

OPRACOWANIE - KOŁOKWIUM z DSP

Opracował na podstawie materiałów do egzaminu - Paweł Wańtowski

Uzupełnienia i korekty - dr Krzysztof Kardach 2010-01-07 13:41

1. Tabelki

Tabelka „arytmetyczna”

w.		DEC	HEX	BIN	12-bitowy akumulator
	1	2	3	4	5
1	WA		0x	B	
2	WB		0x	B	
3	WC=WA+WB		0x	B	
4	WC=WA-WB		0x	B	
5	WC=WA*WB		0x	B	

Rozwiązanie dotyczy przykładowej tabelki z grupy A z 2004 roku. Dotyczą jej pierwsze 4 zadania na kolokwium:

1. Dla liczb **A=0,75** i **B=-0,125** uzupełnić tabelę w kolumnach 2,3,4 zakładając notację **U2** i format **I1Q5**.

2. Zakładając prace procesora na słowie **6-bitowym** i kodowanie **U2, I1Q5**, oraz akumulator **12-bitowy** i włączony **SXM** uzupełnij tabelę w kolumnie 5 wykonując odpowiednie rozkazy:

dla wiersza 1 **LD#WA,2,A**

dla wiersza 2 **LD#WB,A**

Wyniki należy zapisać w notacji binarnej.

3. Wpisz do tabelki zadania 1 w kolumnie 5 (wiersze 3,4 i 5) binarną zawartość akumulatora po wykonaniu odpowiednio operacji zapisanych w kolumnie 1.

4. Jak zmieni się wynik operacji w wierszu 5, jeśli w naszym ćwiczebnym procesorze będzie ustawiony odpowiednik bitu **FRCT** (Fractional).

ad.1. UWAGA! Wyniki operacji należy zamieścić zgodnie z wymaganymi formatem i notacją.

Kodujemy **A** i **B** w kodzie **U2** (jeśli ktoś do tej pory nie wie, jak się to robi, to niech zajrzy tutaj <http://www.i-lo.tarnow.pl/edu/inf/alg/num/pages/018.php>). Inne przydatne materiały odnośnie systemów liczbowych i ich kodowania można znaleźć na poniższych stronach:

<http://network.page.com.pl/materiały/Konwersje.htm>

<http://www.i-lo.tarnow.pl/edu/inf/alg/num/pages/014.php>

Procesor pracuje na słowie **6-bitowym**, zaś format **I1Q5** oznacza, że z tych 6 bitów jeden zostanie przeznaczony na zakodowanie części całkowitej liczby, zaś pozostałe 5 na zakodowanie części ułamkowej. Tak więc:

$$WA = 0,75_d = 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} = 01\ 1000_b$$

$$WB = -0,125_d$$

Aby uzyskać **WB**, kodujemy **0,125** jak wyżej (a), następnie negujemy wszystkie bity (b) i dodajemy **1** do pozycji najmłodszego bitu (c):

(a)

$$0,125_d = 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} = 000100_b$$

(b)

$$\sim 0,125_d = 1 \dots 1111 \ 1011_b$$

(c)

$$\begin{array}{r} 1 \dots 111 \ 1011 \\ + \quad \quad \quad 1 \\ \hline = 1 \dots 111 \ 1100 \end{array}$$

UWAGA! Trzeba pamiętać o „1-kach z przodu”) dla utrzymania wartości i znaku liczby i wyników wszelkich operacji !!! (SXM)

Zatem $WB = -0,125_d = 1 \dots 11 \ 1100_b$. i w 6-cio bitowym rejestrze zmieszczą się tylko wytłuszczone bity.

Teraz można wykonać następujące działania:

- **dodawanie**

$$\begin{array}{r} \quad \quad 01 \ 1000 \\ + 1 \dots 1111 \ 1100 \quad \text{(to liczba ujemna!)} \\ \hline = 0 \dots 0001 \ 0100 \end{array}$$

Jak widać wynik jest dodatni. „0-ra” z przodu (tu zaznaczone na żółto) dla dodatnich a „1-ki” dla ujemnych wykraczające poza rejestr trzeba pominąć, bo nie mieszczą się w rejestrze wyniku, czyli prawidłowo jest:

$$WA + WB = 01 \ 0100_b = 0,625_d$$

Analogicznie jest dla odejmowania (uwzględniamy tylko 6 najmłodszych bitów wyniku) i mnożenia (uwzględniamy tym razem 12 najmłodszych bitów wyniku).

- **odejmowanie** - jeżeli ktoś nie lubi bawić się w pożyczki i przenoszenie, alternatywnie zamiast wykonywać odejmowanie $A-B$ można wykonać dodawanie $A+(\sim B)+1$, otrzymany wynik będzie taki sam (pamiętać o SXM!):

$$\begin{array}{r} \quad \quad 01 \ 1000 \\ + \quad 00 \ 0011 \\ + \quad \quad \quad 1 \\ \hline \quad 01 \ 1100 \end{array}$$

$$WA - WB = 01 \ 1100_b = 0,875_d$$

UWAGA: jeżeli w wyniku odejmowania dziesiętnego (dla kodowania I1Q5 U2) otrzymana zostanie liczba mniejsza od **-1**, to wykraczamy poza zakres reprezentacji! Dzieje się tak z powodu przekroczenia zakresu dozwolonych wartości dla formatu **I1Q5 U2** (pozwala on na minimalną wartość równą **-1**). **To jest celowy haczyk i należy napisać o tym w tabelce lub pod nią.**

Np. dla $-1,125_d$ i reprezentacji jak wyżej na 6-ciu bitach możemy posłużyć się kalkulatorem z konwersją DEC \leftrightarrow HEX. Zatem,

$$-1,125_d * 2^5 = -36_d = F \dots FDC_h = 1 \dots 1101 \ 1100_b$$

Zatem z zamieszczoną uwagą o wykroczeniu poza zakres reprezentacji prawidłowa odpowiedź na pytanie o zawartość rejestru (takiego 6-cio bitowego!) po wykonaniu operacji to $01 \ 1100_b$ oraz DC_h . Jeśli zaś pytanie pada o wynik kodowania **I1Q5 U2** wówczas odpowiedź jest krótka – nie da się zakodować takiej liczby.

- **mnożenie** – by zmieścić wynik mnożenia dwóch liczb 6-cio bitowych **I1Q5 U2** potrzebny jest rejestr dwukrotnie większy. Dla uniknięcia błędów należy przygotować do operacji binarnej oba czynniki stosownie je rozszerzając znakowo co najmniej do 12 bitów poprzez dostawienie **6 starszych bitów** wypełnionych wartością taką, jaką ma najstarszy bit wagowy ze **znakiem** danej liczby w **U2**:

WA = 011000_b → 000000 011000_b

WB = 111100_b → 111111 111100_b

Po dokonaniu tego można pomnożyć **WA * WB** (tu oczywiście wygodniej będzie wymnożyć **WB * WA**):

```

      1...11 1111 1111 1100
*      0000 0001 1000
-----
      0000 0000 0000
      0 0000 0000 000
      00 0000 0000 00
      1...1.....1 1111 1111 1110 0
      1...1.....1 1111 1111 1100
      0 0000 0000 000
      ... ..
  
```

1111...1.. 1111 **1111 1010 0000**

WA * WB = 11 1101_b = -0,09375

Wynikiem mnożenia, reprezentowanym zgodnie z przyjętą tu notacją na 6-ciu bitach, który należy zamieścić w tabelce w wierszu 5 kolumnie 4 są bity na pozycjach zaznaczonych powyżej na czerwono. (proszę zwrócić uwagę na wynikającą z położenia przecinka dodatkową 1-kę „z przodu” i fakt wyboru tylko starszych 6-ciu bitów wyniku)

Natomiast wynikiem, jaki trafi do 12-to bitowego akumulatora bez korekcyjnego przesunięcia, będzie podkreślony fragment wyniku mnożenia.

Wynik ten można również otrzymać inaczej;

-0,125 * 0,75 = -0,09375

-0,09375 * 2⁵ = -3_d = F ... FFD_h = 1 ... 1111 1101_b

Mając już wyniki dziesiętnie i binarnie, możemy łatwo przekodować BIN na HEX. Są to dwie różne drogi rozwiązywania, obie prawidłowe i muszą dać takie same wartości liczb!.

Pierwsza: ogranicza się do 6-ciu bitów i tłumaczy dokładnie co widać;

WA = 011000_b = 01 1000_b

1 8_h

czyli po prostu przekształcamy to co widać. Młodsza „czwórka” bitów na liczbę HEX z zakresu 0-F i starszą dwójkę bitów też na liczbę HEX tyle że z zakresu 0-3:

01 1000₂ = 18_h

I tutaj nie ma problemów dla liczb z zakresu reprezentacji, dodatnich i ujemnych.

Dla przykładowej wartości; **-0,09375_d = 11 1101_b = 3D_h** stąd wartość **-1 + 0,5 + 0,25 + 0,125 + 0,03125 = -0,09375**

Ale dla liczb przekraczających zakres reprezentacji potrzebna jest stosowna uwaga o której napisano wcześniej i zapis zależy od sformułowania pytania.

Zatem dla; $-1,125_d = 1101\ 1100_b$
w 6-cio bitowym rejestrze będzie miejsce tylko dla $1C_h$ no i niezbędna jest **adnotacja o przekroczeniu zakresu reprezentacji**
Zaś wartość będzie $-4 + 2 + 0,5 + 0,25 + 0,125 = -1,125$
(to żółte to dzięki adnotacji i tym dwóm „żółtym” bitom).

Druga: nie pomija starszych bitów a po prostu je uwzględnia. Dla dodatnich zatem uzupełnia zerami (przykład dla **WA**):

$$WA = 01\ 1000_b = 0001\ 1000_b$$

$\quad\quad\quad 1\quad\quad 8_h$

czyli po prostu przekształcamy każdą „czwórkę” binarną na liczbę w HEX:

$$0001\ 1000_b = 18_h$$

Dla ujemnych uzupełniamy zgodnie z zasadami rozszerzenia znakowego 1-kami.

Dla przykładowej wartości; $-0,09375_d = 1111\ 1101_b = FD_h$
stąd wartość $-4 + 2 + 1 + 0,5 + 0,25 + 0,125 + 0,03125 = -0,09375$

Dla liczb przekraczających zakres reprezentacji musimy uwzględnić uzupełnienie zgodnie z wynikiem konwersji.

Zatem dla; $-1,125_d = 1101\ 1100_b = DC_h$
i tutaj wartość $-4 + 2 + 0,5 + 0,25 + 0,125 = -1,125$
(to zielone to z uzupełnienia tymi dwoma „zielonymi” bitami).

Tutaj jeszcze jedna drobna uwaga. Przy przekroczeniu reprezentacji pojawia się jeszcze subtelność – jak sformułowane jest pytanie. Jeśli domaga się ono np. podania zawartości rejestru 6-cio bitowego po operacji na operandach U2 I1Q5 wówczas trzeba zawrzeć to co zmieści się w tym rejestrze (czarne bity w ostatnim przykładzie, ale koniecznie z adnotacją, że to tylko kawałek liczby). Jeśli zaś pytanie będzie wymagało podania wyniku zakodowanego U2 I1Q5, a wynik przekracza zakres reprezentacji no to nie ma co podać. Pole powinno zostać puste ale opatrzone adnotacją, że wynik poza zakresem reprezentacji.

