Beágyazott Rendszerek

Egyetemi Jegyzet

1. Kiadás

Dr. Sütő József

2025

Tartalomjegyzék

1. Bevezetés

A számítógépes rendszerek használata a mechanikus műveletek és az emberi munka kiváltására már a kezdetektől fogva nyilvánvaló volt. Az integrációs technológia exponenciális növekedése az 1970-es évek óta lehetővé teszi számunkra, hogy egy számítógép központi feldolgozó egységét egyetlen integrált áramkörbe (chipbe) helyezzünk, de ezek a feldolgozó egységek még nagyon egyszerűek voltak. Az első mikroprocesszort, az Intel 4004-et, egy számológéphez tervezték, ami nevezetesen egy beágyazott rendszernek tekinthető. Mivel az integrált áramkörök tervezése akkor is és ma is drága és időigényes folyamat, a hardver sokoldalú újra felhasználásának lehetősége a szoftver módosításával kulcsfontosságú áttörést jelentett.

Az integrációs technológia ma már lehetővé teszi, hogy egy teljes számítógépet alakítsanak ki egy chipen belül. Ez volt az egyik fő hajtóereje a beágyazott rendszerek megjelenésének, amik hamar elterjedtek az ipar minden területén. Ami hatalmas lökést adott a beágyazott rendszerek fejlődésének az az autóipar volt. A mai autókban több, mint 100 beágyazott rendszer tálálható, amik a szórakoztató elektronikát, a motort, az üzemanyag levegő keverését, a fékrendszert és még számos egyéb alkatrészt vezérelnek. A járműiparban jól megfigyelhető az egyre kifinomultban működtetés, amit beágyazott rendszerekkel lehetett megoldani.

Napjainkban, a mindennapi életünkben is már mindenhol találkozhatunk beágyazott rendszerekkel. De pontosan mit is jelent a beágyazott rendszer kifejezés? Ha definiálni szeretnénk a beágyazott rendszereket vagy más néven **beágyazott számítógépeket**, akkor azt mondhatjuk, hogy **egy beágyazott számítógép egy eszköz belsejében lévő számítógép, amelyet egy előre meghatározott feladat végrehajtására használnak**. Bizonyos esetekben a beágyazott rendszereket **kiberfizikai rendszereknek** nevezik, ahol a fizikai rendszer, szorosan kölcsönhatásba lép egy számítógépes rendszerrel. A cél, hogy a mechanikus működést egy kifinomultabb vezérlőrendszerrel helyettesítsék. A járműiparnál maradva, erre egy jó példa az üzemanyag befecskendezés. A mechanikus elven működő karburátort ma már felváltott a beágyazott rendszeren alapú motorvezérlő, ami a motorba helyezett szenzorok mért értékeire támaszkodva számolja ki a megfelelő üzemanyag-levegő keverési arányt.

Az is érdemes megemlíteni, hogy minden eszköz, ami a manapság népszerű **dolgok Internete (IoT)** témakörbe tartozik, szintén beágyazott rendszerek. Ahogy a név is utal rá, ezek az eszközök többnyire vezeték nélkül, közvetlen vagy közvetett módon kommunikálnak az Interneten keresztül.

Ha számítógépről beszélünk, akkor a legtöbb embernek a laptopja vagy az asztali számítógépje jut az eszébe. Viszont a teljes kép ettől összetettebb, mivel a számítógépes rendszereket négy nagy csoportra tudjuk bontani. Ezek közül az egyik a **személyi számítógépek** (PC) csoportja, amibe a hordozható és az asztali számítógépek is tartoznak. Ezeknek az eszközöknek a közös jellemzője, hogy általános felhasználási célra lettek tervezve, így számos dolgot tudunk velük végezni a filmnézéstől a szoftverfejlesztésig.

A következő kategória a **szerver számítógépek** kategóriája. A szerverek hasonló technológiával készülnek, mint az asztali számítógépek, de nagyobb számítási, tárhely- és ki-bemeneti kapacitást biztosítanak. Ezen túl, két további fontos jellegzetességet kell kiemelni a szerverek kapcsán. Az egyik az, hogy a felhasználók tipikusan a számítógépes hálózatokon keresztül érik el a szervereket. A másik, a **megbízhatóság**, amit sok esetben a **redundancia** biztosít. Mivel a fizikai eszközök meghibásodnak („…ha valami egyáltalán elromolhat, az el is romlik"), ezért előre fel kell készülni a meghibásodásra. Ez a gyakorlatban azt jelenti, hogy a szervereken tárolt adatok több merevlemezen is tárolódnak (redundánsan), illetve több tápegységgel vannak felszerelve így, ha az egyik meghibásodik, akkor a másik tápegység át tudja venni a helyét. A szerverek esetében nagy figyelmet kap a megbízhatóság, mert a szerver összeomlása tipikusan sokkal költségesebb, mint egyetlen felhasználó PC-jének a meghibásodása.

A harmadik kategória a **szuperszámítógépeké**. Ez a legmisztikusabb kategória, mivel a legtöbb ember nem gyakran lát szuperszámítógépet. A szuperszámítógépek több tízezer processzort foglalnak magukba és hatalmas tárolási kapacitással rendelkeznek. A gyakorlatban nagy számítási teljesítményt igénylő tudományos és mérnöki számítások elvégzésére használják, mint például időjárás előrejelzésre, olajlelőhelyek felkutatására, genetikai projektek.

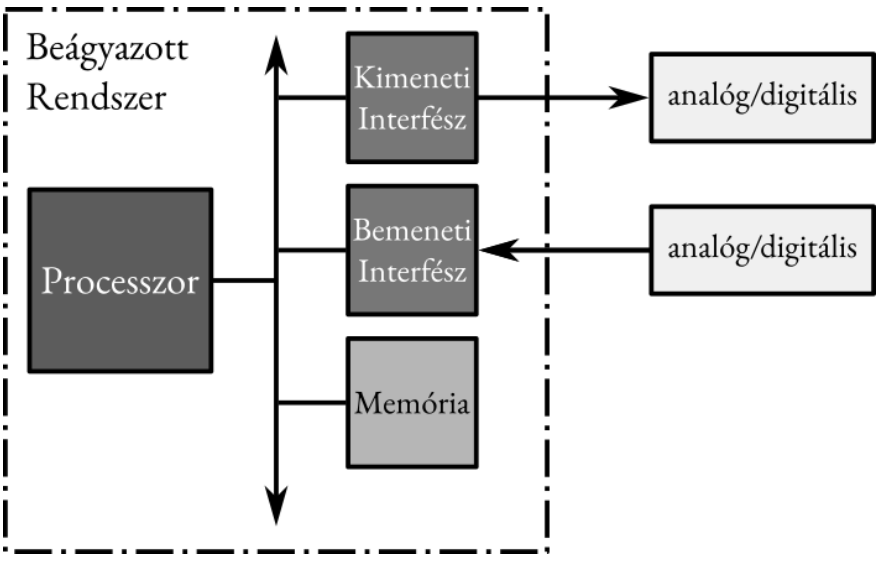
A legutolsó és egyben legtágabb számítógépes kategória a **beágyazott rendszereké**. A beágyazott rendszerek mindenhol körül vesznek bennünket. Megtalálhatók a háztartási eszközökben, a szórakoztató elektronikai eszközökben, a személy és teherautókban,

repülőkben, gyermekjátékokban és még hosszasan lehetne sorolni az alkalmazási területeket.

Tehát egy PC-t nem tekintünk beágyazott rendszernek, mivel az általános felhasználási célra lett létrehozva. Viszont egy fali termosztát már beágyazott rendszer, hiszen csak a fűtési rendszer szabályozására tudjuk használni. Fontos kihangsúlyozni, hogy **a beágyazott rendszereket egyetlen alkalmazás megvalósítására tervezték**! Itt is fontos a megbízhatóság, amit a beágyazott rendszerek egy részénél az egyszerű működési elvre alapoznak, de a szervereknél megszokott redundancia is előfordul.

Most már tudjuk, hogy minden számítógépes rendszert, a négy nagy csoport valamelyikébe sorolható. Azonban a csoporttól függetlenül, minden számítógépes rendszernél megtalálható az 1. ábrán is látható három fő rendszerelem és az azokat összekötő buszrendszer: **központi feldolgozó egység vagy más néven processzor (CPU), a memória (felejtő és nem felejtő), valamint a ki-bemeneti (IO) interfészek**.

Az 1. ábrán is látható fő komponensek, minden számítógépes rendszerben ugyanazokat az alapvető funkciókat látják el: adatok bevitele, adatok kivitele, adatok feldolgozása és adatok tárolása. Logikailag a processzor további két fő alkotóelemre bontható fel. Az egyik az **adatút**, amely a processzor és a memória közötti adatcseréért és ennek feldolgozásáért felelős. A másik a **vezérlőegység**, ami vezérlő jeleket generál az adatút, a memória és az IO komponensek működésének szabályozásához a processzor által aktuálisan végrehajtott utasítás alapján.

