Obsługa pamięci w architekturze Out of Order

Maciej Buszka

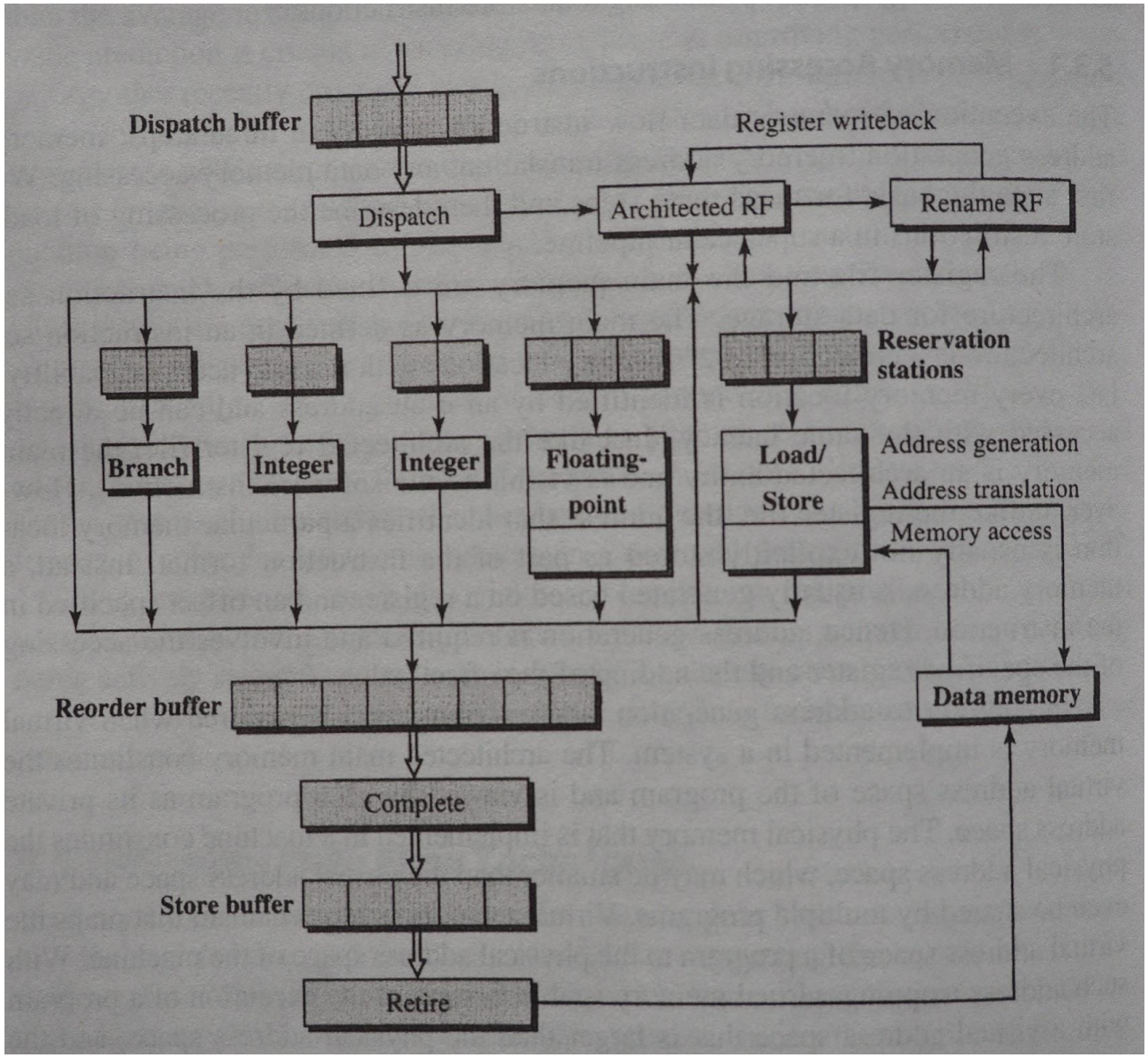
Zależności danych między instrukcjami

1. WAW - Write after Write
2. WAR - Write after Read
3. RAW - Read after Write

Jako, że w architekturze Out of Order stan pamięci musimy zmieniać w kolejności programu, tak też będą wykonywane zapisy. Zatem dwie pierwsze zależności zostaną zachowane niezależnie od przyjętych optymalizacji sprzętowych. W dalszej części musimy zadbać o zachowanie zależności RAW.

System jednoprocesorowy

Model podstawowy



Model podstawowy

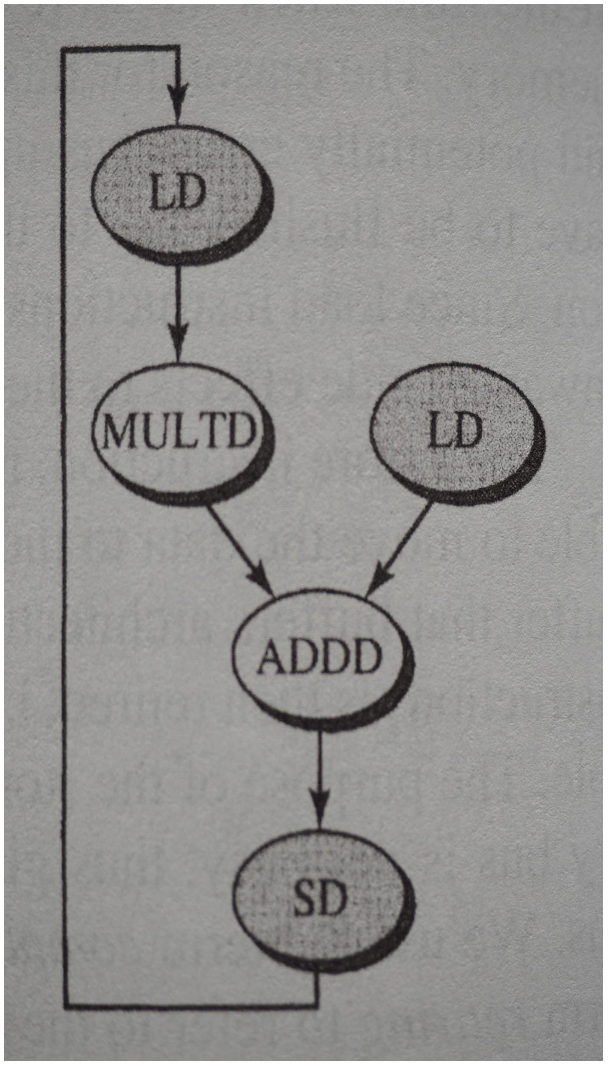
- Zakładamy standardowy model adresowania tj. adres w rejestrze + stałe przesunięcie
- Instrukcje ze stacji rezerwacji są zlecane w kolejności programu
  a. Odczyt jest zależny tylko od wartości z rejestru adresowego
  b. Zapis dodatkowo potrzebuje wartości z rejestru danych
- Dwa pierwsze etapy wykonania są wspólne dla zapisów i odczytów:
  a. Obliczenie adresu wirtualnego
  b. Translacja adresu (za pomocą TLB)
- W trzecim etapie:
  a. instrukcja odczytu wykonuje dostęp do cache
     - w przypadku cache miss konieczne jest wstrzymanie potoku load/store
  b. natomiast zapis od razu wędruje do bufora przestawiania
- Podczas zatwierdzania instrukcje zapisu są przenoszone do bufora zapisu skąd zostaną potem zapisane do cache
  a. W przypadku wyjątku musi on zostać opróżniony do pamięci

