Architektura komputerów - ULTIMATE

SonMati
Doxus

19 czerwca 2015

Teoria

1 Historia rozwoju komputerów

1. Liczydło
2. Pascalina - maszyna licząca Pascala (dodawanie i odejmowanie)
3. Maszyna mnożąca Leibniza (dodawanie, odejmowanie, mnożenie, dzielenie, pierwiastek kwadratowy
4. Maszyna różnicowa - Charles Babbage, obliczanie wartości matematycznych do tablic
5. Maszyna analityczna - Charles Babvage, programowalna za pomocą kart perforowanych
6. Elektryczna maszyna sortująca i tabelaryzująca Holleritha 1890
7. Kalkulator elektromechaniczny Mark I, tablicowanie funkcji, całkowanie numeryczne, rozwiązywanie równań różniczkowych, rozwiązywanie układów równań liniowych, analiza harmoniczna, obliczenia statystyczne
8. Maszyny liczące Z1: pamięć mechaniczna, zmiennoprzecinkowa reprezentacja liczb, binarna jednostka zmiennoprzecinkowa
9. Z3: Pierwsza maszyna w pełni automatyczna, kompletna w sensie Turinga, pamięć przekaźnikowa
10. Colossus i Colossus 2
11. ENIAC
12. EDVAC - J. von Neumann (wtedy utworzył swoją architekturę)
13. UNIVAC I (pierwszy udany komputer komercyjny)
14. IBM 701, potem 709
15. po 1955 zaczyna się zastosowanie tranzystorów w komputerach (komputery II generacji)
16. po 1965 komputery III generacji z układami scalonymi
17. od 1971 komputery IV generacji - z układami scalonymi wielkiej skali inegracji VLSI

2 Architektura CISC

2.1 Znaczenie

Complex Instruction Set Computers

2.2 Przyczyny rozwoju architektury CISC

- Drogie, małe i wolne pamięci komputerów
- Rozwój wielu rodzin komputerów
- Duża popularność mikroprogramowalnych układów sterujących (prostych w rozbudowie)
- Dążenie do uproszczenia kompilatorów: im więcej będzie rozkazów maszynowych odpowiadających instrukcjom języków wyższego poziomu tym lepiej; model obliczeń pamięć – pamięć.

2.3 Cechy architektury CISC

- Duża liczba rozkazów (z czego te najbardziej zaawansowane i tak nie były używane)
- Duża ilość trybów adresowania (związane z modelem obliczeń)
- Duży rozrzut cech rozkazów w zakresie:
  - złożoności
  - długości (szczególnie to - nawet kilkanaście bajtów)
  - czasów wykonania
- Model obliczeń pamięć - pamięć
- Niewiele rejestrów - były droższe niż komórki pamięci i przy przełączaniu kontekstu obawiano się wzrostu czasu przełączania kontekstu (chowanie rejestrów na stos i odwrotnie)
- Przerost struktury sprzętowej przy mało efektywnym wykorzystaniu list rozkazów

CIEKAWOSTKA: Przeanalizowano jakieś tam programy i w procesorze VAX 20% najbardziej złożonych rozkazów odpowiadało za 60% kodu, stanowiąc przy tym ok 0.2% wywołań.
W procesorze MC68020 71% rozkazów nie zostało nawet użytych w badanych programach

3 Architektura RISC

3.1 Znaczenie

Reduced Instruction Set Computers.

3.2 Przyczyny rozwoju

- Poszukiwanie optymalnej listy rozkazów
- Chęć wykonania mikroprocesora o funkcjach pełnego ówczesnego procesora

3.3 Pierwszy procesor RISC

Procesor RISC I (1980), D. Patterson (Berkeley University)
Założenia projektowe:

- Wykonanie jednego rozkazu w jednym cyklu maszynowym
- Stały rozmiar rozkazów – uproszczenie metod adresacji
- Model obliczeń rejestr – rejestr: komunikacja z pamięcią operacyjną tylko za pomocą rozkazów LOAD i STORE.
- Wsparcie poprzez architekturę języków wysokiego poziomu.

Efekty realizacji fizycznej:

- 44 420 tranzystorów (ówczesne procesory CISC zawierały ok. 100 000 tranzystorów)
- lista rozkazów = 32 rozkazy
- dwustopniowy potok – strata tylko 6% cykli zegara, zamiast 20% (w związku z realizacją skoków)

3.4 Cechy architektury RISC

1. Stała długość i prosty rozkaz formatu
2. Nieduża liczba trybów adresowania
3. Niezbyt obszerna lista rozkazów
4. Model obliczeń rejestr-rejestr - dostęp do pamięci operacyjnej tylko w rozkazach LOAD i STORE
5. Duży zbiór rejestrów uniwersalnych
6. Układ sterowania – logika szyta
7. Intensywne wykorzystanie przetwarzania potokowego
8. Kompilatory o dużych możliwościach optymalizacji potoku rozkazów

3.5 Format rozkazu procesora RISC I

7	1	5	5	1	13
OPCODE	SCC	DEST	SRC1	IMM	SRC2

- OPCODE–kod rozkazu
- SCC – ustawianie (lub nie) kodów warunków
- DEST – nr rejestru wynikowego
- SRC1 – nr rejestru zawierającego pierwszy argument
- IMM – wskaźnik natychmiastowego trybu adresowania
- SRC2 – drugi argument lub nr rejestru (na 5 bitach)

3.6 Realizacja wybranych rozkazów

3.6.1 Rozkazy arytmetyczne

- **Tryb rejestrowy**: (IMM=0) R[DEST] ← R[SRC1] op R[SRC2]
- **Tryb natychmiastowy:** (IMM=1) R[DEST] ← R[SRC1] op SRC2

3.6.2 Rozkazy komunikujące się z pamięcią

- **LOAD** R[DEST] ← M[AE]P
- **STORE** M[AE] ← R[DEST]

3.6.3 Adres efektywny

- **Tryb z przesunięciem** AE = R[SRC1] + SRC2 = RX + S2
- **Inny zapis powyższego** AE = RX + S2
- **Tryb absolutny** AE = R0 + S2 = S2 (R0 ≡ 0)
- **Tryb rejestrowy pośredni** AE = RX + 0 = RX

Tryb absolutny oraz tryb rejestrowy pośredni są przypadkami szczególnymi.

Tabela 1: Rejestry

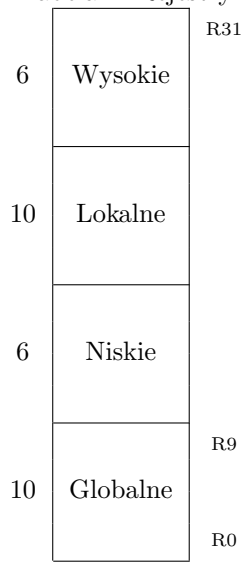
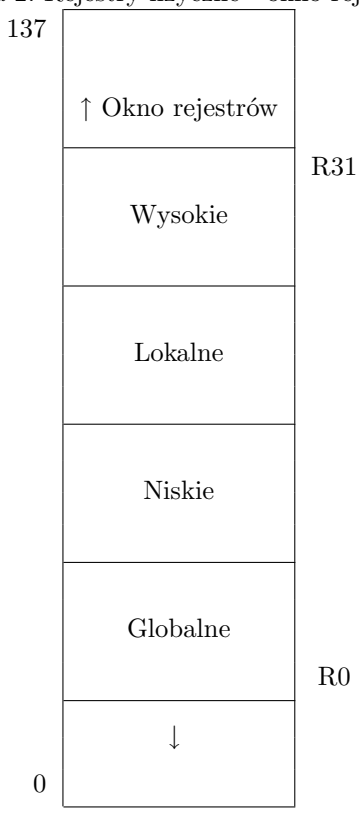


Tabela 2: Rejestry fizyczne - okno rejestrów



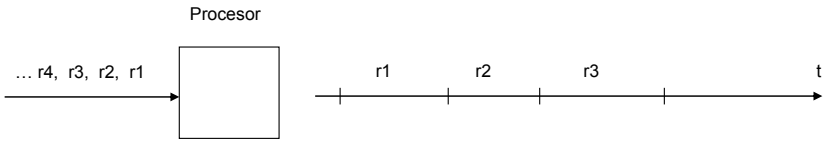
3.7 Logiczna organizacja rejestrów procesora RISC I

3.8 Okno rejestrów

4 Mechanizmy potokowe

4.1 Realizacja rozkazów w procesorze niepotokowym

Rozkazy wykonywane są liniowo w czasie - jeden po drugim, w takiej kolejności w jakiej przyjdą do procesora.

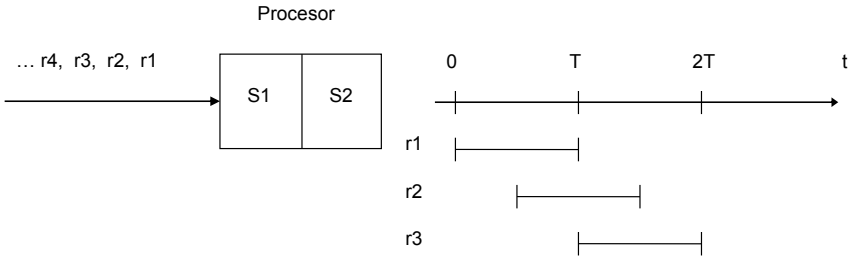


4.2 Potokowe wykonanie rozkazówdla prostej organizacji cyklu rozkazowego

Prosty podział procesora na moduły:

- S1 - pobranie rozkazu
- S2 - wykonanie rozkazu

Zakładając, że czas pracy obu modułów jest równy, wówczas 3 rozkazy mogą zostać wykonane w 2 okresach. 1 T - pobranie i wykonanie rozkazu. W momencie gdy pierwszy rozkaz zostanie pobrany, w chwili 0.5 T S1 może pobrać kolejny.



4.3 Podział cyklu rozkazowego na większą liczbę faz

Na przykładzie cyklu rozkazowego komputera Amdahl 470:

1. Pobranie rozkazu
2. Dekodowanie rozkazu
3. Obliczenie adresu efektywnego
4. Pobranie argumentów
5. Wykonanie operacji
6. Zapis wyniku

Zasada działania jest dokładnie taka sama jak w przypadku podziału na dwie fazy. Załóżmy, że jeden rozkaz wykonuje się w 7iu taktach zegarowych. 1 T = 7 F. Wówczas w momencie gdy rozkaz numer 1 znajduje się w 5tym takcie wykonania rozkaz numer 5 może zostać pobrany.

Rozkazy	Fazy zegarowe						
	1	2	3	4	5	6	7
S1	r1	r2	r3	r4	r5	r6	r7
S2		r1	r2	r3	r4	r5	r6
S3			r1	r2	r3	r4	r5
S4				r1	r2	r3	r4
S5					r1	r2	r3
S6						r1	r2

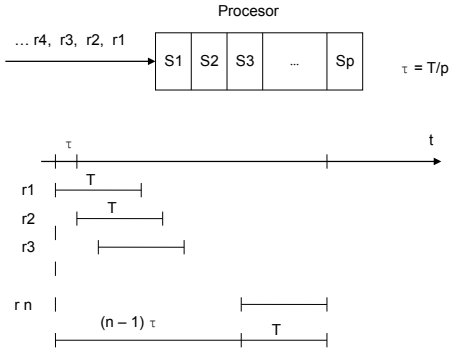
Tabela 3: Realizacja ciągu rozkazów w wielostopniowym procesorze potokowym.

4.4 Analiza czasowa potokowej realizacji ciągu rozkazów

Założenia:

- **P** - liczba faz
- **T** - okres
- $\frac{T}{P} = \tau$ - czas wykonania pojedynczej fazy

$(n-1) \times \tau$ - czas rozpoczęcia wykonywania n-tego rozkazu.



4.5 Przyspieszenie dla potokowego wykonania rozkazów

- Czas wykonywania rozkazu w procesorze niepotokowym (dla n rozkazów)

$$t = n \times T$$

- Czas wykonywania rozkazu w procesorze potokowym dla idealnego przypadku, gdy $\tau = \frac{T}{P}$

$$t = (n-1) \times \tau + T = (n - 1 + P) \times \frac{T}{P}$$

- Przyspieszenie jest stosunkiem czasu wykonywania rozkazów dla procesora niepotokowego do czasu dla procesora potokowego.

$$\lim_{n \to \infty} \frac{n \times T}{(n - 1 + P) \times \frac{T}{P}} = P$$

Maksymalne przyspieszenie (dla modelu idealnego) jest równe ilości faz.

4.6 Problemy z potokową realizacją rozkazów

Problemem związanym z realizacją potokową jest **zjawisko hazardu**.

- **Hazard sterowania** – problemy z potokową realizacją skoków i rozgałęzień.
- **Hazard danych** – zależności między argumentami kolejnych rozkazów
- **Hazard zasobów** – konflikt w dostępie do rejestrów lub do pamięci

4.7 Rozwiązanie problemu hazardu sterowania

- Skoki opóźnione
- Przewidywanie rozgałęzień

4.8 Skoki opóźnione

4.8.1 Założenia

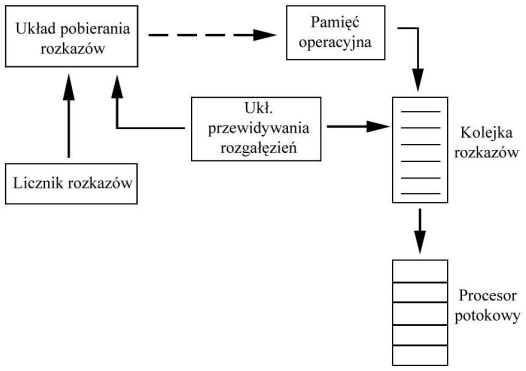
- Rozkaz następny po skoku jest zawsze całkowicie wykonywany
- To znaczy, że efekt skoku jest opóźniony o jeden rozkaz

4.8.2 Działanie

Zmienia kod programu w trakcie kompilacji, jeśli widzi taka potrzebę. Sprowadza się to do dwóch możliwości:

- Modyfikacja programu - dodanie rozkazu NOP po instrukcji skoku JMP
- Optymalizacja programu - zmiany kolejności wykonywania rozkazów

4.9 Przewidywanie rozgałęzień



4.9.1 Strategie

1. **Statyczne**
   - przewidywanie, że rozgałęzienie (skok warunkowy) zawsze nastąpi
   - przewidywanie, że rozgałęzienie nigdy nie nastąpi
   - podejmowanie decyzji na podstawie kodu rozkazu rozgałęzienia (specjalny bit ustawiany przez kompilator)
2. **Inne**
   - przewidywanie, że skok wstecz względem licznika rozkazów zawsze nastąpi
   - przewidywanie, że skok do przodu względem licznika rozkazów nigdy nie nastąpi
3. **Dynamiczne**
   - Tablica historii rozgałęzień.

4.9.2 Tablica historii rozgałęzień

Składa się z:

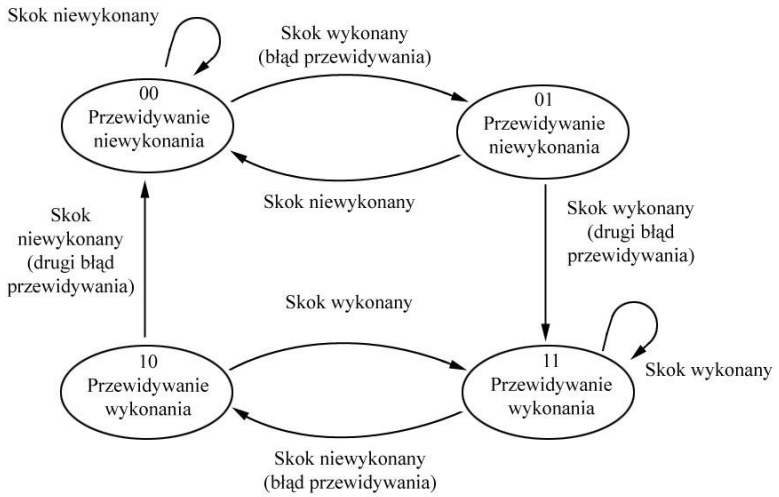
- Bit ważności
- Adres rozkazu rozgałęzienia
- Bity historii
- Adres docelowy rozgałęzienia (opcja)

Operacje wykonywane na tablicy historii rozgałęzień

- Sprawdzenie, czy adres rozkazu rozgałęzienia jest w tablicy
  - **Nie** – wtedy:
    - przewidywanie rozgałęzienia jest wykonywane według jednej ze strategii statycznych
    - do tablicy jest wpisywany adres rozkazu rozgałęzienia, informacja o wykonaniu/niewykonaniu rozgałęzienia (bit historii) i (opcjonalnie) adres docelowy rozgałęzienia

  - **Tak** - wtedy:
    * przewidywanie rozgałęzienia jest wykonywane według bitów historii
    * do tablicy jest wpisywana informacja o wykonaniu/niewykonaniu rozgałęzienia (uaktualnienie bitów historii)

- 1 bit historii - algorytm przewidywania rozgałęzień dla jednego bitu historii - kolejne wykonanie rozkazu rozgałęzienia będzie przebiegało tak samo jak poprzednie.
- 2 bity historii
  - algorytm przewidywania rozgałęzień dla dwóch bitów historii bazuje na 2-bitowym automacie skończonym.
  - Interpretacja dwóch bitów historii (x y):
    * y: historia ostatniego wykonania skoku (0 – nie, 1 – tak)
    * x: przewidywanie następnego wykonania skoku (0 – nie, 1 – tak)
    * Ogólna zasada przewidywania - zmiana strategii następuje dopiero po drugim błędzie przewidywania.



4.10 Metody rozwiązywania hazardu danych

4.10.1 Co to jest?

Hazard danych - zależności między argumentami kolejnych rozkazów wykonywanych potokowo.

4.10.2 Metody usuwania hazardu danych

Jest kilka sposobów:

- Sprzętowe wykrywanie zależności i wstrzymanie napełniania potoku
- Wykrywanie zależności na etapie kompilacji i modyfikacja programu (np. dodanie rozkazu NOP)
- Wykrywanie zależności na etapie kompilacji, modyfikacja i optymalizacja programu (np. zamiana kolejności wykonywania rozkazów)
- Wyprzedzające pobieranie argumentów (zastosowanie szyny zwrotnej)

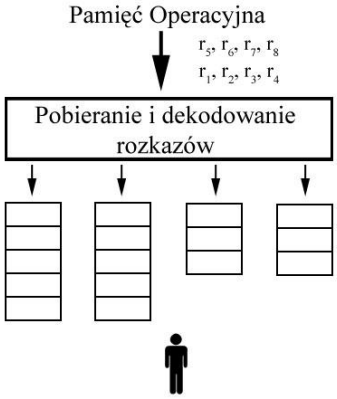
4.10.3 Problem

Jeśli faza wykonania rozkazu nie będzie mogła być wykonana w jednym takcie (np. dla rozkazów zmiennoprzecinkowych), to zachodzi konieczność wstrzymania napełniania potoku.

5 Architektura superskalarna

5.1 Co to jest?

Architektura umożliwiająca wykonanie w jednym takcie większej od 1 liczby instrukcji.



5.2 Cechy architektury superskalarnej

Możliwość wykonania kilku rozkazów w jednym takcie, co powoduje konieczność:

- Kilku jednostek potokowych
- Załadowania kilku rozkazów z pamięci operacyjnej w jednym takcie procesora

5.3 Zależności między rozkazami

5.3.1 Prawdziwa zależność danych

Read After Write (RAW)
Występuje w momencie kiedy jeden rozkaz wymaga argumentu obliczanego przez poprzedni rozkaz. Opóźnienie eliminowane za pomocą "wyprzedzającego pobierania argumentu" - dana nie jest zapisywana do rejestru, tylko pobierana bezpośrednio z poprzedniego rozkazu, który znajduje się w akumulatorze (jeżeli dobrze rozumiem rysunek ze slajdu 21, wykład 4).

5.3.2 Zależność wyjściowa

Write After Write (WAW)
Gdy rozkazy zapisująca dane do tego samego rejestru wykonują się równolegle to drugi z nich musi czekać aż pierwszy się zakończy. Układ sterujący musi kontrolować tego typu zależność.

5.3.3 Antyzależność

Write After Read (WAR)
W przypadku gdy pierwszy rozkaz czyta wartość rejestru, a drugi zapisuje coś do tego rejestru i oba wykonują się równolegle, to drugi musi czekać aż pierwszy odczyta swoje.

5.3.4 Wnioski

- Dopuszczenie do zmiany kolejności rozpoczynania wykonania (wydawania) rozkazów i / lub zmiany kolejności kończenia rozkazów prowadzi do możliwości wystąpienia zależności wyjściowej lub antyzależności.
- Zawartości rejestrów nie odpowiadają wtedy sekwencji wartości, która winna wynikać z realizacji programu

5.4 Metody eliminacji zależności

5.4.1 Metoda przemianowania rejestrów

- Stosowana w przypadku zwielokrotnienia zestawu rejestrów.
- Rejestry są przypisywane dynamicznie przez procesor do rozkazów.
- Gdy wynik rozkazu ma być zapisany do rejestru Rn, procesor angażuje do tego nową kopię tego rejestru.
- Gdy kolejny rozkaz odwołuje się do takiego wyniku (jako argumentu źródłowego), rozkaz ten musi przejść przez proces przemianowania.
- Przemianowanie rejestrów eliminuje antyzależność i zależność wyjściową.

Przykład procesu przemianowania rejestrów:

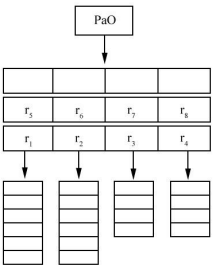
- I1: R3b ← R3a op R5a
- I2: R4b ← R3b + 1
- I3: R3c ← R5a + 1
- I4: R7b ← R3c op R4b

W powyższym przykładzie rozkaz I3 może być wykonany jako drugi (co zapobiegnie zależnościom RAW między I1 i I2 oraz I3 i I4), lub nawet jako pierwszy.

6 Architektura VLIW

6.1 Co to jest?

VLIW - *Very Long Instruction Word.*



6.2 Cechy

- Wspólna pamięć operacyjna
- Szeregowanie rozkazów

6.3 Szeregowanie rozkazów przez kompilator

- Podział rozkazów programu na grupy
- Sekwencyjne wykonywanie grup
- Możliwość równoległej realizacji rozkazów w ramach grupy
- Podział grupy na paczki
- Paczka = 3 rozkazy + szablon (3 x 41 + 5 = 128 bitów)
- Szablon - informacja o jednostkach funkcjonalnych, do których kierowane mają być rozkazy i ewentualna informacja o granicach grup w ramach paczki

6.4 Redukcja skoków warunkowych - predykacja rozkazów

Rozkazy uwarunkowane - uwzględnianie warunku w trakcie realizacji rozkazu.

6.5 Spekulatywne wykonanie rozkazów LOAD

- Problem: chybione odwołania do PaP (cache) i konieczność czekania na sprowadzenie do PaP linii danych
- Rozwiązanie: przesunięcie rozkazów LOAD jak najwyżej, aby zminimalizować czas ewentualnego oczekiwania.
- Rozkaz CHECK sprawdza wykonanie LOAD (załadowanie rejestru)

7 Wielowątkowość

7.1 Co to jest?

- Cecha systemu operacyjnego umożliwiająca wykonywanie kilku wątków w ramach jednego procesu
- Cecha procesora oznaczająca możliwość jednoczesnego wykonywanie kilku wątków w ramach jednego procesora (rdzenia)

7.2 Sprzętowa realizacja wielowątkowości

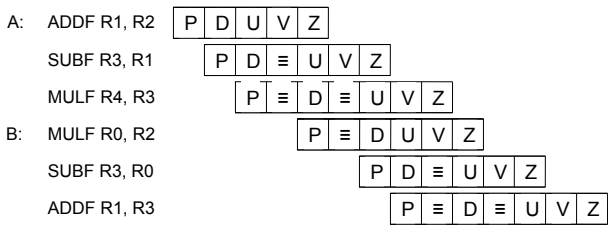
Celem współbieżnej realizacji dwóch (lub więcej) wątków w jednym procesorze (rdzeniu) jest minimalizacja strat cykli powstałych w trakcie realizacji pojedynczego wątku w wyniku:

- chybionych odwołań do pamięci podręcznej,
- błędów w przewidywaniu rozgałęzień,
- zależności między argumentami kolejnych rozkazów

7.3 Wielowątkowość gruboziarnista

Coarse-grained multithreading.

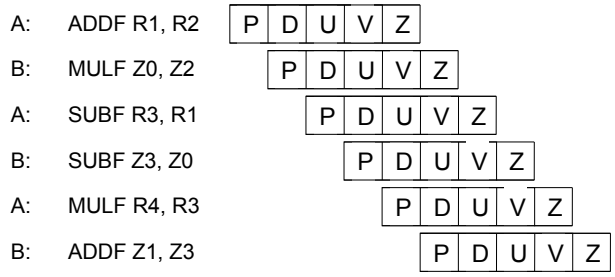
- Przełączanie wątków następuje przy dłuższym opóźnieniu wątku w potoku (np. chybione odwołanie do pamięci podręcznej (nawet L2))
- W niektórych rozwiązaniach rozpoczęcie nowego wątku następuje dopiero po opróżnieniu potoku
- Zaletą jest prostota procesu przełączania wątków
- Wadą takiego rozwiązania są straty czasu przy krótszych opóźnieniach potoku



7.4 Wielowątkowość drobnoziarnista

Fine-grained multithreading.

- Przełączanie wątków następuje po każdym rozkazie
- Wątek oczekujący (np. na dostęp do pamięci) jest pomijany
- Zaletą jest unikanie strat nawet przy krótkich opóźnieniach wątków
- Istotnym wymaganiem dla procesora jest szybkie (w każdym takcie) przełączanie wątków
- Pewną wadą jest opóźnienie realizacji wątków w pełni gotowych do wykonania



7.5 Warunki sprzętowej realizacji wielowątkowości

- powielenie zestawów rejestrów uniwersalnych (lub powielenie tabel mapowania rejestrów)
- powielenie liczników rozkazów
- powielenie układów dostępu do pamięci podręcznej (tabel stron)
- powielenie sterowników przerwań

7.6 Wielowątkowość w procesorze dwupotokowym

Reguły realizacji i przełączania wątków:

1. **Wielowątkowość gruboziarnista**
   - wątek realizowany w kolejnych taktach do momentu wstrzymania rozkazu
   - do obu potoków wprowadzane są rozkazy tylko jednego wątku (w jednym takcie!)
2. **Wielowątkowość drobnoziarnista**
   - w kolejnych taktach realizowane są naprzemiennie rozkazy kolejnych wątków (przełączanie wątków co takt)
   - do obu potoków wprowadzane są rozkazy tylko jednego wątku (w jednym takcie!)
3. **Wielowątkowość współbieżna** (SMT -Simultaneous multithreading)
   - wątek realizowany do momentu wstrzymania rozkazu
   - do obu potoków w jednym takcie mogą być wprowadzane rozkazy różnych wątków

7.7 Mankamenty współbieżnej wielowątkowości

- Rywalizacja wątków w dostępie do pamięci podręcznej - mniejsza wielkość PaP przypadająca na wątek
- Większe zużycie energii (w porównaniu z procesorami dwurdzeniowymi)
- Możliwość monitorowanie wykonania jednego wątku przez inny wątek (złośliwy), poprzez wpływ na współdzielone dane pamięci podręcznej - kradzież kluczy kryptograficznych

8 Klasyfikacja komputerów równoległych

8.1 Formy równoległości w architekturze komputerów

8.1.1 Równoległość na poziomie rozkazów

Wykonywanie w danej chwili wielu rozkazów w jednym procesorze.

