Politechnika Gdańska, Wydział ETI

Procesory sygnałowe

ADSP-21161 SHARC®
Processor
Instrukcja laboratoryjna

Gdańsk 2006

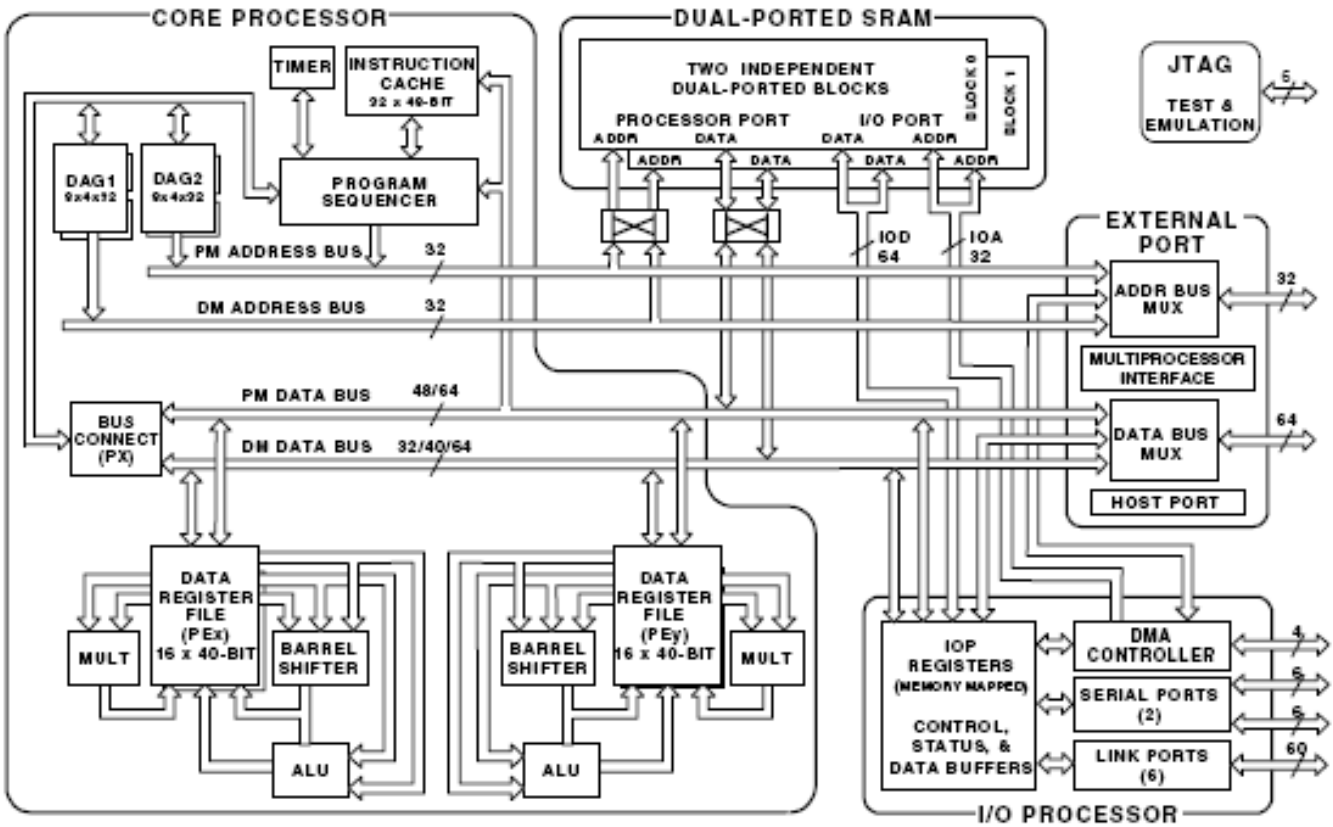
SPIS TREŚCI

1. Wstęp

Niniejsza instrukcja stanowi materiał przygotowujący do ćwiczenia laboratoryjnego z mikroinformatyki pod nazwą "Procesory sygnałowe". Ćwiczenie przeprowadzane jest w oparciu o system uruchomieniowy SHARC EZ-KIT Lite wyposażony w procesor sygnałowy ADSP-21161 SHARC firmy Analog Devices.

2. Architektura procesora sygnałowego - wybrane zagadnienia.

Procesor sygnałowy ADSP-21161 SHARC jest wysoko wydajnym 32-bitowym procesorem sygnałowym przeznaczonym do zastosowań w dziedzinie przetwarzania sygnałów akustycznych i wizyjnych. Rysunek 1 przedstawia blokowy schemat architektury tego procesora.



Rys. 1

W konstrukcji procesora można wyróżnić kilka zasadniczych bloków:

- Core Processor - stanowiąca rdzeń systemu jednostka odpowiedzialna za wykonywanie programu i wszystkie operacje obliczeniowe. Jest ona zgodna konstrukcyjnie ze starszymi procesorami serii ADSP - 21000.
- Dual-Ported SRAM - dwuportowa pamięć RAM podzielona na dwa bloki.
- I/O Processor - procesor wejścia-wyjścia odpowiedzialny za niezależną od jednostki obliczeniowej wymianę danych z otoczeniem
- External Port - moduł odpowiedzialny za komunikację procesora z innymi procesorami w systemach wieloprocesorowych lub za komunikację z procesorem nadrzędnym (nie musi to być procesor sygnałowy)
- JTAG - port umożliwiający podłączenie emulatora EZ-ICE.

2.1. Core Processor

Procesor ten składa się z następujących elementów:

- dwóch jednostek obliczeniowych (computation units)
- sekwencera programu (program sequencer)
- dwóch generatorów adresów dla danych (data adress generators DAG)
- timera
- pami ęci cache dla instrukcji (instruction cache)
- zbioru rejestrów (data register file)

2.1.1. Jednostki obliczeniowe

Rdzeń procesora składa się z dwóch jednostek obliczeniowych:

- ALU
- jednostki mnożącej (multiplier)
- jednostki przesuwającej (shifter)

Jednostki te mogą przetwarzać dane w następujących formatach:

- 32-bitowy stałoprzecinkowy
- 32-bitowy zmiennoprzecinkowy
- 40-bitowy zmiennoprzecinkowy

Jednostki te wykonują obliczenia w jednym cyklu. Dana wyjściowa z jednej jednostki może stanowić daną wejściową dla drugiej jednostki obliczeniowej. Możliwe jest również równoległe wykonywanie operacji np. dodawania i mnożenia dzięki temu, że operacje te wykonywane są przez różne jednostki obliczeniowe. Wymagany jest wtedy jednak określony wybór rejestrów przechowujących argumenty dla operacji.

2.1.2. Zbiór rejestrów (Data Register File) *2

Zawarte w "rdzeniu" rejestry są rejestrami ogólnego przeznaczenia i wykorzystywane są do przesyłania danych pomiędzy jednostkami obliczeniowymi a magistralami danych oraz do przechowywania wyników pośrednich. Zbiór rejestrów składa się z dwóch zestawów po 16 rejestrów 40-bitowych każdy. Umożliwia to szybkie przełączanie kontekstów

2.1.3. Sekwencer programu i generatory adresów

Dwa dedykowane generatory adresów oraz sekwencer programu wypracowują adresy wykorzystywane w trakcie dostępu do pamięci. Pozwala to na wykonywanie operacji obliczeniowych z maksymalną efektywnością. Wykorzystując pamięć cache programu procesor jest w stanie w tym samym czasie pobrać instrukcję oraz dwa operandy z pamięci. Dokładne informacje na temat generatorów adresów zamieszczone są w dalszej części instrukcji.

2.1.4. Pamięć cache

Sekwencer programu zawiera 32-słowowy cache instrukcji, który pozwala w efekcie na trzymagistralową operację pobrania kodu rozkazu oraz dwóch argumentów.

2.1.5. Przerwania

System przerwań procesora ADSP-21161 oparty jest na czterech zewnętrznych przerwaniach sprzętowych oraz grupie przerwań wewnętrznych. Trzy z przerwań sprzętowych są przerwaniami do zastosowań ogólnych, czwarte jest specjalnym przerwaniem powodującym reset procesora. Przerwania wewnętrzne pochodzą od timera, sterownika DMA, przepełnienia buforów cyklicznych, przepełnienia stosu, wyjątków w obliczeniach arytmetycznych. Generowane są również przerwania definiowane przez użytkownika oraz związane z pracą wieloprocesorową

2.1.6. Timer

Programowany timer pozwala na cykliczne generowanie przerwania. Wyposażony jest w 32-bitowy rejestr, który jest dekrementowany w każdym cyklu. Gdy osiągnie 0, generowane jest przerwanie.

2.1.7. Magistrale procesora

"Rdzeń" procesora wyposażony jest w cztery magistrale: magistralę adresową pamięci programu, magistralę adresową pamięci danych, magistralę danych pamięci programu i magistralę danych pamięci danych. Pamięć danych przeznaczona jest do przechowywania wartości zmiennych natomiast pamięć programu przechowuje zarówno program, jak też dane nie ulegające zmianie w trakcie działania programu np. tablice współczynników. Dzięki temu w pewnych przypadkach jeżeli kod instrukcji dostarczany jest z pamięci cache, można w jednym cyklu pobrać kod rozkazu i dwa argumenty. Wielkości magistral pokazane są na rys. 1.

2.1.8. Wewnętrzne przesyły danych

Prawie każdy rejestr w "rdzeniu" procesora klasyfikowany jest jako rejestr uniwersalny. Zaimplementowana jest grupa instrukcji pozwalających

na wewnętrzne przesyły pomiędzy rejestrami lub rejestrami i pamięcią programu lub danych włączając w to rejestry sterujące i rejestry statusu. Rejestr PX pozwala na wymianę danych pomiędzy magistralą danych pamięci programu a magistralą danych pamięci danych.

2.1.9. Przełączanie kontekstów

Większość rejestrów procesora posiada swoje alternatywne odpowiedniki w drugim zestawie co pozwala na szybkie przełączenie kontekstów przy np. obsłudze przerwań. O tym, który zestaw rejestrów jest dostępny decyduje ustawienie stosownego bitu w odpowiednim rejestrze sterującym.

2.1.10. Lista rozkazów

Lista rozkazów procesorów rodziny ADSP-21000 dostarcza dużą gamę rozkazów pozwalających na efektywne programowanie. Tzw. instrukcje wielofunkcyjne pozwalają na równoległe wykonywanie obliczeń i przesyłanie danych, jak również na jednoczesne wykonywanie mnożenia i operacji na ALU. Każda instrukcja może być wykonana w jednym cyklu procesora. Język assemblera wykorzystuje notację algebraiczną co zwiększa czytelność kodu.

2.2. Dwuportowa pamięć wewnętrzna

Procesor ADSP-21161 wyposażony jest w 1 Mb pamięci RAM zorganizowanej w dwa bloki po 0.5Mb każdy.

Każdy blok pamięci ma dostęp dwuportowy co pozwala na jednoczesny dostęp "rdzenia" procesora oraz procesora wejścia/wyjścia lub sterownika DMA.

Cała pamięć może być zorganizowana w słowa o długości 16, 32 lub 48 bitów. W procesorze ADSP-21161 pamięć może być skonfigurowana jako 32K słów 32-bitowych, 64K słów 16-bitowych lub 21.25K słów 48-bitowych dla instrukcji i 40-bitowych dla danych.

Każdy z bloków pamięci może przechowywać zarówno dane jak i program. Jednak zalecana jest taka organizacja, w której jeden blok przechowuje program, a drugi dane. Wtedy magistrale danych obsługują jeden blok, a magistrale programu drugi.

2.3. Pamięć zewnętrzna i interfejs urządzeń peryferyjnych

Procesor ADSP-21161 jest wyposażony w interfejs komunikacyjny z pamięcią zewnętrzną i urządzeniami peryferyjnymi. Zewnętrzna pamięć może być rozbudowana do 254M słów. Wszystkie wewnętrzne magistrale są multipleksowane i przetwarzane na 32-bitową zewnętrzną magistralę adresową i 48-bitową zewnętrzną magistralę danych. Zewnętrzna pamięć może być 16, 32 lub 48-bitowa.

2.4. Interfejs procesora nadrzędnego

Procesor ADSP-21161 może być w łatwy sposób dołączony do standardowej magistrali mikroprocesora tak 16 jak i 32-bitowego. Transmisja może być realizowana z prędkościami do szybkości zegara procesora DSP włącznie. Do połączenia tego dedykowane są cztery kanały DMA, i procesor nadrzędny może dzięki temu bezpośrednio odczytywać i zapisywać pamięć DSP.

2.5. Procesor wejścia/wyjścia

Procesor wejścia/wyjścia zawiera cztery porty szeregowe, dwa 8-bitowe tzw. link ports i sterownik DMA.

2.5.1. Porty szeregowe

Cztery synchroniczne porty szeregowe pozwalają na połączenie z szeroką gamą cyfrowych i cyfrowo-analogowych urządzeń zewnętrznych. Mogą one działać z maksymalną prędkością odpowiadającą szybkości zegara procesora czyli transmisja może odbywać się z prędkością do 40Mb/s. Funkcje nadawcze i odbiorcze każdego portu są niezależne. Dane z portów szeregowych mogą być transmitowane via DMA do lub z pamięci.

2.5.2. Link ports

Link ports to specyficznego typu porty równoległe. Porty te są najczęściej wykorzystywane do komunikacji w systemach wieloprocesorowych.
Są dwa, każdy o wielkości 8 bity. Mogą być taktowane dwa razy na cykl procesora co pozwala na przesył 8 bitów na cykl. Link ports mogą działać niezależnie i równolegle co pozwala na maksymalny transfer 240Mb/s. Dane są pakowane w słowa 32 lub 48-bitowe i mogą być odczytywane bezpośrednio przez "rdzeń" procesora lub za pośrednictwem DMA przesyłane do pamięci.
Procesor ADSP-21061 nie jest wyposażony w link ports.

2.5.3. Sterownik DMA

Procesor ADSP-21161 jest wyposażony w 14 kanałów DMA, dwa dla link ports, osiem dla portów szeregowych i cztery dla interfejsu zewnętrznego.
Poprzez DMA do procesora mogą być przesyłane zarówno dane jak i program. Możliwe są przesyły DMA pomiędzy pamięcią wewnętrzną i zewnętrzną lub urządzeniem zewnętrznym lub procesorem nadrzędnym.

Również dane z/do portów szeregowych lub link ports mogą być przesyłane poprzez DMA. Przy przesyłach zewnętrznych dane są automatycznie formatowane w słowa 16, 32 lub 48-bitowe.

3. Dane szczegółowe

Trzecia część instrukcji zawiera wybór informacji z danych technicznych procesora ADSP-21161, których znajomość jest niezbędna do przeprowadzenia zaplanowanego ćwiczenia laboratoryjnego. Znajdują się tu informacje dotyczące sprzętowej strony zagadnienia, jak również wybrane rozkazy assemblera oraz informacje na temat łączenia procedur assemblerowym z programami w języku C.

3.1 Organizacja generatorów adresów DAG

Generatory adresów (DAG) procesora generują adresy danych przenoszonych do i z Pamięci Danych (Data Memory - DM) i Pamięci Programu (Program Memory – PM). Poprzez generowanie adresów, DAG umożliwia programom odwoływanie się do adresów pośrednio, poprzez użycie rejestru DAG zamiast adresu bezwzględnego. Architektura DAG, która jest przedstawiona na Rysunki 3-1**Błąd! Nie można odnaleźć źródła odwołania.**, wspomaga kilka funkcji, które minimalizują przepełnienia przy procesie dostępu do danych. Do funkcji tych zaliczamy:

- **Dostarczenie adresu i post-modyfikacja** – dostarcza adresu podczas przenoszenia danych i automatycznie inkrementuje przechowywane adresu do następnego przeniesienia.
- **Dostarczanie adresu pre-modyfikowanego** – dostarcza zmodyfikowany adres podczas przenoszenia danych bez inkrementacji przechowywanych adresów.
- **Modyfikacja adresu** – inkrementuje przechowywane adresy bez przenoszenia danych.
- **Adresowanie bit-reversed** – dostarcza adres bit-revesed podczas przenoszenia danych bez odwracania przechowywanych adresów.
- **Broadcast'owe przenoszenie danych –** dostarcza podwójnego przenoszenia danych do komplementarnych rejestrów w każdym elemencie przetwarzania do wspomagającego trybu SIMD

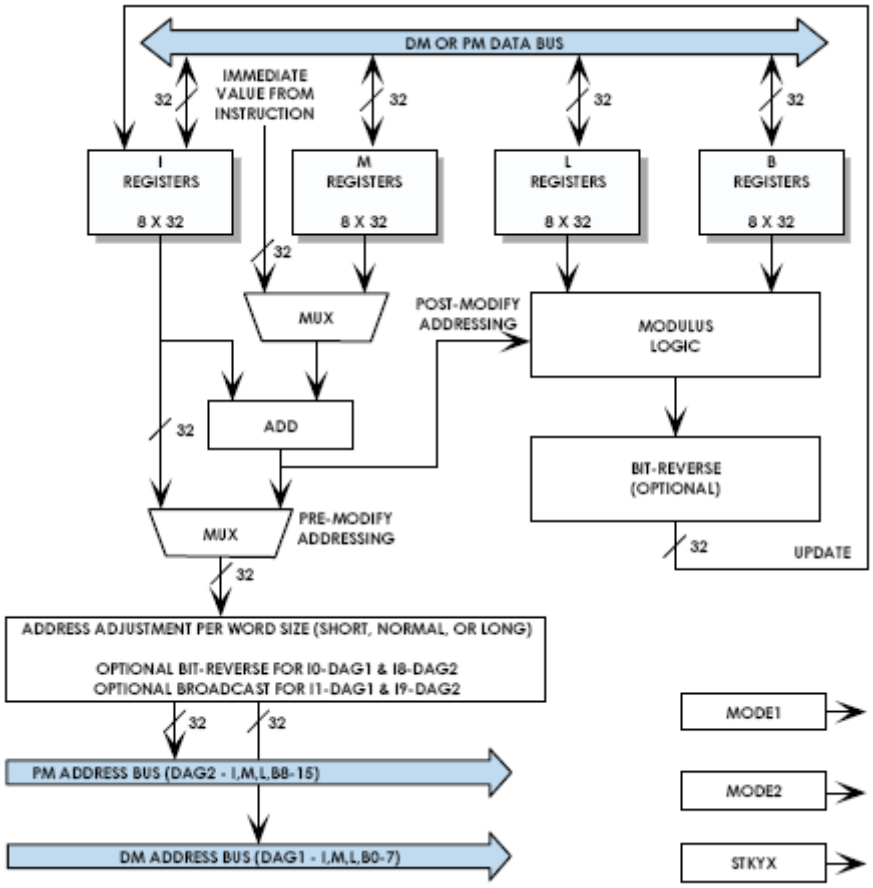
Jak pokazano na Rysunku 3-2, każdy DAG posiada cztery typy rejestrów. Rejestry te przechowują wartości, które DAG używa do generowania adresów. Cztery typy rejestrów to:

- **Rejestry indeksowe (I0-I7 dla DAG1 i I8-I15 dla DAG2).** Rejestr indeksowy przechowuje adres i pełni funkcję wskaźnika do pamięci. Na przykład, DAG interpretuje składnię DM(I0,0) i PM(I8,0) w instrukcji jako adres.
- **Rejestry modyfikujące (M0-M7 dla DAG1 i M8-M15 dla DAG2).** Rejestr modyfikujący zapewnia przyrost lub krok z jakim rejestr jest pre- lub post-modyfikowany podczas przesuwania rejestru. Na przykład, instrukcja DM(I0, M1) kieruje DAG na wyjście adres z rejestru I0 następnie modyfikuje zawartość I0 używając rejestru M1.
- **Rejestry długości i indeksowe (L0-L7 i B0-B7 dla DAG1 oraz L8-L15 i B8-B15 dla DAG2).** Rejestry długości i indeksowe ustalają zakres adresów i adres startowy dla buforów cyklicznych.

