Kompliator C/C++

Sekcja 14

Cechy kompilatora C ADSP-211xx

•Kompiluje pliki źródłowe C i C++ zgodne ze standardem ANSI/ISO
  - obsługuje proponowany standard Zagnieżdzone C++
  - definiuje podzbiór pełnego standardu C++ ISO/IEC 14882:1998
•Obsługuje C-Rozszeżenia specyficzne dla architektury 21K
•Zawiera bibliotekę funkcji ANSI, matematycznych, DSP
•Kompilator może być użyty do generowania plików wykonywalnych
  (format elf) z kodu C, asemblera i plików obiektów
•Debugger C i C mieszanego z Asemblerem na poziomi kodu źrodłowego
•Obsługuje interpreter asemblera in-line
•Obsługuje interfejs ze źródłem asemblera

Programowanie w C dla DSP

Czemu pisać w C?
- ‘podtrzymywalny’ kod
- przenośny kod
- ‘learning curve’ – prościej się uczyć

Jakie są wady pisania w C?
- wydajność kodu (rozmiar, ilość cykli)
- wolniejsza obsługa przerwań

Optymalną metodą jest mieszanie kodu C z assemblerem

Kompliator C ADSP-211xx

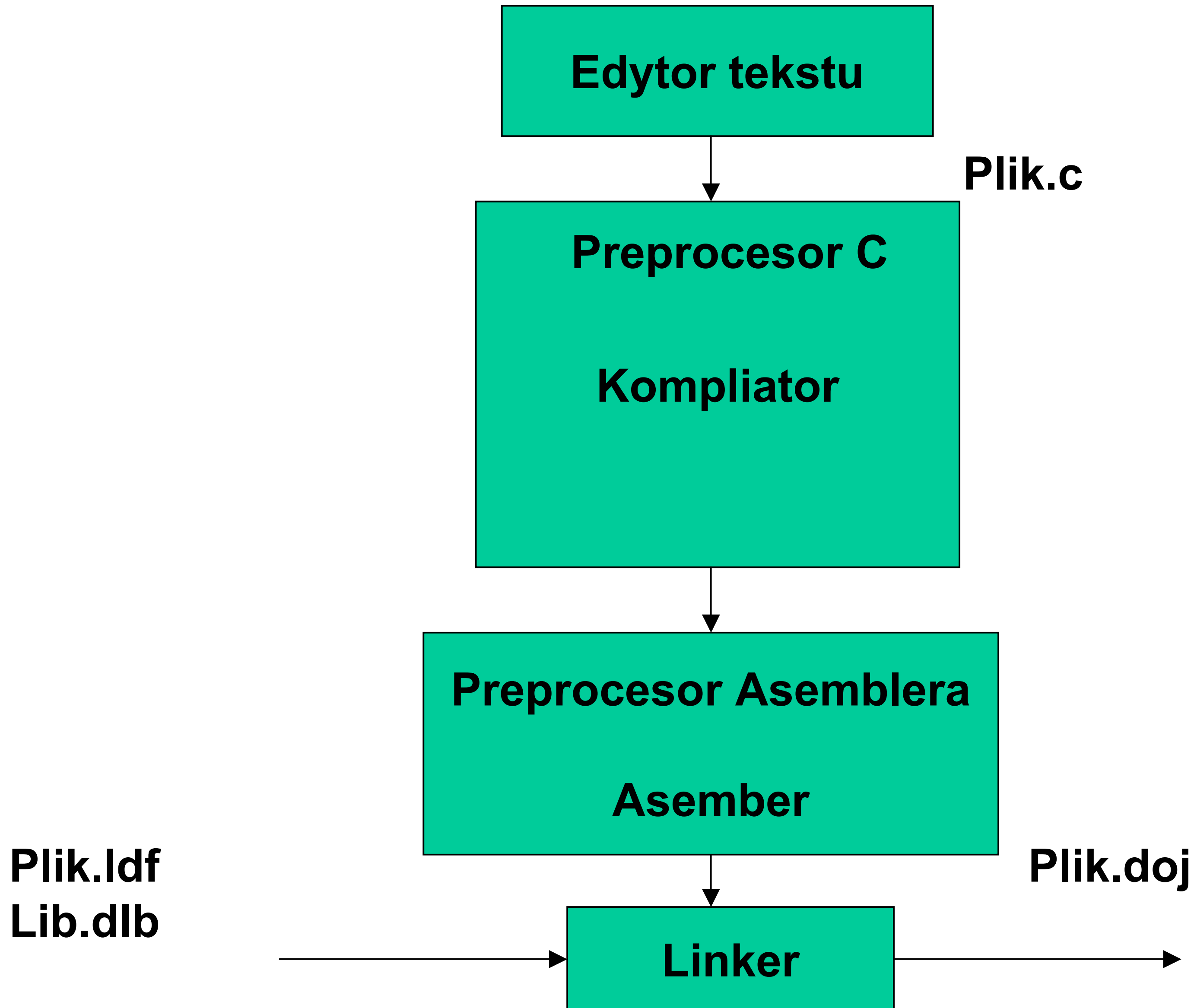
Kompliator
- wywoływany przez IDDE (wpisany w ustawieniach)
- wywoływany z poziomu DOS (cc21k.exe)

Plik linkera (LDF – Linker description file)
- określa segmenty w pamięci dla kodu i danych
- określa segment w pamięci na stos (do przekazywania parametrów przy wywoływaniu przerwania)
- określa segment w pamięci na heap (malloc)

Run Time Header (161_hdr.doj):
- zapewnia obsługę przerwań
- inicjalizuje środowisko C
- musi być zlinkowany razem z kodem C

Interfejs C/Assembler
- Prolog/Epilog

Proces ‘budowania’ (Build)



