

Számítógép architektúrák alapjai

Készítette: Simon Péter ¹

2021. december 22.

¹Hallgatói jegyzet Durczy Levente előadásai alapján

Tartalomjegyzék

1. Alapfogalmak	3
1.1. Architektúra fogalma	3
1.2. Számítási modellek	3
1.2.1. A számítási modell, az architektúra és a programnyelv kapcsolata	3
1.2.2. Számítási modellek csoportosítása	3
1.2.3. Adataalapú modellek közös tulajdonságai	4
1.2.4. Neumann modell	4
1.2.5. Adatfolyam modell	5
1.2.6. Applikatív modell	5
1.3. Az architektúráról bővebben	5
1.3.1. ISA	6
1.3.2. Fizikai architektúra	6
2. Adattér	7
2.1. Az adattér fogalma	7
2.2. Típusai	7
2.3. Memóriatér	7
2.3.1. Virtuális memória	7
2.4. Regisztertér	8
2.4.1. Típusai	8
2.4.2. Egyszerű regiszterkészlet	8
2.4.3. Adattípusonként különböző regiszterek	8
2.4.4. Többszörös regiszterkészlet	8
3. Adatmanipulációs fa	10

3.1. Az adatmanipulációs fa fogalma	10
3.2. Részei	10
3.3. Adattípusok	10
3.3.1. Elemi addattípusok	10
3.4. Műveletek	12
3.4.1. Utasítás végrehajtás menete	12
3.4.2. Utasítások részei	14
3.4.3. Utasítások típusai címek száma alapján	14
3.5. Operandus típusok	14
3.5.1. Architektúrák osztályozása operandus típusok szerint	15
3.6. Címzési módok	15
3.6.1. Relatív címszámítás	15
3.6.2. Az index regiszter	15

1. fejezet

Alapfogalmak

1.1. Architektúra fogalma

A számítógép architektúra fogalmat először Amdahl, az IBM mérnöke használta először a 360-as család bejelentésekor. Definíciója szerint ez az a struktúra, amit a gépi kódú programozónak értenie kell, hogy helyes programot tudjon írni az adott gépre. Tehát a regiszterek, memória, utasításkészlet, címzési módok és utasításkódok összessége, mind logikai, mind hardveres szinten.

1.2. Számítási modellek

A számítási modell a számításra vonatkozó alapelvek egy absztrakciója. A számítási modelleket a következő absztrakciós jellemzőkkel írhatjuk le:

- min hajtjuk végre a számítást (általában adatokon - adat alapú)
- hogyan képezzük le a számítási feladatot
- milyen módon vezéreljük a végrehajtási sorrendet

1.2.1. A számítási modell, az architektúra és a programnyelv kapcsolata

Egy számítógép tervezését a számítási modellel kell kezdeni, ami meghatározza, hogy mit szeretnénk csinálni. Ehhez szükség van egy specifikációs eszközre, amit a programnyelv képvisel (pl. Neumann modell megvalósítási eszköze a BASIC, Fortran). Ezután jön az architektúra, ami a számítási modell implementációs eszköze, a "vas". Ez hajtja végre az adott programnyelven definiált feladatokat.

1.2.2. Számítási modellek csoportosítása

Számítási modelljük szerint

- szekvenciális
- párhuzamos

Vezérlés meghajtása szerint

- vezérlés meghajtott
- adat meghajtott
- igény meghajtott

Probléma leírása szerint

- procedurális
- deklaratív

Első sorban aszerint különböztetjük meg őket, hogy min hajtjuk végre a számítást. Az adatalapú modellek:

- Neumann modell
- adatfolyam modell
- applikatív modell (igénymeghajtott)

Az adatalapú modelleken kívül léteznek még objektum alapú, predikátum logika alapú, tudás alapú és hibrid modellek. A mai processzorokban a Neumann és az adatfolyam modellek keverednek.

