ARCHITEKTURA PROCESORÓW SYGNAŁOWYCH

Opracowanie: Grzegorz Szwoch

Politechnika Gdańska, Katedra Systemów Multimedialnych

Procesor sygnałowy

Przypomnijmy naszą definicję:

Cyfrowy procesor sygnałowy (*digital signal procesor*, DSP) jest to układ elektroniczny wyspecjalizowany w optymalnym przetwarzaniu próbek cyfrowego sygnału, wykonując powtarzalne operacje na kolejnych próbkach.

Ten wykład odpowie na pytanie:

Jakie cechy architektury procesora sygnałowego sprawiają, że przetwarza on próbki sygnału „optymalnie”,
lepiej niż zwykły procesor CPU.

Główne cechy architektury DSP

Najważniejsze cechy architektury DSP, odróżniające go od procesorów ogólnego przeznaczenia:

- architektura harwardzka,
- przetwarzanie potokowe,
- adresowanie kołowe,
- specjalne rozkazy procesora (MAC, wektoryzacja)

Elementy procesora sygnałowego

Najważniejsze elementy procesora sygnałowego:

- ALU – jednostka obliczeń arytmetyczno-logicznych, operacje:
  + – AND OR NOT XOR
- jednostka mnożąca (*multiplier*)
- FPU – jednostka do obliczeń zmiennoprzecinkowych
- rejestry (*registers*) – komórki przechowujące dane,
  na których operuje procesor
- akumulator (*accumulator*) – specjalny rejestr
  do przechowywania cząstkowych wyników obliczeń
- generator adresów
- szyny danych (*buses*) – linie wymiany danych między
  pamięcią a rejestrami

Wykonywanie operacji

Przykład obliczeń zadanych przez programistę:

$$y = 0.5 * a + 0.3 * b + 0.2 * c$$

Typowy tok operacji na procesorze sygnałowym:

- odczyt danych z pamięci (a, b, c, stałe), zapisanie ich w rejestrach procesora
- wykonanie kolejnych etapów obliczeń (* * + * +), zapisywanie ich wyników w akumulatorze
- zapisanie końcowego wyniku w pamięci (*y*)

Akumulator

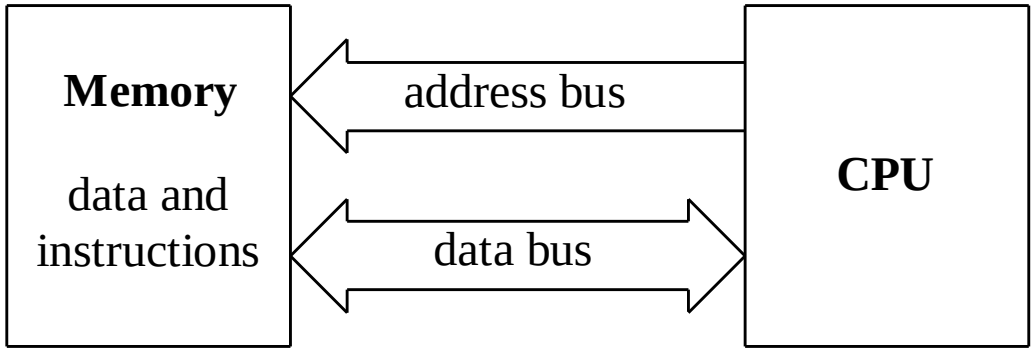
- Akumulator (*accumulator*) jest specjalnym rejestrem procesora, w którym zapisywane są wyniki większości operacji arytmetyczno-logicznych.
- Na procesorze 16-bitowym, wynik mnożenia dwóch liczb 16 b daje wynik 32-bitowy – akumulator musi mieć min. 32 b.
- Sumowanie kolejnych wyników mnożenia może jednak przekroczyć zakres 32 bitów.
- Dlatego akumulator posiada dodatkowe bity (*guard bits*).
- Na procesorach 16-bitowych akumulator ma zazwyczaj długość 40 bitów.
- Programista DSP może zapisywać i odczytywać wartości zmiennych w akumulatorze i w innych rejestrach.

Architektury procesora

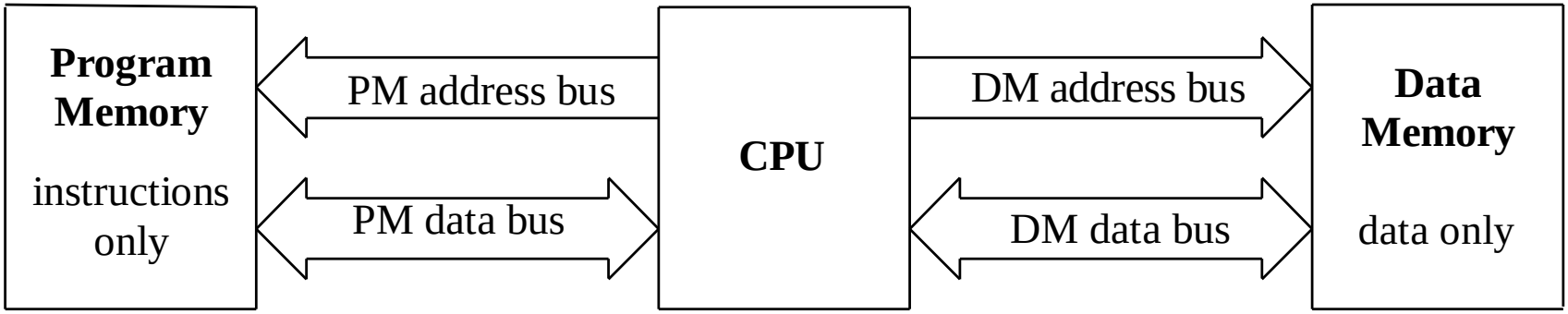
- Architektura von Neumanna
  - wspólna pamięć dla programu (instrukcji) i danych
  - stosowana w klasycznych mikroprocesorach, np. w PC.
- Architektura harwardzka
  - osobne obszary pamięci dla programu i danych
  - możliwość jednoczesnego dostępu do obu pamięci
  - stosowana m.in. w procesorach sygnałowych
- Rozszerzenia architektury harwardzkiej na DSP:
  - podwójne szyny danych (*dual memory access*)
  - pamięć cache dla instrukcji
  - kontroler danych wejściowych/wyjściowych