I tutaj jedna ważna uwaga. Przy operacjach arytmetycznych nie należy oglądać się na ustawiony lub nie bit SXM. TUTAJ ROZSZERZENIE ZNAKOWE MUSI CAŁY CZAS DZIAŁAĆ BO INACZEJ NIE ZACHOWALI BYŚMY WARTOŚCI !!!

ad.2. Bit SXM wpływa na to, jak uzupełniane są starsze bity w akumulatorze gdy ładujemy do niego liczby „krótsze” - reprezentowane na mniejszej liczbie bitów niż długość akumulatora:

- jeżeli bit SXM jest **włączony**, to uzupełniamy w zależności od znaku, tj. gdy liczba jest dodatnia uzupełniamy zerami, a gdy ujemna - jedynkami
- gdy bit SXM jest **wyłączony**, to zawsze uzupełniamy zerami

Mamy wykonać następujące rozkazy:

LD#WA,2,A ← załaduj **WA** do akumulatora i przesun o **2** w lewo ($\ast 2^{SHIFT}$)

LD#WB,A ← załaduj **WB** do akumulatora

Dla **WA** będzie to wyglądać następująco (poglądowo, bo w rzeczywistości rzecz jasna nie byłoby „pustych miejsc”):

- uzupełniamy **starsze bity** w zależności od **znaku** (czyli tutaj zerami),
- ładujemy uzupełnione **WA** do akumulatora: $_ \text{0000 } \text{0001 } \text{1000}$
- przesuwamy o 2 pozycje w lewo: $_ _ _ _ \text{0000 } \text{0110 } \text{00_}$
- a na **najmłodszych** **zawsze** wstawiamy zera: $\text{0000 } \text{0110 } \text{0000}$

Postępując analogicznie (oczywiście tym razem bez przesuwania), dla **WB** mamy:

$\text{1111 } \text{1111 } \text{1100}$

Można jeszcze rozpatrzeć podobny przykład ale z przesunięciem w drugą stronę np;

LD#WB,-4,A ← załaduj **WB** do akumulatora i przesun o 4 w prawo ($\ast 2^{\text{SHIFT}}$)

- uzupełniamy **starsze bity** w zależności od **znaku** (czyli tutaj zerami),
- ładujemy uzupełnione **WA** do akumulatora: $_ \text{1111 } \text{1111 } \text{1100}$
- teraz przesuwamy o 4 pozycje w prawo: $_ _ _ _ \text{1111 } \text{1111 } \text{1111 } \text{1000}$
- a wychodzące poza rejestr **najmłodsze bity obcinamy**:
 $\text{1111 } \text{1111 } \text{1111}$

Warto tutaj zrobić jeszcze jedną uwagę. Najpierw rozszerzamy liczbę a potem przesuwamy i ładujemy.

ad.3. Dla dodawania i odejmowania postępujemy podobnie, jak w punkcie poprzednim, pamiętając o włączonym bicie SXM. Inaczej mówiąc ładujemy wyniki operacji do akumulatora i odpowiednio uzupełniamy zerami lub jedynkami w zależności od znaku. W przypadku mnożenia wynik jest dłuższy i po prostu przepisujemy do akumulatora najmłodsze 12 bitów otrzymane w wyniku pisemnego mnożenia binarnego, czyli w naszym wypadku tę podkreśloną część wyniku ze strony 3:

$\text{1111 } \text{1010 } \text{0000}$

ad.4. Bit FRCT powoduje skasowanie „nadmiarowego” znaku liczby, co dzieje się poprzez przesunięcie liczby o 1 w lewo, czyli jeśli mamy w akumulatorze wynik mnożenia:

$\text{1111 } \text{1010 } \text{0000}$

to po przesunięciu o jeden w lewo mamy:

$\text{1111 } \text{0100 } \text{0000}$

Powstała wskutek przesunięcia „pusta” pozycję na najmłodszym bicie **zawsze** uzupełniamy **zerem**.

Jeszcze uwaga dla mających kłopot z rozumieniem określenia „nadmiarowego znaku”. Proponuję policzyć pozycje po przecinku obu czynników, następnie określić położenie przecinka w wyniku (dokładnie tak, jak to się robi przy mnożeniu pisemnym liczb dziesiętnych) i natychmiast ujawni się dodatkowe miejsce przed przecinkiem i wynika z tego konieczność korekcyjnego przesunięcia w lewo realizowanego za sprawą bitu FRCT.

I to już wszystko, po prawidłowym uzupełnieniu tabelka „arytmetyczna” powinna zawierać następujące wartości:

A = 0,75, B = -0,125

w.		DEC	HEX	BIN	ACC 12-bitowy akumulator
	1	2	3	4	5
1	WA	0,75	0x18	01 1000 B	0000 0110 0000 B
2	WB	-0,125	0x3C	11 1100 B	1111 1111 1100 B
3	WC=WA+WB	0,625	0x14	01 0100 B	0000 0001 0100 B

4	WC=WA-WB	0,875	0x1C	01 1100 B	0000 0001 1100 B
5	WC=WA*WB	-0,09375	0x3D	11 1101 B	1111 1010 0000 B

Poniżej 4 inne tabelki z grup od roku 2004+. z rozkazami do wykonania w kolumnie 5

dla wiersza 1	LD #WA,3,ACC
dla wiersza 2	LD #WB,-3,ACC
dla wiersza 3	ADD #WA,#WB,ACC
dla wiersza 4	SUB #WA,#WB,ACC
dla wiersza 5	MPY #WA,#WB,ACC

A = -0,875, B = 0,5

w.		DEC	HEX	BIN	ACC 12-bitowy akumulator
	1	2	3	4	5
1	WA	-0,875	0x24	10 0100 B	1111 1001 0000 B
2	WB	0,5	0x10	01 0000 B	0000 0001 0000 B
3	WC=WA+WB	-0,375	0x34	11 0100 B	1111 1111 0100 B
4	WC=WA-WB	-1,375*	0xD4; 0x14*	01 0100* B	1111 1101 0100 B
5	WC=WA*WB	-0,4375	0x32	11 0010 B	1110 0100 0000 B

*przekroczenie zakresu reprezentacji poniżej granicznej -1,;

UWAGA, licząc binarnie trzeba pamiętać o rozszerzeniu znakowym (uzupełnieniu „jedynek” z przodu) inaczej otrzymamy **BŁĘDNY WYNIK 0,625**

*reprezentuje tylko zawartość 6-ciu najmłodszych bitów po obcięciu starszych

A = 0,125, B = -0,75

w.		DEC	HEX	BIN	ACC 12-bitowy akumulator
	1	2	3	4	5
1	WA	0,125	0x04	000100B	0000 0001 0000 B
2	WB	-0,75	0x28	101000B	1111 1110 1000 B
3	WC=WA+WB	-0,625	0x2C	101100B	1111 1110 1100 B
4	WC=WA-WB	0,875	0x1C	011100B	0000 0001 1100 B
5	WC=WA*WB	-0,09375	0x3D	111101B	1111 1010 0000 B

A = -0,25, B = 0,375

w.		DEC	HEX	BIN	ACC 12-bitowy akumulator
	1	2	3	4	5
1	WA	-0,25	0x38	11 1000 B	1111 1110 0000 B
2	WB	0,375	0x0C	00 1100 B	0000 0000 1100 B
3	WC=WA+WB	0,125	0x04	00 0100 B	0000 0000 0100 B
4	WC=WA-WB	-0,625	0x2C	10 1100 B	1111 1110 1100 B
5	WC=WA*WB	-0,09375	0x3D	11 1101 B	1111 1010 0000 B

A = 0,75, B = -0,25

w.		DEC	HEX	BIN	ACC 12-bitowy akumulator
	1	2	3	4	5
1	WA	0,75	0x18	01 1000 B	0000 1100 0000 B
2	WB	-0,25	0x38	11 1000 B	1111 1111 1111 B
3	WC=WA+WB	0,5	0x10	01 0000 B	0000 0001 0000 B
4	WC=WA-WB	1*			0000 0010 0000 B
5	WC=WA*WB	-0,1875	0x3A	11 1010 B	1111 0100 0000 B

*przekroczenie zakresu reprezentacji; Tej liczby nie da się przedstawić na 6-ciu bitach.

ALE w akumulatorze o podwójnej długości da się przedstawić bo ten 6-ty bit nie będzie już skrajnym bitem z informacją o znaku jak jest to w „krótkim rejestrze”!

Tabela „rozkazowa”

Sprowadza się do odpowiedniej interpretacji rozkazów – (pod tabelką przykładowa instrukcja interpretacji krok po kroku). Górna tabela poza zadaniem wartości bitów ustalających sposób interpretacji rozkazów zawiera dane o zawartości fragmentów pamięci danych. Zwracam uwagę na wskaźnik strony DP, który może być podany a czasami trzeba go określić w oparciu o adres pamięci w tabelce. **Wszystkie liczby wstawione do tabelki podczas rozwiązywania są w HEX, a nie w DEC!**

Dane:	DP=0		DP=2		DP=6	
CPL=0	60	60	200	120	300	130
CMPT=0	61	40	201	70	301	80
Adr./Dane HEX	62		202	20	302	100
	Adres	Wartość	Adres	Wartość	Adres	Wartość

	Program	A	B	DP	AR0	AR1	AR2
1	LD #0,DP			0			
2	STM #2,AR0				2		
3	STM #200h,AR1					200	
4	STM #300h,AR2						300
5	LD @0x61,A	40					
6	ADD *AR1+,A	160				201	
7	SUB @60h,A,B		100				
8	ADD *AR1+,B,A	170				202	
9	LD #6,DP			6			
10	ADD @2,A	270					
11	ADD *AR2+,A	3A0					301
12	SUB *AR2+,A	320					302
13	SUB #64,A	2E0					
14	ADD *AR2-0,A,B		3E0				300
15	SUB *AR2,B,A	2B0					
16	STM #160,AR0				A0		
17	ADD *AR1-0,A,A	2D0				162	
18	STL A,*AR1-					161	

Przykładowa interpretacja kolejnych wierszy tabeli:

1. Załaduj liczbę binarną 0 do DP. (włączamy w ten sposób 0-wą stronę pamięci danych)
2. Załaduj liczbę binarną 2 do AR0.
3. Załaduj liczbę 200h (albo inaczej 0x200) do AR1.
4. Załaduj liczbę 0x300 do AR2.
5. Załaduj wartość komórki o adresie 0x61 do A (patrz nad tabelką).
6. Dodaj do akumulatora A zawartość komórki pamięci danych o adresie równym zawartości AR1 (zawartość AR1 to 0x200, a odpowiadająca temu adresowi zawartość komórki to 0x120 – patrz nad tabelką), dodaj do tego zawartość A, zapisz wynik w A i potem inkrementuj zawartość AR1.
7. Odejmij od akumulatora A zawartość komórki pamięci danych o adresie 0x60 (adres 0x60 zawartość równa 0x60 – patrz nad tabelką) i wynik zapisz w B.
8. Dodaj do akumulatora B zawartość komórki pamięci danych o adresie równym wartości AR1 (zawartość AR1 to 0x201, a odpowiadająca temu adresowi wartość komórki to 0x70 – patrz nad tabelką), a zapisz wynik w A, potem inkrementuj zawartość AR1.

