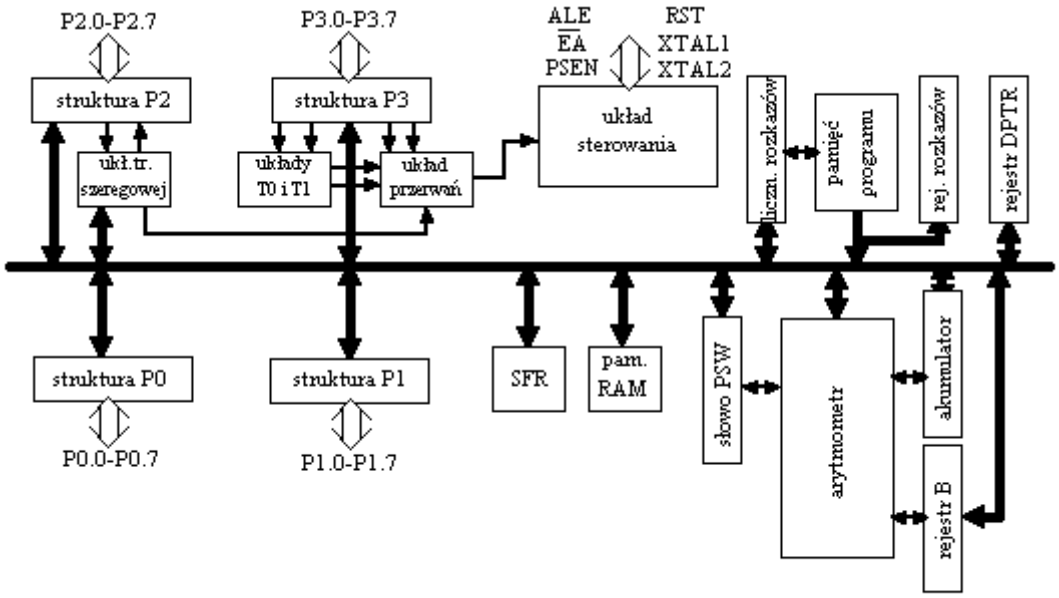
Temat:
Asembler i język C – wprowadzenie w efektywne programowanie niskopoziomowe.

1.Zagadnienia architektury

Ogólnie schemat blokowy mikrokontrolera `80C51` przedstawiono na rysunku 1.



Rysunek 1. Schemat blokowy mikrokontrolera 80C52

Układ sterowania jest synchronizowany zegarem zbudowanym na bazie rezonatora kwarcowego, o częstotliwości 11059200Hz. Do zadań tego układu jest odpowiednie pobudzanie pozostałych układów wewnętrznych w ramach cyklu maszynowego tak aby możliwe było wykonywanie programu, zawartego w pamięci. Cykl maszynowy procesora zajmuje 12 taktów (w przypadku standardowej wersji mikrokontrolera – użyty w STRC51), pogrupowane są one w stany – oznaczane S1, S2, ... S6. Długość trwania każdego stanu to 2 takty. Z pełną częstotliwością, mikrokontroler wykonuje przejścia miedzy taktami. Czas wykonania pojedynczych instrukcji wynosi: 1, 2, 4 cykli maszynowych (w zależności od rodzaju instrukcji, od używanych w niej trybów adresowania).

Pamięć widziana przez program wykonywany w mikrokontrolerze `80C51` dzieli się na:

-pamięć programu (`CODE`),

-pamięć danych: - pamięć wewnętrzną danych (`DATA`),
- pamięć zewnętrzną danych (`XDATA`),
- obszar pamięci - specjalnych zastosowań (`SFR`).

W rodzinie C51 istnieją inne rodzaje pamięci ich analiza wymaga skupienia uwagi na konkretnym modelu mikrokontrolera – co w przypadku korzystania z płytki `STRC51` nie ma znaczenia.