Architektury procesora

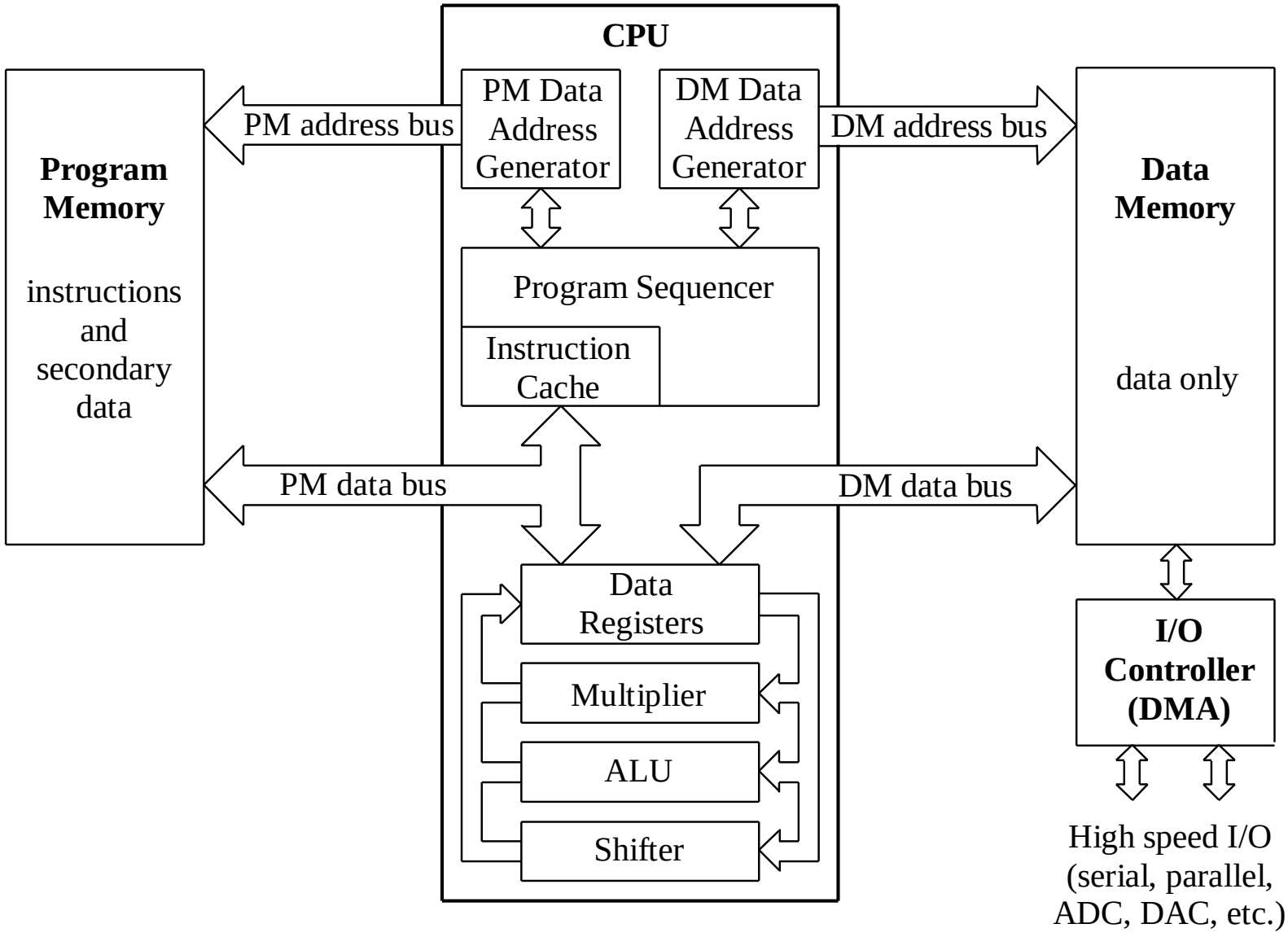
Von Neumann Architecture (*single memory*)



Harvard Architecture (*dual memory*)



Schemat ideowy procesora sygnałowego



Cykle procesora

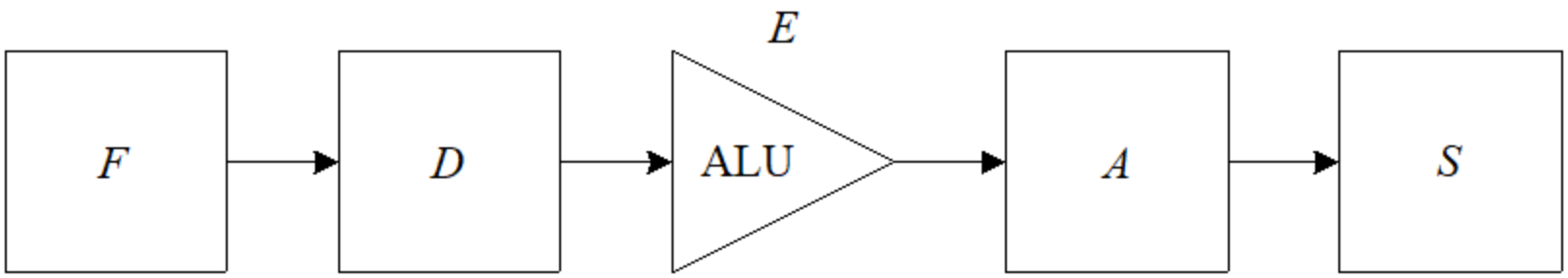
- Procesor jest taktowany zegarem (*clock*), jego częstotliwość jest ustalana przez układ PLL (*phase-locked loop*).
- Każdy impuls zegara wywołuje cykl (*cycle*) procesora.
- Wykonanie każdej instrukcji wymaga określonej liczby cykli.
- Częstotliwość zegara określa liczbę cykli, jaką mamy do dyspozycji aby wykonać program. Np. częstotliwość 100 MHz oznacza, że mamy 100 milionów cykli na sekundę.
- Jeżeli np. przetwarzamy sygnał audio spróbkowany z 48 kHz, na przetworzenie jednej próbki sygnału mamy ok. 2083 cykli.