- Mechanizmy potokowe - w procesorach CISC i RISC
- Architektura superskalarna i VLIW

8.1.2 Równoległość na poziomie procesorów

Wykonywanie w danej chwili wielu rozkazów w wielu procesorach.

- Komputery wektorowe
- Komputery macierzowe
- Systemy wieloprocesorowe
- Klastry (systemy wielokomputerowe)

8.2 Rodzaje równoległości w aplikacjach

8.2.1 Równoległość poziomu danych

DLP - Data Level Parallelism.
Pojawia się kiedy istnieje wiele danych, które mogą być przetwarzane w tym samym czasie.

8.2.2 Rówoległość poziomu zadań

TLP - Task Level Parallelism.
Pojawia się kiedy są tworzone zadania, które mogą być wykonywane niezależnie i w większości równolegle.

8.3 Drogi wykorzystania równoległości aplikacji w architekturze komputerów

- **Równoległość poziomu rozkazów** (ILP - Instruction Level Parallelism) - odnosi się do przetwarzania potokowego i superskalarnego, w których w pewnym (niewielkim) stopniu wykorzystuje się równoległość danych.
- **Architektury wektorowe i procesory graficzne** - wykorzystują równoległość danych poprzez równoległe wykonanie pojedynczego rozkazu na zestawie danych.
- **Równoległość poziomu wątków** (TLP - Thread Level Parallelism) - odnosi się do wykorzystania równoległości danych albo równoległości zadań w ściśle połączonych systemach (ze wspólną pamięcią), które dopuszczają interakcje między wątkami.
- **Równoległość poziomu zleceń** (RLP - Request Level Parallelism) - odnosi się do równoległości zadań określonych przez programistę lub system operacyjny. Ta forma równoległości jest wykorzystywana w systemach luźno połączonych (z pamięcią rozproszoną) i klastrach.

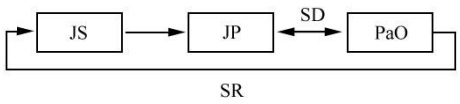
8.4 Klasyfikacja Flynna

M. Flynn, 1966

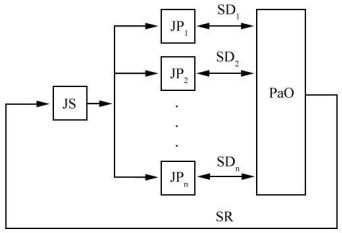
8.4.1 Kryterium klasyfikacji

Liczba strumieni rozkazów i liczba strumieni danych w systemie komputerowym.

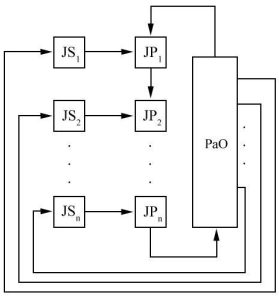
- **SISD**: Single Instruction, Single Data Stream



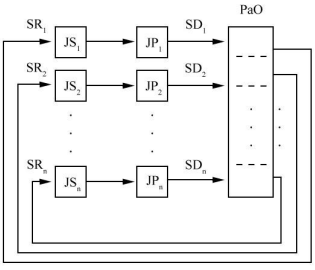
- **SIMD**: Single Instruction, Multiple Data Stream



- **MISD**: Multiple Instruction, Single Data Stream



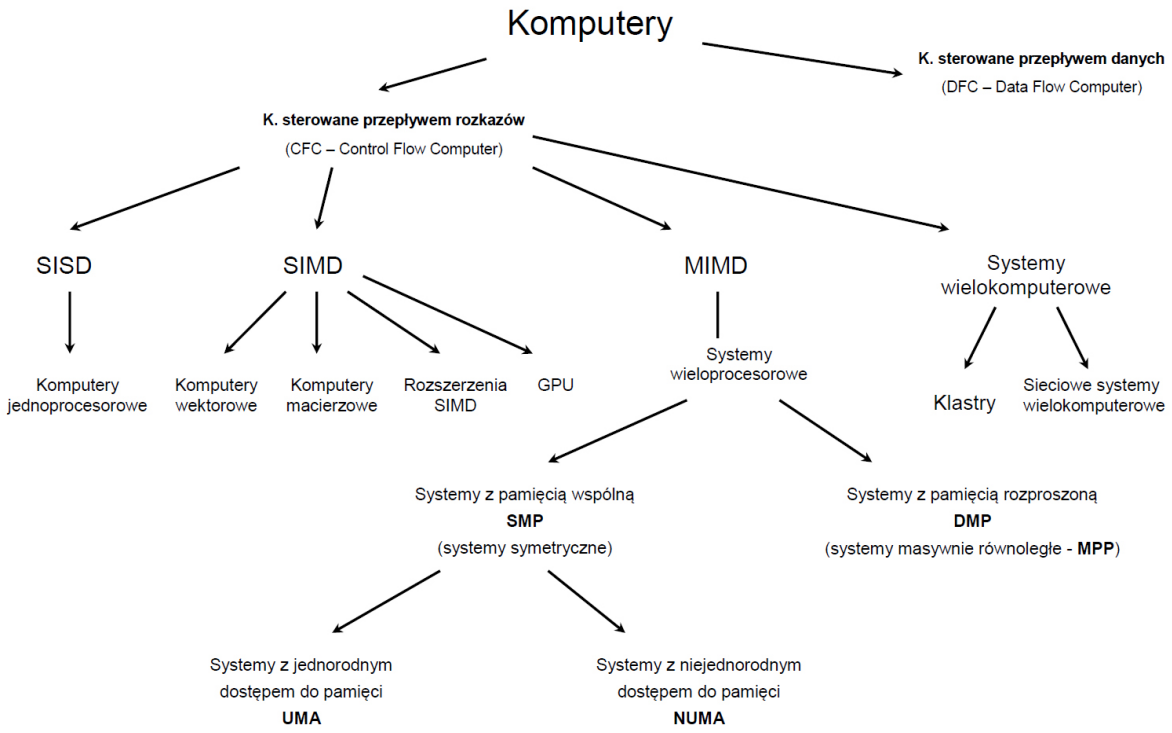
- **MIMD**: Multiple Instruction, Multiple Data Stream



Gdzie:

- **JS** – Jednostka sterująca
- **JP** – Jednostka przetwarzająca
- **PaO** – Pamięć operacyjna

8.4.2 Klasyfikacja opisowa



9 Architektura SIMD

9.1 Co to jest?

- Tłumaczenie *Single Instruction Multiple Device*

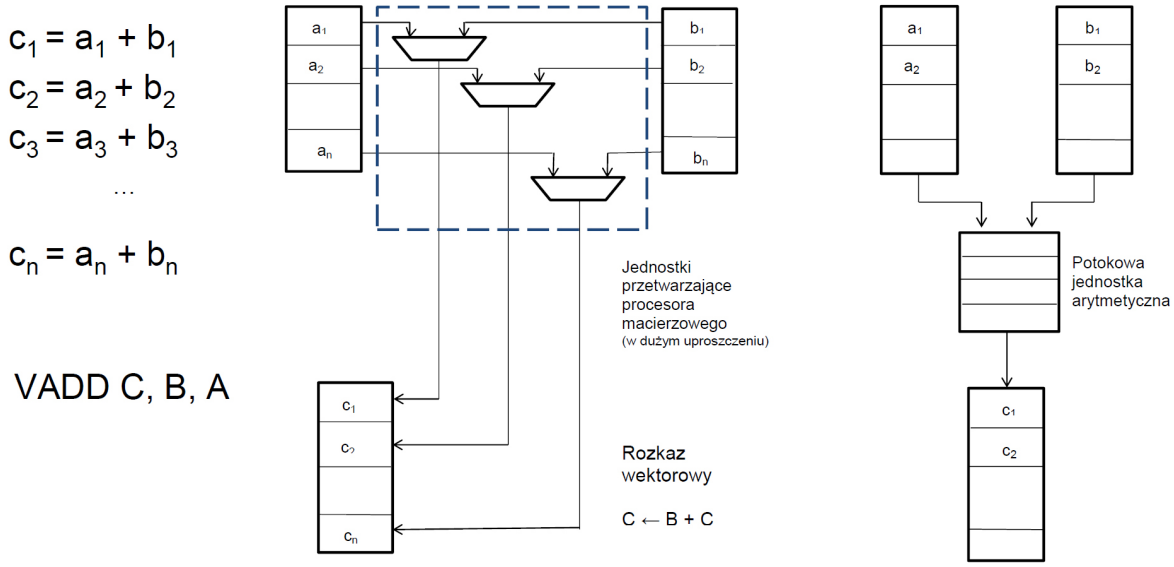
- Cecha wyróżniająca dla programisty - rozkazy wektorowe (rozkazy z argumentami wektorowymi).

- Dwa różne podejścia do sprzętowej realizacji rozkazów wektorowych:

  – Komputery (procesory) macierzowe

  – Komputery wektorowe

Idee realizacji obu (macierzowy i wektorowy):



9.2 Komputery wektorowe

9.2.1 Lokalizacja wektorów danych

- Pamięć operacyjna (STAR 100)
- Rejestry wektorowe (Cray -1)

9.2.2 Przykład rozkazu

Rozkaz dodawania wektorów: VADDF A,B,C,n
Czas wykonania:

$$t_w = t_{start} + (n-1) \times \tau$$

W komputerze macierzowym czas wykonywania tego rozkazu jest równy *const*.

9.2.3 Przyspieszenie

Przyspieszenie jest stosunkiem czasu wykonywania w komputerze klasycznym (szeregowo) do czasu wykonywania w komputerze wektorowym.

$$a = lim_{n\to\infty} \frac{15 \times \tau \times n}{t_{start} + (n-1) \times \tau} = 15$$

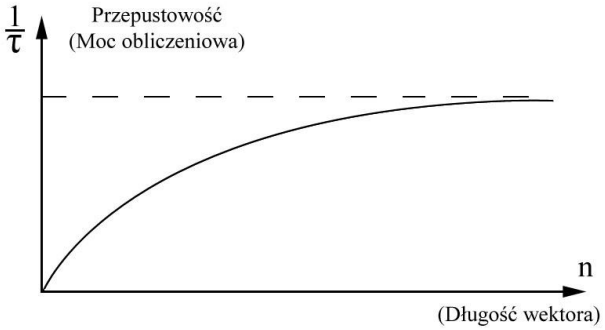
9.2.4 Przepustowość

Przepustowość (moc obliczeniowa) jest stosunkiem ilości operacji zmiennoprzecinkowych do czasu ich wykonania.

$$Przep = lim_{n\to\infty} \frac{n}{t_{start} + (n-1) \times \tau} = \frac{1}{\tau}$$

Wymiarem przepustowości jest FLOPS.

9.2.5 Zależność mocy obliczeniowej od długości wektora



9.2.6 Podsumowanie

1. Hardware
    - rozkazy wektorowe
    - duża liczba potokowych jednostek arytmetycznych (specjalizowanych)
    - duża liczba rejestrów (nawet kilkaset tysięcy)
2. Software
    - klasyczne języki: Fortran, C
    - klasyczne algorytmy
    - kompilatory wektoryzujące

9.2.7 Zastosowanie

- Numeryczna symulacja ośrodków ciągłych
- Równania różniczkowe, równania różnicowe, układy równań algebraicznych (rachunek macierzowy)
- Dziedziny zastosowań:
    - prognozowanie pogody
    - symulacja aerodynamiczna
    - sejsmiczne poszukiwania ropy naftowej i innych surowców
    - symulacja reakcji jądrowych
    - medycyna i farmacja
    - obliczenia inżynierskie dużej skali

9.3 Komputery macierzowe

9.3.1 Co to jest?

Architektura komputerów macierzowych - model SIMD w dwóch wariantach:

- SIMD - DM (z pamięcią rozproszoną)
- SIMD - SM (z pamięcią wspólną)

9.3.2 Elementy komputera macierzowego

1. **Jednostka sterująca** - procesor wykonujący rozkazy sterujące i skalarne oraz inicjujący wykonanie rozkazów wektorowych w sieci elementów przetwarzających.
2. **Elementy przetwarzające** (procesorowe) - jednostki arytmetyczno-logiczne wykonujące operacje elementarne rozkazów wektorowych.
3. **Sieć łącząca** - łączy elementy przetwarzające między sobą lub z modułami pamięci operacyjnej; warianty:
   - sieć statyczna: pierścień, gwiazda, krata, drzewo, hipersześcian
   - sieć dynamiczna: jednostopniowa; wielostopniowa (wyróżnia połączenia blokujące i nieblokujące)

9.3.3 Podsumowanie

- Architektura SIMD
- Jednostka sterująca + jednostka macierzowa
- Rozkazy wektorowe - wykonywane synchronicznie w sieci (macierzy) EP
- Skomplikowana wymiana danych między EP
- Trudne programowanie - konieczność tworzenia nowych wersji algorytmów

9.4 Model SIMD w procesorach superskalarnych

9.4.1 Technologia MMX

- 8 rejestrów 64-bitowych MMX
- Nowe typy danych
- Rozszerzony zestaw instrukcji (57 instrukcji)
- Realizacja operacji na krótkich wektorach wg modelu SIMD

9.5 Technologia SSE

- 8 rejestrów 128-bitowych
- Osiem 16-bitowych argumentów (elementów wektora) typu integer
- Cztery 32-bitowe argumenty integer/fplub dwa 64-bitowe
- Operacje zmp na 4-elementowych wektorach liczb 32-bit (pojed. prec.)

10 Karty graficzne i architektura CUDA

10.1 Charakterystyka

- GPU - Graphics Processing Unit
- Wcześniejsze GPU - specjalizowane języki (HLSL, GLSL czy NVIDIA Cg), tylko rendering
- CUDA (Compute Unified Device Architecture) -architektura wielordzeniowych procesorów graficznych (GPU)
- Uniwersalna architektura obliczeniowa połączona z równoległym modelem programistycznym
- wsparcie dla języków C/C++

- GPGPU = GPU + CUDA
- CUDA - obsługiwana przez karty graficzne GeForce i GeForce Mobile od serii 8 (GeForce 8800), nowsze układy z rodzin Tesla i Quadro, Fermi, obecnie Kepler

10.2 Architektura CUDA

- W miejsce oddzielnych potoków przetwarzających wierzchołki i piksele wprowadzenie uniwersalnego procesora przetwarzającego wierzchołki, piksele i ogólnie geometrię, a także uniwersalne programy obliczeniowe
- Wprowadzenie procesora wątków eliminującego „ręczne" zarządzanie rejestrami wektorowymi
- Wprowadzenie modelu SIMT (single-instruction multiple-thread), w którym wiele niezależnych wątków wykonuje równocześnie tę samą instrukcję
- Wprowadzenie współdzielonej pamięci oraz mechanizmów synchronizacji wątków (barrier synchronization) dla komunikacji między wątkami

10.3 Multiprocesor strumieniowy

Architektura GT 200.

- 8 rdzeni C1 -C8 (SP)
- podręczna pamięć instrukcji (ang. instruction cache),
- podręczna pamięć danych (ang. constant cache) - pamięć tylko do odczytu,
- pamięć współdzielona (ang. shared memory)
- 16 384 rejestry,
- jednostka arytmetyczna wykonująca obliczenia zmiennoprzecinkowe podwójnej precyzji (fp64),
- dwie jednostki arytmetyczne przeznaczone do obliczania funkcji specjalnych (ang. special function unit),
- pamięć globalna

10.4 Model programistyczny CUDA

- Specjalny kompilator NVCC
- Podział programu na kod wykonywany przez procesor (ang. *Host code*) i przez urządzenie (kartę graficzną) (ang. Device code) - kernel
- Realizacja operacji równoległych według modelu SIMT (*Single Instruction Multiple Threading*)

10.5 Wykonanie obliczeń z użyciem architektury CUDA (5 faz)

1. Przydzielenie w pamięci globalnej obszaru pamięci dla danych, na których będą wykonywane obliczenia przez kernel.
2. Przekopiowanie danych do przydzielonego obszaru pamięci.
3. Zainicjowanie przez CPU obliczeń wykonywanych przez GPU, tj. wywołanie kernela.
4. Wykonanie przez wątki (z użyciem GPU) obliczeń zdefiniowanych w kernelu.
5. Przekopiowanie danych z pamięci globalnej do pamięci operacyjnej.

10.6 CUDA procesor (rdzeń)

- Potokowa jednostka arytmetyczna zmp
- Potokowa jednostka arytmetyczna stp
- Ulepszona realizacja operacji zmp FMA (fused multiply-add) dla pojedynczej i podwójnej precyzji

11 Wątki

11.1 Co to jest?

- Wątek reprezentuje pojedynczą operację (a single work unit or operation)
- Wątki są automatycznie grupowane w bloki, maksymalny rozmiar bloku = 512 wątków (w architekturze Fermi i wyższych –1024 wątki).
- Bloki grupowane są w siatkę (grid -kratę)
- Grupowanie wątków –bloki o geometrii 1, 2 lub 3-wymiarowej
- Grupowanie bloków –siatka (grid) o geometrii 1, 2-wymiarowej
- Wymaga się, aby bloki wątków tworzących siatkę mogły się wykonywać niezależnie: musi być możliwe ich wykonanie w dowolnym porządku, równolegle lub szeregowo.

11.2 Grupowanie wątków w bloki i siatkę

- Siatka o geometrii jednowymiarowej (trzy bloki wątków)
- Każdy blok -geometria dwuwymiarowa (wymiary 2 x 3)

11.3 Sprzętowa organizacja wykonywania wątków

- Przy uruchomieniu *kernela* wszystkie bloki tworzące jego siatkę obliczeń są rozdzielane pomiędzy multiprocesory danego GPU
- Wszystkie wątki danego bloku są przetwarzane w tym samym multiprocesorze
- W danej chwili (cyklu) pojedynczy rdzeń multiprocesora wykonuje jeden wątek programu
- Multiprocesor tworzy, zarządza, szereguje i wykonuje wątki w grupach po 32, nazywanych wiązkami (*warp*).
- Wiązki są szeregowane do wykonania przez *warp scheduler*. Wiązka wątków jest wykonywana jako jeden wspólny rozkaz (analogia do rozkazu SIMD, tzn. rozkazu wektorowego)
- Sposób wykonania wiązki wątków (rozkazu SIMD) zależy od budowy multiprocesora:
  - Dla architektury Fermi (32 procesory w jednym multiprocesorze / 2 *warp-schedulery* = 16 procesorów na 1 wiązkę) wiązka jest wykonywana jako 2 rozkazy - wiązka jest dzielona na dwie połówki (*half–warp*) wykonywane jako 2 rozkazy (te same, ale na dwóch zestawach danych).
  - Dla architektury Tesla (8 procesorów w jednym multiprocesorze, 1 *warp-scheduler*) wiązka jest dzielona na cztery ćwiartki (*quarter-warp*) wykonywane jako 4 kolejne rozkazy (te same, ale na czterech zestawach danych).
- Konstrukcja *warp schedulera* umożliwia uruchomienie wielu wiązek wątków współbieżnie - *warp scheduler* pamięta wtedy adresy wiązek, przypisane im rozkazy SIMD oraz ich stan (gotowość do wykonania lub stan oczekiwania na pobranie danych z pamięci).
- Współbieżne uruchomienie wielu wiązek pozwala zmniejszyć straty związane z oczekiwaniem na dane (zwykle długi czas dostępu do pamięci).

12 Rodzaje pamięci multiprocesora

12.1 Pamięć globalna

Duża pamięć, o czasie życia aplikacji (dane umieszczone w tej pamięci są usuwane po zakończeniu aplikacji), dostępna dla każdego wątku w dowolnym bloku, ale o dość długim czasie dostępu wynoszącym ok. 400-600 taktów zegara.

12.2 Pamięć współdzielona

Niewielka pamięć o czasie życia bloku (zakończenie działania bloku powoduje usunięcie danych w niej przechowywanych), dostępna dla każdego wątku w bloku dla którego jest dedykowana, o bardzo krótkim czasie dostępu.

12.3 Pamięć stałych

Niewielki fragment pamięci globalnej, który jest cache-owany, przez co dostęp do niego jest bardzo szybki. Jest ona tylko do odczytu. Czas życia pamięci stałych oraz jej dostępność jest taka sama jak pamięci globalnej.

12.4 Rejestry

Niewielka, bardzo szybka pamięć o czasie życia wątku (po zakończeniu wątku dane z rejestrów są usuwane). Tylko jeden wątek może w danym momencie korzystać z danego rejestru.

12.5 Pamięć lokalna i pamięć tekstur

Podobnie jak w przypadku pamięci stałych, są to dedykowane fragmenty pamięci globalnej. Pamięć lokalna jest wykorzystywana do przechowywania danych lokalnych wątku, które nie mieszczą się w rejestrach, a pamięć tekstur posiada specyficzne metody adresowania i cache-owanie specyficzne dla zastosowań graficznych.

13 Systemy wieloprocesorowe (UMA)

13.1 Rodzaje

- Systemy z pamięcią wspólną
- Systemy z pamięcią rozproszoną

13.2 Systemy z pamięcią wspólną

- Systemy z jednorodnym dostępem do pamięci (UMA – *Uniform Memory Access*)
- Systemy z niejednorodnym dostępem do pamięci (NUMA – Non - *Uniform Memory Access*)

13.2.1 Klasyfikacja

- Systemy ze wspólną magistralą
- Systemy wielomagistralowe
- Systemy z przełącznicą krzyżową
- Systemy z wielostopniową siecią połączeń
- Systemy z pamięcią wieloportową
- Systemy z sieciami typu punkt - punkt

13.3 Skalowalność

System skalowalny - System, w którym dodanie pewnej liczby procesorów prowadzi do proporcjonalnego przyrostu mocy obliczeniowej.

13.4 Systemy ze wspólną magistralą

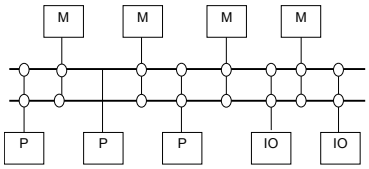
- Prostota konstrukcji – niska złożoność układowa całości
- Niski koszt
- Łatwość rozbudowy – dołączenia kolejnego procesora, ale tylko w ograniczonym zakresie
- Ograniczona złożoność magistrali (jej szybkość jest barierą)
- Niska skalowalność

13.4.1 Protokół MESI

Ten rodzaj systemu posiada problem zapewnienia spójności pamięci podręcznych (*snooping*). W celu rozwiązania go wykorzystuje w tym celu protokół MESI:

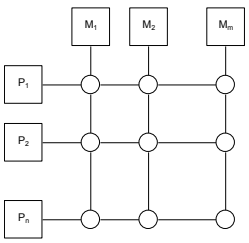
- I - invalid
- S - shared
- E - exclusive
- M - modified

13.5 Systemy wielomagistralowe



- Wielokrotnie zwiększona przepustowość
- Konieczność stosowania układu arbitra do sterowania dostępem do magistral
- Rozwiązania kosztowne

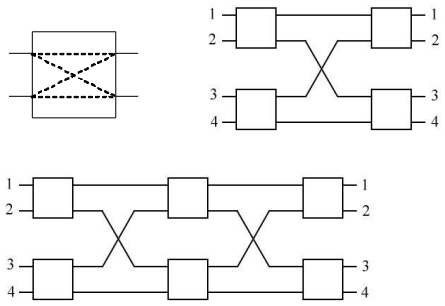
13.6 Systemy z przełącznicą krzyżową



- Zadania każdego przełącznika
  - Rozwiązywanie konfliktów dostępu do tego samego modułu pamięci
  - Zapewnienie obsługi równoległych transmisji, krzyżujących się w przełączniku
- Duża przepustowość
- Duża złożoność układowa $O(n^2)$

- Wysoki koszt
- Problem: realizacja techniczna przełącznicy krzyżowej dla dużych n – wykorzystanie wielostopniowych sieci połączeń

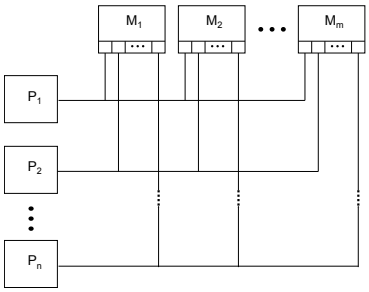
13.7 Systemy z wielostopniową siecią połączeń



Przykłady:

- Nieblokująca sieć Closa
- Sieć wielostopniowa Benesa
- Sieć wielostopniowa typu Omega

13.8 Systemy z pamięcią wieloportową



- Każdy procesor ma niezależny dostęp do modułów pamięci – poprawa wydajności
- Układ sterowania modułu pamięci rozstrzyga konflikty dostępu – większa złożoność
- Możliwość skonfigurowania części pamięci jako prywatnej dla jednego lub kilku procesorów
- Stosowana technika zapisu do pamięci cache – *write through* – poprzez własną magistralę i port w module
- Ograniczona liczba portów w module

13.9 Systemy z sieciami typu punkt-punkt

13.9.1 HyperTransport (HT)

- Technologia wprowadzona w 2001 przez HyperTransportConsortium(AMD, Apple, Cisco, Nvidia, Sun i in.)
- Przeznaczona do łączenia procesorów, pamięci i układów we/wy -technologia HT zastąpiła wspólną magistralę
- Dla łączenia wielu procesorów stosowana razem z techniką NUMA

- Topologia punkt -punkt
- Sieć dwukierunkowa, szeregowo/równoległa, o wysokiej przepustowości, małych opóźnieniach,
- Łącza o szerokości 2, 4, 8, 16, or 32 bity. Teoretyczna przepustowość: 25.6 GB/s (3.2GHz x 2 transfers per clock cycle x 32 bits per link) dla jednego kierunku lub 51.2 GB//s łącznie dla obu kierunków transmisji
- Transmisja pakietów składających się z 32-bitowych słów

13.9.2 Intel QuickPath Interconnect (QPI)

- Następca Front-SideBus (FSB) -magistrali łączącej procesor z kontrolerem pamięci
- Tworzy bardzo szybkie połączenia między procesorami i pamięcią oraz procesorami i hubami we/wy
- Tworzy mechanizm „scalablesharedmemory" (NUMA) –zamiast wspólnej pamięci dostępnej przez FSB, każdy Procesor ma własną dedykowaną pamięcdostępną przez IntegratedMemory Controleroraz możliwość dostępu do dedykowanej pamięci innych procesorów poprzez QPI
- Podstawową zaletą QPI jest realizacja połączeń punkt-punkt (zamiast dostępu przez wspólną magistralę)

13.10 Podsumowanie

- Symetryczna architektura – jednakowy dostęp procesorów do pamięci operacyjnej oraz we/wy
- Utrzymanie spójności pamięci podręcznych (cache):
  - *snooping* - metoda starsza i mało skalowalna (głównie w systemach ze wspólną magistralą)
  - katalog - metoda lepiej skalowalna, stosowana razem z sieciami typu punkt - punkt
- Łatwe programowanie (realizacja algorytmów równoległych)
- Niska skalowalność:
  - Mechanizm *snoopingu*, zapewniający spójność pamięci podręcznych węzłów systemów UMA nie jest skalowalny dla bardzo dużej liczby węzłów.
  - W systemach z dużą liczbą węzłów, posiadających lokalną pamięć, dla zapewnienia spójności pamięci podręcznych jest stosowane rozwiązanie oparte na katalogu.