Ustawianie trybów DAG

Rejestr MODE1 kontroluje tryby pracy DAGs. Tabela A-2 zawiera listę wszystkich bitów dla MODE1. Następujące bity w MODE1 kontrolują tryby DAG:

- **Circular buffering enable (buforowanie cykliczne).** Bit 24 (CBUFEN) zezwala na cykliczne buforowanie (dla 1) lub wyłącza cykliczne buforowanie (dla 0).
- **Broadcast register loading enable (ładowanie rejestru broadcast), DAG1-I1.** Bit 23 (BDCST1) zazwala rejestrowi broadcast na ładowanie do komplementarnych rejestrów przeniesień indeksowanych z I1 (dla 1) lub wyłącza ładowanie broadcast (dla 0).



Rysunek 3-1

- **Broadcast register loading enable (ładowanie rejestru brodcast), DAG2-I9.** Bit 22 (BDCST9) zazwala rejestrowi broadcast na ładowanie do komplementarnych rejestrów przeniesień indeksowanych z I9 (dla 1) lub wyłącza ładowanie broadcast (dla 0).

- **SIMD mode enable (tryb SIMD).** Bit 21 (PEYEN) zezwala na obliczenia dla PEy—SIMD trybu—(dla 1) lub wyłącza PEy—SISD tryb—(dla 0).

- **Rejestry pomocnicze dla DAG2 lo, I,M,L,B8-11.** Bit 6 (SRD2L)
  Rejestry pomocnicze dla DAG2 hi, I,M,L,B12-15. Bit 5 (SRD2H)
  Rejestry pomocnicze dla DAG1 lo, I,M,L,B0-3. Bit 4 (SRD1L)
  Rejestry pomocnicze dla DAG1 hi, I,M,L,B4-7. Bit 3 (SRD1H)
  Bity te wybierają odpowiedni zespół rejestrów pomocniczych (dla 1) lub wybierają odpowiedni zespół rejestrów głównych—zespół dostępny po resecie—(dla 0).

- **Bit-reverse addressing enable (adresowanie bit-reverse), DAG1-I0.** Bit 1 (BR0) zezwala na adresowanie bit-reversed przeniesień adresowanych na I0 (dla 1) lub wyłącza adresowanie bit-reversed (dla 0).

- **Bit-reverse addressing enable (adresowanie bit-reverse), DAG2-I8.** Bit 0 (BR8) zezwala na adresowanie bit-reversed przeniesień adresowanych na (dla 1) lub wyłącza adresowanie bit-reversed (dla 0).

Tryb cyklicznego buforowania

Bit CBUFEN rejestru MODE1 ustawia buforowanie cykliczne—tryb, w którym DAG dostarcza zakres adresów w zakresie ograniczonej długości bufora (ustawianej przez rejestr L), zaczynając od adresu bazowego (ustawianego przez rejestr B), oraz inkrementuje adresy przy każdym dostępie poprzez zmianę wartości (ustawianej przez rejestr M).

ⓘ W wersjach 1.0 i późniejszych procesora ADSP-21161, bit Circular Buffer Enable (CBUFEN) w SYSCON jest ustawiany (=1) przy resecie. Dla wcześniejszych wersji silikonowych 0.x, bit ten jest zerowany (=0) przy resecie. Zmiana ta została wprowadzona aby zapewnić kompatybilność kodu z rodziną ADSP-2106x SHARC (ADSP-21060/1/2 i ADSP-21065L), gdzie buforowanie cykliczne jest aktywne po resecie.

Jednakże, Buforowanie cykliczne jest niekatywne po resecie w ADSP-21160. Zwróć na to uwagę podczas [portowania] kodu z procesora ADSP-21160 na procesor ADSP-21161.

Podczas używania buforów cyklicznych, DAG może wygenerować przerwanie na buforze przepełnionym (wrap around – zapętlanie).

Broadcast Loading Mode

Bity BDCST1 i BDCST9 rejestru MODE1 ustawiają tryb broadcast loading—wielokrotny rejestr ładuje się w pojedynczej komendzie ładowania. Kiedy jest ustawiony bit BDCST1 (1), DAG ładuje rejestr podwójnych danych na instrukcję wykorzystującą do adresu rejestru I1. DAG ładuje zarówno określony rejestr (rejestr jawny) w jednym elemencie przetwarzania, jak i ładuje rejestr komplementarny dla tego rejestru (rejestr niejawny) w drugim elemencie przetwarzania. Bit BDCST9 rejestru MODE1 umożliwia to dla rejestru I9.

Umożliwienie rejestrom, zarówno DAG1 jak i DAG2, ładowania broadcast nie wpływa w żaden sposób na rejestr zapamiętywany lub ładowany do uniwersalnych rejestrów, inaczej niż rejestr plikowy rejestrów danych. Tablica 4-1 pokazuje efekty operacji ładowania rejestru w obu elementach przetwarzania przy aktywnym ładowaniu broadcast rejestru. W Tablicy 4-2 zauważ, że Rx i Sx są komplementarnymi rejestrami danych.

Tablica 4-1. Podwójne przetwarzanie elementu ładowania broadcast rejestru

Instruction syntax	Rx = DM(I1,Ma); {Syntax #1} Rx = PM(I9,Mb); {Syntax #2} Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Syntax #3}
PEx explicit operations	Rx = DM(I1,Ma); {Explicit #1} Rx = PM(I9,Mb); {Explicit #2} Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Explicit #3}
PEy implicit operations	Sx = DM(I1,Ma); {Implicit #1} Sx = PM(I9,Mb); {Implicit #2} Sx = DM(I1,Ma), Sx = PM(I9,Mb); {Implicit #3}
1. Note that the letters a and b (as in Ma or Mb) indicate numbers for modify registers in DAG1 and DAG2. The letter a indicates a DAG1 register and can be replaced with 0 through 7. The letter b indicates a DAG2 register and can be replaced with 8 through 15.	

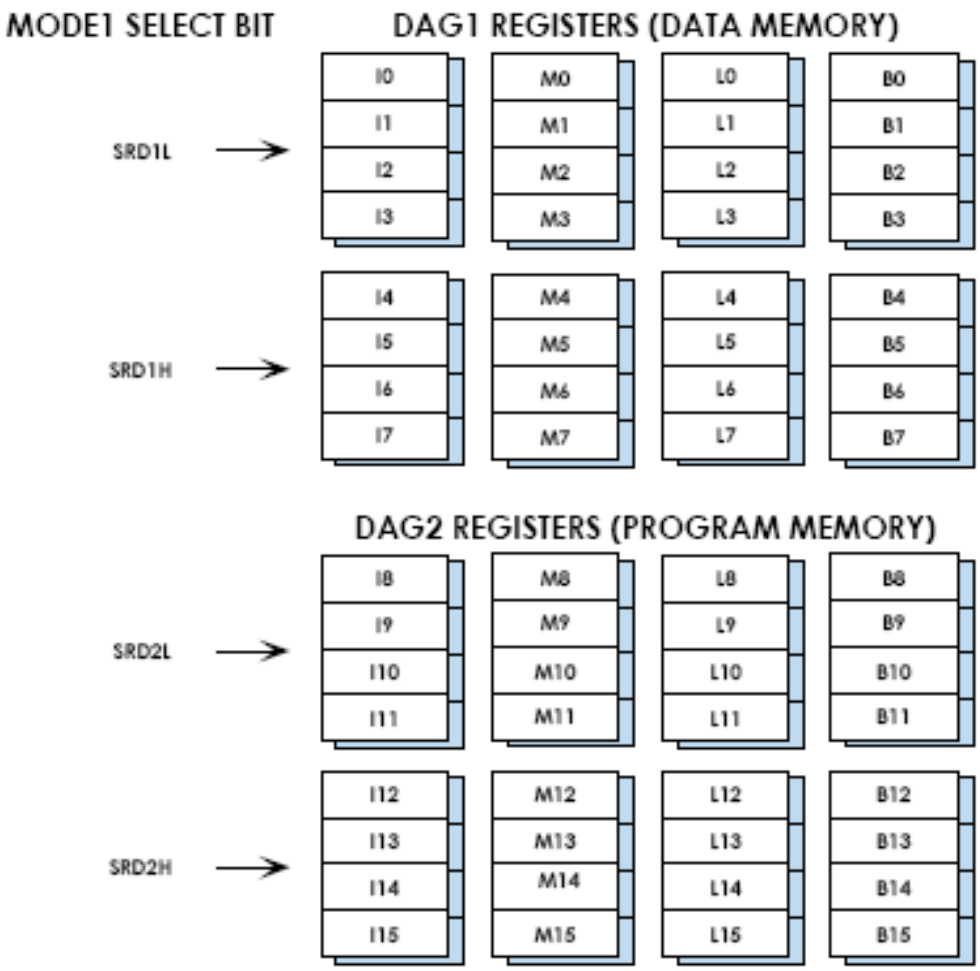
Bit PEYEN (wybór trybu SISD/SIMD) nie wpływa na operacje broadcast'u. Ładowanie broadcast jest szczególnie użyteczne w aplikacjach SIMD, gdzie algorytm wymaga identycznych danych ładowanych przy każdym przetwarzaniu elementu.

Alternatywne (pomocnicze) rejestry DAG

Każdy DAG posiada zestaw alternatywnych rejestrów. Aby ułatwić szybkie przełączanie context'ów, procesor włącza alternatywny zestaw rejestrów dla danych, wyników i rejestrów generatora danych. Podczas gdy rejestry alternatywne stają się dostępne, sterujące są bity w rejestrze MODE1. Gdy są niedostępne, zawartość rejestrów alternatywnych nie jest dostępna dla opracji procesora. Zauważ, że występuje opóźnienie jednego cyklu pomiędzy zapisaniem do MODE1 a możliwością dostępu zespołu rejestrów alternatywnych. W tym rozdziale zostały opisane zespoły rejestrów alternatywnych DAG.

Bity rejestru MODE1 mogą aktywować zespoły rejestrów alternatywnych w obrębie DAG'ów: niższa połowa DAG1 (I,M,L,B0-3), wyższa połowa DAG1 (I,M,L,B4-7), niższa połowa DAG2 (I,M,L,B8-11), i wyższa połowa DAG2 (I,M,L,B12-15). Rysunek 4-1 przedstawia podstawowe i alternatywne zespoły rejestrów generatora adresów.

Aby współdzielić dane pomiędzy context'ami, program umieszcza dane, które maja być współdzielone, w jednej połowie rejestrów aktualnego lub drugiego DAG i aktywuje zespoły rejestrów alternatywnych drugiej połowy. Następujący przykład demonstruje w jaki sposób kod powinien traktować jeden cykl opóźnienia pomiędzy



Rysunek 3-2

instrukcją ustawiającą bit w MODE1 a momentem kiedy otrzymujemy dostęp do rejestrów alternatywnych. Zauważ, że możliwe jest użycie każdej instrukcji, która nie zwraca się do przełączanego rejestru pliku zamiast instrukcji OP.

```
BIT SET MODE1 SRD1L;     /* Activate alternate dag1 lo regs */
NOP;                     /* Wait for access to alternates */
R0=DM(i0,m1);
```

Tryb adresowania bit-reverse

Bity BR0 i BR8 rejestru MODE1 ustawiają tyb adresowania bit-reverse — wyjściowe adresy w odwrotnej kolejności bitów. Kiedy ustawiony jest bit BR0 (1), DAG1 odwraca bity 32-bitowych adresów wyjściowych z I0. Gdy ustawiony jest bit BR8, DAG2 odwraca bity 32-bitowego adresu wyjściowego z I8. Generatory adresów odwracają kolejność botów tylko adresów wyjściowych z I0 lub I8; zawartości tych rejestrów pozostają niezmienione. Tryb adresowania bit-reverse dokonuje zarówno pre- jak i post-modyfikacji operacji. Następujący przykład pokazuje, w jaki sposób tryb bit-reverse mode wpływa na adres wyjściowy:

```
BIT SET Mode1 BR0;   /* Umożlliwienie adresowania bit-rev. dla DAG1 */
I0=0x8a000;          /* Ładowanie do I0 bazowego adresu bit reverse
                        bufora, DM(0x51000) */
M0=0x4000000;        /* Ładowanie do M0 wartości dla post-modyfikacji */
R1=DM(I0,M0);        /* Ładowanie do r1 adresu DM - DM(0x51000), który
                        jest odwrotną kolejnością bitów 0x8a000, następnie
post-modyfikacja I0 dla kolejnego dostępu
(0x8a0000x4000000)=0x408a000, co jest odwrotną kolejnością bitów
DM(0x51020) */
```

Oprócz trybu adresowania bit-reverse, procesor posiada pomocniczą instrukcję bit-reverse (BITREV). Instrukcja ta odwraca kolejność bitów zawartości wybranego rejestru.

Użycie statusu DAG

Generatory adresów mogą adresować w ograniczonym zakresie adresów, pracując cyklicznie na tych danych (lub buforach). Przepełnienie buforu (lub zapętlenie) występuje za każdym razem gdy DAG wraca przed adres bazowy bufora.

Generatory adresów mogą dostarczać do buforów danych powodujących przepełnienie podczas wykonywanie adresowania bufora cyklicznego dla I7 lub I15. Kiedy pojawia się przepełnienie bufora (operacja cyklicznego buforowania inkrementuje rejestr I register poza koniec bufora),odpowiedni DAG ustawia flagę przepełnienia bufora w rejestrze sticky status (STKYx). Przepełnienie bufora może również wygenerować przerwanie maskowane. Dwoma sposobami wykrycia przepełnienia bufora przy buforowaniu cyklicznym są:

- **Przerwania.** Umożliwienie przerwań i użycie procedury obsługi przerwań w celu natychmiastowego wykrycia warunku przepełnienia. Metoda ta jest odpowiednia gdy ważne jest wykrycie każdego przepełnienia w chwili jego pojawienia, na przykład w “ping-pong” lub przy obsłudze wymiany wskaźników buforów I/O.
- **Rejestry STKYx.** Użycie instrukcji BIT TST w celu sprawdzenia flag przepełnienia w rejestrze STKY po szeregu operacji. Jeśli ustawiona jest flaga przepełnienia, bufor został przepełniony – zapętlony – przynajmniej raz. Metoda ta jest użyteczna kiedy przechwycenie przepełnienia nie jest krytyczne.

Operacje DAG

Procesor generatorów adresów DAG, aby wygenerować adresy, wykonuje kilka typów operacji. Jak pokazano na Rysunku 4-1, rejestry DAG, MODE1, MODE2, i rejestry STKYx biorą udział w operacjach DAG. W tej części przedstawiono szczegóły dotyczące operacji DAG:

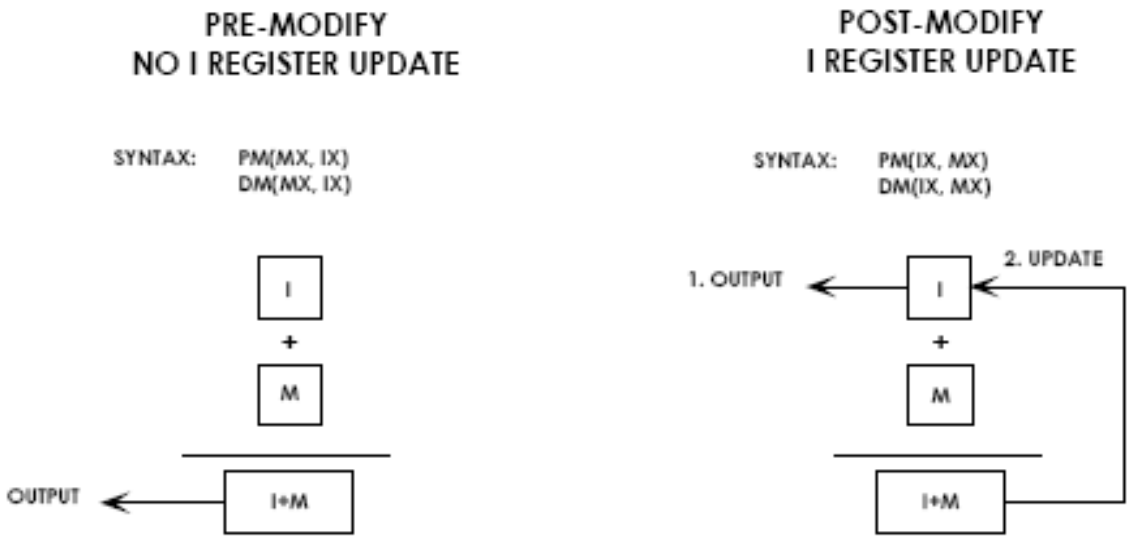
- Adresowanie za pomocą DAG
- Adresowanie buforów cyklicznych
- Modyfikowanie rejestrów DAG

Ważnym elementem wynikającym z Rysunku 3-1 jest to, że DAG automatycznie reguluje adres wyjściowy względem rozmiaru słowa lokacji adresu (krótkie, normalne lub długie słowo). To dopasowanie adresu umożliwia bezpośrednie użycie adresu przez pamięć wewnętrzną.

ⓘ Tryb SISD/SIMD, dostęp do rozmiaru słowa i lokacji danych (wewnętrzne/zewnętrzne) wpływają na operacje dostępu danych.

Adresowanie za pomocą DAG

Generatory adresów wspomagają dwa rodzaje zmodyfikowanego adresowania—generowanie adresu, który jest inkrementowany przez wartość lub rejestr. W adresowaniu pre-modyfikowanym, DAG dodaje offset, rejestr M lub bezpośrednią wartość, do rejestru I dając na wyjściu wynikowy adres. Adresowanie pre-modyfikowane nie zmienia (lub aktualizuje) rejestru I. Drugim typem modyfikowanego adresowania jest adresowanie post-modyfikowane. W adresowaniu post-modyfikowanym, DAG daje na wyjściu niezmienioną wartość rejestru I następnie dodaje rejestr M lub bezpośrednią wartość, uaktualniając wartość rejestru I. Rysunek 3-3 porównuje adresowani pre- i post-modyfikowane.



Rysunek 3-3. Operacje Pre i Post-modyfikacji

Różnica w składni assemblera procesora pomiędzy pre- a post-modyfikacją polega na pozycji indeksu i modyfikatora w instrukcji. Jeżeli rejestr I występuje przed modyfikatorem – instrukcja jest

operacją post-modyfikacyjną. Jeżeli modyfikator występuje przed rejestrem I, instrukcja jest instrukcją pre-modyfikowaną bez aktualizacji. Następująca instrukcja zwraca lokalizację pamięci programu wskazywaną bezpośrednio przez wartość w I15 i zapisuje wartość I15 + M12 do rejestru I15.

```
R6 = PM(I15,M12); /* Adresowanie post-modyfikowane z aktualizacją */
```

Porównując, poniższa instrukcja zwraca lokalizację pamięci programu bezpośrednio przez wartość I15 + M12 i nie zmienia wartości w I15:

```
R6 = PM(M12,I15); /* Pre-modify addressing without update */
```

Rejestry modyfikujące (M) mogą pracować z każdym rejestrem indeksowym (I) w tym samym DAG (DAG1 or DAG2).

