Zastosowania procesorów sygnałowych

ARCHITEKTURA PROCESORÓW SYGNAŁOWYCH

Opracowanie: Grzegorz Szwoch

Politechnika Gdańska, Katedra Systemów Multimedialnych

Elementy procesora sygnałowego

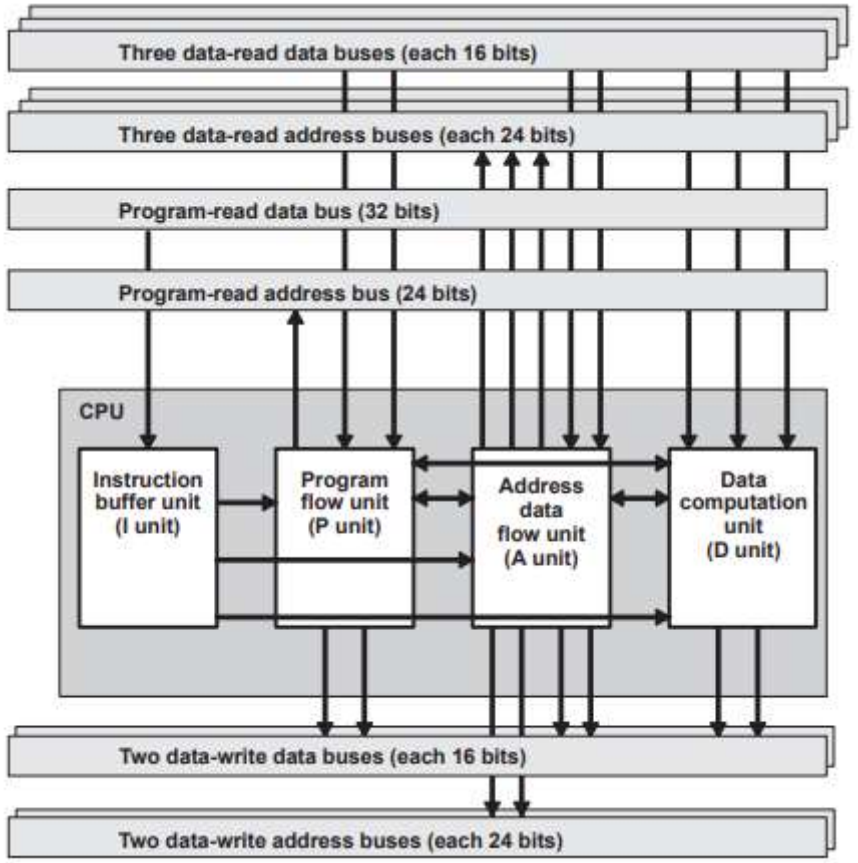
Najważniejsze elementy procesora sygnałowego:

- 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
- akumulatory (*accumulator*) – specjalne rejestry do przechowywania cząstkowych wyników obliczeń
- generator adresów – do odczytu danych z pamięci
- jednostka sterująca wykonywaniem instrukcji (*program flow unit*)
- szyny (*buses*) – linie wymiany danych i instrukcji między pamięcią a rejestrami

Przykład architektury procesora sygnałowego

Na przykładzie procesora sygnałowego C5535

- *I Unit* – jednostka bufora instrukcji, pobiera instrukcje z pamięci do PS.
- *P-Unit* – jednostka sterowania programem, wyznacza kolejność wykonywania instrukcji, obsługuje instrukcje skoku, pętle, itp.
- *A-Unit* – jednostka przepływu danych, generuje adresy pamięci do odczytu i zapisu danych.
- *D-Unit* – jednostka przetwarzania danych, wykonuje instrukcje na pobranych danych.
- Szyny (*bus*) – wymiana instrukcji i danych między pamięcią a procesorem.