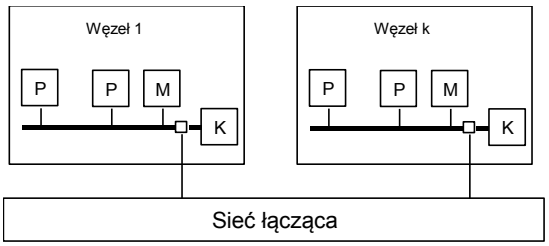
14 Systemy NUMA

Systemy wieloprocesorowe z niejednorodnym dostępem do pamięci.
NUMA (*Non-Uniform Memory Access*).

14.1 Rodzaje

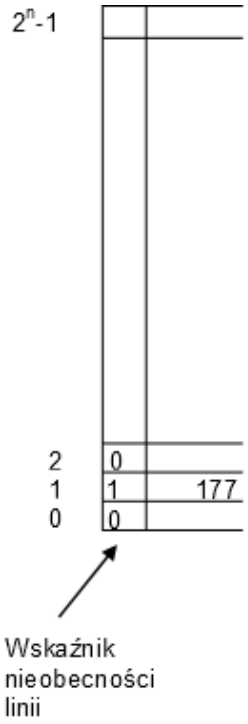
- NC-NUMA (*Non-cached NUMA*)
  - Odwołania do nielokalnej PaO przekierowywane do odległych węzłów
  - Odwołania do nielokalnej PaO wolniejsze ok. 10 razy
- CC-NUMA (*Cache Coherent NUMA*)
  - Wykorzystuje węzły i katalogi

14.2 Węzeł



14.3 Katalog

14.3.1 Najprostsza postać katalogu



14.3.2 Rozmiar katalogu

- dla PaO węzła = 1 GB i linii = 128 B katalog musiałby mieć 2^{23} pozycji
- dla PaO węzła = 8 GB i linii = 128 B katalog musiałby mieć 2^{26} pozycji
- dlatego katalog praktycznie jest zazwyczaj realizowany jako pamięć asocjacyjna, o znacznie mniejszych rozmiarach

14.3.3 Różne warianty organizacji katalogu

Zawartość pozycji katalogu:

1. numer węzła aktualnie posiadającego daną linię
2. k numerów węzłów aktualnie posiadających daną linię
3. n bitów (1 bit na węzeł) wskazujących posiadanie linii przez węzeł
4. element listy pól z numerami węzłów aktualnie posiadających daną linię

14.4 System DASH

14.4.1 Co to jest?

- Pierwszy system CC-NUMA wykorzystujący katalog
- 16 węzłów połączonych kratą
- Węzeł: 4 procesory MIPS R3000 + 16 MB RAM + katalog
- Linia PAP = 16 B
- Katalog = 1 M wierszy 18-bitowych
  - Wiersz katalogu= 16 bitów obecności linii w węzłach + stan linii
  - **Stan linii**: uncached, shared, modified

14.4.2 Interpretacja stanu linii

- **uncached** – linia pamięci jest tylko w pamięci lokalnej (domowej)
- **shared** – linia PaP została przesłana do odczytu do kilku węzłów (ich pamięci lokalnych)
- **modified** – linia PaP jest w pamięci domowej nieaktualna (została zmodyfikowana w innym węźle)

14.4.3 Operacja odczytu

- Stan żądanej linii = *uncached* lub *shared* → linia jest przesyłana do węzła żądającego; stan linii := *shared*.
- Stan żądanej linii = *modified* → sterownik katalogu domowego żądanej linii przekazuje żądanie do węzła x posiadającego linię. Sterownik katalogu węzła x przesyła linię do węzła żądającego oraz węzła domowego tej linii; stan linii := *shared*.

14.4.4 Operacja zapisu

- Przed zapisem węzeł żądający linii musi być jedynym jej posiadaczem.
- Węzeł żądający posiada linię
  - stan linii = *modified* → zapis jest wykonywany
  - stan linii = *shared* → węzeł przesyła do domowego katalogu tej linii żądanie unieważnienia innych kopii linii; stan linii := *modified*
- Węzeł żądający nie posiada linii → wysyła żądanie dostarczenia linii do zapisu
  - stan linii = *uncached* → linia jest przesyłana do węzła żądającego; stan linii := *modified*
  - stan linii = *shared* → wszystkie kopie linii są unieważniane, potem jak dla *uncached*
  - stan linii = *modified* → przekierowanie żądania do węzła x posiadającego linię. Węzeł x unieważnia ją u siebie i przesyła żądającemu.

14.4.5 Katalog czy snooping

Mechanizm spójności oparty o katalog jest bardziej skalowalny od mechanizmu „snooping", ponieważ wysyła się w nim bezpośrednią prośbę i komunikaty unieważniające do tych węzłów, które mają kopie linii, podczas gdy mechanizm „snooping" rozgłasza (*broadcast*) wszystkie prośby i unieważnienia do wszystkich węzłów.

14.5 Podsumowanie

- PaO fizycznie rozproszona, ale logicznie wspólna
- Niejednorodny dostęp do pamięci - PaO lokalna, PaO zdalna
- Utrzymanie spójności pamięci podręcznych (cache) - katalog
- Hierarchiczna organizacja: procesor – węzeł (system UMA) – system NUMA
- Zalety modelu wspólnej pamięci dla programowania
- Dobra efektywność dla aplikacji o dominujących odczytach z nielokalnej pamięci
- Gorsza efektywność dla aplikacji o dominujących zapisach do nielokalnej pamięci
- Skalowalność: 1024 – 2560 rdzeni

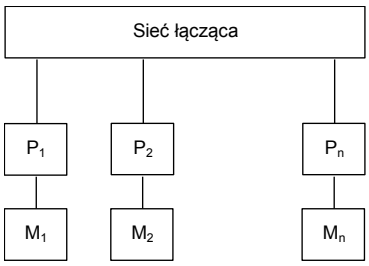
15 Systemy SMP - podsumowanie

- Symetryczna architektura – jednakowy dostęp procesorów do pamięci operacyjnej oraz we/wy (na poziomie fizycznych przesyłów – tylko w systemach wieloprocesorowe z pamięcią wspólną fizycznie - UMA)
- Utrzymanie spójności pamięci podręcznych (cache):
  - systemy UMA - snooping lub katalog
  - systemy NUMA - katalog
- Łatwe programowanie (realizacja algorytmów równoległych)
- Niska (UMA) i średnia (NUMA) skalowalność

16 Systemy MMP

Systemy wieloprocesorowe z pamięcią rozproszoną.
MPP – *Massively Parallel Processors*.

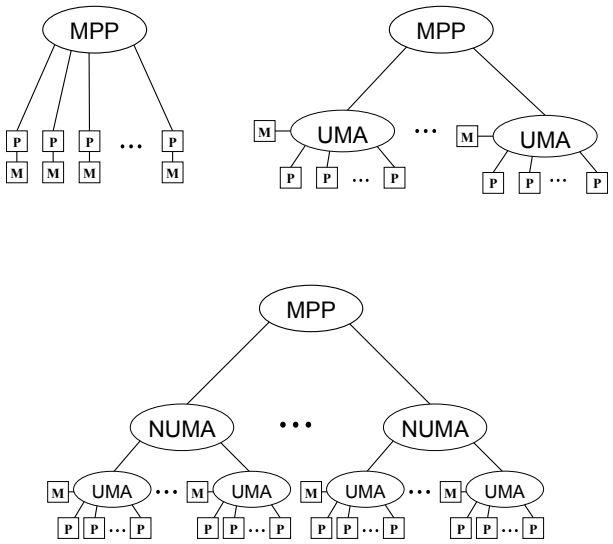
16.1 Uproszczona organizacja



16.2 Hierarchiczna organizacja

16.2.1 Rodzaje węzłów

- Procesor + PaO
- System wieloprocesorowy z pamięcią wspólną UMA
- System wieloprocesorowy NUMA (z niejednorodnym dostępem do pamięci)



16.3 Topologia

Sieci łączące węzły MPP.

- Hipersześcian
- Krata 2D, 3D
- Torus 2D, 3D
- Przełącznica krzyżowa (hierarchiczne przełącznice krzyżowe)
- Sieci wielostopniowe (Omega, Butterfly, grube drzewo, Dragonfly i inne)
- Sieci specjalizowane (*proprietary / custom network*)

16.4 Obsługa przesyłu komunikatów

Obsługa przesyłu komunikatów w węzłach systemu wieloprocesorowego.

- Programowa obsługa przesyłu przez procesory węzłów pośredniczących (systemy I generacji)
- Sprzętowa obsługa przesyłu przez routery węzłów pośredniczących, bez angażowania procesorów (systemy II generacji)

16.5 Narzędzia programowe

Narzędzia programowe wspierające budowę programów z przesyłem komunikatów:

- PVM (Parallel Virtual Machine)
- MPI (Message Passing Interface)
- Inne (Cray SHMEM, PGAS)

16.6 Podsumowanie

- Hierarchiczna architektura
- Węzeł: procesor, system UMA, (system NUMA)
- Bardzo duża skalowalność
- Wolniejsza (na ogół) komunikacja – przesył komunikatów
- Dedykowane, bardzo szybkie, sieci łączące

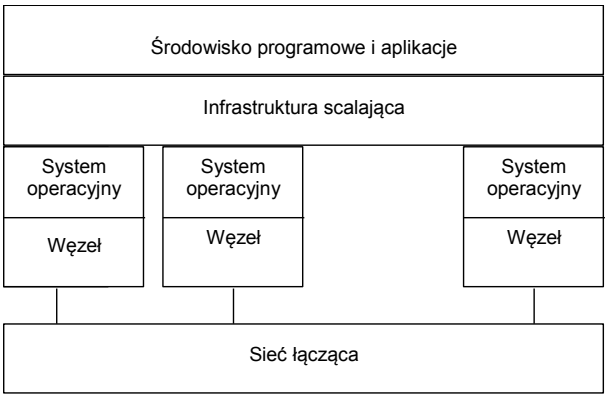
17 Klastry

Klaster (komputerowy) (ang. *cluster*).

17.1 Definicja

- Klaster to zestaw pełnych komputerów (węzłów) połączonych razem, pracujących jako jeden system.
- Wikipedia:
  Klaster komputerowy -grupa połączonych jednostek komputerowych, które współpracują ze sobą w celu udostępnienia zintegrowanego środowiska pracy
- A computer cluster consists of a set of loosely connected computers that work together so that in many respects they can be viewed as a single system.

17.2 Ogólna struktura systemów typu klaster



17.3 Ogólna charakterystyka klastrów

17.3.1 Węzły

- Serwery SMP
- Pełne komputery: PC, stacje robocze

17.3.2 System operacyjny

- Linux
- Unix
- Windows

17.3.3 Infrastruktura scalająca

- MPI (*Message Passing Interface*)
- PVM (*Parallel Virtual Machine*)
- SSI (*Single System Image*)

17.3.4 Komunikacja między węzłami

- Przesył komunikatów

17.3.5 Sieci łączące

- Sieci specjalizowane – starsze rozwiązania
- Sieci LAN (*Local Area Network*)
- Sieci specjalizowane -nowsze rozwiązania

17.3.6 Cele budowy klastrów

- Wysoka wydajność (klastry wydajnościowe, klastry obliczeniowe) (*High-performace clusters*)
- Wysoka niezawodność (klastry niezawodnościowe) (*High-availability clusters*)
- Równoważenie obciążenia (*Load balancing clusters*)

17.3.7 Inne

- Zależność moc obliczeniowa –niezawodność
- Korzystny wskaźnik wydajność -cena

17.4 Sieci łączące klastrów

17.4.1 Sieci specjalizowane - starsze rozwiązania

- HiPPI (*High Performance Parallel Interface*) – pierwszy standard sieci "near-gigabit" (0.8 Gbit/s) - kabel 50-par (tylko superkomputery)
- Memory Channel

17.4.2 Sieci LAN

- Ethernet
- Fast Ethernet
- Gigabit Ethernet

17.4.3 Sieci specjalizowane - nowsze rozwiązania

Systemowe SAN.

- Myrinet
- Quadrics (QsNet)
- SCI (Scalable Coherent Interface)
- InfiniBand

17.5 Fibre Channel

- Technologia sieciowa o dużej przepustowości (16 Gb/s) stosowana zwykle do łączenia komputera z pamięcią zewnętrzną
- Standard magistrali szeregowej definiujący wielowarstwową architekturę
- Powstał w 1988 jako uproszczona wersja HIPPI
- Łącza światłowodowe i miedziane
- Głównym stosowanym protokołem jest SCSI, ponadto ATM, TCP/IP

17.6 Sieci łączące -różnice

- Parametry – przepustowość, czas opóźnienia
- Topologia
  - Ethernet – magistrala, gwiazda, struktury hierarchiczne
  - Sieci specjalizowane – sieć przełączników (*switched fabric*) – popularna topologia „grubego drzewa"

17.7 Klastry o wysokiej niezawodności

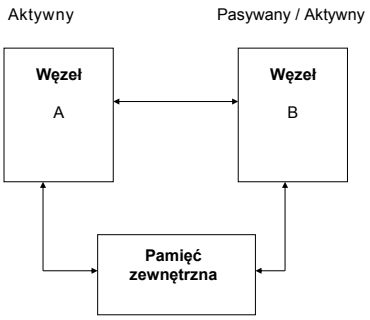
17.7.1 Czynniki tworzące wysoką niezawodność klastrów

1. Redundancja węzłów (mocy obliczeniowej)
2. Dostęp do wspólnych zasobów (pamięci zewnętrznych)
3. Mirroring dysków
4. Mechanizmy kontrolujące funkcjonowanie węzłów
5. Redundancja sieci łączących (dla 3 rodzajów sieci)
6. Redundancja zasilania

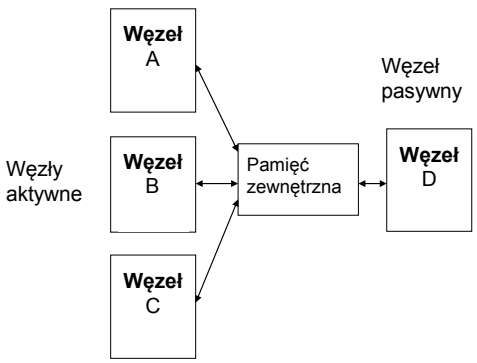
17.7.2 Redundancja węzłów / mocy obliczeniowej

Tryby pracy węzłów:

- Model klastra „aktywny - pasywny"
- Model klastra „aktywny - aktywny"



- Modele mieszane

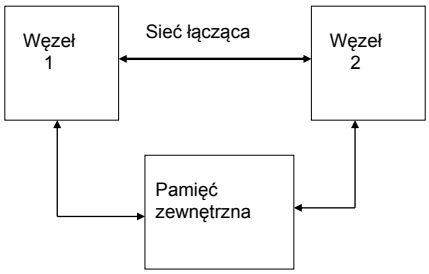


17.7.3 Warianty modelu mieszanego

- *N + 1* – jeden węzeł dodatkowy, o uniwersalnych możliwościach zastąpienia każdego z pozostałych.
- *N + M* – zwiększenie liczby dodatkowych węzłów do M w celu zwiększenia redundancji w przypadku dużej różnorodności usług świadczonych przez węzły.
- *N – to – N* – kombinacja modeli „aktywny -aktywny" oraz „N + M" –redystrybucja zadań węzła uszkodzonego na kilka innych węzłów aktywnych

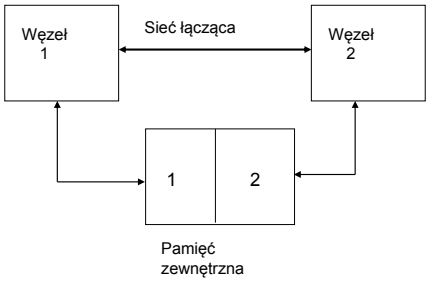
17.8 Warianty dostępu do wspólnych zasobów

17.8.1 Zasada „współdziel wszystko"



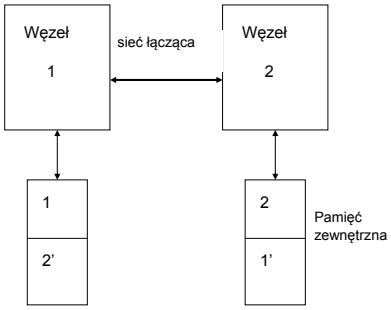
Wszystkie węzły mają dostęp do wspólnej pamięci zewnętrznej (kontrola dostępu przez mechanizmy blokad).

17.8.2 Zasada „nie współdziel nic"



Węzły nie współdzielą tego samego obszaru dysku –każdy ma dostęp do własnej części; po awarii czynny węzeł otrzymuje prawo dostępu do całego dysku.

17.8.3 Mirroring



Każdy węzeł zapisuje dane na własny dysk i automatycznie (pod kontrolą odpowiedniego oprogramowania) jest tworzona kopia tego zapisu na dyskach innych węzłów.

17.9 Podsumowanie

- Węzły – typowe serwery SMP „z półki” + pełna instancja Systemu Operacyjnego
- Sieci łączące – standardy: Gigabit Ethernet, Infiniband
- Komunikacja między węzłami (procesami) – przesył komunikatów
- Bardzo wysoka skalowalność
- Cele budowy: wysoka wydajność lub/i wysoka niezawodność, równoważenie obciążenia
- Korzystny wskaźnik: cena/wydajność

Pytania zamknięte

1. **Moc obliczeniowa komputerów wektorowych**

   - Zależy od liczby stopni potoku.
     Moc obliczeniowa nie jest zależna od liczby stopni potoku. Ta jedynie wpływa na ilosć rozkazów jakie mogą być wykonane w chwili czasu w jednostce potokowej.

   - **Jest odwrotnie proporcjonalna do długości taktu zegarowego**
     Tak, obliczamy ją wzorem $Przep = lim_{n\to\infty} \frac{n}{t_{start}+(n-1)\times\tau} = \frac{1}{\tau}$

   - Jest wprost proporcjonalna do długości taktu zegarowego
     Nie, patrz wyżej.

   - Zależy odwrotnie proporcjonalnie od liczby jednostek potokowych połączonych łańcuchowo.
     Nie, idea operacji wektorowej na komputerze wektorowym zakłada jedną jednostkę potokową. Ich zwiększenie nie powinno wpłynąć bezpośrednio na moc.

   - **Zmierza asymptotycznie do wartości maksymalnej wraz ze wzrostem długości wektora**
     Tak, istnieje pewna wartość maksymalna do której moc dąży logarytmicznie wraz ze wzrostem długości wektora.

   - Nie zależy od długości wektora
     Bzdura, patrz wyżej.

   - Zależy liniowo od długości wektora
     Bzdura, patrz wyżej.

2. **Komputery macierzowe**

   - **Mają w liście rozkazów m.in. rozkazy operujące na wektorach danych**
     Tak, te komputery są rozwinięciem komputerów wektorowych i muszą mieć rozkazy wektorowe. Komputery macierzowe posiadają po n *jednostek przetwarzających, które potrafią razem obliczyć* n *składowych wektora.*

   - Mają macierzowe potokowe układy arytmetyczne
     Nie, posiadają natomiast jednostki przetwarzające. Z kolei potokową jednostkę arytmetyczną posiadają komputery wektorowe.

- Mają w typowych rozwiązaniach zestaw pełnych procesów połączonych siecią połączeń
  Nie, w typowym rozwiązaniu jest jeden pełny procesor z wieloma jednostkami potokowymi, które są połączone siecią łączącą (statyczną lub dynamiczną). Sieć połączeń pełnych procków posiadają superkomputery z top500 (Nie jestem pewien tej odpowiedzi).

- **Wykonują synchroniczną operację wektorową w sieci elementów przetwarzających**
  Tak właśnie działają.

3. **Czy poniższa lista jest rosnąco uporządkowana według skalowalności:**

   - Systemy ściśle połączone, systemy ze wspólną pamięcią, systemy SMP
     Systemy SMP to cała kategoria systemów z pamięcią wspólną, z kolei systemy ściśle połączone i systemy ze wspólną pamięcią są ***równoznaczne*** *- są przeciwieństwem do systemów luźno powiązanych (z pamięcią rozproszoną).*

   - **Systemy ze wspólną magistralą, systemy wielomagistralowe, systemy z przełącznicą krzyżową**
     Są to systemy wieloprocesorowe (UMA) z pamięcią wspólną, patrz: Klasyfikacja 13.2.1
     - Systemy ze wspólna magistralą - *najprostsze i najmniej skalowalne*
     - Systemy wielomagistralowe - *szybsze i bardziej złożone, wciąż kiepsko skalowalne*
     - Systemy z przełącznicą krzyżową - *duża szybkość i złożoność obliczeniowa, trudne w rozbudowie*
     Ogółem jest to dolna półka tych systemów.

   - Systemy SMP, systemy z pamięcią wieloportową, systemy z przełącznicą krzyżową
     SMP to rodzaj architektury, z kolei w systemach UMA systemy z przełącznicą krzyżową są mniej skalowalne niż systemy z pamięcią wieloportową, patrz: Klasyfikacja 13.2.1.

   - NUMA, MPP, SMP
     MPP jest znacznie bardziej skalowalny niż SMP (pamięć rozproszona > pamięć wspólna). NUMA to systemy SMP z niejednorodnym dostępem do pamięć - są bardziej skalowalne niż zwykłe SMP (dzięki szybkiej pamięci lokalnej cache*), ale mniej niż MPP.*

   - **Systemy z pamięcią wspólną, systemy o niejednorodnym dostępie do pamięci, z pamięcią rozproszoną**
     *Dwa pierwsze to rodzaje systemów SMP. Najmniej skalowalne są systemy z pamięcią wspólną, domyślnie o jednorodnym dostępie do pamięci (UMA). Niejednorodny dostęp do pamięci wspólnej (NUMA) jest szybszy, ponieważ wykorzystuje pamięć lokalną procesora, węzły i katalogi. Mechanizm katalogów jest o wiele bardziej skalowalny niż mechanizm "*snoopingu*", wykorzystywany w UMA. Następnie system z pamięcią rozproszoną to MPP - system masywnie równoległy. Jest najbardziej skalowalny ze wszystkich.*

   - SMP, NUMA, klastry, UMA
     SMP jest najmniej skalowalny z wymienionych. UMA ma jednorodny dostęp do pamięci i jest mniej skalowalna od NUMA. Klastry są najbardziej skalowalne (nie wiem czy mniej lub bardziej do MPP).

   - **Systemy symetryczne, o niejednorodnym dostępie do pamięci, systemy z przesyłem komunikatów**
     Systemy symetryczne to SMP z jednorodnym dostępem do pamięci. Systemy SMP z niejednorodnym dostępem są bardziej skalowalne. Z kolei systemy z przesyłem komunikatów sugerują system MPP, z pamięcią rozproszoną - jest on najbardziej skalowalny.

4. **Przetwarzanie potokowe**

- Nie jest realizowane dla operacji zmiennoprzecinkowych
  Nie ma takiego ograniczenia. Przetwarzanie potokowe dotyczy optymalizacji czasu wykonywania rozkazów - podziału realizacji rozkazu na fazy. Owszem, dla argumentów zmiennoprzecinkowych mogą wystąpić problemy związane z czasem obliczeń (uniemożliwienie wykonania rozkazu w jednym takcie), co może zablokować napełnianie potoku, jednak nie uniemożliwia to zastosowania potoku.

- Nie jest realizowane w procesorach CISC
  Przetwarzanie potokowe znalazło zastosowanie głównie w architekturze RISC, jednak CISC też z niej korzysta. Przykłady: VAX 11/780 (CISC), Ultra SPARC III (RISC)

- **Daje przyspieszenie nie większe od liczby segmentów (stopni) jednostki potokowej**
  Tak, przyspieszenie jest stosunkiem czasu wykonywania n *rozkazów dla procesora niepotokowego oraz czasu dla procesora potokowego. W idealnym przypadku, gdy każdy stopień dzieli okres rozkazu po równo, a liczba rozkazów dąży do nieskończoności, stosunek ten jest równy P - ilości stopni.*

- W przypadku wystąpienia zależności między danymi wywołuje błąd i przerwanie wewnętrzne.
  Hm, dobre pytanie. Tak, zależności danych mogą wystąpić (zjawisko hazardu) i rozdupić program, ale po to właśnie istnieją mechanizmy by temu zapobiegać. Każda szanująca się architektura to potrafi: albo sprzętowo, albo na etapie kompilacji, która modyfikuje i optymalizuje program. A jeżeli po modyfikacji pewien rozkaz nie wykona się w jednym takcie, napełnianie potoku jest przerywane (ale błędu chyba nie wywala), patrz wyżej.

- Jest realizowane tylko dla operacji zmiennoprzecinkowych
  Pfff, no chyba nie XD Jest realizowane dla każdego rodzaju rozkazu.

5. **W procesorach superskalarnych**

- **Liczba rozkazów, które procesor może wykonać w 1 takcie zależy od liczby jednostek potokowych w procesorze**
  Procesory superskalarne posiadają wiele jednostek potokowych, które są konieczne by móc wykonywać wiele rozkazów w jednym takcie. Od ich liczby zależy owa liczba rozkazów.

- Liczba rozkazów, które procesor może wykonać w jednym takcie, zależy od liczby stopni potoku.
  Nie, liczba stopni potoku mówi, na ile części dzieli się dany rozkaz w tej jednostce potokowej. One umożliwiają wykonanie wielu rozkazów w jednej jednostce czasu, jednak nie przekłada się to bezpośrednio na liczbę rozkazów, ze względu na zawikłania czasowe, oraz nie jest to idea procesora superskalarnego.

- Liczba rozkazów pobieranych z pamięci, w każdym takcie musi przekraczać liczbę jednostek potokowych
  Liczba pobranych rozkazów powinna być co najmniej równa ilości jednostek potokowych.

- **Liczba rozkazów, które procesor może wykonać w taktach zależy od liczby jednostek potokowych w procesorze**
  Tak, patrz pierwsza odpowiedź.

6. **Systemy SMP**

- Wykorzystują protokół MESI do sterowania dostępem do wspólnej magistrali
  *Ten protokół wykorzystują systemy **UMA** (podkategoria systemów SMP) ze wspólną magistralą w celu zapewnienia spójności pamięci podręcznych (*snooping*). Mogą też używać katalogów, ale podkategoria **NUMA** wykorzystuje wyłącznie katalogi.*

- Posiadają skalowalne procesory
  SMP należy do systemów wieloprocesorowych, ale te nie muszą być skalowalne.

- Posiadają pamięć fizycznie rozproszoną, ale logicznie wspólną
  Nie, pamięć jest fizycznie wspólna. Fizycznie rozproszoną pamięć posiadają systemy MPP.

7. **Komputery wektorowe**

- Posiadają jednostki potokowe o budowie wektorowej
  Nie, posiadają potokowe jednostki arytmetyczne, które nie są wektorowe.

- **Posiadają w liście rozkazów m.in. rozkazy operujące na wektorach danych**
  Jak najbardziej, nie mogłyby się bez tego obejść.