Instukcje mogą używać, jako modyfikatora, liczb (bezpośrednich wartości) zamiast rejestru M. Rozmiar bezpośredniej wartości, która może modyfikować rejestr I zależy od typu instrukcji. Dla wszystkich pojedynczych operacji dostępu danych bezpośrednie wartości modyfikujące mogą mieć szerokość do 32 bitów. Instrukcje, które łączą adresowanie DAG z obliczeniami ograniczają bezpośrednią wartość modyfikującą. W tych instrukcjach (obliczenia wielofunkcyjne) szerokość bezpośredniej wartości modyfikującej wynosi do 6 bitów. Przykład instrukcji przyjmującej modyfikator do 32-bitów:

```
R1=DM(0x40000000,I1); /* DM address = I1+0x4000 0000 */
```

Przykład instrukcji akceptującej modyfikator do 6 bitów:

```
F6=F1+F2,PM(I8,0x0B)=ASTAT; /* PM address = I8, I8=I8+0x0B */
```

Zauważ, że adresowanie pre-modyfikowane nie może zmieniać obszaru adresu. Na przykład, adres pre-modyfikowany w przestrzeni wewnętrznej procesora nie powinien generować adresu w zewnętrznej przestrzeni pamięci.

Adresowanie buforów cyklicznych

Generatory adresów wspomagają adresowanie buforów cyklicznych- zakres adresów zawierających dane, które DAG przekazuje cyklicznie, “zapętla się”, aby powtórzyć przekazywanie zakresu adresów. Aby zaadresować cykliczny bufor, post-modyfikuje i aktualizuje indeks przy każdym dostępie z dodatnią lub ujemną wartością (rejestr M lub bezpośrednia wartość). Jeżeli wskaźnik indeksowy będzie miał wartość spoza bufora, DAG odejmie lub doda długość bufora od lub do wartości, ustawiając wskaźnik indeksowy na początek bufora. Pomoc generatorów adresów przy adresowaniu buforów cyklicznych pokazany jest na Rysunku 3-1, a przykład adresowania bofora cyklicznego na Rysunku 3-4.

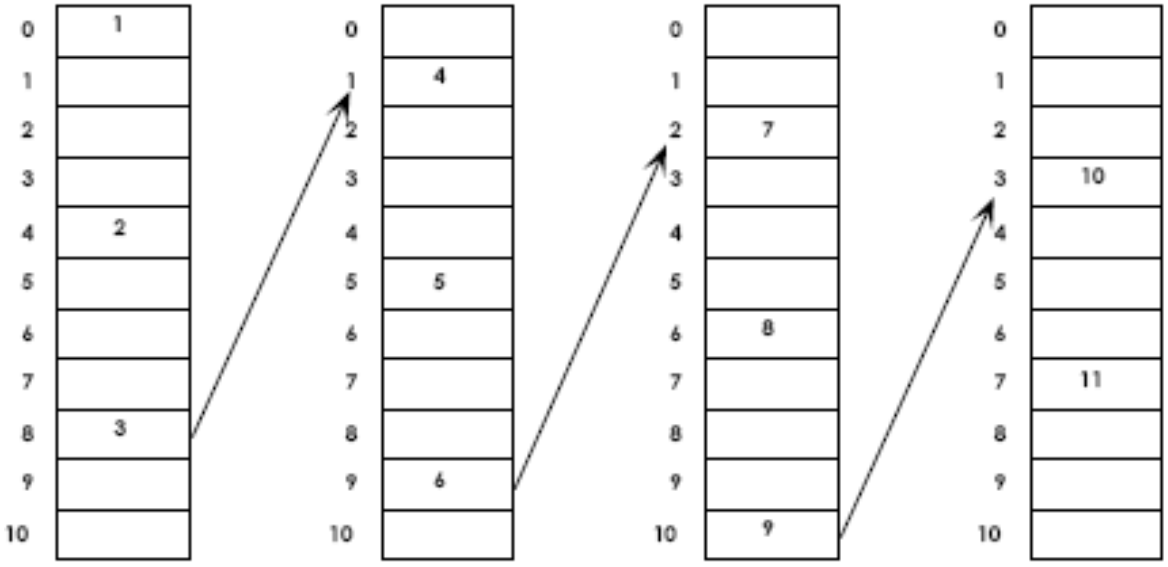
Adres startowy, do którego powraca DAG jest nazywany bazowym adresem bufora (rejestr B). Nie ma żadnych wymagań dotyczących wartości adresu bazowego dla bufora cyklicznego.

ⓘ Buforowanie cykliczne może wykorzystywać tylko adresowanie post-modyfikowane. Architektura DAG, pokazana na Rysunku 3-1, nie może wspomagać adresowania pre-modyfikowanego dla buforowania cyklicznego ponieważ buforowanie cykliczne wymaga, aby indeks była ktualizowany przy każdym dostępie.

Ważne jest, aby zauważyć, że DAG nie wykrywa przepełnienia lub niedomiaru mapy pamięci. Jeśli wywołany adres post-modyfikowany ma wartość I+M > 0xFFFF FFFF lub I–M < 0, buforowanie cyklicznie może funkcjonować nieprawidłowo. Również, długość bufora cyklicznego nie powinna umożliwiać buforowi osiągnięcie wierzchołka mapy pamięci.

```
THE FOLLOWING SYNTAX SETS UP AND ACCESSES A CIRCULAR BUFFER WITH:
        LENGTH = 11
        BASE ADDRESS = 0X55000
        MODIFIER = 4

        BIT SET MODE1 CBUFEN;          /* ENABLES CIRCULAR BUFFER ADDRESSING; JUST ONCE IN PROGRAM  */
        B0 = 0X55000;                  /* LOADS B0 AND L0 REGISTERS WITH BASE ADDRESS */
        L0 = 0XB;                      /* LOADS L0 REGISTER WITH LENGTH OF BUFFER */
        M1 = 0X4;                      /* LOADS M1 WITH MODIFIER OR STEP SIZE */
        LCNTR = 11, DO MY_CIR_BUFFER UNTIL LCE;   /* SETS UP A LOOP CONTAINING BUFFER ACCESSES */
        R0 = DM(I0,M1);                /* AN ACCESS WITHIN THE BUFFER USES POST MODIFY ADDRESSING */
        ...                            /* OTHER INSTRUCTIONS IN THE MY_CIR_BUFFER LOOP */
        MY_CIR_BUFFER: NOP;            /* END OF MY_CIR_BUFFER LOOP */
```



THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
NOTE THAT "0" ABOVE IS ADDRESS DM(0X55000). THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Rysunek 3-4

Jak przedstawiono na Rysunku 3-4, program przebiega w następujących krokach aby ustawić bufor cykliczny:

1. Umożliwienie buforowania cyklicznego (BIT SET Mode1 CBUFEN;). Operacja jest potrzebna tylko raz w programie.
2. Ładowanie adresu bazowego bufora do rejestru B. Operacja ta automatycznie ładuje odpowiadający rejestr I.

3. Ładowanie długości bufora do odpowiadającego rejestru L. Na przykład, L0 odpowiada B0.

4. Ładowanie zmodyfikowanej wartości (rozmiar kroku) do rejestru M w odpowiadającym DAG. Na przykład, M0 - M7 odpowiadają B0. Alternatywnie, program może użyć wartości bezpośredniej jako modyfikatora.

Po tych przygotowaniach, DAG używa logicznego modułu z Rysunku 3-1 aby przeprowadzić adresowanie buforów cyklicznych.

W procesorze ADSP-21161, programy umożliwiają buforowanie cykliczne poprzez ustawienie bitu CBUFEN w rejestrze MODE1 register. Bit ten ma odpowiadający maskowalny bit w rejestrze MMASK. Ustawienie odpowiadającego bitu MMASK powoduje wyczyszczenie bitu BUFEN poprzez instrukcję (PUSH STS), wywołanie przerwania zewnętrznego, przerwania układu czasowego lub przerwania wektorowego. Właściwość ta pozwala programom blokadę buforowania cyklicznego podczas obsługi przerwań, które nie używa buforowania cyklicznego. Poprzez blokadę buforowania cyklicznego, procedura nie wymaga zapamiętywania i odtwarzania rejestrów DAG s B i L.

Wyczyszczenie bitu CBUFEN blokuje buforowanie cykliczne dla wszystkich ładowanych danych i przechowywanych operacji. W zamian, DAG wykonuje standardowe ładowanie post-modyfikowane i przechowywanie dostępów, ignorując wartości rejestrów B i L. Zauważ, że zapisywanie do rejestru B modyfikuje odpowiadający mu rejestr I, niezależnie od stanu bitu CBUFEN. Instrukcja MODIFY wykonuje się niezależnie od stanu bitu CBUFEN. Instrukcja MODIFY zawsze wykonuje modyfikacje rejestru indeksowego bufora cyklicznego jeżeli odpowiadające rejestry B i L są przygotowane, niezależnie od stanu bitu CBUFEN.

ⓘ W wersjach 1.0 i późniejszych procesora ADSP-21161, bit Circular Buffer Enable (CBUFEN) w SYSCON jest ustawiany (=1) przy resecie. We wcześniejszych silikonowych wersjach 0.x, bit ten jest czyszczony (=0) przy resecie. Zmiana ta została wprowadzona aby zapewnić kompatybilność kodu z rodziną ADSP-2106x SHARC (ADSP-21060/1/2 i ADSP-21065L), gdzie buforowanie cykliczne jest aktywowane przy resecie. Jednakże, buforowanie cykliczne jest nieaktywne przy resecie dla ADSP-21160. Miej to na uwadze podczas portowania kodu z procesora ADSP-21160 do procesora ADSP-21161.

Przy pierwszym post-modyfikowanym dostępie do bufora DAG wystawia na szynę danych wartość rejestru I następnie modyfikuje adres poprzez dodanie wartości modyfikującej. Jeśli zauktalizowana wartość indeksu jest w granicach długości bufora, DAG zapisuje wartość do rejestru I. Jeśli zaktualizowana wartość jest poza długością bufora, DAG odejmuje lub odejmuje wartość rejestru L przed zapisem do rejestru I zaktualizowanej wartości indeksu. W równaniu, post-modyfikacja i operacje powrotu działają nastepująco:

- Jeśli M jest dodatnie:

  Inew = Iold + M if Iold + M < baza bufora + długość (koniec bufora)

  Inew = Iold + M – L if Iold + M ≥ baza bufora + length (koniec bufora)

- Jeśli M jest ujemne:

  Inew = Iold + M if Iold + M ≥ baza bufora (początek bufora)

  Inew = Iold + M + L if Iold + M < baza bufora (początek bufora)

Do adresowania buforów cyklicznych DAG używa wszystkich czterech typów rejestrów. Przy buforowaniu cyklicznym rejestry te pracują w następujący sposób:

• Rejestr indeksowy (I) zawiera wartości, które DAG wystawia na szynę adresową.

• Rejestr modyfikujący (M) zawiera wielkości post-modyfikujące (dodatnie lub ujemne), które DAG dodaje do rejestru I na końcu każdego dostępu do pamięci. Rejestrem M może być każdy rejestr M w tym samym generatorze adresów, tak samo jak rejestr I i nie musi mieć tego samego numeru. Wartością modyfikującą mogą być, zamiast rejestru M, wartości bezpośrednie. Rozmiar wartości modyfkującej, z rejestru M lub bezpośredniej, muszą być mniejsze niż długość (rejestr L) bufora cyklicznego.

• Rejestr długości (L) ustawia rozmiar bufora cyklicznego i zakres adresów, w obrębie których krąży rejestr I DAG. L musi być dodatnie i nie może mieć wartości większej niż 231 – 1. Jeżeli wartość rejestru L wynosi zero, działanie jego bufora cyklicznego zostaje wstrzymane.

• Rejestr bazowy (B), lub rejestr B plus rejestr L, ma wartość, z którą DAG porównuje zmodyfikowaną wartość I po każdym dostępie. Gdy rejestr B jest ładowany, odpowiadający rejestr I jest jednocześnie ładowany tą samą wartością. Gdy I jest ładowany, B pozostaje bez zmian. Programy mogą odczytywać rejestry B i I niezależnie.

W każdym DAG jest jeden zestaw rejestrów (I7 i I15), który może generować przerwanie przy przepełnieniu bufora cyklicznego. (zapętlenie adresu).

Kiedy program musi użyć I7 lub I15 poza buforowanie cyklicznym i procesor ma niemaskowalne przerwania przepełnienia bufora cyklicznego, program powinien uniemożliwić generowanie tych przerwań poprzez ustawienie wartości rejestrów B7/B15 i L7/L15 zapobiegających pojawianiu się przerwań. Jeśli I7 miał dostęp do adresów z zakresu 0x1000–0x2000, program powinien ustawić B7=0x0000 i L7=0xFFFF. Ponieważ procesor generuje przerwania

bufora cyklicznego oparte na równaniach powrotu, ustawienie rejestru L na zero nie koniecznie wywoła pożądany skutek. Jeśli program używa obu przerwań przepełnienia bufora cyklicznego, powinien uniknąć używania odpowiadającego rejestru(ów) I (I7 lub I15), gdzie rozgałęzianie przerwań nie jest konieczne.

W przypadku przerwania przepełnienia bufora cyklicznego, jeśli CBUFEN = 1 i rejestr L7 = 0 (lub L15 = 0), wtedy przerwanie CB7I (lub CB15I) pojawia się przy każdej zamianie I7 (lub I15), po tym jak rejestr indeksowy (I7 lub I15) przekroczy wartość rejestru bazowego (B7 lub B15). Zachowanie to jest niezależne od context'u podstawowych i pomocniczych rejestrów DAG.

Modyfikacja rejestrów DAG

DAG wspomaga dwie operacje, które modyfikują wartość adresu w rejestrze indeksowym bez wyprowadzania adresu. Te dwie operacje, odwracanie kolejności bitów adresu i modyfikacja adresu, są użyteczne przy adresowaniu bit-reverse i przechowywaniu wskaźników.

Instrukcja MODIFY modyfikuje adresy w każdym rejestrze indeksowym DAG (I0-I15) bez dostępu do pamięci. Jeśli odpowiadające rejestrowi I rejestry B i L są usawione na buforowanie cykliczne, instrukcja MODIFY wykonuje specyficzny powrót bufora (w razie potrzeby). Składnia MODIFY jest podobna do adresowania post-modyfikowanego (indeks, następnie modyfikator). MODIFY akceptuje, jako modyfikator, zarówno 32-bitowe wartości bezpośrednie jak i rejestr. Następujący przykład dodaje 4 do I1 i aktualizuje I1 nową wartością:

```
MODIFY(I1,4);
```

Instrukcja BITREV modyfikuje i odwraca kolejność bitów adresów w każdym rejestrze indeksowym DAG (I0-I15) bez dostępu do pamięci. Instrukcja ta jest niezależna od trybu bit-reverse. Instrukcja

BITREV dodaje 32-bitową wartość bezpośrednią do rejestru indeksowego DAG, przerowadza bit-reverse wyniku, i zapisuje wynik do tego samego rejestru indeksowego. Następujący przykład dodaje 4 do I1, przeprowadza bit_reverse wyniku i aktualizuje I1 nową wartością:

BITREV(I1,4);

Adresowanie w trybach SISD i SIMD

Tryb Single-Instruction, Multiple-Data (SIMD - Pojedyncza Instrukcja, Wielu-Danych) (bit EYEN =1) nie zmienia operacji adresowania w DAG, ale zmienia ilość danych, które są przenoszone podczas każdego dostępu. DAG wystawia na szyny danych te same adresy w trybach SIMD i SISD. W trybie SIMD, pamięć procesora oraz elementy przetwarzania pobierają dane ze szczególnej lokacji (jawnie) w składni instrukcji i komplementarnej (niejawnej) lokacji.

Generatory ardesów, rejestry i pamięć

Rejestry DAG są częścią zespołu rejestru uniwersalnego procesora. Programy mogą ładować rejestry DAG z pamięci, innego uniwersalnego rejestru lub wartością bezpośrednią. Programy mogą przechowywać zawartość rejestrów DAG w pamięci lub w innym uniwersalnym rejestrze.

Rejestry DAG wspomagają dwukierunkowe przesyłanie danych rejestr-rejestr. Kiedy rejestr DAG jest źródłem danych, celem przeznaczenia może być rejestr danych plików rejestrowych**.** Taki transfer skutkuje tym, że zawartość pojedynczego rejestru źródłowego jest kopiowana do komplementarnego rejestru danych, w każdym elemencie przetwarzania.

Programy powinny zachować ostrożność w przypadku gdy rejestr DAG jest przeznaczeniem transferu z rejstru danych plików rejestrowych. Programy powinny używać operacji warunkowych aby wybrać jeden element przetwarzania lub żadnego jako źródło. Mając dwa elementy przetwarzania przyczyniające się do wartości źródła skutkuje to tym, że zapis elementu PEx ma pierwszeństwo przez zapisem elementu PEy.

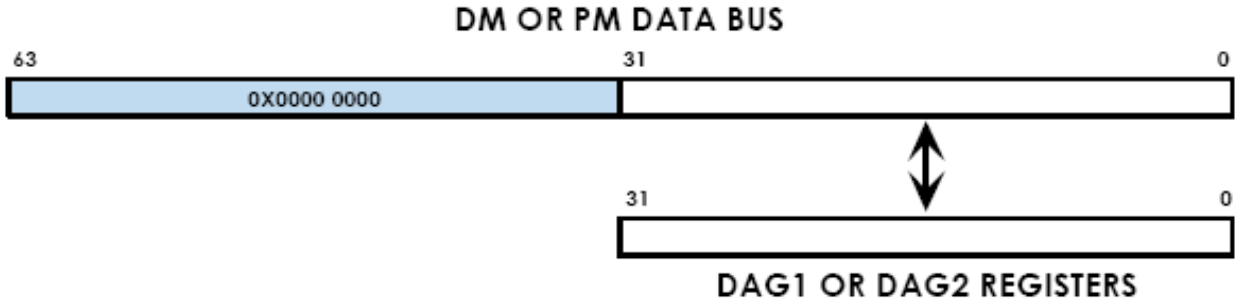
W przypadku gdzie rejestr DAG jest zarówno źródłem jak i przeznaczeniem, operacja przeniesienia danych wykonuje się tak jakby tryb SIMD był wyłączony. (PEYEN cleared).

DAG Register-to-Bus Alignment

Występują trzy rodzaje przypisania słowa do rejestrów DAG i szyny danych PM lub DM. Słowo normalne, normalne słowo o rozszerzonej precyzji i Długie słowo (Normal word, Extended-precision Normal word, i Long word)

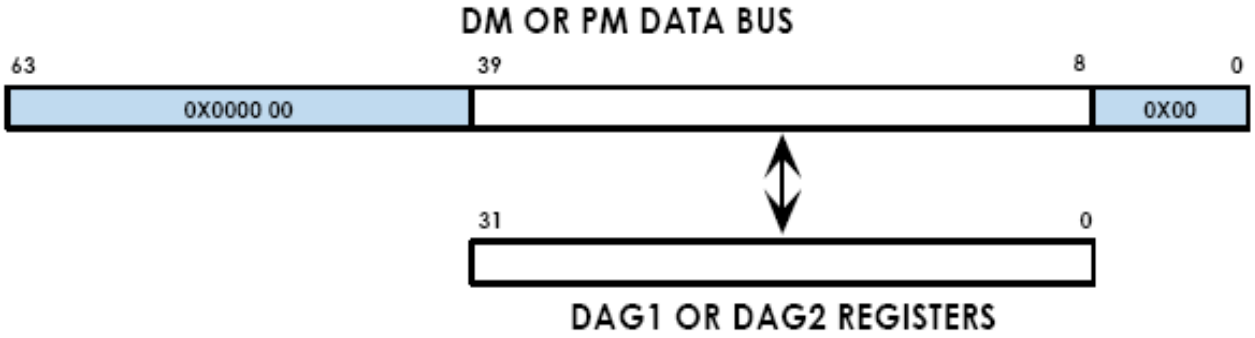
.