Model podstawowy

- Wszystkie operacje load/store są wykonywane w porządku programu
- Ale wykonanie pozostałych instrukcji pozostaje poza porządkiem, co pozwala na zamaskowanie np cache miss
- W oczywisty sposób model ten zachowuje wszystkie zależności danych
- Ale nie jest aż tak wydajny jak byśmy chcieli:
  - DIV R2, R7, R8
  - ST R1, R2
  - LD R3, R4 ; Ta linijka musi czekać na wynik DIV, mimo, że od niej nie zależy

Przykładowy program do optymalizacji

```
; Calculate Y = aX + B

 1.  LD      F0, a
 2.  ADDI R4, Rx, 512        ; last address
 3.  Loop:
 4.  LD      F2, 0(Rx)       ; load X(i)
 5.  MULTD   F2, F0, F2      ; a * X(i)
 6.  LD      F4, 0(Ry)       ; load Y(i)
 7.  ADDD    F4, F2, F4      ; a * X(i) + Y(i)
 8.  SD      0(Ry), F4
 9.  ADDI    Rx, Rx, 8       ; incr. X idx
10.  ADDI    Ry, Ry, 8       ; incr. Y idx
11.  SUB     R20, R4, Rx     ; check if end
12.  BNZ     Loop, R20
```



Przykładowy program do optymalizacji

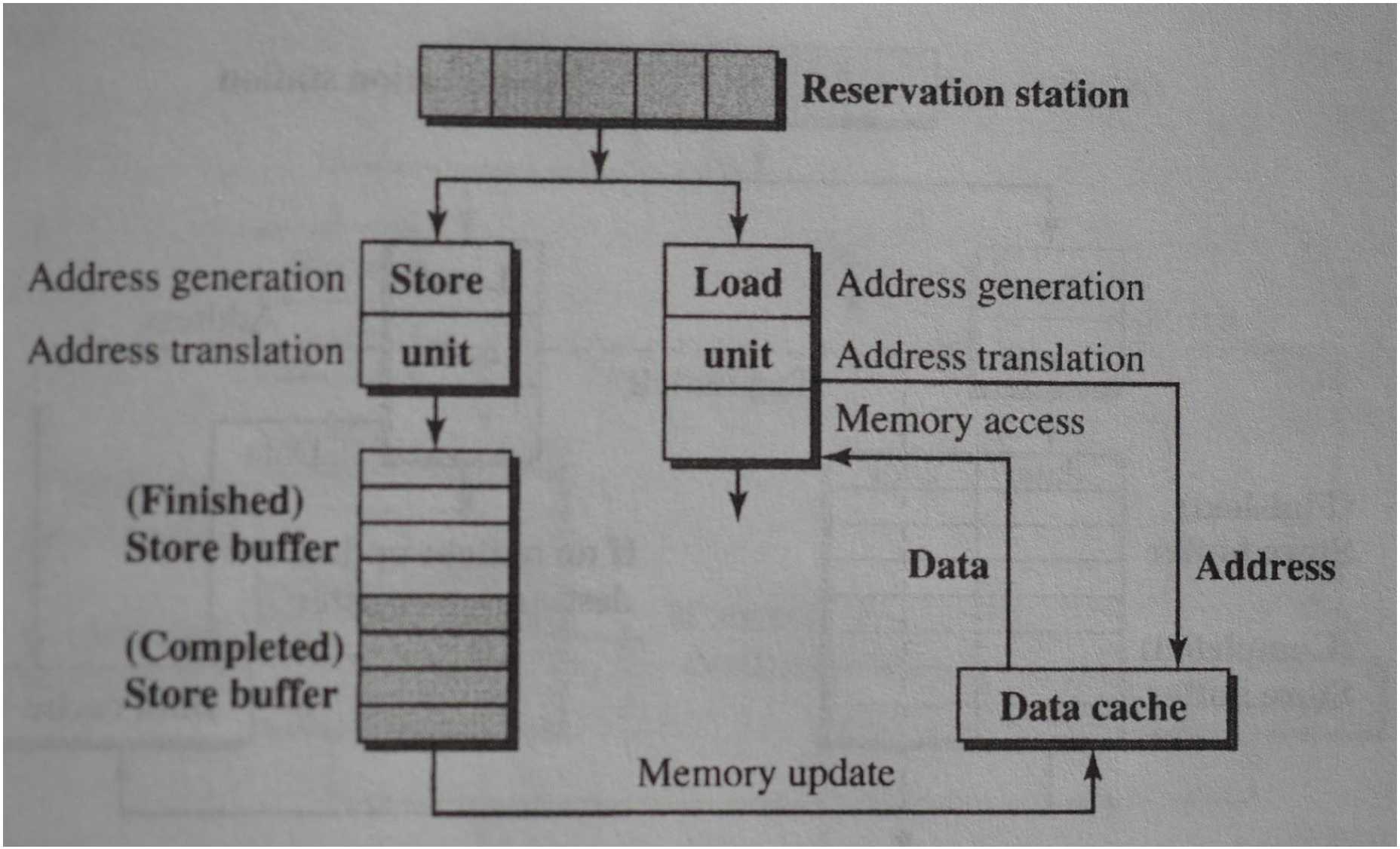
```
; Calculate Y = aX + B

 1.  LD      F0, a
 2.  ADDI    R4, Rx, 512     ; last address
 3.  Loop:
 4.  LD      F2, 0(Rx)       ; load X(i)
 5.  MULTD   F2, F0, F2      ; a * X(i)
 6.  LD      F4, 0(Ry)       ; load Y(i)
 7.  ADDD    F4, F2, F4      ; a * X(i) + Y(i)
 8.  SD      0(Ry), F4
 9.  ADDI    Rx, Rx, 8       ; incr. X idx
10.  ADDI    Ry, Ry, 8       ; incr. Y idx
11.  SUB     R20, R4, Rx     ; check if end
12.  BNZ     Loop, R20
```