Po wygenerowaniu sygnału RESET (sprzętowo i programowo) rdzeń mikrokontrolera zaczyna pobieranie programu od adresu 0 w pamięci programu (`CODE`). W tabeli 1. zawarto mapę pamięci.

Adres	Zawartość
`0x0000`	Obsługa procedury `RESET`
`0x0003`	Obsługa procedury przerwania `INT0`
`0x000B`	Obsługa procedury przerwania zegarowego `T0`
`0x0013`	Obsługa procedury przerwania `INT1`
`0x001B`	Obsługa procedury przerwania zegarowego `T1`
`0x0023`	Obsługa procedury przerwania od złącza szeregowego `RS232`

Tabela 1. Mapa pamięci wektorów przerwań `80C51`

W rodzinie serii `C51` istnieją inne procedury przerwania, zależą od producenta i typu układu scalonego.

Wewnątrz mikrokontrolera mamy do dyspozycji ośmiobitowe rejestry uniwersalne A,B oraz cztery banki rejestrów ogólnego przeznaczenia R0-R7 (przełączane przez specjalny rejestr PSW). Istnieje też pewien specjalny rejestr DPTR przydatny podczas odwołań do pamięci. Jest to rejestr 16 bitowy, który można modyfikować przez odwołanie do dwóch rejestrów DPL i DPH, lub za pomocą niektórych instrukcji do całego słowa.
Sterowanie działaniem mikrokontrolera oparte jest na specjalnym rejestrze PC. Wielkość tego rejestru to 16 bitów, ona rzutuje wielkości pamięci programu (64KB). Nie jest możliwe bezpośrednie modyfikowanie jego zawartości. Są specjalne rozkazy sterujące których wykonanie zmienia jego zawartość. Mimo iż teoretycznie wielkość pamięci równa 64KB jest domyślną, lecz są instrukcje między - innymi skoku - które nie umożliwiają skoku poza obszar strony (2KB). Obszar pamięci programu i pamięci danych jest fizycznie rozdzielony i dla pełnej interferencji wymaga stosowania specjalnych instrukcji.
Instrukcje rozpoznawane przez mikrokontroler dzielą się na:
-wykonujące operacje arytmetyczne,

Mnemonik	**Opis**	**Argumenty**	**Wynik/ modyfikacja znaczników**	**Liczba taktów**	**Liczba bajtów**
ADD	dodawanie arytmetyczne (A=A + arg),	A, Rn	A/C,OV,AC	1	1
		A, direct	A/C,OV,AC	1	2
		A, @Ri	A/C,OV,AC	1	1
		A, #data	A/C,OV,AC	1	2
ADDC	dodawanie arytmetyczne z przeniesieniem (A=A+ arg +C),	A, Rn	A/C,OV,AC	1	1
		A, direct	A/C,OV,AC	1	2
		A, @Ri	A/C,OV,AC	1	1
		A, #data	A/C,OV,AC	1	2
SUBB	odejmowanie arytmetyczne z „przeniesieniem-pożyczką” (A=A-... -C)	A, Rn	A/C,OV,AC	1	1
		A, direct	A/C,OV,AC	1	2
		A, @Ri	A/C,OV,AC	1	1
		A, #data	A/C,OV,AC	1	2
INC	inkrementacja arg=arg+1	A	A	1	1
		Rn	Rn	1	1
		direct	direct	1	1
		@Ri		1	1
		DPTR	DPTR	2	1
DEC	dekrementacja arg=arg-1	A	A	1	1
		Rn	Rn	1	1
		direct	direct	1	2
		@Ri		1	1
MUL	mnożenie liczb 8 bitowych, z wynikiem 16 bitowym	A i B	A(LSB) B(MSB) /OV, C=0	4	1
DIV	dzielenie liczby 8 bitowej przez 8 bitowa	A przez B	A (całkowita), B (reszta) /OV, C=0	4	1
DA	poprawka dziesiętna, czyli operacja przekształcenia liczby w postać BCD	A	A/C	1	1