Ustawienia kompliatora

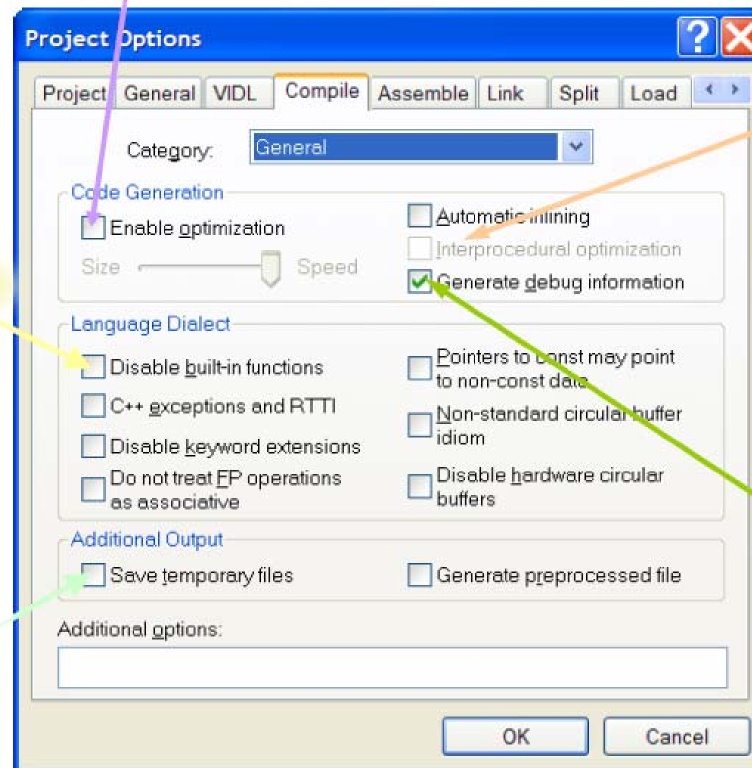
Compile/General Property Page
− włączenie debugowania i optymalizacji
− Zatrzymaj po : preprocessingu, kompliowaniu
• Compile/Preprocessor Property Page
− podaje definicje procesora
− określa ścieżki do dołączania plików
• Compile/Warning Property Page
− włącza poziomy ostrzeżeń
• Compile/DSP specific Property Page
− Określa zarezerwowane rejestry
− Wybiera prezycję 32 lub 64 bit

Ustawienia kompliatora

Odpowiada –O. Pozwala kompliatorowi optymalizować kod aby zwiększyć szybkość działania programu

Odpowiednik (-ipa). Pozwala kompliatorowi zoptymalizować kod całosciowo (na podstawie wszystkich plików żrodłowych), a nie tylko wewnątrz danej jednostki translacyjnej.

Odpowiada –no-builtins. Zezwala na użycie tylko funkcji ANSI

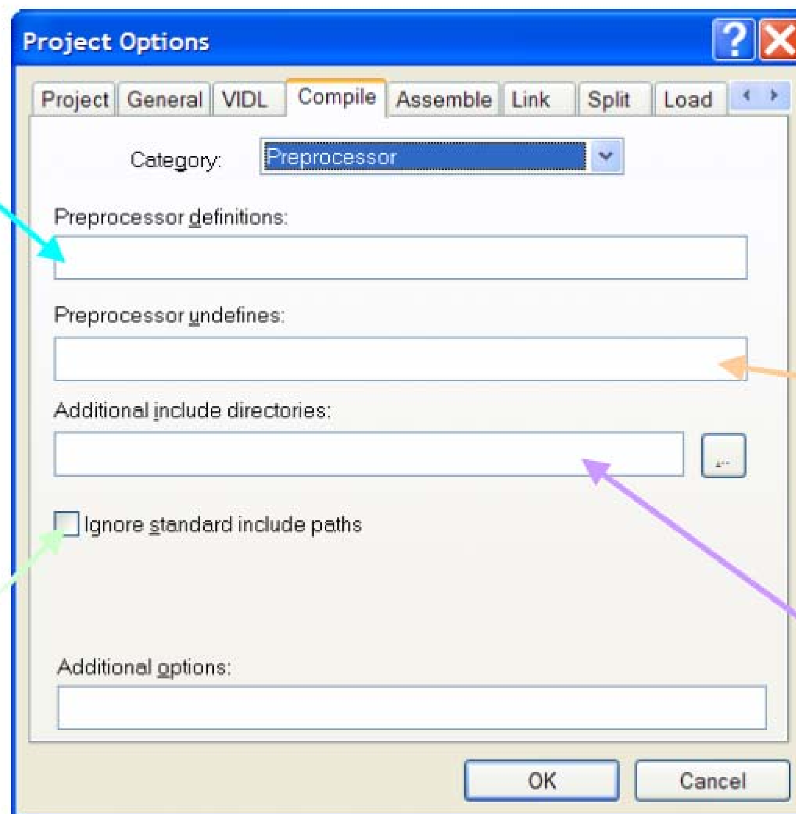


Generuje informacje do Debuggowania (DWARF-2). Pozwala użytkownikom u Ustawiać breakpointy i Debuggować (-o)

Odpowiada –save-temps. Powoduje że kompliator Nie kasuje plików tymczasowych

Ustawienia kompliatora: preprocessor

Tu dodaj makra, które chcesz zdefiniować. Oddziel je przecinkami. Odpowiednik ‘-D’



Tu wpisz makra, które chcesz usunąć. Oddziel je przecinkami. Odpowiednik ‘-U’

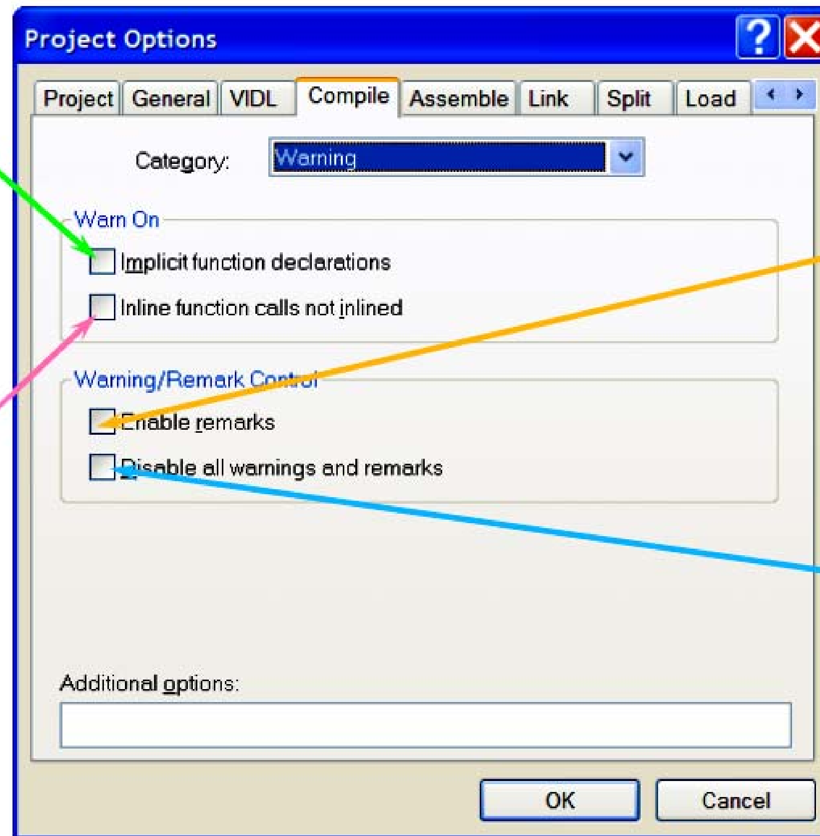
Powoduje że preprocesor Ignoruje standardowe ścieżki include przy szukaniu plików. Odpowiednik –no-std-inc

Tu wpisz ścieżki w których preprocessor ma szukać plików include. Oddziel Je przecinkami. Odp ‘-I’.