- **Wykorzystują od kilku do kilkunastu potokowych jednostek arytmetycznych**
  Tak, tych jednostek może być wiele, można to zauważyć na przykładzie komputera Cray-1 (wykład 7-8, slajd 31)

- Posiadają listę rozkazów operujących wyłącznie na wektorach
  Zdecydowanie nie. Owszem, te komputery posiadają rejestry wektorowe i wektorowe jednostki zmiennoprzecinkowe, ale nie jest to wszystko. Mają również normalne rejestry, adresację, jednostki skalarne i możliwość wykonywania na nich operacji.

8. **Procesory wektorowe**

- **Mogą być stosowane w systemach wieloprocesorowych**
  Domyślnie procesory wektorowe mogą pracować pojedynczo, ale mogą być częścią takiego systemu. Poza tym nie znalazłem nic, co by temu przeczyło. Jest też np. CUDA - architektura wielordzeniowych procesorów graficznych. Sama architektura SIMD działa na wielu procesorach.

- Mają listę rozkazów operującą jedynie na wektorach
  Nie, posiadają też m.in. potokowe jednostki arytmetyczne oraz jednostki skalarne, do operowania na zwykłych liczbach.

- **Mają moc kilka razy większą od procesorów skalarnych**
  Tak, przyspieszenie jest ilorazem czasu wykonywania na procesorze niewektorowym do czasu wykonywania na procesorze wektorowym. Np. dla rozkazu dodawania n *wektorów przyspieszenie wyliczane jest wg wzoru* $a = \frac{15\tau n}{t_{start}+(n-1)\tau}$*, gdzie przy* n *dążącym do nieskończoności* a *jest równe 15.*

9. **Systemy MPP są zbudowane z węzłów którymi mogą być**

   - **Systemy SMP**
     Węzłami mogą być zarówno systemy UMA, jak i NUMA. Ponadto dopuszcza się zwykłe procesory z pamięcią operacyjną. Patrz: organizacja MPP 16.2. Są to ***jedyne*** *możliwe rodzaje węzłów.*

   - Klastry
     Patrz wyżej.

   - Konstelacje
     Patrz wyżej.

   - **Systemy NUMA**
     Patrz wyżej.

   - **Procesory**
     Patrz wyżej.

10. **W architekturze NUMA**

    - **Dane są wymieniane między węzłami w postaci linii pamięci podręcznej (PaP)**
      Tak, każdy procesor / węzeł posiada swoją własną szybką pamięć podręczną. Pamięć ta jest publiczna - inne procesory mają do niej dostęp, ale wymiana informacji na linii moja pamięć - inny procesor *jest znacznie wolniejsza niż* procesor - jego pamięć*.*

    - Spójność PaP węzłów jest utrzymywana za pomocą protokołu MESI
      Protokół MESI jest wykorzystywany w architekturze ***UMA*** *do* snoopingu - *zapewnienia spójności pamięci podręcznych procków.*

    - Czas dostępu do pamięci lokalnej w węźle jest podobny do czasu dostępu do pamięci nielokalnej
      Odwołanie do nielokalnej pamięci są znacznie wolniejsze niż do lokalnej, ok. 10-krotnie bardziej. Dotyczy to głównie architektury NC-NUMA, patrz: rodzaje systemów NUMA 14.1

    - **Czas zapisu danych do pamięci nielokalnej może być znacznie dłuższy od czasu odczytu z tej pamięci**
      Patrz wyżej.

    - **Każdy procesor ma dostęp do pamięci operacyjnej każdego węzła**
      Patrz wyżej.

    - Procesy komunikują się poprzez przesył komunikatów
      Przesył komunikatów występuje w systemach MPP, gdzie pamięć jest rozproszona fizycznie i logicznie. W NUMA jest fizycznie rozproszona między węzłami (do przesyłu informacji wykorzystywana jest sieć łączącą węzły), ale stanowi logicznie jedną całość.

    - **Pamięć operacyjna jest rozproszona fizycznie pomiędzy węzłami, ale wspólna logicznie**
      Patrz wyżej.

11. **Mechanizmy potokowe stosowane są w celu**

- Uszeregowania ciągu wykonywanych rozkazów
  Nie, zupełnie nie o to chodzi. Ciąg może zostać uszeregowany przez kompilator w celu optymalizacji. Jednak celem tego mechanizmu jest zrównoleglenie wykonywania rozkazów → zmiana kolejności ich realizacji nie jest założeniem.

- **Uzyskania równoległej realizacji rozkazów**
  No tyć. Potoki umożliwiają realizację wielu rozkazów jednocześnie dzieląc jednostkę centralną na wg stopni, jak np. pobranie rozkazu i wykonania rozkazu. Dzięki temu dwa rozkazy mogą wykonywać się jednocześnie, oba w innych fazach (jednostkach czasu).

- **Przyspieszenia realizacji rozkazów**
  Tak, to główny cel. Umożliwienie wykonania rozkazów umożliwia przyspieszenie, które oblicza się jako stosunek czasu wykonywania rozkazów w procesorze niepotokowym do czasu realizacji w procesorze potokowym. W idealnym przypadku jest ono równe P *- ilości podziałów / stopni / faz / zwał jak zwał.*

12. **Protokół MESI**

- Jest wykorzystywany do sterowania dostępem do magistrali w systemie SMP
  Protokół MESI wykorzystywany jest do zapewniania spójności pamięci podręcznych cache *w architekturze SMP (*snooping*), a dokładniej w UMA. NUMA korzysta tylko z katalogów.*

- **Zapewnia spójność pamięci cache w systemie SMP**
  Do tego właśnie służy.

- Służy do wymiany komunikatów w systemie MPP
  Patrz wyżej.

- Chroni przed hazardem w procesorach superskalarnych
  Patrz wyżej.

13. **Mechanizm skoków opóźnionych**

- **Polega na opóźnianiu wykonywania skoku do czasu wykonania rozkazu następnego za skokiem**
  Tak, cały ten mechanizm sprowadza się do opóźnienia efektu skoku o jeden rozkaz. Zapewnia to, że rozkaz następny po skoku zawsze będzie wykonywany w całości.

- Wymaga wstrzymania potoku na jeden takt.
  Nie, mechanizm potoków nie musi być wstrzymywany. Mechanizm ten zmienia postać programu w trakcie kompilacji, ale na samą realizację potoku nie ma wpływu (afaik, not sure).

- Powoduje błąd na końcu pętli
  Pfff, jak programista ssie patę to tak, jednak w założeniu tak się nie dzieje.

- **Wymaga umieszczenia rozkazu NOP za rozkazem skoku lub reorganizacje programu**
  Tak, mechanizm sprowadza się do tego, i tylko do tego, patrz pierwsza odpowiedź.

14. **Charakterystyczne cechy architektury MPP**

- Spójność pamięci podręcznej wszystkich węzłów
  Spójność wręcz nie powinna być zapewniana, każdy element są swoją własną, odrębną pamięć.

- **Fizycznie rozproszona PaO**
  Tak, jest fizycznie i logicznie rozproszona.

- Fizycznie rozproszona PaO, ale logicznie wspólna
  Nie, taka jest cecha systemów NUMA.

- **Przesył komunikatów między procesorami**
  Tak, to metoda synchronizacji wykonywania zadań.

- Niska skalowalność
  Nie, jest przeogromna.

- Jednorodny dostęp do pamięci wszystkich węzłów
  Nie, pamięć jest rozproszona.

15. **Jak można ominąć hazard danych**

- Poprzez rozgałęzienia
  Nie, rozgałęzienie to po prostu instrukcje typu IF, które tworzą takie rozgałęzienia. Mechanizm przewidywania rozgałęzień jest stosowany do usuwania hazardu sterowania związanego ze skokami i rozgałęzieniami.

- Poprzez uproszczenie adresowania - adresowanie bezpośrednie.
  Bullshit. Nie wiem w czym miało by pomóc uproszczenia adresowania, poza pójściem w stronę RISCu, ale na hazard to nie pomoże. Tym można tylko skrócić czas odwołania się do danych.

- **Przez zamianę rozkazów**
  Tak, i na tym polega mechanizm skoków opóźnionych, które mogą program zmodyfikować (dodać rozkaz NOP) albo zoptymalizować, właśnie zamieniają rozkazy kolejnością.

16. **Cechy architektury CISC**

- Czy może być wykonana w VLIW
  Nie, architektura VLIW dotyczy mikroprocesorów i miała na celu jak największe zmniejszenie jednostki centralnej i jej rozkazów (RISC).

- **Czy występuje model wymiany danych typu pamięć - pamięć**
  Tak, posiada również niewielką ilość rejestrów.

- Jest mała liczba rozkazów
  Nie, w tej architekturze jest PEŁNA (complex) lista rozkazów. Niektóre z zaawansowanych pleceń nawet nie były wykorzystywane, i bum! tak powstał RISC.

17. **Cechy architektury RISC**

- **Czy występuje model wymiany danych typu rej-rej**
  Tak, a komunikacja z pamięcią operacyjną odbywa się wyłącznie za pomocą rozkazów LOAD i STORE.

- **Jest mała liczba trybów adresowania**
  Tak, raptem 4 w procesorze RISC I podczas gdy CISCi mogą mieć ich kilkanaście, w tym takie bardzo złożone.

- Jest wykonywanych kilka rozkazów w jednym takcie
  Fałsz. Prawdziwe wykonywanie wielu rozkazów w jednym takcie wymaga superskalarnosci - wielu jednostek potokowych. Cechą architektury RISC jest potokowość, ale pojedyncza.

- **Jest wykonywanych kilka rozkazów w jednym takcie (w danej chwili czasu)**
  Chodzi o przetwarzanie potokowe. Tu jest haczyk - pierwszy procesor RISC I (1980) stawiał sobie za cel wykonanie jednego rozkazu w jednym takcie *i dokładnie tak brzmiało jego założenie projektowe. Jednak jego fizyczna realizacja (1982) posiadała dwustopniowy potok. Również w wykładach jako cecha tej architektury jest napisane "Intensywne wykorzystanie przetwarzania potokowego", co odnosi się do faktu, że obecnie nie ma procesora typu RISC, który go nie ma. Wg mnie prawda.*

- Jest wykonywanych kilka instrukcji procesora w jednym rozkazie asemblerowym
  Nic mi na ten temat nie wiadomo. Brzmi jednak zbyt hardo i odlegle od tematu zmniejszania ilości rozkazów.

- **Układ sterowania w postaci logiki szytej**
  Tak.

18. **Przepustowość (moc obliczeniowa) dużych komputerów jest podawana w:**

- **GFLOPS**

- Liczbie instrukcji wykonywanych na sekundę

- **Liczbie operacji zmiennoprzecinkowych na sekundę**

- Mb/sek
  To jest do zapamiętania na prostu - takie są standardy

19. **Podstawą klasyfikacji Flynna jest**

- Liczba jednostek przetwarzających i sterujących w systemach komputerowych
- Protokół dostępu do pamięci operacyjnej
- Liczba modułów pamięci operacyjnej w systemach komputerowych
- **Liczba strumieni rozkazów i danych w systemach komputerowych**
  To po prostu należy zapamiętać. ***Kryterium klasyfikacji Flynna jest*** **liczba strumieni rozkazów** ***oraz*** **liczba strumieni danych** ***w systemie komputerowym. NIC WIĘCEJ, NIC MNIEJ.***
  Albo inaczej: $Liczba\_strumieni \times (rozkazow + danych)$

20. **Rozkazy wektorowe mogą być realizowane przy wykorzystaniu**

- **Macierzy elementów przetwarzających**
  Tak, komputery macierzowe operują na rozkazach wektorowych.
- Zestawu procesorów superskalarnych
  Procesory superskalarne w założeniu nie posiadają rozkazów wektorowych.
- **Technologii MMX**
  Tak, jest to pochodna technologia modelu SIMD, wykonuje operacje na krótkich wektorach (64-bit)
- Sieci połączeń typu krata
  Jest to sieć połączeń, która łączy jednostki przetwarzające w komputerze macierzowym. Raczej na wektorach na część komputera nie działa.
- **Potokowych jednostek arytmetycznych**
  Tak, takie znajdują się w komputerach wektorowych.

21. **Architektura superskalarna**

- Dotyczy systemów SMP
  Zdecydowanie nie tylko. Architektura superskalarna wymaga mechanizmu potokowego, czyli dotyczy głównie architektury RISC.
- Wymaga zastosowania protokołu MESI
  Nie, architektura superskalarna wymaga jedynie zastosowania co najmniej dwóch jednostek potokowych.
- **Umożliwia równoległe wykonywanie kilku rozkazów w jednym procesorze**
  Tak, i taki jest cel jej istnienia. Umożliwia to mechanizm potokowy.
- Wywodzi się z architektury VLIW
  Wręcz odwrotnie, to VLIW wykorzystuje architekturę superskalarną na której opiera swój podział rozkazów na paczki.

22. **Klastry**

- Mają średnią skalowalność
  Mają największą skalowalność ze wszystkich poznanych systemów. Łatwiej go rozszerzać niż MPP, ponieważ jest jednym wielkim systemem komputerowym.

- Wykorzystują model wspólnej pamięci
  Nie, jest rozproszona fizycznie i logicznie.

- **W węzłach mogą wykorzystywać systemy SMP**
  Tak, serwery SMP są jednymi z dopuszczalnych węzłów. Drugimi są pełne komputery PC. Patrz: węzły w klastrach 17.3.1

- **Do komunikacji między procesami wykorzystują przesył komunikatów**
  Tak, bo jest efektownym rozwiązaniem, i tylko go wykorzystują.

- Wykorzystują przełącznicę krzyżową jako sieć łączącą węzły
  Nie, ona jest wykorzystywana tylko w systemach UMA, gdzie pamięć wspólna jest fizycznie jednorodna.

- **W każdym węźle posiadają pełną instalację systemu operacyjnego**
  Tak, węzłem musi być albo serwer SMP, albo PC, które muszą mieć swoje systemy operacyjne.

23. **Pojęcie równoległości na poziomie rozkazów**

- Dotyczy architektury MIMD
  Nie, ten rodzaj równoległości dotyczy mechanizmów potokowych (CISC i RISC), architektury superskalarnej oraz VLIW.

- **Odnosi się m.in. do przetwarzania potokowego**
  Tak, ideą mechanizmu potoków jest zrównoleglenie rozkazów i możliwość wykonywania wielu z nich w tej samej chwili czasu.

- Dotyczy architektury MPP
  Nie, patrz wyżej.

- **Dotyczy m.in. architektury superskalarnej**
  Tak, patrz wyżej.

24. **Systemy wieloprocesorowe z pamięcią wspólną**

- Zapewniają jednorodny dostęp do pamięci
  Nie, bo NUMA nie zapewnia. Jednorodny dostęp występuje wtedy, gdy procesor ma dostęp wyłącznie do swojej pamięci podręcznej. Niejednorodny wtedy, gdy ma dostęp również do pamięci nielokalnej (pamięci podręcznej innych procesorów.)

- **Mogą wykorzystywać procesory CISC**
  Nie ma takiego ograniczenia.

- **Są wykorzystywane w klastrach**
  Tak, węzły w postaci serwerów SMP.

- Wykorzystują przesył komunikatów między procesorami
  Nie, wykorzystują protokół MESI do snoopingu *oraz katalogi w celu zapewnienia spójności pamięci podręcznej.*

- **Wykorzystują katalog do utrzymania spójności pamięci podręcznych**
  Patrz wyżej.

25. **Hazard danych**

- **Czasami może być usunięty przez zmianę kolejności wykonania rozkazów**
  Tak, służy do tego mechanizm skoków opóźnionych, który odbywa się na poziomie kompilacji programu.

- Nie występuje w architekturze superskalarnej
  Występuje wszędzie tam gdzie jest potokowe przetwarzania rozkazów.

- Jest eliminowany przez zastosowanie specjalnego bitu w kodzie program
  Nic mi o tym nie wiadomo. Pewne dodatkowe bity są wykorzystywane w mechanizmie przewidywania rozgałęzień, który służy do eliminacji hazardu, jednak on to odbywa się PRZED realizacją programu i sprowadza się do zmiany kolejnosci wykonywania rozkazów przez kompilator. Nic nie dodaje do treści programu.

- Może wymagać wyczyszczenia potoku i rozpoczęcia nowej (...)
  Nie wiem jak hazard danych może czegokolwiek wymagać skoro jest zjawiskiem ubocznym i je eliminujemy. Sprzętowa i programowa eliminacja hazardu jedynie może doprowadzić do ***wstrzymania*** *napełniania potoku.*

26. **Przetwarzanie wielowątkowe**

- **Zapewnia lepsze wykorzystanie potoków**
  Tak, ma na celu minimalizację strat cykli w trakcie realizacji wątku, jakie mogą powstać na wskutek:
  - chybionych odwołań do pamięci podręcznej;
  - błędów w przewidywaniu rozgałęzień;
  - zależności między argumentami

- **Minimalizuje straty wynikające z chybionych odwołań do pamięci podręcznej**
  Tak, patrz wyżej.

- **Wymaga zwielokrotnienia zasobów procesora (rejestry, liczniki rozkazów, itp.)**
  Niestety tak, jest to warunek sprzętowej realizacji wielowątkowości.

- Nie może być stosowane w przypadku hazardu danych
  Nie, hazard danych wynika z zależności między argumentami, które są naturalnym ryzykiem przy stosowaniu mechanizmu potokowego. Nie powinny być blokowane z tego powodu, tym bardziej, że wielowątkowość ma dodatkowo chronić liczbę cykli przed zgubnym wpływem hazardu.

27. **Okna rejestrów**

- Chronią przez hazardem danych
  Lolnope, od tego są mechanizmy skoków opóźnionych i przewidywania rozgałęzień. Okno rejestrów zapewnia ciągłe i optymalne wykonywanie procedur.

- **Minimalizują liczbę odwołań do pamięci operacyjnej przy operacjach wywołania procedur**
  Tak, dokładnie do tego one służą. Rejestr niski procedury A staje się rejestrem wysokim procedury B itd. Innymi słowy, procedura A wywołuje procedurę B, i tak dalej. I po coś w tym wszystkim są rejestry globalne.

- Są charakterystyczne dla architektury CISC
  Nie, zostały zaprojektowane specjalnie dla architektury RISC. Jako pierwszy posiadał je procesor RISC I.

- Są zamykane po błędnym przewidywaniu wykonania skoków warunkowych.
  W mechanizmie prognozowania rozgałęzień jest możliwość błędnego przewidywania. Jednak błędna prognoza powoduje tylko zmianę strategii (przewidywanie wykonania lub niewykonania), a nie zamykanie okna.

- **Są przesuwane przy operacjach wywołania procedur**
  Tak, z każdą nową wywołaną procedurą okno rejestrów przesuwane jest w dół (ze 137 do 0)

- Są przesuwane przy wystąpieniu rozkazów rozgałęzień.
  W architekturze SPARC okno jest przesuwane rozkazami SAVE i RESTORE, na życzenie programisty, a nie na wskutek rozkazów warunkowych.

28. **Tablica historii rozgałęzień**

- **Zawiera m.in. adresy rozkazów rozgałęzień**
  Tak, tablica ta zawiera bit ważności, adres rozkazu rozgałęzienia*, bity historii oraz* adres docelowy rozgałęzienia*.*

- **Pozwala zminimalizować liczbę błędnych przewidywań rozgałęzień w zagnieżdżonej pętli**
  Tak, z tego co wiem jest strategią dynamiczną i najbardziej optymalną ze wszystkich - skończony automat przewidywania rozgałęzień oparty na tej tablicy (z dwoma bitami historii) może być zrealizowany na dwóch bitach.

- Nie może być stosowana w procesorach CISC
  Ten mechanizm służy zabezpieczeniu przed hazardem, który występuje w przetwarzaniu potokowym, a z tego korzystają zarówno CISC jak i RISC.

- Jest obsługiwana przez jądro systemu operacyjnego
  Chyba nie, ten mechanizm znajduje się w sprzęcie procesora.

29. **Rozkazy wektorowe**

- Nie mogą być wykonywane bez użycia potokowych jednostek arytmetycznych
  Mogą. Komputery macierzowe ich nie posiadają i wykonują rozkazy wektorowe sprawnie.

- **W komputerach wektorowych ich czas wykonania jest wprost proporcjonalny do długości wektora**
  Tak, na przykładzie rozkazu dodawania wektorów widać, że czas rośnie równomiernie wraz z ilością elementów wektora.
  $t_w = t_{start} + (n-1) \times \tau$

- **Są charakterystyczne dla architektury SIMD**
  Tak, z niej się zrodziły, tak samo jak m.in. technologie MMX i SSE.

- Są rozkazami dwuargumentowymi i w wyniku zawsze dają wektor
  Nie, mogą operować na 1 argumencie na przykład. Rozkaz może być też 3 argumentowy, jak rozkaz dodawania VADD. Pierwszym argumentem jest rejestr docelowy, zawartość pozostałych dwóch jest dodana.

30. **Model SIMD**

- Był wykorzystywany tylko w procesorach macierzowych
  Nie, o niego oparte są również m.in. procesory wektorowe, GPU, technologie MMX oraz SSE. Nie, był również wykorzystywany w komputerach wektorowych, rozszerzeniach SIMD oraz GPU.

- **Jest wykorzystywany w multimedialnych rozszerzeniach współczesnych procesorów**
  Tak, multimedialne rozszerzenie służą do działania na wielu rejestrach jednocześnie, co umożliwiają m.in. technologie MMX i SSE.

- **Jest wykorzystywany w heterogenicznej architekturze PowerXCell**
  Być może XD

- **Zapewnia wykonanie tej samej operacji na wektorach argumentów**
  Znaczenie: Single Instruction Multiple Device. *Wykonuje jedną instrukcję na wielu urządzeniach.*

- **Jest podstawą rozkazów wektorowych**
  Pod ten model są zaprojektowane rozkazy wektorowe - jedno instrukcja, wiele urządzeń.

- Jest podstawą architektury procesorów superskalarnych
  Nie, jest wykorzystywany w rozszerzeniach wektorowych, ale nie jest podstawą.

31. **Przesył komunikatów**

- **Ma miejsce w systemach MPP**
  Tak, w MPP oraz w klastrach.

- W systemach MPP II-giej generacji angażuje wszystkie procesory na drodze przesyłu
  Nie, bo w II generacji są routery węzłów pośredniczących, które nie angażują procesorów. W I generacji wszystkie komputery były zaangażowane, ponieważ rolę routera pełnił sam procesor.

- **Ma miejsce w klastrach**
  Tak, patrz wyżej.

32. **Cechami wyróżniającymi klastry są**

- **Niezależność programowa każdego węzła**
  Tak, bo każdy węzeł na swój osobny system operacyjny.

- Fizycznie rozproszona, ale logicznie wspólna pamięć operacyjna
  Jest fizycznie i logicznie rozproszona.

- Nieduża skalowalność
  Kurwa, nie XD Klastry mają arcydupną skalowalność.

- **Na ogół duża niezawodność**
  Tak, po to się je buduje i na ogół ją mają. Redundancja węzłów, mirroring dysków, kontrola funkcjonowania węzłów. Patrz: Niezawodność klastrów 17.7

33. **Systemy wieloprocesorowe z pamięcią rozproszoną (MPP)**

- **Wyróżniają się bardzo dużą skalowalnością**
  Posiadają ogromną skalowalność, tylko klastry mają lepszą.

- Są budowane z węzłów, którymi są klastry
  Węzłami mogą być tylko systemy UMA i NUMA oraz zwykłe pojedyncze procesory.

- Realizują synchronicznie jeden wspólny program
  Nie muszą być synchronicznie wykonywane (są synchronizacje, ale jeden węzeł może realizować szybciej pewne części programu)

- Wymagają zapewnienia spójności pamięci podręcznych pomiędzy węzłami
  Wręcz nie powinny jej zapewniać, każdy węzeł pracuje osobno.

- **Wymianę danych i synchronizację procesów w węzłach realizują poprzez przesył komunikatów.**
  Tak, pamięć jest logicznie rozproszona, a węzły są fizycznie oddzielne, do komunikacji wystarcza tylko przesył komunikatów.

- W większości przypadków wykorzystują nietypowe, firmowe rozwiązania sieci łączących węzły systemu.
  Nie wiem czy w większości przypadków, ale na slajdach większość jest poświęcona typowym rozwiązaniom sieciowym (hipersześcian, krata, torus, przełącznica krzyżowa), a custom network *to tylko dodatek.*

- Wykorzystują katalog do utrzymania spójności pamięci węzłów systemu.
  Między węzłami w MPP wykorzystuje się przesył komunikatów. Katalogi sa wykorzystywane przez systemy UMA i NUMA, które mogą być węzłami w MPP. Jednak komunikacja między samymi węzłami ich nie wykorzystuje.

34. **Problemy z potokowym wykonywaniem rozkazów skoków (rozgałęzień) mogą być wyeliminowane lub ograniczone przy pomocy**

- Zapewnienia spójności pamięci podręcznej
  Nie, to problem komputerów wieloprocesorowych.

- **Tablicy historii rozgałęzień**
  Tak, to najprawdopodobniej najlepszy służący ku temu mechanizm. Stara się przewidywać czy skok będzie wykonany bądź nie, wykorzystuje do tego kilka strategii.

- Techniki wyprzedzającego pobrania argumentu
  Nie, ten mechanizm służy do eliminacji hazardu danych - zależności między argumentami.

- **Wystawienia do programu rozkazów typu „nic nie rób”**
  Tak, tym rozkazem jest NOP *i jest wstawiany przez mechanizm skoków opóźnionych, który służy do zabezpieczania potoku.*

- Protokołu MESI
  Nie, on jest od zapewnienia spójności pamięci wspólnej czy jakoś tak.

- **Wykorzystania techniki skoków opóźniających**
  Tak, umożliwiają ona modyfikację programu (wstawienie rozkazu NOP), albo jego optymalizację (zamiana kolejności wykonywania rozkazów.) Mechanizm ten opóźnia efekt skoku o jeden rozkaz, co zapewnia, że rozkaz po skoku będzie w całości wykonany.

- Technologii MMX
  Polega zupełnie na czym innym.

35. **W architekturze ccNUMA**

- **Każdy procesor ma dostęp do pamięci operacyjnej każdego węzła**
  Tak, ponieważ architektura NUMA opiera się o niejednorodny dostęp do pamięci - każdy procesor ma pełny dostęp do pamięci lokalnej oraz nielokalnej, czyli pamięci podręcznych wszystkich innych procesorów.

- Spójność pamięci pomiędzy węzłami jest utrzymywana za pomocą protokołu MESI
  Nie, nie jest potrzebna spójność pamięci, ponieważ każdy procesor odczytuje potrzebne mu zmienne itp. pośrednio przez katalog.

- **Dane są wymieniane między węzłami w postaci linii pamięci podręcznej**
  Każda linia posiada pewną liczbę bajtów, które inne procesory mogą pobierać. Możliwe, że i katalogi korzystają z wymiany danych poprzez linie.

- **Pamięć operacyjna jest fizycznie rozproszona pomiędzy węzłami, ale wspólna logicznie**
  Dokładnie tak.

36. **Dla sieci systemowych (SAN) są charakterystyczne**

- **Przesył komunikatów w trybie zdalnego DMA**
  Tak, bo przesyłamy dane między prockami, a DMA wykonuje to najszybciej.

- **Bardzo małe czasy opóźnień**
  Tak, rzędu pojedynczych mikrosekund.

- Topologia typu hipersześcian
  Bullshit, sieć jest taka jak topologia systemu, nie buduje się osobnej topologii.

- Niska przepustowość
  Noelonie, do pamięci potrzebna jest duża przepustowość, bo przechodzi przez nią dużo danych.