Wykonywanie instrukcji

Wykonanie każdej instrukcji składa się z kilku etapów:

- F (*fetch*) – pobranie instrukcji z pamięci (lub z *cache*)
- D (*decode*) – zdekodowanie instrukcji
- E (*execute*) – wykonanie instrukcji
- A (*access*) – otwarcie dostępu do pamięci
- S (*store*) – zapisanie wyniku operacji

Czasami wyróżnia się tylko etapy F, D, E.



Przetwarzanie sekwencyjne

W przetwarzaniu sekwencyjnym, kolejna operacja może rozpocząć się dopiero wtedy, gdy cała poprzednia operacja została ukończona.

Przetwarzanie niepotokowe

Clock cycle	1	2	3	4	5	6	7	8	9	10	
Instr. 1	*F*1	*D*1	*E*1	*A*1	*S*1						
Instr. 2						*F*2	*D*2	*E*2	*A*2	*S*2	
⋮											

Przetwarzanie potokowe

Przetwarzanie potokowe (*pipelining*):

- Wykonywany jest pierwszy etap (F) pierwszej instrukcji.
- Gdy procesor przechodzi do drugiego etapu (D), równocześnie rozpoczyna wykonywanie (F) kolejnej instrukcji.
- Instrukcje wykonywane są „na zakładkę”, co daje bardzo duże przyspieszenie wykonywania programu.
- Procesory sygnałowe stosują przetwarzanie potokowe.
- Sytuacje konfliktowe (*hazard*) powodują przerwanie potoku, np. instrukcja skoku do innego miejsca w programie. Trzeba wtedy wycofać częściowo wykonane instrukcje i wznowić potok od nowego miejsca po skoku.

Przetwarzanie potokowe

Przetwarzanie potokowe

Clock cycle	1	2	3	4	5	6	7	8	9	10	
Instr. 1	*F*1	*D*1	*E*1	*A*1	*S*1						
Instr. 2		*F*2	*D*2	*E*2	*A*2	*S*2					
Instr. 3			*F*3	*D*3	*E*3	*A*3	*S*3				
Instr. 4				*F*4	*D*4	*E*4	*A*4	*S*4			
Instr. 5					*F*5	*D*5	*E*5	*A*5	*S*5		
Instr. 6						*F*6	*D*6	*E*6	*A*6	*S*6	
⋮											

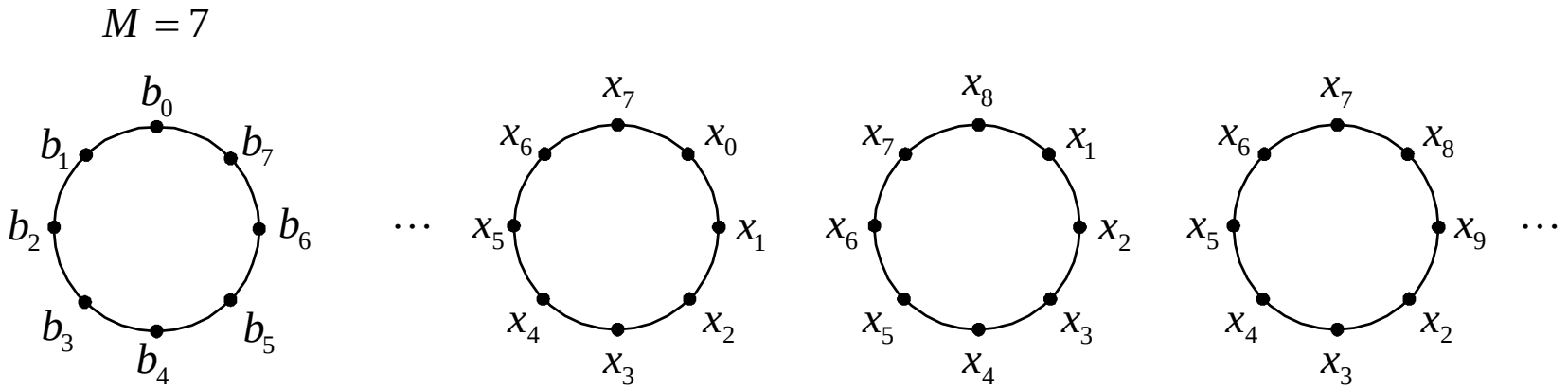
Bufor liniowy

Typowa sytuacja w przetwarzaniu sygnału (np. filtr FIR):

- przetwarzamy *N* ostatnich próbek sygnału
- przechowujemy je w buforze w pamięci
- przychodzi nowa próbka:
  - usuwamy najstarszą,
  - przesuwamy próbki o jedno miejsce,
  - dopisujemy nową na końcu.
- Jest to bufor liniowy.
- Tracimy cykle procesora na przesuwanie *N*-1 próbek w pamięci.

