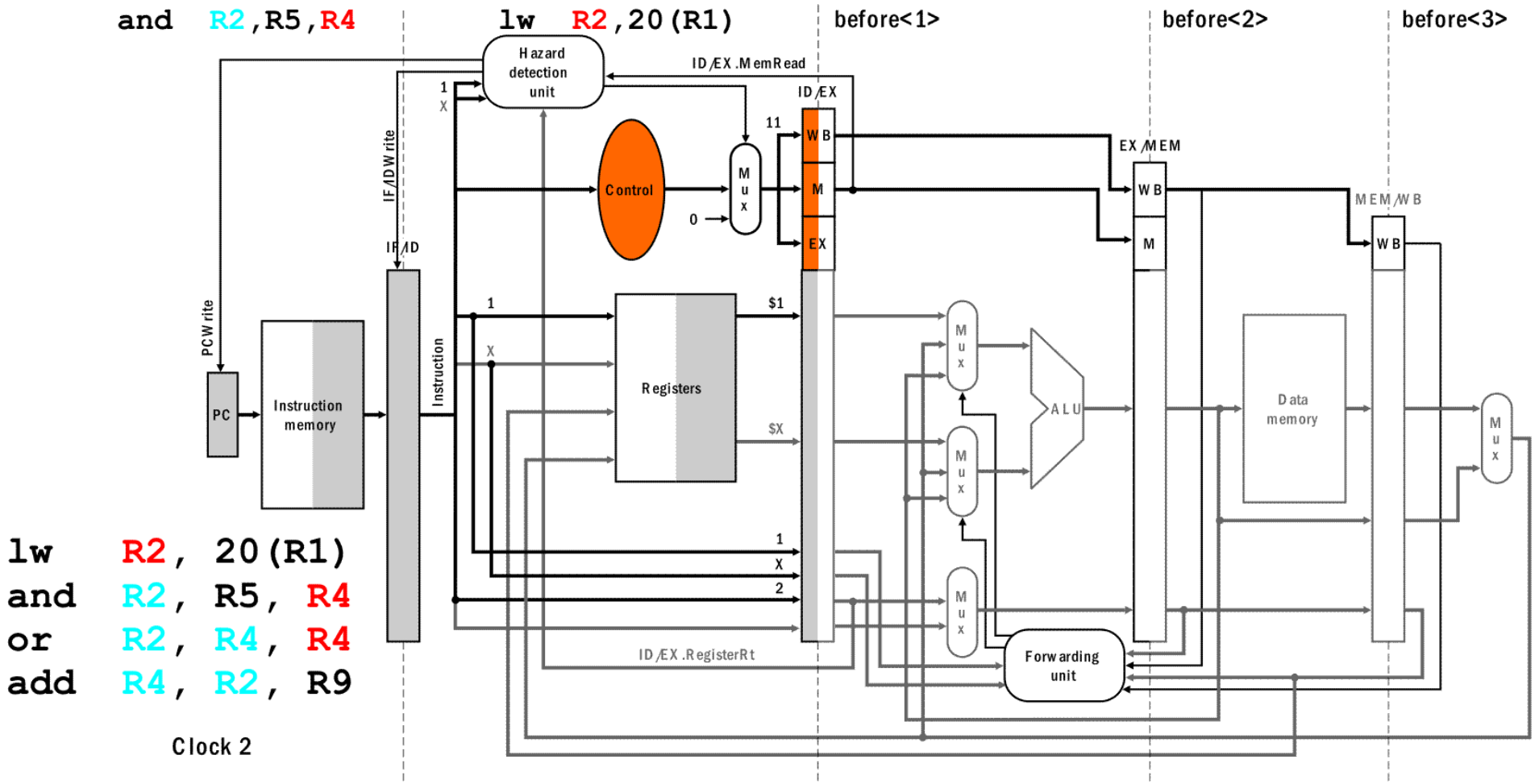
Sprzętowe wstrzymywanie kolejki

- Detekcja "trudnych" konfliktów danych możliwa jest tylko w fazie dekodowania (ID)
- Konieczny jest dodatkowy układ wykrywania konfliktów typu Read After Load (w omawianej architekturze)
- Układ detekcji konfliktów i wstrzymywania kolejki powinien być niezależny od sterowania i forwarding'u
- Blok logiki kombinacyjnej wykrywający:
  - instrukcję Load w fazie Ex i numer rejestru do zmiany (ID/Ex.MemRead oraz ID/Ex.Rt)
  - instrukcję R-type lub Store w razie ID i numery operandów (IF/ID.Rs oraz IF/ID.Rt)