37. **W systemach wieloprocesorowych katalog służy do**

- Śledzenia adresów w protokole MESI
  Nie no kurwa, szanujmy się, katalogi powstały po to by wyprzeć protokół MESI.

- Sterowania przesyłem komunikatów
  Nie, w UMA i NUMA nie ma przesyłu komunikatów, to jest w MPP i klastrach.

- **Utrzymania spójności pamięci w systemach o niejednorodnym dostępie do pamięci**
  No tak, patrz poprzednie pytania. To jest problem z jakimi borykają się UMA i NUMA. Katalog można stosować do jego rozwiązania w obu architekturach.

- **Realizacji dostępu do nielokalnych pamięci w systemach NUMA**
  Katalogi i węzły stanowią mechanizm do wymiany informacji między prockami i ich pamięciami.

38. **W procesorach superskalarnych**

- **Jest możliwe równoległe wykonywanie kilku rozkazów w jednym procesorze (rdzeniu)**
  Tak, własnie taka jest idea stworzenia procesorów superskalaranych, by móc w jednym takcie wykonać > 1 liczby instrukcji. Zapewnia to niepojedyncza liczba jednostek potokowych.

- **Rozszerzenia architektury wykorzystujące model SIMD umożliwiają wykonanie rozkazów wektorowych**

- Nie występuje prawdziwa zależność danych
  Niestety występuje, i prawdę mówiąc, występuje tutaj każdy rodzaj zależności między rozkazami: prawdziwa zależność danych, zależność wyjściowa oraz antyzależność.

- **Mogą wystąpić nowe formy hazardu danych: zależności wyjściowe między rozkazami oraz antyzależności**
  Tak, patrz wyżej.

39. **Efektywne wykorzystanie równoległości na poziomie danych umożliwiają**

- **Komputery wektorowe**
- **Komputery macierzowe**
- **Klastry**
- **Procesory graficzne**
- **Rozszerzenia SIMD procesorów superskalarnych**

  Ogółem zastosowanie tej równoległości jest możliwe gdy mamy do czynienia z wieloma danymi, które mogą być przetwarzane w tym samym czasie. A grafika, wektory, macierze itp. do takich należą.

40. **Wielowątkowość współbieżna w procesorze wielopotokowym zapewnia**

- **Możliwość wprowadzenia rozkazów różnych wątków do wielu potoków**
  Tak, jest to charakterystyczna cecha tego typu wielowątkowości. Z kolei wielowątkowości grubo- i drobnoziarniste umożliwiają wprowadzenie do wielu potoków wyłącznie *jednego wątku (w jednym takcie!)*
- **Realizację każdego z wątków do momentu wstrzymania któregoś rozkazu z danego wątku**
  Tak, wątek jest realizowany do momentu wstrzymania rozkazu. Tę samą cechę posiada wielowątkowość gruboziarnista. Z kolei wielowątkowość drobnoziarnista w kolejnych taktach realizuje naprzemiennie rozkazy kolejnych wątków.
- Przełączanie wątków co takt
  Nie, to umożliwia tylko wielowątkowość drobnoziarnista.
- Automatyczne przemianowanie rejestrów
  Głowy nie dam, ale chyba żadna wielowątkowość nie zapewnia automatycznego przemianowania.

41. **Architektura CUDA**

- **Umożliwia bardzo wydajne wykonywanie operacji graficznych**
  Tak, ta architektura jest rozwinięciem mechanizmów wektorowych oraz macierzowych i jest przeznaczona specjalnie dla przetwarzania grafiki.

- **Stanowi uniwersalną architekturę obliczeniowa połączoną z równoległym modelem programistycznym**
  Tak, pomimo specjalizacji graficznej, architektura ta jest uniwersalna i zdolna do wszystkiego. Procesory posiadają uniwersalne programy obliczeniowe, a CUDA posiada model programistyczny (oraz podział programu na 5 faz). Składa się on z:
  - Kompilatora NVCC;
  - Podział programu na kod wykonywany przez procesor (host code) oraz kartę graficzną (kernel);
  - Realizacja obliczeń równoległych wg modelu SIMT (Single Instruction Multiple Threading)

- **Realizuje model obliczeniowy SIMT**
  *Tak, patrz wyżej. Działanie: wiele niezależnych wątków wykonuje tę samą operację. Architektura posiada również mechanizm synchronizacji wątków (*barrir synchronization*) dla komunikacji oraz współdzielona pamięć.*

- Jest podstawą budowy samodzielnych, bardzo wydajnych komputerów
  Komputery CUDA nie są ogólnego zastosowania, tylko do ogólnych problemów numerycznych. Na pewno nie są podstawą, bo np. komputer ... (dokończyć by trza)

42. **Spójność pamięci podręcznych w procesorze wielordzeniowym może być m.in. zapewniona za pomocą**

- Przełącznicy krzyżowej
  Nie, to tylko jakieś rozwiązanie sieci połączeń.

- Katalogu
  ie,to bardziej zaawansowany shit służący do komunikacji.

- **Protokołu MESI**
  Tak, i tylko to do tego służy.

- Wspólnej magistrali
  Nie, ona służy do komunikacji i synchronizacji (?) dostępu do pamięci.

43. **Metoda przemianowania rejestrów jest stosowana w celu eliminacji:**

- Błędnego przewidywania rozgałęzień
  Nie, do tego służy m.in. tablica historii rozgałęzień.

- Chybionego odwołania do pamięci podręcznej
  Nie, to jest problem architektury VLIW i eliminuje się do przez przesunięcie rozkazów LOAD jak najwyżej, tak aby zminimalizować czas ewentualnego oczekiwania

- Prawdziwej zależności danych
  Nie, od tego jest metoda wyprzedzającego pobierania argumentu.

- **Zależności wyjściowej między rozkazami.**
  Tak, ta metoda eliminuje powyższy i poniższy problem. Polega na dynamicznym przypisywaniu rejestrów do rozkazów.

- **Antyzależności między rozkazami**
  Patrz wyżej.

44. **W systemach wieloprocesorowych o architekturze CC-NUMA**

- **Spójność pamięci wszystkich węzłów jest utrzymywana za pomocą katalogu**
  Tak, w NUMA do zachowania spójności danych można stosować wyłącznie katalogi (a np. protokołu MESI nie).

- **Pamięć operacyjna jest rozproszona fizycznie pomiędzy węzłami, ale wspólna logicznie**
  Dokładnie tak.

- Każdy procesor ma bezpośredni dostęp do pamięci operacyjnej każdego węzła
  Nie, dostęp jest pośredni. Pamięć w CC-NUMA jest fizycznie rozproszona, więc coś musi pośredniczyć w tej wymianie danych. Służy do tego mechanizm katalogów i węzłów. Procesor zgłasza zapotrzebowanie na linię pamięci do katalogu i ją później otrzymuje.

- **Dane są wymieniane między węzłami w postaci linii pamięci podręcznej**
  Tak, ponieważ w architekturze NUMA wymiana rekordów pamięci następuje z użyciem całej linii. Np. jeśli procek chce pobrać jednego floata 4-bajtowego, a linia ma 16 bajtów, to musimy pobrać całą linię (ale i tak odbywa się to bardzo szybko).

45. **W tablicy historii rozgałęzień z 1 bitem historii można zastosować następujący algorytm przewidywania (najbardziej złożony)**

- Skok opóźniony
  Nie, skoki opóźnione nie służą do przewidywania rozgałęzień, są zupełnie innym mechanizmem eliminacji hazardu.

- Przewidywanie, że rozgałęzienie (skok warunkowy) zawsze nastąpi
  Nie, to strategia statyczna, która może być wykonywana gdy adres rozkazu rozgałęzienia NIE jest w tablicy. Nie wykorzystuje bitu historii.

- Przewidywanie, że rozgałęzienie nigdy nie nastąpi
  Nie, to strategia statyczna, która może być wykonywana gdy adres rozkazu rozgałęzienia NIE jest w tablicy. Nie wykorzystuje bitu historii.

- **Przewidywanie, że kolejne wykonanie rozkazu rozgałęzienia będzie przebiegało tak samo jak poprzednie**
  Tak, i to jest wszystko na co stać historię 1-bitową. Historia 2-bitowa umożliwia interpretację:
  - historii ostatniego wykonania skoku - tak lub nie;
  - przewidywania następnego wykonania skoku - tak lub nie
  A zamiana strategii następuje dopiero po drugim błędzie przewidywania.

- Wstrzymanie napełniania potoku
  Nie, wstrzymywanie potoku mogą spowodować algorytmy zajmujące się eliminacją hazardu danych - zależnosci między argumentami.

46. **Do czynników tworzących wysoką niezawodność klastrów należą**

- **Mechanizm mirroringu dysków**
  Tak, bo system operacyjny może.

- **Dostęp każdego węzła do wspólnych zasobów (pamięci zewnętrznych)**
  Tak, w razie czego można podpiąć i korzystać z dodatkowej pamięci.

- **Redundancja węzłów**
  No tak, jest.

- Mechanizm "heartbeat"
  Co to jest?

- Zastosowanie procesorów wielordzeniowych w węzłach
  Nie, nie chodzi o liczbę rdzeni, ale o to, że każdy węzeł jest osobnych systemem, serwerem lub pecetem.

47. **Konsekwencją błędu przy przewidywaniu rozgałęzień może być:**

- Wstrzymanie realizowanego wątku i przejście do realizacji innego wątku
  Przewidywanie rozgałęzień odbywa się lokalnie, osobno dla każdego wątku.

- **Konieczność wyczyszczenia kolejki rozkazów do potoku**
  W przypadku jeżeli mechanizm przewidywania rozgałęzienia się pomyli i zacznie pobierać rozkazy z błędnego rozgałęzienia, potok rozkazów musi zostać wyczyszczony. Czyli np. w IFie miało być true*, a okazało się, że* false*, to należy zbędne pobrane rozkazy wyczyścić.*

- Konieczność wyczyszczenia tablicy historii rozgałęzień.
  W przypadku błędu należy tablicę aktualizować, aby w przyszłości to rozgałęzienie było przewidywane z większą dokładnością.

- Przerwanie realizowanego procesu / wątku i sygnalizacja wyjątku
  Nie, należy jedynie zmienić strategię / przejść do innej gałęzi, a nie usuwać proces. Błąd przewidywania rozgałęzień nie jest czymś na tyle złym, by przerywać program. To naturalna konsekwencja potokowości, z którą należy się uporać.

- Konieczność przemianowania rejestrów w procesorach
  Przemianowanie rejestrów występuje by procesorach skalarnych w celu równoległej realizacji zadań na potokach.

48. **Katalog może być stosowany do:**

- Utrzymania spójności pamięci podręcznych poziomu L1 i L2 w procesorach wielordzeniowych.
  Wiadomo, że latalog jest wykorzystywany do zachowania spójności pamięci w systemach wieloprocesorowych, UMA i NUMA. Ale czy w wielordzeniowym? A chuj wie.

- **Utrzymania spójności pamięci wszystkich węzłów w systemach CC-NUMA**
  Systemy NUMA wykorzystują katalogi do zachowania spójności pamięci, i tylko katalogi.

- Do utrzymania spójności pamięci węzłów systemów wieloprocesorowych z pamięcią rozproszoną (MPP)
  Jeśli chodzi o zachowanie spójności pamięci w obrębie węzła, to ***tak****, bo węzłem może być system UMA lub NUMA. Ale, jeśli chodzi o zachowanie spójności pamięci między węzłami, to* ***nie****, bo do tego służy przesył komunikatów.*

- Sterowania realizacją wątków w architekturze CUDA
  Zdecydowanie bullshit, CUDA nie wykorzystuje katalogu. W CUDA wątkami steruje osobny procesor wątków.

49. **Sprzętowe przełączenie wątków może być wynikiem:**

- **Chybienia przy odwołaniu do pamięci podręcznej.**
  Patrz: realizacja sprzętowej wielowątkowości 7.2

- Upływu zadanego czasu (np. taktu)
  Bullshit, szanujmy się. Można to zrobić programowo, ale nie sprzętowo.

- Wystąpienia rozkazu rozgałęzienia
  Dopiero jak nastąpi błąd przewidywania. Przy drabince IFów byłby armageddon.

- **Błędnego przewidywania rozgałęzień**
  Patrz: realizacja sprzętowej wielowątkowości 7.2

- Przesunięcia okien rejestrów
  To tylko zmiana rejestrów, niezwiązana z wątkami.

50. **W architekturze CC-NUMA czas dostępu do pamięci operacyjnej może zależeć od:**

- **Rodzaju dostępu (odczyt - zapis)**
  Zapis jest znacznie wolniejszy, bo wymaga aktualizacji całej spójnej logicznie pamięci.

- Stanu linii (zapisanego w katalogu), do której następuje odwołanie
  Jak linia znajduje się już w katalogu, to następuje odczyt tylko.

- **Położenia komórki, do której odwołuje się rozkaz (lokalna pamięć węzła – pamięć innego węzła)**
  Dostęp do pamięci nielokalnej (innego procka) jest znacznie dłuższy (ok. 10-krotnie)

- Odległości węzłów, zaangażowanych w wykonanie rozkazu, w strukturze sieci łączącej
  Różnice są rzędu mikrosekund, nie jest to znaczący czas, tylko margines błędu najwyżej.

51. **Wyprzedzające pobranie argumentu pozwala rozwiązać konflikt wynikający z:**

- Zależności wyjściowej miedzy rozkazami
  Tę zależność musi kontrolować układ sterujący.

- **Prawdziwej zależności danych**
  Tak, do tego służy, patrz: prawdziwa zależność danych 5.3.1

- Błędnego przewidywania rozgałęzień
  Nie powoduje konfliktów, należy je tylko obsłużyć i ograniczyć liczbę występowań.

- **Antyzależności miedzy rozkazami**
  Rozkazowi, który odczytuje jakąś zmienną, od razu jest podrzucana jej zmieniona wartość. zamiast przechodzić przez rejestr pośredni (zapis i ponowny odczyt).

Pytania otwarte

1 2010, Termin I

1. **Które cechy architektury CISC zostały zmienione i w jaki sposób w architekturze RISC?**

   - RISC:
     - niezbyt duża lista rozkazów, rozkazu wykonywane w zasadzie w 1 cyklu (zmiana)
     - niewielka liczba trybów adresowania (zmiana)
     - stała długość i prosty format rozkazu (chyba też zmiana)
     - model obliczeń rejestr-rejestr, wszystkie argumenty są w rejestrach (zmiana)
     - dostęp do pamięci jedynie dzięki rozkazom STORE i LOAD
     - duża ilość rejestrów uniwersalnych (chyba też zmiana)
     - jednostka sterująca zbudowana jako układ
     - intensywne wykorzystanie potokowości
     - stosowanie kompilatorów o dużych możliwościach optymalizacyjnych
   - Cechy:
     - duża liczba rozkazów
     - duża liczba trybów adresacji (5 – 20, Vax → 20)
     - model obliczeń pamięć – pamięć
     - komplikowana struktura sprzętu, przy małym wykorzystaniu rozkazów złożonych
     - uży rozrzut cech rozkazów w zakresie: złożoności, długości, czasu wykonania

2. **Wymień najważniejsze cechy architektury procesorów superskalarnych.**

   - kilka potokowych jednostek operacyjnych
   - wykonywanie kilku rozkazów w 1 takcie
   - konieczność pobrania z PaO kilku rozkazów jednym takcie

3. **Wyjaśnij na czym polega problem zależności między danymi przy potokowej realizacji rozkazów. Podaj przykłady rozwiązania tego problemu w jednopotokowych procesorach RISC.**

   - $I1 : R3 \leftarrow R3 op R5$
     $I2 : R4 \leftarrow R3 + 1$
     $I3 : R3 \leftarrow R5 + 1$
     $I4 : R7 \leftarrow R3\ op\ R4$
   - – prawdziwa zależność danych (RAW) - rozkaz I2 musi czekać na wykonanie rozkazu I1, podobnie I4 musi czekać na I3
     – zależność wyjściowa (WAW) - gdyby rozkazy I1 oraz I3 były realizowane równolegle (np. w różnych jednostkach funkcjonalnych), to wykonanie I3 musi się zakończyć po I1.
     – antyzależność (WAR) - w przypadku równoległej realizacji rozkazów I2 oraz I3 (lub zmiany kolejności tych rozkazów), wykonanie rozkazu I3 nie może być zakończone, dopóki nie nastąpi pobranie argumentu (odczyt) w rozkazie I2.

- Rozwiązania:
  - NOP
  - Proces przemianowania rejestrów (może to pomóc; nie jestem pewien)
  - Wstrzymanie napełniania potoku
  - Optymalizacja kodu na poziomie kompilacji / linkowania (odpowiedzialność zrzucona na kompilator)

4. **Omów budowę systemów o niejednorodnym dostępie do pamięci (ccNUMA)**
   Komputery w architekturze NUMA należą do systemów MIMD o niejednolitym dostępie do pamięci. Bierze to się stąd, iż każdy procesor ma własną pamięć; jednak do tej pamięci ma również dostęp każdy inny procesor, ztą tylko różnicą, że czas dostępu do takiej pamięci jest dużo większy niż czas dostępu do pamięci, którą dany procesor posiada. Zaliczenie procesora z pamięcią i jednostką zarządzającą pamięcią nazywa się węzłem. Zadaniem takiej jednostki zarządzającej pamięcią jest odpowiednie kierowanie adresów i danych w zależności o którą pamięć nam chodzi (lokalną dla danego procesora czy też znajdującą się przy innym procesorze). We współczesnych rozwiązaniach praktycznych duży nacisk kładzie się na zgodność (spójność) pamięci podręcznej (są to systemy typu CC-NUMA)
   MIMD typu C.C.-NUMA (ze spójnością pamięci podręcznej).

5. **Wymień i krótko scharakteryzuj rozwiązania konstrukcyjne serwerów używanych do budowy klastrów.**

   - Serwery wolnostojące
   - Serwery stelażowe
     - Zalety:
       * możliwość instalowania obok siebie serwerów różnych dostawców
     - Wady:
       * kable zasilające
       * kable sieciowe
       * chłodzenie
   - Serwery kasetowy
     - Zalety:
       * jedna obudowa typu blade
       * wspólne zasilanie i chłodzenie - jeden kabel zasilający
       * prostsze okablowanie sieciowe
       * proste zarządzanie
       * mniejsze wymiary i duża gęstość upakowania
     - Wady:
       * możliwość instalowania tylko identycznych serwerów

2 2012, Termin I

1. **Które cechy architektury CISC zostały zmienione i w jaki sposób w architekturze RISC?**
   Patrz: 1

2. **Porównaj realizację rozkazów wektorowych w komputerach wektorowych, macierzowych i procesorach superskalarnych z rozszerzeniami SIMD.**

   W zasadzie komputery wektorowe zaliczane są do architektury SISD gdzie mamy do czynienia z jedną jednostką wykonawczą i dlatego też operacja na kolejnej danej wektora rozpoczyna się pewien czas dalej, związany z wykonaniem jednej części potoku. Komputery macierzowe należą zaś do klasy SIMD, gdzie istnieje wiele jednostek wykonawczych (elementów procesorowych) sterowanych synchronicznie przez jednostkę sterującą oraz każda jednostka wykonawcza korzysta ze swoich własnych danych. Tak więc odpowiednio rozmieszczając dane możemy dokonać operacji na wszystkich danych wektora w jednym czasie. Dodatkową różnicą jest fakt, iż wektoryzacją programu w komputerze wektorowym zajmuje się kompilator, natomiast w komputerze macierzowym nie jest to już takie łatwe, gdyż wszystko zależy od połączeń EP.

   Co do komputerów superskalarnych wykonywanie rozkazu odbywa się jednocześnie na wielu "porcjach danych". Ważnym aspektem jest możliwość wystąpienia problem zależności danych przy potokowej ich realizacji (nie jestem pewien czy dobrze ująłem

3. **Scharakteryzuj ogólnie systemy wieloprocesorowe z rozproszoną pamięcią.**

   MPP (*Massively Parallel Processing*) to rozwiązanie bazujące na architekturze MIMD z pamięcią rozproszoną. Komputer MPP zbudowany jest z dużej ilości niezależnych procesorów (niekoniecznie superszybkich – w kupie siła!), wyposażonych we własną pamięć podręczną CACHE i zwykłą pamięć, połączonych ze sobą za pomocą sieci – przez nią wysyła się komunikaty. Takie podejście do sprawy zapewnia dużą i łatwą skalowalność – dołożenie 1000 nowych procesorów nie wiąże się z dużym nakładem pracy – nie trzeba budować nowych układów zarządzających dostępem do pamięci. Każdy EP (Element Przetwarzający – procesor) może wykonywać inne zadanie, przez co może być widziany jako osobny komputer (np. przez system operacyjny). Systemy MPP wymagają jednak dużego nakładu pracy z punktu widzenia oprogramowania, aby w pełni wykorzystać moc komputera, trzeba zadbać o odpowiednie rozłożenie zadań między procesory (np. fork() <rotfl>).

4. **Wyjaśnij na czym polega problem zależności między danymi przy potokowej realizacji rozkazów. Podaj przykłady rozwiązania tego problemu w jednopotokowych procesorach RISC. Jak komplikuje się ten problem w procesorach superskalarnych?**
   Patrz: 3

5. **Jakie zalety i jakie warunki wykonania ma sprzętowa, współbieżna realizacja wielu wątków w jednym rdzeniu?**

   Cel współbieżnej realizacji dwóch (lub więcej) wątków w jednym procesorze (rdzeniu): Minimalizacja strat cykli powstałych w trakcie realizacji pojedynczego wątku w wyniku:

   - chybionych odwołań do pamięci podręcznej,
   - błędów w przewidywaniu rozgałęzień,
   - zależności między argumentami kolejnych rozkazów.

   Warunki sprzętowej realizacji wielowątkowości:

   - powielenie zestawów rejestrów uniwersalnych (lub powielenie tabel mapowania rejestrów)
   - powielenie liczników rozkazów
   - powielenie układów dostępu do pamięci podręcznej (tabel stron)
   - powielenie sterowników przerwań

3 2012, Termin II

1. **Scharakteryzuj architekturę RISC.**
   Patrz: 1

2. **Wyjaśnij na czym polega problem potokowej realizacji rozkazów skoków (rozgałęzień). Podaj przykłady rozwiązań tego problemu we współczesnych procesorach.**

   Problem polega na tym, że rozkaz skoku może spowodować problem „przeładowania potoku" czyli jego wyczyszczenia oraz ponownego wypełnienia - zapewnienie stałego dopływu rozkazów do potoku. Pierwsze rozwiązania opierały się na:

   - wcześniejszym wyznaczeniu adresu rozgałęzienia (skoku) oraz pobraniu razem z rozkazem następnym za rozgałęzieniem również rozkazu docelowego (jeszcze przed rozstrzygnięciem spełnienia warunku rozgałęzienia
   - powielenie potoku (pierwszych stopni) i rozpoczęcie równoległego pobierania dwóch strumieni rozkazów (do chwili rozstrzygnięcia rozgałęzienia), później unieważnienie jednego strumienia

   Te działania doprowadziły finalnie do aktualnego podejścia czyli przewidywania rozgałęzień. Wyróżniamy następujące strategie przewidywania rozgałęzień:

   (a) Statyczne
      - przewidywanie, że rozgałęzienie (skok warunkowy) zawsze nastąpi
      - przewidywanie, że rozgałęzienie nigdy nie nastąpi (Motorola 68020, VAX 11/780)
      - odejmowanie decyzji na podstawie kodu rozkazu rozgałęzienia (specjalny bit ustawiany przez kompilator) (Ultra SPARC III)

   (b) Dynamiczne
      - Czyli tablica historii rozgałęzień

   (c) Inne
      - przewidywanie, że skok wstecz względem licznika rozkazów zawsze nastąpi
      - przewidywanie, że skok do przodu względem licznika rozkazów nigdy nie nastąpi
      - przełączanie wątków zamiast prognozowania rozgałęzień w niektórych procesorach ze sprzętowym wsparciem wielowątkowości

3. **Porównaj realizację rozkazów wektorowych w komputerach wektorowych i macierzowych.**
   Patrz: 2

4. **Scharakteryzuj architekturę systemu UMA.**

   Jeden z rodzajów systemu wieloprocesorowego, charakteryzuje się wspólną pamięcią operacyjną dla wszystkich procesorów, każdy procesor ma własną pamięć podręczną (cache). Czas dostępu do pamięci jest dla wszystkich procesów identyczny. Charakterystyczne dla SMP.

5. **Wymień cele sprzętowej, współbieżnej realizacji wielu wątków w jednym rdzeniu.**
   Cel współbieżnej realizacji dwóch (lub więcej) wątków w jednym procesorze (rdzeniu): Minimalizacja strat cykli powstałych w trakcie realizacji pojedynczego wątku w wyniku:

   - chybionych odwołań do pamięci podręcznej,
   - błędów w przewidywaniu rozgałęzień,
   - zależności między argumentami kolejnych rozkazów.

4 2013, Termin I

1. **Wymień cechy architektury superskalarnej.**
   Patrz: 2

2. **Uszereguj systemy wieloprocesorowe i klastry rosnąco wg skalowalności.**
   Według mnie chociaż nie jestem tego pewien to systemy wieloprocesorowe trzeba by uszeregować tak:

   - komputery macierzowe (jeżeli traktować elementy przetwarzające jako osobne jednostki) - ciężko zwiększać elementy przetwarzające lub układy macierzowe jako peryferia komputera
   - UMA - ciężkie do rozbudowy ze względu na pojedynczą magistralę lub przełącznicę
   - NUMA - średnie w rozbudowie
   - MPP - łatwe w rozbudowie; wystarczy dokładać procesory i zadbać o przepływ danych (wymiana komunikatów)
   - Klastry - wydajnościowe i niezawodnościowe; pracujące na wspólnych dyskach; każdy z nich ma osobny, niezależnie utrzymywany system operacyjny

3. **Scharakteryzuj ogólnie systemy typu klaster.**

   - Sieci łączące – standardy: Gigabit Ethernet, Infiniband
   - Komunikacja między węzłami (procesami) – przesył komunikatów
   - Bardzo wysoka skalowalność
   - Cele budowy: wysoka wydajność lub/i wysoka niezawodność
   - Korzystny wskaźnik: cena/wydajność
   - Według mnie jeszcze oprogramowanie, które umożliwi prawidłową pracę tych wszystkich komputerów

4. **Wyjaśnij, na czym polega problem zależności między danymi przy potokowej realizacji rozkazów. Podaj przykłady rozwiązania tego problemu w jednopotokowych procesorach RISC i w procesorach superskalarnych.**
   Patrz: 3

5. **Co jest celem sprzętowego sterowania współbieżną realizacją wielu wątków w jednym rdzeniu? Jakie są metody i jakie warunki sprzętowej realizacji wielowątkowości?**
   Patrz: 5

5 2015, Termin 0

1. **Omów najważniejsze cechy architektury CUDA w szczególności model pamięci.**

   - Cała architektura utworzona z wątków, które są grupowane w bloki.
   - Wątki niezależnie wykonują tę samą operację.
   - Każdy wątek ma swoją lokalną pamięć, dodatkowo mają wspólną pamięć globalną i współdzieloną.
   - Za komunikację między wątkami odpowiadają mechanizmy synchronizacji (*barrier synchronization*)

2. **Omów budowę systemów o niejednorodnym dostępie do pamięci (NUMA).**

   - Systemy NUMA zbudowane są z wielu procesorów, gdzie każdy procesor posiada własną, bardzo szybką pamięć lokalną.
   - Dodatkowo procesory posiadają wspólną, ogromną pamięć wspólną.
   - Dostęp do nielokalnej pamięci jest znacznie wolniejszy niż do lokalnej (ok. 10 razy).