Bufor kołowy

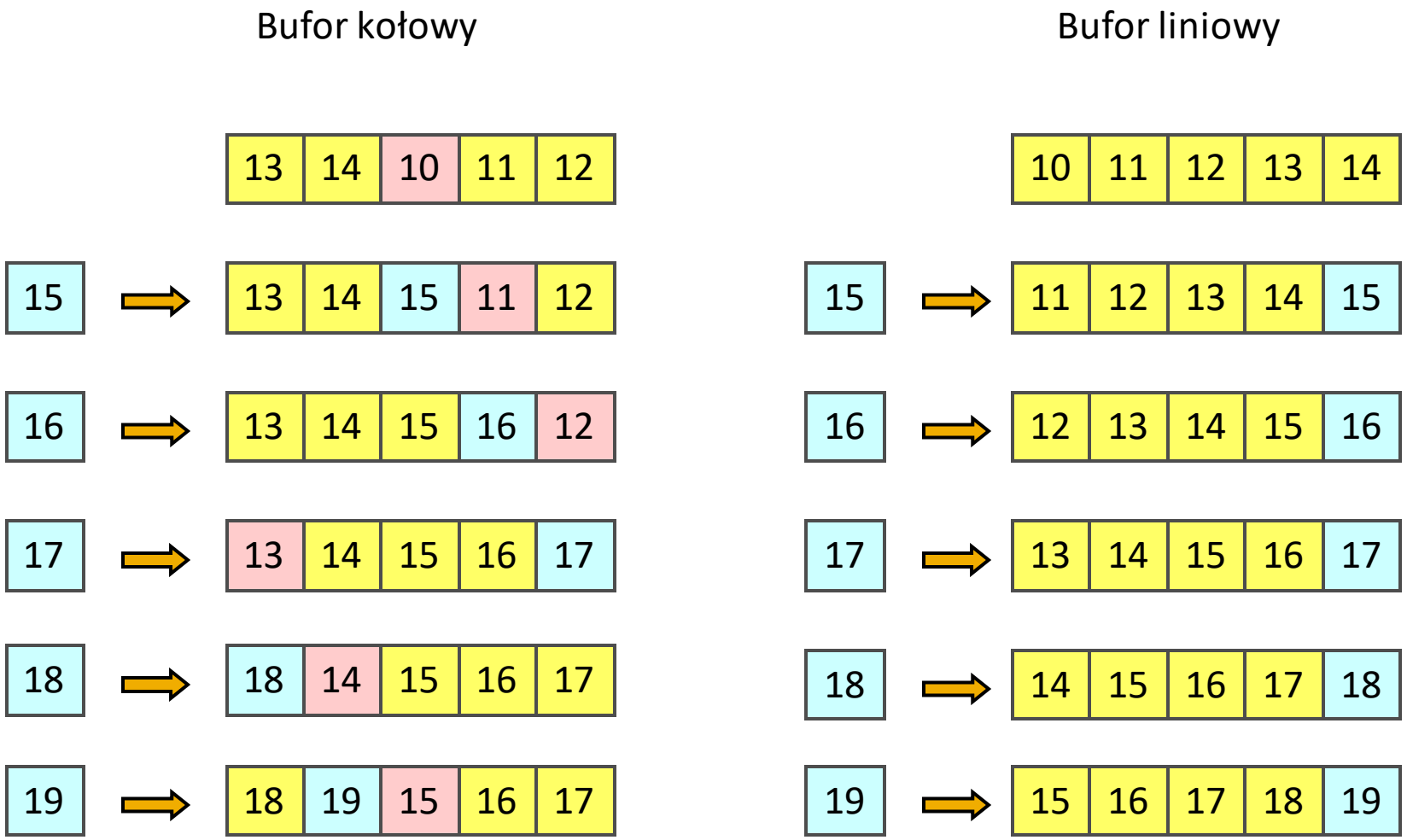
- Bufor kołowy (*circular bufer*) można koncepcyjne przedstawić jako pierścień.
- Wskaźnik (*pointer*) wyznacza miejsce, w którym znajduje się najstarsza próbka sygnału.
- Nowa próbka zastępuje najstarszą, wskaźnik jest przesuwany.
- Nie ma potrzeby kopiowania danych.



Adresowanie kołowe

- W praktyce, bufor kołowy jest nadal liniowym obszarem w pamięci.
- Wskaźnik jest przesuwany, wskazuje kolejność przetwarzania próbek.
- Wskaźnik musi być zawijany po dojściu na koniec bufora.
- Bufor jest fizycznie liniowy, ale używane jest adresowanie kołowe (*circular addressing*).
- Procesory sygnałowe, w odróżnieniu od zwykłych mają sprzętowo zaimplementowane adresowanie kołowe.

Bufor kołowy i liniowy - ilustracja



Adresowanie kołowe

Na zwykłym procesorze musimy ręcznie „zawijać” indeks.

```
bufor[indeks] = nowa_probka;          // zapis
// … tutaj wykonujemy obliczenia
indeks = indeks + 1;                  // przesunięcie indeksu
if (indeks == N)                      // jesteśmy na końcu
    indeks = 0;                       // zawijamy indeks
```

Na DSP wykorzystujemy adresowanie kołowe:

- w asemblerze – rozkazem procesora,
- w C – specjalną funkcją wewnętrzną (*intrinsic*):

```
bufor[indeks] = nowa_probka;          // zapis
// … tutaj wykonujemy obliczenia
indeks = _circ_incr(indeks, 1, N)     // przesunięcie indeksu
                                      // z zawinięciem
```

Instrukcja MAC

- Typowa operacja w przetwarzaniu sygnałów: przemnożenie liczb, dodanie wyniku do sumy

  $$y \leftarrow y + a * x$$

- MAC – *multiply and accumulate*, przemnóż i dodaj.
- Na zwykłym procesorze wymaga to wykonania osobno mnożenia, a potem dodawania.
- Procesory sygnałowe mają zaimplementowane MAC sprzętowo, jako pojedynczą instrukcję procesora.
- Przyspiesza to wykonywanie obliczeń – mniej cykli.
- Większość współczesnych DSP potrafi wykonywać dwie operacje MAC jednocześnie (*dual MAC*).