Ustawienia kompliatora: ostrzeżenia

Ostrzega przy niejawnych deklaracjach funkcji – np. gdy funkcja nie ma globalnego prototypu ale jest zdefiniowana przed wywołaniem. Odpowiada -flags-complier

Ostrzega kiedy kompliator nie może wygenerować kodu 'inline' dla funkcji która została zdefiniowana ze słowem Kluczowym inline. Odpowiada -no-inline



Włącza wyświetlanie Uwag kompliatora, o znaczeniu diagnostycznym (mniej ważne od ostrzeżeń) Odpowiada -Wremarks

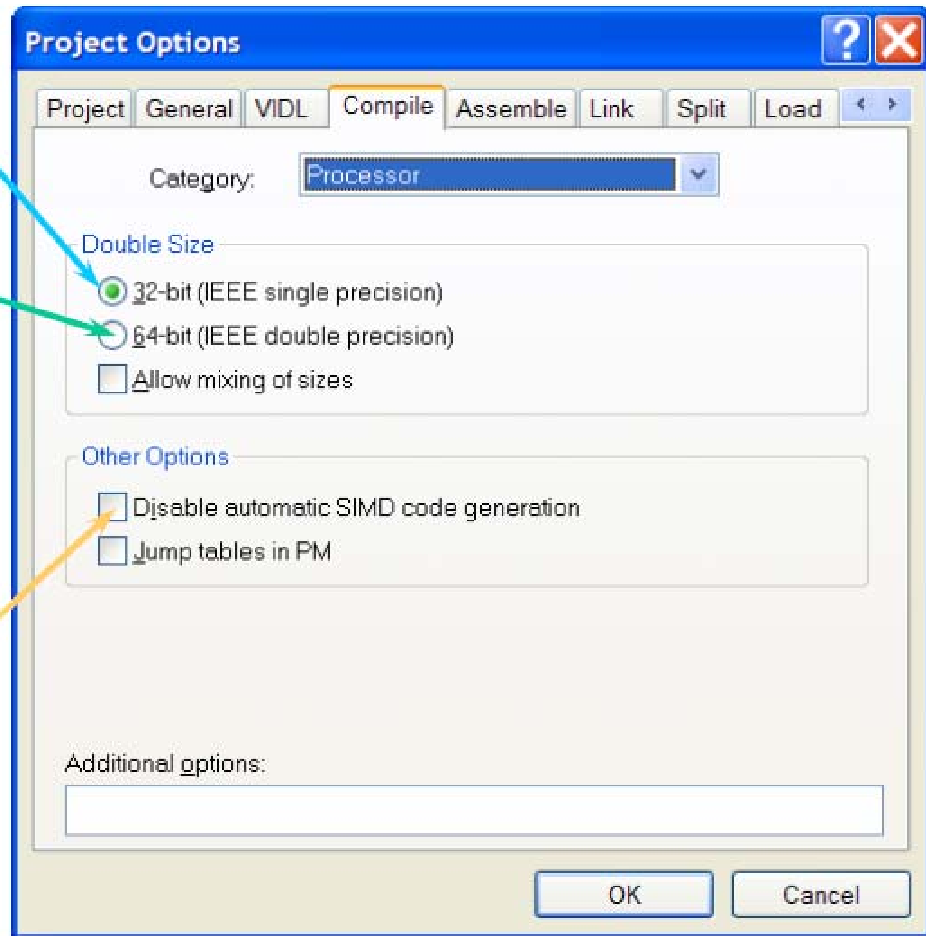
Wyłącza wyświetlanie ostrzeżeń. Będzie Raportować tylko błędy. Odpowiada –w.

Ustawienia kompliatora: DSP

Ustawia double-size
jako 32bit. Odpowiada
-double-size-64

Ustawia double-size
jako 64bit. Kompliator używa
Wolniejszych funkcji aby
zapewnić obsługę 64bit.
Odpowiada -double-size-64

Wyłącza automatyczne
Generowanie kodu SIMD.
Można je wymusić za
Pomocą ‘SIMD_for’.



Obsługiwane formaty danych:

Type	Bit Size	Result of sizeof operator
`int`	32 bits signed	1
`unsigned int`	32 bits unsigned	1
`long`	32 bits signed	1
`unsigned long`	32 bits unsigned	1
`char`	32 bits signed	1
`unsigned char`	32 bits unsigned	1
`short`	32 bits signed	1
`unsigned short`	32 bits unsigned	1
`pointer`	32 bits	1
`float`	32 bits float	1
`fract`	32 bits fixed-point	1
`double`	either 32 or 64 bits float (default 32)	either 1 or 2 (default 1)
`long double`	64 bits float	2

LDF (Linker Description File) dla programów C

Opis pamięci
- określa segmenty pamięci
- mapuje sekcje wejściowe (nazwy produkowane przez kompliator) odpowiednim segmentom pamięci

Obsługa stosu
- rozmiar i adres stosu
- położenie stosu

Obsługa heap
- Rozmiar, położenie i nazwa
- lokalizacja

Opisy pamięci

Określa segmenty pamięci

- **nagłówek runtime (lokacja tablicy wektorów przerwań)**
- **kod inicjalizujący**
- **kod**
- **dane**
- *** stos**
- **heap (możliwe wiele obszarów heap)**

Mapuje sekcje wejściowe segmentom pamięci

Nazwa sekcji	**Użycie**
seg_rth	Nagłówek runtime (161_hdr.obj)
seg_init	Inicjalizacja obszaru danych
seg_pmco	Inicjalizacja obszaru kodu
seg_dmda	Zmienne globalne, ciągi znakowe, stałe
seg_pmda	Zmienne

Sekcja pamięci pliku LDF:

```
MEMORY
{
        seg_rth { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48) }
        seg_init { TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48) }
        b0_code { TYPE(PM RAM) START(0x00040200) END(0x000419ff) WIDTH(48) }
        b0_data { TYPE(PM RAM) START(0x00042a00) END(0x00043fff) WIDTH(32) }
        b1_data{ TYPE(DM RAM) START(0x00050000) END(0x00051fff) WIDTH(32) }
        seg_heap { TYPE(DM RAM) START(0x00052000) END(0x00052fff) WIDTH(32) }
        seg_stak { TYPE(DM RAM) START(0x00053000) END(0x00053fff) WIDTH(32) }
}
Processor p0
{
OUTPUT( test1.dxe)
LINK_AGAINST($COMMAND_LINE_LINK_AGAINST)
Section
    { run_time_header {INPUT_SECTIONS($OBJECTS(seg_rth)$LIBRARIES(seg_rth))} > seg_rth
    seg_init {INPUT_SECTIONS($LIBRARIES(seg_init))} > seg_init
    pm_code {INPUT_SECTIONS($OBJECTS(seg_pmco)$LIBRARIES(seg_pmco))} > b0_code
    pm_data {INPUT_SECTIONS($OBJECTS(seg_pmda)$LIBRARIES(seg_pmda))} > b0_data
    dm_data {INPUT_SECTIONS($OBJECTS(seg_dmda)$LIBRARIES(data))} > b1_data }
}
```