- Widać, że każda iteracja tej pętli jest niezależna
- Jednakże jeśli utrzymywać pełną kolejność dostępów do pamięci, to każda iteracja musi czekać na zakończenie poprzedniej.
- Wystarczyłoby pozwolić odczytom z pamięci “przeskoczyć” nad zapisami

Motywacje

- Można zaobserwować, że odczyty zazwyczaj znajdują się na początku ciągu instrukcji wykonujących jakieś obliczenia
- Zatem krytyczne dla sprawnego wykonania jest jak najwcześniejsze ich zakończenie
- Odczyt można wykonać przed zapisami o ile nie ma konfliktu adresów (tzn nie naruszamy zależności RAW), a także przed pozostałymi odczytami
- Jeśli mamy konflikt, chcielibyśmy jak najszybciej przekazać wartość zapisaną do odczytu

Nowa organizacja jednostki load/store



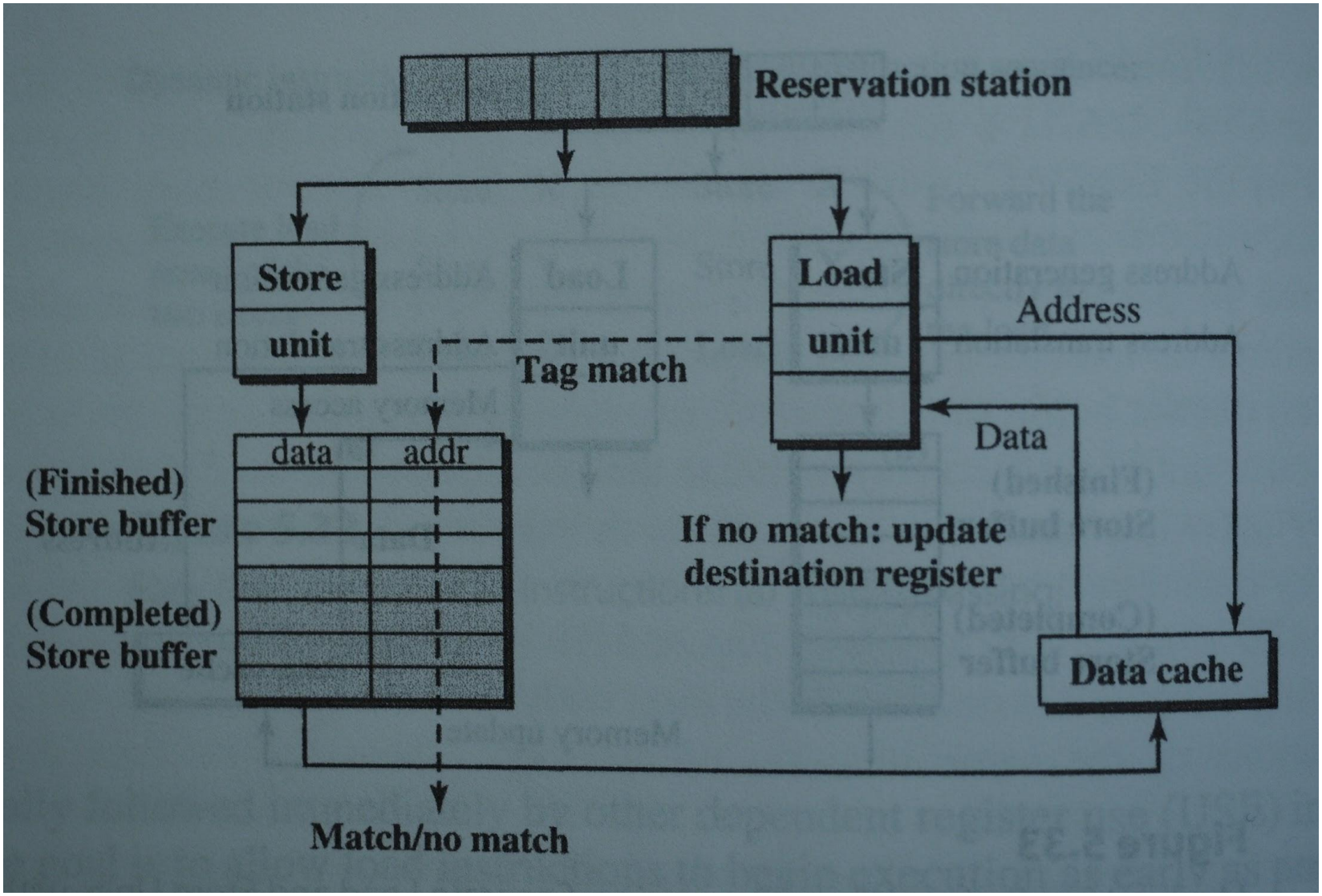
Nowa organizacja jednostki load/store

- Instrukcje są zlecane ze wspólnej stacji rezerwacji w kolejności programu (na razie)
- Mamy dwie osobne jednostki odczytów i zapisów
- Bufor zapisów jest kolejką FIFO podzieloną na dwa regiony
  - Zakończony - znajdują się tu zapisy z obliczonym już adresem, ale nadal mogą być spekulatywne
  - Zatwierdzony - zapisy zostały już zatwierdzone w buforze przestawiania, tutaj oczekują jeszcze na zapis do cache
  - Podobnie jak w podstawowym modelu, w przypadku wyjątku należy opróżnić zatwierdzony region do pamięci