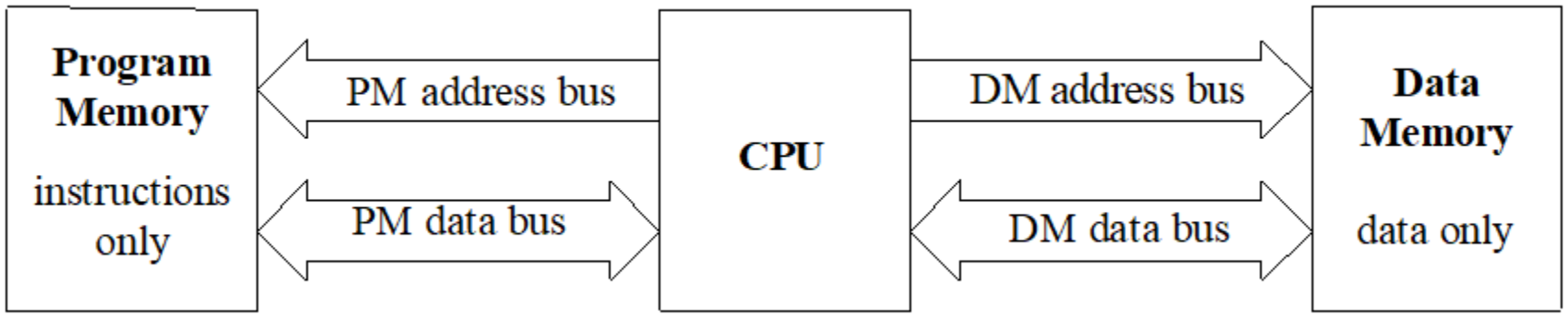


Akumulator

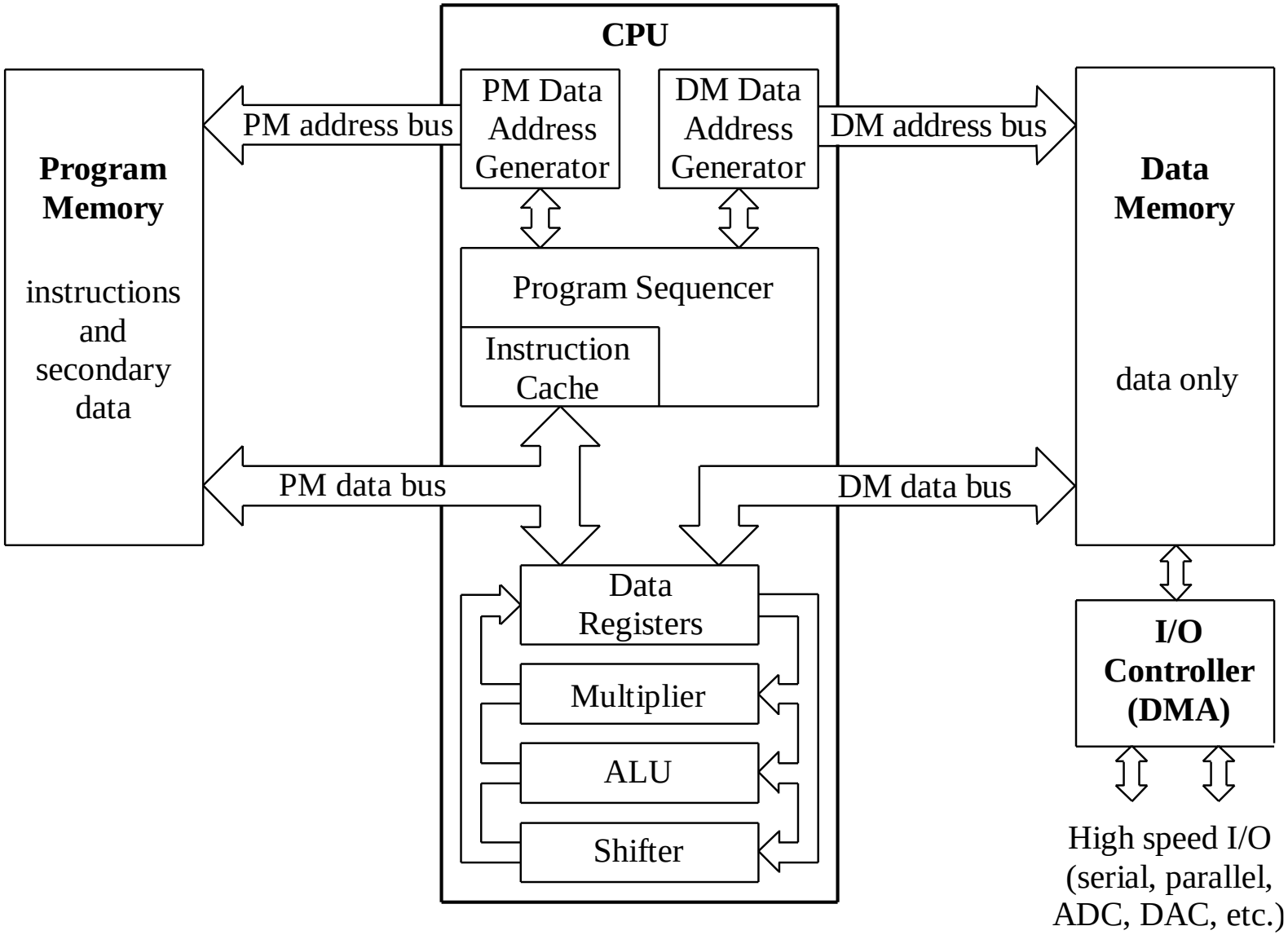
- Akumulator (*accumulator, ACU*) jest specjalnym rejestrem procesora, w którym zapisywane są wyniki większości operacji arytmetyczno-logicznych.
- Mnożenie dwóch liczb 16-bitowych daje wynik 32-bitowy
  – akumulator musi mieć minimum 32 bity.
- Sumowanie kolejnych wyników mnożenia może jednak przekroczyć zakres 32 bitów.
- Dlatego akumulator posiada dodatkowe bity (*guard bits*).
- Procesor C5535 ma 4 akumulatory o długości 40 bitów (5 bajtów).
- Programista może zapisywać i odczytywać wartości zmiennych w akumulatorze i w innych rejestrach pisząc kod w Asemblerze.
- Procesor C5535: zmienne typu *long long* są zapisywane w akumulatorze.