Przykładowy program C:

```
// Simple dot product example
float dot(float* x, float* y, int n);
   float dot(float* x, float* y, int n)
   {
      int i;
      float sum=0;
      for(i=0;i<n;i++)
      {
         sum += x[i]*y[i];
      }
      return(sum);
   }
```

Kod assemblera stworzony przez kompliator:

Analog Devices VisualDSP++ - [Target: ADSP-21161 ADSP-2116x Simulator] - [Project: testc.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

dot.s

```
    .file ".\dot.c";

    .section/pm seg_pmco;

    // brought to you by a Waveguide/Bril compiler
.epctext:

_dot:
// line ".\dot.c":5
        modify(i7,-4);
// line 10
        r2=pass r12, i12=r8;
        if le jump(pc,.P1L1) (DB);
// line 5
        i4=r4;
        r8=m13;
        r12=r8+1, r8=pm(i12,m14);
// line 10
        r1=max(r12, r2), r2=dm(i4,m6), r4=pm(i12,m14);
// line 12
        F12=F2*F8, r2=dm(i4,m6);
        r8=m13;
// line 10
        lcntr=r1, do(pc,.P1L4-1)until lce;

.P1L3:
        F12=F2*F4, F8=F8+F12, r2=dm(i4,m6), r4=pm(i12,m14);
// line 12
        //end loop .P1L3;

.P1L4:

.P1L1:
// line 15
        r0=pass r8, i12=dm(m7,i6);
        jump(m14,i12)(DB); rframe; nop;

._dot.end:
    .global _dot;
    .type _dot,STT_FUNC;
```

dot.c

```
float dot(float* x, pm float* y, int n);

float dot(float* x, pm float* y, int n)
{
    int i;
    float sum=0;

    for(i=0;i<n;i++)
    {
        sum += x[i]*y[i];
    }

    return(sum);
}
```

```
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c ".\dot.c"  -O1 -save-temps  -proc ADSP-21161  -o ".\Debug\dot.doj"
".\dot.c", line 12: warning: loop would vectorise if alignment were known -
          consider inserting SIMD_for pragma if all memory accesses are
          aligned on even-word boundaries
Build completed successfully.
```

Console Build

Ready | Halted | Line 12, Col 26

Słowo kluczowe KEYWORD

Każe kompliatorowi umieścić zmienną lub funkcję w sekcji Innej niż domyślna stosowana przez kompliator

Przykład

```
static section("data2") int x;
```

Sekcja obiektu – później zmapujemy ją do segmentu pamięci w pliku LDF

Proces budowania oprogramoania

Etap 1, przykład: źródło C z różnymi sekcjami

foo.C

```
section ("extern") int array[256];

section ("foo") void bar(void)
{
   int foovar;
   foovar = 1;
   foovar ++;
}
```

C-Compiler → Assembler →

foo.DOJ

```
Object Section = extern
Type           = DM
Width          = 32
....................................
_array [00]
_array [01]
  ...
_array [256]
```

```
Object Section = foo
Type           = PM
Width          = 48
....................................
 _bar :
 r0 = dm(_foovar);
 r0 = r0 + 1;
```

```
Object Section = seg_stak
Type           = DM
Width          = 32
....................................

 _foovar: 1
```

Stos:

32 bitowa struktura rosnąca w pamięci od wyższych do niższych adresów.

Zarządzana przez wskaźnik ramki I6 oraz wskaźnik stosu I7
- **wskaźnik ramki wskazuje adres początku ramki**
- **wskaźnik stosu wskazuje następną wolną lokację na stosie**

Ramka stosu zawiera
- **zmienne lokalne**
- **zmienne tymczasowe**
- **parametry funkcji**

LDF i stos

System uruchomieniowy „C" zależy od inicjalizacji wskaźnika Ramki, I6 oraz wskaźnika stosu, I7

Wartości te są inicjalizowane stałymi zdefiniowanymi w pliku opisu linkera (LDF).

ldf_stack_space
ldf_stack_length

Zmienne użyte do inicjalizacji wskaźników ramki i stosu są Zdefiniowane i inicjalizowane w pliku assemblera seg_init.asm

__lib_stack_space
__lib_stack_lenght
__inits

LDF – ustawienie stosu (tylko kompilator C)

Linker przelicza stałe inicjalizujące stos LDF z deklaracji stosu

```
stackseg {
ldf_stack_space = . ;
ldf_stack_length = MEMORY_SIZEOF(seg_stak);
} >
```

Ta sekcja musi być dołączona w części SECTIONS() pliku LDF Jeśli programujemy w C

LDF i Heap

Cztery funkcję biblioteki są używane do alokowania i zwalniania pamięci z obszaru Heap.
- malloc, calloc, realloc, free

Seg_init.asm zawiera zmienne inicjalizowane stałymi określonymi w pliku LDF.
- ldf_heap_space
- ldf_heap_length
- ldf_heap_end

Wiele obszarów heap jest dopuszczalnych (wymaga to dodania kodu do LDF oraz seg_init.asm)

LDF i heap
(tylko kompilator C)

Sekcja wyjściowa 'heap' wylicza wartości inicjalizujące heap z deklaracji segmentów heap

```
heap {
ldf_heap_space = .;
ldf_heap_length = MEMORY_SIZEOF(seg_heap);
ldf_heap_end = ldf_heap_space+ ldf_heap_length-1
} > seg_heap
```

Sekcja ta musi być umieszczona w części 'Section' pliku LDF jeśli programujemy w C

Dla każdego obszaru heap ten kod musi być skopiowany.

Nagłówek Run Time (RTH)

RTH w połączeniu z funkcją seg_init() ustawia środowisko C:
- inicjalizuje stos
- inicjalizuje heap
- inicjalizuje rejestry

RTH w połączeniu z funkcją C signalxx(), interruptxx() zapewnia wsparcie obsługi przerwań:
- włączenie przerwań
- zachowanie/odtworzenie kontekstu
- serwis (?) przerwań

RTH umieszczany jest w położeniu wektora przerwań i może być modyfikowany przez programistę (161_hdr.asm)

Stos Run Time
Seg_init.asm

SEG_INIT.ASM

- ustawia stos Run Time (tworzy zmienne zawierające długość i położenie stosu)
- ustawia heap
- Używa stałych zdefiniowanych w pliku LDF (patrz poprzedni slajd)

```
. segment/pm seg_init;
.global ___lib_stack_space;
.var ___lib_stack_space = ldf_stack_space;
.global ___lib_stack_length;
.var ___lib_stack_length = ldf_stack_length;
.global ___inits;
.var ___inits = 0x0;
```

SEG_INIT.ASM jest linkowany do archiwum libc.dlb

RTH i przerwania.