1.2.3. Adatalapú modellek közös tulajdonságai

- az adatok általában típussal rendelkeznek (pl. 16 bit int) - vannak elemi és összetett adattípusok
- a típus meghatározza az adat értelmezési tartományát, értékészletét és az elvégezhető műveleteket

1.2.4. Neumann modell

A Neumann-elvű számítógépek a számításokat adatokon hajtják végre, amiket egy változó értékű változókészlet képvisel. A végrehajtási sorrend vezérlés meghajtott, tehát van egy statikus utasításszekvencia, amit egy speciális regiszter biztosít (program counter). A program counter egy inkrementálódó változó, mindig a végrehajtandó utasításra mutat. A végrehajtási sorrendtől vezérlési feladatokat ellátó utasításokkal lehet eltérni (pl. jump, if).

A Neumann elv követelményrendszere előírja változók létrehozását, adatmanipulációs és vezérlés átadási utasítások deklarációját. Az ilyen nyelveket hívjuk imperatív (parancs) programnyelveknek (pl. C, Pascal, Assembly).

Ezeket a követelményeket az architektúra kielégíti, pl. lehetővé teszi, hogy a memóriában elhelyezkedő változók korlátlan számban módosíthatók legyenek a program futása során. Ezen kívül biztosítja a megfelelő regisztereket az adatoknak és speciális regisztereket mint pl. program counter.

Az adatok és az utasítások a memóriában helyezkednek el. A számítási feladat műveletek elemi műveletek sorozataként értelmezhető. Egy számítási feladat leképezhető adat manipuláló utasítások sorozatával. Az adat manipuláló utasítások az utasítások sorrendjében vannak végrehajtva, ezért ez egy vezérlés meghajtott modell. A vezérlést a program counter biztosítja, a sorrendet a programozó határozza meg. Az explicit vezérlés átadó utasításokkal lehet eltérni az implicit szekvenciától.

Következmények:

- előzmény érzékenység: mivel az adatok változhatnak bármikor a végrehajtás során, a végrehajtás sorrendje nem mindegy
- alapvetően szekvenciális végrehajtást biztosít
- egyszerűen implementálható
- az adatmanipuláló utasítások nem szándékos állapotmódosulást okozhatnak (pl. overflow) - ezeket mellékhatásoknak hívjuk, kezelni kell őket

1.2.5. Adatfolyam modell

A számítást itt is adatokon hajtjuk végre, de:

- az adatokat bemenő adathalmaz képviseli
- egyszeres értékadás lehetséges
- a megoldandó feladatot adatfolyam gráffal és input adatok halmazával képezzük le (a gráfban a csomópontok a szakosodott végrehajtó egységek)
- szakosodott végrehajtó egységeket használ
- a végrehajtást az adat vezérli - adatvezérelt, azaz a szükséges adatok rendelkezésre állásakor azonnal működésbe lép a végrehajtó egység
- az adat meghajtott program utasításai semmilyen szempontból nem rendezettek
- az adatokat utasításon belül tároljuk és nem az operatív tárban
- magas a kommunikációs és szinkronizációs igénye

1.2.6. Applikatív modell

A számítási feladatot egy komplex függvény formájában adjuk meg. Ez is adatalapú modell, de deklaratív jellegű, tehát valamennyi, az adott probléma megoldásához szükséges tény és relációt deklarálunk. Mindezt egy végrehajtó mechanizmus (a függvény) feldolgozza. A vezérlés igény meghajtott (lazy, lusta modell).

1.3. Az architektúráról bővebben

A 70-es években Bell és Newell kibővítette az architektúra definícióját és négy szintet határoztak meg:

- PMS - processor, memory, switches: a számítógépek globális leírása
- programozási szint:
 - magas szint
 - alacsony szint
- logikai tervezési szint
- áramkörüi szint

Az architektúra másik megfogalmazásban a külső jellemzők, a belső felépítés és a működés együttesét jelenti. Megkülönböztetünk logikai és fizikai architektúrát, valamint ezeken belül processzor és számítógép szinteket.