DAG przypisuje słowo normalne (32-bit) transferu adresowanego do niższej części bitów szyny. Te transfery pomiędzy pamięcią a 32-bitowymi rejestrami DAG1 lub DAG2 używają 64-bitowych szyn danych DM i PM. Transfer ten pokazany jest na Rysunku 3-5.



Rysunek 3-5

DAG przypisuje normalne słowo o rozszerzonej (40-bit) transferu adresowego lub transferu rejestr-rejstr do bitów 39-8 szyny. Te transfery pomiędzy 40-bitowym rejestrem danych a 32-bitowymi rejestrami DAG1 lub DAG2 wykorzystują 64-bitową szynę danych DM i PM. Sytuacja ta przedstawiona jest na Rysunku 3-6.



Rysunek 3-6

Słowo długie (64-bit) adresowanego przesyłania danych pomiędzy pamięcią a 32-bitowymi rejestrami DAG1 lub DAG2 trafia do dwóch rejestrów DAG i wykorzystana jest 64-bitowa szyna danych DM i PM. Praca szyn w tym przypadku została przedstawiona na Rysunku 3-7

Rysunek 3-7

Jeżeli transfer długiego słowa określa parzysty numer rejestru DAG (n.p., I0 or I2), wtedy wartość parzystego rejestru jest przekazywana w niższej połowie 64-bitowej szyny, a wartość rejestru parzystego+1 przenoszona na wyższej części (bity 63-32) szyny danych.

W obu przypadkach: parzystym i nieparzystym numerze, wyraźnie określono rejestry źródłowe DAG lub wyjście bitów 31-0 długiego słowa adresowanej pamięci.

Ograniczenia transferu rejestrów DAG

Dwa typy ograniczeń transferu to: warunki wstrzymania i warunki niedozwolone, których procesor nie wykrywa.

Przy przesyłaniu danych do i z rejestrów DAG, automatycznie dodawany jest przez procesor dodatkowy cykl (NOP). Kiedy instrukcja ładująca rejestr DAG jest poprzedzona instrukcją korzystającą z rejestru tej samej pary[1] rejestrów DAG adresowania danych, zmodyfikowaną instrukcja lub niebezpośrednim skokiem, procesor wstawia dodatkowy cykl (NOP) pomiędzy te dwie instrukcje. Tego rodzaju wstrzymania występują, ponieważ ta sama szyna jest wykorzystywana w obu operacjach w tym samym cyklu. Dlatego, druga operacja musi być opóźniona. Przypadek ten powoduje opóźnienie ponieważ ukazuje się zależność zapisu/odczytu, gdzie zapis jest do I0 w jednym cyklu. Powoduje to, że zapis i odczyt do rejestru jest niemożliwy w jednym cyklu. Zauważ, że jeśli wszystkie instrukcje sprecyzowały I1, blokata będzie się stale pojawiać, ponieważ przesyłanie danych rejestru procesora DAG może okazać się w parach. DAG wykrywa zależności zapisu/odczyty z dokładnością do pary

:

```
I0=8;
DM(I0,M1)=R1;
```

[1] Rejestry DAG są dostępne parami są rozdzielone dla dostępu w pojedynczym cyklu. Parowane są nieaprzysty-parzysty. Na przykład. I0 i I1 są parą oraz I2 i I3 są parą

Inne sekwencje instrukcji szczególnie powodują nieprawidłowy wynik pracy procesora i są oflagowane jako błędy przez oprogramowanie assemblerowskie procesora. Instrukcje tego typu mogą być wykonane przez procesor, ale dadzą niepoprawny wynik:

- Instrukcja, która przechowuje rejestr DAG w pamięci używając pośredniego adresowania z tego samego DAG, z lub bez aktualizacji rejestru indeksowego. Instrukcja zapisuje niepoprawne dane do pamięci albo aktualizuje niepoprawny rejestr indeksowy.

  Nie próbuj tego: DM(M2,I1)=I0; lub DM(I1,M2)=I0;

  Ta przykładowe instrukcje nie działają ponieważ oba rejestry I0 i I1 są rejestrami DAG1.

- Instrukcja, która ładuje rejestr DAG z pamięci używając pośredniego adresowania z tego samego DAG, z aktualizacją rejestru indeksowego. Instukcja również ładuje rejestr DAG albo aktualizuje rejesttr indeksowy, ale nie oba na raz.

  Nie próbuj tego: L2=DM(I1,M0);

  Ta przykładowa instrukcja nie działa ponieważ oba rejestry L2 i I1 są rejestrami DAG1.

Instrukcje DAG – Podsumowanie

Tabele 4-3 do Tabeli 4-9 przedstawiają listę instrukcji DAG. W tabelach tych, zwróć uwagę na znaczenie poniższych symboli:

- **I15-8** wskazują na rejestry indeksowe DAG2: I15, I14, I13, I12, I11, I10, I9, lub I8, a **I7-0** wskazują na rejestry indeksowe DAG1 I7, I6, I5, I4, I3, I2, I1, or I0.
- **M15-8** wskazują na rejestry modyfikujące DAG2: M15, M14, M13, M12, M11, M10, M9, lub M8, a **M7-0** wskazują na rejestry modyfikujące DAG1 M7, M6, M5, M4, M3, M2, M1, or M0.
- **Ureg** wskazuje rejestr uniwersalny. Lista uniwersalnych rejesrów procesora Tablica A-1.
- **Dreg** wskazuje dowolny rejestr danych;
- **Data32** wskazuje dowlną wartość 32-bitową, **Data6** wskazuje dowolną wartość 6-bitową.

Table 4-2. Post-Modify Addressing, Modified By M Register and Updating I Register

DM(I7-0,M7-0)=Ureg (LW); {DAG1}
PM(I15-8,M15-8)=Ureg (LW); {DAG2}
Ureg=DM(I7-0,M7-0) (LW); {DAG1}
Ureg=PM(I15-8,M15-8) (LW); {DAG2}
DM(I7-0,M7-0)=Data32; {DAG1}
PM(I15-8,M15-8)=Data32; {DAG2}

Table 4-3. Post-Modify Addressing, Modified By 6-Bit Data and Updating I Register

DM(I7-0,Data6)=Dreg; {DAG1}
PM(I15-8,Data6)=Dreg; {DAG2}
Dreg=DM(I7-0,Data6); {DAG1}
Dreg=PM(I15-8,Data6); {DAG2}

Table 4-4. Pre-Modify Addressing, Modified By M Register (No I Register Update)

DM(M7-0,I7-0)=Ureg (LW); {DAG1}
PM(M15-8,I15-8)=Ureg (LW); {DAG2}
Ureg=DM(M7-0,I7-0) (LW); {DAG1}
Ureg=PM(M15-8,I15-8) (LW); {DAG2}

Table 4-5. Pre-Modify Addressing, Modified By 6-Bit Data (No I Register Update)

DM(Data6,I7-0)=Dreg; {DAG1}
PM(Data6,I15-8)=Dreg; {DAG2}
Dreg=DM(Data6,I7-0); {DAG1}
Dreg=PM(Data6,I15-8); {DAG2}

Table 4-6. Pre-Modify Addressing, Modified By 32-Bit Data (No I Register Update)

Ureg=DM(Data32,I7-0) (LW); {DAG1}
Ureg=PM(Data32,I15-8) (LW); {DAG2}
DM(Data32,I7-0)=Ureg (LW); {DAG1}
PM(Data32,I15-8)=Ureg (LW); {DAG2}

Table 4-7. Update (Modify) I Register, Modified By M Register

Modify(I7-0,M7-0); {DAG1}
Modify(I15-8,M15-8); {DAG2}

Table 4-8. Update (Modify) I Register, Modified By 32-Bit Data

Modify(I7-0,Data32); {DAG1}
Modify(I15-8,Data32); {DAG2}

Table 4-9. Bit-Reverse and Update I Register, Modified By 32-Bit Data

Bitrev(I7-0,Data32); {DAG1}
Bitrev(I15-8,Data32); {DAG2}

3.2. Plik konfiguracyjny pamięci

Jak już wcześniej wspomniano pamięc omawianych procesorów podzielona jest na dwa bloki i zasadniczo zgodnie z zaleceniami jeden blok powinien stanowić pamięć programu, a drugi pamięć danych. Aby jednak w sposób logiczny zorganizować pamięć stosuje się opis struktury pamięci przy pomocy plików .ach. Poniżej przedstawiono plik ez-kit.ach, który opisuje podział pamięci dla systemu uruchomieniowego EZ-KIT i pozwala na uruchamianie programów napisanych w języku C. Nazwy segmentów są w tym przypadku jednoznacznie określone i należy się do nich odwoływać w czasie pisania procedur assemblerowych łączonych z programami w C.

EZ-KIT Architecture File

```
!---------------------------------------------------------------------
.SYSTEM         SHARC_EZKIT_Lite;
!
! This architecture file is required for used with the SHARC EZ-KIT
! Lite development software. It is structured for use with the C
! compiler but also can be used with assembly code.
!
! This architecture file allocates:
!     Internal  133 words of 48-bit run-time header in memory block 0
!               16 words of 48-bit initialization code in memory block 0
!              619 words of 48-bit kernel code in memory block 0
!               7424 words of 48-bit C code space in memory block 0
!               4K words of 32-bit PM C data space in memory block 0
!
!               8K words of 32-bit C DM data space in memory block 1
!               4K words of 32-bit C heap space in memory block 1
!             3712 words of 32-bit C stack space in memory block 1
!              384 words of 32-bit kernel data in memory block 1

.PROCESSOR = ADSP21061;

!    ------------------------------------------------------------
!   Internal memory Block 0
!              ---------------------------------------------------------------
.SEGMENT/RAM/BEGIN=0x00020000 /END=0x00020084 /PM/WIDTH=48          seg_rth;
.SEGMENT/RAM/BEGIN=0x00020085 /END=0x00020094 /PM/WIDTH=48         seg_init;
.SEGMENT/RAM/BEGIN=0x00020095 /END=0x000202ff /PM/WIDTH=48         seg_knlc;
.SEGMENT/RAM/BEGIN=0x00020300 /END=0x00021fff /PM/WIDTH=48         seg_pmco;
.SEGMENT/RAM/BEGIN=0x00023000 /END=0x00023fff /PM/WIDTH=32        seg_pmda;


!    ------------------------------------------------------------
!   Internal memory Block 1
!              ---------------------------------------------------------------
.SEGMENT/RAM/BEGIN=0x00024000 /END=0x00025fff /DM/WIDTH=32          seg_dmda;
.SEGMENT/RAM/BEGIN=0x00026000 /END=0x00026fff /DM/WIDTH=32 /cheap  seg_heap;
.SEGMENT/RAM/BEGIN=0x00027000 /END=0x00027e7f /DM/WIDTH=32         seg_stak;
.SEGMENT/RAM/BEGIN=0x00027e80 /END=0x00027fff /DM/WIDTH=32        seg_knld;


!    ------------------------------------------------------------
!   External Memory Select 1 is reserved for the UART.
!   ------------------------------------------------------------

.ENDSYS;
```

3.3 Lista rozkazów - przegląd

W tej części instrukcji umieszczone są podsumowujące składnie dla każdej instrukcji.

Przegląd rozdziałów

W tym rozdziale umieszczone zostały następujące tematy.

- “Podsumowanie obliczeń i przenoszenia/modyfikowania”
- “Podsumowanie programu kontroli przepływu”
- “Podsumowanie przenoszenia natychmiastowego”
- “Podsumowanie pozostałych operacji”
- “Podsumowanie typów rejestrów”
- “Podsumowanie adresowania pamięci”
- “Instruction Set Notation Summary”
- “Podsumowanie warunków wykonania kodu”
- “Podsumowanie testowania warunków SISD/SIMD”
- “Podsumowanie instrukcji skrótowców kodu operacji”
- “Kodowanie uniwersalnego rejestru”
- “Instrukcja ADSP-21160 mapowania kodu operacji”

Podsumowanie obliczeń oraz przenoszeni/modyfikowania

Instrukcje obliczeń oraz przenoszenia/modyfikowania są zaklasyfikowane do Grupy I instrukcji i zapewniają obsługę działań matematycznych, warunkowych, dostępu do pamięci/rejestru. Szereg tabel umieszczonych w poniżej podsumowuje instrukcje Grupy I.

Typ 1: Compute, Dreg«···»DM | Dreg«···»PM

compute	, DM(Ia, Mb) = dreg1 , dreg1 = DM(Ia, Mb)		, PM(Ic, Md) = dreg2 , dreg2 = PM(Ic, Md)	;

Typ 2: Compute

IF COND compute ;

Typ 3: Compute, ureg«···»DM | PM, register modify

IF COND compute	, DM(Ia, Mb) , PM(Ic, Md)	= ureg (LW);	
	, DM(Mb, Ia) , PM(Md, Ic)	= ureg (LW);	
	, ureg =	DM(Ia, Mb) (LW); PM(Ic, Md) (LW);	
	, ureg =	DM(Mb, Ia) (LW); PM(Md, Ic) (LW);	

Type 4: Compute, dreg«···»DM | PM, data modify

IF COND compute	, DM(Ia, <data6>) , PM(Ic, <data6>)	= dreg ;
	, DM(<data6>, Ia) , PM(<data6>, Ic)	= dreg ;
	, dreg =	DM(Ia, <data6>) ; PM(Ic, <data6>) ;
	, dreg =	DM(<data6>, Ia) ; PM(<data6>, Ic) ;

Typ 5: Compute, ureg«···»ureg | Xdreg<->Ydreg

IF COND compute,	ureg1 = ureg2	;
	X dreg <-> Y dreg	

Typ 6: Immediate Shift, dreg«···»DM | PM

IF COND shiftimm	, DM(Ia, Mb) , PM(Ic, Md)	= dreg ;
	, dreg =	DM(Ia, Mb) ; PM(Ic, Md) ;

Typ 7: Compute, modify

IF COND compute	, MODIFY	(Ia, Mb) ; (Ic, Md) ;

Podsumowanie programu kontroli przebiegu

Instrukcje programu kontroli przebiegu są zaklasyfikowane jako Grupa II instrukcji, i umożliwiają kontrolę przebiegu wykonywania programu. Poniższe tabele podsumowują Grupę II instrukcji.

Typ 8: Direct Jump | Call

IF COND JUMP	<addr24> (PC, <reladdr24>)		(DB) (LA) (CI) (DB, LA) (DB, CI)	;
IF COND CALL	<addr24> (PC, <reladdr24>)		(DB) ;	

Typ 9: Indirect Jump | Call, Compute

IF COND JUMP	(Md, Ic) (PC, <reladdr6>)	(DB) (LA) (CI) (DB, LA) (DB, CI)	, compute , ELSE compute	;
IF COND CALL	(Md, Ic) (PC, <reladdr6>)	(DB)	, compute , ELSE compute	;

Typ 10: Indirect Jump | Compute, dreg«…»DM

IF COND Jump	(Md, Ic) (PC, <reladdr6>)	,Else	compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;	

Typ 11: Return From Subroutine | Interrupt, Compute

IF COND RTS	(DB) (LR) (DB, LR)	, compute , ELSE compute	;
IF COND RTI	(DB)	, compute , ELSE compute	;

Typ 12: Do Until Counter Expired

LCNTR =	<data16> ureg	, DO	<addr24> (PC, <reladdr24>)	UNTIL LCE;

Typ 13: Do Until

DO	<addr24> (PC, <reladdr24>)	UNTIL termination ;

Podsumowanie przeniesienia natychmiastowego

Instrukcje natychmiastowego przeniesienia są zaklasyfikowane do Grupy III instrukcji, zapewniają obsługę dostępu do pamięci/rejestru. Poniższe tabele przedstawiają podsumowanie Grupy III instrukcji.

Typ 14: Ureg«···»DM | PM (direct addressing)

DM(<addr32>) PM(<addr32>)	= ureg (LW);

ureg =	DM(<addr32>) (LW); PM(<addr32>) (LW);	

Typ 15: Ureg«···»DM | PM (indirect addressing)

DM(<data32>, Ia) PM(<data32>, Ic)	= ureg	(LW);

ureg =	DM(<data32>, Ia) PM(<data32>, Ic)	(LW);

Typ 16: Immediate data···»DM | PM

DM(Ia, Mb) PM(Ic, Md)	= <data32> ;

Typ 17: Immediate data···»Ureg

ureg = <data32> ;

Podsumowanie pozostałych operacji

Pozostałe instrukcje są zaklasyfikowane jako Grupa IV instrukcji, udostępniają system rejestrów, umożliwiają operowanie na bitach, i obsługę o niskim poborze mocy. Poniższe tabele podsumowują Grupę IV instrukcji.

Typ 18: System Register Bit Manipulation

BIT	SET	sreg <data32> ;
	CLR	
	TGL	
	TST	
	XOR	

Typ 19: I Register Modify | Bit-Reverse

MODIFY	(Ia, <data32>)	;
	(Ic, <data32>)	
BITREV	(Ia, <data32>)	;
	(Ic, <data32>)	

Typ 20: Push, Pop Stacks, Flush Cache

PUSH	LOOP ,	PUSH	STS ,	PUSH	PCSTK , FLUSH CACHE ;
POP		POP		POP	

Typ 21: Nop

```
NOP ;
```

Typ 22: Idle

```
IDLE ;
```

Typ 25: Cjump/Rframe

CJUMP	function (PC, <reladdr24>)	(DB) ;

RFRAME ;

Podsumowanie typów rejestrów

Tablica 2-1 oraz Tablica 2-2 zawierają listę rejestrów. Rejestry z Tablicy 2-1 są częścią rdzenia procesora DSP. Rejestry z Tablicy 2-2 mieszczą się w zintegrowanym procesorze I/O oraz zewnętrznej sekcji portów zewnętrznych DSP.