-wykonujące operacje logiczne,

Mnemonik	Opis	Argumenty	Wynik/ modyfikacja znaczników	Liczba taktów	Liczba bajtów
XRL	suma modulo 2, arg1 = arg1 XOR arg2	A, Rn	A	1	1
		A, direct	A	1	2
		A, @Ri	A	1	1
		A, #data	A	1	2
		direct, A	direct	1	2
		direct, #data	direct	2	3
CLR	zerowanie bajtowe, arg = 0	A	A		
CPL	negowanie bitów w bajcie	A			
RL	obrót w lewo (bit 7 jest tracony w jego miejsce kopiowany jest bit 6, ...)	A	A	1	1
RR	obrót w prawo (bit 0 jest tracony w jego miejsce wstawiany jest bit 1, ...)	A	A	1	1
RLC	obrót w lewo przez bit C (bit 7 kopiowany jest w C, a bit C w bit 0, ...)	A	A/C	1	1
RRC	obrót w prawo przez bit C (bit 0 kopiowany jest w C, a bit w C na bit 7,...)	A	A/C	1	1
SWAP	zamienia miejscami cztery starsze bity z czterema młodszymi bitami.	A	A	1	1

-wykonujące operacje logiczne na bitach,

Mnemonik	Opis	Argumenty	Wynik/ modyfikacja znaczników	Liczba taktów	Liczba bajtów
CLR	zerowanie podanego bitu	A	A	1	1
		bit	bit	1	1
		C	C=0	1	1
SETB	ustawienie bitu	C	C=1	1	1
		bit.	bit	1	2
CPL	neguje podany bit	A	A	1	1
		bit	bit	1	2
		C	C=!C	1	1
ANL	wykonuje iloczyn logiczny dwóch bitów, wynik zapisuje w pierwszym.	A, Rn	A	1	1
		A, direct	A	1	2
		A, @Ri	A	1	1
		A, #data	A	1	2
		direct, A,	Direct	1	2
		direct, #data	Direct	2	3
		C, bit	C/C	2	2
		C, /bit	C/C	2	2
ORL	wykonuje sumę logiczną dwóch bitów, wynik zapisuje w pierwszym	A, Rn	A	1	1
		A, direct	A	1	2
		A, @Ri	A	1	1
		A, #data	A	1	2
		direct, A	direct	1	2
		direct, #data	direct	2	3
		C, bit	C/C	2	2
		C, /bit	C/C	2	2

-wykonujących przesyłanie danych,

Mnemonik	Opis	Argumenty	Wynik i modyfikacja znaczników	Liczba taktów	Liczba bajtów
MOV	kopiowanie wartości	A, Rr	A	1	1
		A, direct	A	1	2
		A, @Ri	A	1	1
		A, #data	A	1	2
		Rn, A	Rn	1	1
		Rn, direct	Rn	2	2
		Rn, #data	Rn	1	2
		direct, A	direct	1	2
		direct, Rn	direct	2	2
		$direct_1$, $direct_2$	direct	2	3
		direct, @Ri	direct	2	2
		direct, #data	direct	2	3
		@Ri, A		1	1
		@Ri, direct		2	2
		@Ri, #data		2	2
		C, bit	C/C	1	2
		bit, C	bit	2	2
		DPTR, $\#data_{16}$	DPTR	2	3
MOVC	kopiowanie danej z obszaru pamięci programu	A, @A+PC	A	2	1
		A, @A+DPTR	A	2	1
MOVX	kopiowanie do/z pamięci danych zewnętrznych	A, @Ri	A	2	1
		A, @DPTR	A	2	1
		@Ri, A		2	1
		@DPTR, A		2	1
XCH	Zamiana zawartości dwóch rejestrów między sobą	A, Rn	A	1	1
		A, direct	A	1	2
		A, @Ri	A	1	1
XCHD	Zamień młodsze cztery bity miedzy rejestrami	A, @Ri	A	1	1
PUSH	Połóż na stos, z sprzętową zmianą SP (wskaźnika stosu)	arg. bezpośredni		2	2
POP	zdejmij ze stosu, z sprzętową zmianą SP (wskaźnika stosu)	arg. bezpośredni		2	2