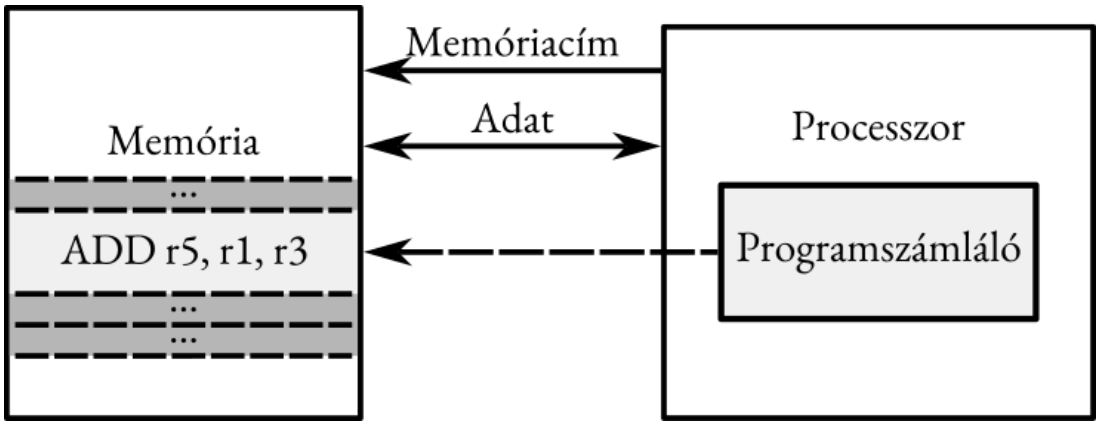


1. ábra. A számítógépes rendszerek fő komponensei.

Ahogy láttuk, a be és kimeneti interfészek minden egyes számítógépes rendszernél fontos szerepet töltenek be. Viszont, a számítógépes rendszer típusától függően az interfészek típusai és a hozzá kapcsolt **perifériás eszközök (perifériák)** teljesen eltérőek lehetnek. Ha veszünk egy személyi számítógépet, akkor ott a jól ismert perifériák a monitor, billentyűzet, nyomtató stb. Ezzel szemben, a beágyazott rendszereknél a perifériák különböző szenzorok (hőmérő, gyorsulás, mágneses tér stb.) és beavatkozók (motorok, kapcsolók).

A fő komponensek közül, elsőként fókuszáljunk a CPU-ra, ami számos belső regiszterrel rendelkezik. A regiszterek többsége egy program utasításainak végrehajtásához szükséges információt tárolják. A belső regiszterek között vannak speciális feladatokat ellátók is. Az egyik ilyen regiszter a programszámláló (PC), amely a soron következő utasítás memóriacímét tárolja. A PC tartalma alapján, a CPU kiolvassa az utasítást a memóriából, dekódolja és végrehajtja.

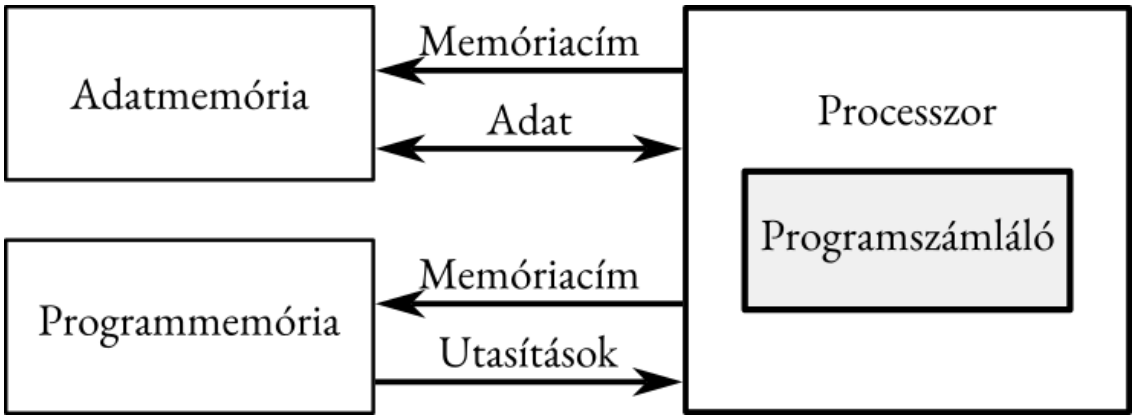
A memória adatokat és utasításokat is tárol, és a sorai (más szóval rekeszei vagy **szavai**) a memóriacím megadásával olvasható vagy írható. Az 1. ábrán a memória szemléltetése le van egyszerűsítve, mert a legtöbb rendszernél a memória több rétegből áll és hierarchikusan épül fel. Sőt, a végrehajtandó program utasításai és a program által felhasznált adatok akár eltérő memóriákban lehetnek letárolva. Az olyan számítógépet, amelynek memóriája adatokat és utasításokat is tárol, **Neumann architektúrájú** számítógépnek nevezzük (2. ábra). Ebben az esetben, a processzor az adatokat és az utasításokat is ugyan abból a fizikai memóriából olvassa ki.



2. ábra. Neumann architektúra.

A Neumann memóriaszervezés alternatívája a **Harvard architektúra**, amely majdnem olyan régi, mint a Neumann. Itt két különálló fizikai memória van fenntartva a program

utasításainak és az adatok tárolására (3. ábra). Mivel dedikált memória van az utasításoknak, így programszámláló a programmemóriára mutat, nem pedig az adatmemóriára. A program és az adatmemória szétválasztása nagyobb teljesítményt biztosít az által, hogy a memória kiolvasás párhuzamosítható. Ebből fakadóan, a Harvard architektúrát ma is széles körben alkalmazzák a **digitális jelfeldolgozó processzoroknál (DSP)**.



3. ábra. Harvard architektúra.

2. CPU karakterisztika

A számítógépes rendszer típusától függetlenül, mindegyikben megtalálható legalább egy processzormag (1. ábra). Viszont attól függően, hogy milyen számítógépes rendszerről van szó, a processzormagok karakterisztikája nagyon eltérő lehet. Az egyik fontos jellemzője a processzormagnak, a belső regisztereinek a mérete. Ennek alapján megkülönböztetünk 8, 16, 32, és 64-bites processzormagokat. Ezek a méretek a 8 többszörösei és nem véletlenül. Korábbi tantárgyaknál már mindenki találkozott a bájt fogalmával, ami 8-bit. Ezt vették alapul a regiszter méretek meghatározásánál, aminek eredményeként egy 32 vagy 64 bites regiszterbe 4 és 8 bájtnyi információt tudunk tárolni. Napjainkban, a személyi számítógépek szinte mindegyike 64-bites processzormagokat használ. Ezzel szemben a beágyazott rendszerek vezérlőegységeinél a 8-bites processzorok sem ritkák. Erre egy jó példa az Arduino Uno mikrovezérlő kártyán is használt 8-bites ATmega328P mikrovezérlő, amit sok esetben használnak beágyazott rendszerek vezérlésére. A 8 bites processzormaggal ellátott vezérlőegységeket alacsony költségű alkalmazásokhoz tervezték. A 16 bites maggal rendelkezőket, a már kifinomultabb alkalmazásokhoz, míg a 32 vagy 64 biteseket a számításigényes alkalmazásokhoz.

A processzormagnak egy másik fontos jellemzőre, a működést ütemező órajel frekvenciája. Ez néhány megahertz (MHz) és több, mint 3 gigahertz (GHz) között mozog. A személyi számítógépeknél ez többnyire 3 GHz felett van. Ezzel szemben, ha ismét az ATmega328P mikrovezérlőt hozzuk fel példaként, akkor ennek az órajel 16 MHz.

Egy további fontos jellemző, a processzor **utasításkészletének** komplexitása. Egy processzor utasításkészlete alatt azokat az elemi utasításokat értjük, amelyeket a processzor képes dekódolni és elvégezni. Azt is szokták mondani, hogy az utasítás készlet interfészként működik a programozó és a processzor között. Habár ma már szinte mindenki valamilyen magas szintű programozási nyelvet használ szoftverfejlesztésre, de ahhoz, hogy megértsük a programok hatékonyságát tisztában kell lennünk azzal, hogy a programot futtató processzor milyen elemi utasításokat képes végrehajtani. Számos processzor létezik, amelyek eltérő utasításkészlettel rendelkeznek. Ha a beágyazott rendszerek világára gondolunk, ott is rengeteg, eltérő célra létrehozott processzor típust tudunk megnevezni, mint például a ARM, a RISC-V, PICx vagy a C55x és C64x

processzorok. Ha a processzorokat az utasítás készletük komplexitása alapján akarjuk kategorizálni, akkor két kategóriába lehet őket bontani. Léteznek komplex utasításkészletű processzorok (**CISC**) és csökkentett utasításkészletű processzorok (**RISC**). A CISC processzorok nagy mennyiségű (tipikusan több száz), különféle utasítást biztosítanak, amelyek összetett feladatokat hajthatnak végre, mint például a karakterlánc-keresést. Általában több, különböző hosszúságú utasításformátumot használnak. Ezzel szemben a RISC processzorok kevesebb (kevesebb, mint 100) és egyszerűbb utasításokat képesek végrehajtani. Ezek a processzorok, általában a betöltés/visszaírás alapú utasításvégrehajtást követik- Ez annyit takar, hogy a műveletek nem hajthatók végre közvetlenül a memóriában, csak a regisztereken, ezért az operandusokat be kell tölteni a regiszterekbe, majd a művelet elvégzését követően, az eredményt vissza kell írni a memóriába.