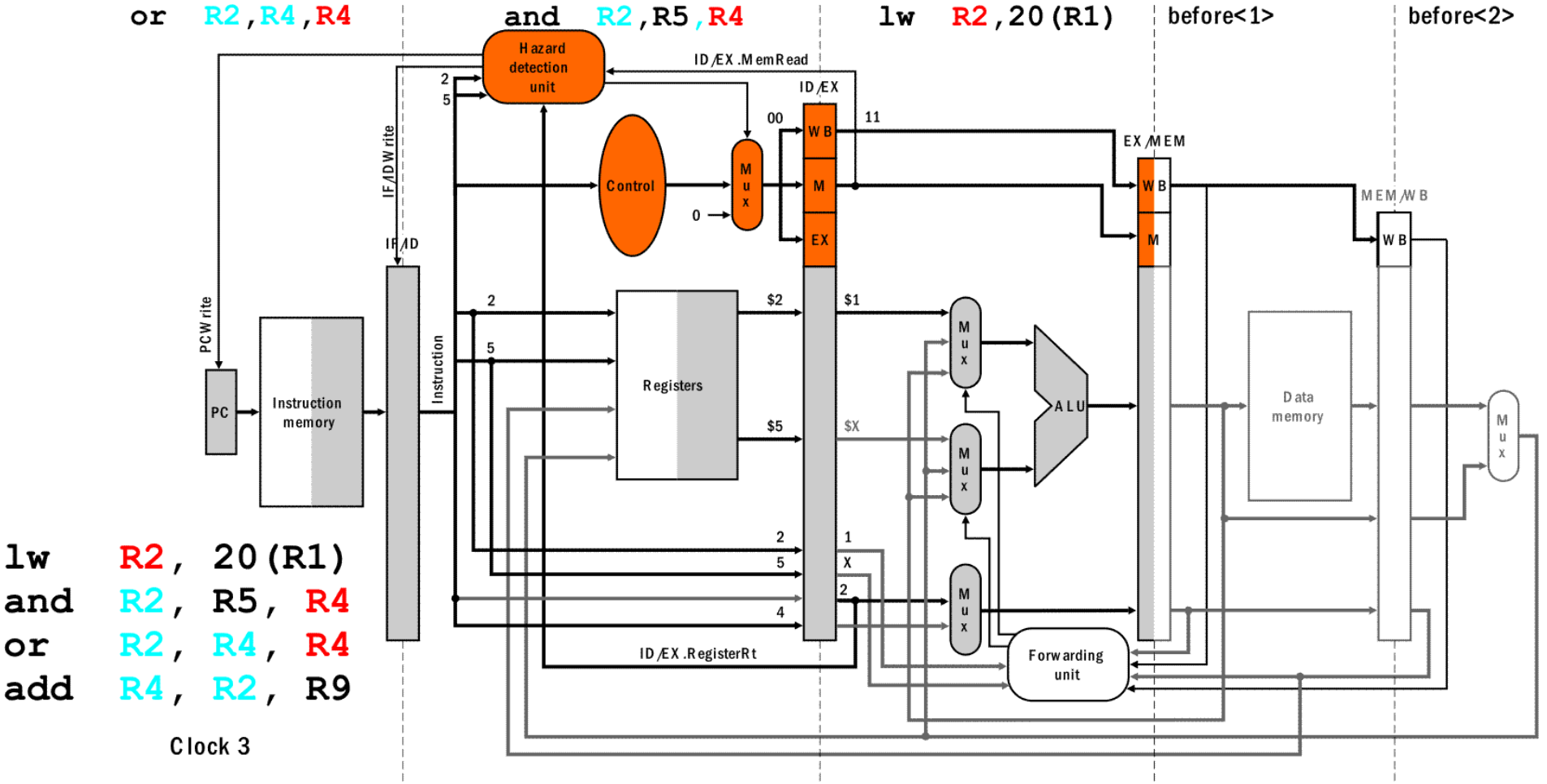
Sprzętowe wstrzymywanie kolejki



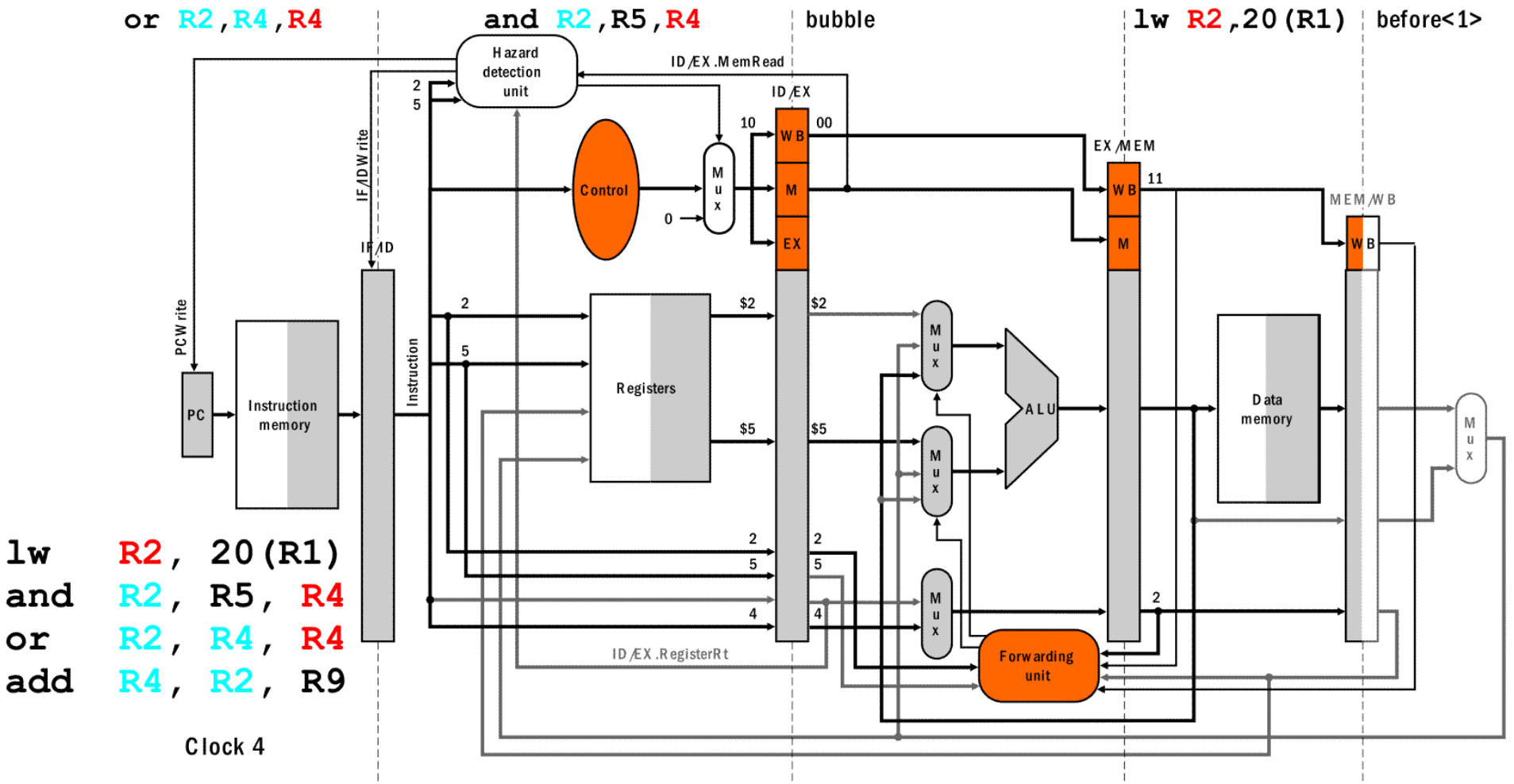
Wstrzymywanie kolejki - w działaniu (1)