-kontrolujące proces wykonywania programu,

Mnemonik	Opis	Argumenty	Wynik/ modyfikacja znaczników	Liczba taktów	Liczba bajtów
NOP	nic nie wykonuj (elementarna instrukcja o czasie trwania jednego cyklu maszynowego)			1	1
ACALL	wywołanie podprogramu w stronie 2kB, zapisuje na stos zawartość PC i wykonuje skok do adresu początku procedury, podanej jako argument	addr (11 bitowy)		2	2
LCALL	wywołanie podprogramu, zapisuje na stos zawartość PC i wykonuje skok do adresu początku procedury, podanej jako argument	addr (16 bitowy)		2	3

RET	powrót z procedury, pobiera ze stosu zawartość licznika rozkazów i przekazuje sterowanie do znajdującego się tam adresu	brak		2	1
RETI	powrót z procedury obsługi przerwania, pobiera ze stosu zawartość licznika rozkazów i przekazuje sterowanie do znajdującego się tam adresu	brak		2	1
LJMP	rozkaz skoku w obszarze 64kB, wpisuje do licznika rozkazów podany adres	addr (16 bitowy).		2	3
AJMP	rozkaz skoku w stronie 2kB, zastępuje 11 młodszych bitów PC wartością podaną w wywołaniu	addr (11 bitowy).		2	2
SJMP	rozkaz skoku w obszarze 256B, następuje zmiana zawartości PC o wartość w kodzie U2 podaną w wywołaniu (PC=PC+arg_{U2}).	arg_{U2}		1	2
JMP	rozkaz skoku w trybie pośrednim, przepisuje do PC wartość będącą sumą zawartości rejestru indeksowego i bazowego (PC=@A+DPTR).	@A+DPTR		2	1
JZ	skok warunkowy, wykonuje skok, jeżeli zawartość akumulatora równa jest 0	arg_{U2}		2	2
JNZ	skok warunkowy, wykonuje skok, jeżeli zawartość akumulatora jest różna od 0	arg_{U2}		2	2
JC	skok warunkowy, wykonuje skok, jeżeli bit przeniesienia C jest ustawiony	arg_{U2}		2	2
JNC	skok warunkowy, wykonuje skok, jeżeli bit przeniesienia C jest wyzerowany	arg_{U2}		2	2
JB	skok warunkowy, wykonuje skok, jeżeli dany bit jest ustawiony	bit, arg_{U2}		2	3
		C, arg_{U2}		2	3
JNB	skok warunkowy, wykonuje skok, jeżeli dany bit jest wyzerowany	bit, arg_{U2}		2	3
JBC	skok warunkowy, gdy dany bit jest ustawiony, to nastąpi skok i wyzerowanie bitu	bit, arg_{U2}			
CJNE	skok warunkowy, zależny od wyniku porównania, następuje porównanie operandów, jeżeli są różne, to następuje skok, znacznik C ustawiany, jeżeli pierwszy operand mniejszy od drugiego	A, direct, rel	/C	2	3
		A, #data, rel	/C	2	3
		Rn, #data, rel	/C	2	3
		@Ri, #data, rel	/C	2	3
DJNZ	skok warunkowy, zależny od wyniku dekrementacji, następuje dekrementacja zawartości operandu, jeżeli zawartość nie osiągnęła zera, to nastąpi skok	A, direct, rel		2	3
		A, #data, rel		2	3
		Rn, #data, rel		2	3
		@Ri, #data,rel		2	3