Tablica 2-1. Rejestry uniwersalne (Universal Registers - Ureg)

Register Type	Register(s)	Function
Register File (ureg & dreg)	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15	Processing element X register file locations, fixed-point
	F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15	Processing element X register file locations, floating-point
	S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15	Processing element Y register file locations, fixed-point
	SF0, SF1, SF2, SF3, SF4, SF5, SF6, SF7, SF8, SF9, SF10, SF11, SF12, SF13, SF14, SF15	Processing element Y register file locations, floating-point
Program Sequencer	PC	Program counter (read-only)
	PCSTK	Top of PC stack
	PCSTKP	PC stack pointer
	FADDR	Fetch address (read-only)
	DADDR	Decode address (read-only)
	LADDR	Loop termination address, code; top of loop address stack
	CURLCNTR	Current loop counter; top of loop count stack
	LCNTR	Loop count for next nested counter-controlled loop

Register Type	Register(s)	Function
System Registers (sreg & ureg)	MODE1	Mode control & status
	MODE2	Mode control & status
	IRPTL	Interrupt latch
	IMASK	Interrupt mask
	IMASKP	Interrupt mask pointer (for nesting)
	MMASK	Mode mask
	FLAGS	Flag pins input/output state
	LIRPTL	Link Port interrupt latch, mask, and pointer
	ASTATx	Element x arithmetic status flags, bit test flag, etc.
	ASTATy	Element y arithmetic status flags, bit test flag, etc.
	STKYx	Element x sticky arithmetic status flags, stack status flags, etc.
	STKYy	Element y sticky arithmetic status flags, stack status flags, etc.
	USTAT1	User status register 1
	USTAT2	User status register 2
	USTAT3	User status register 3
	USTAT4	User status register 4

Tablica 2-2 I/O and Multiplier Registers

Register Type	Register(s)	Function
IOP registers (system control)	SYSCON	System control
	SYSTAT	System status
	WAIT	Memory wait states
	VIRPT	Multiprocessor IRQ
IOP registers (system control)	MSGR0, MSGR1, MSGR2, MSGR3, MSGR4, MSGR5, MSGR6, MSGR7	Message registers
	BMAX	Bus timeout max
	BCNT	Bus timeout count
	ELAST	External address last
IOP registers (DMA)	EPB0, EPB1, EPB2, EPB3	External port FIFO buffers
	DMAC10, DMAC11, DMAC12, DMAC13	DMA controls (EPB0-3)
	DMASTAT	DMA status
	II0, IM0, C0, CP0, GP0, DB0, DA0	DMA 0 parameters (SPORT0 RX)
	II1, IM1, C1, CP1, GP1, DB1, DA1	DMA 1 parameters (SPORT1 RX)
	II2, IM2, C2, CP2, GP2, DB2, DA2	DMA 2 parameters (SPORT0 TX)
	II3, IM3, C3, CP3, GP3, DB3, DA3	DMA 3 parameters (SPORT1 TX)

Register Type	Register(s)	Function
IOP registers (DMA)	II4, IM4, C4, CP4, GP4, DB4, DA4	DMA 4 parameters (LBUF0)
	II5, IM5, C5, CP5, GP5, DB5, DA5	DMA 5 parameters (LBUF1)
	II6, IM6, C6, CP6, GP6, DB6, DA6	DMA 6 parameters (LBUF2)
	II7, IM7, C7, CP7, GP7, DB7, DA7	DMA 7 parameters (LBUF3)
	II8, IM8, C8, CP8, GP8, DB8, DA8	DMA 8 parameters (LBUF4)
	II9, IM9, C9, CP9, GP9, DB9, DA9	DMA 9 parameters (LBUF5)
	II10, IM10, C10, CP10, GP10, EI10, EM10, EC10	DMA 10 parameters (EPB0)
	II11, IM11, C11, CP11, GP11, EI11, EM11, EC11	DMA 11 parameters (EPB1)
	II12, IM12, C12, CP12, GP12, EI12, EM12, EC12	DMA 12 parameters (EPB2)
	II13, IM13, C13, CP13, GP13, EI13, EM13, EC13	DMA 7 parameters (EPB3)
IOP registers (Link ports)	LBUF0, LBUF1, LBUF2, LBUF3, LBUF4, LBUF5	Link port buffers
	LCTL0, LCTL1	Link buffer control
	LCOM	Link common control
	LAR	Link assignment
	LSRQ	Link service request
	LPATH1, LPATH2, LPATH3	Link path (mesh)
	LPCNT	Link path count (mesh)
	CNST1, CNST2	Link constant (mesh)

Register Type	Register(s)	Function
IOP registers (SPORTs)	STCTL0, SRCTL0, TX0, RX0, TDIV0, RDIV0, MTCS0, MRCS0, MTCCS0, MRCCS0, SPATH0, KEYWD0, KEYMASK0	SPORT 0 registers
	STCTL1, SRCTL1, TX1, RX1, TDIV1, RDIV1, MTCS, MRCS1, MTCCS1, MRCCS1, SPATH1, KEYWD1, KEYMASK1	SPORT 1 registers
Multiplier registers	MR, MR0, MR1, MR2,	Multiplier results
	MRF, MR0F, MR1F, MR2F	Multiplier results, foreground
	MRB, MR0B, MR1B, MR2B	Multiplier results, background

Memory Addressing Summary

ADSP-21160 DSP opiera się na następujących typach adresowania:

Adresowanie bezpośrednie

Adres bezwsględny (instrukcje typów 8, 12, 13, 14)

dm(0x000015F0) = astat;

if ne jump label2; {'label2' jest etykietą adresu}

PC- adresowanie względne (instrukcje typów 8, 9, 10, 12, 13)

call(pc,10), r0=r6+r3;

do(pc,length) until sz; {'length' jest zmienną}

Adresowanie (przy użyciu rejestrów DAG):

Post-modifikacja za pomocą rejestru M, aktualizacja rejestru I (instrukcje typów 1, 3, 6, 16)

```
f5=pm(i9,m12);

dm(i0,m3)=r3, r1=pm(i15,m10);
```

Pre- modifikacja za pomocą rejestru M, bez aktualizacji (instrukcje typów 3, 9, 10)

```
r1=pm(m10,i15);

jump(m13,i11);
```

Post-modyfikacja poprzez bezpośrednią wartość, aktualizacja rejestru I (instrukcja typu 4)

```
f15=dm(i0,6);

if av r1=pm(i15,0x11);
```

Pre- modyfikacja poprzez bezpośrednią wartość, bez aktualizacji (instrukcje typów 4, 15)

```
if av r1=pm(0x11,i15);

dm(127,i5)=laddr;
```

Instruction Set Notation Summary

Normy opisów składni instrukacji dla ADSP-210xx przedstawiono w Tabeli 2-3. Zostały tutaj również przedstawione pozostałe części składni instrukcji i informacji na temat kodu operacji.

Tabela 2-3. Instruction Set Notation

Notation	Meaning
UPPERCASE	Explicit syntax—assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)
;	Semicolon (instruction terminator)
,	Comma (separates parallel operations in an instruction)
italics	Optional part of instruction
`| option1 |` `| option2 |`	List of options between vertical bars (choose one)
compute	ALU, multiplier, shifter or multifunction operation (see the chapter "Computations Reference").
shiftimm	Shifter immediate operation (see the chapter "Computations Reference").
cond	Status condition (see condition codes in Table 2-4 on page 2-18)
termination	Loop termination condition (see condition codes in Table 2-4 on page 2-18)
ureg	Universal register
cureg	Complementary universal register (see Table 2-10 on page 2-28)
sreg	System register
csreg	Complementary system register (see Table 2-10 on page 2-28)
dreg	Data register (register file): R15-R0 or F15-F0
cdreg	Complementary data register (register file): R15-R0 or F15-F0 (see Table 2-10 on page 2-28)
creg	One of 32 cache entries, an entry consisting of a CH, CL, & CA

Notation	Meaning
Ia	I7-I0 (DAG1 index register)
Mb	M7-M0 (DAG1 modify register)
Ic	I15-I8 (DAG2 index register)
Md	M15-M8 (DAG2 modify register)
<datan>	n-bit immediate data value
<addrn>	n-bit immediate address value
<reladdrn>	n-bit immediate PC-relative address value
+1	the incremented data, address or register value
(DB)	Delayed branch
(LA)	Loop abort (pop loop and PC stacks on branch)
(CI)	Clear interrupt
(LR)	Loop reentry
(LW)	Long Word (forces Long word access in Normal word range)

Podsumowanie warunków wykonania kodu

W instrukcjach warunkowych, wykonanie całej instrukcji jest zależne od sprecyzowanych warunków (kontynuacji lub zakończenia). Tabela 2-4 zawiera kody, których można użyć jako warunków (IF i DO UNTIL).

Table 2-4. Warunki IF oraz mnemoniki zakończenia Do/Until

Condition From	Description	True if...	Mnemonic
ALU	ALU = 0	AZ = 1	EQ
	ALU ≠ 0	AZ = 0	NE
	ALU > 0	footnote[1]	GT
	ALU < zero	footnote[2]	LT
	ALU ≥ 0	footnote[3]	GE
	ALU ≤ 0	footnote[4]	LE
	ALU carry	AC = 1	AC
	ALU not carry	AC = 0	NOT AC
	ALU overflow	AV = 1	AV
	ALU not overflow	AV = 0	NOT AV
Multiplier	Multiplier overflow	MV = 1	MV
	Multiplier not overflow	MV= 0	NOT MV
	Multiplier sign	MN = 1	MS
	Multiplier not sign	MN = 0	NOT MS
Shifter	Shifter overflow	SV = 1	SV
	Shifter not overflow	SV = 0	NOT SV
	Shifter zero	SZ = 1	SZ
	Shifter not zero	SZ = 0	NOT SZ

Condition From	Description	True if…	Mnemonic
Bit Test	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF
Flag Input	Flag0 asserted	FI0 = 1	FLAG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLAG0_IN
	Flag1 asserted	FI1 = 1	FLAG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLAG1_IN
	Flag2 asserted	FI2 = 1	FLAG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLAG2_IN
	Flag3 asserted	FI3 = 1	FLAG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLAG3_IN
Mode	Bus master true		BM
	Bus master false		NOT BM
Sequencer	Loop counter expired (Do)	CURLCNTR = 1	LCE
	Loop counter not expired (If)	CURLCNTR ≠ 1	NOT ICE
	Always false (Do)	Always	FOREVER
	Always true (If)	Always	TRUE

1 ALU greater than (GT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or $\overline{AZ}$ = 0
2 ALU less than (LT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 1
3 ALU greater equal (GE) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 0
4 ALU lesser or equal (LT) is true if: [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or $\overline{AZ}$ = 1

Podsumowanie testowania warunków SISD/SIMD

DSP obsługuje wykonanie warunku inaczej dla trybów SISD i SIMD. Są trzy sposoby, którymi warunki różnią się w trybie SIMD:

- W instrukcji obliczenia warunkowego (If ... Compute), każdy element przetwarzania przeprowadza obliczenie oparte na wyznaczaniu warunku w danym elemencie przetwarzania.
- W instrukcji warunkowej kontroli programu (If ... Jump/Call), układ sekwencyjny adresujący wykonuje skok/wywołanie oparte na logicznym mnożeniu warunków obu elementów przetwarzania.
- W instrukcjach obliczania warunkowego z klauzulą Cause, każdy element przetwarzania przeprowadza obliczenie Ese oparte na wyznaczaniu odwrotności warunku w danym elemencie przetwarzania.

Tabela 2-5 i Tabela 2-6 porównują wykonanie warunku SISD i IMD If-Else, które są dostępne w instrukcjach typów 9, 10 i 11.

Tabela 2-5. Wykonanie warunków w trybie SISD

Conditional test	ELSE modifier	Results for Type 11 (RTS)
0 (false)	0 (without else)	rts nops, compute nops
0 (false)	1 (else)	rts nops, compute executes
1 (true)	0 (without else)	rts executes, compute executes
1 (true)	1 (else)	rts executes, compute nops

Tabela 2-6 Wykonanie warunków w trybie SIMD

Conditional test		Else modifier	Results for Type 11 (RTS)
PEx	PEy		
0	0	0	rts nops, pex compute nops, pey compute nops
0	1	0	rts nops, pex compute nops, pey compute executes
1	0	0	rts nops, pex compute exe., pey compute nops
1	1	0	rts exe., pex compute exe., pey compute exe.
0	0	1	rts nops, pex compute exe., pey compute exe.
0	1	1	rts nops, pex compute exe., pey compute nops
1	0	1	rts nops, pex compute nops, pey compute exe.
1	1	1	rts exe., pex compute nops, pey compute nops

Podsumowanie instrukcji skrótowców kodu operacji

W kodzie operacji ADSP-21160 DSP, niektóre bity są jawnie zdefiniowane jako zera lub jedynki. Wartości pozostałych bitów lub pól ustawiane są poprzez rózne parametry instrukcji. Elementy z Tablicy 2-7 definiują te bity kodu operacji i pól. Nieokreślone bity są ignorowane w trakcie dekodowania instrukcji przez procesor, jednakże są zarezerwowane dla poźniejszego użycia.

Table 2-7. Skrótowce kodu operacji

Bit/Field	Description	States	
A	Loop abort code	0	Do not pop loop, PC stacks on branch
		1	Pop loop, PC stacks on branch
ADDR	Immediate address field		
AI	Computation unit register	0000	MR0F
		0001	MR1F
		0010	MR2F
		0100	MR0B
		0101	MR1B
		0110	MR2B
B	Branch type	0	Jump
		1	Call

Bit/Field	Description	States	
BOP	Bit Operation select codes	000	Set
		001	Clear
		010	Toggle
		100	Test
		101	XOR
COMPUTE	Compute operation field (see "Computations Reference" on page 7-1)		
COND	Status Condition codes	0–31	
CI	Clear interrupt code	0	Do not clear current interrupt
		1	Clear current interrupt
CREG	Instruction cache entry	0–31	
CS	Instruction cache register select code	00	Lower half of instruction RAM entry
		01	Upper half of instruction RAM entry
		11	Address CAM entry
CU	Computation unit select codes	00	ALU
		01	Multiplier
		10	Shifter
DATA	Immediate data field		
DEC	Counter decrement code	0	No counter decrement
		1	Counter decrement

Bit/Field	Description	States	
DMD	Memory access direction	0	Read
		1	Write
DMI	Index (I) register numbers, DAG1	0–7	
DMM	Modify (M) register numbers, DAG1	0–7	
DREG	Register file locations	0–15	
E	ELSE clause code	0	No ELSE clause
		1	ELSE clause
FC	Flush cache code	0	No cache flush
		1	Cache flush
G	DAG/Memory select	0	DAG1 or Data Memory
		1	DAG2 or Program Memory
INC	Counter increment code	0	No counter increment
		1	Counter increment
J	Jump Type	0	Non-delayed
		1	Delayed
L	Long Word memory address	0	Access size based on memory map
		1	Long word (64-bit) access size
LPO	Loop stack pop code	0	No stack pop
		1	Stack pop
LPU	Loop stack push code	0	No stack push
		1	Stack push

Bit/Field	Description	States	
LR	Loop reentry code	0	No loop reentry
		1	Loop reentry
NUM	Interrupt vector	0–7	
PMD	Memory access direction	0	Read
		1	Write
PMI	Index (I) register numbers, DAG2	8–15	
PMM	Modify (M) register numbers, DAG2	8–15	
PPO	PC stack pop code	0	No stack pop
		1	Stack pop
PPU	PC stack push code	0	No stack push
		1	Stack push
RELADDR	PC-relative address field		
S	UREG transfer/instruction cache read-load select	0	instruction cache read-load
		1	ureg transfer
SPO	Status stack pop code	0	No stack pop
		1	Stack pop
SPU	Status stack push code	0	No stack push
		1	Stack push
SREG	System Register code	0–15 (see "Universal Register Codes" on page 2-26)	
TERM	Termination Condition codes	0–31	
U	Update, index (I) register	0	Pre-modify, no update
		1	Post-modify with update

Bit/Field	Description	States
UREG	Universal Register code	0–256 (see "Universal Register Codes" on page 2-26)
RA, RM, RN, RS, RX, RY	Register file locations for compute operands and results	0–15
RXA	ALU x-operand register file location for multifunction operations	8–11
RXM	Multiplier x-operand register file location for multifunction operations	0–3
RYA	ALU y-operand register file location for multifunction operations	12–15
RYM	Multiplier y-operand register file location for multifunction operations	4–7

Kody rejestru uniwersalnego

Tablica 2-8, Tablica 2-9, Tablica 2-10 2-28 oraz Tablica 2-11 zawierają listę kodów bitowych dla rejestrów, które pojawiają się w polach kodu.

Tablica 2-8. Rejestry uniwersalne

Register	Description
PC	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR	fetch address
DADDR	decode address
LADDR	loop termination address
CURLCNTR	current loop counter

Register	Description
LCNTR	loop counter
R15–R0	X element register file locations
S15–S0	Y element register file locations
I15–I0	DAG1 and DAG2 index registers
M15–M0	DAG1 and DAG2 modify registers
L15–L0	DAG1 and DAG2 length registers
B15–B0	DAG1 and DAG2 base registers
PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

Tablica 2-9. Rejestry uniwersalne i systemowe

Register	Description
MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
MMASK	Mode mask
FLAGS	Flag pins input/output state
ASTATx	X element arithmetic status

Register	Description
STKYx	X element sticky status
ASTATy	Y element arithmetic status
STKYy	Y element sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2
USTAT3	user status reg 3
USTAT4	user status reg 4

Tablica 2-9. Rejestry komplementarne (Ureg–Cureg)

Register Type	SIMD Mode Complementary Registers
Data register (dreg & ureg)	R0–S0 R1–S1 R2–S2 R3–S3 R4–S4 R5–S5 R6–S6 R7–S7 R8–S8 R9–S9 R10–S10 R11–S11 R12–S12 R13–S13 R14–S14 R15–S15

Register Type	SIMD Mode Complementary Registers
System register (sreg & ureg)	USTAT1–USTAT2 USTAT3–USTAT4 ASTATx–ASTATy STKYx–STKYy
Bus exchange register (ureg)	PX1–PX2

Tablica 2-11 pokazuje w jaki sposób kod rejestru UREG pojawia się w PEx

Tablica 2-11. Kody uniwersalnego rejestru przetwarzania elementu X ((SISD/SIMD)

Bits: *3210* ⇩	Bits: *7654* 0000	0001	0010	0011	0100	0101	0110	0111
0000	R0	I0	M0	L0	B0	S0	FADDR	USTAT1
0001	R1	I1	M1	L1	B1	S1	DADDR	USTAT2
0010	R2	I2	M2	L2	B2	S2		MODE1
0011	R3	I3	M3	L3	B3	S3	PC	MMASK
0100	R4	I4	M4	L4	B4	S4	PCSTK	MODE2
0101	R5	I5	M5	L5	B5	S5	PCSTKP	FLAGS
0110	R6	I6	M6	L6	B6	S6	LADDR	ASTATx
0111	R7	I7	M7	L7	B7	S7	CURL-CNTR	ASTATy
1000	R8	I8	M8	L8	B8	S8	LCNTR	STKYx
1001	R9	I9	M9	L9	B9	S9	EMUCLK	STKYy
1010	R10	I10	M10	L10	B10	S10	EMUCLK2	IRPTL
1011	R11	I11	M11	L11	B11	S11	PX	IMASK
1100	R12	I12	M12	L12	B12	S12	PX1	IMASKP
1101	R13	I13	M13	L13	B13	S13	PX2	LRPTL
1110	R14	I14	M14	L14	B14	S14	TPERIOD	USTAT3
1111	R15	I15	M15	L15	B15	S15	TCOUNT	USTAT4

Tablica 2-12 pokazuje w jaki sposób kod rejestru UREG pojawia się w PEy

Tablica 2-11. Kody uniwersalnego rejestru przetwarzania elementu Y (SIMD)

Bits: *3210* ⇩	Bits: *7654* 0000	0001	0010	0011	0100	0101	0110	0111
0000	S0	I0	M0	L0	B0	R0	FADDR	USTAT2
0001	S1	I1	M1	L1	B1	R1	DADDR	USTAT1
0010	S2	I2	M2	L2	B2	R2		MODE1
0011	S3	I3	M3	L3	B3	R3	PC	MMASK
0100	S4	I4	M4	L4	B4	R4	PCSTK	MODE2
0101	S5	I5	M5	L5	B5	R5	PCSTKP	FLAGS
0110	S6	I6	M6	L6	B6	R6	LADDR	ASTATy
0111	S7	I7	M7	L7	B7	R7	CURL-CNTR	ASTATx
1000	S8	I8	M8	L8	B8	R8	LCNTR	STKYy
1001	S9	I9	M9	L9	B9	R9	EMUCLK	STKYx
1010	S10	I10	M10	L10	B10	R10	EMUCLK2	IRPTL
1011	S11	I11	M11	L11	B11	R11	PX	IMASK
1100	S12	I12	M12	L12	B12	R12	PX2	IMASKP
1101	S13	I13	M13	L13	B13	R13	PX1	LRPTL
1110	S14	I14	M14	L14	B14	R14	TPERIOD	USTAT4
1111	S15	I15	M15	L15	B15	R15	TCOUNT	USTAT3