Odczyty wyprzedzające (Load bypassing)

- Zakładając zlecanie instrukcji ze stacji rezerwacji w kolejności programu wszystkie wcześniejsze zapisy które nie zostały jeszcze zapisane do pamięci znajdować się będą w buforze zapisów
- Podczas obsługi instrukcji odczytu, w trzecim kroku potoku, równolegle z dostępem do pamięci należy przeszukać bufor zapisów celem znalezienia konfliktu adresów
- Jeżeli istnieje konflikt to instrukcja pozostanie w stacji rezerwacji, oczekując na wykonanie zapisu (widać tu potencjał na optymalizację)
- W przeciwnym przypadku, odczyt zakończy się sukcesem
- Można porównywać tylko część adresu fizycznego, lub dolne bity adresu wirtualnego (będzie wtedy więcej fałszywych konfliktów, ale żaden prawdziwy nie zostanie pominięty)

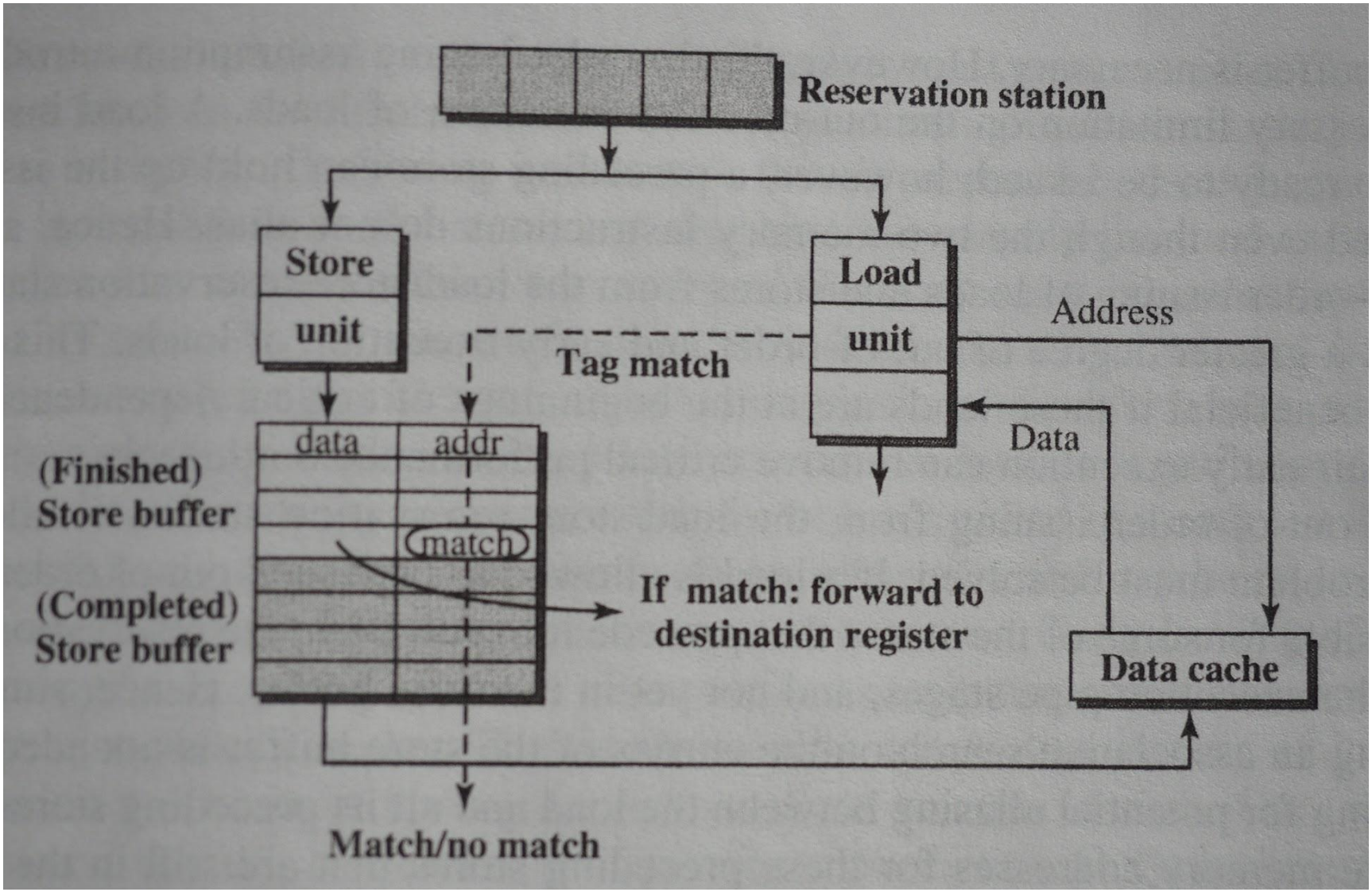
Odczyty wyprzedzające (Load bypassing)



Przekazywanie do odczytu (Load forwarding)

- W trakcie przeszukiwania bufora zapisów, w przypadku konfliktu, wartość ostatniego zapisu może zostać przekazana jako wynik odczytu
- Wymaga porównywania pełnych adresów fizycznych
- Sensowne przy częstych konfliktach adresów (np jeżeli ISA ma mało rejestrów i program musi zapisywać zmienne na stosie)