Użyto następujących symboli:

Rn - dowolny rejestr z zestawu:
podstawowego: `R0,..,R7,`
drugiego: `R0',..,R7',`
trzeciego: `R0'',...,R7'',`
czwartego: `R0''',..,R7'''` ,

direct - komórka pamięci w obszarze wewnętrznym (`DATA`),

@Ri - zawartość komórki pamięci adresowanej przez rejestr Ri gdzie może to być R0,R1,
#data - wartość wbudowana w rozkaz,
arg_{U2},rel - przesuniecie

2.Modularność programów tworzonych przy pomocy kompilatora SDCC

Projekty tworzone przy użyciu pakietu SDCC, mogą być modularyzowane. Każda z grup funkcji kluczowych może być zawarta w innym pliku. Przy wykorzystaniu narzędzi do automatyzacji kompilacji (make) można sprawniej i w bardziej przejrzysty sposób tworzyć oprogramowanie.
Przykładem projektu składającego się z dwóch modułów jest zmodyfikowana wersja projektu prezentowanego w ramach I ćwiczenia w I semestrze. Zadaniem programu tak utworzonego jest generowanie sygnału akustycznego za pomocą wbudowanego głośniczka.
Głównym plikiem projektu jest Makefile opisujący sposób tworzenia wyników:

```
SDCCCFLAGS  = --model-small
ASLINKFLAGS = --code-loc 0x4000 --data-loc 0x08 \
              --xram-loc 0x0000
%.rel : %.c
     sdcc $(SDCCCFLAGS) -c $<
%.rel : %.s
     asx8051 -losgp $<

all: main.hex

main.hex: main.c main.rel buzzer.s buzzer.rel
     sdcc $(SDCCCFLAGS) $(ASLINKFLAGS) main.rel buzzer.rel
     packihx main.ihx > main.hex
     copyclip main.hex

clean:
     rm -f *.asm *.hex *.ihx *.lnk *.lst *.map *.rel *.rst *.sym
```

W pierwszych liniach ustalane są opcje kompilatora. Oznaczenie SDCCFLAGS jest od tego momentu (w ramach tego pliku makefile) rozpoznawana i podstawiane jako ciąg --model-small. Podobnie sprawa wygląda z ASLINKFLAGS. Warto zwrócić uwagę na możliwość umieszczania symbolu '\' na końcu linii, interpretowany on jest jako znak kontynuacji linii (szczególnie przydatne gdy ilość parametrów jest zbyt duża).
Program make stosuje pojecie reguł i zależności. Fraza:

```
%.rel : %.c
     sdcc $(SDCCCFLAGS) -c $<
```

Jest traktowana jako reguła tworzenia zbiorów z rozszerzeniem .rel (linia rozpoczyna się od symbolu reguły może zawierać dowolny ciąg znaków jak i znaki specjalne, np.: %.rel, dokładniejsze informacje można znaleźć w dokumentacji do pakietu make). W podanym przykładzie zostanie wywołany program SDCC z parametrami: $(SDCCCFLAGS) -c $<, gdzie znaki: $< będą oznaczały ogólną nazwę pliku z rozszerzeniem .c, stosowaną do reguły (tutaj regułe ustaliła linia rozpoczynając się od %.rel czyli np.: main.rel).
Warto zwrócić uwagę iż w każdej regule w pierwszej linii znajduje się jej nazwa, następnie po znaku ':' znajduje się zależność (dependenecs), a w następnej linii (koniecznie rozpoczynająca się od znaku <TAB>) jest definicja czynności do wykonania. Taki układ może być wielo poziomowy – z zagnieżdżeniami (nie mylić z zagnieżdzeniami języków wysokiego poziomu). Np.:

```
all: main.hex
main.hex: main.c main.rel buzzer.s buzzer.rel
     sdcc $(SDCCCFLAGS) $(ASLINKFLAGS) main.rel buzzer.rel
```

Widać że obiekt all będzie tworzony w zależności od obiektu main.hex, z kolei main.hex zalezy od całej listy obiektów: main.c, main.rel, buzzer.s, buzer.rel.