9. Załaduj liczbę 6 do DP. (włączamy w ten sposób 6-tą stronę pamięci danych)
10. Mając DP=6 (zrobiliśmy to w wierszu 9) patrzymy do tabelki nad główną tabelą i szukamy DP=6 (pierwsza mała, górna tabelka od prawej strony). @2 w wierszu 10 mówi, że interesuje nas adres 2 komórki na 6-tej stronie pamięci danych ($6 * 0x80 + 2 = 0x302$), a więc w tym przypadku adres 302h. Zatem zawartość komórki o adresie 0x302 czyli 0x100 dodajemy do A i wynik zapisujemy w A.
11. Zawartość komórki pamięci danych o adresie równym zawartości AR2 (zawartość AR2 to 0x300, a odpowiadająca temu adresowi zawartość komórki danych to 0x130 – patrz nad tabelką) dodaj to do zawartości akumulatora A i zapisz wynik w A, potem inkrementuj zawartość AR2.
12. Zawartość komórki pamięci danych o adresie równym zawartości AR2 (zawartość AR2 to 0x301, a odpowiadająca temu adresowi wartość komórki to 0x80 – patrz nad tabelką), odejmij od zawartości akumulatora A i zapisz wynik w A, następnie inkrementuj zawartość AR2.
13. Podaną w rozkazie (adresacja natychmiastowa) wartość dziesiętną 64 (co daje 40h) odejmij to od zawartości akumulatora A i zapisz wynik w A.
14. Zawartość komórki pamięci danych o adresie równym zawartości AR2 (zawartość AR2 to 0x302, a odpowiadająca temu adresowi zawartość komórki to 0x100 – patrz nad tabelką). Dodaj to do zawartości akumulatora A i wynik zapisz w B. Teraz *AR2-0 oznacza: od tego, co jest w AR2, odejmij to, co jest w AR0 ($0x302-2=0x300$) i zapisz w AR2.
15. Zawartość komórki pamięci danych o adresie równym zawartości AR2 (zawartość AR2 to 0x300, a odpowiadająca temu adresowi zawartość komórki pamięci danych to 0x130 – patrz nad tabelką). Odejmij to od B i zapisz wynik w A.
16. Załaduj liczbę dziesiętną 160 (co daje A0h) do AR0.
17. Zawartość komórki pamięci danych o adresie równym zawartości AR1 (zawartość AR1 to 0x202, a odpowiadająca temu adresowi zawartość komórki to 0x20 – patrz nad tabelką). Dodaj to do A i zapisz wynik w A. Teraz AR1-0 oznacza: od tego, co jest w AR1, odejmij to, co jest w AR0 i zapisz w AR1 ($202h-A0h=162h$).
18. To, co jest w A, załaduj do komórki o adresie równym zawartości AR1.

UWAGA: w powyższej tabelce znajduje się kolejny drobiazg. W obszarze danych (nad tabelką) jest puste miejsce pod adresem 0x62 sugerujące miejsce czekające na wstawienie czegoś. Tymczasem z 18 wiersza tabelki wynika, że trzeba załadować wartość znajdującą się w akumulatorze AL. do komórki o adresie 0x162 (i nie jest to omyłka). W takim przypadku można obok tabelki narysować poglądowo fragment pamięci z komórką o adresie 0x162, do której ma trafić wartość zawarta w akumulatorze, np. tak jak obok:

.	
.	
160	
161	
162	2D0
.	
.	

Poniżej wypełnione tabelki dla innych dwóch grup z 2004 roku.

Dane:

	DP=0		DP=2		DP=4	
	Adres	Wartość	Adres	Wartość	Adres	Wartość
CPL=0	60	60	100	120	200	130
CMPT=0	61	40	101	60	201	50
Adr./Dane HEX	62	240	102	20	202	100

Program	A	B	DP	AR0	AR1	AR2
LD #0,DP			0			
STM #2,AR0				2		
STM #100h,AR1					100	
STM #200h,AR2						200
LD @0x61,A	40					
ADD *AR1+,A	160				101	
SUB @60h,A,B		100				
ADD *AR1+,B,A	160				102	
LD #4,DP			4			
ADD @1,A	1B0					
ADD *AR2+,A	2E0					201
SUB *AR2+,A	290					202
SUB #64,A	250					
ADD *AR2-0,A,B		350				200
SUB *AR2,B,A	220					
STM #160,AR0				A0		
ADD *AR1-0,A,A	240				62	
STL A,*AR1-					61	

Dane:

	DP=0		DP=2		DP=4	
	Adres	Wartość	Adres	Wartość	Adres	Wartość
CPL=0	60	60	100	120	200	130
CMPT=0	61	140	101	30	201	70
Adr./Dane HEX	62	310	102	20	202	100

Program	A	B	DP	AR0	AR1	AR2
LD #0,DP			0			
STM #2,AR0				2		
STM #100h,AR1					100	
STM #200h,AR2						200
LD @0x61,A	140					
ADD *AR1+,A	260				101	
SUB @60h,A,B		200				
ADD *AR1+,B,A	230				102	
LD #4,DP			4			
ADD @1,A	2A0					
ADD *AR2+,A	3D0					201
SUB *AR2+,A	360					202
SUB #64,A	320					
ADD *AR2-0,A,B		420				200
SUB *AR2,B,A	2F0					
STM #160,AR0				A0		
ADD *AR1-0,A,A	310				62	
STL A,*AR1-					61	

Dane:

	DP=0		DP=2		DP=4	
	Adres	Wartość	Adres	Wartość	Adres	Wartość
CPL=0	60	60	100	120	200	130
CMPT=0	61	140	101	30	201	470
Adr./Dane HEX	62	FF10	102	20	202	100

Program	A	B	DP	AR0	AR1	AR2
LD #0,DP			0			
STM #2,AR0				2		
STM #100h,AR1					100	
STM #200h,AR2						200
LD @0x61,A	140					
ADD *AR1+,A	260				101	
SUB @60h,A,B		200				
ADD *AR1+,B,A	230				102	
LD #4,DP			4			
ADD #112,A	2A0					
ADD *AR2+,A	3D0					201
SUB *AR2+,A	FF FFFF FF60*					202
SUB #64,A	FF FFFF FF20					
ADD *AR2-0,A,B		20				200
SUB *AR2,B,A	FF FFFF FEF0					
STM #160,AR0				A0		
ADD *AR1-0,A,A	FF FFFF FF10				62	
STL A,*AR1-					61	

*Uwaga, od tego momentu wkraczamy w liczby ujemne

I jeszcze jedna uwaga. Wyniki obliczeń podlegają wpływowi ustawienia bitu OVM. W zamieszczonych przykładach nie było powodu tym się zajmować bo nie zbliżaliśmy się do granic nasycania. Ale problem istnieje!

2. Pytania z testów - opracowanie

Wymień główne cechy wyróżniające procesory sygnałowe od innych procesorów i mikrokontrolerów.

- sprzętowa jednostka mnożąca (MAC)
- szybki shifter (Barrel Shifter) do skalowania danych
- sprzętowe nasycanie i zaokrąglanie
- sprzętowy mechanizm realizacji pętli poprzez repetycję rozkazów i bloków rozkazów
- specjalizowane rozkazy do przetwarzania sygnałów (FIRS, SUBC, POLY,...)
- jednostki arytmetyczne dla obliczeń na adresach
- liczne, specjalizowane rejestry do adresacji pośredniej
- rozbudowany mechanizm modyfikacji adresów wpomagający specyficzne korekty adresów np. dla potrzeb FFT
- sprzętowy mechanizm obsługi buforów kołowych
- sprzętowe, wewnątrz struktury procesora wsparcie mechanizmu debugowania i emulacji
- zwielokrotnienie i specjalizacja magistral w tym osobne magistrale danych i programu procesora
- rozbudowane systemy pamięci notatnikowych (cache)
- znaczne moce obliczeniowe wyrażane w MIPS i FLOPS w zależności od typu i specjalizacji procesora

- wydzielone magistrale specjalizowane do obsługi strumienia danych sygnału (zwykle magistrale szeregowo)
- ultra małe obudowy (BGA, TQFP)

Jaka jest w procesorach C54xx rola rejestrów statusowych i dlaczego jest ich aż trzy?

Rejestry te służą do zachowania informacji o stanie pracy procesora i wybranych ustawieniach. Jest ich aż trzy (ST0, ST1, PMST) z racji tego, że zachodzi potrzeba przechowania liczby informacji, która nie da się przechować w mniejszej liczbie rejestrów.

Jakie zmiany w architekturze wprowadzone w kolejnych generacjach procesorów pozwoliły na zwiększenie szybkości wykonania programu?

Na przykładzie różnic między C54xx, C55xx, C6000:

- zwielokrotnienie zasobów
 - MAC i ALU
 - Akumulatorów
 - dodatkowe generatory adresów
 - dodatkowe jednostki przetwarzania równoległego (do 8-miu)
- poszerzenie magistral
- poszerzenie listy rozkazów / procedur specjalizowanych
- rozbudowa mechanizmów dostępu do danych i programu
- rozbudowa mechanizmu cache wielopoziomowego
- zwiększenie równoległości przetwarzania
- wydłużenie słowa adresowego (architektura WLIV)
- wprowadzenie sprzętowych jednostek zmiennoprzecinkowych operacji
- wprowadzenie specjalizowanych jednostek – koprocessorów - np. do obsługi różnych standardów czy peryferii.
- Zwielokrotnienie rdzeni procesorów DSP

Od czego można uzależnić przebieg programu w procesorach rodziny C54xx?