1.3.1. ISA

A processzor szintű logikai architektúra az ISA - Instruction Set Architecture. Ez írja le az utasításkészletet, pl. x86. Az ISA komponensei:

- adattér
- adatmanipulációs fa
- állapottér
- állapot műveletek

1.3.2. Fizikai architektúra

A számítógép szintű fizikai architektúra elemei:

- processzor
- memória
- buszrendszer

A processzor szintű fizikai architektúra négy részre bontható:

- műveletvégző egység (ALU)
- vezérlő
- I/O
- megszakításrendszer

Az I/O és a megszakításrendszer a más rendszerekkel összekapcsoló illesztő eszközök.

2. fejezet

Adattér

2.1. Az adattér fogalma

Az adattér egy olyan tér, mely biztosítja az adatok tárolását oly módon, hogy azok a CPU által közvetlenül manipulálhatók legyenek. Az adattér közvetlenül címezhető a processzor által.

2.2. Típusai

Az adattér két típusát különböztetjük meg:

- memóriatér
- regisztertér

2.3. Memóriatér

A memóriateret leginkább a mérete jellemzi. A címzéséhez címbuszra van szükség, ennek szélessége meghatározza a memóriatér maximális méretét. A címtérnél megkülönböztetjük a modell címtérét és a valós címtérét, mivel az adott installáció nem feltétlen éri el az elméleti modell méretét.

2.3.1. Virtuális memória

A 60-as években a kis memóriák és az elméleti és valós memóriatér eltérő mérete miatt megjelent a virtuális memória. Alap koncepciói:

- két különböző memóriatér létezik: amit a programozó lát (virtuális memória) és a CPU által látott fizikai memória
- létezik olyan, a felhasználó számára transzparens folyamat, amely a program futása közben az éppen nem használt adatokat a valós memóriából a virtuális memóriába mozgatja, majd szükség esetén visszateszi
- létezik olyan (egyirányú) folyamat, amely a virtuális memória címeket dinamikusan, tehát futási időben valós címekké alakítja

2.4. Regisztertér

A regisztertér az adattér nagy teljesítményű, általában viszonylag kis része. Általában nem része a címtérnek.

2.4.1. Típusai

- egyszerű regiszterkészlet
- adattípusonként különböző regiszterek
- többszörös regiszterkészlet

2.4.2. Egyszerű regiszterkészlet

Az egyszerű regiszterkészlet típusai:

- egyetlen regiszter (akkumulátor), hátránya, hogy nagyon lassú
- több, dedikált adatregiszter
- univerzális regiszterkészlet, jelentős teljesítmény növekedést eredményezett, a programozó szabadon felhasználhatja a regisztereket (pl. a gyakran használt változók folyamatosan a regiszterben maradhattak)
- stack regiszterek

Stack regiszterek

A stack regisztereknél egy stack pointer (SP) mutat a "zsák" tetejére (mindig az utoljára betöltött adat). Előnye, hogy nem kell címezni, így egyszerű, rövid utasításokkal kezelhető és gyors. Hátránya viszont, hogy mindig csak a tetejéhez férünk hozzá, így az operandus kiolvasás csak szekvenciálisan történhet. A szekvenciális kiolvasás lassú, szűk keresztmetszetet jelent sok adatnál.

2.4.3. Adattípusonként különböző regiszterek

Itt különféle adattípusokhoz különféle regiszterek állnak rendelkezésre, elsősorban lebegőpontos adatoknál. A lebegőpontos regisztereknél külön van bontva a mantissza és a karakterisztika, mivel mindkettőn műveletet kell végezni, így lehetőséget ad a párhuzamosításra a külön tárolás. Az Intel 1998-ban bevezette még a SIMD adattípust, amit multimédiás alkalmazásokhoz használtak.