Podczas wywołania programu `make`, szuka on w aktualnym katalogu pliku `Makefile` (lub zależnie od wersji innej nazwy) a następnie interpretuje go. W przypadku podania w linii poleceń: `make clean`, tworzony jest wynik opisany zależnością `clean`: (w tym przypadku jest to czyszczenie aktualnego katalogu z wyników kompilacji – **UWAGA!** Programy asemblerowe najlepiej opatrzyć rozszerzeniem `.s` zamiast `.asm`, dzięki temu nie ulegnie on skasowaniu za pomocą celu `clean`).
W przypadku nie podania argumentu w wywołaniu, szukany jest pierwszy cel i on staje się celem domyślnym (tutaj: `all: main.hex`).
Podobnie jak to miało miejsce w I ćwiczeniu w I semestrze zawartość pliku `main.c` wygląda następująco:

```
void BUZZER1(void);
void BUZZER0(void);

xdata at 0x8000 unsigned char U12; /*umiejscowienie klawiatury (bufora)*/

void main (void){
 unsigned char i,r=0;
 for(;;){
   if((U12 & 0x0f)!=0x0f){          /*czy wcisnieto klawisz*/
     if((r & 0x01)==0)              /*jezeli tak to generujemy fale dzwiekowa*/
       BUZZER1();
     else
       BUZZER0();
   }
   r++;
   for(i=0; i<70; i++);             /* drobne opoznienie  */
 }
}
```

Który to program cyklicznie wywołuje dwie funkcje: `BUZZER1()` i `BUZZER0()`. Ich prototypy są umiejscowione w pierwszych liniach:

```
void BUZZER1(void);
void BUZZER0(void);
```

Wskazują one kompilatorowi iż, można spodziewać się rozwinięcia tych funkcji. Definicje tych funkcji zostaną rozwinięte w pliku `buzzer.s` który wygląda następująco:

```
     .module buzzer
     .globl _BUZZER1
     .globl _BUZZER0

     .area OSEG
_T1  =    0x00B5

     .area CSEG    (CODE)
_BUZZER1:
     setb _T1
     ret
_BUZZER0:
     clr  _T1
     ret
```

Wszelkie oznaczenia zaczynające się od znaku '.' są traktowane jako dyrektywy asemblera. Natomiast znak `';'` traktowany jest jako komentarz.
Dokładny opis używanego cross-asemblera w języku angielskim można znaleźć na:
`http://cygnus.tele.pw.edu.pl/olek/doc/np/obce/asxhtm.html`.

Wyjaśnienia wymagają linie:

```
    .globl _BUZZER1
    .globl _BUZZER0
```

Oznaczają one iż symbole `_BUZZER1` i `_BUZZER0` będą widoczne w globalnym projekcie podczas linkowania (są one oznaczeniem funkcji). W pliku `.s` musi się znaleźć rozwinięcie tych funkcji. Wszelkie nazwa musi być poprzedzona znakiem '_', dla zapewnienia zgodności nazw obiektów w plikach tworzonych podczas kompilacji.
Zdefiniowano także symbol `_T1` w obszarze `OSEG`, jest to odpowiednik deklaracji w „C":

```
    sbit at 0xB5 T1;
```