Az alapvető RISC/CISC jellemzésen túl, a processzorokat annak alapján is jellemezhetjük, hogy milyen címzési módokat támogat az operandusok elérésére, az adatfeldolgozó utasítások hány operandussal képesek dolgozni, illetve, hogy rendelkeznek e hardveres gyorsítóval bizonyos műveletekhez.

3. Beágyazott rendszerek vezérlőegységei

A beágyazott rendszerek vezérlésére több, eltérő képeséggel bíró számítógépes egység is használható a rendszer feladatától függően. Ezek közül a legnépszerűbb a **mikrovezérlő**. A mikrovezérlő egy egychipes számítógép, amely processzort, memóriát és ki-bemeneti interfészeket tartalmaz. A mikrovezérlő kifejezés általában egy viszonylag szerény számítási kapacitású CPU-val rendelkező számítógépre utal, ami egyetlen **integrált áramkörben (IC)** van kialakítva. A mikrovezérlőknek létezik egy speciális változata, amit dedikáltan digitális jelfeldolgozási célokra hoztak létre. Ezzel, majd egy későbbi fejezetben fogunk foglalkozni. Ha már mikrovezérlőről van szó, akkor szeretném felhívni a figyelmet arra, hogy a mikrovezérlő és a **mikroprocesszor** két teljesen eltérő fogalom, amit a hallgatók sokszor összekevernek. A mikrovezérlő egy teljes számítógépes rendszer processzorral, memóriával és ki-bemeneti interfészekkel. Ezzel szemben, a mikroprocesszor kifejezéssel tipikusan a személyi számítógépek processzorára szoktak hivatkozni.

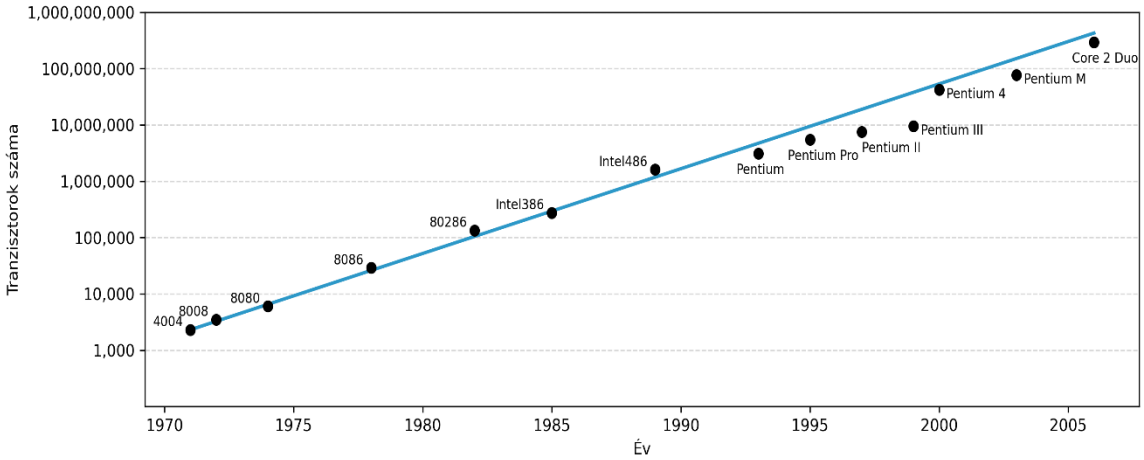
A beágyazott rendszereknek egy másik gyakori vezérlőegysége, az **egykártyás számítógép**. A név onnan ered, hogy egyetlen nyomatott áramköri lapból áll, amelynek a legfontosabb eleme a **System on Chip (SoC)**. Az SoC is egy teljes értékű számítógépes rendszer egyetlen IC-be integrálva, hasonlóan a mikrovezérlőkhöz. Viszont számos lényeges eltérés van a mikrovezérlők és az SoC-k között, ami kiterjed a számítógépes rendszerek mindhárom fő komponensére. Ezek az eltérések arra is visszavezethetők, hogy az SoC-k **integráltsági foka** magasabb a mikrovezérlőkhöz képest. Az integráltsági fok alatt az IC-ben található tranzisztorok számát értjük (később ezzel részletesebben is foglalkozunk). Az SoC-ben általában több processzor mag található, amelyek számítási teljesítményben felülmúlják a mikrovezérlők processzorait. Az SoC-k memóriakapacitásban is felülmúlják a mikrovezérlőket és a ki-bemeneti interfészeik típusa is részben eltérő. Habár a ki-bemeneti interfészek között vanna olyanok, amelyek mindkét eszközben megtalálhatóak, mint például a soros kommunikációt támogatók, de az SoC-knál olyan speciális interfészek is fellelhetők, amik a mikrovezérlőkben nem. Erre egy jó példa a kijelző vezérlő, amely azért fontos az SoC-kban, mert a mobil eszközeinkben (PMD) is SoC-k töltik be a vezérlőegység szerepét. De, mielőtt teljesen leírnánk a

mikrovezérlőt az SoC-vel szemben, ki kell emelni, hogy ma már a legtöbb mikrovezérlő beépítetten tartalmaz analóg-digitális átalakítót, ami az SoC-kből hiányzik. Ezen felül, az energiafogyasztás is a mikrovezérlők mellett szól. Egyrészt, a mikrovezérlők kevesebb energiát használnak fel az SoC-khez képest. Másrészt, az időzített alvó állapot sem oldható meg olyan egyszerűen az SoC-val ellátott egykártyás számítógépeknél, mint a mikrovezérlőknél, holott ennek nagyon fontos szerepe van az energiamegtakarításban.
Végül, az **FPGA (Field Programmable Gate Array)** is használható beágyazott rendszerek vezérlőegységeként. Ez egy kicsit kilóg a sorból, mivel az FPGA-kat áramkörök tervezésére készítették és nem egy szoftver utasításainak végrehajtásra. Az utasítások végrehajtásához processzorra van szükség. Az újabb FPGA-k közül már több is tartalmaz fizikai processzor magot, így az utasítás végrehajtás is megoldható. De, ha fizikailag nem is érhető el processzor mag, akkor is ki lehet alakítani úgynevezett **lágy processzor** magokat az FPGA „szövetében". Sajátot is lehet tervezni, de a gyártók is biztosítanak lágy processzor magokat a fejlesztők számára, ami jócskán megkönnyíti a fejlesztők életét.

4. Integrált áramköri technológia

Napjainkban az okos eszközök korszakát éljük. Ahogy már korábban említettem, az okos eszközökben, valamint az egykártyás számítógépeknél valamilyen SoC tölti be a központi vezérlőegység szerepét. Az SoC egy teljes számítógépes rendszert takar, ami egy megközelítőleg 2x2 cm méretű IC-be van integrálva. Azokban, akiknek van vagy volt asztali számítógépe, felmerülhet a kérdés, hogy hogyan tudnak egy teljes számítógépes rendszert, tipikusan több processzor maggal (erről később) kialakítani egy ilyen kicsi IC-ben? Erre az integrált áramköri technológia fejlődése válaszol.

Gordon Moore, az Intel egyik alapítója, 1965-ben azt a figyelemre méltó jóslatot tette, hogy az integrált áramkörökben lévő tranzisztorok száma évente meg fog duplázódni. Ezt a jóslatot **Moore törvénynek** nevezték el (nem túl megfontoltan). Egy évtizeddel később, a jóslata annyival módosult, hogy a tranzisztorszám kétévente duplázódott. Ez a jóslata hosszú ideig pontosnak bizonyult, és a Moore törvénye 50 évig érvényes volt, ahogy azt a 4. ábra is szemlélteti. Az ábrán az Intel cég processzorait látjuk, a megjelenési évük és a bennük található tranzisztorok száma függvényében. A grafikon egy **exponenciálisan** növekvő tendenciát mutat a tranzisztorszám növekedésben.