Architektura harwardzka

- Procesory sygnałowe są zazwyczaj oparte na architekturze harwardzkiej, w której są osobne linie (szyny) do przesyłania instrukcji programu oraz do przesyłania danych z pamięci.
- Dzięki temu transmisja instrukcji i danych może odbywać się równocześnie.
- Tradycyjne procesory mają zwykle architekturę von Neumanna – wspólna pamięć programu i danych, wspólna szyna do przesyłania.
- W procesorach sygnałowych stosuje się też podwójne szyny danych (*dual memory access*), co umożliwia np. równoczesny zapis jednej liczby i odczyt drugiej.



Ogólny schemat procesora sygnałowego



Cykle procesora

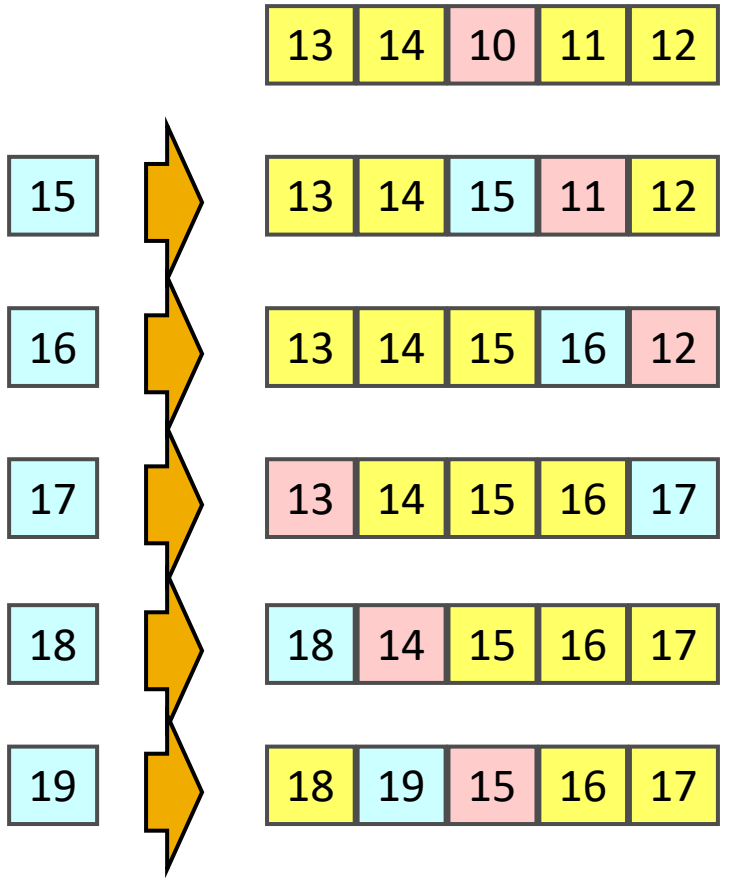
- Procesor jest taktowany zegarem (*clock*), jego częstotliwość jest ustalana przez układ PLL (*phase-locked loop*).
- Każdy impuls zegara wywołuje cykl (*cycle*) procesora.
- Wykonanie większości instrukcji wymaga jednego cyklu. Bardziej złożone instrukcje mogą zużywać 2 lub więcej cykli.
- Częstotliwość zegara określa liczbę cykli, jaką mamy do dyspozycji w ciągu sekundy, aby wykonać instrukcje (tzw. budżet procesora). Np. częstotliwość 100 MHz oznacza, że mamy 100 milionów cykli na sekundę.
- Jeżeli np. przetwarzamy sygnał audio z częstotliwością próbkowania 48 kHz, na przetworzenie jednej próbki sygnału mamy ok. 2083 cykli.