Określenie obszarów (tu: `.area`) jest dość istotne. Dzięki takim dyrektywom można instruować kompilator jak i gdzie ma umieszczać fragmenty kodu i rozmieszczać obiekty. W obszarze zdefiniowanym jako:

```
    .area CSEG    (CODE)
```

mogą znaleźć się rozwinięcia funkcji lub definicje stałych osadzone w pamięci kodu. Obszarami są:

```
.area REG_BANK_0    (REL,DATA) - obszar rejestrów zestawu podstawowego
.area DSEG    (DATA)           - obszar pamięci RAM wewnętrznej
.area OSEG    (OVR,DATA)       - obszar pamięci RAM wewnętrznej, nakładkowanej
.area SSEG    (DATA)           - obszar pamięci RAM związanej z stosem
.area ISEG    (DATA)           - obszar danych adresowanych bezpośrednio
.area BSEG    (BIT)            - obszar adresowany bitowo
.area XSEG    (XDATA)          - obszar danych w pamięci XRAM
.area XISEG   (XDATA)          - obszar danych w pamięci XRAM wstępnie inicjowany
.area CSEG    (CODE)           - obszar pamięci kodu programu i danych stałych
```

Więcej informacji można znaleźć w dokumentacji kompilatora języka cross-asemblera (`asx8051.exe`).

3.Wstawki asemblerowe

Istnieje także możliwość tworzenia modułów całkowicie w „C" z fragmentami kodu funkcji napisanymi w asemblerze. Jest to użyteczne zwłaszcza gdy tworzenie całych modułów asemblerowych jest nie potrzebne a tylko wybrane fragmenty (nawet nie całe funkcje) mają być optymalne.
Podczas pisania wstawek, należy zwrócić uwagę na problemy zasobów. Funkcje całkowicie napisane w „C" maja przydzielane sobie zasoby automatycznie, podczas kompilacji. W przypadku używania wstawek asemblerowych dostęp do zasobów musi być nadzorowany przez programistę (niemalże ręcznie). Między innymi nazwy etykiet (labels) musza być tworzone w taki sposób aby nie kolidowały z automatycznie tworzonymi przez kompilator „C" i powinny kończyć się znakiem „$".
Generalnie wstawki asemblerowe mają postać:

```
_asm
  ...
_endasm
```

To co znajduje się miedzy słowami `_asm` i `_endasm` musi być zapisane bez wyjątków w asemblerze.

4.Przekazywanie argumentów i zwracanie wyników z funkcji.

Problem przekazywania argumentów w języku „C" jest dość rozległy. Każda architektura docelowa inaczej wspiera przekazywanie argumentów. W przypadku mikrokontrolerów jednoukładowych o małych zasobach pamięciowych (w szczególności o płytkim stosie) stosowane są rejestry do przekazywania danych miedzy funkcjami.
O ile w przypadku funkcji o prototypie:

```
    void DoFunc(void);
```

nie przekazywane żadne argumenty, ani żaden wynik nie jest zwracany.

Wyniki zwracane są przez:

`char` lub `unsigned char`	-	`dpl`
`int` lub `unsigned int`	-	`dptr`
`void *` (oraz jemu podobne, choć nie wszystkie)	-	`dptr`(młodsze słowo), `b`(starsze słowo)
`float`	-	`dptr, b`
`long` lub `unsigned long`	-	`dptr`(młodsze słowo), `b`(starszy bajt starszego słowa), `a`(młodszy bajt starszego słowa)

Dla wybranych przykładów przekazywanie argumentów jest następujące:

`...DoFunc(char c);`	-	`c->dpl`
`...DoFunc(char c1, char c2);`	-	`c1->dpl`, `c2->`przez adres w pamięci RAM
`...DoFunc(int c);`	-	`c->dptr`
`...DoFunc(int c1, int c2);`	-	`c1->dptr`, `c2->`przez adres w pamięci RAM
`...DoFunc(long c);`	-	`c->dptr`(młodsze słowo), `ab`(starsze słowo)
`...DoFunc(float c);`	-	`c->dptr, a b`
`...DoFunc(void *c);`	-	`c->dptr, b`