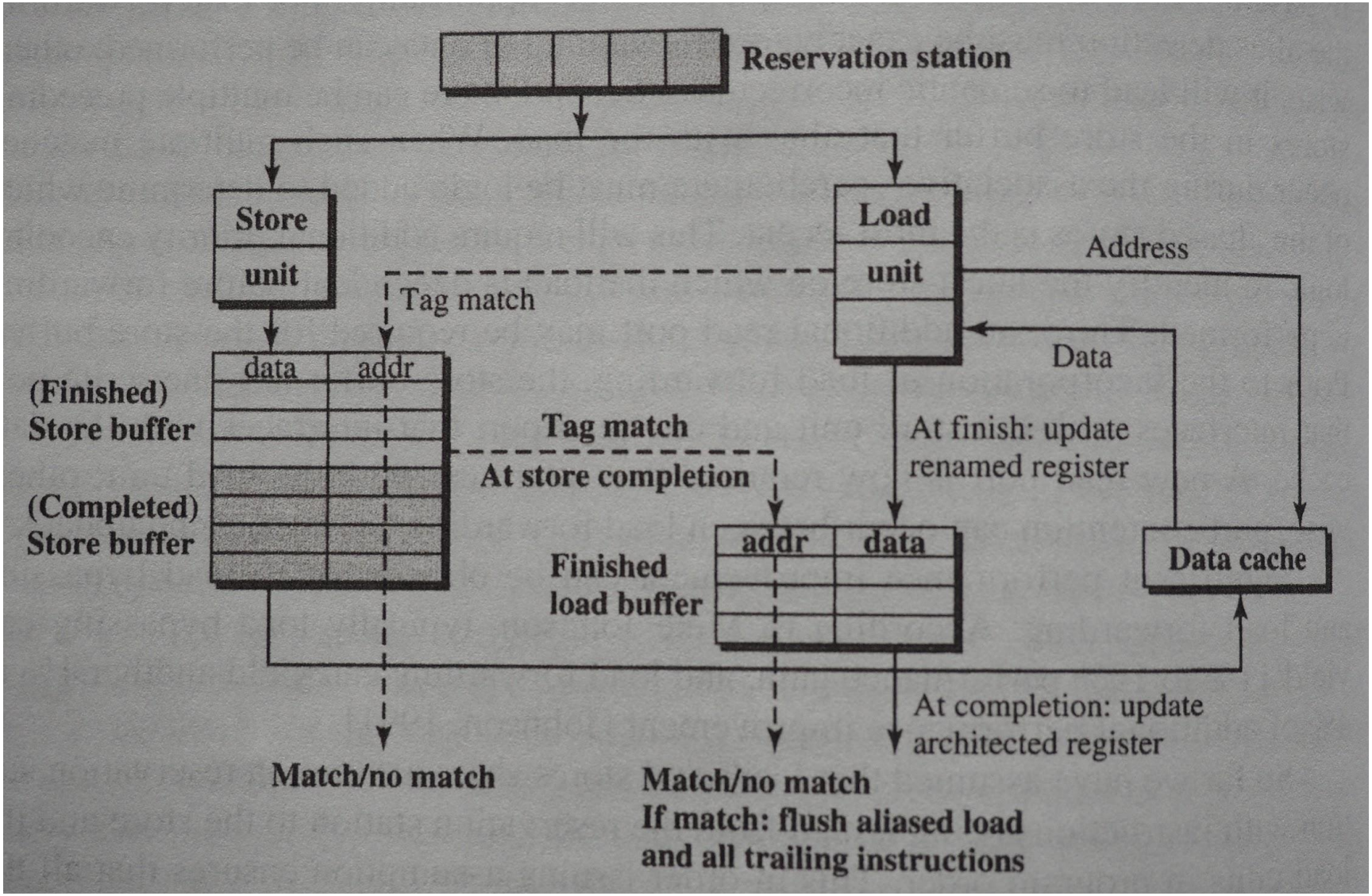
Przekazywanie do odczytu (Load forwarding)



Spekulatywne odczyty wyprzedzające

- Odczyty mogą być zlecane ze stacji rezerwacji poza porządkiem programu
- Dzięki temu odczyty nie muszą czekać na poprzedzające je niezależne zapisy
  - DIV R2, R7, R8
  - ST R1, R2
  - LD R3, Mem[R4] ; Ta linijka nie musi już czekać
- Jak poprzednio, przy odczycie równolegle z dostępem do cache, procesor sprawdza bufor zapisów w poszukiwaniu konfliktów
- Odczyty są zapisywane dodatkowo do bufora odczytów i oczekują na zatwierdzenie
- Po każdym zatwierdzeniu zapisu procesor sprawdza czy nie ma odczytów w konflikcie i w razie potrzeby unieważnia taki odczyt oraz kolejne instrukcje (podobnie jak w przypadku spekulacji podczas skoków)

Spekulatywne odczyty wyprzedzające



Przykłady wykonania

1. LD R1, Mem[A1]
2. ST Mem[A2], R2

- odczyt zostanie wykonany w porządku programu
- nie ma możliwości konfliktu

Przykłady wykonania

1. ST Mem[A2], R2
2. LD R1, Mem[A1]

- Odczyt zostanie wykonany spekulatywnie
- Jeżeli A1 != A2, po zatwierdzeniu instr. 1 i 2 odczyt przestanie być spekulatywny
- Jeżeli A1 == A2, to po zatwierdzeniu ST, procesor musi usunąć LD i następujące po nim instrukcje, po czym rozpocząć wykonanie ponownie (być może na nowo sprowadzając instrukcje)