Generalnie sekwencyjny przebieg rozkazu modyfikują skoki. W tym skoki warunkowe mają szczególne znaczenie, bo realizowane są w zależności od spełnienia lub nie warunku lub warunków. Pytanie dotyczy wskazania od czego można uzależnić przebieg programu czyli jakie warunki jesteśmy w stanie wykorzystać w tych warunkowych skokach, a zatem;

Wiele stanów jest wykrywane i sygnalizowane flagami;

- OVA i OVB czyli przekroczenia w każdym z akumulatorów
- C – Carry – przeniesienie w użytym akumulatorze,
- TC – bit, gdzie trafiają wyniki operacji logicznych

Inne elementy mogą być wykrywane bezpośrednio (zwykle komparatorami);

- Zawartość akumulatora i jej relacja względem zera,
- Relacja między zawartościami obu akumulatorów,
- Zawartość rejestru adresowego i jego wartość zerowa,
- Stan wejścia sygnałowego BIO
- Stan testowanego dowolnego bitu w pamięci danych (trafi do TC)
- Zawartość całej komórki w pamięci danych
- Iloczyn do trzech warunków równocześnie w rozkazach warunkowych

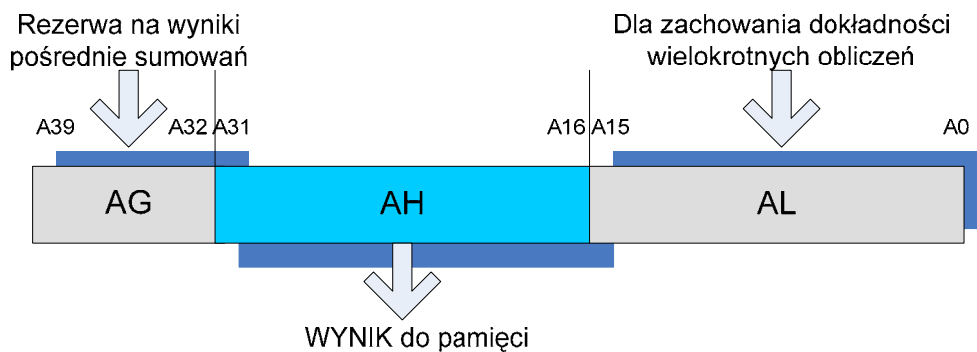
Warunki te mogą być wykorzystywane w takich rozkazach sterujących przebiegiem programu jak:

- instrukcji skoku (B, BACC, BANZ, BC)

- instrukcji wywołań procedur (CALL, CALA, CC)
- instrukcji odpowiedzialnych za obsługę przerwań (INTR, TRAP)
- instrukcji powrotu (RETE, RET)
- instrukcji manipulujących danymi na stosie (FRAME, POPD, PSHD, PSHM...)
- instrukcji odpowiedzialnych za repetycję (RPT, RPTB, RPTZ)
- innego typu instrukcji (RESET, SSBX, RSBX, NOP, IDLE)

Dlaczego w procesorach sygnałowych rejestr akumulatora jest ponad dwa razy większy od rozmiaru słowa, jakim pracują?

Aby przyjąć wynik mnożenia liczb binarnych, potrzebny jest akumulator będący dwukrotnie większy niż rozmiar słowa. Wynik mnożenia odbierany do przechowania w pamięci danych znajduje się w takiej sytuacji na starszej części akumulatora – AH. Młodsza część akumulatora – AL młodsze 16 bitów – ma za zadanie zapewnienie większej rozdzielczości dla sumowania wyników mnożeń i uniknięcia kumulowania błędów na skutek przedwczesnego ograniczania rozdzielczości reprezentacji. Ta młodsza część akumulatora może uzyskać swój wpływ na wartość wyniku poprzez operację zaokrąglania (Rounding). Z kolei dodawanie może doprowadzić do wyniku wykraczającego ponad 32 bity akumulatora stąd rezerwę do sumowania lub wyników pośrednich sumowania zapewniają bity AG - GUARD (jest ich osiem).



Co to jest przetwarzanie nakładkowe, na czym polega i czemu służy?

Przetwarzanie nakładkowe, albo kolejka (żargonowo pipelining), jest to sposób wykorzystania zasobów procesora do realizacji rozkazów tak, by żaden jego fragment „nie stał beczynnym”. Uwarunkowane jest podziałem realizacji rozkazu na kolejne fazy wykonywane w pojedynczych cyklach rozkazowych i możliwościami bloków przetwarzających procesora oraz magistral transportu danych i rozkazów. Polega on na równoczesnym wykonywaniu różnych faz kolejnych rozkazów programu. Przykładowo, wykonując fazę Pre-Fetch dla jednej instrukcji, procesor może jednocześnie wykonywać fazę Fetch poprzedniej instrukcji, fazę Decode dla jeszcze wcześniejszej instrukcji itd. Dzięki temu rozkazy pobierane do kolejki są wykonywane quasi równolegle – po jednej fazie z każdego z kolejnych rozkazów – jak gdyby cały jeden rozkaz w jednej fazie.

Technika ta nie skraca czasu wykonywania pojedynczego rozkazu ale dzięki nakładaniu na siebie rozkazów pozwala na skrócenie wykonywania sekwencji rozkazów i przyspieszenie wykonania całego programu.

W przetwarzaniu kolejkowym pobierając zawartość kolejnych komórek z pamięci programu natrafia się na trudności związane z wykonywaniem skoków. Typowy skok przerywa ciągłość kolejki uniemożliwiając wykorzystanie wszystkich rozkazów pobranych do kolejki. (Jeśli wykonujemy skok nie ma uzasadnienia do wy-

konywania rozkazów po skoku.) Przeciwdziałać tym trudnościom można przez psecyficzną modyfikację programu polegającą na;

1. wykorzystaniu rozkazów skoków z opóźnieniem (np. BD, RD, ...) i
2. przestwieniu niektórych rozkazów sprzed rozkazu skoku (np. B, R, ...) za rozkazy (np. BD, RD, ...).

Dlaczego pojedyncza magistrala zewnętrzna procesora DSP stanowi istotne ograniczenie dla jego szybkości działania?

Jeżeli pamięć programu i danych będą umiejscowione na zewnątrz procesora, to pojedyncza magistrala zewnętrzna może transportować tylko jeden obiekt. Albo kod rozkazu, albo jedną daną. To zaś uniemożliwia wykorzystanie walorów kolejki i szybkość realizacji programu spada. Może to spowodować obniżenie efektywności o co najmniej 50%.

Dlaczego w procesorze DSP stosuje się wiele równoległych magistral transportowych?

Dlatego, że inaczej nie można wykorzystać walorów przetwarzania nakładkowego (kolejki). Jest ono tylko wtedy efektywne, gdy możliwe jest pobieranie w jednym cyklu zarówno oparandów do przetwarzania (nawet dwóch równocześnie) jak też i kodu kolejnego rozkazu oraz odsyłanie wyniku operacji do pamięci. Wielość magistral do równoległych transferów, praca z pamięcią podwójnego dostępu wraz ze specjalnymi technikami wykonywania rozkazów (rozказы z opóźnieniem) i modyfikacja kolejności rozkazów przekłada się na szybszą realizację całego programu.

Co to jest DARAM i dlaczego jest korzystna w procesorach DSP C54xx?

DARAM (Double Access RAM) jest to pamięć zezwalająca na 2 dostępy w jednym cyklu procesora w każdym z bloków pamięci. Oznacza to, że zarówno CPU jak i peryferia mogą dokonywać odczytu i zapisu w tym samym cyklu. Część rozkazów może być efektywnie wykonywana tylko gdy ich operandy rozmieszczone są w DARAM (np. rozказы z grupy MAC). Istotne jest ponadto, że pamięć DARAM z przestrzeni pamięci danych można przełączyć (uwidocznic) do przestrzeni pamięci programu. Służy do tego odpowiednie ustawienie bitu OVLY.

Wymień tryby adresacji stosowane w rodzinie procesorów TMS320C54xx i podaj przykłady rozkazów stosujących je.

Adresacja	Przykład	Przeznaczenie, zalety
Natychmiastowa (Immediate)	LD #10,A	- operand bezpośrednio w kodzie rozkazu - użyteczne do inicjalizacji
Absolutna (Absolute)	STL A,*(y)	- używa 16- bitowego adresu dowolnej komórki - wymusza dwusłowy rozkaz
Akumulatorem (Accumulator)	READA x	- adresem operandu w pamięci programu jest zawartość akumulatora
Pośrednia (Indirect)	LD *AR1,A	- adresem operandu jest zawartość aktywnego rejestru (ARi) użyta jako wskaźnik
Bezpośrednia (Direct)	LD @x, A	- adresacja względem wskaźnika strony -DP albo wskaźnika stosu -SP (decyduje bit CPL)
Na stosie (Stack)	PSHM AG	- push / pop danej z pamięci danych lub z MMRs (rejestrów widocznych w przestrzeni pamięci) jako adres używana jest zawartość wskaźnika

stosu SP

W rejestrach „zmapowanych” w pamięci MMR LDM ST1,B - adresacja w obszarze rejestrów MMR (strona zerowa pamięci danych) i z użyciem ich nazw (po włączeniu .mmregs)
- szybki dostęp do rejestrów MMR

Jak rozumiesz i co określa pojęcie trybu adresacji?

Tryb adresacji jest sposobem definiowania dostępu do operandu w treści rozkazu (podawania adresu w rozkazie). Określa on, w jaki sposób instrukcje sięgają do swoich operandów w pamięci. Wyróżniamy następujące tryby adresacji:

- natychmiastowy np. LD #10,A
- absolutny LD A, *(y)
- akumulatorem READ A x
- pośredni LD *AR1,A
- bezpośredni LD @x,A
- za pomocą wskaźnika stosu PSHM ST1
- MMR LDM ST!,B

Tryby zdresacji i ich rozumienie i sprawność wykorzystania to kluczowy element efektywnego programowania przetwarzania we wszystkich procesorach nie tylko sygnałowych.

Wymień sposoby modyfikacji zawartości rejestrów adresowych procesorów C54xx i podaj ich przykładowe przeznaczenie.

Opcja	Składnia	Sposób realizacji
Bez modyfikacji	*ARn	ARn bez zmian
Post-Inkrement / Post-Dekrement	*ARn+ *ARn-	post inkrementacja o 1 post dekrementacja o 1
Post-Indeksowana	*ARn+0 *ARn-0	post inkrementacja o zawartość AR0 post dekrementacja o zawartość AR0
Post-Mod-Kołowa (circular)	*ARn+% *ARn-% *ARn+0% *ARn-0%	kołowo post inkrementacja o 1 kołowo post dekrementacja o 1 kołowo post inkrementacja o zawartość AR0 kołowo post dekrementacja o zawartość AR0
Post-z odwr. Bitów (Bit-Reversed)	*ARn+0B *ARn-0B	post inkrementacja o AR0 z odwróc. bitów post dekrementacja o AR0 z odwróc. bitów
Pre-modyfikacja	*ARn(lk) *+ARn(lk) *+ARn(lk)% *+ARn	chwilowe pre *(ARn+l), bez zmiany ARn! trwałe pre *(ARn+l), trwałe kołowe pre *(ARn+l), trwałe pre *(ARn+1) ale tylko do zapisu

Przykładowe zastosowania:

- inkrement/dekrement – dostęp do tablic, wektorów, sygnałów
- kołowe – dostęp do tablic i wektorów ale ze sprzętową kontrolą przemieszczania się w buforze (zapewnia automatyczny skok na/przez początek/koniec bufora), obsługa buforów współczynników i próbek dla filtrów, transformat i transferu danych
- z odwróceniem bitów - dla szybkiej transformaty Fouriera (FFT) i innych transformat wykorzystujących własności symetrii funkcji sin/cos

Co to są sekcje programu i do czego są używane?

Sekcje to fragmenty programu zawierające jednorodne obiekty; kod, stałe, zmienne lub układy we/wy. Są one zdefiniowane za pomocą dyrektyw w zbiorach źródłowych.