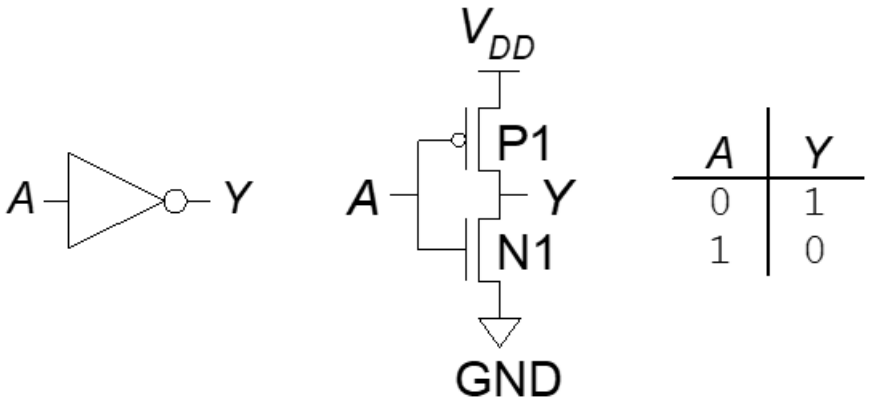


4. ábra. Tranzisztorszám növekedés az Intel processzorokban.

A Moore törvény exponenciális tendenciáját követve, az évek során, a tranzisztorok száma több százról több száz millióra növekedett az integrált áramkörökben. Sajnos, a 2010-es éveket követően a Moore jóslat már nem állta meg a helyét! Ennek szemléltetésére vegyünk egy 2010-ben megjelent Intel mikroprocesszort, ami 1 170 000000 tranzisztorral

rendelkezett. A Moore törvény szerint a várható tranzisztorszám 2016-ban 18 720 000000 kellett volna lennie. Ehelyett, az adott évben megjelent Intel mikroprocesszor „mindössze” 1 750 000000 tranzisztorral rendelkezett (közel tízszeres eltérés). Habár a félvezető technológia még most is fejlődik, de lassabban és nem olyan előre megjósolható módon, mint a múltban.

Eddig tranzisztorokról beszéltünk, de miért olyan fontos a tranzisztorszám? Egyszerűen azért, mert a tranzisztor a digitális áramkörök elemi „építőkövének” tekinthető. Ebben a jegyzetben, egy tranzisztort, elektromos árammal vezérelt, kétállásos kapcsolónak fogunk tekinteni, ami nyitott és zárt állapotban lehet. Ha vannak tranzisztoraink, akkor azokból bármelyik elemi logikai kaput fel tudjuk építeni. Ennek szemléltetésére vegyük a legegyszerűbbet, az inverz kaput, aminek a szimbóluma, felépítése és igazság táblája az 5. ábrán látható. A kapuk megvalósításához MOSFET (Metal-Oxid-Semiconductor Field-Effect-Transistor) térvezérlésű tranzisztorokat szoktak használni, abból is mind a P, mind az N típusút. A P és N típusú tranzisztorokat vegyesen alkalmazzák a kapu létrehozásához, ahol a P típusú tranzisztor fogja a kapu kimenetét a tápfeszültségre (VDD) felhúzni, míg az N típusú húzza le a kimenetet a földre (GND). Az 5. ábrán jól látható, hogy a bemeneti érték (A) vezérli a tranzisztorok kapuit. Ha A = 0, akkor a P1-el jelölt tranzisztor zár és a kimenet Y = 1. Ha A = 1, akkor az N1-el jelölt tranzisztor kerül zárt állapotba és így Y = 0. Ha ezt a működési elvet összevetjük a tranzisztor igazságtáblázatával (5. ábra), akkor azt a konklúziót vonhatjuk le, hogy ha egy P és egy N csatornás MOSFET tranzisztort az 5. ábrán látható módon kötünk össze, azzal egy inverz kaput hozunk létre. Ez a kapu igazságtáblájával igazolható, hiszen az összes bemeneti kombinációra a megfelelő kimeneti értéket adja.



A	Y
0	1
1	0

5. ábra. Az inverz kapu szimbóluma (baloldalt), felépítése (középen) és igazságtáblája (jobboldalt).

Ahogy digitális technikán tanultál róla, ha vannak logikai kapuink, akkor azokból tetszőleges **kombinációs logikai áramköröket** lehet kialakítani. Például komparátor, összeadó, szorzó, osztó, vagy dekódoló áramkört. Sőt, logikai kapukból tárolókat is meg tudunk valósítani, gondolj csak a latch-ekre. Ha elérhetők a szükséges kombinációs áramköri komponensek és vannak tárolóink is, akkor ezek felhasználásával létre tudunk hozni egy processzort. Azt kell látni, hogy a digitális áramkörök hierarchikusan épülnek fel, ahol a tranzisztorok töltik be az elemi építőkövek szerepét. Ha van elegendő tranzisztor, akkor ez lehetővé teszi a teljes számítógépes rendszer kialakítását egyetlen IC-ben.

Mivel a tranzisztorszám (integráltsági fok) egy nagyon fontos jellemzője az integrált áramköröknek, így nem meglepő módon az integrált áramkörök osztályozására különböző kategóriákat vezettek be (SSI, MSI stb.), a tranzisztorszám alapján. Ennek egy összefoglalása található az 1. táblázatban. Annak ellenére, hogy létezik ULSI (ultra nagy léptékű) kategória a több tízmillió tranzisztort magába foglaló IC-kre, de ennek ellenére, a gyakorlatban sokszor ezekre is **VLSI** IC-ként szoktak hivatkozni.

Kategória	**Tranzisztorszám**	**Megjegyzés**
SSI (Small-Scale Integration)	10 – 100	Egyszerű logikai kapuk, flip-flopok, pl. 7400 sorozat NAND kapu
MSI (Medium-Scale Integration)	100 – 1000	Multiplexerek, számlálók, dekódolók
LSI (Large-Scale Integration)	1000 – 100000	Egyszerű mikroprocesszorok, memóriák (pl. 1 Kbit RAM)
VLSI (Very Large-Scale Integration)	100000 – 10 000000	Mikroprocesszorok, mikrokontrollerek (pl. Intel 80486)
ULSI (Ultra Large-Scale Integration)	10 000000 – 100 000000+	Modern CPU-k, GPU-k, SoC-k

1. táblázat. Integrált áramköri kategóriák.

6. Beágyazott rendszerek jellemzői

6.1 Beágyazott rendszerek korlátjai

A beágyazott rendszerek számos korlátozás néznek szembe az általános célú számítógépekkel szemben. Ezek közül az egyik a **méretbeli megkötés**, ami nagymértékben függ a felhasználás jellegétől. Vegyünk például egy orvosdiagnosztikai eszközt, mondjuk egy vércukorszintmérőt, amit a megfigyelt személynek egész nap magán kell viselni. Ahhoz, hogy az eszköz ne zavarja a viselőjét, az eszköznek kis méretűnek és könnyűnek kell lenni. Ebből adódóan ez egy fontos tervezési szempont a gyártók számára. Egy másik fontos megkötés a beágyazott rendszerekkel szemben az **energiafelhasználás**. A rendszerek többsége akkumulátorról üzemel, ahol a cél az, hogy egy teljesen feltöltött akkumulátorral a lehető leghosszabb ideig működjenek. Ehhez szükség van az energia felhasználását minimalizáljuk. Szemléltetésként, a személyi számítógépek átlagos energiafelhasználása néhány száz watt, ezzel szemben a beágyazott rendszerek többsége csak néhány wattot (vagy mikrowattot) fogyasztanak. Az energiafogyasztáshoz kapcsolódóan azt gondolhatnánk, hogy a beágyazott rendszerben a legtöbb energiát a vezérlőegység használja fel. De ez sok esetben nem igaz, mert amennyiben a rendszer tartalmaz kijelzőt, akkor ennek az energiafogyasztása meghaladhatja a vezérlőegységét. A kijelző a felhasználási felületet biztosítja. Ez egy fontos része számos beágyazott rendszernek, így akár annak alapján is lehet csoportosítani a rendszereket, hogy rendelkeznek kijelzővel (headed) vagy sem (headless).

Tegyük fel, hogy a te feladatot a rendszer megtervezése és úgy döntesz, hogy az energiafelhasználás csökkentése érdekében nem használsz kijelzőt. Viszont ebben az esetben is meg kell oldani a felhasználó és a rendszer közötti kommunikációt. Erre több lehetőség is van. Az egyik, hogy egy parancssoros felületet és egy parancsértelmezőt implementálsz a rendszert vezérlő szoftverbe. Tipikusan, a felhasználó valamilyen soros kommunikációs protokollon (ezek később kerülnek bemutatásra) keresztül tudja elérni ezt a felületet. Egy másik megoldás, amivel a routereknél találkozhattál, az a webes felhasználói felület, ami szintén a rendszert vezérlő szoftverbe van implementálva.