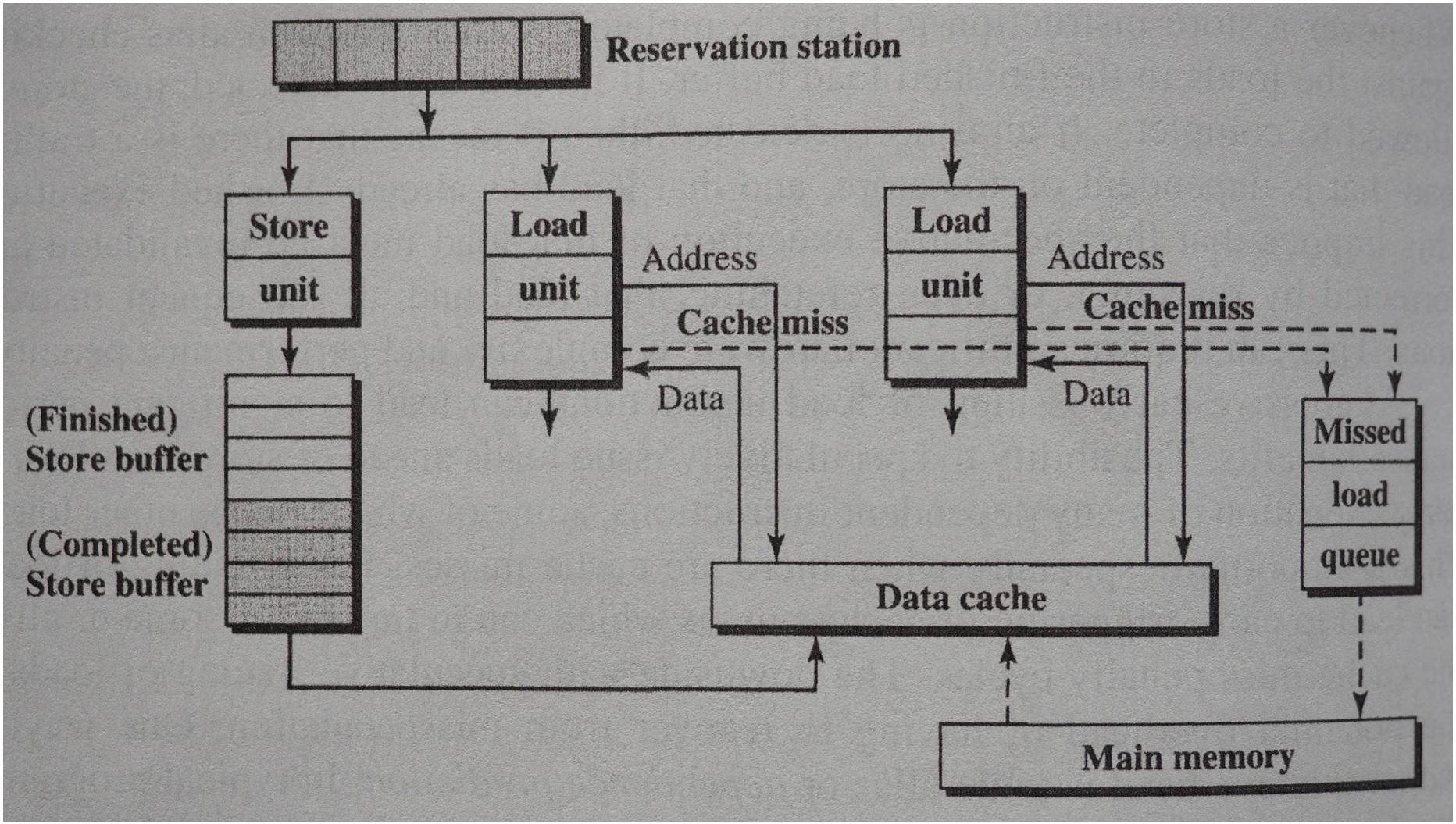
Dual-ported cache

- Rozszerzamy procesor o dodatkową jednostkę zapisów, natomiast cache ma dwa porty odczytu
- Taki cache można zaimplementować jako kilka rozłącznych zbiorów pamięci, jeżeli adresy do których wykonywany jest jednoczesny dostęp pochodzą z różnych zbiorów, mogą zostać wykonane równolegle, w przeciwnym przypadku trzeba je serializować
- Taka organizacja pozwala na sprawną obsługę nagłych serii dostępów do pamięci
- Studia empiryczne - wystarczy 8 zbiorów pamięci

Non-blocking cache

- Kolejnym rozszerzeniem cache jest dodanie kolejki oczekujących odczytów
- W przypadku, gdy adres do którego instrukcja się odwołuje nie znajduje się w cache, instrukcja ta jest wstawiana do kolejki, a cache miss jest obsługiwany przez kontroler cache
- Po zwolnieniu jednostki kolejna instrukcja może rozpocząć dostęp do pamięci, co pozwala na zamaskowanie cache miss
- W zależności od organizacji pamięci wiele instrukcji może na raz oczekiwać na obsługę cache miss
- Studia empiryczne - ok 15% zmniejszony koszt obsługi cache miss