Instrukcja mapowania kodu operacji ADSP-21160

Tablicae 2-13. Kody operacji ADSP-21160 DSP (Bity 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 1: Compute, Dreg«···»DM | Dreg«···»PM"	001			DMD	DMI			DMM			PMD	DM DREG				PMI			PMM		
"Type 2: Compute"	000			00001							COND										
"Type 3: Compute, ureg«···»DM | PM, register modify"	010			U	I			M			COND					G	D	L	UREG>		
"Type 4: Compute, dreg«···»DM | PM, data modify"	011			0	I			G	D	U	COND					DATA					
(a) "Type 5: Compute, ureg«···»ureg | Xdreg<->Ydreg"	011			1	0	SRC UREG					COND					SU			DEST UREG>		
(b) "Type 5: Compute, ureg«···»ureg | Xdreg<->Ydreg"	011			1	1		Y DREG				COND										
(a) "Type 6: Immediate Shift, dreg«···»DM | PM"	100			0	I			M			COND					G	D	DATAEX			
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Tablica 2-14. Kody operacji ADSP-21160 DSP (Bity 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PM DREG				COMPUTE																						
				COMPUTE																						
<UREG				COMPUTE																						
DREG				COMPUTE																						
<DEST UREG				COMPUTE																						
X DREG				COMPUTE																						

DREG				0	SHIFTOP						DATA								RN				RX			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Tablica 2-15. Kody operacji ADSP-21160 DSP (Bity 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
(b) "Type 6: Immediate Shift, dreg«···»DM | PM"	000			00010							COND							DATAEX			
"Type 7: Compute, modify"	000			00100						G	COND					I			M		
(a) "Type 8: Direct Jump | Call"	000			00110					B	A	COND										
(b) "Type 8: Direct Jump | Call"	000			00111					B	A	COND										
(a) "Type 9: Indirect Jump | Call, Compute"	000			01000					B	A	COND					PMI			PMM		
(b) "Type 9: Indirect Jump | Call, Compute"	000			01001					B	A	COND					RELADDR					
(a) "Type 10: Indirect Jump | Compute, dreg«···»DM"	110			D	DMI			DMM			COND					PMI			PMM		
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Tablica 2-16. Kody operacji ADSP-21160 DSP (Bity 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				0	SHIFTOP						DATA								RN				RX			

	COMPUTE

J		CI	ADDR

J		CI	RELADDR

J	E	CI		COMPUTE

J	E	CI		COMPUTE

DREG	COMPUTE

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Tablica 2-17. Kody operacji ADSP-21160 DSP (Bity 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
(b) "Type 10: Indirect Jump | Compute, dreg«···»DM"	111			D	DMI			DMM			COND					RELADDR					
(a) "Type 11: Return From Subroutine | Interrupt, Compute"	000			01010							COND										
(b) "Type 11: Return From Subroutine | Interrupt, Compute"	000			01011							COND										
(a) "Type 12: Do Until Counter Expired"	000			01100				DATA>													
(b) "Type 12: Do Until Counter Expired"	000			01101				0	UREG												
"Type 13: Do Until"	000			01110							TERM										
"Type 14: Ureg«···»DM | PM (direct address-ing)"	000			100			G	D	L	UREG						ADDR (upper 5 bits)					
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Tablica 2-18. Kody operacji ADSP-21160 DSP (Bity 26–0) – c.d.

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DREG				COMPUTE																						
J	E	L R		COMPUTE																						
J	E			COMPUTE																						
<DATA				RELADDR																						
				RELADDR																						
				RELADDR																						
ADDR (lower 27 bits)																										
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Tablica 2-19. Kody operacji ADSP-21160 DSP (Bity 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 15: Ureg«···»DM | PM (indirect addressing)"	101			G	I			D	L	UREG							DATA (upper 5 bits)				
"Type 16: Immediate data···»DM | PM"	100			1	I			M		G							DATA (upper 5 bits)				
"Type 17: Immediate data···»Ureg"	000			01111				0	UREG								DATA (upper 5 bits)				
"Type 18: System Register Bit Manipulation"	000			10100				BOP					SREG				DATA (upper 5 bits)				
(a) "Type 19: I Register Modify | Bit-Reverse"	000			10110				0	G				I				DATA (upper 5 bits)				
(b) "Type 19: I Register Modify | Bit-Reverse"	000			10110				1	G				I				DATA (upper 5 bits)				
"Type 20: Push, Pop Stacks, Flush Cache"	000			10111				LPU	LPO	SPU	SPO	PPU	PPO	FC							
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Tablica 2-20. Kody operacji ADSP-21160 DSP (Bity 26–0) – c.d.

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Tablica 2-21. Kody operacji ADSP-21160 DSP (Bity 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 21: Nop"	000			00000					0												
"Type 22: Idle"	000			00000					1												
Type 23: Idle16	Not supported on ADSP-21160																				
Type 24: creg«···»ureg	Not documented on ADSP-21160																				
(a) "Type 25: Cjump/Rframe"	0001				1000				0000				0100				0000				0
(b) "Type 25: Cjump/Rframe"	0001				1000				0100				0100				0000				0
(c) "Type 25: Cjump/Rframe"	0001				1001				0000				0000				0000				0
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Tablica 2-22. Kody operacji ADSP-21160 DSP (Bity 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000			ADDR																							
000			RELADDR																							
000			0000				0000				0000				0000				0000				0000			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

3.4 Łączenie procedur asemblerowych z programami w języku C

3.4.1 Funkce języka assemblerowego

Procedury assemblerowe mogą być łączone w programami w języku C. Z reguły są one umieszczane w plikach z rozszerzeniem `.s` lub `.asm`. Istnieje wiele zagadnień wymagających pisania programów w językach C-podobnych:

Wykorzystanie rejestrów
Wyszukiwanie parametrów
Wywoływanie innych funkcji
Wejście do funkcji
Wyjście z funkcji
Zasady nazywania symboli w języku assemblerowym

3.4.1.1 Wykorzystanie rejestrów

Należy uwzględnić kilka kwestii podczas używania rejestrów ADSP-21xxx przy programowaniu w języku asseblrerowskim w środowisku języka C:

- Większość rejestrów procesora jest dostępna dla programistów w języku asseblera.
- Część rejestrów jest specjalnego przeznaczenia i może być używana tylko do tych celów. Na przykład, niektóre rejestry powinny zawierać ustalone wartości.
- Jeżeli rejestry są używane w języku assemblerowym, przy zapisie i odczycie zawartości rejestórw muszą być zachowane zasady.
- Zaleca się używania makr z `asm_sprt.h` do zapisywania i odwoływania się do rejestrów na stosie.

3.4.1.1.1 Rejestry specjalnego przeznaczenia

Dwa rejestry – *wskaźnik stosu* i *wskaźnik ramki* – używane są do obsługi stosu w czasie działanie języka C. Stos, w trakcie działania C, jest wykorzystywany do przechowywania automatycznych zmiennych, parametrów przebiegu programu, przechowywania adresu powrotu funkcji i przechowywania pośrednich wyników obliczeń.

Wskaźnik stosu, rejestr I7, wskazuje *szczyt stosu*. Szczyt stosu jest kolejnym wolnym adresem na stosie. Adres stosu maleje do adresu 0. Zatem, jeżeli wartość jest "wyrzucana" na stos, jest zapisywana w miejsce gdzie wskazuje wskaźnik stosu, a wskaźnik stosu jest dekrementowany.

Wskaźnik ramki, rejestr I6, wskazuje początek ramki aktualnej funkcji.

3.4.1.1.2 Rejestry o ustalonej wartości

Rejestry M

Kilka rejestrów M jest używana do przechowywania ustalonych wartości. Nagłówek uruchomienia, który jest wykonywany przy uruchomieniu procesora, ustawia rejestry na ich wyznaczone wartości i wywołuje funkcję `main()`. Kompilator przyjmuje, że rejestry te posiadają ich przeznaczoną wartość. Funkcje języka assemblerowego nie mogą zmieniać tych wartości, ale mogą opierać się na ich zawartości, które wynoszą:

Rejestry DAG1	Rejestry DAG2	Wartość
M5	M13	0
M6	M14	1
M7	M15	-1

Rejestry L

Kompilator wymaga, aby rejestry L zawierały same zera. Wszystkie rejestry L, oprócz L6 i L7, mogą być ustawiane w języku assemblerowym na dowolną wartość. Jednakże, przed powrotem z funkcji assemblerowej, musza być z powrotem ustawione na zero.

3.4.1.1.3 Zapisywanie i odczytywanie rejestrów

Kompilator robi założenie, w jaki sposób funkcje traktują rejestry. Jeżeli kompilator „wie", że funkcja nie zmieni zawartości rejestru, kompilator może pozwolić na przechowanie danych w tym rejestrze podczas wykonywania funkcji. Jednakże, jeżeli kompilator stwierdzi, że funkcja zmieni zawartość rejestru, kompilator będzie unikał przechowywania użytecznych informacji w tym rejestrze podczas wykonywania funkcji.

Są dwa rodzaje rejestrów:

- *Rejestry kompilatora* są rejestrami, które – jak zakłada kompilator – są zabezpieczone w trakcie wywołania funkcji.

- *Rejestry tymczasowe* są rejestrami, które – w założeniu kompilatora – nie są zabezpieczone w trakcie wykonywania funkcji.

Zauważ: nie jest konieczne aby aktualnie wywoływana funkcja nadpisywała rejestry tymczasowe; w celach bezpieczeństwa, kompilator zakłada, że zostały one nadpisane.

Rejestry	Compiler Registers	Scratch
Rejesty danych	R3, R5, R6, R7, R9, R10, R11, R13, R14, R15	R0, R1, R2, R4, R8, R12
Rejestry indeksowe	I0, I1, I2, I3, I5, I8, I9, I10, I11, I14, I15	I4, I12
Rejestry modyfikujące	M0, M1, M2, M3, M8, M9, M10, M11	M4, M12
Pozostałe rejestry	MRF, MRB, MODE1, MODE2, USTAT1, USTAT2	

Niektóre zasady dotyczące zapisywania i odczytywania rejestrów:

1. Rejestry mogą być zapisywane poprzez wypchnięcie na stos, Wykonywane jest to w następujący sposób

```
dm(i7,m7)=r3;
```

   Instrukcja ta umieszcza na stosie rejestr R3 i dekrementuje wskaźnik stosu.

   Zauważ: rejestr M7 ma wartość ustaloną -1.

2. Wartość może być zdjęta ze stosu poprzez odczytanie jej, i skorygowanie wskaźnika stosu. W tym celu może zostać użyta następująca instrukcja:

```
r3=dm(1,i7);
modify(i7,1);
```

3. Wskaźnik stosu musi zawsze wskazywać pierwsze wolne miejsce na stosie.

   Zauważ: Jedynym wyjątkiem od tej zasady jest wartość wskaźnika stosu na końcu funkcji wywołanej instrukcją skoku. Hardware ADSP-210xx oczekuje na przerwania w trakcie tego cyklu, więc nie trzeba się martwić o to, że stos zostanie zniekształcony.

Niepoprawnym przykładem, jest następujący fragment kodu, gdyż przerwania może zdeformować stos.

```
modify(i7,1); /* NIE zalecane */
r3=dm(0,i7); /* NIE zalecane */
```

Jeżeli przerwanie pojawi się po modyfikacji a przed odczytem stos będzie wskazywał niepoprawne dane. Obsługa przerwanie nadpisze te dane.

4. Na początku obsługi funkcji języka assemblerowego, wszystkie rejestry kompilatora, które są wykorzystywane w trakcie funkcji, muszą zostać zapisane. Na końcu funkcji asseblerowej wszystkie te rejestry muszą zostać przywrócone.

5. Przed wywołaniem funkcji, wszystkie rejestry tymczasowe zawierające używane dane muszą zostać zapisane. Po wywołaniu funkcji wszystkie te rejestry musza zostać przywrócone.

3.4.1.1.4 Makra dostępu do stosu

Plik nagłówkowy `asm_sprt.h` zawiera wiele makr użytecznych do połączenia C i języka assemblerowego. Na przykład makro `puts()` wykonuje:

```
puts=r7;
```

Instrukcja ta zapisuje `r7` na stos. Podobnie makro `gets()` odczytuje ze stosu. Na przykład w celu odczytania ostatnio zapisanej wartości na stosie i zapisania jej do rejestru `r9` użyj instrukcji:

```
r9=gets(1);
```

Można użyć makra `restore_reg` w połączeniu z makrem `save_reg` aby zapisać i odczytać wszystkie rejestry plików rejestrowych (R0 – R15). Makra te można zaleźć w `sm_sprt.h`. Używaj ich jako wzór do konstrukcji kodu przy zapisywaniu i odczytywaniu rejestru.

3.4.1.1.5 Zespół rejestrów pomocniczych

Pliki rejestrowe rodziny procesorów ADSP-210xx posiadają zespół rejestrów pomocniczych dla rejestrów podstawowych. Środowisko uruchomieniowe C nie używa żadnych spośród rejestrów pomocniczych. Dlatego, zespołu rejestrów pomocniczych można dowolnie używać w języku assemblerowym. Środowisko uruchomieniowe C nie jest zakłócane poprzez używanie zespołu rejestrów pomocniczych. Po powrocie do używania zestawu rejestrów podstawowych, są one w stanie w jakim zostały.

3.4.1.2 Wyszukiwanie parametrów

W tej sekcji został opisany sposób dostępu do parametrów funkcji w języku C.

3.4.1.2.1 Gdzie znajdują się parametry

W środowisku C parametry są przekazywane do funkcji poprzez umieszczenie ich w rejestrach na stosie, zgodnie z poniższymi regułami:

1. W rejestrach może być przekazywane do trzech argumentów:

   Pierwszy argument przekazywany przez rejestr jest umieszczany w `R4`.

   Drugi argument przekazywany przez rejestr jest umieszczany w `R8`.

   Trzeci argument przekazywany przez rejestr jest umieszczany w `R12`.

2. Jeżeli jeden argument zostaje zapisany na stosie, wszystkie pozostałe (na prawo od niego) również zostają zapisane na stosie.

3. Wszystkie wartości szersze niż 32-bity są przenoszone na stos. Dotyczy to tym samym zmiennych typu `double` i `complex`, i struktur przekazywanych przez wartość.

   Za każdym razem, gdy typ `double` lub `float` jest umieszczany na stosie, bardziej znaczące słowo zostaje umieszczone w niższym adresie, a mniej znaczące słowo – w wyższym adresie.

   Za każdym razem gdy typ `complex` jest umieszczany na stosie, część rzeczywista jest zapisywana pod niższym adresem, a część urojona – pod wyższym adresem.

4. Ostatni nazwany argument wywoływanej funkcji z wieloma argumentami zostaje zapisany na stosie.

3.4.1.2.2 Przykłady przekazywania parametrów

Rozważmy przykładowy prototyp funkcji:

```
foo(int a, float b, char c, float d);
```

Pierwsze trzy argumenty `a, b, c` przekazywane są odpowiednio przez rejestry `R4, R8 i R12.` Czwarty parametr `d` jest zapisywany na stosie.

Następny przykład ilustruje przekazywaniu argumentu typu `double.`

```
bar(int a, double b, char c, float d);
```

Pierwszy argument `a` jest przekazywany w `R4.` Ponieważ drugi argument `b` jest wielosłownym argumentem, jest przekazywany na stos. W rezultacie, pozostałe argumenty `c` i `d` są przekazywane na stos.

Następny przykład pokazuje skutek przekazywania zmiennej ilości argumentów przy przekazywaniu parametrów.

```
test(float a, int b, char c,...);
```

Tutaj, pierwsze dwa argumenty, `a` i `b`, są przekazywane w rejestrach `R4` i `R8.` Ponieważ `c` jest ostatnim nazwanym argumentem, przekazywany jest na stos, tak jak pozostałe argumenty.

3.4.1.2.3 Dostęp do parametrów na stosie

Kiedy argumenty są umieszczane na stosie, zapisywane są w kolejności od prawej do lewej. Argument najbardziej na prawo zapisywany jest pod wyższym adresem w stosunku do argumentu najbardziej na lewo.

Poniższy przykład przedstawia w jaki sposób przekazywane są na stos.

```
test( int a, char b, float c, int d, int e, long f);
```

Parametry `a, b,` i `c` przekazywane są w rejestrach, ponieważ są parametrami o długości pojedynczego słowa. Pozostałe parametry – `d, e,` i `f` są przekazywana na stos.

Wszystkie parametry przechowywane na stosie są dostępne względem wskaźnika stosu, rejestru I6. Pierwszy parametr zapisany na stosie, `d`, jest zapisany pod adresem `sp+1`. Aby mieć do niego dostęp, można użyć następującej instrukcji:

```
r3=dm(1,i6);
```

Drugi parametr zapisany na stosie, jest pod adresem `sp+2` i może być dostępny dzięki instrukcji:

```
r3=dm(2,i6);
```

Trzeci parametr zapisany na stosie, `f`, jest typu `long`, więc jego bardziej znaczące słowo zapisane jest pod adresem `sp+3`, a mniej znaczące słowo na szczycie stosu. Dostęp do `f(MSW)` – Most Significant Word – można uzyskać dzięki instrukcji:

```
r3=dm(3,i6);
```

3.4.1.2.4 Makra do parametrów

Plik `asm_sprt.h` zawiera makro `reads()` służące do odczytywania parametrów ze stosu. Na przykład w celu odczytania do rejestru R5 drugiego wcześniejszego w stosunku do ostatnio przekazanego na stos parametru, należy użyć instrukcji:

```
r5=reads(2);.
```

3.4.1.3 Wywoływanie funkcji

Aby wywołać funkcję w środowisku C należy przejść przez protokół wywołujący. Makra w `asm_sprt.h` przeznaczone są do ułatwienia tego.

3.4.1.3.1 Protokół wywołania

Wywołanie funkcji w środowisku C składa się z kilku kroków:

1. Przesłanie argumentów do funkcji.

2. Adres powrotu musi być zapisany na stosie. Adres powrotu musi być adresem **bezpośrednio poprzedzającym** adres gdzie wykonanie ma być wznowione.

3. Musi zostać wykonany opóźniający rozgałęziony skok do funkcji wywoływanej. Skok jest używany zamiast wywołania, ponieważ powrót jest obsługiwany przez inny, niebezpośredni, skok. Preferuje się skoki do wywoływania funkcji, ponieważ wywołania są ograniczone przez możliwość zagnieżdżania stosu on-chip PC. Skoki nie mają ograniczeń zagnieżdżania, ponieważ mają możliwość zapisywania adresów powrotu w pamięci zewnętrznej. Użyta forma opóźnionego rozgałęzienia instrukcji skoku więc ustawiony wskaźnik ramki może znajdować się w dwóch gałęziach, które nie mogą być przerwane.

4. Musi zostać ustawiony wskaźnik ramki. Aktualny wskaźnik ramki funkcji, `I6`, zostaje zapisany do `R2`, a następnie wskaźnik stosu, `I7`, jest zapisywany do `I6`, aby stworzyć wskaźnik ramki wywoływanej funkcji.

5. W trakcie powrotu funkcji, może być konieczne ustawienie wskaźnika stosu w celu usunięcia argumentów funkcji ze stosu. Wykonane jest to poprzez dodanie stałej, liczby pozycji stosu zajmowanej przez argumenty, do wskaźnika stosu.