- W celu zapewnienia spójności między pamięciami podręcznymi stosowane są węzły i katalogi, i tylko katalogi, które są znacznie lepsze od protokołu MESI.
- Można wyróżnić podkategorie systemów: NC-NUMA (non-cached) i CC-NUMA (cache coherent)
- Systemy te posiadają średniej jakości skalowalność
- Pamięć podręczna jest fizycznie rozproszona, ale logicznie wspólna.
- Dostęp do pamięci nielokalnej odbywa się z użyciem katalogów oraz odczytywanie całych linii pamięci podręcznej.
- Architektura NUMA jest bardzo efektywna dla aplikacji, które częściej odczytują z nielokalnej pamięci i nieefektywna dla aplikacji, które części zapisują do niej.
  Przy zapisie trzeba zaktualizować stan tej linii we wszystkich węzłach, które je pobrały do siebie.

3. **Porównaj realizację rozkazów wektorowych w komputerach wektorowych i macierzowych.** 9.2

   - W komputerze wektorowym szybkość realizacji rozkazów rośnie logarytmicznie wraz ze zwiększeniem długości wektora.
   - Efektywność wykonywania rozkazów w komputerze wektorowym dąży to pewnej stałej wartości, która jest efektywnością idealną.
   - Z kolei w komputerze macierzowym ten sam rozkaz może być wykonywany w jednym kroku (jeśli jest wystarczająca liczba jednostek przetwarzających), czyli w stałym czasie.

4. **Wymień różnice między systemami SMP a MPP.**

   - SMP posiada pamięć logicznie wspólną, ale może ona być fizycznie jednorodna (UMA) lub rozproszona (NUMA). Z kolei MPP posiada pamięć fizycznie i logicznie rozproszoną.
   - MMP jest znacznie bardziej skalowalne niż SMP (UMA - słabo, NUMA - średnio).
   - W SMP wymagany jest mechanizm do zachowania spójności pamięci podręcznych procesorów - może to być protokół MESI (UMA) lub katalogi (UMA i NUMA).
   - W MPP do kontroli wystarcza przesył komunikatów.
   - W SMP istnieje jedna wspólna pamięć, z kolei w MPP każdy węzeł ma swoją pamięć lokalną plus opcjonalnie wspólną.
   - W MPP jest wolniejsza komunikacji między węzłami sieci - potrzebna jest wymiana większej liczby informacji.

5. **Omów rozwiązania stosowane w klastrach o wysokiej niezawodności.**

   - Redundancja węzłów (mocy obliczeniowej) - większa moc obliczeniowa, więcej mocy w zapasie
   - Dostęp do wspólnych zasobów (pamięci zewnętrznych) - więcej pamięci
   - Mirroring dysków - zabezpieczenie przed utratą danych
   - Mechanizmy kontrolujące funkcjonowanie węzłów - efektywniejsza praca całości jako jednego systemu
   - Redundancja sieci łączących (dla 3 rodzajów sieci) - przypadku błędu sieci klaster nie przestaje działać, tylko inna sieć przejmuje kontrolę
   - Redundancja zasilania - wincyj mocy i prundu

Zadania egzaminacyjne

1 Sparc

Uwagi:

- W języku asemblera SPARC komentarze są oznaczane przez znak wykrzyknika (!), a nie średnika (;). W listingach są średniki ze względu na wbudowany listingu asemblera w latexie.

1.1 Laborka: min, max oraz max - min

Ocena nieznana.

1.1.1 Funkcja w języku C

```
#include <stdio.h>
extern int minmax(int *tab, int n, int *max, int *min);

int main()
{
    int i, N, *tab;
    int max, min, span;
    scanf("%i", &N);
    if (N < 0) {
        printf("N < 0!\n");
        return -1;
    }
    tab = malloc(N*sizeof(*tab));
    for(i = 0; i < N; ++i)
        scanf("%i", tab + i);
    span = minmax(tab, N, &max, &min);
    printf("min = %i, max = %i, span = %i\n",
    min, max, span);
    free(tab);
    return 0;
}
```

1.1.2 Odpowiednik w SPARCu

```
.global minmax
.proc 4
; rejestry:
; %i0 - adres do tablicy
; %i1 - ilosc liczb (N)
; %i2 - adres do max
; %i3 - adres do min
;
; %l6 - pomocnicza do przechowania przesuniecia w bajtach
; %l7 - pomocnicza do porownywania z min/max
; %l1 - max
; %l2 - min
minmax:
    save %sp, -96, %sp ; przesuniecie okienka
    ; zaladuj wartosci dla max i min, gdy n <= 0
    mov 0, %l1
    mov 0, %l2
    ; sprawdz czy n > 0
    subcc %i1, 1, %i1
    blt end
    nop
    ; zaladuj startowe max (%l1) i min(%l2) z pierwszej liczby
    ld [%i0], %l1
    mov %l1, %l2
petla:
    ; sprawdz koniec petli
    blt end
    ; wylicz adres i zaladuj kolejna liczbe
    smul %i1, 4, %l6
    ld [%i0+%l6], %l0    ; %l0 - obecna liczba

    ; update max
    subcc %l0, %l1, %l7 ; %l1 - max
    blt next
    nop
    mov %l0, %l1
next:
    ; update min
    subcc %l0, %l2, %l7 ; %l2 - min
    bgt next2
    nop
    mov %l0, %l2

next2:
    ba petla
    subcc %i1, 1, %i1

end:
    ; zapisz wynik
    st %l1, [%i2]
    st %l2, [%i3]
    sub %l1, %l2, %i0
    ret
    restore ; odtworzenie okienka
```

1.2 Laborka, szukanie min i max

Ocena nieznana.

```
.global _start
;' defajny:'
define (ilosc, 4)
define (delta, 3)
;' rejestry:'
define (adres, l3)          ;' %l3 - przesuniecie tablicy'
define (min, l1)            ;' %l1 - min'
define (max, l2)            ; ' %l2 - max'
define (indeks, l4)         ;' %l4 - indeks'
define (pobrane, l5)        ;' %l5 - pobrana wartosc'
define (tmp, l6)            ;' %l6 - pomocniczy'
define (rozpietosc, o0)     ;' %o1 - zwrocona rozpietosc'
;' kod:'
_start:
    save %sp, -96, %sp
    mov 0, %adres
    mov 0, %indeks
    ;'zapis wartosci poczatkowych min i max'
    ld [%adres], %min
    ld [%adres], %max
    ;'dodajemy delte do pierwszej komorki'
    add %min, delta, %tmp
    st %tmp, [%adres]
petla:
    add %indeks, 1, %indeks     ;'indeks++'
    add %adres, 4, %adres       ;'przesuniecie na kolejny element'
    cmp %indeks, ilosc
    be end
    nop
    ;'zwiekszamy o delte'
    ld [%adres], %pobrane
    add %pobrane, delta, %pobrane
    st %pobrane, [%adres]
;'sprawdzamy czy pobranma wartosc jest mniejsza niz min'
    subcc %pobrane, %min, %tmp
    bg dalej
    nop
    mov %pobrane, %min
    ba petla_end
    nop
dalej:
    ;'sprawdzamy czy pobrana wartosc jest wieksza niz max'
    subcc %pobrane, %max, %tmp
    bl petla_end
    nop
    mov %pobrane, %max
petla_end:
    ba petla
    nop
    end:
    sub %max, %min, %rozpietosc     ;'zapisujemy rozpietosc'
    ret
```

1.3 2008, I termin, Jerzy Respondek

1.3.1 Treść

Napisz funkcję w asemblerze procesora SPARC obliczającą sumę liczb naturalnych od 1 do danej n jako argument funkcji. Założyć, że $n >= 1$.
Przykład: f(5) = 1 + 2 + 3 + 4 + 5 = 15

1.3.2 Propozycja rozwiązania 1

```
.global funkcja
.proc 4
funkcja:
    save %sp, -96, %sp        ; trzeba tutaj to robić ???
    mov %i0, %l0              ; a
    mov 1, %l1                ; liczba naturalna
    mov 0, %l2                ; wynik
pętla:
    add %l1, %l2, %l2         ; liczba + suma = suma
    add %l1, 1, %l1           ; liczba++
    subbcc %l0, 1, %l0        ; a--
    bl koneic
    nop
    ba pętla
    nop
koniec:
    mov %l2, %i0              ; wynik
    ret
    restore
```

1.3.3 Propozycja rozwiązania 2

```
.global sumator
.proc 4
sumator:
    save %sp, -96, %sp        ! przesunięcie okna
    mov %i0, %l1              ! a w l1
    mov %l1, %l0              ! suma = a
   petla:
    subcc %l1, 1, %l1         ! dekrementacja licznika
    bneg koniec
    add %l0, %l1, %l0         ! suma += licznik
    ba petla
koniec:
    mov %l0, %i0              ! zwrócenie sumy
    ret
    restore                   ! przywrócenie stanu okna
```

1.4 2010, I termin, Jerzy Respondek

1.4.1 Treść

Napisz w asemblerze procesora SPARC funkcję obliczającą sumę kwadratów wszystkich liczb całkowitych z przedziału a do b. Założyć $a < b$, np.
f(2, 5) = 2 * 2 + 3 * 3 + 4 * 4 + 5 * 5
Nagłówek funkcji ma mieć postać:

```
int f(int a, int b)
```

1.4.2 Propozycja rozwiązania

1.5 2012, I termin, Jerzy Respondek

1.5.1 Treść

Napisz w asemblerze procesora SPARC funkcję realizującą dokładnie tę samą operację co jej odpowiednik w języku C:

```
int f(int *tab, int n)
{
    int i, suma = 0;
    for(i = 0; i < n; i++)
    {
        suma -= (2 * i + 1) * tab[i];
        suma *= suma;
    }
    return suma;
}
```

1.5.2 Propozycja rozwiązania 1

```
.global func
.proc 4

funkcja:
    save %sp, -96, %sp
    mov %i0, %l0            ; wskaźnik tablicy, tak podano argument
    ld [%i0], %l1           ; wartość tablicy spod wskaźnika odczytujemy
        poprzez LD
    mov %i1, %l2            ; rozmiar
    mov 1, %l3              ; i
    mov 0, %l4              ; temp
    mov 0, %l5              ; suma
pętla:
    subcc %l2, 1, %l2       ; n--
    bl koniec               ; if n < 0 koniec
    nop

    smul %l3, 2, %l4        ; temp = 2*i
    add %l4, 1, %l4         ; temp = temp +1 = 2*i+1
    smul %l1, %l4, %l4      ; temp = temp * tab[i] = (2*i+1)*tab[i]
    subcc %l5, %l4, %l5     ; suma = suma - temp = suma - (2*i+1)*tab[i]

    smul %l5, %l5, %l5      ; suma = suma * suma
    add %l0, 4, %l0         ; *tab++ przesuwamy sie o 4 na kolejny element
        bo tyle ma int
    ld [%l0], %l1           ;pobieramy nowy element
    ba pętla
    nop

koniec:
    mov %l5, %i0            ; zwracamy wynik w i0 bo po restore zamienia si
        ę input na output
    ret                     ; ret bo było save
    restore
```

1.5.3 Propozycja rozwiązania 2

```
.global fun
.proc 4

;    a(n) = a(n - 1) ^ k + n * k; a(0) = 1
fun:
    save %sp, -96, %sp
    ; %i0 == n
    ; %i1 == k

    subcc %i0, 1, %o0    ; %o0 == n - 1
    bneg return1
    nop

    ; trzeba obliczyc a(n - 1)
    mov %i1, %o1
    call fun
    nop

    ; %o0 == a(n - 1)
    mov %i1, %l1          ; %l1 == k
    mov 1, %l2            ; %l2 == 1 (tu bedzie wynik potegowania)
power:
    umul %l2, %o0, %l2
    subcc %l1, 1, %l1     ; dekrementuj licznik petli
    bg power              ; skok, gdy licznik > 0
    nop

    ; %l2 == a(n - 1) ^ k
    umul %i0, %i1, %i0
    ; %i0 == n * k
    add %i0, %l2, %i0
    ; %i0 == a(n - 1) ^ k + n * k == a(n)
    ba return
    nop

return1:
    mov 1, %i0
return:
    ret
    restore
```

1.6 2013, I termin, Jerzy Respondek

1.6.1 Treść

Napisz w asemblerze procesora SPARC funkcję zwracającą *a(n)* wyliczoną z poniższego wzoru rekurencyjnego, a pobierającą dwa argumenty: n oraz k, obydwa typu *unsigned int*.

$$a(n) = a(n-1)^k + n \cdot k, \quad a(0) = 1, \quad n = 1, 2, 3, ...$$

1.6.2 Rozwiązanie nr 1 by Doxus

```
.global _start

_start:
    MOV     0x05,   %g1             ;! g1 - K
    MOV     0x0A,   %o7             ;! rej o7 i i7 -> N (lokalne)
    MOV     %o7,    %g7             ;! N absolutne

_petla:
    SAVE    %sp,    -96,    %sp     ;! otworzenie okna

    SUBcc   %i7,    0x00,   %g0     ;! sprawdzenie, czy to dno
        rekurencji
    BE      _nzero
    NOP

    SUB     %i7,    0x01,   %o7     ;! wykonanie rekurencji
    BA      _petla
    NOP

_nzero:
    MOV     0x00,   %i5
    MOV     0x00,   %g2             ;! g2 temp n
_petlapowrot:
    RESTORE                         ;! zamknięcie koła

    MOV     %i5,    %l0             ;! obliczenia
    MOV     0x01,   %l1             ;! temp k
_petlamnoz:
    UMUL    %i5,    %l0,    %l0     ;! obliczenia zgodnie ze wzorem
    ADD     %l1,    0x01,   %l1
    SUBcc   %l1,    %g1,    %g0
    BNE     _petlamnoz

    UMUL    %g2,    %g1,    %l2
    ADD     %l1,    %l2,    %o5

    ADD     %g2,    0x01,   %g2

    SUBcc   %g2,    %g7,    %g0     ;! czy koniec odkręcania koła
    BLE     _petlapowrot
    NOP

    MOV     %i5,    %g1             ;! g1 - wynik koncowy
    NOP                             ;! kuniec
```

1.6.3 Rozwiązanie nr 2

Podobno otrzymano za to 5, choć rozwiązanie NIE JEST w pełni poprawne.

```
.global fun
.proc 4

fun:
    save %sp,-96,%sp

    mov %i0 , %l0              ; l0 - n
    mov %i1 , %l1              ; l1 - k
    mov 0, %l2                 ; power
    mov 1, %l3                 ; a(n) = 1

    subcc %i0 , 1, %i0
    bl theEnd                  ; if n = 0 then jump to theEnd
    nop

    mov %l0 , %l2              ; power = n
    smul %l2 , %l1 , %l2       ; power = power * k
    add %l2 , %l1 , %l2        ; power = power + k

    call fun                   ; call recursion
    mov %i0 , %l3              ; get score of recursion

expo:
    smul %l3 , %l3 , %l3
    subcc %l2 , 1, %l2
    bl theEnd
    nop
    ba expo
    nop
theEnd:
    mov %l3 , %i0              ; return score
    ret
    restore

.end
```

1.7 2014, 0 termin, Jerzy Respondek

1.7.1 Treść

Napisz w asemblerze procesora SPARC funkcję realizującą dokładnie tę samą operację, co jej odpowiednik w języku C:

```
int f(int *tab, int n)
{
    int i, suma = 0;
    for(i = 0; i < n; i++)
    {
        suma += i*tab[i];
    }
    return suma;
}
```

1.7.2 Rozwiązanie

```
.global _start

_start:
! i1 wskaznik na pocz tablicy
! i2 n

! i0 - wyjsciowa suma (RESTORE spowoduje ze bedzie to w rej.
! wyjsciowych funkcji nadrzędnej

! l0, l1 - wskaznik na el. tablicy, n
! l2 - iterator
! l3 - suma

mov %i1, %l0
mov %i2, %l1
mov 0x00, %l2
mov 0x00, %l3
_loop:
! if sprawdzają czy i < n
subcc %l2, %l1, %g0
bge _koniec
nop

ld [%l0], %l7
umul %l7, %l2, %l7
add %l7, %l3, %l3

add %l2, 0x01, %l2

ba _loop
add %l0, 0x04, %l0        !można zamienić miejscami i dać nop, ale tak
    optymalniej

_koniec:
mov %l3, %i0
```

2 PVM

2.1 Wstęp z laborek, szukanie min i max

2.1.1 Treść

Napisać program znajdujący minimum i maksimum z macierzy.
Hello.c - program główny, rodzic; Hello_other.c - program podrzędny, potomek.

2.1.2 Rozwiązanie

Program przekazuje kolejne wiersze macierzy do programów potomnych, które znajdują lokalne minimum i maksimum. Program zbiera wszystkie minima i maksima do tablicy o rozmiarze wysokości macierzy. Pod koniec sam ręcznie wylicza min i max z tych dwóch tablic.
Należy pamiętać, że programy potomne muszą fizycznie znajdować się na dyskach innych komputerów w sieci PVM.
Program działający, oceniony na 5.

```
/* - Autorzy:
   -- Forczu Forczmański
   -- Wuda Wudecki
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pvm3.h"
#define WYSOKOSC 5       // liczba wierszy
#define SZEROKOSC 5      // liczba kolumn
/// Program rodzica
main()
{
    // dane potrzebne do obliczeń
    int matrix[WYSOKOSC][SZEROKOSC];
    int min_result[WYSOKOSC], max_result[WYSOKOSC];
    int minimum, maksimum;
    // wypełnienie macierzy danymi
    int i, j;
    for ( i = 0; i < WYSOKOSC; ++i )
        for ( j = 0; j < SZEROKOSC; ++j)
            matrix[i][j] = rand() % 30;
    // wypisanie macierzy na konsoli
    for ( i = 0; i < WYSOKOSC; ++i )
    {
        for ( j = 0; j < SZEROKOSC; ++j)
            printf("%d ", matrix[i][j]);
        printf("\n\n");
    }
    // pobranie informacji
    int ilhost, ilarch;
    struct pvmhostinfo * info;
    pvm_config(&ilhost, &ilarch, &info);
    printf("Liczba hostow: %d\n", ilhost);

    int id1 = 0;
    int tid;
```

```
// Dla każdego hosta - inicjujemy go
for ( i = 0; i < ilhost; i++ )
{
    pvm_spawn( "/home/pvm/pvm3/sekcja11/bin/LINUX/hello_other", 0,
        PvmTaskHost, info[i].hi_name, 1, &tid);
    if ( tid < 0 )
    {
        ilhost--;
        continue;
    }
    printf("tid: %d\n", tid);
    pvm_initsend(PvmDataDefault);
    // wysyłamy:
    // id wiersza
    pvm_pkint(&id1, 1, 1);
    // elementy wiersza
    pvm_pkint(&matrix[id1][0], SZEROKOSC, 1);
    pvm_send(tid, 100);
    id1++;
}
//// Wykonywanie programu aż do przedostatniej pętli
int bufid, child_tid, child_id1, tmp;
while ( id1 < WYSOKOSC )
{
    bufid = pvm_recv(-1, 200);
    pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
    printf("recv: %d\n", child_tid);
    // pobranie id wiersza
    pvm_upkint(&child_id1, 1, 1);
    // pobranie nowych min / max
    pvm_upkint(&min_result[child_id1], 1, 1);
    pvm_upkint(&max_result[child_id1], 1, 1);
    // wysłanie nowych danych
    pvm_initsend(PvmDataDefault);
    // id kolejnego wiersza
    pvm_pkint(&id1, 1, 1);
    // nowy wiersz
    pvm_pkint(&matrix[id1][0], SZEROKOSC, 1);
    pvm_send(child_tid, 100);
    id1++;
}
//// Odebranie ostatnich danych
for (i = 0; i < id1 - ilhost + 1; i++ )
{
    bufid = pvm_recv(-1, 200);
    pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
    printf("recv: %d\n", child_tid);
    // pobranie id wiersza
    pvm_upkint(&child_id1, 1, 1);
    // pobranie nowych min / max
    pvm_upkint(&min_result[child_id1], 1, 1);
    pvm_upkint(&max_result[child_id1], 1, 1);
}
```

```
    // uzysaknie minimum z wiersza
    minimum = min_result[0];
    maksimum = max_result[0];
    for (j = 1; j < WYSOKOSC; j++)
    {
        if ( max_result[j] > maksimum )
            maksimum = max_result[j];
        if ( min_result[j] < minimum )
            minimum = min_result[j];
    }
    printf("Uzyskane wartosci:\nMIN: %d, MAX: %d\n", minimum, maksimum);
    pvm_exit();
    return 0;
}
```

Program potomka

```
#include <stdio.h>
#include <math.h>
#include "pvm3.h"
#define WYSOKOSC 5      // liczba wierszy
#define SZEROKOSC 5     // liczba kolumn
/// Program potomka
int main()
{
    int masterid, id1, j, curr_row[SZEROKOSC], curr_min, curr_max;
    // pobierz id rodzica
    masterid = pvm_parent();
    if (masterid == 0)
        exit(1);
    while(1)
    {
        pvm_recv(masterid, 100);
        // pobranie wartości:
        // id wiersza
        pvm_upkint(&id1, 1, 1);
        pvm_upkint(&curr_row[0], SZEROKOSC, 1);
        // uzysaknie minimum z wiersza
        curr_min = curr_max = curr_row[0];
        for (j = 1; j < SZEROKOSC; j++)
        {
            if ( curr_row[j] > curr_max )
                curr_max = curr_row[j];
            if ( curr_row[j] < curr_min )
                curr_min = curr_row[j];
        }
        // wysłanie nowych danych
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&id1, 1, 1);
        pvm_pkint(&curr_min, 1, 1);
        pvm_pkint(&curr_max, 1, 1);
        pvm_send(masterid, 200);
    }
    pvm_exit();
    return 0;
}
```

2.2 Laborki, odejmowanie macierzy

2.2.1 Treść

Odejmowanie macierzy.

2.2.2 Rozwiązanie

Ocena nieznana.

```
/* AK Lab 2 - PVM
    Anna Kusnierz
    Tomasz Szoltysek
    Temat: Odejmowanie dwoch macierzy
*/
#include <stdio.h>
#include <math.h>
#include "pvm3.h"
#define MATRIX_SIZE 20
int main()
{
    int i,j;
    int count = 0;       //licznik wierszy macierzy
    int rescount;
    int tidmaster, ilhost, ilarch, bufid,t_id,bytes,msgtag;
    struct pvmhostinfo info;

    int a[MATRIX_SIZE][MATRIX_SIZE], b[MATRIX_SIZE][MATRIX_SIZE], r[
       MATRIX_SIZE][MATRIX_SIZE];
    FILE *txt = fopen("result.txt","w");

    for(i=0;i<MATRIX_SIZE;i++)
    {
        for(j=0;j<MATRIX_SIZE;j++)
        {
            a[i][j] = rand();
            b[i][j] = rand();
        }
    }
    fprintf(txt,"Macierz A:\n
        ----------\n\n");
    for(i=0;i<MATRIX_SIZE;i++)
    {
        for(j=0;j<MATRIX_SIZE;j++)
            fprintf(txt,"%d\t",a[i][j]);
        fprintf(txt,"\n");
    }
    fprintf(txt,"Macierz B:\n
        ----------\n\n");
    for(i=0; i<MATRIX_SIZE;i++)
    {
        for(j=0;j<MATRIX_SIZE;j++)
            fprintf(txt,"%d\t",b[i][j]);
        fprintf(txt,"\n");
    }
```

```
    tidmaster = pvm_mytid();
    pvm_config(&ilhost, &ilarch, &info);
    printf("%d",ilhost);
    for(i=0; i < (ilhost > MATRIX_SIZE ? MATRIX_SIZE : ilhost) ;i++)
    {
        pvm_spawn("/home/pvm3/pvm3/sekcja4/hello_other",0,PvmTaskHost,info[
            i].hi_name,1,&t_id);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&a[count][0],MATRIX_SIZE,1);
        pvm_pkint(&b[count][0],MATRIX_SIZE,1);
        pvm_pkint(&count,1,1);
        pvm_send(t_id,100);
        ++count;
    }
    while(count<MATRIX_SIZE)
    {
        bufid = pvm_recv(-1,200);
        pvm_bufinfo(bufid,&bytes,&msgtag,&t_id);
        pvm_upkint(&rescount,1,1);
        pvm_upkint(&r[rescount][0],MATRIX_SIZE,1);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&a[count][0],MATRIX_SIZE,1);
        pvm_pkint(&b[count][0],MATRIX_SIZE,1);
        pvm_pkint(&count,1,1);
        pvm_send(t_id,100);
        ++count;
    }
    for(i = 0; i < (ilhost > MATRIX_SIZE ? MATRIX_SIZE : ilhost); i++)
    {
        bufid = pvm_recv(-1,200);
        pvm_bufinfo(bufid,&bytes,&msgtag,&t_id);
        pvm_upkint(&rescount,1,1);
        pvm_upkint(&r[rescount][0],MATRIX_SIZE,1);
        pvm_kill(t_id);
    }
    fprintf(txt,"Macierz wynikowa:\n
        ----------\n\n");
    for(i=0; i<MATRIX_SIZE;i++)
    {
        for(j=0;j<MATRIX_SIZE;j++)
            fprintf(txt,"%d\t",r[i][j]);
        fprintf(txt,"\n");
    }
    fclose(txt);
    exit(0);
}
```

Program potomny

```
#include <stdio.h>
#include "pvm3.h"
#define MATRIX_SIZE 20
int main()
{
    int masterid, count, i;
    double vecta[MATRIX_SIZE], vectb[MATRIX_SIZE], vectr[MATRIX_SIZE];
    masterid = pvm_parent();
    if(masterid == 0) exit(1);  //zabezpieczenie przed uruchomieniem z
       poziomu rodzica
    //OBSŁUGA OBLICZEŃ i WYSYŁANIA WYNIKÓW
    while(1)
    {
        pvm_recv(masterid,100);
        pvm_upkdouble(&vecta[0],MATRIX_SIZE,1);
        pvm_upkdouble(&vectb[0],MATRIX_SIZE,1);
        pvm_upkint(&count,1,1);
        for(i = 0; i < MATRIX_SIZE; i++)
            vectr[i] = vecta[i] - vectb[i];
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&count,1,1);
        pvm_pkdouble(&vectr[0],MATRIX_SIZE,1);
        pvm_send(masterid,200);
    }
    return(0);
}
```

2.3 Laborki, Szyfr Cezara

2.3.1 Treść

Szyfr Cezara

2.3.2 Rozwiązanie

Podobno działa, ocena nieznana.

```
/// program główny
main()
{
    if(pvm_parent() == PvmNoParent)
        rodzic();
    else
        potomek();
    return 0;
}
```

```
/// Potomek
void potomek()
{
    int masterid, dlugosc, pozycja;
    char znak;
    masterid = pvm_parent();     /*kto mnie stworzyl*/
    while(1)
    {
        pvm_recv(-1,100);
        pvm_upkbyte(&znak,1,1);
        pvm_upkint(&pozycja,1,1);
        if(znak != '␣')
        {
            if (((znak >= 'A') && (znak <= 'Z')) ||
                ((znak >= 'a') && (znak <= 'z')))
            {
                if (znak == 'x')
                    znak = 'a';
                if (znak == 'y')
                    znak = 'b';
                if (znak == 'z')
                    znak = 'c';
                if (znak == 'X')
                    znak = 'A';
                if (znak == 'Y')
                    znak = 'B';
                if (znak == 'Z')
                    znak = 'C';
            }
            else
                znak += 3;
        }
        pvm_initsend(PvmDataDefault);
        pvm_pkbyte(&znak,1,1);
        pvm_pkint(&pozycja,1,1);
        pvm_send(masterid,200);
    }
}
```

```
/// Rodzic
void rodzic()
{
    char* text = "ala ma kota";
    char* text2;
    int length = strlen(text);
    int tidmaster, ilhost, ilarch, *tid, i, procesy = 0, budif, temp_id,
       zm1, zm2;
    struct pvmhostinfo *info;
    if ((tidmaster = pvm_mytid()) < 0) exit(1);
    pvm_config(&ilhost, &ilarch, &info);
    tid = (int*) calloc (ilhost, sizeof(int));
    text2 = (char*) calloc (length + 1, 1);
    text2[length] = '\0';
    for(i = 0; i < ilhost; ++i) {
        if(i >= length) break;
        pvm_spawn("/home/pvm/pvm3/sekcja15/bin/LINUX/hello", 0, PvmTaskHost
            , info[i].hi_name, 1, &tid[i]);
        pvm_initsend(PvmDataDefault);
        pvm_pkbyte((text + i), 1, 1);
        pvm_pkint(&i, 1, 1);
        pvm_send(tid[i], 100);
        ++procesy;
    }
    i = ilhost;
    while (1) {
        budif = pvm_recv(-1, 200);
        char temp;
        int recIndex;
        pvm_upkbyte(&temp, 1, 1);
        pvm_upkint(&recIndex, 1, 1);
        text2[recIndex] = temp;
        if (i >= length)    break;
        else {
            pvm_bufinfo(budif, &zm1, &zm2, &temp_id);
            printf("Od: %d \n", temp_id);
            pvm_initsend(PvmDataDefault);
            pvm_pkbyte((text + i), 1, 1);
            pvm_pkint(&i, 1, 1);
            pvm_send(temp_id, 100);
            ++i;
        }
    }
    for (i = 0; i < (ilhost-1); ++i) {
        budif = pvm_recv(-1, 200);
        char temp;
        int recIndex;
        pvm_upkbyte(&temp, 1, 1);
        pvm_upkint(&recIndex, 1, 1);
        text2[recIndex] = temp;
        pvm_bufinfo(budif, &zm1, &zm2, &temp_id);
        printf("Od: %d \n", temp_id);
        pvm_kill(temp_id);
    }
    printf("Tekst: %s\n", text2); pvm_exit();
}
```

3 Egzamin 2012, T1, Hafed Zghidi

3.1 Treść

Naszym zadaniem jest przeszukiwanie fragmentu obrazu (*wzorzec*) o rozmiarze $n \times n$ w obrazie *Obraz* o rozmiarze $m \times n (m >> n)$. Do tego celu służy nam odpowiednia funkcja

```
int find_pattern(x, y, n)
```

gdzie x, y to współrzędne w obrazie, a n rozmiar wzorca. Funkcja zwraca 1, jeśli wzorzec znajduje się w obrazie lub 0 jeśli go nie ma. Wzorzec może się powtarzać w obrazie.
Napisać w oparciu o środowisko PVM kod programu rodzica zliczającego ile razy wzorzec znajduje się w obrazie. Program powinien zapewnić równoległe rozwiązanie tego zadania z zachowaniem dynamicznego podziału zadań. Dodatkowe punkty można uzyskać za zapewnienie lepsze granulacji podziału zadań.