Sekcje dzielimy wg. zawartości na;

- sekcja inicjalizowana (kod programu, predefiniowane stałe),
- sekcja nieinicjalizowana (rezerwacja obszarów pamięci na zmienne czy stałe) i wg. opisu na
- sekcja nazwana (opatrzone nazwą)
- sekcja nienazwana (bez nazwy)

Sekcje są umieszczane przez linker we wskazanych obszarach pamięci zgodnie z zapisem zbioru konfiguracyjnego. Sekcje o tych samych nazwach łączone są we wspólne obszary ułatwiając organizację danych w pamięci.

Co to jest dyrektywa assemblera i do czego służy?

Dyrektywa assemblera jest to polecenie definiujące mu sposób traktowania danego fragmentu programu. Są elementem sterowania asemlacją programu. Nie są tłumaczone na rozkazy programu a jedynie uruchamiają sposób działania asemlera. Dyrektywy mogą służyć np. do zdefiniowania sekcji w zbiorach źródłowych, uaktywnienia własności asemlera, itd. Są one poleceniami tekstowymi i zaczynają się od kropki.

Przykładami dyrektyw mogą być:

.mmregs	; włącza predefiniowane nazwy rejestrów MMR
.sect „kot”	; kończy poprzednio zdefiniowaną sekcję i otwiera nową ; inicjalizowaną i nazwaną „kot” sekcję na kod programu ; lub dane
.text	; kończy poprzednio zdefiniowaną sekcję i otwiera nową ; inicjalizowaną i nazwaną sekcję na kod programu
.bss test,n	; kończy poprzednio zdefiniowaną sekcję i otwiera nową ; nieinicjalizowaną sekcję o nazwie test na n słów danych
.usect „pies”, n	; kończy poprzednio zdefiniowaną sekcję i otwiera nową ; nieinicjalizowaną i nazwaną „pies” sekcję dla danych ; rezerwując dla nich n słów w pamięci danych
tab .word 4, 0x13, 66h	; kończy poprzednio zdefiniowaną sekcję i otwiera nową ; inicjalizowaną sekcję dla trzech wartości 4, 13h, 66h ; zaczynającą się od adresu „tab”

Objaśnij zadania linkera w środowisku programów do generacji kodu procesora DSP.

Linker łączy plik *.obj i generuje docelowy plik wyjściowy *.out. Rozmieszcza on i łączy jednoimienne sekcje w obszarach pamięci wskazanych w zbiorze/poleceniach konfiguracyjnych. Linker może generować różne, pomocne w analizie i uruchamianiu programu zbiory np. *.map – mapę pamięci, *.lst – pełnego listingu programu, *.hex – zbiór dla programatora pamięci, itd. Zajmuje się on rozmieszczeniem relokowalnych zbiorów *.obj a w nich symboli i sekcji, by przypisać je do ostatecznych adresów oraz decyduje o zewnętrznych powiązaniach między plikami wejściowymi i bibliotekami. Do prawidłowego działania linkera niezbędny jest zbiór konfiguracyjny linkera - Linker Command File (w CCS ma on rozszerzenie .cmd).

Wymień czynniki decydujące o szybkości realizacji programu w DSP.

a) wynikające z budowy procesora

- częstotliwość taktowania procesora
- przetwarzanie nakładkowe
- zwielokrotnienie magistral
- rozkazy specjalizowane i ukierunkowane na aplikacje
- zastosowanie adresacji kołowej lub z odwracaniem bitów
- rozkazy skoków z opóźnieniem
- łączone warunki dla skoków i operacji warunkowych
- wykonywanie rozkazów w trybie repetycji
- zaawansowana obsługa pośrednich wyników operacji
- operacje dwusłowe
- wielkość pamięci wewnętrznej, szczególnie DARAM
- ilość i sposób wykorzystania przerwań oraz ich ewentualne kolizje z trybami repetycji

b) wynikające ze sposobu przygotowania programu

- wykorzystanie wymienionych wyżej możliwości sprzętowych
- podział programu pomiędzy assembler i języki wysokiego poziomu
- rozmieszczenie danych w pamięciach SARAM / DARAM, pamięci zewnętrznej i/lub wewnętrznej

Omów sposoby realizacji pętli i stosowane tam rozkazy.

Do realizacji pętli mogą zostać wykorzystane następujące rozwiązania:

- repetycja pojedynczego rozkazu realizowana instrukcją RPT lub RPTZ, pozwala na powtórzenie instrukcji od 1 do 65536 razy. Różnica między RPT a RPTZ polega na tym, że instrukcja RPTZ resetuje wskazany w rozkazie akumulator A lub B przez rozpoczęciem pętli.
RPT n \rightarrow n+1 powtórzeń
- repetycja bloku rozkazów realizowana za pomocą instrukcji RPTB. Pozwala ona na powtórzenie bloku instrukcji od 1 do 65536 razy. Liczbę obiegów pętli ustala się przez zawartość BRC (Block Repeat Counter) plus jeden.
BRC=n \rightarrow n+1 powtórzeń
- instrukcje skoku warunkowego, wykonujące skok tylko wtedy, gdy spełniony jest dany prosty lub złożony warunek (w przeciwnym razie wykonanie programu przechodzi do następnej instrukcji). Wyróżniamy dwie instrukcje skoku warunkowego:
 - BC: przeładowuje PC bezpośrednim 16-bitowym adresem, gdy spełniony jest prosty lub złożony warunek. Zazwyczaj wykorzystywane do testów arytmetycznych wykonywanych na zawartości akumulatora lub testowania flag.
 - BANC: przeładowuje PC, jeżeli zawartość wybranego rejestru pomocniczego AR nie jest równa zero. Rejestr ten wykorzystywany jest jako licznik pętli gdyż rozkaz BANC poza sprawdzaniem wykonuje również jego dekrementację. Wykorzystuje się to w szczególności do implementacji pętli FOR.
- instrukcje skoku bezwarunkowego – dla pętli bez końca.
- w obsłudze pętli:
rozkazy skoków czy repetycji bloków muszą zawierać etykiety wskazujące zakres skoku / pętli

B „etykieta”
przed wejściem w obręb pętli należy utworzyć licznik pętli np.
MVK 40, ctr ;ctr licznikiem
we wnętrzu pętli zapewnić dekrementację licznika pętli
SUB ctr, 1, ctr

Co to są tryby repetycji i czemu służą w procesorach DSP rodziny C'54xx?

Tryb repetycji polega na powtarzaniu rozkazu lub bloku rozkazów. W trybie tym dzięki sprzętowej obsłudze licznika pętli nie tracimy czasu na rozkazy sprawdzające licznik i realizujące skok. Stąd pętle takie są bardziej efektywne, tylko użyteczna część pętli zajmuje czas wykonania.

Repetycja pojedynczego rozkazu:

- uruchamiana instrukcją RPT lub RPTZ, pozwala na powtórzenie następnej instrukcji od 1 do 65536 razy. Różnica między RPT a RPTZ polega na tym, że instrukcja RPTZ resetuje akumulator A lub B przez rozpoczęciem pętli.

RPT n → n+1 powtórzeń. Niestety pętli takiej nie można przerwać przerwaniem!

Repetycja bloku rozkazów:

- uruchamiana za pomocą instrukcji RPTB „etykieta”
albo z zastosowaniem mechanizmu opóźnienia RPTBD „etykieta”.

Pozwala na powtórzenie od 1 do 65536 razy bloku instrukcji od inicjującego rozkazu do rozkazu opatrzonego etykietą. Liczba przebiegów zadana jest zawartością BRC (Block Repeat Counter) plus jeden; BRC=n → n+1 powtórzeń. Zakres pętli zadawany jest zawartościami rejestrów RSA – adres początku pętli, REA – adres końca pętli.

Dla sekwencji rozkazów: CALL adres ;proc_adres adres procedury
STM #(S+L),SP

a) ile cykli procesora upłynie od początku tej sekwencji do rozpoczęcia wykonywania pierwszej instrukcji z wywoływanej procedury? - 4 cykle, a procesor zablokuje wykonanie rozkazu STM (w przypadku gdy będzie tam rozkaz CALLD będą też 4 cykle, ale rozkaz STM zostanie wykonany)

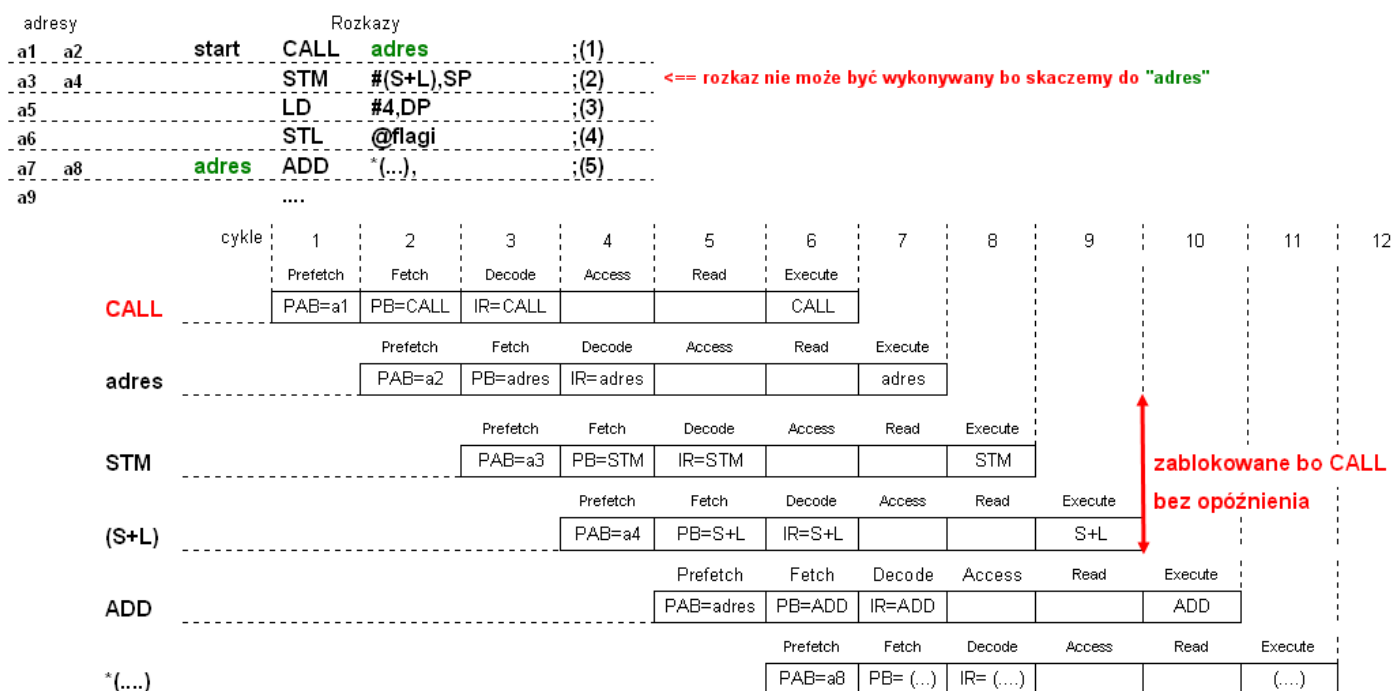
b) ile cykli procesora upłynie od początku tej sekwencji do końca drugiego rozkazu? - 9 cykli i jak poprzednio, procesor zablokuje wykonanie rozkazu STM (w przypadku z CALLD będzie też 9 cykli, ale rozkaz STM zostanie wykonany)

c) ile słów zajmują te rozkazy w pamięci programu? - 4 (2 słowa CALL + 2 słowa STM), w przypadku z CALLD będzie tak samo

d) ile słów zajmują te rozkazy w pamięci danych? - 0 słów (tak samo CALLD)

e) ile słów użyją te rozkazy w pamięci danych? (uzasadnij odpowiedź) - 2 słowa, bo STM nie zostanie w prawdzie wykonany, procesor zablokuje jego wykonanie, ale CALL odeśle na stos adres kolejnego rozkazu po CALL. Zaś stos mieści się w pamięci danych. Zatem użyjemy komórki z przestrzeni stosu oraz SP, który jest rejestrem MMR znajdującym się w pamięci danych. (By ujawnić własny sposób myślenia i mieć szansę na zdobycie punktów należy skorzystać z tego wezwania do uzasadnienia odpowiedzi.)