2.4.4. Többszörös regiszterkészlet

Ez a legfejlettebb megoldás, egymásba ágyazott eljárások gyorsítására szolgál. Ehhez szükséges a kontextus fogalmának bevezetése. A kontextus a regisztertér állapota az állapotterrel együtt.

Az eljárások közötti váltásoknál szükséges a kontextusok közötti váltás, amit a kontextuskapcsoló végez. Ha a kontextusok közötti váltásnál a kontextust a memóriatérből kell betölteni, nagyon lassú lesz, ha viszont regiszterek között kell csak váltani, akkor gyors. Ezért vezettek be több regiszterkészletet, a cél minden kontextus számára különálló regiszterkészlet biztosítása. A programozó ezek közül csak egyet lát, a kontextus váltásokat a processzor kezeli. A megvalósításhoz szükség van még egy általános regiszterkészletre, ami a regiszterek közti paraméter átadást biztosítja.

Típusai

- több, egymástól független regiszterkészlet, hátránya, hogy a paraméter átadás csak az operatív táron keresztül történhet, ami lassítja az eljárást
- átfedő regiszterkészlet, egymásba ágyazott eljárásokhoz dolgozták ki, lényege, hogy egy regisztert három részre bontottak: ins, locals, outs. Úgy alakították ki, hogy az egyik regiszter kimenő területe ugyanazon a címen helyezkedik el, mint a másik bemenője, így nagyon könnyű az operátor átadás. Hátránya, hogy merev a struktúra és hogy a regiszterkészletek száma korlátozza a regisztertérben tartható kontextusokat.
- stack cache: 1982-ben vezették be, a mai processzorokban is hasonló struktúrákat alkalmaznak. A stack és a közvetlen elérésű cache kombinálása. Kezelése a compiler feladata, ami minden eljáráshoz hozzárendel egy regiszterkészletet. A hozzárendelt regiszterkészlet az aktiválási rekord, aminek az első eleme a stack pointer mutat. A regiszterek nem csak a stack pointeren keresztül érhetők el, hanem közvetlenül is, a displacement pointerrel. Bizonyos korlátok között az aktiválási rekord bármilyen hosszú lehet, tehát a kiosztás rugalmas. Előnye, hogy nincsenek üres helyek, így nincsenek fölöslegesen lefoglalt regiszterek és nincs túlsordulás se.

3. fejezet

Adatmanipulációs fa

3.1. Az adatmanipulációs fa fogalma

Az adatmanipulációs fa megmutatja a potenciális adatmanipulációs lehetőségeket. Bizonyos részfái megmutatják egy konkrét implementáció adatmanipulációs lehetőségeit.

3.2. Részei

- 1. szint: adattípusok, azaz miket értelmезünk az adott architektúrában (pl. 1 byte FX, 2 byte FX, FP, stb.)
- 2. szint: műveletek szintje, azaz az adattípusokkal milyen műveletek végezhetők (pl. összeadás, kivonás, logikai műveletek)
- 3. szint: operandusok típusai (pl. két operandusos, három operandusos). Megkülönböztetünk memória, regiszter és akkumulátor operandus típusokat.
- 4. szint: címzési módok (pl. regiszter+displacement, pc+displacement, index regiszter+displacement, direkt, indirekt)
- 5. szint: gépi kód

3.3. Adattípusok

Megkülönböztetünk elemi és összetett adattípusokat. Elsősorban az elemi adattípusokkal foglalkozunk. Az összetett adattípusok elemiekből épülnek fel, ha különböző típusokból áll össze, akkor rekordnak hívjuk, ha azonos típusokból, akkor megkülönböztetünk vektor (1D tömb), tömb (többdimenziós), szöveg, verem, sor, lista, fa, halmaz adattípusokat.

3.3.1. Elemi adattípusok

Megkülönböztetünk numerikus, karakteres, logikai, pixel és még több adattípust.