Kod mieści się w tabeli wektorów przerwań (IVT)

Dispatcher

- **interruptxx() gdzie xx to blank, cb, f, s, lub ss**
- **włącza przerwania**
- **przypisuje wybrane ISR każdemu przerwaniu**
- **zapisuje i odtwarza kontekst procesora (nie dotyczy ss)**

Zmień RTH dla szybkiej obsługi przerwań

Włączanie przerwań

Przerwania włącza się w trakcie działania procesora, za pomocą funkcji biblioteki jak poniżej

```
interruptcb(sig, isr_name);
interrupt(sig, isr_name);
interruptf(sig, isr_name);
interrupts(sig, isr_name);
interruptss(sig, isr_name);
```

Przykład: `interrupts(SIG_TMZI, MyTimerISR);`

Funkcja `interruptxx()` **realizuje:**

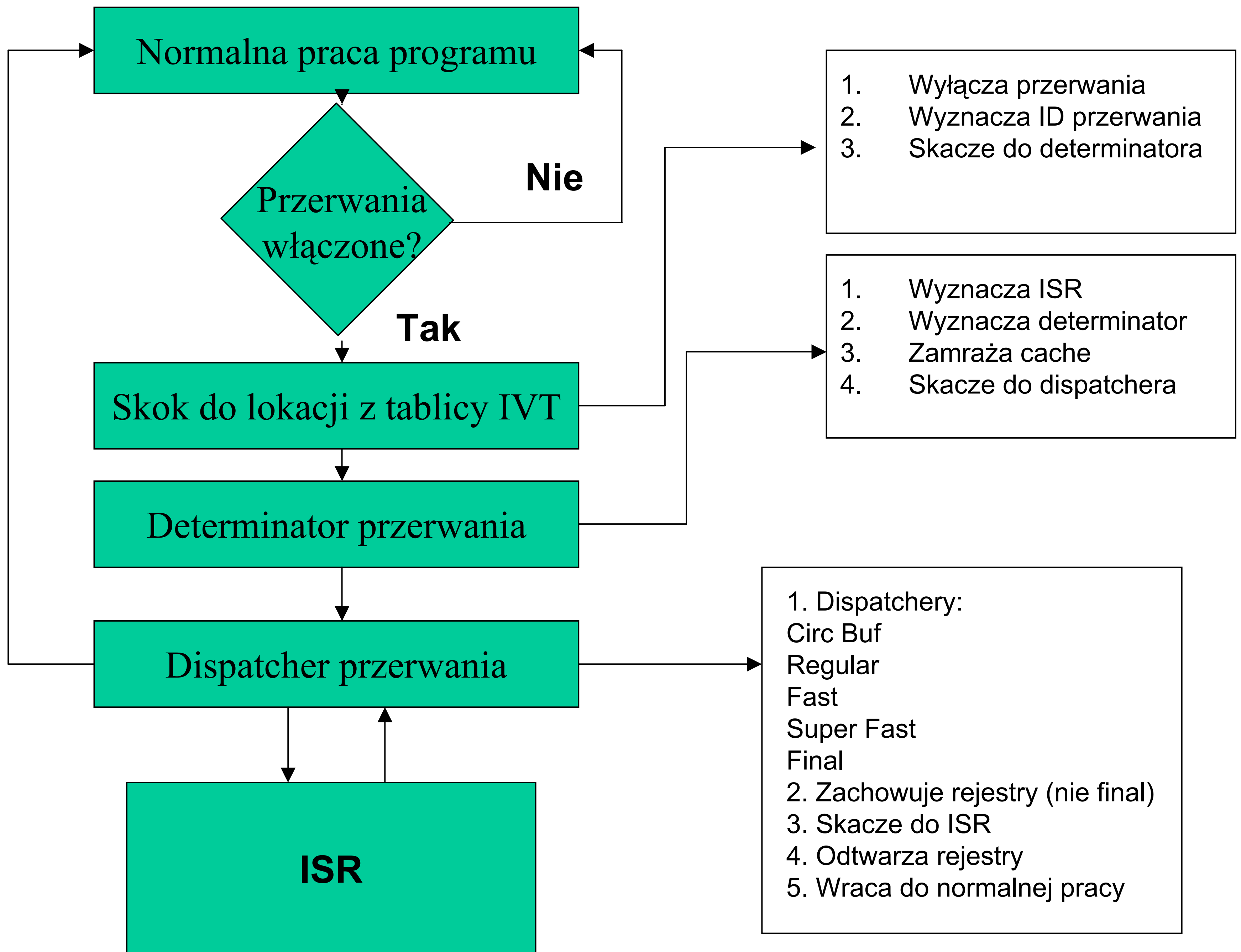
- włączenie przerwań w rejestrze MODE1
- ustawienie odpowiedniego bitu w rejestrze IMASK
- mapuje odpowiednią funkcję (isr_name) jako procedure obsługi przerwania

`signalxx(sig,isr_name);` **włącza jedną instancję przerwania.**

Zachowanie/odtworzenie stanu procesora

Przed wywołaniem ISR, dispatcher zachowuje stan procesora.
Po wywołaniu ISR, dispatcher odtwarza stan procesora.

Nazwa	liczba cykli	funkcja C	co jest zachowane?
Circular Buffer	332	Interruptcb()	Zachowuje i odtwarza wszystko, łącznie z licznikiem pętli, zeruje Rejestr długości
Regular	292	Interrupt()	Tak jak powyżej, bez zerowania.
Fast	66	Interruptf()	Zachowuje rejestry tymczasowe.
Super fast	44	Interrupts()	Używany jest drugi zestaw Rejestrów. Zagnieżczanie przerwań Wyłączone
Final	39	Interruptss()	Bazuje na kodzie obsługi przerwania do zachowania/odtworzenia kontekstu. Zagnieżdzanie przerwań dozwolone.


Normalna praca programu
Przerwania włączone?
Nie
Tak
Skok do lokacji z tablicy IVT
Determinator przerwania
Dispatcher przerwania
ISR
1. Wyłącza przerwania
2. Wyznacza ID przerwania
3. Skacze do determinatora
1. Wyznacza ISR
2. Wyznacza determinator
3. Zamraża cache
4. Skacze do dispatchera
1. Dispatchery:
Circ Buf
Regular
Fast
Super Fast
Final
2. Zachowuje rejestry (nie final)
3. Skacze do ISR
4. Odtwarza rejestry
5. Wraca do normalnej pracy

Interfejs Assemblera

- Funkcje assemblera wywoływalne z poziomu C

- Komendy assemblera wywoływane z wewnątrz funkcji (assembler in-line)

- Łączenie zmiennych C z symbolami assemblera

Funkcje asseblera wywoływalne z poziomu C

Jest szereg problemów związanych z pisaniem procedur Assemblera wywoływalnych z C, które należy poznać.