W przypadku, gdy będzie tam rozkaz CALLD – będą tym bardziej dwa słowa. Rozkaz STM zostanie wykonany i zapisze wartość do wskaźnika stosu SP, który jest rejestrem MMR znajdującym się w pamięci danych.



W jaki sposób i po co programista może określać/zmieniać położenie tablicy wektorów przerwań (początków procedur przerwań)?

Po resecie sprzętowym procesor nadając wartość rejestrowi IPTR równą 0x1FF będzie sięgał do tablicy wektorów przerwań zaczynającej się od adresu 0xFF80. Domyślnie, dla takiej sytuacji tablica wektorów przerwań jest lokowana w zakresie adresów od FF80h do FFFFh w przestrzeni pamięci programu.

Można przygotować inną/inne tablice w przestrzeni pamięci programu, zaczynające się od adresów równych (IPTR)*128 i wskazać ją procesorowi do użycia poprzez nadanie odpowiedniej wartości rejestrowi IPTR a następnie wykonanie **programowego** reset (czyli rozkazu RESET).

Mechanizm taki jest zaimplementowany z tego powodu, by użytkownik mógł reorganizować strukturę przerwań swego programu w zależności od potrzeb.

Np. gdy nie chce używać domyślnych wektorów przerwań rezydujących w pamięci ROM układu może przesunąć wskazanie tablicy wektorów do dowolnej 128-słowej przestrzeni w pamięci programu i tam przygotować jej własną wersję.

Co to jest i czemu służy w procesorach rodziny C'54xx IPTR?

IPTR (czyli Interrupt Pointer) to 9-cio bitowy rejestr, fragment rejestru PMST. Jego zawartość stanowi najstarsze 9 bitów adresu w tablicy wektorów przerwań. Uzupełniona kodowanym na 5 bitach numerem przerwania i najmłodszymi dwoma bitami 00 tworzy adres początkowy w tablicy wektorów przerwań procesora. Te 9 bitów uzupełnione 7-mioma zerami tworzy adres położenia pierwszego wektora w tej tablicy. Jest to wektor przerwania RESET złożony z pierwszych czterech słów programu jego obsługi – startu procesora. Po sprzętowym RESET procesora IPTR = 1 1111 1111B = 0x1FF co lokuje początek tej tablicy pod adresem 0xFF80 (bo jego zawartość umieszczana jest na najstarszych bitach uprzednio wyzerowanego rejestru PC). Programista może przełączyć procesor do odczytywania tablicy wektorów przerwań z innego miejsca w pamięci programu przez zmianę zawartości IPTR i wykonanie programowego RESET.

Co to jest tablica wektorów przerwań i do czego ona służy?

Tablica wektorów przerwań jest to obszar w pamięci programu procesora, gdzie umieszczone są czterosładowe wektory przerwań, będące początkami procedur obsługi przerwań odpowiadających danym lokacjom w tablicy. Domyślnie rozpoczyna się ona pod adresem 0xFF80 ale programista może wskazać procesorowi inne jej położenie w pamięci programu, odpowiednio przygotowując wcześniej tam jej zawartość i modyfikując zawartość IPTR przed sprzętowym RESET.

Tablica ta służy wiązaniu odpowiednich przerwań procesora z obsługującymi je procedurami obsługi – czyli procedurami reakcji na fakt wystąpienia danego przerwania.

Co to jest przerwanie?

Przerwanie (ang. interrupt) – to mechanizm służący synchronizacji przebiegu programu z niezależnymi od programu zdarzeniami. Służą do tego sygnały informujące o wystąpieniu zdarzenia, procedury reagowania na zdarzenia – obsługi tych zdarzeń, oraz mechanizmy maskowania i szeregowania ważności tych zdarzeń – ich priorytetów. Decydują one, czy zgłoszenie zdarzenia zostanie zauważone (obsłużone) przez procesor a w przypadku równoczesnego zgłoszenia dwóch zdarzeń rozstrzygają, które z nich należy obsłużyć najpierw.

Zdarzenia mogą być wewnętrzne np. zmiany w zasobach wewnętrznych procesora (przepełnienie licznika, koniec transmisji danych, koniec przetwarzania wewnętrznego przetwornika A/C, itp.) albo zdarzenie zewnętrzne, które generują sygnały doprowadzone do wejść przerwań zewnętrznych (INT0, INT1, ... INTn). Przerwanie może wystąpić w dowolnym momencie (w dowolnej fazie cyklu procesora) niezależnie od programu, ale zawsze wymaga dokończenia właśnie realizowanego rozkazu przed przystąpieniem do oceny i obsługi oczekującego, najważniejszego przerwania. Pojawienie się niezamaskowanego przerwania powoduje wstrzymanie aktualnie wykonywanego programu i wykonanie przez procesor kodu procedury obsługi przerwania (ISR, lub interrupt handler).

Mówiąc o przerwaniach należy starannie formułować wypowiedzi bo w technicznym żargonie często terminem „przerwania” określa się zarówno sygnały jak i same programy obsługi przypisane zdarzeniom czy też same zdarzenia obsługiwane tym mechanizmem.

Co to jest procedura obsługi przerwania i jakie są jej główne cechy?

Procedura obsługi przerwania (ISR) to przygotowany fragment programu zawierający sekwencję rozkazów opisujących sposób reagowania procesora (systemu) na występujące zdarzenie – jego sygnał.

Dla prawidłowego działania procedury obsługi przerwania należy poza jej przygotowaniem ustawić globalną maske przerwań - INTM, odpowiedni bit indywidualnej maski danego przerwania w rejestrze masek przerwań - IMR, wartość wskaźnika stosu - SP i umieścić początek procedury obsługi przerwania w tablicy wektorów przerwań – przygotować odpowiedni wektor.

Procedura obsługi przerwania jest zatem programem wywoływanym pow ustawieniu flagi danego przerwania, który po dokończeniu wykonywanego rozkazu i przy dopuszczeniu przerwania odpowiednimi stanami masek globalnej i indywidualnej zostanie uruchomiony począwszy od własnego wektora w tablicy wektorów przerwań. Po dostrzeżeniu sygnału przerwania (zawsze badanego na końcu każdego rozkazu), stwierdzeniu dopuszczalności przerwania (odpowiedniego ustawienia masek) a przed jej przywołaniem procesor:

- Automatycznie kasuje flagę przerwania,

- automatycznie odsyła na stos jedynie zawartość rejestru PC (czyli adres następnego rozkazu do wykonania po ukończeniu obsługi przerwania),
- automatycznie ustawia globalną maskę przerwania INTM co daje zablokowanie przerwania. (Zatem inne przerwy nie mogą wystąpić bez zgody programisty),
- tworzy przypisany dla zidentyfikowanego przerwania adres początkowy wektora w tablicy wektorów przerwania (pierwszego rozkazu procedury ISR) i umieszcza go w PC,
- odczytuje pierwszy rozkaz spod utworzonego adresu (4-ro słowowy wektor w tablicy zawiera zwykle skok do dalszego ciągu programu procedury ISR),
- dla przerwania zewnętrznego sygnalizuje rozpoczęcie wykonywania procedury linią INTA procesora ,

Procedurę ISR charakteryzuje zwykle

- na początku procedury konieczność zachowania (zwykle na stosie) stanu rejestrów procesora używanych w trakcie jej działania i odtworzenie ich zawartości na końcu procedury. (Zachowywanie rejestrów na stosie i pobieranie ich na końcu odbywają się w odwrotnej kolejności.)
- kończenie procedury rozkazem RET[D] lub RETE[D] by odblokować system przerwania.

Co wiąże, a co różni indywidualną maskę przerwania i flagę przerwania?

I flaga i maska są występującymi w różnych rejestrach bitami (przerzutnikami). Wiaże je to samo przerwanie, tyle że maska jest bitem blokującym lub dopuszczającym obsługę przerwania a flaga jest bitem zgłoszenia rządania obsługi przerwania.

Od strony operacyjnej łączy pewne podobieństwo. Flaga jest ustawiana sprzętowo za sprawą wystąpienia zewnętrznego sygnału, choć może być również ustawiona programowo. Jednak kasowana jest WYŁĄCZNIE sprzętowo na początku obsługi przerwania i po RESET. Zaś maska może być ustawiana i kasowana programowo a dodatkowo ustawiana jest sprzętowo przy rozpoczynaniu obsługi przerwania i po RESET.

Czego dotyczą operacje „context save” i „context restore” w procedurach ISR i jakim podlegają zasadom?

Operacje te dotyczą zachowania i odtworzenia stanu kluczowych rejestrów procesora oraz tych, które będą używane w trakcie ISR. Dotyczy to rejestrów statusowych procesora ST0, ST1, PMST oraz jeśli uruchomiona możliwość innego przerwania (zagłębiania przerwania) - również masek przerwania IMR. Potrzeba zachowania stanu rejestrów używanych w procedurze obsługi przerwania wynika z konieczności powrotu do przerwanej programu głównego z takim stanem procesora jaki został zastany przez przerwanie.

Operacje te realizowane są za pomocą następujących rozkazów:

Rozkaz	Opis
PSHM mmr	SP -1 → SP, potem odesłanie rejestru mmr na STOS
POPM mmr	pobranie danej ze STOSu do rejestru mmr, potem SP + 1 → SP
PSHD Smem	SP -1 → SP, potem odesłanie komórki Smem z pamięci danych na STOS
POPD Smem	pobranie danej ze STOSu do komórki Smem pamięci danych, potem SP + 1 → SP
FRAME K	modyfikacja wskaźnika stosu SP, SP + K → SP

Trzeba pamiętać o odwrotnej kolejności pobierania danych ze stosu do kolejności zapisywania na stosie.

Co to jest stos i jaka jest zasada jego działania i do czego on służy?

