Wokół języka C — część 3

Asembler i adresowanie

Zgodnie z zapowiedzią, spróbujmy zrozumieć na pozór dziwne opisy działania poszczególnych rozkazów asemblera.

Opis działania

Jak więc rozumieć zaznaczony czerwono opis działania rozkazu **LD Rd,-X**?

LD Rd, -X	Load Indirect and Pre-Decrement	X ← X - 1, Rd ← (X)

Początkującym tego rodzaju zapisy rzeczywiście wydają się dziwne, ale wcale nie znaczy to, że w informatyce wykorzystujemy jakąś zupełnie inną matematykę. Matematyka (logika) jest tylko jedna. Po prostu przyjęło się, że pewne operacje i działania *zapisujemy inaczej*, niż to jest od wieków przyjęte w innych dziedzinach. Czyli do opisu tych samych działań stosujemy różne *notacje* – sposoby zapisu.

Przyzwyczailiśmy się, że działania arytmetyczne (i logiczne) zapisujemy tak: 2 + 3 = 5, czyli *najpierw podajemy składniki, a na końcu wynik*. Najogólniej biorąc, *odwrotnie jest w informatyce: najpierw zapisujemy wynik, a potem „składniki”*. Dotyczy to nie tylko działań arytmetycznych, ale też bardziej ogólnych.

Można byłoby dyskutować, jaka notacja, czyli sposób zapisu, jest bardziej logiczna czy intuicyjna. Można byłoby przywołać postać Jana Łukasiewicza, polskiego logika i filozofa, twórcy tak zwanej *notacji polskiej*. Mógłbym też opowiedzieć, jak pod koniec lat 70. stałem się szczęśliwym posiadaczem znakomitego na owe czasy kalkulatora Omron, którego oprócz mnie i jednego kolegi, nikt w naszym technikum obsługiwać nie umiał – właśnie z uwagi na wykorzystanie w nim RPN (odwrotnej notacji polskiej).

Zamiast zagłębiać się w takie skądinąd interesujące szczegóły powiedzmy, że w informatyce stosujemy specyficzny, w pewnym sensie „odwrotny” sposób zapisu. Dotyczy to zarówno języka C, jak i omawianego teraz asemblera i kodu maszynowego.

Nie ma rady – trzeba się przyzwyczaić! I tak działanie rozkazu przekopiowania (przesunięcia) danych z rejestru źródłowego Rr do rejestru przeznaczenia Rs NIE jest opisane w sposób znany z klasycznej matematyki **Rr → Rd**, tylko „odwrotnie”: **Rd ← Rr**. W katalogu rozkaz ten jest opisany następująco:

MOV Rs, Rr	Move Between Registers	Rd ← Rr

Polecenie ADD, które dodaje zawartość dwóch rejestrów, NIE jest opisane: **Rr + Rd → Rd**, tylko „odwrotnie”: **Rd ← Rd + Rr**.

Rys. 18

Mnemonics	Operands	Description	Operation	Flags
ADD	Rd, Rr	Add two Registers without Carry	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add two Registers with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H
SBC	Rd, Rr	Subtract two Registers with Carry	Rd ← Rd - Rr - C	Z,C,N,V,H
SBCI	Rd, K	Subtract Constant from Reg with Carry.	Rd ← Rd - K - C	Z,C,N,V,H

Mnemonik	Operandy	Flagi	Działanie	Opis działania
ADD	Rd,Rr	Z,C,N,V,H,S	Rd = Rd + Rr	Dodaj bez Carry
ADC	Rd,Rr	Z,C,N,V,H,S	Rd = Rd + Rr + C	Dodaj z Carry
SUB	Rd,Rr	Z,C,N,V,H,S	Rd = Rd - Rr	Odejmij bez Carry
SUBI	Rd,K8	Z,C,N,V,H,S	Rd = Rd - K8	Odejmij stałą
SBC	Rd,Rr	Z,C,N,V,H,S	Rd = Rd - Rr - C	Odejmij z Carry
SBCI	Rd,K8	Z,C,N,V,H,S	Rd = Rd - K8 - C	Odejmij stałą z Carry

Mnemonik	Opis działania	Operacja	Zmieniane flagi
ADD Rd, Rr	Dodaj	Rd := Rd + Rr	Z,C,N,V,H
ADC Rd, Rr	Dodaj uwzględniając przeniesienie	Rd := Rd + Rr + C	Z,C,N,V,H
SUB Rd, Rr	Odejmij	Rd := Rd - Rr	Z,C,N,V,H
SUBI Rd, K	Odejmij bezpośrednio	Rd := Rd - K	Z,C,N,V,H
SBC Rd, Rr	Odejmij uwzględniając pożyczkę	Rd := Rd - Rr - C	Z,C,N,V,H
SBCI Rd, K	Odejmij bezpośrednio uwzględniając pożyczkę	Rd := Rd - K - C	Z,C,N,V,H