- Użycie rejestrów

- **Rejestry „Dedicated” (dedykowane)**
- **Rejestry „Call preserved”**
- **Rejestry „Scratch”**
- **Rejestry „User”**

- Przekazywanie parametrów

Przekazywanie parametrów

Rejestr	Typ przekazywanego lub zwracanego parametru
r4	Przekazuje pierwszy 32-bitowy parametr
r8	Przekazuje drugi 32-bitowy parametr
r12	Przekazuje trzeci 32-bitowy parametr
stack(stos)	Przekazuje czwarty i pozostałe parametry
r0	Zwraca parametry int, long, char, float, shor, pointer
r0,r1	Zwraca long double i struktury dwusłowowe. Umieszcza MSW w r0 i LSW w r1
r1	Zwraca adres wyników większych niż dwa słowa. r1 zawiera adres pierwszego słowa rezultatu

Rejestry dedykowane

Zestaw rejestrów które kompliator C rezerwuje do swojego użytku

Rejestr	Wartość	Reguły zmiany
m5, m13	0	Nie zmieniać
m6,m14	1	Nie zmieniać
m7,m15	-1	Nie zmieniać
b6,b7	Baza stosu	Nie zmieniać
l6,l7	Długość stosu	Nie zmieniać
l0,l1, l2, l3, l4, l5, l8, l9, l10, l11, l12, l13, l14, l15	0	Zmieniać tymczasowo, odtwarzać po skończeniu

Rejestry zachowywane przy skokach

Mogą być używane w funkcjach assemblera, ale ich wartości powinny zostać zachowane i odtworzone – zakłada się że ich wartości są niezmienne pomiędzy wywołaniami funkcji

b0	b1	b2	b3	b5	b8
b9	b10	b11	b14	b15	
i0	i1	i2	i3	i5	i8
i9	i10	i11	i14	i15	mode1
mode2	mrb	mrf	m0	m1	m2
m3	m8	m9	m10	m11	r3
r5	r6	r7	r9	r10	r11
r13	r14	r15			

Rejestry „Scratch” (tymczasowe)

ich zwartość NIE MUSI być zachowywana – można używać swobodnie w procedurach asemblera

b4	b12	b13	r0	r1	r2	r4	r8	r12
i4	i12	m4	m12	i13	PX	USTAT1	USTAT2	

Rejestry użytkownika

Rejestry te mogą zostać zarezerowane jeśli podamy kompliatorowi parametr -reserve

Rejestr	Wartość	Reguły zmieniania
i0, b0, l0, m0, i1, b1, l1, m1, i8, b8, l8, m8, i9, b9, l9, m9, mrb, ustat1, ustat2, ustat3, ustat4	zdefiniowana przez użytkownika	Jesli nie zarezerwowane, można zmieniać tymczasowo, odtwarzać po zakończeniu. Jesli nie zarezerwowane, użycie nie jest limitowane

Funkcje assemblera wywoływane z poziomu C

Dane są marka w pliku asm_sprt.h, które ułatwiają wywoływanie funkcji.
-zachowanie/odtworzenie rejestrów (puts, reads)
-Przekazanie parametrow na/ze stosu (reads)
-odtworzenie ramki i wskaźnika stosu (exit)

`PUTS =x;` (wysyła wartość rejestru x na stos)

puts=R9; -> dm(i7,m7)=R9;

`READS(n);` (czyta n-tą wartość ze stosu. Używane do przekazywania parametrów. R0=Reads(1); -> R0=dm(1,i7);

`EXIT;` (Odtwarza ramkę i wskaźnik stosu i skacze do adresu powrotu)

```
Exit ->   i12=dm(m7,i6);
          jump(m14,i12)(db);
          nop;
          rframe;
```

Assembler In-line

Bezpośrednie wywołanie komendy assemblera uzyskujemy przy pomocy komendy asm()

Example:

```
asm("bit set mode2 0x20;");
asm("ustat1 = 0x1234; )
asm("dm(IOFLAG)=ustat1;");
```

Uwaga: Kod powstały przy użyciu tej metody może być mniej wydajny - kompilator lubi zamienić kolejność instrukcji

Nazewnictwo przy mieszaniu C i ASM

Program w C	Część w asemblerze
int c_var; /*zmienna globalna*/	.extern _c_var;
void c_function();	.extern _c_func;
extern int asm_var	.global _asm_var;
extern void asm_func();	.global _asm_func; …… _asm_func:

Aby nazwać symbol assembera, odpowiadający symbolowi w c, dodaj podkreślenie przed jego nazwą. Zdefiniuj jako zmienną globalną w C i jako EXTERN w assemblerze

Aby użyć zmiennej lub funkcji assemblera w programie w C, zdefiniuij symbol za pomocą dyrektywy .GLOBAL w assemblerze i jako EXTERN w programie C

Rozszerzenia ANSI C

Typy danych PM, DM do obsługi podwójnej pamięci.

```
int pm x /* declares data in program memory */
int data_buf /* dm is default type */
```

- **INLINE włącza kod funkcji do kodu funkcji wywołującej**
- **Eliminuje starty na wywołanie funkcji**

```
inline int add_one (int *a)
{ (*a)++; }
```

- **Słowo kluczowe VOLATILE zapobiega zmianie położenia lub łączenia twoich instrukcji assemblera lub deklaracji zmiennych.**

```
volatile int x;
```

Przykład-
Dodajmy 5 liczb za pomocą assemblera

Następujący przykład to program w C który wykonuje wywołanie do funkcji z pamięci assemblera. Funkcja dodaje pięć liczb które są przekazywane jako parametry

Kod w C

```
#include <stdio.h>
extern int add5(int,int,int,int,int); /* Function is located in assembly module */
volatile int sum;  /* Variable only used in assembly sub-routine*/
                   /* volatile keeps sum from being optimized out */
main()
{
int a=1; int b=2; int c=3; int d=4; int e=5; /* Initialize parameters */
int result=0;
result = add5(a,b,c,d,e); /* Call to the ADD5 function */
exit(0);
}
```

Procedura Assemblera

```
/* Assembly Routines with Parameters Example - _add5 */
/* int add5 (int a, int b, int c, int d, int e); */
/* This is an assembly language routine that will add 5 numbers */
#include <asm_sprt.h> /* Header file that defines the stack manipulation macros */
.segment/pm seg_pmco;
.global _add5;
.extern _sum;
_add5:
r4=r4+r8; /* Add the first and second parameter */
r4=r4+r12; /* Add the third parameter */
r8=reads(1); /* Put the fourth parameter in R8 */
r4=r4+r8; /* Add the fourth parameter */
r8=reads(2); /* Put the fifth parameter in R8 */
r0=r4+r8; /* R0 is always the return value */
dm(_sum) = R0; /* Place the sum in the global variable (C is unaware of this
assignment)*/
leaf_exit; /* Restores frame and stack pointers */
.endseg;
```