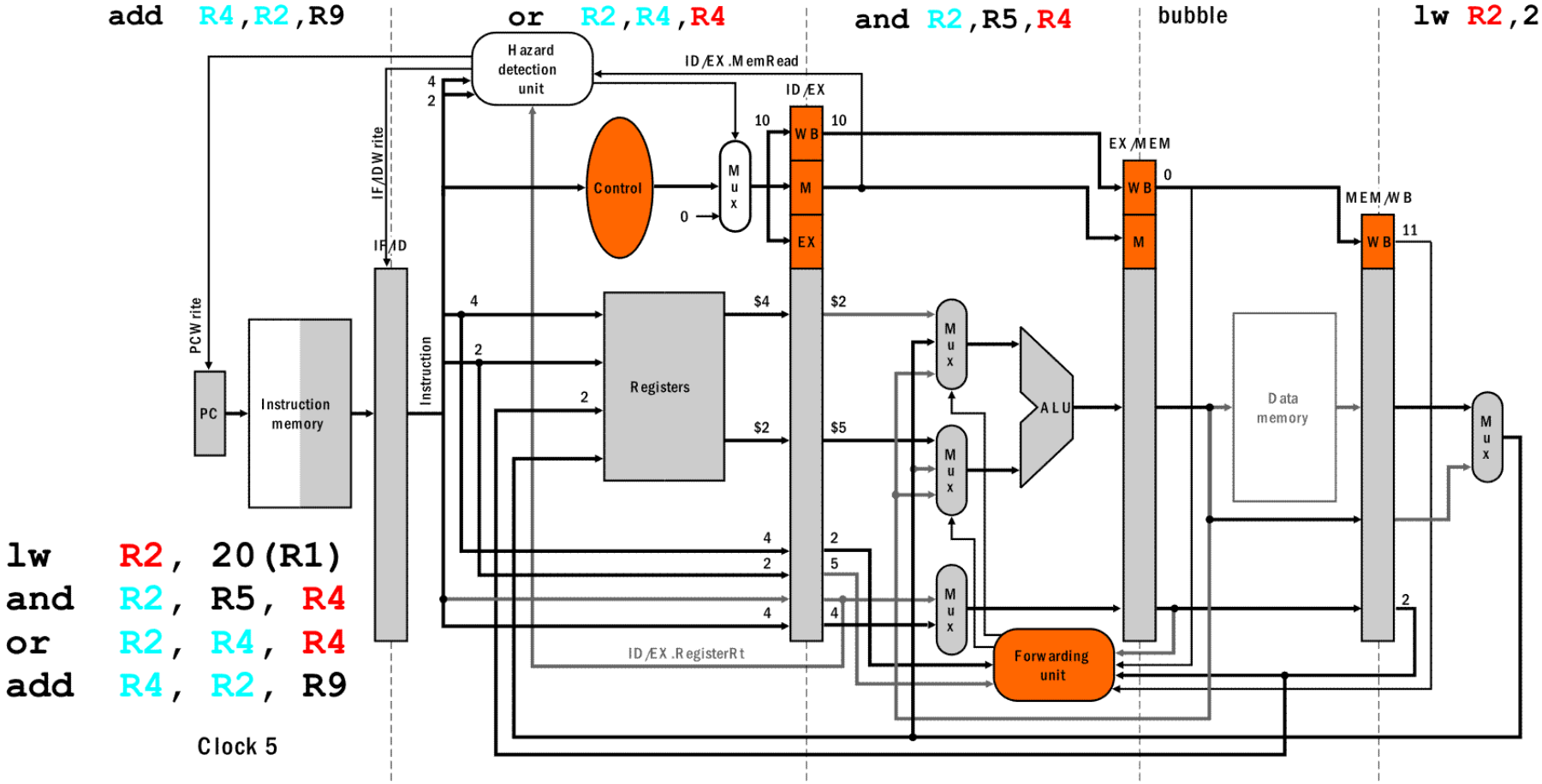
Wstrzymywanie kolejki - w działaniu (2)