Potokowe wykonywanie instrukcji

- Wykonanie instrukcji przez procesor składa się z trzech faz:
  - *F* – *fetch*, pobranie instrukcji z pamięci,
  - *D* – *decode*, przygotowanie instrukcji do wykonania,
  - *E* – *execute*, wykonanie żądanej operacji,
- Przy sekwencyjnym wykonywaniu instrukcji, każda instrukcja wymaga min. 3 cykli.
- Procesor sygnałowy wykonuje instrukcji w sposób potokowy (*pipelining*), „na zakładkę”. Gdy wykonywana jest faza *E* instrukcji, jednocześnie wykonywana jest faza *D* kolejnej instrukcji i faza *F* jeszcze kolejnej.
- Instrukcje warunkowe (*if*) powodują rozgałęzienie kodu (*branching*) i mogą spowodować przerwanie potoku (trzeba go zacząć od nowa).

Bufor kołowy i adresowanie kołowe

- Często musimy mieć pewną liczbę ostatnich wartości. Gdy przychodzi nowa dana, musimy usunąć najstarszą i zastąpić ją nową.
- Bufor liniowy jest niewydajny, wymaga przesuwania danych za każdym razem.
- Bufor kołowy (*circular buffer*) używa wskaźnika (indeksu) dla komórki zapisu i odczytu.
- Wskaźnik przechodzi w sposób kołowy, przeskakuje między końcem a początkiem bufora.
- Procesor sygnałowy obsługuje adresowanie kołowe - wskaźnik sam się „zawija”.
- W języku C na procesorze C5535: instrukcja *_circ_incr*.



Instrukcja MAC

- Typowa operacja w przetwarzaniu sygnałów (np. filtry FIR, splot): przemnożenie liczb, dodanie wyniku do sumy (akumulacja):

  y := y + a * x

- Na zwykłym procesorze wymaga to wykonania osobno mnożenia (MPY), a potem dodawania (ADD), a więc minimum 2 cykle procesora.
- Procesory sygnałowe mają specjalną instrukcję MAC – *multiply and accumulate*, wykonywaną w jednym cyklu procesora.
- Przetwarzanie filtrem FIR o długości 100 przy częstotliwości próbkowania 48 kHz: oszczędność 4,8 mln cykli na sekundę.
- Niektóre PS potrafią wykonywać dwie operacje MAC jednocześnie (*dual MAC*).
- Zwykłe procesory (CPU) obsługują MAC jako rozszerzenie (np. SSE).
- Procesor C5535: instrukcja *_smac*.

Wektoryzacja

- Zmiennoprzecinkowy procesor sygnałowy potrafi wykonywać operacje na 4-bitowych liczbach *float* i 8-bitowych liczbach *double*.
- Istnieje możliwość „upakowania” dwóch liczb *float* do jednej liczby *double* i wykonania jednej operacji (np. mnożenia) zamiast dwóch.
- Taki sposób obliczeń nazywa się wektoryzacją. Wykorzystywany jest model SIMD – *single instruction, multiple data* (jedna instrukcja, wiele danych).
- Zmniejsza to liczbę cykli potrzebnych do wykonania np. mnożenia dwóch tablic przez siebie (kosztem bardziej skomplikowanego kodu).
- Procesory Intel i ARM posiadają rozszerzenia instrukcji procesora wykonujące wektoryzację (współczesne procesory: instrukcje 256-bitowe).

