

Az ATMEL processzorok PIC-es szemmel

Vagy nyolc évig programoztam PIC-et és bár évekkkel ezelőtt történt, még vissza tudok emlékezni az áttérés nehézségeire. Mivel mostanában több olyan kérdést kaptam, amiből az derült ki, hogy PIC-es gyakorlatuk miatt egyes érdeklődők félreértettek ezt-azt, megpróbálom PIC-es szemmel bemutatni az Atmel-es családot.

Nézzük először az előnyöket, aztán azt, amiről az Atmel-es hittérítők szemérmesen hallgatnak, illetve hogyan lehet ezeket a hátrányokat csökkenteni.

Előnyök:

- Minden Atmel processzor a kezdetektől fogva áramkörben beforrasztva is ugyanolyan módon programozható. Nincs tehát a PIC-eknél megszokott reflex, hogy megjelent egy új család, vegyél új programozót.
- A programozó egy darab HC244 IC-vel megoldható, bár sokan még ezt is megspórolják, akiknek izmosabb printer portjuk van, egyszerűen pár szál madzagból áll a programozójuk.
- A fordító ugyanúgy ingyenes, mint az MPLAB, mind az assembly, mind a C esetében. Nem csak Windows alatt, hanem a Linux világ is erősen támogatja a processzorcsaládot.
- Egy megszokott funkció a processzor minden tagjánál ugyanott van. Itt nem az a koncepció, hogy a nagyobb processzornál máshova kerül a többletfunkció, hanem a kicsiből maradnak el ehhez képest bizonyos, a nagyban meglevő elemek. Így pl. a 2313-nak nincs A és C portja, csak B és D. Ez akkor jön jól nagyon, ha a programozó menet közben „kinő” egy processzort, nem kell átírnia a programját, mert hiszen pl. az UART a D porton van akár kicsi, akár nagy processzorról van szó.
- A családnak még a legkisebb tagjában is 32 regiszter van, melyek közül egyeseket bizonyos utasítások regiszterpárokként használnak a PIC-ben megszokott FSR-hez hasonlóan. Még a kis processzoroknak is három ilyen pointerük van (**X**, **Y** és **Z**), amelyek bármelyike használható a beépített RAM címzésére, írására, olvasására. Az egyik ezek közül a saját programmemória címzésére is alkalmas. (**Z**)
- Ezekkel a regiszterek úgy is tudnak FSR-szerűen működni, hogy az utasítás után automatikusan növelik vagy csökkentik magukat. (Pl. növelés az olvasás után: **ld r20, X+**, míg csökkentés az olvasás előtt: **ld r20, -X**). Ez akkor is automatikusan megtörténik, ha pl. a pár alsó tagja 0xff-ből 0-ba fordul át. A pár felső tagja automatikusan követi, teljes egészében 16 bites pointerként viselkednek, ugyanakkora végrehajtási idő alatt.
- E párok közül kettő még úgy is használható, hogy egy nem az adott címről, hanem attól egy adott mértékben eltolt címről akarok olvasni vagy írni. (Pl. **ldd r20, Y+8** nem az Y regiszterpár által mutatott címről, hanem annál 8-cal magasabb címről olvas.)

- A beépített RAM terület közvetlenül is írható/olvasható, nem csak pointerrel. (Pl. **lds (memcim),r20**) Természetesen bármely regiszterbe beolvastathatók, bármelyikből írhatók memóriába.
- Nincs W regiszter. Mind a 32 regiszter tudja mindazokat az aritmetikai műveleteket, amikben PIC esetén az egyik operandust a W-be kell tenni. (Vagy ha úgy tetszik, 32 db W regiszterem van.) Az **add r20,r21** és az **add r21,r20** ugyanazt a regiszterpárat adja össze ugyan, de azzal, hogy melyiket írom előbb, én döntöm el, hogy melyikbe kerüljön a végeredmény. A szubrutin hívásokból mélysége nem kötött, csak a RAM szab határt neki.

Korlátok:

- A 32 regiszter (**r0...r31**) mindegyike tud ugyan aritmetikai műveletet, de konstansokkal az alsó 16 nem tölthető fel közvetlenül. (Azaz pl. az **ldi r15,3** nem létezik, csak de az **ldi r16,3** már igen.)

E hátrány csökkentése:

A programom elején ilyesfajta üresjáratot csinálok, amikor úgyis kell egy kis időzítés az elektronika analóg részeinek ébredéséig.

```
#define nulla r8
#define egy r9
stb.
```

```
ldi r16,0
mov r8,r16
ldi r16,1
mov r9,r16
ldi r16,2
mov r10,r16
ldi r16,4
mov r11,r16
ldi r16,8
mov r12,r16
ldi r16,0xf
mov r13,r16
ldi r16,0xf0
mov r14,r16
ldi r16,0xff
mov r15,r16
```

Azaz a másra nehezkesebben használható regisztereket olyan konstansokkal, bitmaszkokkal töltöm fel, amiket a programomban gyakran használok. Ezzel a programban utasításidő megtakarítást tudok elérni, mert vannak regisztereim előkészített értékekkel. Pl. nyolcasával növeléshez nem kell betöltenem a 8-at, hogy hozzáadhassam máshoz, pl ha r20-at akarom nyolcasával növelni, akkor nem az lesz, hogy:

```
ldi r21,8
add r22,r21
```

Hanem csak
add r22,nyolc

Kevesebb idő és nem kellett az r21-et sem rontanom.

- Bár értelmezett pl. a subi r16,konstans utasítás, hiányzik a párja. Nincsen addi r16,konstans utasítás.

A hátrány csökkentése:

Valójában nincs hátrány. A konstans helyett a komplementjét még fordítási időben lehet képezni, így a hiányzó utasítás teljes értékű helyettesítője a subi r16,-konstans (mínusz konstans) utasítás.

- Az aritmetika tekintetében ortogonális utasításkészlet bitműveletekre, egyes speciális utasításokra, (pl. portműveletek) nem valósul meg. Nem lehet pl. tetszőleges portokra feltételes bitteszt utasítást kiadni. Ezeket a hatásköröket egyszerűen tudni kell. Még a kezdetem kezdetén szerkesztettem ehhez egy két oldalas referenciát, amin színekkel jelölve vannak az utasítások hatáskörei. Ez kezdetben talán túl tömörnek tűnik, de minden lényeges benne van, később biztosan hasznos lehet.
Innen letölthető: <http://www.evoran.hu/atmel/avrinstr.pdf>