Protokół wywołania jest różny dla funkcji w języku C i zasad języka assemblerowego. Nie zakłóca to procedur wywołania w C. Kolejność wywołania jest różna dla ADSP-21020 i ADSP-2106x.

Użyj następującego kodu instrukcji do wywołania funkcji (`foo()`) dla ADSP-21020:

```
R2 = I6; /* Przechowanie starej ramki */
I6 = I7; /* Zamiana stosu i ramki */
JUMP (PC, _foo) (DB); /* SKOK do foo() */
DM(I7, M7) = R2; /* zapisanie starej ramki */
DM(I7, M7) = PC; /* zapisanie adresu powrotu */
```

Użyj następującego kodu instrukcji do wywołania funkcji (`foo()`) dla ADSP-2106x:

```
CJUMP _foo (DB); /* SKOK do foo(), zamiana */
DM(I7, M7) = R2; /* zapisanie starej ramki */
DM(I7, M7) = PC; /* zapisanie adresu powrotnego */
```

Stara ramka i adres powrotu są zapamiętywane przez „caller”, więc przedstawione wstępne „przygotowania” składają się tylko z wymiany danych między rejestrami.

Ostatnią procedurą instrukcji ADSP-21020 jest:

```
JUMP (M14, I12) (DB); /* Powrót do caller */
I7 = I6; /* Czyszczenie stosu */
I6 = DM (0, I16); /* Przywrócenie starej ramki */
```

Ostatnią procedurą instrukcji ADSP-2106x jest:

```
JUMP (M14, I12) (DB); /* Powrót do caller */
RFRAME; /* Przywrócenie stosu i ramki */
NOP; /* Użyte w celu ułatwienia op! */
```

3.4.1.3.2 Makra wywołania funkcji

Plik `asm_sprt.h` zawiera makra wykonujące wszystkie wymagane kroki przy wywoływaniu funkcji.

W celu umieszczenia wartości na stosie, użyj makra `puts()`. Na przykład, aby umieścić wartość rejestru `R3` na stosie, należy użyć instrukcji:

```
puts=R3;
```

W celu wywołania funkcji użyj makra `ccall()`. Na przykład, w celu wywołania funkcji `foo()`:

```
ccall(_foo);
```

Makro `ccall()` umieszcza adres powrotu na stosie, ustawia wskaźnik ramki, i wykonuje skok do następnej funkcji.

Makro `alter()` może służyć do usuwania wartości ze stosu. Na przykład, aby usunąć ostatnie trzy wartości ze stosu, należy użyć funkcji:

```
alter(3);
```

3.4.1.4 Wejście do funkcji

W trakcie wejścia do funkcji należy postępować zgodnie z protokołem wywołania środowiska uruchomieniowego C. Makra w `asm_sprt.h` przeznaczone są do ułatwienia tego procesu.

3.4.1.4.1 Co jest potrzebne na wejściu funkcji

Na wejściu funkcji, wywoływana funkcja musi zapisać wszystkie potrzebne informacje do powrotu do context'u, z którego jest wywoływana.

1. Wskaźnik ramki wywoływanej funkcji jest zapisywany do rejestru `R2` przez wywoływaną funkcję. Stary wskaźnik ramki musi być zapisany na stosie, aby móc później z niego korzystać.

2. Wywoływana funkcja zapisuje adres powrotu na stosie. Wartość ta musi być zapisana w celu późniejszego jej użycia.

3.4.1.4.2 Makra na wejście do funkcji

Makro `entry`, umieszczone w pliku `asm_sprt.h`, zapisuje na stosie zarówno wskaźnik ramki wywoływanej funkcji, jak i adres powrotu.

3.4.1.5 Wyjście z funkcji

3.4.1.5.1 Co jest potrzebne na wyjściu funkcji

Kilka operacji musi zostać wykonanych na końcu funkcji assemblerowej:

1. Zwracana wartość musi być zapisana w odpowiednim rejestrze(rejestrach). Jeżeli zwracana jest wartość długości pojedynczego słowa musi być zwrócona do rejestru R0. Jeżeli zwracana wartość jest długości dwóch słów musi być zwrócona do rejestrów R0 i R1.

   Jeśli zwracany jest typ `double`, R0 zawiera MSW (Most Significant Word – najbardziej znaczące słowo), R1 zawiera LSW (Less Significant Word – mniej znaczące słowo).

   Jeśli zwracany jest typ `complex`, R0 zawiera część rzeczywistą, R1 zawiera część urojoną.

2. Wartości, które zajmują rejestry kompilatora, które są zapisane na szczycie funkcji muszą być przywrócone do ich poprzedniej lokalizacji. Mogą być odczytywane ze stosu, względem wskaźnika ramki.

3. Musi zostać przywrócony wskaźnik stosu wywoływanej funkcji. Przed kontrolą przeniesienia, funkcja wywołująca kopiuje wskaźnik stosu, `I7`, do wskaźnika ramki, `I6`. W celu przywrócenia wskaźnika stosu wywołującej funkcji kopiuje wskaźnik ramki, `I6`, do wskaźnika stosu, `I7`.

4. Musi zostać przywrócony wskaźnik ramki wywoływanej funkcji. Poprzednio, była przenoszona poprzez wywołanie funkcji z `I6` do `R2`. Na szczyt wywoływanej funkcji i wtedy był przenoszony na stos. Teraz, musi być odtworzone do `I6`.

5. Kontrola musi być zwrócona do funkcji sterującej. Adres powrotny, który był zapisany na stosie przez funkcję wywoływaną, musi być przywrócony do rejestru `I` DAG2.

3.4.1.5.2 Makra powrotu

Kroki 3 – 5 opisane powyżej są włączone do pojedynczego makra, `exit`. Makro `exit` odczytuje adres powrotu ze stosu, przywraca wskaźnik stosu i wskaźnik ramki, oraz przywraca kontrolę wywoływanej funkcji.

3.4.1.6 Funkcje nierozgałęzione

Definicją funkcji nierozgałęzionej jest taka funkcja, która nigdy nie wywołuje innej funkcji. Istnieje kilka optymalizacji, które mogą zostać zastosowane do funkcji nierozgałęzionych, a nie są dozwolone przy funkcjach rozgałęzionych. Szczególnie, w funkcjach nierozgałęzionych nie ma potrzeby zapisywania ani wskaźnika ramki wywoływanej funkcji ani adresu powrotu na stos. Makra `leaf_entry` i `leaf_exit` są analogią do makr `entry` i `exit`, ale są bardziej skuteczne.

UWAGA: Makra te nie zapamiętują ani nie odczytują wartości rejestru `R2` – użycie tych makr nie niszczy zawartości `R2`.

3.4.1.7 Zasady nazywania symboli w języku assemblerowym

W celu połączenia funkcji C z funkcją języka assemblerowego używa się dyrektyw assemblerowych `.grobal` i `.extern`. W trakcie używania w języku assemblerowych, nazwy i zmienne języka C są poprzedzone podkreśleniem „_”.

Poniższy przykład pokazuje użycie połączonych funkcji C i assemblerowej.

Kod w języku C:

```
void asm_func(void);     /* Prototypy funkcji assemblerowe i c */
void c_func(void);

int c_var=10;            /* c_var zdefiniowana jako zmienna globalna;*/
                         /* używana w pliku .asm */
                         /* jako _c_var */

extern int asm_var;      /* asm_var zdefiniowana w pliku .asm */
                         /* jako _asm_var */
main () {
  asm_func();            /* wywołanie funkcji assemblerowej */
  }

void c_func(void) {      /* ta funkcja zostaje wywołana */
                         /* z pliku asm */
  if (c_var != asm_var)
    exit(1);
  else
    exit(0);
}
```

Kod w języku assemblerowym:

```
#include <asm_sprt.h>
.segment/dm seg_dmda;
.var _asm_var=0;         /* Definicja asm_var w tym miejscu */
.global _asm_var;        /* aby funkcja C mogła widzieć zmienną*/
.endseg;

.segment/pm seg_pmco;
.global _asm_func;       /* Definicja _asm_func */
.extern _c_func;         /* Definicja c_func w pliku C */
.extern _c_var;          /* Definicja c_var w pliku C */

_asm_func:
  entry;           /* Najpierw używaj makra
                   z pliku asm_sprt.h */

  r8=dm(_c_var);   /* Dostęp co zmiennej globalnej C */
  dm(_asm_var)=r8; /* przypisanie asm_var do c_var */

  ccall(_c_func); /* wywołanie funkcji C */

  exit; /* makro exit z pliku asm_sprt.h */

.endseg;
```

3.4.1.8 Przykłady

W podrozdziale tym zostaną przypomniane, przy pomocy przykładów, pojęcia do tej pory omawiane.

3.4.1.8.1 Proste programy assemblerowe

Najprostszymi programami assemblera są te, które nie mają argumentów ani nie zwracają wartości. Program, przedstawiony poniżej, może czekać na zdarzenie zewnętrzne, lub opóźnić o kilka cykli dostępność zmiennych globalnych. W tego typu programach zwróć szczególną uwagę na stosowanie rejestrów. Program assemblerowy musi zapisać i przywrócić każdy rejestr kompilatora, który został zmodyfikowany. Ponieważ proste opóźnienie nie wymaga zbyt wielu rejestrów, można użyć rejestrów pomocniczych, które ni muszą zostać zapamiętane.

```
/* void delay ( void );
   Podprogram w języku assembler opóźniający o N cykli
   gdzie N jest wartością zmiennej globalnej del_cycle */

  #include <asm_sprt.h>;

  .segment/pm seg_pmco;
  .extern _del_cycle;
  .global _delay;
  _delay:
     leaf_entry;         /* ta linia musi być jako pierwsza */
                         /* w każdej procedurze języka ssembler */
```

```
    R4 = DM ( _del_cycle);
                        /*użycie r4 ponieważ jest to rejestr */
                        /* pomocniczy i nie musi być chroniony */

    LCNTR = R4, DO d_loop UNTIL LCE;
d_loop: nop;

    leaf_exit;          /* Makro exit jest ostatną linią */
                        /* wykonywaną w każdej procedurze języka */
                        /* assembler. Makro exit zwraca kontrolę /
                        /* do programu wywołującego */
.endseg;
```

Zauważ, że wszystkie symbole z języka, do których następuje odwołanie, C poprzedzone są podkreśleniem. Ponieważ zarówno nazwa programu assemblerowego, `delay`, jak i zmienna globalna `_del_cycle` są dostępne w języku C, poprzedzone są podkreśleniem w kodzie programu assemblerowego..

3.4.1.8.2 Programy assemblerowe z parametrami

Kolejnymi, bardziej skomplikowanymi programami, są programy zawierające argumenty, ale nie zwracające wartości. Poniższy program dodaje pięć licz całkowitych przekazanych do programu przez parametry.

```
/* void add5 (int a, int b, int c, int d, int e);
 Procedura języka assembler dodająca 5 liczb */

#include <asm_sprt.h>

.segment/pm seg_pmco;
.extern _sum_of_5;      /* zmienna, gdzie suma będzie przechowywana */

.global _add5;

_add5:
leaf_entry;

/* pierwsze trzy parametry są przekazywane odpowiednio w r4, r8, r12, */
r4=r4+r8;               /* dodanie pierwszego i drugiego parametru */
r4=r4+r12;              /* dodanie trzeciego parametru */

/* dostęp do czwartego/piątego parametr poprzez reads(1)/reads(2) */
r8=reads(1);            /* przeniesienie czwartego parametry do r8 */
r4=r4+r8;               /* dodanie czwartego parametru */
r8=reads(2);            /* przeniesienie piątego parametru do r8 */
r4=r4+r8;               /* dodanie piątego parametru */

dm(_sum_of_5)=r4;       /* umieszczenie wyniku w zmiennej globalnej */

leaf_exit;
.endseg;
```

3.4.1.8.3 Programy assemblerowe zwracające wartość

Kolejną klasą programów assemblerowych są te, które zarówno posiadają parametry, jak i zwracają wartość. Prostym przykładem tego typu programu jest program dodający dwie liczby i zwracający ich sumę. Zwracane wartości są przechowywane w rejestrze `R0`.

```
/* int add2 (int a, int b);
   Procedura języka assemblera dodająca dwie liczby
   i zwracająca sumę */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* pierwsze dwa parametry przekazywane odpowiednio w r4, r8, */
/* zwracana wartośc jest zawsze zwracana poprzez rejestr r0 */

r0=r4+r8; /* dodanie pierwszego i drugiego parametru, przechowanie w r0*/

leaf_exit;
.endseg;
```

3.4.1.8.4Programy assemblerowe rozgałęzione

Bardziej skomplikowanym przykładem, wywołującym inna funkcję, będzie program zwracający wartość średnio-kwadratową dwóch licz zmiennoprzecinkowych ($z=(x2+y2)^{1/2}$). Pomimo że prosto można wyznaczyć pierwiastek kwadratowy w języku assemblerowym ADSP-21000, przykład ten wykorzystuje funkcję pierwiastka kwadratowego dostępną w bibliotece C. Ilustruje to sposób wywołania funkcji języka C w programie assemblerowym.

```
/* float rms ( float x, float y) ;
An assembly language subroutine to return the rms
z = (x^2 + y^2)^(1/2) */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:

entry;

f4=f4*f4;
f8=f8*f8;
f4=f4+f8;

/* f4 zawiera argument przekazywany do funkcji sqrtf */

/* użycie makra ccall w celu wywołanie funkcji w środowisku C */
```

```
ccall (_sqrtf);

/* f0 zawiera wynik zwracany przez funkcję sqrtf. Wynik musi
Być zwracany w f0, i tam teraz się znajduje */

exit;
.endseg;
```

Jeżeli wywoływana funkcja posiada więcej niż trzy parametry długości pojedynczego słowa, pozostałe parametry muszą zostać przekazane na stos, a usunięte ze stosu po wykonaniu funkcji. Następująca funkcja może wywołać program `add5` opisaną wcześniej.

```
/* int calladd5 ( void ) ;
Podprocedura języka assembler wywołująca inną pdocedurę
Z więcej  niż 3 parametrami. Tutaj: dodawanie liczb 1,2,3,4,5. */

#include <asm_sprt.h>

.segment /pm seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;
r8=2; /* drugi parametr jest przesyłany do r8 */
r12=3; /* trzeci parametr jest przesyłany do r12 */
r4=4; /* czwarty parametr jest przechowywany w r4 w celu zapisania na
       stos */
puts=r4; /* przeniesienie czwartego parametru na stos */
r4=5; /* czwarty parametr jest przechowywany w r4 w celu zapisania na
       stos */
puts=r4; /* przeniesienie piątego parametru na stos */
r4=1; /* pierwszy parametr przesyłany do r4 */

/* użycia makra ccall w celu wywołania funkcji w środowisku C */
ccall (_add5);
alter(2); /* usunięcie dwóch argumentów ze stosu */
r0=dm(_sum_of_5); /* _sum_of_5 jest zmienną gdzie add5 przechowuje swój
                     wynik */
exit;
.endseg;
```

Niektóre funkcje wymagają użycia rejestrów kompilatora. Zmienna musi być przechowywana w rejestrze kompilatora za każdym razem, gdy:

1. Ich okres istnienia obejmuje funkcję wywoływaną, lub

2. Nie ma już dostępnych rejestrów tymczasowych.

Poniższy przykład programu assemblerowego wykonuje operacje na elementach tablicy języka C.

```
/* void foo ( float function(float), float *array, int length);
   Procedura języka assembler wykonująca działania na tablicy języka C
array */

#include <asm_sprt.h>

.segment/pm seg_pmco;
.global _foo;
_foo:
entry;
puts=i8;      /* Używamy i8, rejestru kompilatora, ponieważ nie chcemy go
                 przechowywać dla każdej wywoływanej funkcji.
                 Rejestry kompilatora są chronione
                 podczas wywoływania funkcji */
r0=i1;
puts=r0;      /* z tego samego powodu zapisujemy i1 */

i8=r4;        /* odczyt pierwszego argumentu – adresu funkcji, która
                ma być wywołana */
i1=r8;        /* odczyt drugiego argumentu - tablica C zawierająca
                dane do przetwarzania */
r0=r12;       /* odczyt trzeciego argumentu – liczba pól tablicy */

lcntr=r0, do foo_loop until lce;     /* pętla poprzez pola danych */

f4=dm(i1,m5);             /* pobieranie danych z tablicy, zapisanie ich
                             do f4 jako parametry dla wywoływanej funkcji */
ccall(m13,i8);            /* wywołanie funkcji */
foo_loop: dm(i1,m6)=f0;   /* zapisanie zwracanej wartości w tablicy */
i1=gets(1);               /* przywrócenie wartości i1 */
i8=gets(2);               /* przywrócenie wartości i8 */
exit;
.endseg;
```

3.4.1.8.4 Pełne programy

Poniżej przedstawiony jest program w języku C wywołujący funkcję języka assembler. Funkcja ta oblicza iloczyn skalarny dwóch wektorów. Wektory ich długości przekazywane są poprzez parametry. Ponieważ rejestry pomocnicze używane są do przechowywania pośrednich wartości oraz pośrednich adresów, żaden z rejestrów nie musi być zapisywany ani odzyskiwany.

```
/* dot(int n, dm float *x, pm float *y); Obliczanie iloczynu skalarnego
Wektorów o długości n, jeden w dm drugi w pm. n musi być większe niż 2.*/

#include <asm_sprt.h>

.segment/pm seg_pmco;
                    /* Nazwa funkcji jest tworzona przez
                    poprzedzenie nazwy w C podkreśleniem */

.global _dot;
_dot:

  entry;            /* Zapamiętanie starego wskaźnika ramki i adresu
                    powrotu */
  r0=r4-1,i4=r8;    /* Załadowanie adresu pierwszego wektora do rejestru I
                     i załadowanie r0 długością-1 */

  r0=r0-1,i12=r12; /* Załadowanie adresu drugiego wektora do rejestru I
                    i załadowanie r0 długością-2 (ponieważ wykonujemy 2
                    iteracje osobno poprzez dostarczanie przesuwanie
                    kolejki */

  f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
 /* Zerowanie rejestru przechowującego wynik, rozpoczęcie przetwarzania*/

  f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
/* Drugi etap przetwarzania potokowego, również wykonywany wielokrotnie*/

  lcntr=r0, do dot_loop until lce;  /* Pętla wykonywana (długość-2)razy
                                     trzystopniowe przetwarzanie: czytaj,
                                    pomnóż, dodaj */

dot_loop:
  f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
  f8=f2*f4, f12=f8+f12;                    /* przesunięcie kolejki */

  f0=f8+f12;                          /* koniec z wynikiem w r0,
                                      skąd będzie zwracany */
  exit; /* trzeba przywrócić stary wskaźnik ramki i adres powrotu */
.endseg;
```

3.5. Opis kompilatora C g21k

3.5.1 Wstęp

G21k jest portem analogowym GCC, kompilatora Free Software Foundation's C, przeznaczonego dla procesorów sygnałowych rodziny ADSP-21000. Oddzielne wersje kompilatora są dostępne dla środowiska MS-DOS i UNIX.

G21k jest programem sterującym: kontroluje operacje innych programów (software development tools) podczas kompilacji. Proces kompilacji pliku źródłowego obejmuje kilka narzędzi i plików.

Inne pliki włączone do kompilacji są pikami nagłówkowymi uruchomieniowymi i plikami architektury. Nagłówkowe pliki uruchomieniowe kontrolują inicjalizację środowiska C oraz obsługę przerwań. Pliki architektury kontrolują ustawienia środowiska uruchomieniowego (typu położenie kody i danych, umiejscowienia stosu i sterty).