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

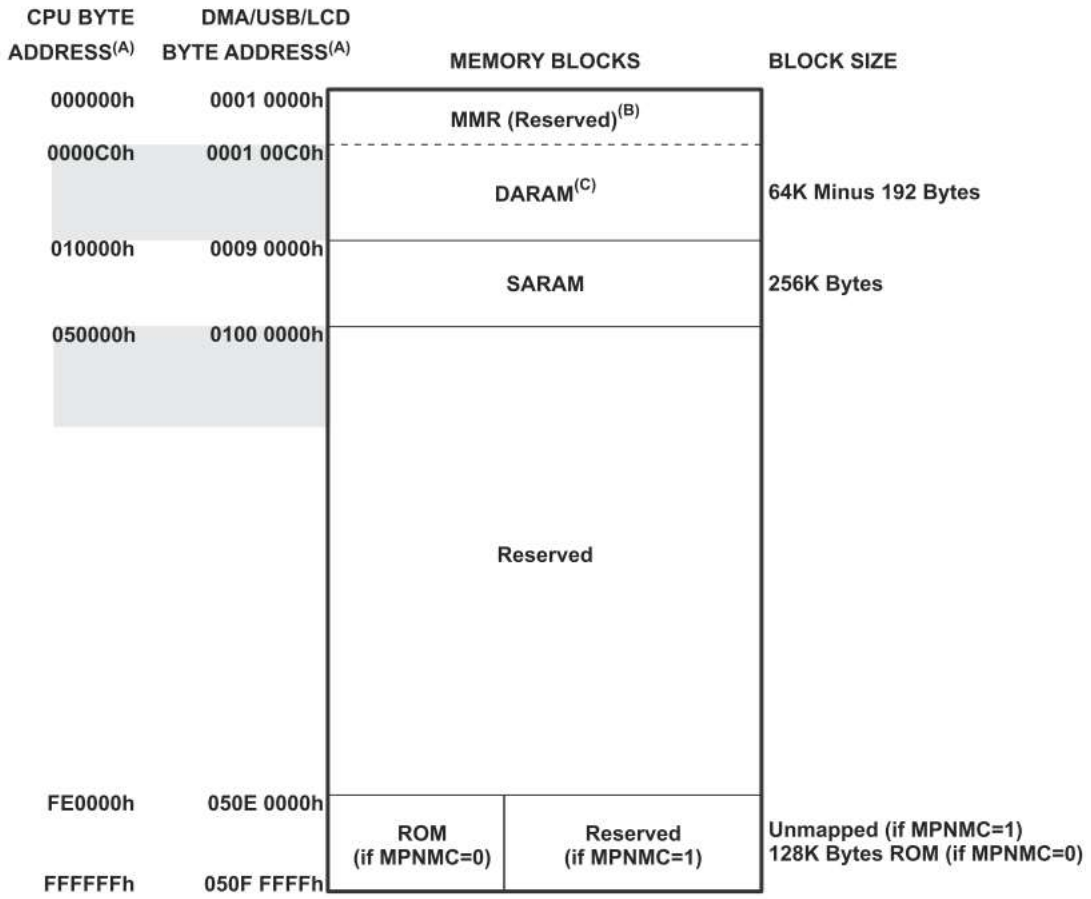
Organizacja pamięci

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 zewnętrzna typu *flash* – program wykonywalny, archiwizacja danych.

Mapa pamięci

- Dostęp do pamięci: poprzez adres.
- Mapa pamięci definiuje zakresy adresów dla różnych obszarów pamięci.
- Procesor C55355 ma dwa rodzaje pamięci:
  - DARAM – pamięć podwójnego dostępu (dwie szyny danych, szybsza), 64 KB,
  - SARAM – pamięć pojedynczego dostępu (jedna szyna danych, wolniejsza), 256 KB.
- Programista może decydować o tym jakie fragmenty kodu są umieszczane w danym obszarze. Np. dane: DARAM,
  kod programu: SARAM.



Mapa pamięci

Przykład definicji mapy pamięci dla kompilatora na procesor C5535.