Rys. 19

DATA TRANSFER INSTRUCTIONS				
Mnemonics	Operands	Description	Operation	Flags
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None
LDI	Rd, K	Load Immediate	Rd ← K	None
LD	Rd, X	Load Indirect	Rd ← (X)	None
LD	Rd, X+	Load Indirect and Post-Increment	Rd ← (X), X ← X + 1	None
LD	Rd, - X	Load Indirect and Pre-Decrement	X ← X - 1, Rd ← (X)	None

Analogicznie „odwrotnie” opisane są także inne rozkazy. Na **rysunku 18** masz trzy różne opisy kilku rozkazów – zwróć uwagę na wyróżnione różowym kolorem „odwrotne” opisy działania. Opisy te mówią, że działania dotyczą zawartości rejestrów.

A teraz zwróć uwagę na **rysunek 19**, gdzie przedstawione są opisy rozkazów wczytywania danych do rejestru (Rd). W rozkazie **LDI Rd, K** do rejestru Rd wpisywana jest (ośmiobitowa) stała K, zawarta w treści rozkazu, co opisane jest **Rd ← K**. W rozkazach **LD** wykorzystuje się adresowanie pośrednie zawartością rejestru wskaźnikowego X (pary rejestrów roboczych R27, R26). A więc *do rejestru przeznaczenia Rd NIE jest wpisywana zawartość (16-bitowego) rejestru X, tylko **zawartość komórki RAM o adresie-numerze, zawartym w rejestrze wskaźnikowym X***. Dlatego w opisie nie piszemy **Rd ← X**, tylko **Rd ← (X)**. Nawias wskazuje tu, że chodzi o adresowanie pośrednie, nie o zawartość rejestru X, tylko zawartość komórki pamięci RAM, której adres-numer zawarty jest w rejestrze X.

Opis rozkazu **LD Rd, X+** jest następujący:

Rd ← (X), X ← X + 1

najpierw do rejestru Rd wpisywana jest zawartość komórki RAM, adresowana przez rejestr X, a potem liczba-adres w rejestrze X jest zwiększana o 1.

Czy teraz jest dla Ciebie jasne, że w przypadku rozkazu **LD Rd,-X** opis **X ← X - 1, Rd ← (X)** wskazuje, iż najpierw zmniejszana jest o 1 zawartość rejestru X, a dopiero potem do rejestru Rd wpisywana jest zawartość komórki RAM, adresowana takim „zmniejszonym” adresem? A teraz kolejny istotny szczegół.

Jeszcze raz flagi

Na rysunku 18 masz kolumnę, w której podane jest, które flagi może zmodyfikować wykonanie danego rozkazu. Chodzi tu o flagi zawarte w rejestrze stanu (SREG). Są one wyróżnione żółtym kolorem na **rysunku 20**. Przypomnijmy, że te flagi są ustawiane automatycznie w wyniku wykonania niektórych rozkazów, głównie arytmetyczno-logicznych, ale także niektórych innych. I tak flagi H, S, V, N, Z, C są ustawiane (wpisywana wartość 1):

N – gdy wynik operacji jest ujemny

Z – gdy wynik operacji jest zerem

C – (Carry) gdy nastąpiło przeniesienie

H (Half Carry) – gdy nastąpi tzw. przeniesienie połówkowe, wykorzystywane przy obliczeniach w kodzie BCD

V – (Two's Complement Overflow Flag) flaga przeniesienia/przepełnienia przy operacjach na liczbach ze znakiem w formacie uzupełnienia do 2

S – flaga znaku; ustawiana, gdy flagi V, N nie są jednakowe ($S = N \oplus V$). Te automatycznie ustawiane flagi niosą pewne ważne informacje o wyniku przeprowadzonej operacji i mogą być użyte do sprawdzania warunków i podejmowania decyzji o przebiegu programu (o skokach i pominięciach (*branch, skip*)).

Jeśli chcesz, samodzielnie zbadaj te sprawy dokładniej. Ale nie jest to konieczne, bo przy programowaniu w języku C tymi szczegółami będzie się zajmował kompilator. My krótko omówimy jeszcze tylko...

Pamięć EEPROM

Jak wiesz, oprócz stałej pamięci programu FLASH i ulotnej pamięci danych RAM, mikrokontrolery jednoukładowe zawierają też pamięć nieulotną EEPROM. Służy ona do zapamiętywania bieżących ustawień, które nie powinny zostać utracone po wyłączeniu zasilania, ale które nie są „na sztywno" wpisane do pamięci FLASH. Jest to więc pomocnicza, nieulotna pamięć danych. Jej wadą jest ograniczona liczba cykli zapisu (rzędu 100000) oraz długi czas zapisu.