3.2 Propozycja rozwiązania

```
#include <stdio.h>
#include "pvm3.h"
#include <string.h>

void parent();
void child();
/// program główny
int main(void) {
    if(pvm_parent() == PvmNoParent)
        parent();
    else
        child();
    return 0;
}
/// rodzic
void parent()
{
    //rozmiary obrazu
    int n = 32;
    int m = 1024;
    //aktualna pozycja
    int x = 0, y = 0;

    //pvm-owe zmienne
    int tIdMaster = 0, *tId;
    int ilHost = 0, ilArch = 0;
    struct pvmhostinfo *info;

    //suma zliczen
    int sum = 0;

    //dane do przeslania, odpowiednio x y n
    int dataToSend[3];

    if( (tIdMaster = pvm_mytid()) < 0 )
        exit(0);

    //pobranie konfiguracji
    pvm_config(&ilHost, &ilArch, &info);
```

```
//stworzenie tablicy z id dzieci
tId = (int*)calloc(ilHost, sizeof(int));

//wyslanie poczatkowych danych
for(int i = 0; i < ilHost; ++i)
{
    pvm_spawn("/home/pvm/...", 0, PvmTaskHost, info[i].hi_name, 1, &tId
        [i]);

    //przygotowanie i wyslanie danych
    pvm_initsend(PvmDataDefault);

    dataToSend[0] = x;
    dataToSend[1] = y;
    dataToSend[2] = n;

    pvm_pkint(dataToSend, 3, 1);
    pvm_send(tId[i], 100);
    //przesuniecie w obrazie o pixel w prawo
    ++x;

    //jesli wyjdzie poza prawa strone, zejscie w dol
    if(x >= (m - n) ) {
        x = 0;
        ++y;
    }
    //sam koniec obrazu - zakoncz
    if(y >= (m - n) )
        break;
}
//jesli zostalo cos jeszcze do wyslania...
if(y >= (m - n) )
    while(true)
    {
        int childId = 0;

        //odbieramy dowolna paczke wyslana przez dzieci
        int bi = pvm_recv(-1, 100);
        int rcvdData = 0;

        //pobieramy jego id
        pvm_bufinfo(bi, &dlbuf, &msgtag, &childId);

        //rozpakowujemy dane, dodajemy do sumu
        pvm_upkint(&rcvdData, 1, 1);

        suma += rcvdData;

        //inicjujemy dane
        dataToSend[0] = x;
        dataToSend[1] = y;
        dataToSend[2] = n;

        pvm_initsend(PvmDataDefault);
        pvm_pkint(dataToSend, 3, 1);
```

```
            //wysylamy do dziecka, z ktorego aktualnie odebrano rezultat
            pvm_send(childId, 100);

            //tak jak poprzednio, przesuniecie w obrazie
            ++x;
            if(x >= (m - n) ) {
                x = 0;
                ++y;
            }
            if(y >= (m - n) )
                break;
        }

    //w tym momencie wyslalismy juz wszystko
    //odbieramy wiec ostatnia paczke od kazdego z dzieci
    for(int i = 0; i < ilHost; ++i)
    {
        int tmp;
        pvm_rcv(masterId, 100);
        pvm_upkint(&tmp, 1, 1);

        suma += tmp;
        pvm_kill(tId[i]);
    }
}
/// Potomek
void child()
{
    //id rodzica
    int masterId = pvm_parent();

    //tablica do ktorej zapiszemy odebrane dane
    int points[3];

    //odbieramy paczke od rodzica
    pvm_rcv(masterId, 100);
    pvm_upkint(points, 3, 1);

    //funkcja z zadania
    int result = find_pattern(points[0], points[1], points[2]);

    //odsylamy dane
    pvm_initsend(PvmDataDefault);
    pvm_pkint(result, 1, 1);
    pvm_send(masterId, 100);
}
```

4 Egzamin 2013, T1, Hafed Zghidi

4.1 Treść

Naszymi danymi wejściowymi są tabela A zawierająca 10000 łańcuchów znakowych i tabela B zawierająca 10000 wartości typu *integer* będących wartościami skrótu łańcuchów tabeli A. Napisać kod programu głównego (rodzic) realizującego w oparciu o środowisko PVM równoległe wykonanie porównywania zgodności łańcuchów z tabeli A ze skrótami z tabeli B. Program potomny ma otrzymywać łańcuch znakowy z tabeli A oraz odpowiadającą mu wartość typu *integer* z tabeli B. Program główny ma zliczyć ile par jest poprawnych, a ile nie zgadza się. Program potomny ma zostać uruchomiony na wszystkich węzłach w maszynie PVM. Rozdział zadania na podzadania dla procesów potomnych ma uwzględniać aspekt równoważenia obciążenia poszczególnych węzłów maszyny PVM (napisać tylko program rodzica).

4.2 Rozwiązanie

```
#include "pvm3.h"
#include <stdio.h>
const int RECORD_NUMBER = 10000
void fill_tables(char* input[RECORD_NUMBER], int output[RECORD_NUMBER]);

int main_pvm()
{
    // dane wejściowe
    char* input[RECORD_NUMBER];
    int output[RECORD_NUMBER];
    // zakładamy, że ta funkcja wypełnia tablice jak należy
    fill_tables(input, output);
    // liczniki wystąpień
    unsigned int true_count = 0;
    // pobranie informacji
    int ilhost, ilarch;
    struct pvmhostinfo * info;
    pvm_config(&ilhost, &ilarch, &info);
    printf("Liczba hostow: %d\n", ilhost);

    int id, tid;
    id = 0;
    int i;
    // dla każdego hosta
    for (i = 0; i < ilhost; i++, id++)
    {
        pvm_spawn("/egzamin/dziecko", 0, PvmTaskHost, info[i].hi_name, 1, &
            tid);
        if (tid < 0)
        {
            ilhost--;
            continue;
        }
        pvm_initsend(PvmDataDefault);
        // wysyłamy:
        pvm_pkint(&id, 1, 1);              // id wiersza
        pvm_pkstr(input[id]);              // łańcuch
        pvm_pkint(&output[id], 1, 1);      // suma kontrolna
        pvm_send(tid, 100);
    }
```

```
{    // ...
     int bufid, child_tid, child_id, tmp, result;
     while (id < RECORD_NUMBER)
     {
          bufid = pvm_recv(-1, 200);
          pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
          // pobranie danych
          pvm_upkint(&child_id, 1, 1);     // id wiersza
          pvm_upkint(&result, 1, 1);       // wynik
          // dziecko zwraca 0, jeśli suma nie była poprawna
          // lub 1, jeśli była poprawna
          true_count = true_count + result;
          // wysłanie nowych danych
          pvm_initsend(PvmDataDefault);
          pvm_pkint(&id, 1, 1);            // id kolejnego wiersza
          pvm_pkstr(input[id]);            // łańcuch
          pvm_pkint(&output[id], 1, 1);    // suma kontrolna
          pvm_send(child_tid, 100);
          id++;
     }
     // odbieranie ostatnich wyników od potomków
     for (i = 0; i < ilhost; i++)
     {
          bufid = pvm_recv(-1, 200);
          pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
          pvm_upkint(&child_id, 1, 1);     // pobranie id wiersza
          pvm_upkint(&result, 1, 1);       // wynik
          true_count = true_count + result;
     }
     // wypisanie wyniku
     printf("Liczba poprawnych sum kontrolnych = %d\n", true_count);
     printf("Liczba niepoprawnych sum kontrolnych = %d", RECORD_NUMBER -
         true_count);
     pvm_exit();

     return 0;
}
```

4.3 Propozycja rozwiązania z forum - dlaczego jest źle i możesz ujebać bezmyślnie przepisując

```
#include <stdio.h>
#include "pvm3.h"
#define SIZE        10000
#define STR_SIZE    256
#define PATH        "pvm_child"
int main()
{
    int tidmaster, *tid, ilhost, ilarch, bufid, a, b, temp, wynik;
    struct pvmhostinfo *info;
    // idea jest taka, by porównać rozmiar stringi, zapisany w drugim
       wymiarze, z wartością w 'wartosci'. Treść zadania mówi o dwóch
       tabelach w jednym wymierze, więc średnie to.
    // Porównywanie stringów na char* z liczbami boli bez gotowych funkcji,
        co? :P
    // A nas to nawet nie interesuje, bo za to odpowiada potomek, zatem
       mamy przykład popieprzenia z poplątaniem XD
    char stringi[SIZE][STR_SIZE];
    int wartosci[SIZE];
    const int ROZMIAR = 10;
    wynik = 0;
    tidmaster = pvm_mytid();          // zupełnie zbędne
    pvm_config(&ilhost, &ilarch, &info);
    // tid nie musi być tablicą, bo nigdy nie korzystamy z innego, niż
       aktualnie wykorzystywany
    tid = (int*)malloc(ilhost,sizeof(int));
    for (a = 0; a < ilhost; a++)
    {
        pvm_spawn(PATH, 0, PvmTaskHost, info[a].hi_name, 1, &tid[a]);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&ROZMIAR, 1, 1);
        temp = STR_SIZE;          // chujowe konsekwencje trzymania stałej
           przez define - brak adresu, hłehłe
        pvm_pkint(&temp, 1, 1);
        for (b = 0; b < ROZMIAR; b++)
        {
            // zabawa z dwoma wymiarami, kiedy wystarczy jeden
            // dziwna idea, raczej nieintuicyjna
            pvm_pkint(&wartosci[a * ROZMIAR + b], 1, 1);
            pvm_pkstr(stringi[a * ROZMIAR + b]);
        }
        pvm_send(tid[a], 100);  // tid jest wykorzystywany tylko w tej pę
           tli, tabela nie sensu
    }
    a = ilhost;      // nikt nie zauważył, że ma tę wartość po wyjściu z pę
       tli?
```

```
{ // ...
    while( a < SIZE/ROZMIAR ) {
        bufid = pvm_recv(-1, 200);
        pvm_upkint(&temp, 1, 1);
        wynik += temp;        // w ANSI C nie ma tego operatora
        pvm_bufinfo(bufid, &temp, &b, &tmp_id);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&ROZMIAR, 1, 1);
        temp = STR_SIZE;      // ZNOWU XD
        pvm_pkint(&temp, 1, 1);
        for (b = 0; b < ROZMIAR; b++)
        {
            pvm_pkint(&wartosci[a * ROZMIAR + b], 1, 1);
            pvm_pkstr(stringi[a * ROZMIAR + b]);
        }
        pvm_send(tmp_id, 100);
        a++;
    }
    for(a = 0; a < ilhost; a++)
    {
        bufid = pvm_recv(-1, 200);
        pvm_upkint(&temp, 1, 1);
        wynik += temp;        // co cóż
        pvm_bufinfo(bufid, &temp, &b, &tmp_id);
        pvm_kill(tmp_id);
    }
    printf("wynik %d",WYNIK);
    return 0;
}
```

4.4 2015, 0 termin, Hafed Zighdi

4.4.1 Treść

Naszym zadaniem jest przeszukanie fragmentu obrazu (*wzorzec*) o rozmiarze nxn w obrazie *Obraz* o rozmiarze $mxm (m >> n)$. Do tego celu służy funkcja *int find_pattern(x, y, n)*, gdzie x, y to współrzędne w obrazie, a n rozmiar wzorca. Funkcja zwraca 1 jeśli wzorzec znajduje się w obrazie lub 0 jeśli go nie ma. Wzorzec możne powtarzać się w obrazie.
Napisać w oparciu o środowisko PVM kod programu rodzica zliczającego ile razy wzorzec znajduje się w obrazie. Program powinien zapewnić równoległe rozwiązywanie zadania z zachowaniem dynamicznego podziału zadań. Dodatkowe punkty można uzyskać za zapewnienie lepszej granulacji podziału zadań.

4.4.2 Rozwiązanie

```
#include "pvm3.h"
extern int m;                   // wymiar obraz
extern int n;                   // wymiar wzorca

int main()
{
    int liczbaWystapien = 0;
    // tablica pikseli, zakladamy ze tak reprezentowany jest obraz
    int obraz[m][m];
    int ilhost, ilarch;          //parametry z PVMa
    struct pvmhostinfo * info;
    pvm_config(&ilhost, &ilarch, &info);
    printf("Liczba hostow: %d\n", ilhost);

    int id_wiersza = 0;
    int tid;
    // Dla każdego hosta - inicjujemy go
    for ( i = 0; i < ilhost; i++ )
    {
        //powołanie potomka
        pvm_spawn( "/potomek", 0, PvmTaskHost, info[i].hi_name, 1, &tid);
        //jeśli nie udało się powołać potomka, zmniejszamy liczbę hostów i
            kontynuujemy
        if ( tid < 0 )
        {
            ilhost--;
            continue;
        }
        pvm_initsend(PvmDataDefault);
        // wysyłamy:
        // elementy obrazu
        for (int i = 0; i < n; i++)
        {
            //pakowanie całego wiersza
            pvm_pkint(&obraz[id_wiersza + i][0], m, 1);
        }
        pvm_send(tid, 100);       //wysłanie wiersza
        id_wiersza++;
    }
```

```
{   // ...
    //// Wykonywanie programu aż do przedostatniej pętli
    int bufid, child_tid, child_liczba_wystapien, tmp;
    while ( id_wiersza < (m - n) )
    {
        // odebranie info o zakończeniu pracy
        bufid = pvm_recv(-1, 200);
        pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
        // pobranie liczby wystapien
        pvm_upkint(&child_liczba_wystapien, 1, 1);
        // zwiekszenie wystapien wzorca
        liczbaWystapien += child_liczba_wystapien;
        // wysłanie nowych danych
        pvm_initsend(PvmDataDefault);
        // nowa czesc obrazu
        for (int i = 0; i < n; i++)
        {
            pvm_pkint(&obraz[id_wiersza + i][0], m, 1);
        }
        pvm_send(child_tid, 100);
        id_wiersza++;
    }
    //// Odebranie ostatnich danych
    for (i = 0; i < id_wiersza - ilhost + 1; i++ )
    {
        bufid = pvm_recv(-1, 200);
        pvm_bufinfo(bufid, &tmp, &tmp, &child_tid);
        // pobranie liczby wystapien
        pvm_upkint(&child_liczba_wystapien, 1, 1);
        // zwiekszenie wystapien wzorca
        liczbaWystapien += child_liczba_wystapien;
    }
    printf("Liczba wystapien wzorca: %d", liczbaWystapien);
    pvm_exit();
    return 0;
};
```

5 Java Spaces

5.1 Wstęp z laborek

5.1.1 Treść

Napisać program zawierający jednego Nadzorcę oraz wielu Pracowników. Nadzorca przekazuje do JavaSpace 2 równe tablice zawierające obiekty typu Integer, a następnie otrzymuje wynikową tablicę zawierającą sumy odpowiadających sobie komórek. Operację dodawania mają realizować Pracownicy.

5.1.2 Rozwiązanie

Zadanie obliczania sumy tabel dzielimy na dwie części: *Task* oraz *Result*. *Taski* są generowane przez *Nadzorcę* i przekazywane *Pracownikom*, ci zaś wykonują zadanie i tworzą obiekty klasy *Result*, a następnie przekazuję je *Nadzorcy*. *Nadzorca* je odbiera, kompletuje i ew. coś z nimi robi.

Nadzorca przydziela tyle zadań, ile potrzebuje, z kolei *Pracownicy* działają w nieskończoność. Aby zakończyć ich pracę, *Nadzorca* musi wysłać zadania z tzw. zatrutą pigułką (ang. *Poisoned Pill*), czyli obiekt zadania z nietypowym parametrem, który sygnalizuje zakończenie pracy. Może to być np. *Boolean* o wartości *false*, *Integer* o wartości -1, itp. Składowymi klas implementujących interfejs Entry nie mogą być typu prostego (int, double itp.), muszą być opakowane (Integer, Double itp.). Najbezpieczniej dawać je wszędzie.

```
/**
* @author Son Mati
* @waifu Itsuka Kotori
*/
public class Task implements Entry {
    public Integer cellID;  // ID komórki tabeli
    public Integer valueA;  // wartość z tabeli A
    public Integer valueB;  // wartość z tabeli B
    public Boolean isPill;  // czy zadanie jest zatrutą pigułką

    // Domyślny konstruktor, musi się znajdować
    public Task() {
        this.cellID = this.valueA = this.valueB = null;
        this.isPill = false;
    }

    public Task(Integer entryID, Integer valueA, Integer valueB, Boolean
        isPill) {
        this.cellID = entryID;
        this.valueA = valueA;
        this.valueB = valueB;
        this.isPill = isPill;
    }
}
```

```java
/**
* @author Son Mati
* @waifu Itsuka Kotori
*/
public class Result implements Entry {
    public Integer cellID, value;
    public Result() {
        this.cellID = this.value = null;
    }
    public Result(final Integer EntryID, final Integer Value) {
        this.cellID = EntryID;
        this.value = Value;
    }
}
```

```java
public class Client {
    protected Integer defaultLease = 100000;
    protected JavaSpace space;
    protected Lookup lookup;
    public Client() {
        lookup = new Lookup(JavaSpace.class);
    }
}
```

```java
/**
* @author Son Mati
* @waifu Itsuka Kotori
*/
public class Worker extends Client {
    public Worker() {
    }
    public void startWorking() {
        while(true) {
            try {
                this.space = (JavaSpace)lookup.getService();
                Task task = new Task();
                task = (Task) space.take(task, null, defaultLease);
                if (task.isPill == true)
                {
                    space.write(task, null, defaultLease);
                    System.out.println("Koniec pracy workera.");
                    return;
                }
                Integer res = task.valueA + task.valueB;
                Result result = new Result(task.cellID, res);
                space.write(result, null, defaultLease);
            }
            catch (Exception ex) {}
        }
    }
    public static void main(String[] args) {
        Worker w = new Worker();     // utworzenie obiektu
        w.startWorking();            // realizacja zadan
    }
}
```

```
/**
* @author Son Mati
* @waifu Itsuka Kotori
*/
public class Supervisor extends Client {
    static final Integer INT_NUMBER = 125;
    public Integer[] TableA = new Integer[INT_NUMBER];
    public Integer[] TableB = new Integer[INT_NUMBER];
    public Integer[] TableC = new Integer[INT_NUMBER];
    // konstruktor
    public Supervisor() {
    }
    // wygenerowanie zawartości tablic
    public void generateData() {
        Random rand = new Random();
        for (int i = 0; i < INT_NUMBER; ++i) {
            TableA[i] = rand.nextInt(INT_NUMBER);
            TableB[i] = rand.nextInt(INT_NUMBER);
            TableC[i] = 0;
        }
    }
    // rozpoczecie pracy
    public void startProducing() {
        try {
            this.space = (JavaSpace)lookup.getService();
            // utworzenie zadania
            for (Integer i = 0; i < INT_NUMBER; ++i) {
                Task task = new Task(i, this.TableA[i], this.TableB[i],
                    false);
                space.write(task, null, defaultLease);
            }
            // pobranie wyniku zadania
            System.out.println("Tablica C:");
            for(Integer i = 0 ; i < INT_NUMBER; ++i) {
                Result result = new Result();
                result = (Result) space.take(result, null, defaultLease);
                TableC[result.cellID] = result.value;
            }
            // utworzenie zatrutej pigulki na sam koniec
            Task poisonPill = new Task(null, null, null, true);
            space.write(poisonPill, null, defaultLease);
        }
        catch (Exception ex) {
        }
    }

    public static void main(String[] args) {
        // utworzenie obiektu
        Supervisor sv = new Supervisor();
        // utworzenie zadan
        sv.generateData();
        sv.startProducing();
    }
}
```

5.2 Zadanie 1

5.2.1 Treść

Napisać program odbierający z przestrzeni JavaSpace kolejno 100 obiektów klasy Zadanie posiadające w atrybucie typ (typu całkowitego) wartość 15 i dla każdego obiektu Zadanie wygenerować i umieścić w przestrzeni JavaSpace obiekt klasy Silnia posiadający jako atrybut... (dalej nie pamiętam dobrze) wartość będącą silnią wartości uzyskanej z liczba(typu całkowitego) z klasy Zadanie.

5.3 2010, I termin, Adam Duszeńko

5.3.1 Treść

Napisać kod programu głównego zarządzającego równoległym wykonywaniem zadania w maszynie JavaSpace polegającym na wyznaczeniu zbioru klatek video zawierających ruch. Wykrywanie ruchu ma odbywać się w procesorach wykonawczych na zasadzie porównania różnicowego, czyli wymaga poddania analizie dwóch kolejnych klatek. W tym celu program główny posługując się *byte[] GetImage()* (przyjąć, że jest zdefiniowana i zaimplementowana) ma pobierać kolejne klatki obrazu i umieszczać je w przestrzeni JavaSpace, wraz z jej kolejnym numerem (numerowania ma odbywać się na poziomie programu głównego). Program główny kończy wysyłania zadań gdy funkcja *GetImage* zwróci wartość *NULL*. Jako wynik swojego działania programy wykonawcze zwracają obiekt odpowiedzi zawierający numer pierwszego obrazu z analizowanej pary oraz wartość logiczną czy para była identyczna czy też zawierała wykryty ruch. Na zakończenie działania program główny po zebraniu wszystkich odpowiedzi powinien wypisać numery obrazów dla których wykryto ruch oraz zakończyć procesy wykonawcze rozsyłając "zatrutą pigułkę". Zaproponować strukturę obiektu zadania i odpowiedzi.

5.3.2 Propozycja rozwiązania 1

Nie jest do końca prawidłowa, ponieważ kod nie jest spójny i nie wiadomo czy analizuje pary klatek.

```
public class Image implements Entry {
    //należy pamiętać o tym aby każde pole było publiczne!
    public byte[] frame;
    public Integer id;
    //wymagane konstruktory
    public Image() {}
    public Image(Integer id, byte[] frame) {
        this.id = new Integer(id);
        this.frame = frame;
    }
}
```

```
//Dane przesyłane jako odpowiedź
public class Result implements Entry {
    public Integer id;
    public Boolean move;
    public Result() {}
    public Result(Integer id, Boolean move) {
        this.id = new Integer(id);
        this.move = new Boolean(move);
    }
}
```

```
public class Program {
    public int defaultLease = 100000;
    public int id = 1;

    public void producer() {
        byte [] img1, img2;
        img1 = getImage();
        try {
            Lookup lookup = new Lookup(JavaSpace.class);
            JavaSpace space = (JavaSpace) lookup.getService();
            img2 = getImage();
            while(true) {
                img = getImage();    // dostarczone w zadaniu
                if (img == null || img2 == null) break; // null = koniec
                Package data = new Data(id, img1, id + 1, img2);
                space.write(data, null, defaultLease);  // paczka do space
                img1 = img2;
                id++;
            }
        // bo breaku przełączamy się w tryb odbierania
        for (int i = 1; i < id; i++) {
            Result result = (Result) space.takeIfExsists(new Result(), null
                , defaultLease);
            if (result.move)
                System.out.println("Ruch obrazków: " + result.id1 + " " +
                    result.id2);
        }
        space.write(new Image(), null, defaultLease);
        } catch (Exception e) {}
    }

    public void consumer() {
        try {
            Lookup lookup = new Lookup(JavaSpace.class);
            JavaSpace space = (JavaSpace) lookup.getService();
            int i = 0;
            while(true) {
                Image img1 = new Image();    img1.id = i++;
                Image img2 = new Image();    img2.id = i;
                img1 = (Image) space.take(img1, null, defaultLese);
                img2 = (Image) space.read(img2, null, defaultLese);
                //czy wysłano "zatrutą pigułkę"
                if (img2.frame == null && img2.id == null) break;
                // czy wykonano ruch na obrazkach
                if (img1.frame.equals(img2.frame)) {
                    // tego chyba nie trzeba nawet wysyłać w tym zadaniu
                    result = new Result(img2.id, false);
                } else {
                    result = new Result(img2.id, true);
                }
                // wysyłanie wyniku do space
                space.write(result, null, defaultLease);
            }
        } catch (Exception e) {}
    }
}
```

5.4 2011, I termin, Adam Duszeńko

5.4.1 Treść

Napisać program umieszczający w przestrzeni JavaSpace **10** obiektów zadań zawierających **dwa** pola typu całkowitego oraz **dwa** pola typu łańcuch znakowy (zawartość nieistotna, różna od NULL), podać deklarację klasy zadań. Następnie odebrać z przestrzeni kolejno **10** obiektów klasy *Odpowiedź* o atrybutach *id* typu *Integer* oraz *wynik* typu *Integer* posiadające w atrybucie id wartość **15**, a następnie wszystkie z atrybutem $id = \mathbf{110}$. Przyjąć, że klasa *Odpowiedź* jest już zdefiniowana zgodnie z powyższym opisem.