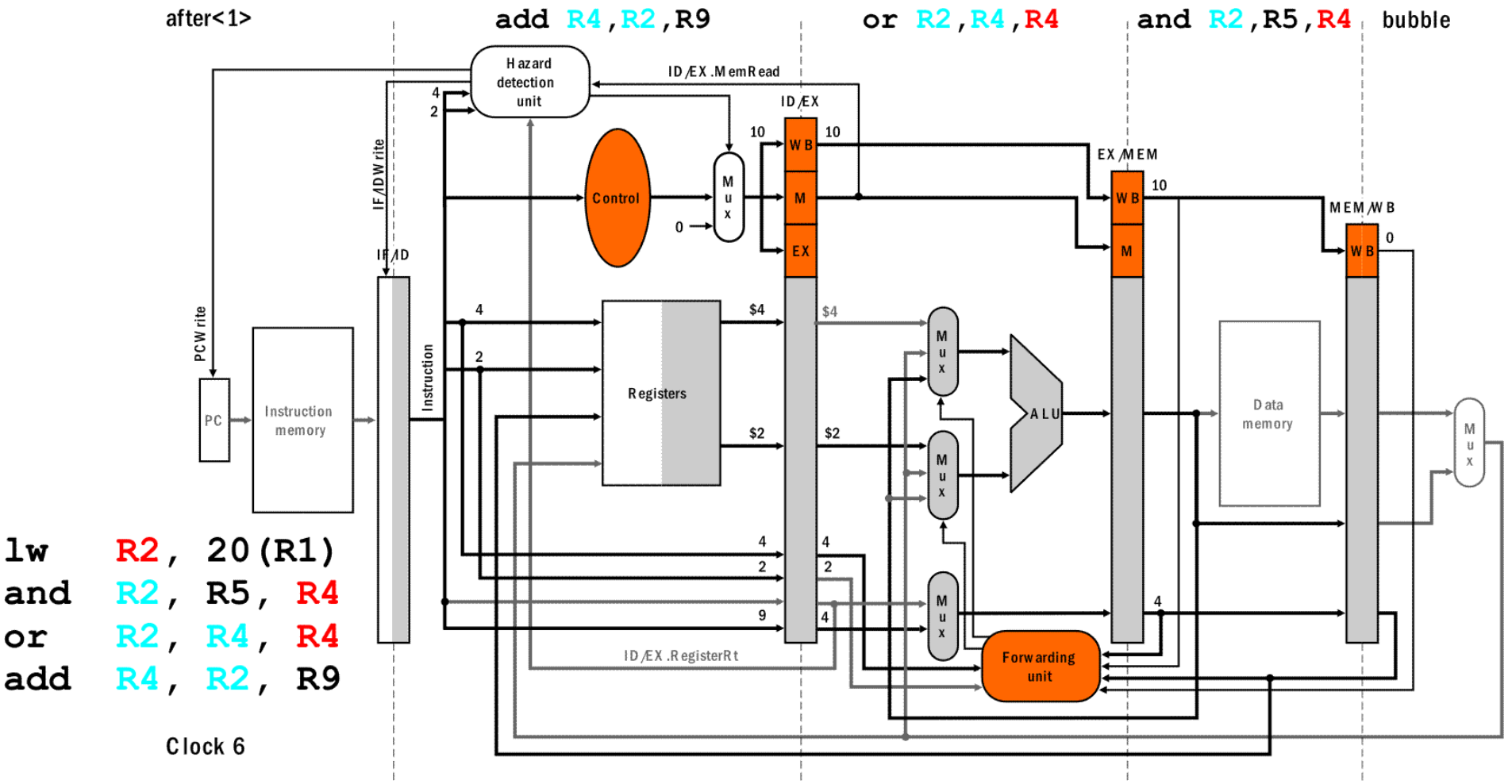
Wstrzymywanie kolejki - w działaniu (3)