Adresowanie i obsługa pamięci EEPROM nie są skomplikowane, ale są specyficzne. Wracając do opisu procesora jako biura, w którym pracuje bardzo skrupulatny, ale słabo rozgarnięty i ograniczony umysłowo urzędnik, powiemy, że ten urzędnik... nawet nie wie o istnieniu pamięci EEPROM. Pamięć EEPROM jest niejako ukryta i można się do niej dostać przez rejestry w podstawowym obszarze I/O. Jest obsługiwana przez cztery ośmiobitowe rejestry o nazwach: **EEARH**, **EEARL** (EEPROM Adress Register), tworzące „podwójny" rejestr adresowy, określający, której komórki EEPROM dotyczy operacja, rejestr danych **EEDR** (EEPROM Data Register), gdzie umieszczane są dane do zapisania w EEPROM lub odczytane z zaadresowanej komórki oraz bity w rejestrze sterującym **EECR** (EEPROM Control Register). Ilustruje to **rysunek 21**, fragment karty katalogowej ATmega328PB. Adresy rejestrów 0x3F...0x42 to numery komórek RAM (numery kolejne tych rejestrów w podstawowej przestrzenni I/O są o 0x20 mniejsze, czyli wynoszą 0x1F...0x22).

Mówiąc najprościej, operacja odczytu komórki pamięci nieulotnej EEPROM polega na wpisaniu adresu-numeru tej komórki do pary rejestrów EEARH, EEARL i zmianie stanu „bitu odczytu" (EERE) w rejestrze EECR – wtedy w rejestrze danych EEDR pojawi się zawartość odczytana z tej komórki. W przypadku zapisu jest podobnie: najpierw trzeba wpisać adres i dane do zapisania, a potem zmienić stan odpowiednich bitów w rejestrze EECR. Jest to nieco bardziej skomplikowane, ponieważ zapis trwa nieporównanie dłużej niż odczyt i trzeba też uwzględnić dodatkowe czynniki.

Podsumowanie

Poświęciliśmy sporo czasu na omówienie rozkazów asemblera i odpowiadających im kodów maszynowych. Omówiliśmy różne tryby adresowania pamięci FLASH, RAM i EEPROM. Jeszcze raz podkreślam, że gdybyś miał programować w asemblerze, musiałbyś dużo dokładniej poznać te wszystkie zagadnienia. Natomiast pisząc programy w języku C, nie musisz rozumieć wszystkich tych szczegółów: Ty napiszesz tylko „w skrócie", co chcesz zrobić, a o dobór odpowiednich wersji rozkazów maszynowych i o inne liczne drobiazgi zadba kompilator. Co prawda Ty nie będziesz wiedział, jak kompilator „rozpisze" Twoje „skrótowe" polecenie na elementarne rozkazy kodu maszynowego, ale ogólnie biorąc, kompilator języka C robi to zdecydowanie lepiej niż kompilator BASCOM-a (program wpisywany do procesora jest krótszy i „szybszy"). Tylko wtedy, gdyby czas wykonywania poszczególnych fragmentów programu był sprawą krytyczną, a Ty chciałbyś mieć stuprocentową kontrolę nad sytuacją, do programu pisanego w C możesz dodać tzw. wstawki asemblerowe. Ale to wyższa szkoła jazdy. Na razie nie musisz zawracać sobie tym głowy.

Niemniej zachęcam, żebyś wrócił jeszcze raz do pierwszej części artykułu o asemblerze, by uporządkować i powiązać podane informacje. Zachęcam, byś samodzielnie nieco bliżej zapoznał się z omówionymi zagadnieniami. Nie musisz zgłębiać i rozumieć wszystkich szczegółów, ale zajrzyj do kart katalogowych procesorów AVR oraz na strony internetowe, omawiające te zagadnienia.

Piotr Górecki

SREG – AVR Status Register **Rys. 20**

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Rys. 21

Numer komórki RAM	Nazwa rejestru	Bity rejestru bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0x3C	EIFR							INTF1	INTF0
0x3D	EIMSK							INT1	INT0
0x3E	GPIOR0	GPIOR0[7:0]							
0x3F	EECR			EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
0x40	EEDR	EEDR[7:0]							
0x41	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
0x42	EEARH					EEAR11	EEAR10	EEAR9	EEAR8
0x43	GTCCR	TSM						PSRASY	PSRSYNC
0x44	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0			WGM01	WGM00