Numerikus adattípusok

- FX
- FP
- BCD (binárisan kódolt decimális)
 - pakolt (1 byte két helyiérték, 4 bit egy decimális szám)
 - pakolatlan (zónázott)

FX adattípusok

Kódolás szerint megkülönböztetünk:

- egyes komplement
- kettes komplement
 - előjeles (1, 2, 4, 8, 16 byte)
 - előjel nélküli
- többletes kódolás

FP adattípusok

Megkülönböztetünk:

- normalizált
 - hexára normalizált
 - binárisra normalizált
 - * VAX
 - * IEEE
 - 1x pontosságú (32 bites)
 - 2x pontosságú (64 bites)
 - kiterjesztett pontosságú (128 bites)
- nem normalizált

Általában a normalizáltat használják.

Karakteres adattípusok

Megkülönböztetünk:

- EBDIC (8 bit, 60-as évek)
- ASCII
 - 7 bites (szabványos)
 - 8 bites (kiterjesztett)
- Unicode (2 byte)

Logikai adattípusok

Megkülönböztetünk:

- 1 byte
- 2 byte
- 4 byte
- változó hosszú

Példák: AND, OR eredményei. Általában 1 bit értékes (általában a legmagasabb helyiérték).

3.4. Műveletek

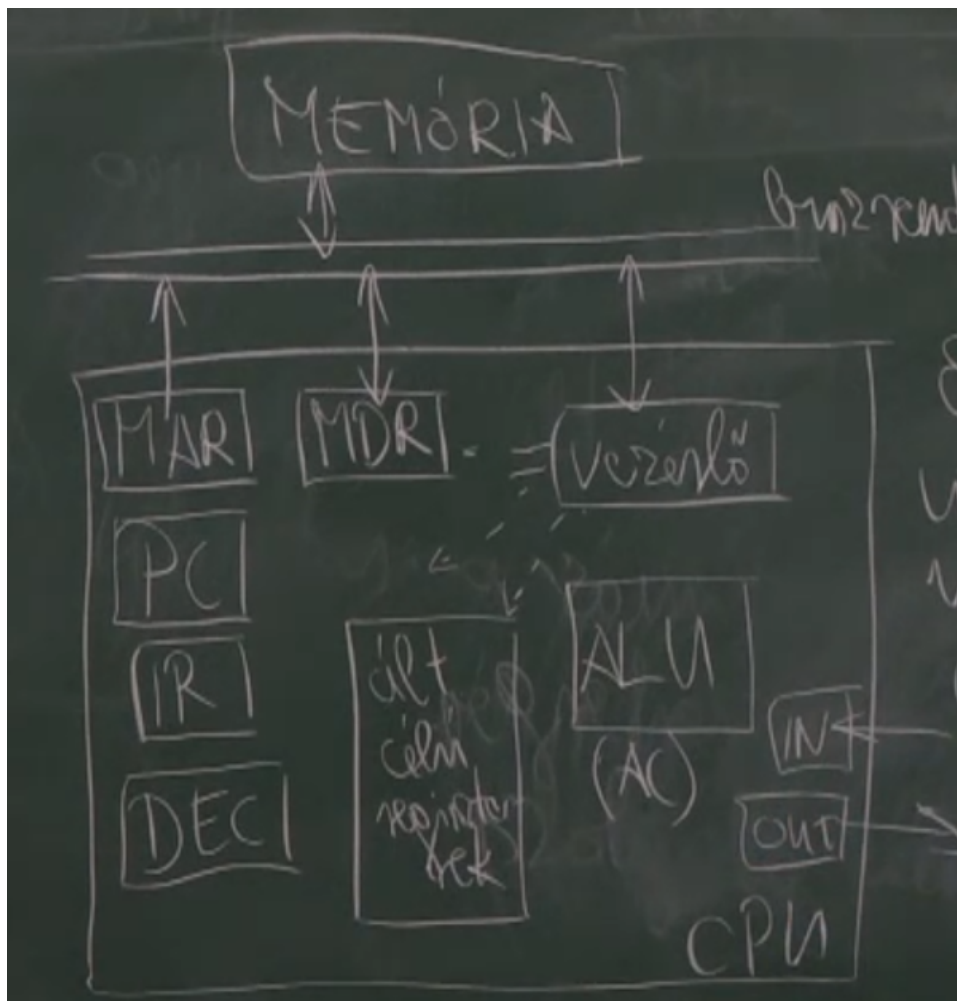
Az adatmanipulációs fa minden művelet esetén megállapítja, hogy milyen utasítás típusok vannak megengedve és milyen operandus típus választható. Az utasítás a számítógép által végrehajtható alapvető feladatok ellátására szolgáló elemi művelet leírása.