Pamięć SARAM jest logicznie podzielona na 3 bloki (SARAM0, SARAM1, SARAM2).

```
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

Konsolidator (linker) budując program dzieli go na sekcje:

- *.text* – kod programu,
- *.bss*, *.data*, *.const* – zmienne globalne i stałe,
- *.stack* – obszar stosu (zmienne deklarowane lokalnie),
- *.sysmem* – sterta (zmienne tworzone dynamicznie przez *malloc*).

Programista decyduje o tym do którego obszaru pamięci trafi dana sekcja.
Może również tworzyć własne sekcje.

```
// .text - kod programu
short bufor[2048];            // .bss   - zmienna globalna, nie zainicjalizowana
short indeks = 1;             // .data  - zmienna globalna, zainicjalizowana
const long CZ_PROB = 48000L;  // .const - stała

void main(void) {
    short wejscie;            // .stack - zmienna lokalna, na stosie
    static short licznik = 0; // .bss   - zmienna statyczna
    // ...
}
```

Definicje pamięci

Plik konfiguracyjny kompilatora określa do którego obszaru pamięci trafia dana sekcja.

```
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 DDR zadeklarowaną w mapie sekcji:

```
ddr > DDR3
```

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

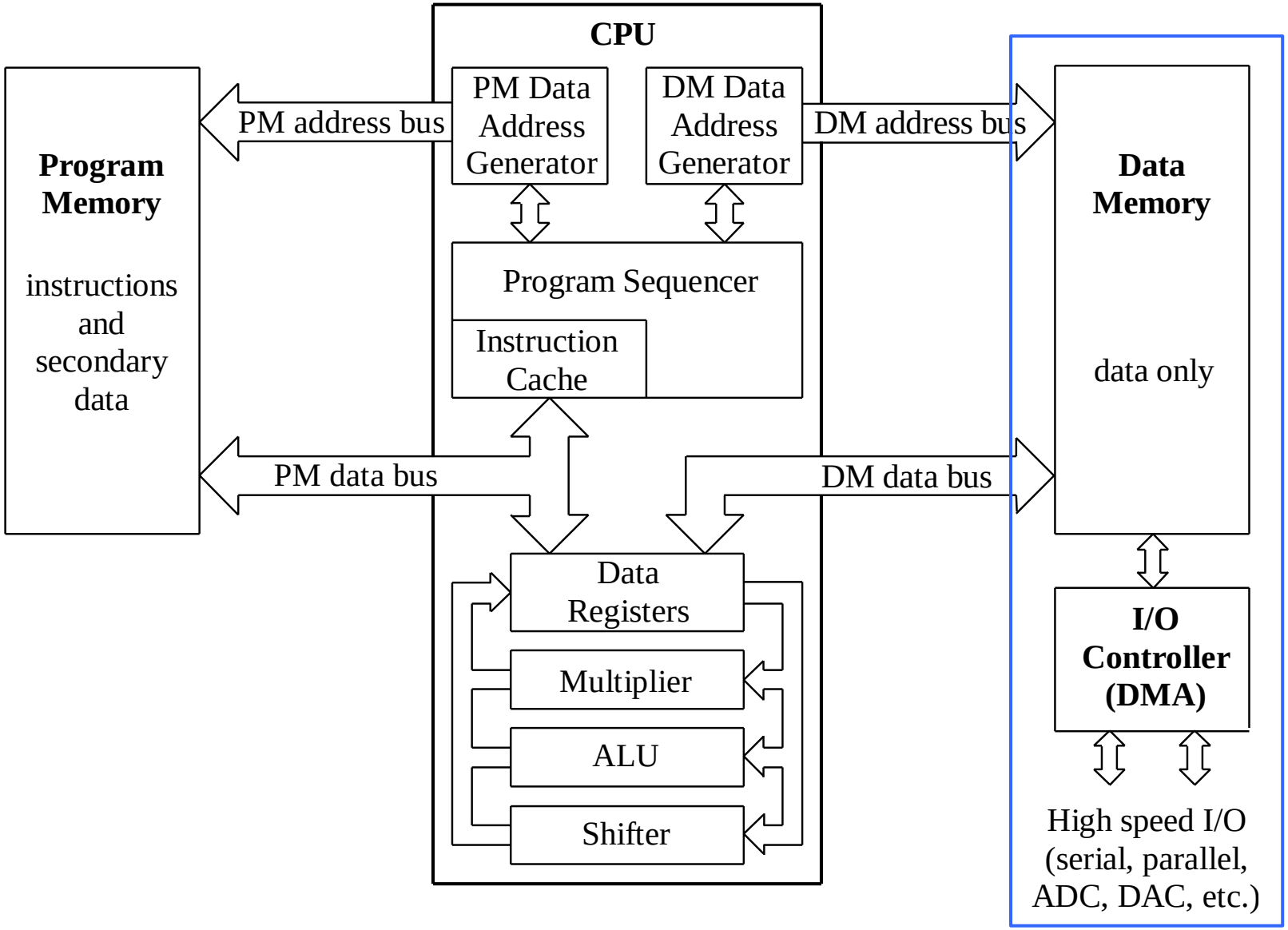
```
#pragma DATA_SECTION(bufor, "ddr");
short bufor[2048];
```

```
short bufor[2048] __attribute__((section("ddr")));
```