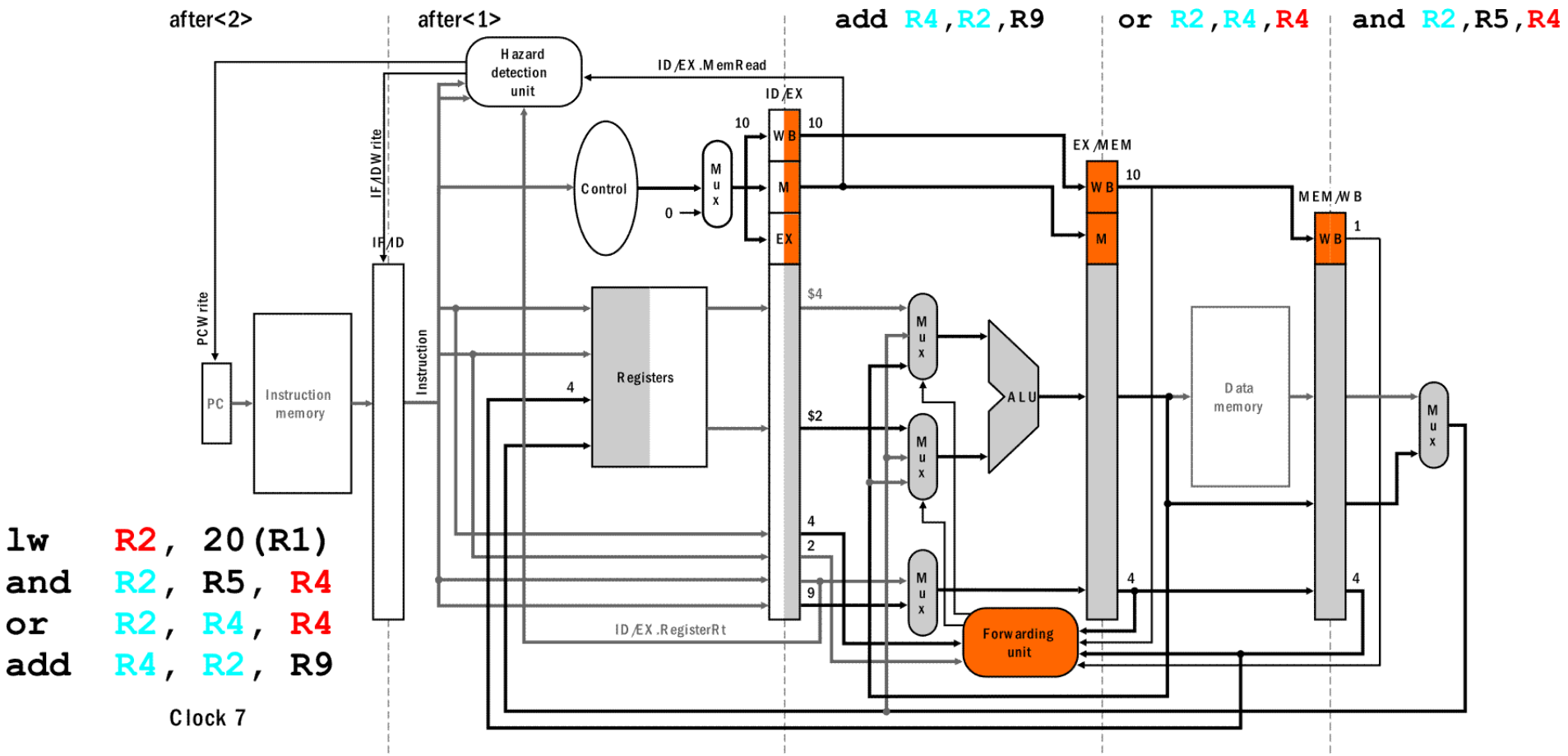
Wstrzymywanie kolejki - w działaniu (4)



Wstrzymywanie kolejki - w działaniu (5)



Wstrzymywanie kolejki - w działaniu (6)