MAC w praktyce

Klasyczny procesor CPU:

```
for (i = 0; i < N; i++) {
    wynik += bufor[indeks] * wsp[indeks];
    indeks = indeks + 1;
    if (indeks == N) indeks = 0;
}
```

Na DSP z instrukcją MAC:

```
for (i = 0; i < N; i++) {
    wynik = _smac(wynik, bufor[indeks], wsp[indeks]);
    poz = _circ_incr(indeks, 1, N);
}
```

SIMD (wektoryzacja)

- Inny typowy przykład: mnożenie dwóch wektorów o dł. *N*.
- Wymaga to wykonania *N* operacji mnożenia.
- Liczby zmiennoprzecinkowe mogą być zapisane z pojedynczą (4 b) lub podwójną (8 b) precyzją.
- Procesor potrafi mnożyć dwie liczby 4 b lub dwie liczby 8 b.
- DSP potrafi zwykle „upakować” dwie liczby 4 b do jednej „liczby” 8 b i wykonać mnożenie liczb 8 b.
- Zmniejsza to liczbę operacji mnożenia z *N* do *N*/2.
- Jest to wektoryzacja, inaczej SIMD (*single instruction, multiple data*) – ta sama operacja na różnych danych.
- Procesory CPU również mają rozszerzenia pozwalające na wektoryzację (SSE).

Wektoryzacja - przykład

Bez wektoryzacji:

```
for (i = 0; i < N; i++) {
    y[i] = a[i] * b[i];
}
```

Z wektoryzacją – więcej kodu, mniej operacji:

```
for (i = 0; i < N; i+=2) {
    _amem8_f2(&y[i]) =
        _dmpysp(_amem8_f2(&a[i]), _amem8_f2(&b[i]));
}
```

Organizacja pamięci

Pamięć w DSP jest logicznie i fizycznie podzielona na kilka poziomów. Każdy kolejny poziom ma „wolniejszy” dostęp.

- L1 – pamięć podręczna (*cache*)
  - do wewnętrznego użytku procesora.
- L2 – pamięć wewnętrzna RAM w procesorze
  - do użytku programisty (program i dane)
  - zwykle mała pojemność (rzędu 1 MB).
- L3 – pamięć zewnętrzna
  - zwykle osobna „kość” pamięci typu DDR
  - znacznie wolniejszy dostęp, duże pojemności (GB)

Pamięć ROM, często typu *flash* – program wykonywalny, stałe.

Pamięć SARAM i DARAM

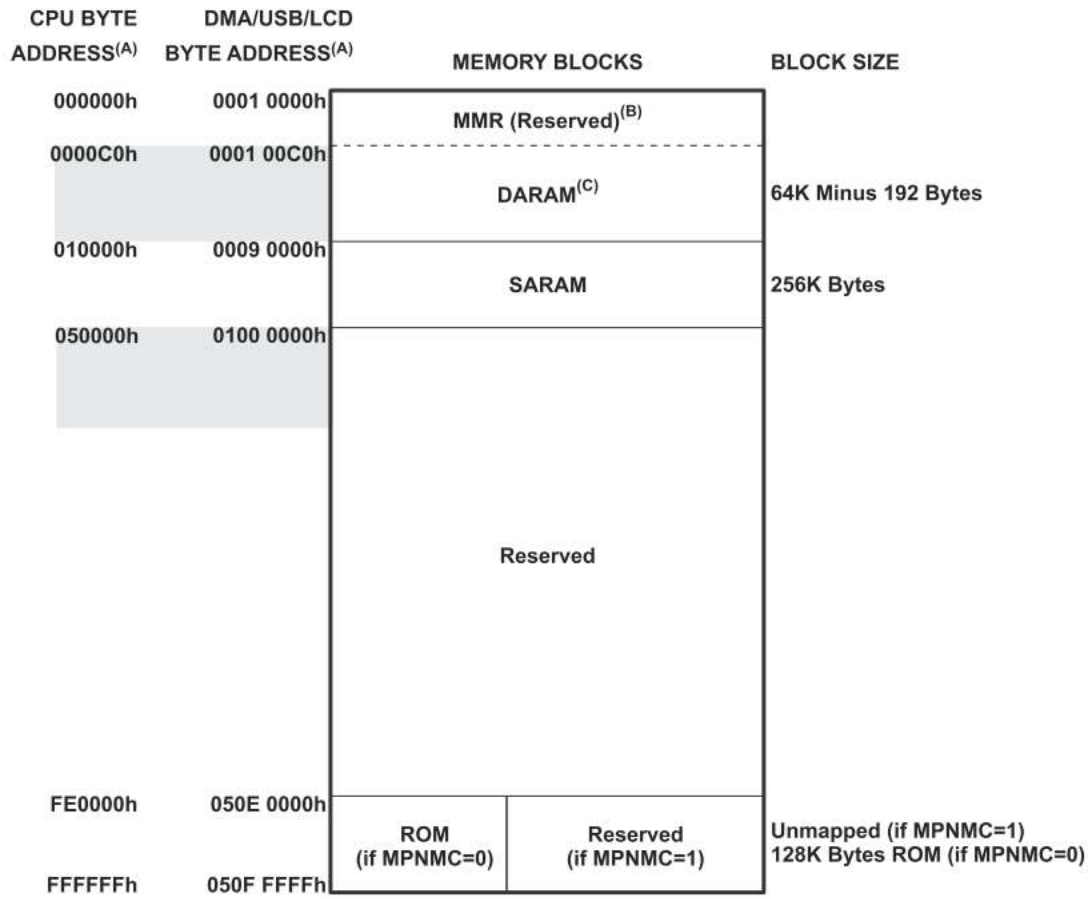
- SARAM (*single access random access memory*)
  – typowa pamięć, można jednocześnie wykonać jedną operację odczytu lub zapisu danych.
- DARAM (*dual access RAM*) – podwojona szyna danych, jednocześnie można wykonać dwie operacje (dwa zapisy, dwa odczyty lub zapis+odczyt).
- Na DSP: albo cała pamięć typu DARAM, albo podział: część pamięci (zwykle mniejsza) jako DARAM, reszta SARAM.
- Podział pamięci na banki (*banks*) – możliwy jednoczesny dostęp do dwóch banków.
- Programista musi przemyśleć które dane odniosą korzyść z DARAM, a które mogą być w SARAM.