Bezpośredni dostęp do pamięci

- Procesor sygnałowy musi wymieniać dane z zewnętrznymi urządzeniami:
  - wejście – np. odczyt danych z czujnika,
  - wyjście – np. przesłanie informacji do wyświetlacza.
- Dane muszą być wymieniane między pamięcią a zewnętrznymi urządzeniami.
- Aktywne przesyłanie danych wymaga aktywności procesora: sprawdzania czy są nowe dane (*polling*) oraz kopiowania danych między pamięcią a wejściem/wyjściem.
- DMA (*direct memory access*) – bezpośredni dostęp interfejsów I/O do pamięci.
- Dane wejściowe są przenoszone do/z pamięci bez konieczności wykonywania instrukcji przez procesor – nie zużywają cykli procesora.
- Znaczące przyspieszenie działania programu wykorzystującego wejście/wyjście.

Bezpośredni dostęp do pamięci



Przerwania

Skąd procesor ma wiedzieć że są dostępne nowe dane?

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

Wyrównanie zmiennych w pamięci

- Każdy typ zmiennej zajmuje określoną liczbę bajtów, np. *long*: 4 bajty.
- Adres zmiennej – liczba całkowita wskazująca na położenie zmiennej w pamięci.
- Niektóre operacje wykonywane na procesorze sygnałowym wymagają aby zmienna w pamięci spełniała warunek wyrównania (*alignment*).
- Np. wyrównanie do 4: adres zmiennej podzielny bez reszty przez 4.
- Wyrównanie jest bardzo często warunkiem, aby kompilator wygenerował zoptymalizowany kod. Trzeba je wymusić przez dyrektywy w kodzie.

```
#pragma DATA_ALIGN(bufor, 4);
short bufor[2048];
```

```
short bufor[2048] __attribute__((aligned(4)));
```

Special Requirements

- Input vector x[ ] and output vector r[ ] must be aligned on 32−bit boundary. (2 LSBs of byte address must be zero)

Uwagi o kompilatorze

- Chcemy aby kompilator wygenerował zoptymalizowany kod pętli wykonujący dwie operacje w jednej instrukcji. Kompilator często 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” i wygeneruje wolniejszy, ale zawsze poprawnie działający kod.
- Musimy „zmuszać” kompilator do optymalizacji stosując specjalne dyrektywy preprocesora (*pragma*).

```
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] = _smpy(a[i], b[i]);
}
```

Podsumowanie: C vs. asembler

- Stosując asembler możemy napisać optymalny kod, ale to na nas spoczywa obowiązek zapewnienia, że kod będzie działał prawidłowo.
- Kompilator C ma zapewnić, że kod ZAWSZE będzie działał bez błędów.
  Jeżeli „widzi” potencjalne ryzyko, wstrzymuje optymalizacje kodu.
- Programista musi stosować „magiczne pragmy” aby zapewnić kompilator że kod może być zoptymalizowany.
- Niestety, często kompilator i tak uważa, że on wie lepiej. Nie generuje takiego kodu, jaki chcemy. (A może ma rację?)
- W takich sytuacjach, jeżeli jesteśmy przekonani że zoptymalizowany kod będzie działał poprawnie, pozostaje nam napisać ten kod samemu, w asemblerze.
- Możemy napisać w asemblerze tylko wybrane procedury i połączyć je z kodem w C.