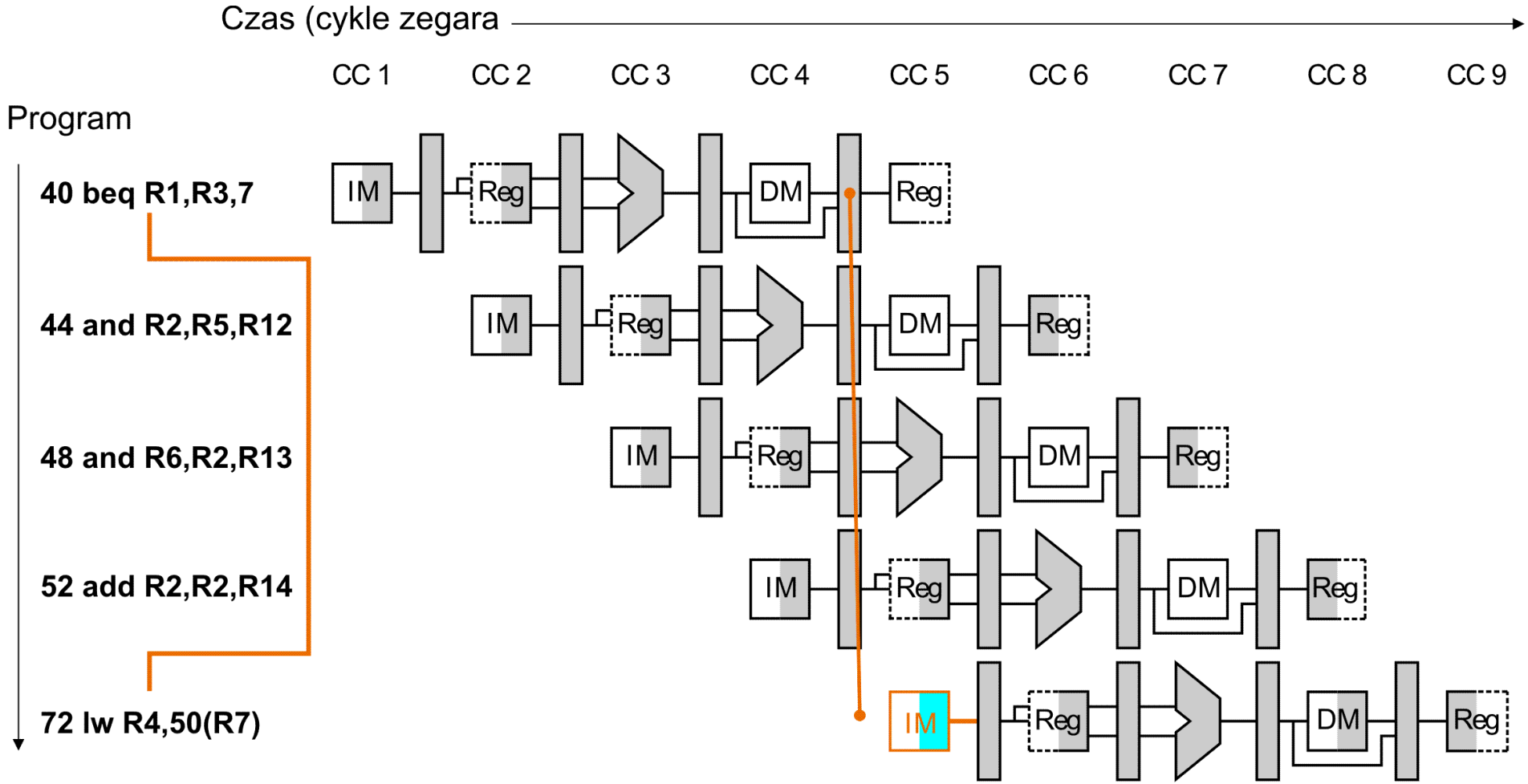


Konflikt sterowania

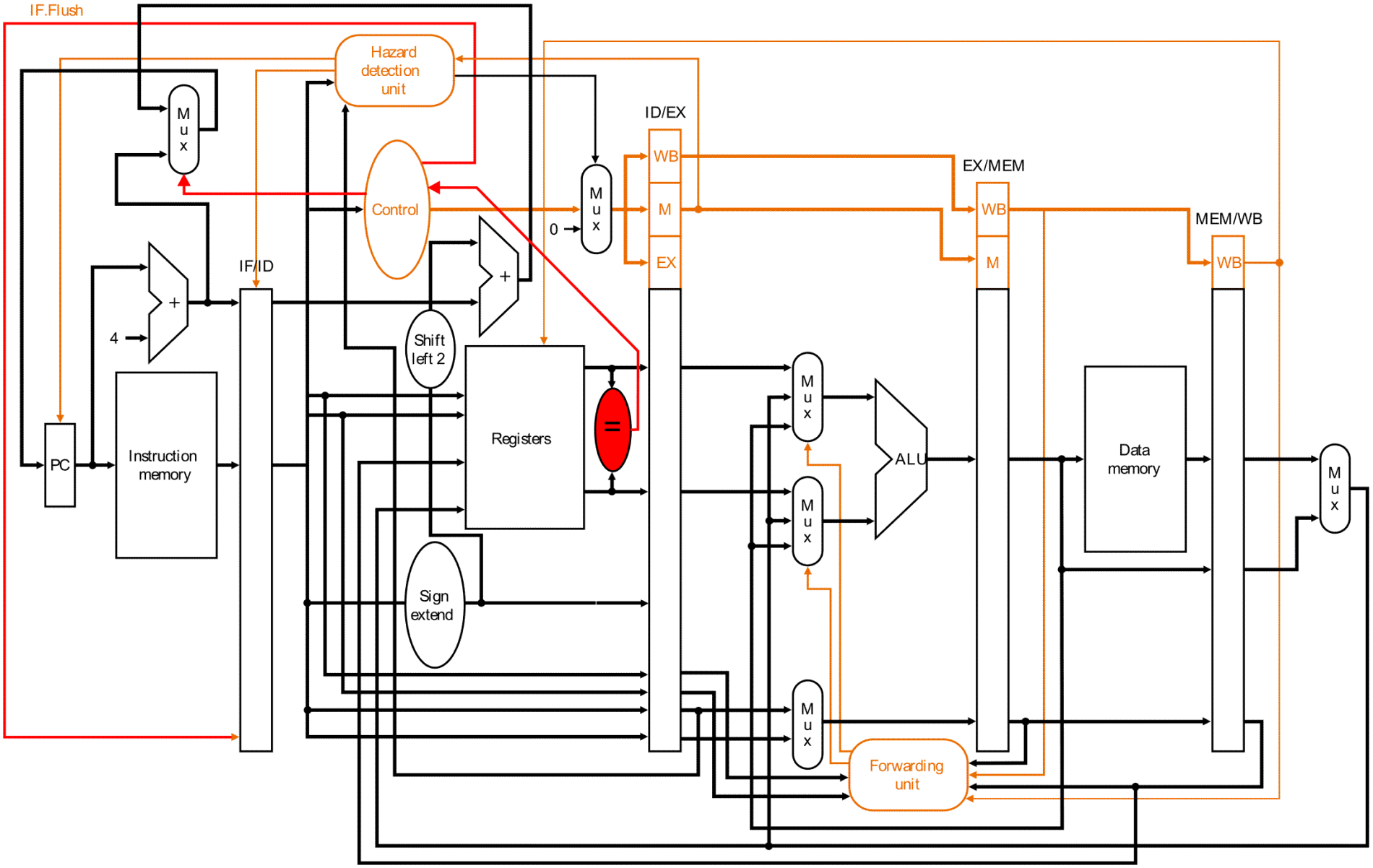
- Każdy skok powoduje zmianę kolejności wykonywania instrukcji – zaburza kolejkę i zmniejsza wydajność (CPI > 1)
- Skoki warunkowe wymagają dodatkowo policzenia warunku skoku, co opóźnia decyzję o skoku
- Wykonanie skoku wymaga (zwykle) unieważnienia kilku instrukcji, które już znajdują się w procesorze
- Efektywne metody redukcji konfliktów sterowania:
  - obliczanie warunku skoku w jak najwcześniejszej fazie (Early Branch Detection)
  - przewidywanie zachowania instrukcji skoku na podstawie jej ostatnich wykonań (Branch History Table)