3.5.2 Wywoływanie G21K

G21K jest wywoływany z linii poleceń, i może być uruchomiony z okna DOS pod Windows'em. Pomimo że dokumentacja ta odwołuje się do kompilatora jako „G21K", w linii poleceń można wpisać `g21k`.

Poniżej przedstawiona jest składnia polecenia G21K

```
g21k [-przełącznik] nazwa_pliku [.ext ] [nazwa_pliku [.ext ]]
```

Niektóre, najczęściej używane opcje przełączników są przedstawione poniżej.

Kompilator może również zaakceptować kilka plików wejściowych, w następujący sposób:

```
g21k [-przełączniki] @file_all
```

Lista plików `file_all` zawiera pliki, które mają zostać skompilowane; musi to być prosty tekst zawierający jedną ścieżkę dostępu/ jedną nazwę pliku w linii. Cecha ta umożliwia funkcjonalność zgodnie z wymaganiami długości linii poleceń DOS.

Przełączniki wiersza poleceń moa być również umieszczane wewnątrz listy `file_all`, ale będą się odnosić do wszystkich plików wejściowych (np. przełączniki nie mogą być osobno podawane dla każdego pliku).

Switch	*Effect*
`-E`	Przetwarzanie tylko pliku źródłowego
`-S`	Wygenerowanie tylko plików assemblerowych
`-c`	Wygenerowanie tylko plików obiektowych (.obj)
`-O`	Optymalizacja kodu przy użyciu kilku optymalizacji
`-O2`	Optymalizacja kodu przy użyciu więcej optymalizacji
`-O3`	Optymalizacja kodu przy użyciu wszystkich optymalizacji
`-v`	Wygenerowanie obszernego pliku wyjściowego
`-ansi`	Wyłączenie wszystkich wyjątków nie-ANSI
`-g`	Utworzenie kodu debugg'owalnego dla CBUG
`-h`	Wyświetl listę przełączników
`-a filename`	Sporządzenie pliku specyfikacji alternatywnej architektury
`-Ipath`	Sporządzenie specyfikacji dodatkowych ścieżek dla załączonych plików
`-Dmacro[=value]`	Definicja makr dla preprocesora C
`-Lpath`	Sporządzenie specyfikacji dodatkowych ścieżek dla plików bibliotek
`-lxxx`	Dołączenie biblioteki `libxxx.a` w linii dołączania bibliotek
`-map`	Wygenerowanie pliku .map (domyślnym jest `21k.map`)
`-o filename`	Plik wyjściowy o nazwie `filename`
`-nomem`	Nie uruchamiaj inicjalizatora pamięci
`-runhdr filename`	Specyfikacja alternatywnego pliku nagłówkowego
`-w`	Zablokuj wszystkie wiadomości ostrzeżeń
`-Wall`	Połącz wszystkie pstrzeżenia
`-Wimplicit`	Ostrzegaj kiedy funkcja jest niejawnie zadeklarowana
`-Wreturn-type`	Ostrzegaj kiedy funkcja domyślnie zwraca typ `int`
`-Wunused`	Ostrzegaj kiedy automatyczna zmienna nie jest użyta
`-Wswitch`	Ostrzegaj kiedy przełącznik nie używa wszystkich typów wyliczeniowych
`-Wcomment`	Ostrzegaj kiedy komentarz zawiera ciąg /*
`-Wfloat-convert`	Ostrzegaj kiedy liczba `float` jest niejawnie zamieniana na `double`, lub `double` jest niejawnie zamieniana na `float`

4. Wydruk programu przykładowego

Program przykładowy stanowić ma bazę, na podstawie której zostanie przeprowadzone ćwiczenie laboratoryjne. Konieczna jest ogólne zaznajomienie się ze strukturą programu bez wnikania w szczegóły np. zawartości rejestrów sterujących itp.

```
/*********************************************************/
/*
Odbiór danych z AD1847 poprzez porty szeregowe a następnie transmisja
danych spowrotem na porty szeregowe na wyjście AD1847. Flaga 2 (i
odpowiadający LED) mrugają okresowo. Pasmo wejściowe i współczynnik
próbkowania AD1847 może być zmieniany w trakcie wykonywania programu
poprzez zmianę wartości zmiennych ordered_rate i ordered_gain.

Poniższe atrybuty pracy AD1847 mogą być zmieniane podczas wykonywania
Programu: pasmo wejściowe (16 poziomów), i współczynnik próbkowania (14 częstosliwości).

Formuła pasma wejściowego: poziom * 1.5dB (poziom domyślny = 0 --> 0dB)

Współczynniki próbkowania (w kHz):
        ( 0) 8              ( 1) 5.5125 (default)
        ( 2) 16             ( 2) 11.025
        ( 4) 27.42857       ( 5) 18.9
        ( 6) 32             ( 7) 12.05
        ( 8) N/A            ( 9) 37.8
        (10) N/A            (11) 44.1
        (12) 48             (13) 33.075
        (14) 9.6            (15) 6.615
*/

/*********************************************************/

/* ADSP-2106x definicje bitów rejestru */

#include <def21060.h>
#include <21060.h>
#include <signal.h>
#include <sport.h>
#include <macros.h>

/*********************************************************/

        /* DMA definicje bitów wskaźnika Chain */
#define CP_PCI 0x20000          /* bity przerwania programu kontrolnego */
#define CP_MAF 0x1ffff          /* ważne bity pola adresu pamięci */

#define SetIOP(addr, val) (* (int *) addr) = (val)
#define GetIOP(addr) (* (int *) addr)

/*********************************************************/

#define SZ_regs_1847 16
int regs_1847[SZ_regs_1847] = {
        /* Zauważ, że bit MCE jest ustawiony przez cały program
        w celu zatrzymania przedwczesnej autokalibracji. */
        0xc000,         /* index 0 – kontrola lewego wejścia */
        0xc100,         /* index 1 – kontrola prwego wejścia */
        0xc280,         /* index 2 – kontrola lewego wejścia aux 1 */
        0xc380,         /* index 3 – kontrola prawego wejścia aux 1 */
        0xc480,         /* index 4 – kontrola lewego wejścia aux 2 */
        0xc580,         /* index 5 – kontrola prawego wejścia aux 2 */
        0xc600,         /* index 6 – kontrola lewgo dac */
        0xc700,         /* index 7 – kontrola prawego dac */
        0xc85c,         /* index 8 – format danych */
        0xc909,         /* index 9 – konfiguracja interfejsu */
        0xca00,         /* index 10 – kontrola pinów */
        0xcb00,         * index 11 – bez rejestru */
        0xcc40,         /* index 12 – pozostałe informacje */
        0xcd00,         /* index 13 – kontrola digital mix control */
        0xce00,         /* index 14 – bez rejestru */
```

```
        0x8f00};            /* index 15 – bez rejestru */
int rx_buf[3];                          /* bufor odbioru */
int tx_buf[3] = {0xcc40, 0, 0};         /* bufor transmisji */

/* DMA chaining Transfer Control Blocks */
typedef struct {
        unsigned lpath3;    /* dla multiprocesorów sieciowych */
        unsigned lpath2;    /* dla multiprocesorów sieciowych */
        unsigned lpath1;    /* dla multiprocesorów sieciowych */
        unsigned db;        /* Rejestr ogólnego przeznaczenia */
        unsigned gp;        /* Rejestr ogólnego przeznaczenia */
        unsigned** cp;      /* Wskaźnik łańcucha do następnego TCB */
        unsigned c;         /* Rejestr liczący */
        int im;             /* Rejestr indeksowy modyfikujący */
        unsigned * ii;      /* Rejestr indeksowy */
} _tcb;

_tcb rx_tcb = {0, 0, 0, 0, 0, 0, 3, 1, 0};      /* odebrane tcb */
_tcb tx_tcb = {0, 0, 0, 0, 0, 0, 3, 1, 0};       /* transmitowane tcb */

int cmd_blk[8];                 /* blok poleceń */

static int xmit_count;
static int * xmit_ptr;

static int current_rate;
static int current_gain;

/*********************************************************/

// Zmienne, które mogą być ustawione przez użytkownika do kontroli kodowania.

volatile static int ordered_gain;
volatile static int ordered_rate;


/*********************************************************/
/* */
/* */
/* */
/*********************************************************/
void set_input_gain( void )
{
   // Jeśli trwa polecenie transmisji, czekaj aż skończy.
   if( xmit_count != 0 )
      return;

   //ograniczenie pasma wejściowego na określoną wartość.
   ordered_gain &= 0x0f;

   // Przenieś nową wartość pasma do poleceń rejestru 1847.
   regs_1847[0] = (regs_1847[0] & ~0x0f) | ordered_gain;
   regs_1847[1] = (regs_1847[1] & ~0x0f) | ordered_gain;

   // Pobierz polecenia 1847 w celu ustawienie pasma wejściowego i przełączyć
   // polecenie na blok poleceń do transmisji.
   cmd_blk[0] = regs_1847[0];
   cmd_blk[1] = regs_1847[1];
   cmd_blk[2] = regs_1847[15];

   // Ustaw wskaźnik i licznik do poleceń transmisji.
   xmit_ptr = cmd_blk;
   xmit_count = 3;

   // Aktualizacja aktualnej wartości do wartości nakaznej.
   current_gain = ordered_gain;

   return;
}

/*********************************************************/
/* */
/* */
/* */
```

```
/*********************************************************/
void set_samplerate( void )
{
   // Jeśli trwa polecenie transmisji, czekaj aż skończy.
   if( xmit_count != 0 )
      return;

   // Ustawienie współczynnika próbkowania na określoną wartość
   ordered_rate &= 0x0f;
   if( ordered_rate == 8 )
      ordered_rate = 9;
   if( ordered_rate == 10 )
      ordered_rate = 11;

   // Przenieś nowy współczynnik próbkowania do poleceń rejestru 1847.
   regs_1847[8] = (regs_1847[8] & ~0x0f) | ordered_rate;

   // Pobierz polecenie 1847 w celu ustawienia wpółczynnika próbkowania i przełączenia
   // polecenia do bloku poleceń transmisji.
   cmd_blk[0] = regs_1847[8];
   cmd_blk[1] = regs_1847[15];

   // Ustaw wskaźnik i licznik do poleceń transmisji.
   xmit_ptr = cmd_blk;
   xmit_count = 2;

   // Aktualizacja aktualnej wartości do wartości nakaznej.
   current_rate = ordered_rate;

   return;
}

/*********************************************************/
/*                                                       */
/* Okresowe przerwanie zegara                            */
/*                                                       */
/*********************************************************/
void timer_lo_prior( int sig_num )
{
   sig_num=sig_num;

   // Ustaw flagę 2 LED.
   set_flag(SET_FLAG2, TGL_FLAG);
}

/*********************************************************/
/*                                                       */
/* szeregowy port transmisji DMA gotowy                  */
/*                                                       */
/*********************************************************/
void spt0_asserted( int sig_num )
{
   // Sprawdź czy pozosotały polecenia do transmisji.
   if( xmit_count )
   {
     // Jeśli tak, przenieś polecenie do bufora transmisji i akutalizuj licznik.
      tx_buf[0] = *xmit_ptr++;
      xmit_count--;
   }
}

/*********************************************************/
/*                                                       */
/* szeregowy port odbioru DMA gotowy                     */
/*                                                       */
/*********************************************************/
void spr0_asserted( int sig_num )
{
   // Kopiuj dane buforów odbioru do danych buforów transmisji.
   tx_buf[1] = rx_buf[1];
   tx_buf[2] = rx_buf[2];
}
```

```
/*********************************************************/
/*                                                       */
/*                                                       */
/*                                                       */
/*********************************************************/
void setup_sports ( void )
{
   /* Konfiguracja szeregowego portu SPORT() SHARC */

   /* Ustawienia komunikacji wielokanałowej */
   sport0_iop.mtcs = 0x00070007; /* transmisja na słowa 0,1,2,16,17,18 */
   sport0_iop.mrcs = 0x00070007; /* odbiór na słowa 0,1,2,16,17,18 */
   sport0_iop.mtccs = 0x00000000; /* bez kompansji przy transmisji */
   sport0_iop.mrccs = 0x00000000; /*bez kompansji przy odbiorze */


   /* REJESTR KONTROLI TRANSMISJI */
   /* STCTL0 <= 0x001c00f2 */
   /* Alternatywnym (i bardziej wydajnym) sposobem wykonania tego będzie */
   /* zapisanie rejestru 32-bitowego całego za jednym razem poleceniem: */
   /* SetIOP(STCTL0, 0x001c00f2); */
   /* Ale następujący sposób jest bardziej opisowy.. */
   sport0_iop.txc.mdf = 1; /* opóźnienie ramki wielokanałowej (MFD) */
   sport0_iop.txc.schen = 1; /* aktywny łańcuch Tx DMA */
   sport0_iop.txc.sden = 1; /* aktywny Tx DMA */
   sport0_iop.txc.lafs = 0; /* Opóźniony TFS (alternatyny) */
   sport0_iop.txc.ltfs = 0; /* Aktyny niski TFS */

   sport0_iop.txc.ditfs = 0; /* niezależny od danych TFS */
   sport0_iop.txc.itfs = 0; /* Wewnętrznie wygenerowany TFS */
   sport0_iop.txc.tfsr = 0; /* Wymagany TFS */
   sport0_iop.txc.ckre = 0; /* Dane and FS na narastające zbocze zegara */
   sport0_iop.txc.gclk = 0; /* Aktyny zegar tylko w trakcie transmisji*/
   sport0_iop.txc.iclk = 0; /* wewnętrznie wygenerowany zegar Tx */
   sport0_iop.txc.pack = 0; /* Rozpakuj słowa 32b do dwóch 16b */

   sport0_iop.txc.slen = 15; /* Długość słowa danych minus jeden */
   sport0_iop.txc.sendn = 0; /* słowo danych endian 1 = pierwszy LSB */
   sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
            /* specyfikator typu danych */
   sport0_iop.txc.spen = 0; /* Aktywny (wyczyszczony dla operacji MC) */

   /* rejestr kontroli odbioru */
   /* SRCTL0 <= 0x1f8c20f2 */
   sport0_iop.rxc.nch = 31; /* wielokanałowa liczba kanałów - 1 */
   sport0_iop.rxc.mce = 1; /* aktywne wielokanałowe */
   sport0_iop.rxc.spl = 0; /* konfiguracja pętli zwortnej (test) */
   sport0_iop.rxc.d2dma = 0; /* aktywna 2-wymiarowa tablica DMA */
   sport0_iop.rxc.schen = 1; /* aktywny łańcuch Rx DMA */
   sport0_iop.rxc.sden = 1; /* aktywny Rx DMA */
   sport0_iop.rxc.lafs = 0; /* opóźniony RFS (alternatywnie) */
   sport0_iop.rxc.ltfs = 0; /* aktywny niski RFS */
   sport0_iop.rxc.irfs = 0; /* wewnętrznie wygenerowany RFS */
   sport0_iop.rxc.rfsr = 1; /* wymagany RFS */
   sport0_iop.rxc.ckre = 0; /* Dane i FS na narastające zbocze zegara */
   sport0_iop.rxc.gclk = 0; /* zegar aktywny tylko w trakcie transmisji*/
   sport0_iop.rxc.iclk = 0; /* wewnętrznie wygenerowany zegar Rx */
   sport0_iop.rxc.pack = 0; /* spakuj dwa słowa 16b do słowa 32b */

   sport0_iop.rxc.slen = 15; /* dłougość słowa danych minus jeden */
   sport0_iop.rxc.sendn = 0; /* słowo danych endian 1 = pierwszy LSB */
   sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
            /* specyfikator typu danych */
   sport0_iop.rxc.spen = 0; /* aktywny (wyczyszczony dla operacji MC) */

   /* aktywne sport0 xmit & rcv irqs (aktywny DMA) */
   interrupt(SIG_SPR0I, spr0_asserted);
   interrupt(SIG_SPT0I, spt0_asserted);

   /* Ustawienie Bloku Kontroli Transmisji dla łańcuchowego DMA */
   tx_tcb.ii = tx_buf; /* adres buforu źródłowego DMA */
   tx_tcb.cp = &tx_tcb.ii; /* definicja wskaźnika do następnego TCB (wskazanie na siebie) */
   SetIOP(CP2, (((int)&tx_tcb.ii) & CP_MAF) | CP_PCI);
            /* definicja wskaźnika do aktualnego TCB (odrzucenie DMA) */
```

```
            /* (SPORT0 transmituje używając kanału 2 DMA) */

   /* Ustawienie Bloku Kontroli Odbioru dla łańcuchowego DMA */
   rx_tcb.ii = rx_buf; /* adres bufora przeznaczenia DMA */
   rx_tcb.cp = &rx_tcb.ii; /* definicja wskaźnika do następnego TCB (wskazanie na siebie) */
   SetIOP(CP0, (((int)&rx_tcb.ii) & CP_MAF) | CP_PCI);
            /* definicja wskaźnika do aktualnego TCB (odrzucenie DMA) */
            /* (SPORT0 odbiera używając kanału 0 DMA ) */
}/

************************************************************/
/* */
/* */
/* */
/************************************************************/
void send_1847_config_cmds( void )
{
   // Ustawienie wskaźnika i licznika do polecenia transmisji.
   xmit_ptr = regs_1847;
   xmit_count = SZ_regs_1847;

   // Czekaj, aż wszystkie polecenia zostaną wysłane.
   while( xmit_count )
      idle();

   // czekaj na autowywołanie startu przez AD1847.
   while( !(rx_buf[0] & 0x0002) )
      idle();

   // czekaj na autowywołanie zakończenia przez AD1847.
   while( rx_buf[0] & 0x0002 )
      idle();

   return;
}

/************************************************************/
/* */
/* */
/* */
/************************************************************/
void init_21k( void )
{
   // dezaktywuj zegar i ustaw współczynnik na 4 Hz.
   timer_off();
   timer_set( 10000000, 10000000 );

   // Inicjalizacja wskaźnika i licznika do polecenia transmisji.
   xmit_count = 0;
   xmit_ptr = regs_1847;
/
   / umożliwij zagnieżdżanie przerwań.
   asm( "#include <def21060.h>" );
   asm( "bit set mode1 NESTM;" );

   // Umożliwij przerwania zegaowe (niskiego priorytetu).
   interrupt( SIG_TMZ, timer_lo_prior );

   // wyłącz flagę LED'ów.
   set_flag( SET_FLAG2, SET_FLAG );

   return;
}

/************************************************************/
/* */
/* */
/* */
/************************************************************/
void main ( void )
{
   int x;

   // Inicjalizacja niektórych rejestrów SHARC.
```

```
    init_21k();

    // Reset CODEC'a
    set_flag( SET_FLAG0, CLR_FLAG ); /* Przenieś CODEC'a do RESET */
    for( x=0 ; x<0xffff ; x++ ) /* Zatrzymaj CODEC w RESET */
    ;
    set_flag( SET_FLAG0, SET_FLAG ); /* zwolnij CODEC z RESET */

    // Konfiguracja szeregowego portu SHARC.
    setup_sports();

    // Wyślij polecenia ustawiania do CODEC.
    send_1847_config_cmds();

    // Włącz wszystkie LED'y.
    set_flag(SET_FLAG2, CLR_FLAG);

    // włącz zegar.
    timer_on();

    // niekończąca się pętla.
    for(;;)
    {
       // sprawdź czy użytkownik nakazał zmianę pasma wejściowego.
       if( current_gain != ordered_gain )
           set_input_gain();

       // sprawdź czy użytkownik nakazał zmianę współczynnika próbkowania.
       if( current_rate != ordered_rate )
           set_samplerate();
       idle();
    };
}/
************************************************************/

// End of file tt.c .

/************************************************************/
```