A rendszer komponenseinek megfontolt kiválasztásán túl, egy másik nagyon fontos lehetőség az energiafelhasználás csökkentésére a vezérlőegység **alvó állapotának**

kihasználása. Számos beágyazott rendszernél nincs szükség arra, hogy egész nap folyamatosan üzemeljen. Sőt vannak olyan rendszerek, amelyek naponta csak néhány percet üzemelnek és a nap hátralévő részében üzemen kívül vannak. Erre egy automatizált rovarcsapdát említenék meg példaként, aminek én voltam a fejlesztője. A csapda váza polikarbonát lapokból lett létrehozva, amiben egy kamerát tartalmazó beágyazott rendszer és a feromonkapszulával ellátott ragacsos lap kapott helye. A csapda működési elve az volt, hogy minden nap reggel kapcsoljon be, készítsen egy képet és egy tanuló modellre támaszkodva számolja meg az elfogott rovarokat, majd az eredmény vezeték nélküli kapcsolaton keresztül küldje el egy távoli szervernek. Ennek a rendszernek naponta csak néhány percre kellett aktív állapotba kerülni, majd ezt követően alvó állapotba kellett átmennie.

Szoftveres szempontból, a vezérlőegységnek alvó állapotba küldése egyszerű. A vezérlőegység felébresztése viszont már más kérdés. Erre két lehetőség közül szoktak választani. Az egyik a külső lábon keresztül érkező ébresztő esemény. Amit egyszerűen úgy lehet elképzelni, hogy ha egy adott lábán a vezérlő egységnek megváltozik a feszültség érték, akkor ez felébreszti a vezérlőegységet alvó állapotból. A másik, talán az előzőnél is praktikusabb ébresztő esemény, a **belső számlálók** által generált ébresztés. Ekkor előre, programozott módon meg tudjuk adni, hogy az alvó állapotba küldéstől számítottan mennyi idő múlva kell a vezérlőegységet felébreszteni.

6.2 Megbízhatóság

A **megbízhatóság** minden szoftvernél és rendszernél fontos, de ez még nagyobb hangsúlyt kap a beágyazott rendszereknél. Gondoljunk csak abba, hogy egy olyan rendszert tervezünk, amit majd országszerte szétszórva kell telepíteni, mint például egy meteorológiai állomást. Ha ezek meghibásodnak valamilyen hardveres vagy szoftveres hiba miatt, akkor a javításuk sokkal több időbe és sokkan nagyobb költségbe fog kerülni, mint a hibák javítása például a webes alkalmazásoknál. Sok beágyazott rendszernek akár évekig kell működnie bármilyen emberi beavatkozás nélkül. Ilyen hosszú távon a kisebb hibák is, mint például a **memória szivárgás** idővel komoly problémává válhatnak. Ebből adódóan fontos a rendszer alapos, sokszor heteken keresztül történő tesztelése és a rendszer erőforrásainak monitorozása.

6.3 Valós idejű működés

Sok beágyazott rendszernél a feladatok elvégzése határidőhöz kötött. A határidő az az időpont, ameddig a feladat elvégzéséhez szükséges számításokat be kell fejezni. Miután a rendszert vezérlő szoftver megkapja a bemeneti adatokat, a kívánt kimenetet elő kell állítani a határidőig, különben a program nem működik megfelelően, még akkor sem, ha végül az előállított kimenet funkcionálisan helyes.

A valós idejű működés alapján, a beágyazott rendszereket két kategóriába sorolhatjuk: **kemény (hard)** és **lágy (soft)**. Mindkét esetben a rendszernek egy előre megszabott határidőn belül kell elvégezni a feladatát. A fő különbség abból fakad, hogy milyen következményekkel jár ennek a követelménynek a megsértése. A kemény valós idejű rendszereknél, ha a kimenet nem áll elő a megszabott határidőig, az meghibásodást vagy akár a rendszer összeomlását eredményezi. Itt a határidő túllépése életeket is veszélyeztethet. Gondoljuk csak a járművek ABS (Anti-lock Braking System) rendszerére.

A lágy rendszereknél a határidő elmulasztása nem okoz biztonsági problémákat, de romló teljesítményt és elégedetlen ügyfeleket eredményez. Itt példaként gondolhatunk egy nyomtatóra, ahol a határidők elmulasztása kevert oldalakhoz vezet.

6.4 Biztonság

Sok beágyazott rendszerben nem fordítanak figyelmet a biztonsági szempontokra, ami sebezhetővé teszi a rendszereket a támadásokkal és az adatlopással szemben. Ez egyre kritikusabb kérdés, mivel egyre több beágyazott rendszert használnak biztonságkritikus eszközökben, amelyeket az emberek naponta használnak. Például gondoljunk a járművekre vagy az orvosi berendezésekben. Ráadásul ezen rendszerek közül sok közvetlenül vagy közvetett módon csatlakozik az Internethez (IoT).

A biztonság a rendszer azon képességére vonatkozik, hogy megakadályozza a rosszindulatú támadásokat. A biztonság három tényezőtől függ. Az egyik a **titkosítás**, amely biztosítja, hogy csak a feladó és a címzett legyen képes értelmezni az üzenet tartalmát (az eszköz által generált vagy a hozzá érkező adatokat üzenetekbe csomagoljuk). Ez úgy érhető el, hogy a feladó titkosítja (kódolja) az üzenetet kiküldés előtt, míg a fogadó visszafejti (dekódolja) azt a beérkezést követően. Az üzenet kódolásának a célja az, hogy azt az üzenetet elfogó támadó ne tudja közvetlenül elolvasni. Egy fontos kritérium a

kódolással szemben, hogy a kódolt üzenetnek nehezen feltörhetőnek kell lennie. Erre a feladatra kétféle titkosítási módszert is alkalmazhatunk. Az egyik a **szimmetrikus kulcsú** titkosítás, a másik az **aszimmetrikus kulcsú** titkosítás. A szimmetrikus kulcsú titkosításnál az AES-t (Advanced Encryption Standard) használják standard titkosítási algoritmusként. Az AES 128 bites blokkokban kódolja az adatokat, és három különböző méretű kulcsot tud használni: 128, 192 vagy 256 bit.

Az aszimmetrikus kulcsú titkosítás két részre osztja a kulcsot: egy **privát kulcsra** és egy **publikus kulcsra**. A kettő úgy kapcsolódik össze, hogy a privát kulccsal titkosított üzenet visszafejthető a publikus kulcs segítségével, de a privát kulcs nem következtethető ki a publikus kulcsból. Mivel a publikus kulcs nem árul el információt az üzenet kódolásáról, nyilvános helyen tárolható, hogy bárki felhasználhassa. Ezt az elvet használja az RSA (Rivest, Shamir, Adleman) is, amely talán a legismertebb kétkulcsos algoritmus.

A biztonság második tényezője az **azonosítás**. Ez abban segít, hogy egyértelműen be lehessen azonosítani az üzenet küldőjét, ezzel elkerülve a harmadik féltől származó „megtévesztő" üzenetek feldolgozását. Az üzenet küldőjének azonosítására használt technika, a **digitális aláírás**, ami szintén az RSA algoritmusra támaszkodik. A digitális aláírás elképzelése az, hogy az üzenetről egy lenyomatot készít, amit bekódol és az üzenettel együtt továbbítja.

Az üzenet lenyomatának (vagy kivonat) elkészítéséhez **hash függvényeket** használnak. A lenyomat generálás célja az aláírás méretének csökkentése (ne a teljes üzenetet kelljen használni), így a lenyomat általában rövidebb, mint maga az üzenet, és nem fedi fel közvetlenül az üzenet tartalmát. Mivel a hash függvény tetszőlege hosszúságú üzenethez egy n-bit hosszúságú lenyomatot készít, így előfordulhat, hogy eltérő üzenetekhez ugyan az a lenyomat generálódik, de ennek a valószínűsége nagyon kicsi. Tehát, a digitális aláírás generálás az aszimmetrikus titkosításra és a hash függvényekre támaszkodik, annak érdekében, hogy az üzenet fogadója egyértelműen be tudja azonosítani a feladót.

A digitális aláírás nem csak a feladó beazonosításában, hanem az üzenet integritásának ellenőrzésében is segít. Az **üzenet integritásának** vizsgálata, a biztonság harmadik komponense. Ez abban segít, hogy jelzi, ammenyiben a feladást követően az üzenet tartalma módosult. Komoly problémákat okozna, ha egy támadó módosítaná vagy lecserélné az üzenet eredeti szövegét egy általa kreált üzenetre.