Mapa pamięci

- Mapa pamięci (*memory map*) jest specyficzna dla danego modelu procesora, określona przez producenta.
- Każdemu typowi pamięci przypisywany jest zakres adresów.
- Adres (*address*) jest liczbą określającą miejsce w pamięci, w którym znajduje się dana zmienna lub stała.
- Mapa pamięci przypisuje logiczne zakresy adresów do fizycznych obszarów pamięci.
- Jest niezbędna w każdym programie na DSP – kompilator musi ją znać.

Mapa pamięci

Przykład mapy pamięci z dokumentacji C5535:



Mapa pamięci

Przykład definicji mapy pamięci dla kompilatora (C5535):

```
MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */

  MMR    (RWIX): origin = 0x000000, length = 0x0000c0  /* MMRs */
  DARAM0 (RWIX): origin = 0x0000c0, length = 0x00ff40  /*  64KB - MMRs */
  SARAM0 (RWIX): origin = 0x010000, length = 0x010000  /*  64KB */
  SARAM1 (RWIX): origin = 0x020000, length = 0x020000  /* 128KB */
  SARAM2 (RWIX): origin = 0x040000, length = 0x00FE00  /*  64KB */
  VECS   (RWIX): origin = 0x04FE00, length = 0x000200  /*  512B */
  PDROM   (RIX): origin = 0xff8000, length = 0x008000  /*  32KB */

 PAGE 2:  /* -------- 64K-word I/O Address Space -------- */

  IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
```

Sekcje pamięci

Logiczne sekcje pamięci są przypisywane do adresów.

Najważniejsze sekcje:

- .text – kod programu
- .stack – obszar stosu (zmienne deklarowane lokalnie)
- .data – zainicjalizowane dane
- .bss – zmienne globalne i statyczne
- .const – stałe
- .sysmem – sterta (zmienne tworzone dynamicznie)

Programista może tworzyć własne sekcje.

Sekcje pamięci

Przykład definicji sekcji pamięci dla kompilatora (C5535):

```
SECTIONS
{
   .text     >> SARAM1|SARAM2|SARAM0  /* Code                          */
   .stack    >  DARAM0                /* Primary system stack          */
   .sysstack >  DARAM0                /* Secondary system stack        */
   .data     >> DARAM0|SARAM0|SARAM1  /* Initialized vars              */
   .bss      >> DARAM0|SARAM0|SARAM1  /* Global & static vars          */
   .const    >> DARAM0|SARAM0|SARAM1  /* Constant data                 */
   .sysmem   >  DARAM0|SARAM0|SARAM1  /* Dynamic memory (malloc)       */
   .switch   >  SARAM2                /* Switch statement tables       */
   .cinit    >  SARAM2                /* Auto-initialization tables    */
   .pinit    >  SARAM2                /* Initialization fn tables      */
   .cio      >  SARAM2                /* C I/O buffers                 */
   .args     >  SARAM2                /* Arguments to main()           */
    vectors  >  VECS                  /* Interrupt vectors             */
   .ioport   >  IOPORT PAGE 2         /* Global & static ioport vars */
   .fftcode  >  SARAM0                /* Sekcje utworzone przez programistę */
   .input    >  DARAM0, align(4)
}
```

Korzystanie z sekcji w kodzie C

W ten sposób bufor zostanie utworzony w pamięci DARAM lub SARAM, w domyślnej sekcji .bss:

```
int bufor[8192];
```

Jeżeli mamy pamięć zewnętrzną zadeklarowaną w mapie sekcji:

```
.ddr > DDR3
```

to w kodzie C możemy utworzyć bufor w pamięci DDR stosując dyrektywę kompilatora (przykład dla procesora TI):

```
#pragma DATA_SECTION(bufor, "ddr");
int bufor[8192];
```

Pamięć wewnętrzna i zewnętrzna

Jakie dane w pamięci wewnętrznej L2 (DARAM/SARAM)?

- kod programu, stos, sterta
- większość typowych zmiennych
- bufory danych, które są często wykorzystywane

Jakie dane w pamięci zewnętrznej L3 (DDR)?

- bardzo duże bufory, nie mieszczące się w pamięci L2
- rzadko wykorzystywane dane
- zarchiwizowane wyniki przetwarzania

Uwagi o sekcjach pamięci

Dotyczy programów tworzonych w C.