Dla pozostałych kombinacji, najlepiej „podpatrzeć" w jaki sposób kompilator tworzy w pliku `ASM`. Podczas kompilacji zgodnie z schematem podanym w pliku `makefile`, tworzone są plik pośrednie z rozszerzeniem `.asm`. Zawierają one wersje asemblerową pliku napisanego w „c". Porównując odpowiednią linię, w pliku `.c` z wersją w pliku `.asm` "dopatrzeć" się jak dana linia została rozwinięta. Każda linia pliku `.c` jest w swoim odpowiedniku `.asm`, opatrzona komentarzem w postaci:

```
;main.c:15: if((r & 0x01)==0)
;     genAnd
     mov  a,#0x01
     anl  a,r2
     mov  r3,a
;     genCmpEq
;       Peephole 132   changed ljmp to sjmp
;       Peephole 199   optimized misc jump sequence
     cjne r3,#0x00,00102$
...
```

fragment poniżej tego wpisu, jest rozwinięciem linii 15 w pliku: `main.c`. Należy zaznaczyć że nie wszystkie linie pliku `.c` musza być rozwinięte w pliku `.asm`. Wynika to z działania optymalizatora wbudowanego w kompilator `SDCC` - patrz linie:

```
;       Peephole 132   changed ljmp to sjmp
;       Peephole 199   optimized misc jump sequence
```

Które jak widać zostały zoptymalizowane - czego raport został umieszczony jako komentarz.

5. Literatura

- Tomasz Starecki, "Mikrokontrolery 8051 w praktyce", Wydawnictwo BTC 2002,
- Piotr i Paweł Gałka, „Podstawy programowania mikrokontrolera 8051", Wydawnictwo: MIKOM, Warszawa.
- Tomasz Starecki, "Mikrokontrolery jednoukładowe rodziny 51", Wydawnictwo NOZOMI 1996,
- Doliński J, „Mikrokomputer jednoukładowy INTEL 8051", PLJ,Warszawa 1993.
- Fręchowicz A, Heyduk A. „Mikrosterowniki rodziny MCS-51", Wydawnictwo Politechniki Śląskiej , Gliwice 1998.
- Majewski J., Kardach K, „Mikrokontrolery jednoukładowe 8051. Programowanie w języku C w przykładach", Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 1996.
- Ryszard Pełka, „MIKROKONTROLERY. Architektura, programowanie, zastosowania", WKŁ.

- Janik R., Świerczek Ł., http://www.zsme.zywiec.pl/prace_uczniow/m8051/index.html,

Zadania dla WSISIZ:

1.wstep - pisanie modularne (w C i ASM), łączenie C z ASM, rejestry, instrukcje [na zaliczenie: buzzer w C z funkcjami w ASM - nie używać wstawek]
2.liczniki i peryferia (led, key), [na zaliczenie: napisac rozwiniêcie funkcji: putled, getchar
3.rs232 - rozwiazane prolemu ladowania danych w HEX z PC do STR51, bez bledow (mozliwe rozwiazania:
-szybki moduł dający pewność nie gubienia znaków,
-moduł z buforami okrężnymi,
-moduł z mechanizmem software'owego kontrolowania przeplywu danych
w podanych rozwiazaniach można kontrolowaæ sumy kontrolne)
4.RTC - problem pisania sytemów obslugi magistrali I2C (zadanie:
-nawiazanie komunikacji z uk³adem PCF8583,
-umiejetnoœæ programowania czasu
-umiejetnoœæ programowania alarmu,
-umiejetnoœæ programowania alarmu z wyorzystanem generowaia przez uk³ad PCF przerwan)
5,6bios 1 - zadanie napisania biosu o cechach:
-musi zawieraæ procedury inicjacji i testu peryferi
-musi zapewniaæ dobrze okreœlone API dla programistów
-mo¿na probowaæ zaimplementowaæ "system plików" tak by po cieplym starcie mozna by³o wybrac aplikacje
-mozna zaimplementowaæ system zapamietywania danych ko figuracynych w pamieci nie ulotnej RTC

DODATKOWE:
-RTOS
-