A beágyazott rendszerek egy jelentős részénél, a fejlesztők nem foglalkoztak a három tényező közül egyikkel sem. Természetesen mindhárom komponens fontos, de most fókuszáljunk csak az üzenet titkosítására. A kérdés az, hogy szükséges-e titkosítani az üzeneteket. A titkosítás szoftveresen vagy hardveresen is megvalósíthatók. Szoftveres szempontból az üzenet kódolása és dekódolás könnyen megoldható, viszont ez plusz számítási terhet jelent a beágyazott rendszerre és ezzel az energiafogyasztást is növeli. Ha a vezérlő eszközben erre van hardveres gyorsító, akkor ez mind időben, mind energiafelhasználásban hatékonyabb lesz, mint a szoftveres megvalósítás, de ilyen dedikált hardver modul csak kevés vezérlőeszközben érhető el (pl.: ESP32 mikrovezérlők). Tehát mérlegelni kell ennek a létjogosultságát, mert vannak olyan rendszerek, amelyek nem feltétlenül igénylik a titkosítást. Vegyünk példaként egy meteorológiai állomást. Ha a támadó képes „elkapni” az állomás üzenetei és azt értelmezni, abból még olyan túl sok haszna nem fog származni. Összességében azt kell mérlegelni, hogy a támadó mekkora haszonra tehet szert, a támadásba fektetett munka függvényében.

7. Teljesítmény mérése

Ma már kézenfekvő, hogy a szoftvereket magasszintű programozási nyelveken fejlesztjük, az elvégzendő feladattól függetlenül, mivel ezek nagy mértékben növelik a programozók produktivitását. A produktivitás növelése abból fakad, hogy a magasszintű nyelvek elfedik az apró részleteket és egy magasabb absztrakciós szintet biztosítanak, egy alacsony szintű nyelvhez képest (Assembly). Az **absztrakció** egy nagyon fontos fogalom, ami segít bennünket a komplexitás kezelésben, az által, hogy elrejti azokat a részleteket, amik nem fontosok a működési mechanizmus megértéséhez. Azonban, ha meg akarjuk érteni egy processzor vagy a teljes számítógépes rendszer teljesítményét, akkor le kell ereszkednünk az elemi utasítások szintjére.

A rendszer teljesítményének mérésére használt egyik metrika a **végrehajtási idő**. Ez egy program végrehajtásához szükséges teljes idő, beleértve a lemezhozzáféréseket, a memória hozzáféréseket, az I/O tevékenységeket, az operációs rendszer terhelését, a CPU végrehajtási idejét és így tovább. Egy másik gyakran használt metrika az **áteresztőképesség**, ami az egységnyi idő alatt elvégzett feladatok számát jelenti. A gyakorlatban különböző teljesítménymutatókra vagy más néven metrikákra, valamint különböző **benchmarkokra** lesz szükségünk a számítógépes rendszerek összehasonlításához. Ebben a jegyzetben mi a végrehajtási időre fogunk támaszkodni egy számítógépes rendszer teljesítményének a mérésekor. A teljesítmény maximalizálása érdekében minimalizálni kell a végrehajtási időt egy adott feladatnál. Ebből adódóan egy számítógép teljesítményét egy adott program végrehajtási ideje alapján is lehet mérni a következő egyszerű összefüggésen keresztül:

$$teljesítmény = \frac{1}{t_i} \tag{1}$$

Erre támaszkodva össze tudjuk hasonlítani a számítógépek teljesítményét egy adott programra nézve. Az (1) formula arra is rámutat, hogy annak a számítógépnek lesz nagyobb a teljesítménye, amelyiken az *i.* tesztprogram futási ideje kisebb volt! Nézzünk erre egy példát.

1. **Példa:** Van két számítógépünk (A és B). Ugyan annak a tesztprogramnak a futási ideje 10 másodperc volt az A gépen és 12 másodperc a B gépen. Melyik volt a hatékonyabb és mennyivel?

   Megoldás: $12/10 = 1.2 \rightarrow$ az A gép 1.2-szer gyorsabb volt, mint a B.

Ahogy korábban említettem, mi a végrehajtási időre fókuszálunk a számítógépes rendszere teljesítményének mérésekor. Azonban a végrehajtási időt is lehet két különböző megközelítés alapján számítani. Az egyik lehetőség az, amikor mérjük a program futása során **eltelt időt**. Ez egyszerűen meg lehet határozni szoftveresen a program indításakor rögzített idő és a futtatást követő időpont különbségéből. Az eltelt idő az adott program elvégzéséhez szükséges teljes időt jelenti, beleértve a lemezhozzáférés idejét, a memória-hozzáférés idejét, a bemeneti/kimeneti (I/O) adatcsere idejét és az operációs rendszer késleltetését. Viszont, ha csak arra vagyunk kíváncsiak, hogy a processzor mennyi ideig dolgozik a tesztprogram végrehajtásán, akkor a CPU végrehajtási időt vagy egyszerűen csak **CPU időt** kell meghatározni. A CPU idő meghatározásához is egy egyszerű formulát kell követnünk, ami a számítógép processzorát működtető órajel ciklusok számának és az órajel periódus idejének (T_{clk}) szorzata:

$$CPU\ idő = órejel\ ciklusok\ \times\ T_{clk} \tag{2}$$

A (2) formulába szereplő órajel ciklusok száma azt jelöli, hogy hány órajelre van szükség összesen a tesztprogramban szereplő utasítások elvégzéséhez. Fontos kihangsúlyozni, hogy minden esetben, amikor utasításokról beszélünk, a tesztprogramot alkotó elemi (Assembly) utasításaira kell gondolnunk. Mivel az órajel frekvenciája (f) fordítottan arányos az órajel periódus idejével ($f_{clk} = 1/T_{clk}$), így a (2) képletet, át lehet úgy alakítani, hogy abban az órajel frekvenciáját használjuk a periódusidő helyett:

$$CPU\ idő = \frac{órejel\ ciklusok}{f_{clk}} \tag{3}$$

A (2) és (3) képletek rámutatnak arra, hogy a mérnök javíthatja a processzor teljesítményét, a programhoz szükséges órajelciklusok számának vagy az órajelciklus periódus idejének csökkentésével.

2. **Példa:** Van egy tesztprogramunk, ami 8 másodpercig fut egy számítógépen, amiben a processzor órajele 2GHz. Szeretnénk a gép processzorát kicserélni

(jelöljük a régi processzort A-val) egy másikra (B) ahhoz, hogy ennek a tesztprogramnak a futási idejét 6 másodpercre csökkenjen. Tudjuk az új processzorról, hogy 1.2-szer annyi órajelre lesz szüksége a tesztprogram végrehajtásához, mint a korábbi processzornak. Határozd meg, hogy mekkora órajellel kell működtetni az új processzort a kitűzött futási idő eléréséhez.

Megoldás: először meg kell határozni az órajel ciklusok száma az első processzornál a CPU idő képletének felhasználásával: $cy_A = 8 \times 2 \times 10^9 = 16 \times 10^9$. Ezt követően ismét a CPU idő képletére támaszkodva kell meghatározni a B processzor órajelét: $\frac{1.2 \times 16 \times 10^9}{6} = \frac{1.2 \times 16 \times 10^9}{6} = 3.2 \times 10^9 = 3.2\text{GHz}$

Figyeld meg, hogy a CPU idő formulájában nem jelenik meg a tesztprogramot alkotó utasítások száma, de nyilván ez egy fontos tényező és valahol meg kell jelennie. Valójában a CPU idő úgy is felfogható, mint a CPU által végrehajtott utasítások száma szorozva az utasítások végrehajtásához szükséges idővel. Ebből fakadóan az órajel ciklusok száma egyenlő a program utasításainak a száma szorozva az utasítások végrehajtásához szükséges átlagos órajel ciklus számmal (CPI). Itt fontos kihangsúlyozni, hogy a CPI érték tesztprogramonként változik, attól függően, hogy milyen elemi utasításokból áll a program. A CPI értékre támaszkodva a CPU idő korábbi formuláját (3) át lehet írni a következő alakba:

$$CPU\ idő = utasításszám \times CPI \times T_{clk} = \frac{utasításszám \times CPI}{f_{clk}} \qquad (4)$$

Ha egy olyan processzornak a CPU idejét vizsgálnánk, amely minden utasítást egyetlen órajel alatt végez el, akkor a CPI idő mindig 1 lenne a tesztprogramtól függetlenül. Azonban a modern processzoroknál, az eltérő típusú utasítások eltérő számú órajel alatt hajtódnak végre. Ha tudjuk, hogy az eltérő utasítás típusok mennyi órajelet igényelnek és azt is, hogy a tesztprogram kategóriánkként hány utasítást foglal magába (C_i), akkor könnyen ki tudjuk számolni a program végrehajtásához szükséges össze órajelciklusok számát (5). A lenti képletben az *i* változó az utasítás kategóriákat jelöli.

$$órajel\ ciklusok = \sum_{i=1}^{n} CPI_i \times C_i \qquad (5)$$

A szemléltetés kedvéért nézzünk egy példát a CPI érték kiszámítására.