5.5 2012, I termin, Adam Duszeńko

5.5.1 Treść

Napisać program umieszczający w przestrzeni JavaSpace **1000** obiektów zadań zawierających **trzy** pola typu całkowitego oraz **dwa** pola typu łańcuch znakowy (zawartość nieistotna, różna od NULL), podać deklarację klasy zadań. Następnie odebrać z przestrzeni kolejno **1000** obiektów klasy *Odpowiedź* o atrybutach *id* typu *Integer* oraz *wynik* typu *Integer* posiadające w atrybucie id wartość **35**, a następnie wszystkie z atrybutem $id = \mathbf{10}$. Przyjąć, że klasa *Odpowiedź* jest już zdefiniowana zgodnie z powyższym opisem.

5.6 2013, I termin

5.6.1 Treść

Napisać program umieszczający w przestrzeni JavaSpace **200** obiektów zadań zawierających **dwa** pola typu całkowitego oraz **dwa** pola typu łańcuch znakowy (zawartość nieistotna, różna od NULL), podać deklarację klasy zadań. Następnie odebrać z przestrzeni kolejno **100** obiektów klasy *Odpowiedź* o atrybutach *id* typu *Integer* oraz *wynik* typu *Integer* posiadające w atrybucie id wartość **35**, a następnie wszystkie z atrybutem *id* = **10**. Przyjąć, że klasa *Odpowiedź* jest już zdefiniowana zgodnie z powyższym opisem.

5.6.2 Rozwiązanie

Klasa Zadanie

```
// deklaracja klasy, muszą być widoczne:
// implementacja interfejsu Entry
public class Zadanie implements Entry {
    // publiczne składowe, opakowujące typy zmiennych
    public Integer liczba;
    public String napis1;
    public String napis2;
    public Boolean poisonPill;
    // konstruktor domyślny, obowiązkowy
    public Zadanie() {
        Random rand = new Random();
        this.liczba = rand.nextInt();
        this.napis1 = Integer.toString(rand.nextInt());
        this.napis2 = Integer.toString(rand.nextInt());
        this.poisonPill = false;
    }
    public Zadanie(Integer liczba, String napis1, String napis2, Boolean
        poisonPill) {
        this.liczba = liczba;
        this.napis1 = napis1;
        this.napis2 = napis2;
        this.poisonPill = poisonPill;
    }
}
```

Nadrzędna klasa Klienta

```
/**
 * @author Son Mati & Doxus
 */
public class Client {
    protected Integer defaultLease = 100000;
    protected JavaSpace space;
    protected Lookup lookup;
    public Client() {
        lookup = new Lookup(JavaSpace.class);
    }
}
```

Klasa Nadzorcy

```
/**
* @author Son Mati & Doxus
*/
public class Boss extends Client {
    // domyślne wartości dla zadania
    static final int DEFAULT_TASK_NUMBER = 200;
    static final int DEFAULT_MAX_MISSES = 100;

    Integer taskNumber;
    Integer maxMisses;

    public Integer getTaskNumber() {
        return taskNumber;
    }
    public Integer getMaxMisses() {
        return maxMisses;
    }
    // obowiązkowy domyślny konstruktor
    public Boss() {
        taskNumber = DEFAULT_TASK_NUMBER;
        maxMisses = DEFAULT_MAX_MISSES;
    }
    public Boss(Integer taskNumber, Integer maxMisses) {
        this.taskNumber = taskNumber;
        this.maxMisses = maxMisses;
    }
    /**
     * Wygenerowanie zadania z losowymi wartościami
     * @param id identyfikator
     * @param poisonPill pigułka, tak czy nie
     */
    public Zadanie generateTask(int id, boolean poisonPill) {
        Random rand = new Random();
        return new Zadanie(id, Integer.toString(rand.nextInt(1000)),
        Integer.toString(rand.nextInt(1000)), poisonPill);
    }
    /**
     * Utworzenie zadaniów
     * @param count ilość zadaniów
     */
    public void createTasksInJavaSpace(int count) {
        try {
            this.space = (JavaSpace)lookup.getService();
            for (int i = 0; i < count; ++i) {
                Zadanie zad = this.generateTask(i, false);
                space.write(zad, null, defaultLease);
                System.out.println("Wygenerowałem zad " + i + " o stringach
                    " + zad.napis1 + " i " + zad.napis2);
            }
        }
        catch(RemoteException | TransactionException ex) {
            System.out.println("Dupa XD");
        }
    }
```

```
public class Boss extends Client {
    /**
    * Uzyskanie odpowiedzi
    * @param id odpowiedzi, ktora nas interesuje
    * @param costam interesujący nas wynik
    * @param count liczba odpowiedzi do odbioru
    */
    public Integer receiveData(Integer id, Integer costam, Integer count) {
        Integer found = 0;
        try {
            this.space = (JavaSpace)lookup.getService();
            Odpowiedz wzor = new Odpowiedz(id, costam);
            for (int i = 0; i < count; i++) {
                // odczyt blokujący, zatrzymuje przepływ dopóki odp się nie
                    pojawi
                Odpowiedz wynik = (Odpowiedz) space.takeIfExists(wzor, null
                    , defaultLease);
                if (wynik != null) {
                    System.out.println("Odpowiedz: id = " + wynik.getId() +
                        ", wynik = " + wynik.getWynik());
                    found++;
                }
            }
        } catch (UnusableEntryException | TransactionException |
            InterruptedException | RemoteException ex) {
        }
        return found;
    }
    /**
    * ZATRUJ DZIECIACZKI XD
    */
    public void poisonKids() {
        // utworzenie zatrutej pigulki
        Zadanie poisonPill = new Zadanie(null, null, null, true);
        try {
            space.write(poisonPill, null, defaultLease);
        } catch (TransactionException | RemoteException ex) {
        }
    }

    public static void main(String[] args) {
        Integer misses = 0;
        Boss boss = new Boss();
        boss.createTasksInJavaSpace(boss.getTaskNumber());
        boss.receiveData(35, null, 100);
        // boss odbiera pozostałe odpowiedzi, o id 10, dopóki nie trafi na
            pewną liczbę chybień
        while(misses < boss.getMaxMisses()) {
            if (boss.receiveData(10, null, 1) == 0)
            misses++;
        }
        boss.poisonKids();
    }
}
```

Klasa Pracownika

```
/**
* @author Son Mati & Doxus
*/
public class Sidekick extends Client {
    // obowiązkowy domyślny konstruktor
    public Sidekick() {
    }
    // praca
    public void zacznijMurzynic() {
        while(true) {
            try {
                Random rand = new Random();
                this.space = (JavaSpace)lookup.getService();
                Zadanie zad = new Zadanie(null, null, null, null);
                zad = (Zadanie) space.takeIfExists(zad, null, defaultLease)
                    ;
                if (zad != null) {
                    if (zad.poisonPill == true) {
                        space.write(zad, null, defaultLease);
                        return;
                    }
                    System.out.println("Odebrałem zadanie o id " + zad.
                        liczba
                    + " i napisach " + zad.napis1 + " i " + zad.napis2);
                }
                Odpowiedz odp = new Odpowiedz(rand.nextInt(51), rand.
                    nextInt(1000));
                space.write(odp, null, defaultLease);
            } catch (TransactionException | RemoteException |
                UnusableEntryException | InterruptedException ex) {
                Logger.getLogger(Sidekick.class.getName()).log(Level.SEVERE
                    , null, ex);
            }
        }
    }
    // obowiązkowy Run
    public static void main(String[] args) {
        Sidekick murzyn = new Sidekick();
        murzyn.zacznijMurzynic();
    }
}
```

5.7 2014, I termin, Adam Duszeńko

5.7.1 Treść

Napisać kod programu głównego wykonawczego do przetwarzania z wykorzystaniem maszyny JavaSpace przetwarzającego obiekty zadań zawierające dwie wartości całkowite, oraz numer obiektu i flagę logiczną początkowo zawierającą wartość *FALSE*. W momencie pobrania obiektu zadania program wykonawczy ma podmienić w przestrzeni JavaSpace pobrany obiekt na ten sam, ale z flagą ustawioną na wartość *TRUE*. Przetwarzanie obiektu realizowane jest w funkcji *int check(int, int)* do której należy przekazać wartości z obiektu zadania. PO skończeniu przetwarzania zadania, przed zwróceniem wyniku, należy usunąć z przestrzeni JavaSpace obiekt przetwarzanego zadania. Wynik funkcji *check* należy umieścić w obiekcie wynikowym którego strukturę proszę zaproponować. Obsłużyć koniec działania programu przez skonsumowanie "zatrutej pigułki".

5.7.2 Propozycja rozwiązania 1

5.8 2015, 0 termin, Adam Duszeńko

5.8.1 Treść

Napisać program umieszczający w przestrzeni JavaSpace **1000** obiektów zadań zawierających **dwa** pola typu całkowitego oraz **dwa** pola typu łańcuch znakowy (zawartość nieistotna, różna od NULL), podać deklarację klasy zadań. Następnie odebrać z przestrzeni **20** obiektów klasy *Odpowiedź* o atrybutach *id* typu *Integer* oraz *wynik* typu *Integer* posiadające w atrybucie id wartość **50** (przyjąć, że klasa *Odpowiedź* jest już zdefiniowana zgodnie z powyższym opisem).

5.8.2 Rozwiązanie

Działające i przetestowane w warunkach domowych na Jini.

Klasa Zadanie

```
/**
* @author Son Mati & Doxus
*/
// deklaracja klasy, muszą być widoczne:
// implementacja interfejsu Entry
public class Zadanie implements Entry {
    // publiczne składowe, muszą być wielkich typów opakowujących
    public Integer liczba;
    public String napis1;
    public String napis2;
    public Boolean poisonPill;
    // konstruktor domyślny, wymagany
    public Zadanie() {
        Random rand = new Random();
        this.liczba = rand.nextInt();
        this.napis1 = Integer.toString(rand.nextInt());
        this.napis2 = Integer.toString(rand.nextInt());
        this.poisonPill = false;
    }
    // konstruktor z parametrami
    public Zadanie(Integer liczba, String napis1, String napis2, Boolean
        poisonPill) {
        this.liczba = liczba;
        this.napis1 = napis1;
        this.napis2 = napis2;
        this.poisonPill = poisonPill;
    }
}
```

Klasa nadzorcy

```
/**
 * @author Son Mati & Doxus
 */
public class Boss extends Client {
    // liczba zadań do wykonania
    static final int TASK_NUMBER = 1000;
    // obowiązkowy domyślny konstruktor
    public Boss() {
    }
    /**
     * Wygenerowanie zadania z losowymi wartościami
     * @param count ilość zadaniów
     */
    public Zadanie generateTask(int id, boolean poisonPill) {
        Random rand = new Random();
        return new Zadanie(id, Integer.toString(rand.nextInt(1000)),
            Integer.toString(rand.nextInt(1000)), poisonPill);
    }
    /**
     * Utworzenie zadaniów
     * @param count ilość zadaniów
     */
    public void createTasksInJavaSpace(int count) {
        try {
            this.space = (JavaSpace)lookup.getService();
            for (int i = 0; i < count; ++i) {
                Zadanie zad = this.generateTask(i, false);
                space.write(zad, null, defaultLease);
                System.out.println("Wtgenerowałem zad " + i + " o stringach
                    " + zad.napis1 + " i " + zad.napis2);
            }
        }
        catch(RemoteException | TransactionException ex) {
            System.out.println("Dupa XD");
        }
    }
```

```
public class Boss extends Client {
    /**
    * Uzyskanie odpowiedzi
    * @param id odpowiedzi, ktora nas interesuje
    * @param costam interesujący nas wynik
    * @param count liczba odpowiedzi do odbioru
    */
    public void receiveData(Integer id, Integer costam, Integer count) {
        try {
            this.space = (JavaSpace)lookup.getService();
            Odpowiedz wzor = new Odpowiedz(id, costam);
            for (int i = 0; i < count; i++) {
                // odczyt blokujący, zatrzymuje przepływ dopóki odp się nie
                    pojawi
                Odpowiedz wynik = (Odpowiedz) space.take(wzor, null,
                    defaultLease);
                System.out.println("Odpowiedz: id = " + wynik.getId() + ",
                    wynik = " + wynik.getWynik());
            }
        } catch (UnusableEntryException | TransactionException |
            InterruptedException | RemoteException ex) {
            Logger.getLogger(Boss.class.getName()).log(Level.SEVERE, null,
                ex);
        }
    }
    /**
     * ZATRUJ DZIECIACZKI XD
     */
    public void poisonKids() {
        // utworzenie zatrutej pigulki
        Zadanie poisonPill = new Zadanie(null, null, null, true);
        try {
            space.write(poisonPill, null, defaultLease);
        } catch (TransactionException | RemoteException ex) {
            Logger.getLogger(Boss.class.getName()).log(Level.SEVERE, null,
                ex);
        }
    }
    /**
     * Obowiązkowy Run dla nadzorcy
     */
    public static void main(String[] args) {
        Boss boss = new Boss();
        boss.createTasksInJavaSpace(TASK_NUMBER);
        boss.receiveData(50, null, 20);
        boss.poisonKids();
    }
}
```

Klasa Pracownika

```
/**
 * @author Son Mati & Doxus
 */
public class Sidekick extends Client {
    // obowiązkowy domyślny konstruktor
    public Sidekick() {
    }
    // rozpoczęcie pracy
    public void zacznijMurzynic() {
        while(true) {
            try {
                Random rand = new Random();
                this.space = (JavaSpace)lookup.getService();
                Zadanie zad = new Zadanie(null, null, null, null);
                zad = (Zadanie) space.takeIfExists(zad, null, defaultLease)
                    ;
                if (zad != null) {
                    if (zad.poisonPill == true) {
                        space.write(zad, null, defaultLease);
                        return;
                    }
                    System.out.println("Odebrałem zadanie o id " + zad.
                        liczba
                    + " i napisach " + zad.napis1 + " i " + zad.napis2);
                }
                Odpowiedz odp = new Odpowiedz(rand.nextInt(51), rand.
                    nextInt(1000));
                space.write(odp, null, defaultLease);
            } catch (TransactionException | RemoteException |
                UnusableEntryException | InterruptedException ex) {
                Logger.getLogger(Sidekick.class.getName()).log(Level.SEVERE
                    , null, ex);
            }
        }
    }
    // obowiązkowy punkt wejścia
    public static void main(String[] args) {
        Sidekick murzyn = new Sidekick();
        murzyn.zacznijMurzynic();
    }
}
```

6 CUDA

6.1 2013, 1 termin, Hafed Zighdi

6.1.1 Treść

W oparciu o środowisko CUDA napisać równoległy program zapewniający wyszukiwanie minimum i maksimum elementów macierzy A i B. Można zakładać, że macierze wejściowe są kwadratowe. Program powinien zwracać 2 macierze zawierające min oraz max elementów macierzy wejściowych:

- minMatrix[i, j] = min(A[i, j], B[i, j])
- maxMatrix[i, j] = max(A[i, j], B[i, j])

należy zaproponować kod kernela oraz jego wywołanie z kodem odpowiedzialnym za przydział pamięci z podziałem na bloki w dwóch wymiarach.

6.1.2 Rozwiązanie

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define MATRIX_SIZE 10

__global__ void transposeKernel(int * matrixA, int * matrixB, int *
    matrixMin, int * matrixMax)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < MATRIX_SIZE && j < MATRIX_SIZE)
    {
        int index = i * MATRIX_SIZE + j;
        matrixMin[index] = matrixA[index] > matrixB[index] ? matrixB[index]
            : matrixA[index];
        matrixMax[index] = matrixA[index] < matrixB[index] ? matrixB[index]
            : matrixA[index];
    }
}

cudaError_t calculateMinMaxWithCuda(int ** matrixA, int ** matrixB, int **
    matrixMin, int ** matrixMax)
{
    // zdefiniowanie macierzy, która jest w pamięci karty
    int * dev_a;
    int * dev_b;
    int * dev_max, dev_min;
    size_t size = MATRIX_SIZE * MATRIX_SIZE;
    // cudne dynksy
    cudaError_t cudaStatus;
    cudaStatus = cudaSetDevice(0);
    // alokacja pamięci po 100 elementów
    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    cudaStatus = cudaMalloc((void**)&dev_min, size * sizeof(int));
    cudaStatus = cudaMalloc((void**)&dev_max, size * sizeof(int));
```

```
{    // Spłaszczamy macierze do jednego wymiaru
    // Kopiujemy po wierszu, 10 x 10
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        cudaMemcpy((dev_a + i * MATRIX_SIZE), matrixA[i], MATRIX_SIZE *
            sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy((dev_b + i * MATRIX_SIZE), matrixB[i], MATRIX_SIZE *
            sizeof(int), cudaMemcpyHostToDevice);
    }
    // bloczek z wątkami 3x3
    dim3 block_size(3, 3);
    // liczba potrzebnych bloków, tak żeby size = liczba wątków
    int block_count = MATRIX_SIZE / block_size.x + (MATRIX_SIZE %
        block_size.x == 0 ? 0 : 1);

    minMaxKernel << <grid_size, block_size >> >(dev_a, dev_b, dev_min,
        dev_max);
    // przywrócenie dwóch wymiarów i przepisanie wyników
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        cudaMemcpy(matrixMin[i], (dev_min + i * MATRIX_SIZE), MATRIX_SIZE *
            sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(matrixMax[i], (dev_max + i * MATRIX_SIZE), MATRIX_SIZE *
            sizeof(int), cudaMemcpyDeviceToHost);
    }
    return 1;
}

int main()
{
    int ** matrixA, ** matrixB;
    int ** matrixMin, ** matrixMax;
    matrixA = (int**)malloc(sizeof(int*)* MATRIX_SIZE);
    matrixB = (int**)malloc(sizeof(int*)* MATRIX_SIZE);
    matrixMin = (int**)malloc(sizeof(int*)* MATRIX_SIZE);
    matrixMax = (int**)malloc(sizeof(int*)* MATRIX_SIZE);
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        matrixA[i] = (int*)malloc(sizeof(int)* MATRIX_SIZE);
        matrixB[i] = (int*)malloc(sizeof(int)* MATRIX_SIZE);
        matrixMin[i] = (int*)malloc(sizeof(int)* MATRIX_SIZE);
        matrixMax[i] = (int*)malloc(sizeof(int)* MATRIX_SIZE);
    }
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        for (int j = 0; j < MATRIX_SIZE; j++)
        {
            matrixA[i][j] = rand() % 20;
            matrixB[i][j] = rand() % 20;
            matrixMin[i] = matrixMax[i] = 0;
        }
    }
    cudaError_t cudaStatus = calculateMinMaxWithCuda(matrixA, matrixB,
        matrixMin, matrixMax);
    return 0;
}
```

6.2 2015, 0 termin, Hafed Zighdi

6.2.1 Treść

W oparciu o środowisko CUDA napisać równoległy program zapewniający transpozycję macierzy kwadratowej ($a_out[i,j] = a_in[j,i]$). Należy zaproponować kod kernela oraz jego wywołanie z kodem odpowiedzialnym za przydział pamięci z podziałem na bloki w dwóch wymiarach.

6.2.2 Rozwiązanie

```
#include <stdio.h>
#include <stdlib.h>
const unsigned int MATRIX_SIZE = 5;

cudaError_t transposeWithCuda(int ** matrix, int ** matrixOut)
{
    int * dev_a, dev_b;
    size_t size = MATRIX_SIZE * MATRIX_SIZE;
    // bloczek z wątkami 2x2, zgodnie z treścią zadania
    dim3 block_size(2, 2);
    // liczba potrzebnych bloków, tak żeby size = liczba wątków
    int block_count;
    block_count = MATRIX_SIZE / block_size.x + (MATRIX_SIZE % block_size.x
       == 0 ? 0 : 1);
    // rozmiar gridu w blokach
    // bloczek jest naszą podstawową jednostką (ma 4 wątki)
    // grid ma mieć tyle bloczków ile potrzebujemy
    dim3 grid_size(block_count, block_count);
    // jakieś dynksy do cudy
    cudaError_t cudaStatus;
    cudaStatus = cudaSetDevice(0);
    //allokacja tablic w pamięci karty graficznej
    cudaStatus = cudaMalloc((void**) &dev_a, size * sizeof(int));
    cudaStatus = cudaMalloc((void**) &dev_b, size * sizeof(int));
    // Spłaszczamy macierze do jednego wymiaru
    // przekopiowanie matrixa do dev_a
    for(int i = 0; i < MATRIX_SIZE; i++)
    {
        cudaStatus = cudaMemcpy((dev_a + i * MATRIX_SIZE), matrix[i],
           MATRIX_SIZE * sizeof(int), cudaMemcpyHostToDevice);
    }
    transposeKernel<<<grid_size, block_size>>>(dev_a, dev_b);
    // Konwersja z wektora 1D na macierz 2D
    // przekopiowanie dev_b do matrixOut
    for(int i = 0; i < MATRIX_SIZE; i++) {
        cudaStatus = cudaMemcpy(matrixOut[i], (dev_b + i * MATRIX_SIZE),
           MATRIX_SIZE * sizeof(int), cudaMemcpyDeviceToHost);
    }
    return 1;
}
```

```
__global__ void transposeKernel(int * a, int * out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < MATRIX_SIZE && j < MATRIX_SIZE)
    {
        out[i * MATRIX_SIZE + j] = a[j * MATRIX_SIZE + i];
    }
}

int main()
{
    int ** matrix;
    int ** matrixOut;
    matrix = (int**) malloc (sizeof(int*) * MATRIX_SIZE);
    matrixOut = (int**) malloc (sizeof(int*) * MATRIX_SIZE);

    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        matrix[i] = (int*) malloc (sizeof(int) * MATRIX_SIZE);
        matrixOut[i] = (int*) malloc (sizeof(int) * MATRIX_SIZE);
    }
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        for (int j = 0; j < MATRIX_SIZE; j++)
        {
            matrix[i][j] = rand() % 20;
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
    // Add vectors in parallel.
    cudaError_t cudaStatus = transposeWithCuda(matrix, matrixOut);

    return 0;
}
```

7 MOSIX

7.1 2013, 1 termin, Daniel Kostrzewa

7.1.1 Treść

Wykorzystując n węzłów klastra napisać funkcję, która obliczy wartość *F(x, y)*. Nagłówek funkcji ma wyglądać następująco:

```
double fun(double x, double y, int k)
```

gdzie wartością zwracaną jest obliczona wartość funkcji; k - liczba elementów iloczynu; x, y - argumenty funkcji. Założyć, że k jest podzielne przez n bez reszty.

7.1.2 Rozwiązanie

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double calculateProduct(double x, double y, double a) {
    return a * sqrt(x * y) / (2 * pow(x, 3) + 5 * y);
}

int main(int argc, char * argv[]) {
    int n = atoi(argv[1]);          // liczba węzłów klastra
    int k = atoi(argv[2]);          // liczba elementów iloczynu
    double x = atof(argv[3]);       // pierwszy argument
    double y = atof(argv[4]);       // drugi argument
    int final_product = 1;          // iloczyn wynikowy
    int process_number = k / n;
    int response_stream[2];         // strumień dla danych
    double part_result = 1;         // jebnięcie potoków
    pipe(response_stream);
    // utworzenie procesów potomnych
    int i, j;
    for (i = 0; i < process_number; i++)
    {
        if (fork() == 0)
        {
            for (j = i * process_number; j < (i + 1) * process_number; j++)
            {
                part_result = part_result * calculateProduct(x, y, j);
                write(response_stream[1], &part_result, sizeof(double));
            }
            exit(0);
        }
    }
    // odczytanie danych częściowych
    for (i = 0; i < process_number; i++)
    {
        read(response_stream[0], &part_result, sizeof(double));
        final_product = final_product * part_result;
    }
    printf("Twój szczęśliwy iloczyn to %g", final_product);
    return 0;
}
```

7.2 2015, 0 termin, Daniel Kostrzewa

7.2.1 Treść

Napisać program, który utworzy n procesów potomnych. Proces zarządzający ma wysyłać zestaw liczb do procesów potomnych (założyć, że liczba przesyłanych danych wynosi k). Procesy potomne mają w pętli wykonywać następujące czynności: czekać na liczbę wysłaną przez proces zarządzający, na podstawie odebranej liczby obliczyć pole koła (odebrana liczba jest promieniem koła), odesłać wynik do procesu zarządzającego. Proces zarządzający po wysłaniu wszystkich liczb przechodzi w stan odbierania danych i sumuje pola kół. Końcowa wartość ma zostać wyświetlona na ekranie.

7.2.2 Rozwiązanie

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159

float calculate_circle_area(int radius)
{
    return PI * pow((float) radius, 2);
}

int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);
    int k = atoi(argv[2]);
    //promien -1 oznacza ze jest to trujaca pigulka zabijajaca proces
       potomny
    int poison = -1;
    //allokacja tablicy k liczb (promieni)
    int * tab = (int*) malloc(sizeof(int) * k);
    int i;
    for (i = 0; i < k; i++)
    {
        tab[i] = rand() % 20;
    }
    int process_number = n / k;
    // 2 strumienie, jeden do wysyłania danych, drugi do odbioru odpowiedzi
    int data_stream[2], response_stream[2];
    // wyniki
    int part;
    int sum = 0;
    // jebnięcie potoków
    // odpowiedzi z procesów potomnych
    pipe(response_stream);
    // in - promienie kół
    // out - otrzymane wyniki - pola kół
    pipe(data_stream);
```

```
// utworzenie procesów potomnych
for (i = 0; i < n; i++)
{
    if (fork() == 0)
    {
        int radius;
        // wykonywanie obliczeń dopóki nie zostanie OTRUTY(!)
        while(true)
        {
            if (read(data_stream[0], &radius, sizeof(int)) != sizeof(int))
                continue;
            if (radius == -1)   // wyłącza się tylko jak otrzymamy pigułkę
                exit(0);
            float result = calculate_circle_area(radius);
            write(response_stream[1], &result, sizeof(float));
        }
    }
}
// wysłanie danych do procesów potomnych
for (i = 0; i < k; i++)
{
    write(data_stream[1], &tab[i], sizeof(int));
}
// Halo odbjoor danych
for (i = 0; i < n; i++)
{
    read(response_stream[0], &part, sizeof(int));
    sum = sum + part;
}
// ZABIJANIE DZIECI
for (i = 0; i < k; i++)
{
    write(data_stream[1], &poison, sizeof(int));
}
// wypisanie odpowiedzi
printf("Suma pól kół: %d", sum);
return 0;
```