Stos jest to fragment obszaru pamięci danych, na którym adresację realizuje rejestr wskaźnika stosu SP. Stos jest realizacją rejestru typu LIFO i charakteryzuje się odwrotną kolejnością pobierania danych ze stosu do kolejności ich zapisywania.

Stos służy głównie do zachowania i ochrony stanu procesora w trakcie realizacji procedur obsługi przerwania (context save/context restore), zachowania adresów procedur przywoływanych rozkazami CALL, przekazu parametrów do procedur i funkcji, itd.

Wskaźnik stosu - SP wskazuje zawsze ostatnią zajętą komórkę stosu (czyli ostatnią odesłaną na stos daną), zatem dla;

Dla CALL: $PC \rightarrow *--SP$

dla odesłania stanu PC na stos najpierw musimy zmniejszyć stan rejestru SP o jeden by wskazać wolną komórkę pamięci na stosie a potem dopiero odesłać daną na wskazaną pozycję.

a dla RET: $*SP++ \rightarrow PC$

odczytujemy (popularnie pobieramy) zawartość szczytu stosu (TOS) i kierujemy do PC a potem zwiększamy wskaźnik stosu.

Dla zdefiniowania stosu należy:

1. Zadeklarować nieinicjalizowaną sekcję odpowiedniego rozmiaru, rezerwującą wystarczający obszar pamięci dla stosu.
2. Sekcję tę skierować (umieścić) za pośrednictwem zbioru konfiguracyjnego linkera w pamięci (najlepiej w pamięci wewnętrznej)
3. Zainicjować wskaźnik stosu (SP) by wskazał "szczyt stosu +1":

Jakie warunki i gdzie można sprawdzać w procesorze C54xx, czego one dotyczą i jakie rozkazy mogą wykorzystywać ich wyniki.

Trzeba zauważyć, że możliwość sprawdzania różnych warunków pozwala na realizację rozgałęzień programu poprzez wykorzystanie skoków warunkowych. W procesorze C5402 te możliwości są bardzo szerokie i obejmują nie tylko nadzór zawartości akumulatorów i ich wzajemnych relacji ale również zawartości rejestrów ARx, indywidualnych bitów w pamięci oraz wejścia sygnału BIO. Realizacji tej kontroli służą liczne flagi, komparatory oraz własności rozkazów.

W procesorach C54xx można sprawdzać następujące warunki dotyczące zawartości akumulatorów: [notacja warunek{wartość}]

AccA	AEQ{A=0}, ANEQ{A<>0},	ALT{A<0},	ALEQ{A<=0},
	AGT{A>0}, AGEQ{A>=0},	AOV{AOV=1},	ANOV{AOV=0},
AccB	BEQ{B=0}, BNEQ{B<>0},	BLT{B<0},	BLEQ{B<=0},
	BGT{B>0}, BGEQ{B>=0},	BOV{BOV=1},	BNOV{BOV=0},

flag sygnalizujących wyniki operacji (w tym na pamięci danych):

$C\{C=1\}$, $NC\{C=0\}$, $TC\{TC=1\}$, $NTC\{TC=0\}$,

Na wartości tych flag wpływają wyniki operacji/rozkazów:

- logiczne AND, OR, XOR (przy czym ANDM, ORM, XORM działając bezpośrednio na pamięci danych)
- testowania pojedynczych bitów (BIT, BITT)
- testowania pól bitów w pamięci danych (BITF)
- testowania relacji młodszych i starszych części akumulatora (CMPS)
- testowania relacji między całymi akumulatorami
- testowania zawartości komórki danych (CMPM)

- testowania relacji {EQ, LT, GT, ERQ} pomiędzy zawartościami ARn i AR0 (CMPR)

Stanu wejścia sygnału BIO - BIO{BIO=low}, NBIO{BIO= high}

Warunki te mogą być wykorzystywane w rozkazach warunkowych skoków (BC, BACC), odwołań (CC) i pominąć (XC). Cechą charakterystyczną jest możliwość łączenia kontroli do trzech warunków powiązanych operacją AND.

Do czego służy w procesorach DSP zegar (timer)?

Zegary (timery) w DSP można zastosować do:

- generację przerw po ustalonym programowo czasie (np. dla RTC)
- generowania impulsów zewnętrznych po ustalonym programowo czasie
- sterowania generacją impulsów PWM
- realizacji przetwornika C/A
- pomiar czasu trwania funkcji czy innych procesów software'owych
- generację impulsów i pomiar ich szerokości
- generacji zdarzeń synchronizujących dla DMA, A/C, C/A i innymi peryferiami.

Co odróżnia standardowy port szeregowy od McBSP w C54xx?

McBSP to wielokanałowy buforowany port szeregowy. Od standardowego portu szeregowego odróżniają go następujące cechy:

- pełny, dwukierunkowy bezpośredni interfejs do układu codec i innych urządzeń szeregowych.
- podwójne buforowanie dla nadawania i potrójne dla odbioru transmisji
- zdolność realizacji wielu standardów komunikacji szeregowej
- praca wielokanałowa maksymalnie do 128 kanałów
- długość słowa: 8-, 12-, 16-, 20-, 24-, 32-bit
- wewnętrzna generacja zegara/ramek z SGR (Sample Rate Generator)
- transmisja 8-bitowa danych z możliwością wyboru pierwszeństwa LSB lub MSB

Ponadto, podobnie jak standardowy port szeregowy, McBSP zapewnia:

- maksymalna szybkość bitowa: 1/2 CPU Clock Rate
- wbudowany komparing wg. praw μ lub A
- programowana polaryzacja zegara / ramek
- potrafi sygnalizować wszystkie typowe błędy i statusy

Na czym polega konfigurowanie do pracy peryferii w procesorach DSP?

Polega na zdefiniowaniu odpowiednich parametrów (głównie przy użyciu CSL - Chip Support Library) i wypełnieniu rejestrów konfiguracyjnych (określeniu ich zawartości), które mogą nadzorować pracę programu. Na CSL składają się:

- struktury danych (myConfig, etc.) - wartości do umieszczenia w rejestrach
- funkcje (DMA_config, etc.) - pozwalają na inicjowanie i zarządzanie zasobami
- makra (DMA_OPT_RMK(), etc.) - zapewniają dostęp z wysokiego poziomu do operacji niskiego poziomu

CSL zapewnia dwie podstawowe możliwości:

- programowanie peryferii
- kierowanie zasobami (utrzymanie sposobu użycia środków)

Do czego są szczególnie przydatne w procesorach DSP kanały DMA i z czym głównie współpracują?

Kanały DMA w procesorach DSP współpracują głównie z McBSP oraz D/A. Można mówić o ich szczególnej przydatności ze względu na to, że:

- DMA dla przesłań może sięgać do każdego zasobu
- prowadzą transfer danych bez zaangażowania CPU, a zatem odciążają procesor
- posiadają autoinit (automatyczne ustawienie następnego kanału do transferu)
- transfer można synchronizować np. z 20 zdarzeniami w C55xx
- każdy z kanałów dysponuje FIFO by zapis mógł wyprzedzać odczyt (gdy zasoby docelowe są zajęte)
- przy wydzieleniu kanału z użyciem DMA obsługiwanych może być do 128 kanałów
- pozwala na niezależny wybór wielu kanałów (słów), które mają być transmitowane i odbierane
- elastycznie indeksowana adresacja DMA pozwala na sortowanie każdego kanału do oddzielnego bufora

Jednym z najlepszych zastosowań DMA w procesorach DSP jest powiązany z sortowaniem automatyczny transfer danych między portami McBSP a RAM zawartym w układzie. Upraszcza to zarządzanie i obsługę wielokanałowych portów szeregowych przez procesor. W miarę rozbudowy mechanizmu DMA w kolejnych generacjach procesorów jego funkcjonalność wzrasta dzięki rozbudowie mechanizmów synchronizacji i wiązania w łańcuchy (sekwencje) powiązanych operacji.

Co to jest i do czego służy emulator (ICE) procesora DSP?

Emulator to urządzenie, którego zadaniem jest zapewnić kontrolę nad pracą procesora DSP. Umożliwia w zasadzie bardzo podobne funkcje jak debugger. Różnica polega na tym, że w przypadku debugera funkcje te są zapewniane i obsługiwane przez program uruchomiony na docelowym procesorze, natomiast emulator częściowo działa na procesorze, wykorzystując jego zasoby m.in. do komunikacji i do nadzoru a częściowo na komputerze nadrzędnym - Host. Wykorzystanie emulatora przekłada się na ułatwienie prac uruchomieniowych oprogramowania procesora, poprawę możliwości diagnostycznych błędów programu działającego z pełną szybkością i z podłączonymi peryferami a dzięki temu możliwość sprawdzenia programu w prawdziwych warunkach i szybsze przygotowanie produktu.

Dla 12-to bitowej reprezentacji liczb kodowanych U2 i I3Q9 określ zakres (MAX, MIN) i rozdzielczość reprezentacji (LSB).

Należy skorzystać z następujących wzorów:

$$\begin{array}{ll} \text{min} & -(2^{I-1}) \\ \text{max} & 2^{I-1}-2^{-Q} \\ \text{rozd.} & 2^{-Q} \end{array}$$

Zatem:

$$\begin{array}{ll} \text{min} & -(2^2) = -4 \\ \text{max} & 2^2-2^{-9} = 4 - 0,001953125 = 3,998046875 \\ \text{rozd.} & 2^{-9} = 0,001953125 \end{array}$$

Po co i jak stosuje się zaokrąglenie wyniku?

W procesorach sygnałowych rodziny 'C5000 wyniki operacji umieszczane są w akumulatorach 40 bitowych. Wynik operacji odsyłanej dalej zwykle mieści się na starszej części akumulatora (AH/BH). Najstarsza część – bity ochronne – (AG/BG) stanowią rezerwę dla sumowania wyników pośrednich operacji a część młodsza (AL/BL) ma zapewnić odpowiednią dokładność obliczeń wyników pośrednich akumulatorze.

W odbieranym wyniku z 16-to bitowej części starszej akumulatora AH można uwzględnić końcówkę wyniku zawartą w części młodszej AL właśnie poprzez użycie wbudowanego w procesor mechnizmu zaokrąglania wyniku. W praktyce oznacza to dodanie do akumulatora wartości 0x00.0000.8000, dzięki czemu jeśli zawartość części AL akumulatora przekracza ½ LSB części AH akumulatora wówczas jego zawartość zostanie zaokrąglona.

Mechanizm ten uruchamiamy specjalizowanym rozkazem RND albo odpowiedni modyfikowanymi rozkazami operacji arytmetycznych np. MACR, MPYR, LDR itp. Zaokrąglenie np. w obliczeniach pętli filtrów stosuje się zazwyczaj dla wyniku ostatniej operacji.

Co w procesorach określa pojęcie rozszerzenia znakowego i dlaczego jest ono tak istotne w DSP?

Procesory sygnałowe dla zachowania odpowiedniej precyzji obliczeń zwykle dysponują akumulatorami co najmniej dwukrotnie większymi od rozmiaru słowa, którym pracują. W przypadku rodziny procesorów 'C5000 pracującej na słowie 16-to bitowym akumulatory mają rozmiar 40-to bitowy.