Non-blocking, dual-ported cache



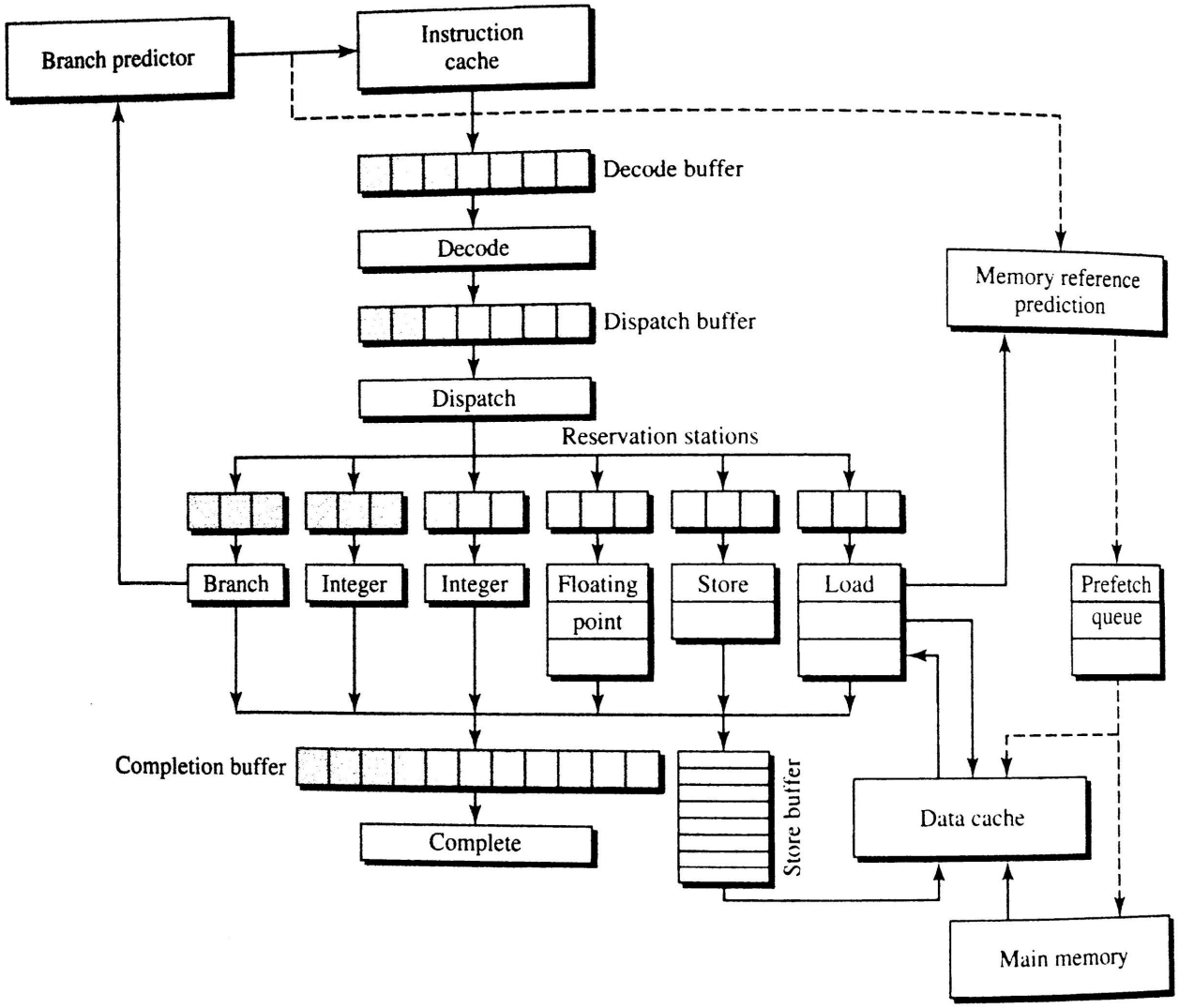
Prefetching cache

- Procesor rozszerzamy o dwie struktury:
  - tablica przewidywanych dostępów (memory reference prediction table)
  - kolejka wyprzedzająca (prefetch queue)
- W najprostszej implementacji wpis do tablicy przewidzianych dostępów zawiera:
  - adres grupy pobierania (fetch group)
  - adres poprzedniego dostępu do pamięci
  - krok (stride) - różnica między adresami poprzednich dwóch dostępów do pamięci tej instrukcji
- W fazie pobierania instrukcji, tablica jest przeszukiwana względem adresu instrukcji (być może spekulatywnego) i w przypadku trafienia, suma poprzedniego adresu dostępu i kroku jest dodawana do kolejki
- Taka organizacja pozwala na efektywne przewidywanie adresów przy regularnych przejściach po tablicy
- Adresy z kolejki powodują dostęp do cache i potencjalne wymuszenie cache miss, zanim właściwa instrukcja wykona dostęp do pamięci

Prefetching cache

- Taka implementacja pozwala znacznie szybciej wymusić konieczne cache miss
- Jednakże:
  - Efektywność jest ograniczona przez efektywność predyktora skoków
  - Zbyt wczesne wywołanie dostępu do cache może usunąć użyteczną linię
  - Zbyt późne dostępy nie dadzą istotnej poprawy
  - Jeżeli dany dostęp znajdował się na spekulatywnej ścieżce wykonania, procesor może zapchać cache niepotrzebnymi danymi
- Zarazem algorytm przewidywania dostępów musi generować poprawne adresy:
  - Co w przypadku struktur danych rozrzuconych po pamięci i połączonych wskaźnikami nie jest proste
  - Twórcy bibliotek i kompilatorów muszą wiedzieć o istnieniu takich optymalizacji

Prefetching cache



Dalsze optymalizacje

- Przewidywanie zależności
  - W poprzednim modelu, procesor zawsze przewidywał brak zależności, co sprawdza się o ile rzeczywiście nie występują one zbyt często
  - Można dostawić predyktor, mówiący czy odczyt zależy od jakiegoś wcześniejszego zapisu, albo znajdujący konkretny zapis
- Przewidywanie adresu odczytu
  - Procesor jest rozszerzony o tablicę indeksowaną adresem instrukcji
  - Jeżeli podczas pobierania instrukcji, jej adres będzie znajdował się w tablicy, to znaczy, że jest ona odczytem
  - W kolejnym cyklu można odczytać z tablicy adres i wykonać spekulatywny odczyt
- Przewidywanie wartości odczytu
  - Jak poprzednio, tylko tablica jest rozszerzona o wartość odczytu

System wieloprocesorowy

Modele spójności

- Rozważymy tutaj dwa interesujące modele:
- Spójność sekwencyjna
  - Mówimy, że system jest spójny sekwencyjnie, jeśli wynik wykonania jest taki sam jak wynik wykonania jakiegoś sekwencyjnego przeplotu wszystkich instrukcji, a instrukcje każdego wątku są wykonywane w kolejności programu
  - Instrukcje pojedynczego wątku są obserwowalne w kolejności programu
  - Istnieje jeden globalny porządek wykonania wszystkich instrukcji
- Spójność x86-TSO
  - Pozwalamy na dwa odejścia względem spójności sekwencyjnej
    - Wątek może zaobserwować własne zapisy przed pozostałymi wątkami
    - Wątek może wykonywać odczyty przed zapisami poprzedzającymi je w porządku programu (zamiana W→R)
- Spójność zrelaksowana
  - Pozwalamy na wszelkie zamiany W→R, W→W, R→RW
  - Wątek może zaobserwować wcześniej zarówno własny jak i cudzy zapis

Implementacja dla spójności sekwencyjnej

- Nie można efektywnie zastosować bufora zapisów, jako że zapisy muszą być jednocześnie widoczne dla wszystkich procesorów
- Dalej można spekulatywnie wykonywać odczyty o ile procesor wie, że dana linia cache nie została zmodyfikowana przez inny procesor
  - Trzeba dodatkowo nasłuchiwać zewnętrznych zmian linii cache
  - Taki mechanizm zapewniają protokoły spójności cache
  - Jeśli procesor nie dostanie żadnej informacji dotyczącej danej linii, to znaczy, że spekulacja się powiodła i można tę instrukcję bezpiecznie zatwierdzić

Przykład

Program 1, A1 = 0

1. ST A1, #1
2. LD R1, A2

Program 2, A2 = 0

1. ST A2, #1
2. LD R2, A1

Wykonanie:

1. LD1 R1, A2
2. ST2 A2, #1 ; Ta instrukcja spowoduje unieważnienie spekulatywnego LD1
3. ST1 A1, #1
4. LD2 R2, A1

Implementacja dla spójności osłabionej (x86-TSO)

- Przyjmujemy model w którym zapisy są dodawane do bufora, a odczyty mogą wyprzedzać zapisy
- Utrzymujemy bufor odczytów w którym zapisana jest informacja, czy z pozostałych procesorów przyszło unieważnienie
- Gdy procesor natrafi na barierę pamięci, musi zapisać wszystkie oczekujące zapisy do pamięci, oraz oznaczyć spekulatywne zapisy w buforze odczytów
- Jeżeli podczas zatwierdzania odczyt ma oznaczoną barierę oraz unieważnienie musi zostać wyrzucony i wykonany na nowo