3.4.1. Utasítás végrehajtás menete

Egy gépi kódú utasítás általában két részből áll: MK, azaz műveleti kód és címrész. Az MK tartalmazza, hogy mit kell csinálni, a címrész pedig, hogy melyik címen lévő adattal. Pl.: ADD r3,r1,r2. Általánosan az utasítás végrehajtás lépései:

1. A processzor megnézi, hogy van-e megszakítás kérés, ha van, kiszolgálja azt
2. Ha nem történt megszakítás kérés, lehívja az utasítást
3. Végrehajtja az utasítást

A processzor főbb regiszterei láthatók a következő ábrán.



3.1. ábra. Egy CPU főbb regiszterei

Egy gépi kódú utasítás elemi műveletek sorozataként írható fel. Példa: ADD r1, r2 utasítás végrehajtása:

1. FETCH - az utasítást le kell hívni:
 - (a) A processzor a program counter (PC) értékét betölti a memória címregiszterbe (MAR).
 - (b) A címregiszterben lévő címen lévő értéket betölti a memória adatregiszterbe (MDR).
 - (c) A memória adatregiszter tartalmát áttölti az utasításregiszterbe (IR)
 - (d) A program counter értékét egy egységgel inkrementálja
2. Dekódolás és operandus betöltés
 - (a) Az utasításregiszter tartalmát betölti a processzor a dekóderbe (DEC)
 - (b) Az utasítás címrész tartalmát betölti a memória címregiszterbe
 - (c) A memória címregiszter értékének címén található adatot betölti a memória adatregiszterbe
 - (d) A memória adatregiszter tartalmát betölti az akkumulátorba
3. Művelet végrehajtás
 - (a) A dekóder megfelelő (másik) címrésze bekerül a MAR-ba
 - (b) A MAR értékének címén lévő adat bekerül az MDR-be
 - (c) Az AC-be betölti az $AC + MDR$ -t (elvégzi az összeadást)

4. Az eredmény tárolása (STORE)

- (a) A dekóder címrésze (ugyanaz mint a dekódolásnál) bekerül a memória címregiszterbe
- (b) Az akkumulátor tartalmát áttölti az MDR-be
- (c) Az MDR tartalmát betölti az MAR által mutatott címre

Példa: JMP utasítás

1. IR-ből betölti a dekóderbe az utasítást
2. A dekóder címrészét betölti MAR-ba
3. A MAR által mutatott értékkel felülírja PC-t

3.4.2. Utasítások részei

- műveleti kód
- operandus
 - cél (dest op)
 - forrás (source op)