Optymalizacja kodu w C

- Optymalizacja może zmniejszyć rozmiar kodu lub **przyspieszyć wykonywanie**
- **Może być włączane i wyłączane parametrem kompliatora**
  - **bez parametru – optymalizacja wyłączona**
  - **-O optymalizacja wydajności**
  - **-Os optymalizacja rozmiaru kodu**
  - **-ipa optymalizacja międzyproceduralna**
  - **-Ov num enable speed vs size optimization (sliding scale)**

(Automatically inlines small functions)

- **Może być dalej kontrolowane z poziomu kodu C**
- **#pragma optimize_off – Wyłącza optymalizcję**
- **#pragma optimize_for_space – zmniejsza rozmiar kodu**
- **#pragma optimize_for_speed – zwiększa wydajność**
- **#pragma optimize_as_cmd_line – przywraca wartości ustalone parametrami**
- Inne usprawnienia
- **PGO (Profile guided Optimization) używane z IPA**
- **Wykorzystaj gotowe funkcje assemblera z biblioteki**
- **Pisz krytyczne sekcje jako funkcje assemblera wywoływane z C**

Obsługa SIMD w C

Optymalizacja jednego / wielu kanałów

- **Ograniczone dane**
  - Tylko pojedyncze słowa **(bez 64-bit doubli czy STRUCT'ów**
  - **Granice podwójnych słów**
  - **Ciągłe**
- **Bez wywołań funkcjii wewnątrz pętli SIMD**
- **Zależności między iteracjami mogą zawieść**

```
a[j] = a[j-1] + 2; //nieprawidłowe dla SIMD
```

Obsługa SIMD w C

Kompilator będzie się starał wykorzystać SIMD wszędzie, gdzie będzie to możliwe.
`#pragma SIMD_for` może zostać użyta aby nakierować komplilator gdy pętla może być optymalizowana dla SIMD, gdy inne informacje nie są dostępne dla komplilaotra
-programista jest wtedy odpowiedzialny za prawidłowe ułożenie danych, współzależności itp.
-w ogólności, zawsze powinno się używać parametru –ipa aby minimalizować rozmiar kodu.

na przykład

```
#pragma SIMD_for
for(i=0;i<N;i++)
{
sum += x[i]*y[i];
}
```

optymalizacja ‘C’ (bez –ipa)

Analog Devices VisualDSP++ - [Target: ADSP-21161 ADSP-2116x Simulator] - [Project: testc.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

dot.s

```
    .section/pm seg_pmco;

    // brought to you by a Waveguide/Bril compiler
.epctext:

_dot:
// line ".\dot.c":5
        modify(i7,-4);
// line 10
        r2=pass r12, i12=r8;
        if le jump(pc,.P1L1) (DB);
// line 5
        i4=r4;
        r8=m13;
        r12=r8+1, r8=pm(i12,m14);
// line 10
        r1=max(r12, r2), r2=dm(i4,m6), r4=pm(i12,m14);
// line 12
        F12=F2*F8, r2=dm(i4,m6);
        r8=m13;
// line 10
        lcntr=r1, do(pc,.P1L4-1)until lce;

.P1L3:
        F12=F2*F4, F8=F8+F12, r2=dm(i4,m6), r4=pm(i12,m14);
// line 12
        //end loop .P1L3;

.P1L4:

.P1L1:
// line 15
        r0=pass r8, i12=dm(m7,i6);
        jump(m14,i12)(DB); rframe; nop;

._dot.end:
    .global _dot;
    .type _dot,STT_FUNC;

.epctext.end:
```

dot.c

```
float dot(float* x, pm float* y, int n);

float dot(float* x, pm float* y, int n)
{
    int i;
    float sum=0;

    for(i=0;i<n;i++)
    {
        sum += x[i]*y[i];
    }

    return(sum);
}
```

main.c

```
#include<stdio.h>;

float dot(float* x, pm float* y, int n);

void main()
{

    float x[8] = {1, 1.1, 1.3, 1.7, 2.3, 5.5, 8.9, 5.3 };
    float pm y[8] = {7, 5.5, 6.5, 7.5, 8.5, 9.5, 2.3, 3.3 };
    float result;

    result = dot(x, y, 8);

    printf("result = %f",result);


}
```

Output Window

```
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\dot.c  -O1 -save-temps  -proc ADSP-21161  -o .\Debug\dot.doj
".\dot.c", line 12: warning: loop would vectorize if alignment were known -
          consider inserting SIMD_for pragma if all memory accesses are
          aligned on even-word boundaries
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\main.c  -O1 -save-temps  -proc ADSP-21161  -o .\Debug\main.doj
"C:\Program Files\Analog Devices\VisualDSP\cc21k.exe" .\Debug\dot.doj .\Debug\main.doj -proc ADSP-21161 -L .\Debug -flags-link -od,.\D
Build completed successfully.
```

Console Build

Ready | Halted | Line 3, Col 41

Optymalizacja 'C' (#pragma SIMD_for, bez –ipa)

Analog Devices VisualDSP++ - [Target: ADSP-21161 ADSP-2116x Simulator] - [Project: testc.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

dot.s *

```
_dot:    modify(i7,-4);
         r2=pass r12, dm(-5,i6)=r13;
         i4=r4;
         if le jump(pc,.P1L1) (DB);
         i12=r8;
         r0=m13;
         r8=r2-1;
         if gt jump(pc,.P1L3) (DB);
         r12=r0+1;
         r12=r2 and r12;
         r2=dm(i4,m5), r12=pm(i12,m13);
         F2=F2*F12, r13=dm(-5,i6);
         F0=F0+F2, i12=dm(m7,i6);
         jump(m14,i12)(DB); rframe; nop;

.P1L3:   r8=ashift r2 by -1;
         r2=m14;
         r2=max(r2, r8);
         s2=r2;
         bit set mode1 0x200000; nop;
         m4=2;
         r1=r2-1, r8=m13;
         if eq jump(pc,.P1L9) (DB);
         m12=m4;
//       -- bubble --
         r2=dm(i4,m4), r4=pm(i12,m12);
         r1=r1-1;
         if eq jump(pc,.P1L11) (DB);
         F13=F2*F4, r2=dm(i4,m4), r4=pm(i12,m12);
         nop;
         lcntr=r1, do(pc,.P1L14-1)until lce;
.P1L13:  F13=F2*F4, F8=F8+F13, r2=dm(i4,m4), r4=pm(i12,m12);
.P1L14:  //end loop .P1L13;

.P1L11:  F1=F2*F4, F8=F8+F13;
.P1L9:   F2=F2*F4;
         F8=F8+F2;
         bit clr mode1 0x200000; nop;
         r2=m13;
         r1=r2+1, r4=s8;
         comp(r12, r1);
         if ne jump(pc,.P1L6) (DB);
         F8=F8+F4;
         F0=F2+F8;
         r2=dm(i4,m5), r12=pm(i12,m13);
         F2=F2*F12;
         F0=F0+F2;
.P1L1:
.P1L6:   i12=dm(m7,i6);
         jump(m14,i12) (DB);
         r13=dm(-5,i6);
         rframe;
```