Rozszerzenie znakowe to mechanizm pozwalający procesorowi w takiej sytuacji na zachowanie znaku danej ładowanej do większego rejestru. Operacja ta realizowana jest automatycznie i może być włączana za pomocą bitu SXM – Sign eXtention Mode – umieszczonego w rejestrze statusowym ST1.

Zasada działania SXM jest następująca:

SXM = 1 → liczby ujemne dopełniane są na starszych bitach jedynkami, zaś dodatnie zerami.

SXM = 0 → brak dopełnienia znakowego, starsze bity pozostają zwykle bez zmian.

Obsługa rozszerzenia znakowego wygląda następująco:

SSBX SXM ;sign-extension mode ON

RSBX SXM ;sign-extension mode OFF

Co to jest Saturation on Store (SST)?

Saturation on Store (SST) - jest to operacja nasycania wyniku przy zapamiętaniu. Włączana jest i wyłączana za pośrednictwem bitu SST w rejestrze statusowym PMST (PMST.0). Gdy SST=1, włączone jest nasycanie wartości z akumulatora przed odesłaniem do pamięci. Nasycanie jest wykonywane po operacji przesunięcia (jeśli rozkaz tego wymaga). Trzeba jednak podkreślić, że odsyłając do pamięci „nasyconą” zawartość nie nasycamy zawartości akumulatora!. Podczas użycia SST wykonywane są zatem następujące operacje:

- 40-bitowa wartość jest przesuwana (w prawo lub lewo) w zależności od instrukcji).
- 40-bitowa wartość jest nasycana do wartości 32-bitowej a sposób nasycenia zależy od bitu SXM.
 - Jeśli SXM = 0, generowana jest następująca 32-bitowa wartość:
 - FFFF FFFFh, jeśli wartość jest większa niż FFFF FFFFh
 - Jeśli SXM = 1, generowana jest następująca 32-bitowa wartość:

- 7FFF FFFFh, jeżeli wartość jest większa niż 7FFF FFFFh
- 8000 0000h, jeżeli wartość jest mniejsza niż 8000 0000h
- Otrzymana zawartość jest przesyłana do pamięci w sposób zależny od instrukcji
- Ważne jest, że wszystkie te operacje na zawartości akumulatora są tylko tymczasowe dla przygotowania wartości do zachowania w pamięci. Zawartość akumulatora po zakończeniu operacji pozostaje taka jak na początku wykonywania rozkazu (niezmieniona), podobnie jak OVx.

Zatem jest to coś w rodzaju ograniczenia napięcia zasilania w układach analogowych, które nie pozwala na wyjście napięciem wyższym, ponad poziom zasilania.

Co to jest Overflow Mode i co zmienia w pracy procesora jego włączenie?

Overflow Mode jest trybem nadzoru przepełnienia zakresu. Włącza się go / wyłącza poprzez modyfikację bitu OVM znajdującym się w rejestrze ST1 (ST1.9). OVM determinuje, jaka wartość jest zawartość akumulatora gdy dojdzie do przepełnienia:

gdy OVM = 0, wyniki w akumulatorze nie podlegają ograniczaniu. Wyniki obejmują wszystkie 40 bitów w akumulatorze

gdy OVM = 1, wyniki w akumulatorze są ograniczane pomiędzy wartościami maksymalnymi; dodatnią 0x00.7FFF.FFFF i ujemną 0x00.8000.0000, nie dopuszczając do przekroczenia zakresu. W związku z tym wynik obliczeń nie przekracza 32-bitów przy przekroczeniu zakresu.

Jak włącza się w procesorach DSP tryb Overflow Mode?

Włączenie to można wykonać kilkoma sposobami.

Albo poprzez ustawienie indywidualnego bitu OVM:

SSBX OVM

Odpowiednio, bit OVM jest kasowany (ustawiony na 0) przez:

RSBX OVM

Albo poprzez ustawianie całej zawartości rejestru statusowego

ORM 0x0200, 7 ; OVM=1

lub jego skasowania

ANDM 0x0FDFF, 7 ; OVM=0

Czym się różni przepełnienie od nasycenia?

Przy nasyceniu podczas przekroczenia zakresu w akumulatorze ustawiana jest maksymalna lub minimalna możliwa wartość, natomiast przepełnienie powoduje „przekręcenie” się wartości, polegające na tym, że bardzo duża wartość dodatnia może się stać bardzo dużą wartością ujemną, i odwrotnie.

Co to jest i jak realizowana adresacja z odwróceniem bitowym (BRA)?

Jest to sposób adresowania przeznaczony do przyspieszenia obliczeń programu transformat wykorzystujących $\sin()$ i $\cos()$ jako funkcje bazowe. BRA bazując na symetrii tych funkcji pozwala na przyspieszenie adresowania w buforach danych lub/i współczynników (zależnie od wariantu realizacji). Procesorowi należy przekazać informację o rozmiarze bufora zapisując ją lub liczbę z niej wynikającą do wskazanego rejestru. (w przypadku 'C5402 wpisujemy liczbę równą połowie rozmiaru bufora obsługiwanego tą adresacją do AR0). Poniżej w tabeli objaśnienie realizacji BRA.

	Licz (dec)	Licz (bin)	BRA (bin)	BRA (dec)	„Motylki“
wiersz	1	2	3	4	5
1	0	000	000	0	M0
2	1	001	100	4	
3	2	010	010	2	M2
4	3	011	110	6	
5	4	100	001	1	M1
6	5	101	101	5	
7	6	110	011	3	M3
8	7	111	111	7	

W trakcie adresacji powiększamy licznik a następnie odwracamy symetrycznie bity w zakresie potrzebnym do adresacji w buforze FFT (tutaj 8 lokacji zatem adresacja na 3 bitach). Połowę liczby wielkości bufora (#4) wpisujemy do AR0. Szczegóły można odnaleźć w przytoczonym dalej przykładzie programu do eksperymentowania z tym rodzajem adresowania.

Mechanizm;

- Odwrócenie bitów w wyznaczonym zakresie ==> do kol. 3
- Zwiększenie licznika z kol.1 o 1
- Odwrócenie bitów w wyznaczonym zakresie ==> do kol. 3
- Zwiększenie licznika z kol.1 o 1
- itd. ... do końca bufora

Proszę dostrzec, że zawartość kolumny 5 demonstruje „uczestników” kolejnych motylków ilustrujących obliczenia transformaty w podręcznikach. Wnikliwym załączam na końcu opisu zależności dla 8-mio punktowej FFT

Formuła programowa wymaga zainicjowania poza rejestrem adresującym również rejestru AR0. Poniżej fragment programu dodemonstracji tego mechanizmu;

```
; Demonstracja BRA - Bit Reversal Addressing
; =====
;          .global start, bufor_1, bufor_2, buf_test ; dla uwidocznienia w debuggerze
;          .mmregs          ; by umożliwić użycie predefiniowanych nazw rejestrów

;          -----   buforek danych do przeniesienia i sprawdzenia
;          kolejności adresacji z odwróceniem bitowym

bufor_1    .sect "buf_test"
           .word 00h      ; kolejne cyfry od 0 do 7 w kolejnych komórkach
           .word 01h
           .word 02h
           .word 03h
           .word 04h
           .word 05h
           .word 06h
           .word 07h

;          -----   bufor_2 - miejsce na przepisanie w zmienionej
;          kolejności po adresacji z odwróceniem bitowym
;          Przetasowane dane znajdują się w kolejnych 8 komórkach

bufor_2    .usect  "bufor_2",8      ; rezerwacja miejsca dla d."przemieszanych"

           .text
start
;
; -----   test adresacji z odwróceniem bitowym
           STM    #bufor_1,AR3      ; rejestr do adresacji BRA
           STM    #bufor_2,AR1      ; rejestr docelowy
           STM    #4,AR0             ; wielkosc bufora FFT/2

           NOP
```

LD	*AR3+0B,A	;0	- pobieranie danej w adresacji BRA
STL	A,*AR1+		; zachowanie danej w buforze wyniku
LD	*AR3+0B,A	;1	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;2	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;3	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;4	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;5	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;6	-"
STL	A,*AR1+		
LD	*AR3+0B,A	;7	-"
STL	A,*AR1+		
NOP			; bufor wynikowego tasowania pelen

Rozpisanie zależności dla FFT 8-mio punktowej N=8

$$X(m) = \sum_{n=0}^{N-1} x(n) * e^{-j2\pi * n * m / N} = \sum_{n=0}^7 x(n) * e^{-j2\pi * n * m / 8} =$$

$$\begin{aligned} X(m) &= \sum_{n=0}^{(N/2)-1} x(2n) * e^{-j2\pi * (2n) * m / N} + \sum_{n=0}^{(N/2)-1} x(2n+1) * e^{-j2\pi * (2n+1) / N} = \\ &= \sum_{n=0}^3 x(2n) * e^{-j2\pi * (2n) * m / 8} + \sum_{n=0}^3 x(2n+1) * e^{-j2\pi * (2n+1) * m / 8} = \end{aligned}$$

$$W_N^k = e^{-j2\pi k / N} \quad W_N^2 = e^{-j2\pi 2 / N} = e^{-j2\pi / (N/2)} \quad W_N^2 = W_{N/2}^1$$

$$X(m) = \sum_{n=0}^3 x(2n) * W_8^{2mn} + W_8^m * \sum_{n=0}^3 x(2n+1) * W_8^{2mn} = \sum_{n=0}^3 x(2n) * W_4^{mn} + W_8^m * \sum_{n=0}^3 x(2n+1) * W_4^{mn} =$$

$$X(m) = \sum_{n=0}^3 x(2n) * W_4^{mn} + W_8^m * \sum_{n=0}^3 x(2n+1) * W_4^{mn}$$

$$X(m+4) = \sum_{n=0}^3 x(2n) * W_4^{mn} - W_8^m * \sum_{n=0}^3 x(2n+1) * W_4^{mn}$$

$$W_N^0 = 1 \quad W_N^{n/N} = -1 \quad W_N^m W_{N/2}^m = W_N^{3m} \quad W_N^m W_{N/2}^m W_{N/4}^m = W_N^{7m}$$

$$X(m) = x(0) + x(2)W_4^m + x(4)W_4^{2m} + x(6)W_4^{3m} + W_8^m [x(1) + x(3)W_4^m + x(5)W_4^{2m} + x(7)W_4^{3m}]$$

$$X(m) = x(0) + W_2^m x(4) + W_4^m x(2) + W_4^{3m} x(6) + W_8^m x(1) + W_8^{5m} x(5) + W_8^{3m} x(3) + W_8^{7m} x(7)$$

$$\begin{aligned} X(m) &= x(0) + W_8^{4m} x(4) + W_8^{2m} x(2) + W_8^{6m} x(6) + W_8^m x(1) + W_8^{5m} x(5) + W_8^{3m} x(3) + W_8^{7m} x(7) \\ &\quad | \text{--- M0 ---} | | \text{--- M2 ---} | | \text{--- M1 ---} | | \text{--- M3 ---} | \end{aligned}$$