3.4.3. Utasítások típusai címek száma alapján

- 4 címes: cél, forrás 1, forrás 2, következő utasítás címe. Hátránya, hogy nehézkes és merev.
- 3 címes: cél, forrás 1, forrás 2. A következő utasítás címét helyettesítették az auto inkrementálódó PC-vel. RISC architektúrák használják. Előnye a szabadság és a párhuzamosítási lehetőség, hátránya, hogy bonyolultabbak és hosszabbak az utasítások.
- 2 címes: nincs cél operandus, csak forrás 1 és 2. Az eredmény felülírja a forrás 1-et. CISC processzorok használják (pl. x86). Előnye az egyszerűbb utasítás, hátránya, hogy felülírja az egyik forrás operandust.
- 1 címes: be kell tölteni az egyik operandust az akkumulátorba, majd a második operandussal módosítjuk AC tartalmát. Pl. ADD utasítás: LOAD[100], ADD[101]. Hátránya, hogy az utasítások száma nő, viszont az utasítások kisebbek és gyorsabban végrehajthatóak. Többszörös összeadásnál hasznos például.
- 0 címes: pl. NOP, CLEAR D (flag), PUSH, POP. A stack utasítások gyorsaságát is ez adja, nem kell külön megcímezni, hanem a stack pointer értékét használja. Előnye, hogy gyors, hátránya, hogy növelik az utasításkészletet.

A három címes utasítások mindhárom operandusa regiszter típusú, a két címeseknél viszont általában a második operandus lehet memória típusú is.

3.5. Operandus típusok

- AC - akkumulátor: gyors, de csak egy van belőle
- memória: nagy, hosszú címe van és viszonylag lassú
- regiszter: gyors, de korlátozott számú
- verem: nagyon gyors, de csak a tetejét látjuk, így sok adatot nem lehet benne tárolni
- immediate: nem egy címet adunk meg, hanem közvetlenül a programba írjuk be az operandus értékét. Nagyon gyors, de a módosításához módosítani kell a programot.

3.5.1. Architektúrák osztályozása operandus típusok szerint

- szabályos: csak egy fajta operandus típus engedélyezett (kivéve akkumulátor, mert abból csak egy van)
 - akkumulátor (akkumulátor-regiszter és akkumulátor-memória)
 - memória (memória-memória, memória-memória-memória)
 - regiszter (regiszter-regiszter, regiszter-regiszter-regiszter)
 - stack (stack-stack, stack-stack-stack)
- kombinált: a különböző operandus típusok megengedettek egy utasításon belül (pl. regiszter + memória) - nem homogén.

A szabályos architektúrák kezelése és tervezése könnyebb. Ilyenek pl. a RISC architektúrák (mint pl. ARM), amik 3 címes regiszter típusú operandusokkal dolgoznak. A memória eléréséhez két kivételes utasítás használható: LOAD és STORE.

Kombinált architektúra pl. a CISC architektúrák (mint pl. x86).

3.6. Címzési módok

A címzési mód maga a címszámítási algoritmus. Ezt 3, egymástól független elem kombinációja adja:

- címszámítás: jelzi, hogy abszolút, vagy relatív címzést használunk. Az abszolút címzés a teljes címet tartalmazza, a relatívnál pedig egy bázishoz képest számoljuk. A relatív gyorsabb, rövidebbek a címek, de deklarálni kell a bázis címet és a címszámítási algoritmust. Elsősorban a relatív címzés van használatban.
- cím módosítás (opcionális komponens): indexelés, auto inkrementálás, auto dekrementálás. Segítségével könnyebben meghatározható a következő operandus címe.
- deklarált (tényleges) cím meghatározása: azt jelenti, hogy a címet direkt vagy indirekt, illetve valós vagy virtuális címként interpretáljuk.

3.6.1. Relatív címszámítás

A CPU címtére nagyon nagy (4-64 TB), így abszolút címzéssel nagyon nagy címeket kéne kezelni. Ezért inkább a relatív címzést használják. Bázis címnek választható pl. a PC, a stack teteje, valamilyen index regiszter, stb.

3.6.2. Az index regiszter

Megkülönböztetünk egyedi és blokkos címzést, jellemzően blokkokat címzünk és töltünk be a gyorsítótárba. Ehhez szükség van egy index regiszterre.