3. **Példa:** Tegyük fel, hogy van két különböző fordító programunk, amivel ugyan azt a C-ben megírt programot fordítjuk le egy adott processzor típusra. Tudjuk, hogy ennek a processzornak az elemi utasításait három kategóriába lehet sorolni (A, B, C) annak alapján, hogy hány órajel alatt hajtódnak végre. A lenti táblázat mutatja be, hogy a két fordító hány elemi utasítással valósítaná meg a programot és azok milyen kategóriába tartoznak. Ennek ismeretében határozd meg a CPI értéket mindkét változatra. Melyik változat a gyorsabb?

	Utasítás csoportok		
	A	**B**	**C**
CPI	**1**	**2**	**3**
1. program kód	3	1	2
2. program kód	5	1	1

Megoldás: Az 1. programkód 6 utasítást hajt végre, míg a 2. kód 7 utasítást. A táblázat adataira támaszkodva ki tudjuk számolni, hogy az 1. kódhoz összesen 11 órajel ciklusra van szükség, míg a 2. kódhoz 10-re. Ebből látszik, hogy a 2. programkód gyorsabb az elsőnél. Az (5) képletet felhasználva, azt kapjuk, hogy az első kód CPI értéke $11/6 = 1.8333$, míg a 2. kód CPI értéke $10/7 = 1.4285$.

A fenti képletek ismeretében az, hogy egy program mennyi ideig használja a processzort, számos dologtól függ. Az egyik maga az az **algoritmus**, amit futtatni kell, mivel ez hatással van az utasítás számra és a CPI értékre. Egy másik a **processzor utasítás készlete**, ami a CPU idő kiszámításához használt mindhárom tényezőre (4) hatással van. Ezen felül a program megírásához használt **programozási nyelv** és a **fordító** is fontos tényezők, mivel ezek is befolyásolják az utasítás számot és a CPI értéket.

Azt is érdemes megjegyezni, hogy a (4) képletben megjelenő órajelciklus idő hagyományosan rögzített volt. Viszont az energiatakarékosság vagy a teljesítmény

ideiglenes növelése érdekében a mai processzorok változtathatják az órajelfrekvenciájukat, ezért a program átlagos órajelfrekvenciáját kell használnunk.
Egy másik metrika a processzorok teljesítményének mérésére (6) a **MIPS** (millió utasítás per másodperc):

$$MIPS = \frac{utasításszám}{CPU\ idő\ \times 10^6} \tag{6}$$

A MIPS egy elég intuitív metrika, mivel minél több utasítást végez el a processzor időarányosan, annál nagyobb lesz a MIPS értéke. Viszont ez nem egy ideális metrika, mivel nem teszi lehetővé az eltérő utasítás készlettel rendelkező processzorok objektív összehasonlítását. Gondoljunk bele abba, hogy ha ugyan azt a magas szintem megírt tesztprogramot lefordítanám egy RISC és egy CISC mikroprocesszorra, akkor is eltérő lenne a MIPS értékeik, ha ugyan annyi időt venne igénybe a program végrehajtása mindkét esetben.

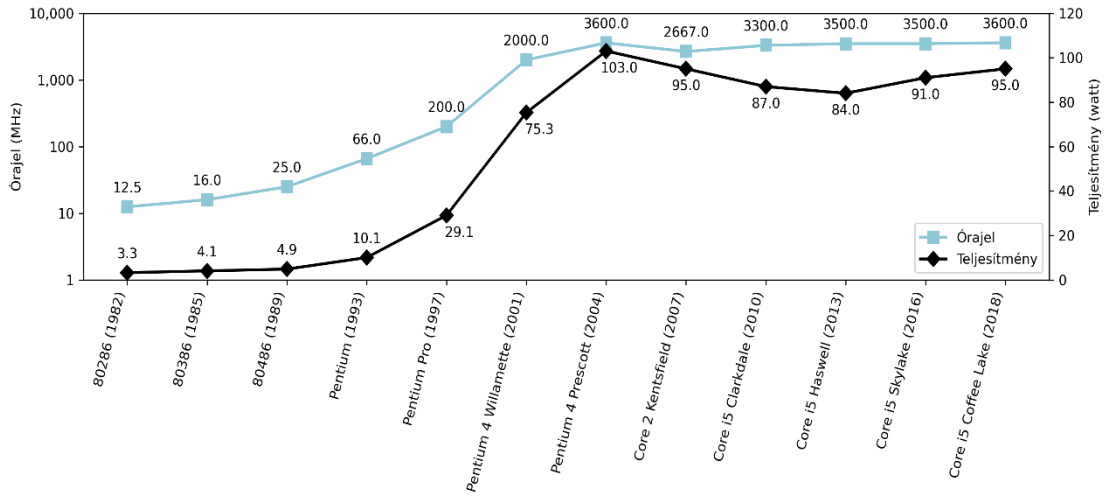
4. **Példa:** Két mikroprocesszort (A és B) szeretnénk összehasonlítani egy tesztprogrammal és a lenti táblázatban megadott adatok állnak a rendelkezésünkre. Ennek alapján döntsd el, hogy melyiknek van a nagyobb MIPS értéke.

	A	**B**
Utasítás szám	80 millió	60 millió
Órajel frekvencia	4 GHz	4 GHz
CPI	1.0	1.1

Megoldás: A MIPS érték kiszámításához a (6) képletet kell használni. Viszont ehhez meg kell határozni a CPU időt (4). A táblázatban megadott információk alapján a CPU idő 20 ms az A esetben és 16.5 ms B-nél. Ezt felhasználva az MIPS értékek kerekítve: 4000 és 3636.

8. Energiafal

Az előző fejezetben láttuk, hogy hogyan lehet kiszámítani a CPU időt, aminek az egyik tényezője az órajel frekvenciája. Ismét a (4) formulát felhasználva azt mondhatjuk, hogy minél nagyobb az órajel frekvenciája, annál kisebb lesz a CPU idő, vagyis annál gyorsabban tudja a processzor végrehajtani a program utasításait. Ebből kiindulva, ha a processzor tervező mérnökök képesek a processzor órajelét növelni, azzal a processzor teljesítményét is javítják. A múltban, ez a megközelítés elég jól működött, ahogy azt a lenti grafikon is szemlélteti. A grafikonon az Intel cég népszerű processzorainak az órajel frekvenciáját és az energia fogyasztását látjuk. Figyeld meg, hogy egy jól látható korrelációs kapcsolat volt az órajel frekvenciája és a processzor energia felhasználása között (mindkettő növekedett) a korai processzoroknál egészen az Pentium 4 Prescott-ig. Ezt követően az órajel frekvenciája nem lépte át a 3.6GHz-t és az energiafelhasználás is 90W körül mozgott.



6. ábra. Az Intel cég processzorainak órajel frekvenciája és energia fogyasztása.

Felmerülhet a kérdés, hogy a Pentium 4 Prescott után miért nem növelték tovább az órajel frekvenciáját, hogy ezzel biztosítsák az újabb processzorok teljesítmény növekedését? Két ok is van rá. Az egyik, hogy az energiafelhasználás csökkentése egyre nagyobb hangsúlyt kapott. A nagy szerverparkoknál az energiafelhasználás teszi ki az egyik legnagyobb költség hányadot. Ezen felül, a beágyazott rendszereknél is kritikus fontosságú az energiafelhasználás, mivel a legtöbb rendszer akkumulátorról üzemel és természetesen a

cél, hogy egy feltöltött akkumulátorral a rendszer a lehető legtöbb ideig képes legyen üzemelni.

A processzor energiafelhasználása az órajel növelésével lineárisan növekedik, ami a processzort alkotó tranzisztorok kapcsolási energiájának becslésére használt formulára vezethető vissza (7), ahol P a tranzisztor teljesítményét jelöli, míg a kapcsolási frekvencia a tranzisztor állapot váltásának gyakorisága, ami függ az órajel frekvenciájától.

$$P \propto \frac{kapacitiv\ terhelés \times feszültség^2 \times kapcsolási\ frekvencia}{2} \qquad (7)$$

A másik fontos gátja a processzor órajel további növelésének a **hőelvezetés**. A processzorban lévő tranzisztorok minél gyakrabban kapcsolnak, annál több energiát használnak és annál több hőt termelnek, amit el kell vezetni, különben a processzor tönkremegy. Erre statikus és dinamikus hűtési módszereket alkalmaznak. A statikus hűtés a processzora felhelyezett hűtőbordát foglalja magába, míg a dinamikus hűtés a hűtőbordákra helyezett ventilátort jelenti.