Przykład

Program 1, A1 = 0

1. ST A1, #1
2. LD R1, A2

Program 2, A2 = 0

1. ST A2, #1
2. LD R2, A1

Wykonanie:

1. LD1 R1, A2
2. LD2 R2, A1
3. ST2 A2, #1 ; Ta instrukcja oznaczy spekulatywny LD1, ale bez bariery będzie ona mogła
4. ST1 A1, #1 ; zostać zatwierdzona

Wynik R1 == R2 == 0

Przykład

Program 1, A1 = 0

1. ST A1, #1
2. MB
3. LD R1, A2

Program 2, A2 = 0

1. ST A2, #1
2. MB
3. LD R2, A1

Wykonanie:

1. LD1 R1, A2
2. LD2 R2, A1
3. ST1 A1, #1
4. MB1 ; Ta instrukcja oznaczy spekulatywny LD1
5. ST2 A2, #1 ; Ta instrukcja unieważnia spekulatywny LD1, zatem przy zatwierdzaniu
6. MB2 ; zostanie ona wyrzucona i wykonana na nowo

Wynik R1 == 1, R2 == 0

PowerPC 620

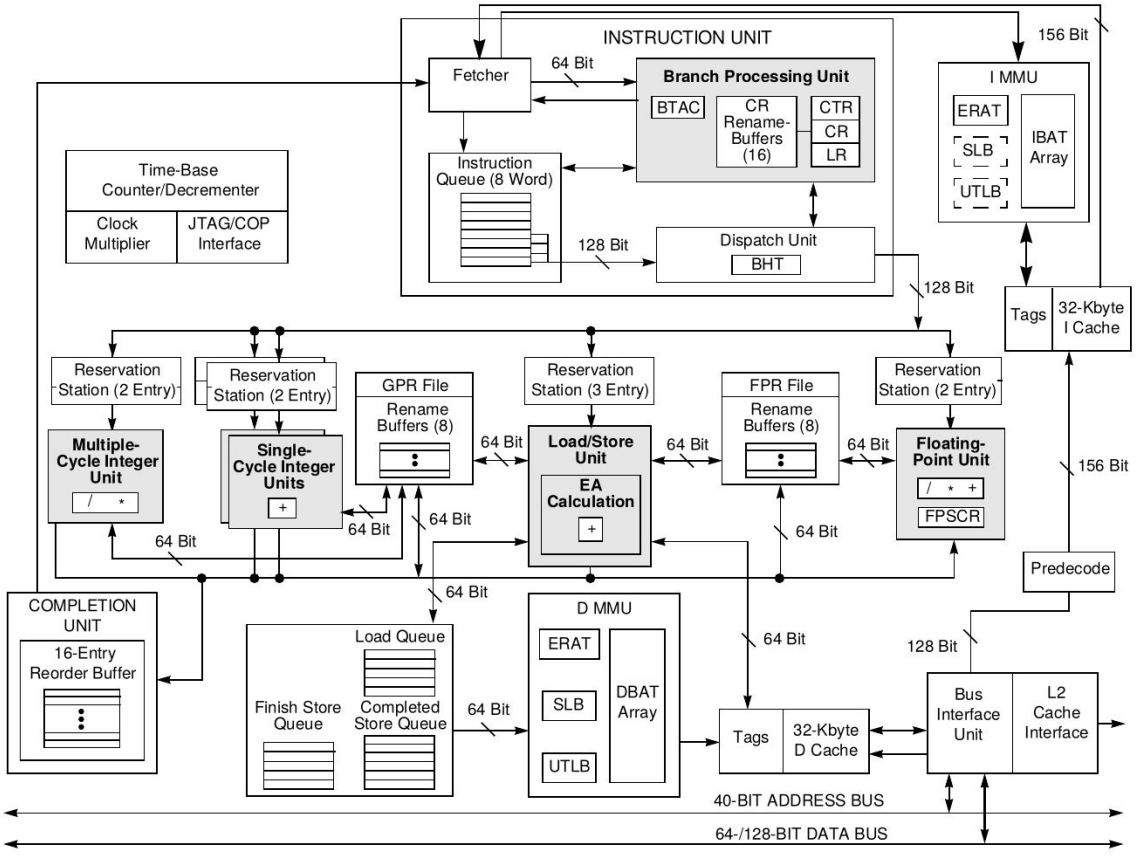


Figure 1. Block Diagram

PowerPC 620

- Implementuje zrelaksowany model spójności
- dual-ported, non-blocking cache (maks 1 zapis i jeden odczyt na raz)
  - Do trzech odczytów oczekujących na obsługę cache miss
- Zapisy wykonywane in-order
  - Wykorzystuje kolejkę zapisów
  - Przy zatwierdzaniu sprawdzane są konflikty z wyprzedzającymi odczytami
- Odczyty mogą wyprzedzić zapisy
  - przy zatwierdzaniu, w przypadku konfliktu instrukcja jest pobrana na nowo (wraz z instrukcjami po niej)
- Różne instrukcje synchronizacji
  - niestety nie jest opisana ich dokładna implementacja

PowerPC 620

Table 6.10
Cache effect data*

Cache Effects	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*hydro2d*	*tomcatv*
Loads/stores with cache hit	94.17	99.57	99.92	99.74	99.99	94.58	96.24
Loads that bypass a missed load	8.45	0.53	0.11	0.14	0.01	4.82	5.45
Loads that bypass a pending store	58.85	21.05	27.17	48.49	98.33	58.26	43.23
Load that aliased with a pending store	0.00	0.31	0.77	2.59	0.27	0.21	0.29
Average number of pending stores per cycle	1.96	0.83	0.97	2.11	1.30	1.01	1.38

*Values given are percentages, except for the average number of pending stores per cycle.

Bibliografia

1. J. P. Shen, M. H. Lipasti: “Modern Processor Design”
2. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, M. O. Myreen: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors” - http://www.cl.cam.ac.uk/users/pes20/weakmemory
3. W. Hu, P. Xia: “Out-of-order execution in sequentially consistent shared-memory systems: Theory and experiments”
4. PowerPC 620™ RISC Microprocessor Technical Summary