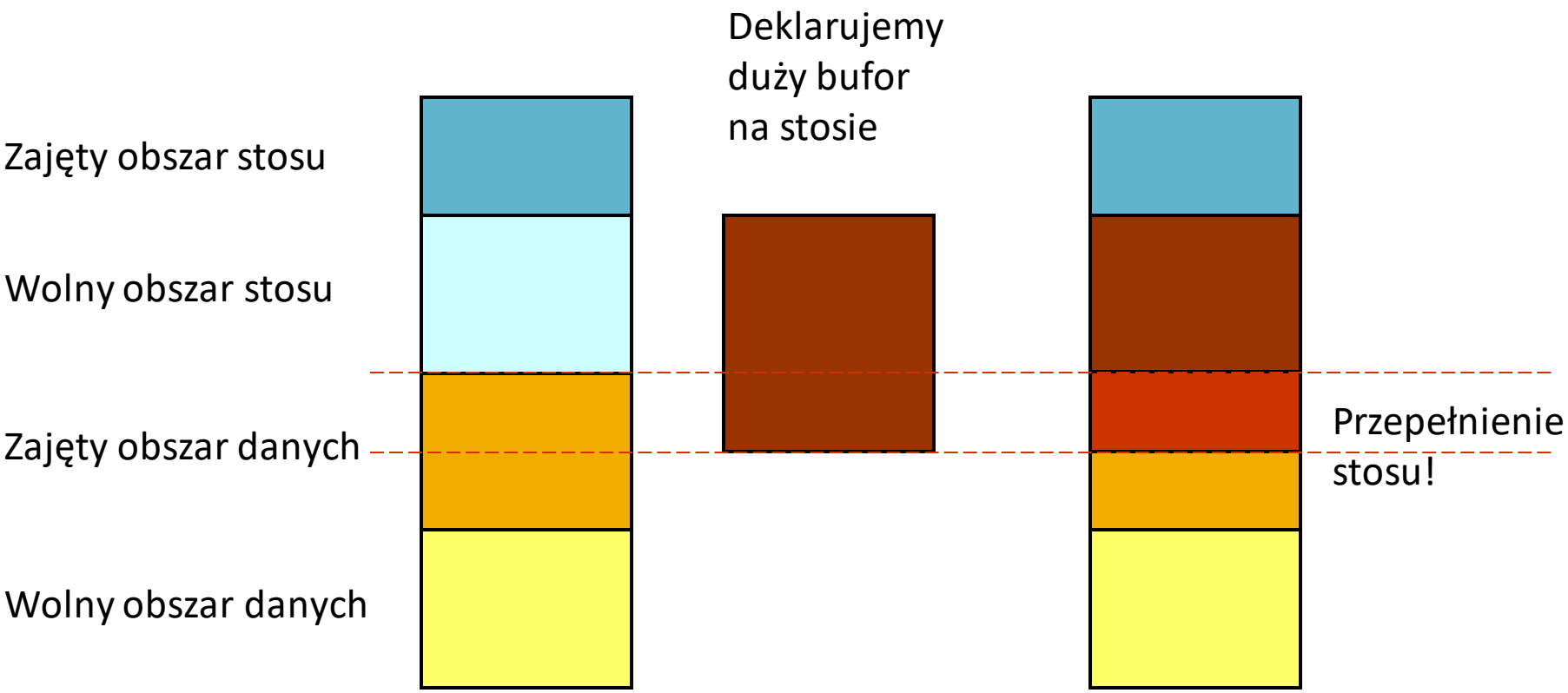
- Zmienne deklarowane globalnie (w głównej części kodu, poza funkcjami) oraz zmienne statyczne – do sekcji *.bss*.
- Zmienne deklarowane wewnątrz funkcji (w tym funkcji *main*) – do sekcji *.stack* (stos).
- Zmienne tworzone dynamicznie (przez *malloc*) – do sekcji *.sysmem* (sterta).
- Stałe (np. tablice współczynników filtru) – do sekcji *.const*.

Praktyczne wnioski:

- nie deklarować dużych buforów wewnątrz funkcji – obszar stosu jest mały, można przepełnić stos
- stałe dane, jak współczynniki filtru, deklarować jako *const*

Przepełnienie stosu

Przekroczenie dostępnego miejsca na stosie
– przepełnienie stosu (*stack overflow*)



Przepełnienie stosu

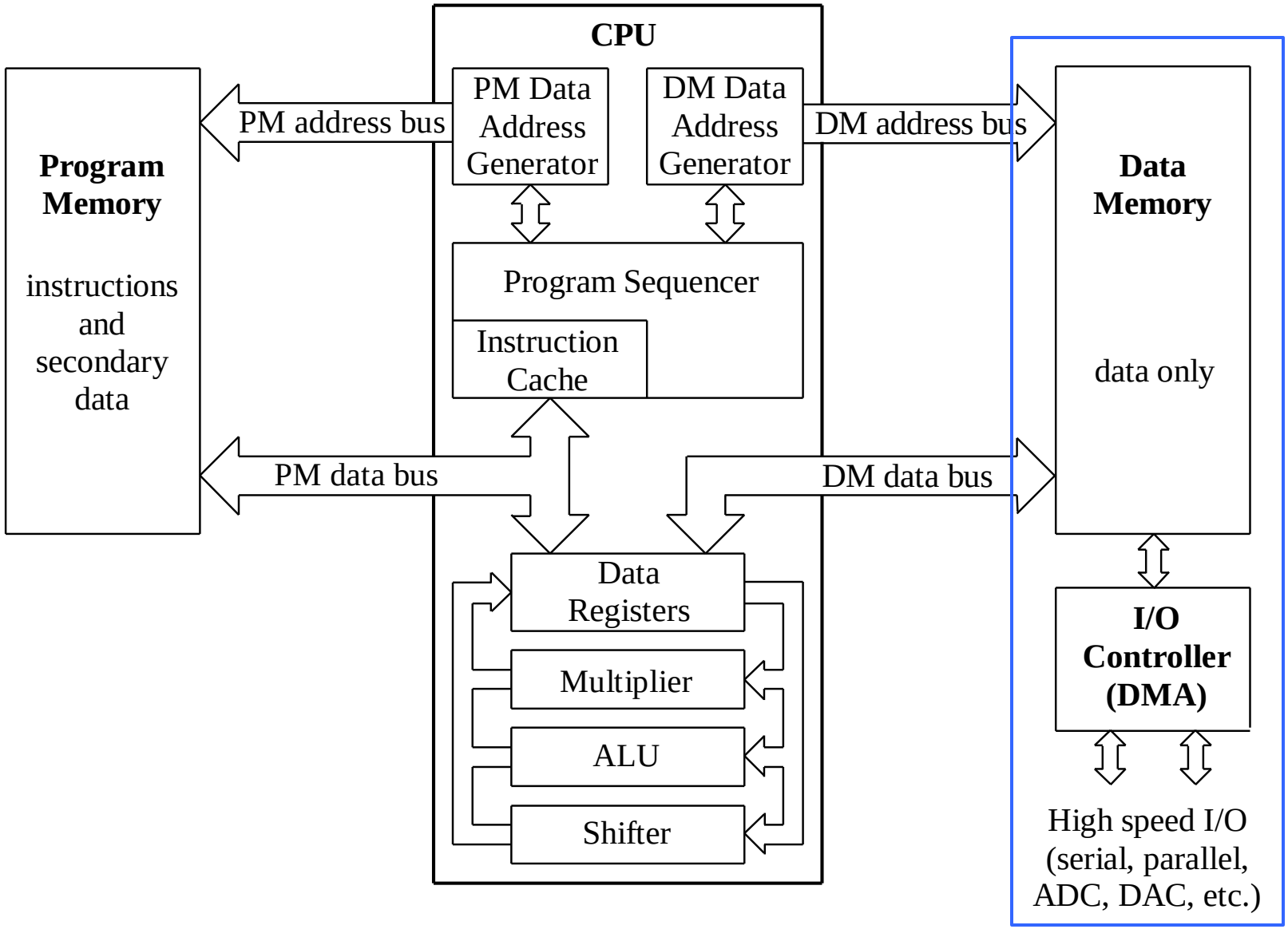
Co się stanie gdy przepełnimy stos?