dot.c

```c
float dot(float* x, pm float* y, int n);

float dot(float* x, pm float* y, int n)
{
    int i;
    float sum=0;

#pragma SIMD_for
    for(i=0;i<n;i++)
    {
        sum += x[i]*y[i];
//      sum += mult(x[i], y[i]);
    }

    return(sum);
}
```

main.c

```c
#include<stdio.h>;

float dot(float* x, pm float* y, int n);

void main()
{

    float x[8] = {1, 1.1, 1.3, 1.7, 2.3, 5.5, 8.9, 5.3 };
    float pm y[8] = {7, 5.5, 6.5, 7.5, 8.5, 9.5, 2.3, 3.3 };
    float result;

    result = dot(x, y, 8);

    printf("result = %f",result);


}
```

Console Build

Ready | Halted | Line 3, Col 41

Optymalizacja ‘C’ (z –ipa)

Analog Devices VisualDSP++ - [Target: ADSP-21161 ADSP-2116x Simulator] - [Project: testc.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

dot.s

```
    .file ".\dot.c";

    .section/pm seg_pmco;

    // brought to you by a Waveguide/Bril compiler
.epctext:

_dot:
// line ".\dot.c":5
        modify(i7,-4);
        i4=r4;
// line 12
        r2=4;
// line 5
        i12=r8;
// line 12
        bit set mode1 0x200000; nop;
        m4=2;
        r8=m13;
        m12=m4;
//      -- bubble --
        r12=dm(i4,m4), r4=pm(i12,m12);
        F12=F12*F4, r1=dm(i4,m4), r4=pm(i12,m12);
        lcntr=r2, do(pc,.P1L4-1)until lce;

.P1L3:
        F12=F1*F4, F8=F8+F12, r1=dm(i4,m4), r4=pm(i12,m12);
        //end loop .P1L3;

.P1L4:
        bit clr mode1 0x200000; nop;
        r2=s8;
        F2=F8+F2, r12=m13;
        F0=F12+F2, i12=dm(m7,i6);
        jump(m14,i12)(DB); rframe; nop;

._dot.end:
    .global _dot;
    .type _dot,STT_FUNC;
```

dot.c

```
float dot(float* x, pm float* y, int n);

float dot(float* x, pm float* y, int n)
{
    int i;
    float sum=0;

    for(i=0;i<n;i++)
    {
        sum += x[i]*y[i];
    }

    return(sum);
}
```

main.c

```
#include<stdio.h>;

float dot(float* x, pm float* y, int n);

void main()
{

    float x[8] = {1, 1.1, 1.3, 1.7, 2.3, 5.5, 8.9, 5.3 };
    float pm y[8] = {7, 5.5, 6.5, 7.5, 8.5, 9.5, 2.3, 3.3 };
    float result;

    result = dot(x, y, 8);

    printf("result = %f",result);

}
```

Output Window

```
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\dot.c  -O1 -ipa -save-temps  -proc ADSP-21161  -o .\Debug\dot.doj
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\main.c  -O1 -ipa -save-temps  -proc ADSP-21161  -o .\Debug\main.doj
"C:\Program Files\Analog Devices\VisualDSP\cc21k.exe" .\Debug\dot.doj .\Debug\main.doj -proc ADSP-21161 -L .\Debug -flags-link -od,.\D
prelinker: Recompiling ".\dot.c"
Build completed successfully.
```

Console | Build

Ready | Halted | Line 10, Col 21

Optymalizacja ‘C’ (z –ipa)

Analog Devices VisualDSP++ - [Target: ADSP-21161 ADSP-2116x Simulator] - [Project: testc.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

dot.s

```
_dot:
// line ".\dot.c":5
// line 12
        r2=3;
        s2=r2;
        modify(i7,-4);
// line 5
        i4=r4;
        i12=r8;
// line 12
        bit set mode1 0x200000; nop;
        m4=2;
        r1=r2-1, r8=m13;
        if eq jump(pc,.P1L3) (DB);
        m12=m4;
//      -- bubble --
        r2=dm(i4,m4), r4=pm(i12,m12);
        r1=r1-1;
        if eq jump(pc,.P1L5) (DB);
        F12=F2*F4, r2=dm(i4,m4), r4=pm(i12,m12);
        nop;
        lcntr=r1, do(pc,.P1L8-1)until lce;

.P1L7:
        F12=F2*F4, F8=F8+F12, r2=dm(i4,m4), r4=pm(i12,m12);
        //end loop .P1L7;

.P1L8:

.P1L5:
        F12=F2*F4, F8=F8+F12;

.P1L3:
        F2=F2*F4;
        F2=F8+F2;
        bit clr mode1 0x200000; nop;
        r4=pm(i12,m13);
        r12=s2;
        F12=F2+F12, r8=m13;
        r2=dm(i4,m5);
        F2=F2*F4, F12=F8+F12, i12=dm(m7,i6);
        jump(m14,i12) (DB);
        F0=F12+F2;
        rframe;
```

dot.c

```
float dot(float* x, pm float* y, int n);

float dot(float* x, pm float* y, int n)
{
    int i;
    float sum=0;

    for(i=0;i<n;i++)
    {
        sum += x[i]*y[i];
    }

    return(sum);
}
```

main.c

```
#include<stdio.h>;

float dot(float* x, pm float* y, int n);

void main()
{

    float x[8] = {1, 1.1, 1.3, 1.7, 2.3, 5.5, 8.9, 5.3 };
    float pm y[8] = {7, 5.5, 6.5, 7.5, 8.5, 9.5, 2.3, 3.3 };
    float result;

    result = dot(x, y, 7);

    printf("result = %f",result);


}
```

Output Window

```
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\dot.c  -O1 -ipa -save-temps  -proc ADSP-21161  -o .\Debug\dot.doj
"C:\Program Files\Analog Devices\VisualDSP\cc21k" -c .\main.c  -O1 -ipa -save-temps  -proc ADSP-21161  -o .\Debug\main.doj
"C:\Program Files\Analog Devices\VisualDSP\cc21k.exe" .\Debug\dot.doj .\Debug\main.doj -proc ADSP-21161 -L .\Debug -flags-link -od,.\D
prelinker: Recompiling ".\dot.c"
Build completed successfully.
```

Console / Build

Ready | Halted | Line 11, Col 6

Ćwiczenia