Konflikt sterowania

- Wykorzystanie warunku skoku dopiero w 5 fazie skutkuje pobraniem 3 zbędnych instrukcji



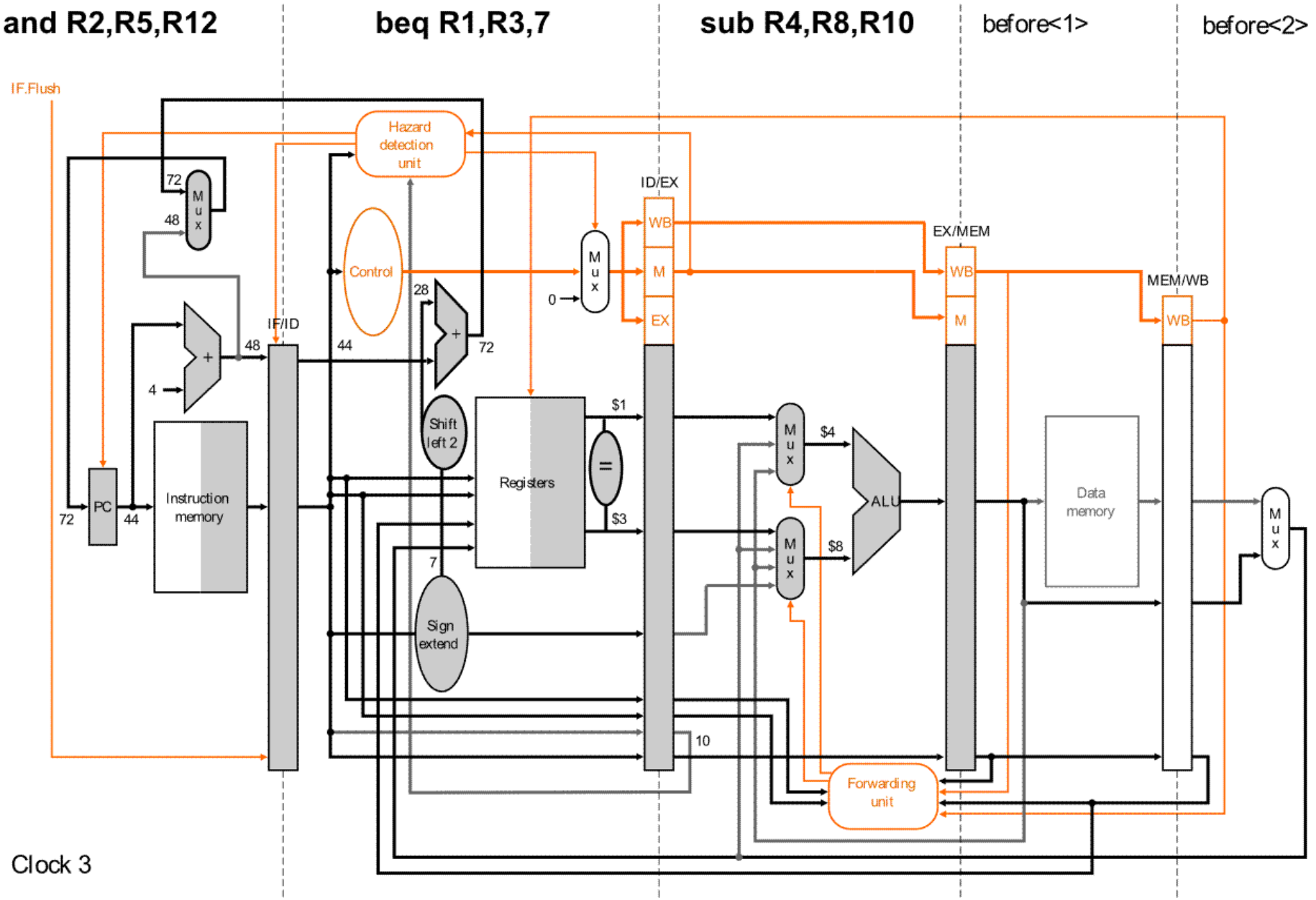
Wczesne wykrywanie skoków



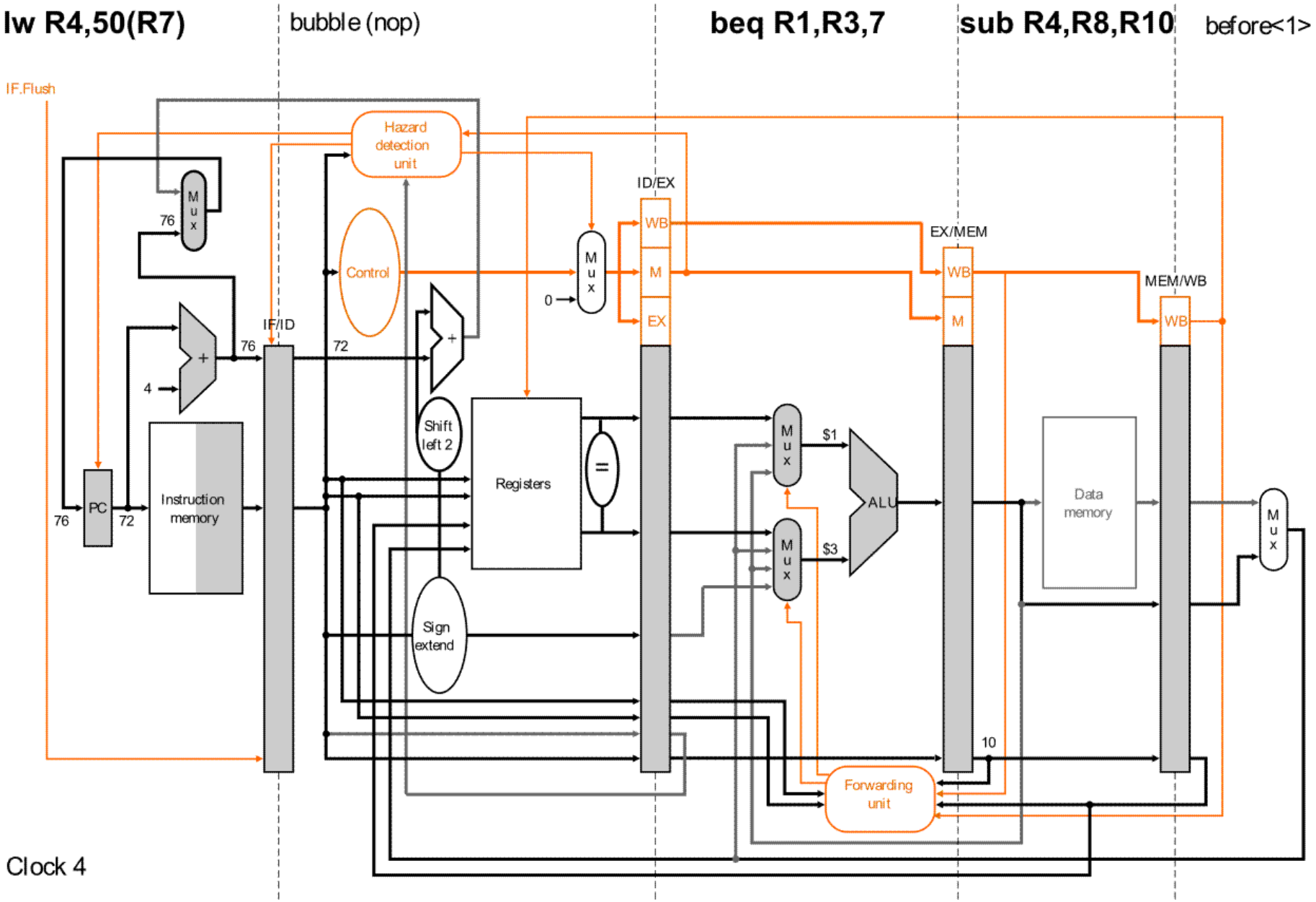
Wczesne wykrywanie skoków

- Przesunięcie obliczania warunku i adresu skoku do fazy ID powoduje, ze tylko jedna „niepewna” instrukcja jest pobrana przez procesor (w fazie IF)
- Obliczanie warunku skoku w fazie ID może dotyczyć tylko prostych (szybkich) obliczeń, np. porównanie
- Trzeba zapewnić dodatkowy sumator w fazie ID dedykowany do obliczania adresu skoku
- Unieważnienie instrukcji w fazie IF sprowadza się do zerowania rejestru IF/ID, co jest równoznaczne z zamianą instrukcji na NOP