- Na zwykłym systemie (np. Windows) program wykona niedozwoloną operację i zostanie zamknięty.
- Na DSP nie ma zabezpieczeń! Skutek zależy od tego co znajdowało się w nadpisanym obszarze pamięci:
  - obszar był pusty – program działa dalej,
  - obszar zawierał dane – program może się zawiesić lub działać dalej, ale generować błędne dane!
- Efekty przepełnienia stosu są bardzo trudne do debugowania.
- Dlatego najlepiej jest przyjąć zasadę: wszelkie bufory (tablice) deklarujemy globalnie, poza funkcjami!

Bezpośredni dostęp do pamięci

- Napływające do procesora dane (próbki sygnału) muszą być zapisywane do pamięci.
- DMA (*direct memory access*) – bezpośredni dostęp interfejsów do pamięci.
- Dane wejściowe są przenoszone do/z pamięci bez konieczności wykonywania instrukcji przez procesor – nie zużywają cykli procesora.
- Znaczne przyspieszenie pracy układu.
- Praktycznie wszystkie procesory sygnałowe obsługują DMA.

Bezpośredni dostęp do pamięci



Przerwania

Skąd program ma wiedzieć że w pamięci są nowe dane?

- Odpytywanie (*polling*) – program cyklicznie sprawdza czy są nowe dane
  - mało wydajne, zużywa cykle na sprawdzanie danych
  - wprowadza opóźnienia
- Przerwania (*interrupts*) – lepszy sposób:
  - po zapisaniu nowych danych, kontroler wysyła przerwanie – jest to sygnał informacyjny
  - programista pisze procedurę obsługi przerwania
  - przerwanie ma wyższy priorytet – „przerywa” działanie głównego programu
  - mniejsze opóźnienia, nie są tracone cykle procesora

Uwagi o kompilatorze C (1)

- Każdy typ zmiennej zajmuje określoną liczbę bajtów, np. typ *float* zajmuje typowo 4 bajty.
- Adres zmiennej – liczba całkowita wskazująca na położenie zmiennej w pamięci.
- Wyrównanie (*alignment*) – wymóg, aby adres był podzielny bez reszty przez rozmiar typu (*float*: przez 4).
- W niektórych przypadkach (operacje typu *dual*) wyrównanie wymaga, aby reszta z dzielenia przez 2×rozmiar była zerowa.
- Wyrównanie jest bardzo często warunkiem, aby kompilator wygenerował zoptymalizowany kod.

Uwagi o kompilatorze (1)

Wyrównanie trzeba wymusić na kompilatorze, stosując tzw. dyrektywy kompilatora (*pragma*).

Przykład dla procesora TI – wyrównanie do 8 bajtów:

```
#pragma DATA_ALIGN(bufor, 8);
int bufor[8192];
```

Uwagi o kompilatorze C (2)

Chcemy, aby kompilator wygenerował kod pętli wykorzystując *dual MAC* – dwie operacje w jednej instrukcji.

Kompilator domyślnie nie zrobi tego, ponieważ nie wie:

- czy na pewno pętla zostanie wykonana parzystą liczbę razy,
- czy nie nastąpi wyjście z pętli,
- czy bufory, na których działa pętla, nie pokrywają się w pamięci.

Skutek: kompilator zagra „bezpiecznie” (wg prawa Murphy’ego) i wygeneruje nieoptymalny kod. Nie ma takich problemów jeżeli piszemy kod w asemblerze – mamy kontrolę nad programem.

Uwagi o kompilatorze C (2)

Ponownie musimy podać kompilatorowi informacje:

- ile razy wykona się pętla (*MUST_ITERATE*),
- jak rozwinąć pętlę (*UNROLL*),
- zapewnić brak nakładania się tablic (*restrict*)

Przykład (procesor TI):

```
void vecmul(int* restrict y, int* restrict a,
            int* restrict b, int n)
{
    int i;
    #pragma MUST_ITERATE(2,,2)
    #pragma UNROLL(2)
    for (i = 0; i < n; i++)
       y[i] = a[i] * b[i];
}
```

Uwagi o kompilatorze C (wnioski)

- Pisząc w asemblerze, możemy napisać optymalny kod, ale to na nas spoczywa obowiązek zapewnienia, że kod działa prawidłowo.
- Kompilator C ma zapewnić, że kod ZAWSZE będzie działał bez błędów. Jeżeli „widzi” ryzyko, wstrzymuje optymalizacje.
- Programista musi stosować „magiczne pragmy” aby zapewnić kompilator o swoich intencjach.
- Niestety, często kompilator i tak uważa, że on wie lepiej ☺. Nie generuje takiego kodu, jaki chcemy.
- W takich sytuacjach zostaje napisać kod samemu, w asemblerze (o ile to kompilator nie ma racji).