  - zakładamy, że NOP ma *opcode* 0 i reszta też jest 0

Wczesne wykrywanie – w działaniu (1)

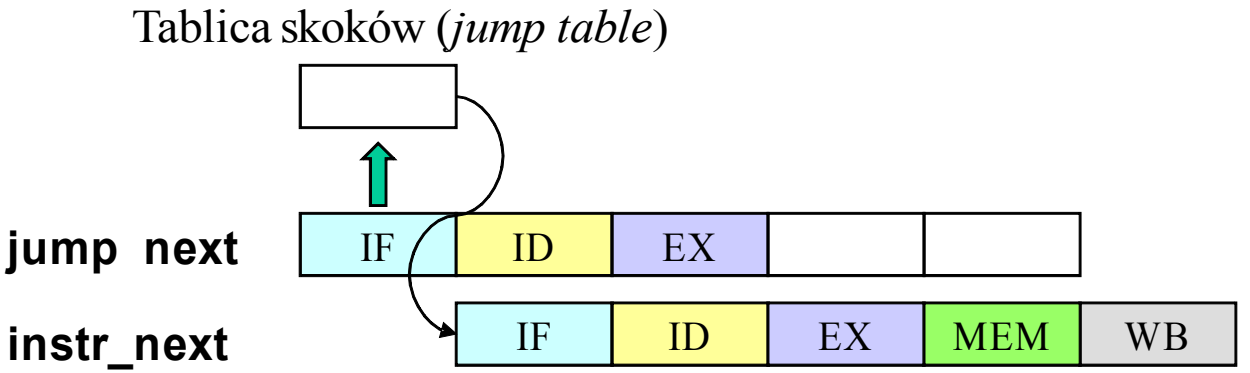


Wczesne wykrywanie – w działaniu (2)



Tablica historii skoków (BHT)

- W BHT zapamiętywane są adresy ostatnio wykonywanych instrukcji skoków i adresy, do których skoki zostały wykonane



  - Podczas pobieranie instrukcji skoku, adres następnej instrukcji jest odczytywany z tablicy skoków (o ile wcześniej już wystąpił)
  - Metoda pozwala zredukować nietrafione instrukcje. do przypadków zakończenia lub rozpoczęcia pętli w programach, czyli sytuacji stosunkowo rzadkich