Még egy további érdekesség figyelhető meg a 6. ábrán. A 80286-os processzorhoz viszonyítottan, a Pentium Prescott-ig, az órajel frekvencia közel 300 szorosára nőtt, de az energia felszanálás „csak" megközelítőleg 30 szorosára. Az eltérő léptékű növekedésre a tranzisztorok feszültség értékének csökkenés ad magyarázatot. A tranzisztorok korai 5V-os működési feszültsége fokozatosan csökkent egészen 1V-ig. A (7) formula alapján a feszültség értéke négyzetesen befolyásolja az energiafelhasználást. Ez a fő oka annak, hogy az energiafelhasználás nem követte azt a növekedési pályát, amit az órajel frekvencia. Sajnos jelenleg az 1V-os feszültség érték egy fizikai gát, amit nem tudnak tovább csökkenteni, mivel ekkor a tranzisztor működése instabillá válna.

9. Többmagos processzorok

A közelmúlt tendenciái alapján a felhasználók hozzászoktak ahhoz, hogy az újabb számítógépes eszközök teljesítménye növekszik és a rajtuk futtatott programok válaszideje csökken. Ezt korábban, részben az órajel növelésével érték el. Azonban, az energiafal ennek a tendenciának gátat szabott, így a mérnököknek más irányba kellett elindulni a processzorok hatékonyságának további növelése érdekében. Ennek eredményeként 2006 körül jelentek meg a többmagos processzorok. Ez egyszerűen annyit jelent, hogy a processzorban (maga az integrált áramkör) több processzor magot helyeztek el. Ahogy korábban már szó esett róla, ez azért vált lehetővé, mert az integrált áramkörökben kialakított tranzisztorok mérete fokozatosan csökkent, így egyre több tranzisztort lehetett egy adott méretű IC-ben kialakítani (Moore törvény).

A többmagos processzorok megjelenését követően a processzor szó túlterhelté vált, mert ez jelentheti magát az IC-t is és a benne lévő magokat is. Ebben a jegyzetben én az IC-re mikroprocesszorként, vagy csak processzorként hivatkozok.

A többmagos mikroprocesszorok megjelenése nagy változást hozott nem csak a tervezőmérnökök, de a szoftverfejlesztők életében is. Ugyanis a több processzor mag megjelenésével a processzor párhuzamosan több feladaton is tudott dolgozni, így az **áteresztőképessége** drasztikusan növekedett. Viszont egy adott tesztprogram **végrehajtási ideje (válaszidő)** nem feltétlenül javult egy egymagos processzorhoz képest. Annak érdekében, hogy a programok ki tudják használni a rendelkezésre álló processzor magokat, a programot párhuzamosítani kell. Ahhoz, hogy ezt megtegyük tisztában kell lenni a párhuzamos programozási technikákkal. Ezen felül, ideális esetben, egységesen kellene megosztani a feladatokat a processzor magok között ahhoz, hogy a párhuzamosítás számottevően csökkentse a program végrehajtási idejét. Erre egy jó analógia, Patterson és Hennessy könyvéből [1] a 8 riporter példája, akik egy közös cikken dolgoznak. Azt gondolnánk, hogy mivel nyolcan írják a cikket, ezért nyolcszor gyorsabban fog elkészül, mintha csak az egyikük dolgozna rajta. Ehhez arra van szükség, hogy a feladatot nyolc egységes részre osszák szét ahhoz, hogy mindenki párhuzamosan tudjon haladni a saját feladatával. De mi van, ha az egyik lassabb, mint a többiek? Ha csak egy is nem készül el időben, akkor a cikk megjelenése késni fog. Egy másik lehetséges probléma,

ha a riporterek túl sokat kommunikálnak egymással, mivel ez is lassítja a feladat befejezését.

Tegyük fel, hogy a párhuzamos programozás specialistája vagy. Sajnos még ebben az esetben sem biztos, hogy előnyödre tudod fordítani a mikroprocesszorban helyet foglaló több processzor magot, mivel lehet, hogy a programkódod túlnyomó többsége nem párhuzamosítható. Ha számszerűsítve szeretnénk vizsgálni azt, hogy egy adott tesztszoftver esetében a processzor magok számának növelése milyen sebességjavulást eredményez, akkor az **Amdahl törvényt** kell alapul venni. Viszont ebben a jegyzetben, ezzel nem fogunk foglalkozni.

10. ARMv7 programozási modell

Programozóként azt reméljük, hogy a szoftverfejlesztést valamilyen magas szintű programozási nyelvvel végezhetjük. De, ha a szoftver futásának a sebessége is fontos tényező, akkor ismernünk kell a szoftvert futtató architektúrát is. Az **architektúra** a processzor utasításkészletét és az operandusok elhelyezkedését (konstans, regiszter, memória) határozza meg. Sokféle architektúra létezik, például: ARM, x86, MIPS, RISC-V. Egy ide kapcsolódó másik fontos fogalom a **mikroarchitektúra**, ami a hardveres megvalósítását jelenti egy adott architektúrának. Vegyük például az x86 architektúrát, amit két nagy cég is gyárt, az Intel és az AMD. Habár a két cég processzorainak architektúrája megegyezik, de a mikroarchitektúra eltérő.

Ebben a fejezetben az ARMv7 architektúrát fogom használni annak szemléltetésére, hogy a C-ben megírt programok hogyan lesznek lefordítva Assembly utasításokra. Az Assembly utasítások felfoghatóak úgy, mint az architektúra utasításkészletének programozóbarát leírása, ahol rövid, könnyen megjegyezhető kulcsszavakkal hivatkoznak az elemi utasításokra. Habár ez alacsony szintű programozást tesz lehetővé, de még így is egy magasabb szintű absztrakciót biztosít, mint a gépi kódokkal történő programozás. Az architektúrától függetlenül, az assembly nyelvek általában soronként egy utasítást tartalmaznak. Ezen felül, a memóriahelyekre hivatkozó címkék az első oszlopban kezdődnek, míg az utasítások a második oszlopban vagy azt követően kell kezdődniük. A megjegyzések egy kijelölt megjegyzéskaraktertől (ARM esetén ;) a sor végéig tartanak.

Az ARM architektúrát az 1980-as években fejlesztette ki az Advanced RISC Machines cég, amit ma ARM Holdings néven ismerünk. Az elmúlt években ARM processzorokból adtak el a legtöbbet. Például 2023-ban több, mint 28 milliárd ARM processzorral ellátott chipet értékesítettek. Szinte minden mobiltelefonban és táblagépben ARM processzormagok találhatóak. Egy durva becslés szerint az emberek több mint 75%-a használ ARM processzorral rendelkező termékeket a mindennapokban. De természetesen nem csak az okos eszközökben vannak ARM processzorok, hanem a beágyazott rendszerek vezérlőegységeiben is és így megtalálhatók a kamerákban, robotokban, autókban, játékgépekben stb.

Függetlenül attól, hogy milyen architektúráról van szó, a tervezőknek közös céljuk van: olyan architektúrát találni, amely megkönnyíti a hardver és a fordítóprogram felépítését, miközben maximalizálja a teljesítményt és minimalizálja a költségeket, valamint az energiafogyasztást. A világhírű John L. Hennessy és David A. Patterson a [1] könyvükben négy processzor tervezési alapelvet vezetett be:

1. A szabályszerűség egyszerűsíti a tervezést
2. A gyakori feladatok legyenek gyorsak
3. A kisebb gyorsabb
4. A jó terv kompromisszumokat igényel

Most nézzük meg, hogy ezek a tervezési alapelvek mit jelentenek a gyakorlatban. Az első megértéséhez kezdjük a lenti egyszerű mintakóddal. A jegyzetben a mintakódok minden esetben úgy lesznek megadva, hogy a bal oldalon lesz egy C-ben megírt kódrészlet, míg a jobb oldalon, ennek a kódrészletnek az Assembly szintű megvalósítása az ARMv7 architektúrán.

```
a = b + c;                    ADD a, b, c
```

A fenti C kódban csak két operandusunk van (b, c), ezért ezt egyetlen `ADD` utasítással meg tudjuk valósítani Assembly utasítások szintjén. Az `ADD` utasításban az első paraméter a célváltozó, míg a további kettő a két operandus. Nézzünk egy másik példát:

```
a = b - c;                    SUB a, b, c
```

Itt ismét két operandusunk van, amelyek között kivonást végzünk. Ezt a műveletet is egyetlen elemi utasítással el tudjuk végezni. Kérdezhetnénk, hogy hogyan jelenik meg ezekben a példákban a szabályszerűség? Hát úgy, hogy minden egyes aritmetikai utasítás csak egyetlen műveletet végez el és az utasítás formátuma konzisztens. Az utasítás „kulcsszóval” kezdődik, majd ezt követi a cél regiszter (ahová az eredményt tároljuk) és végül az operandusok következnek. Ez a kötött formátum megkönnyíti az utasítás kódolását és a hardveres megvalósítást. Az egyszerűség kedvéért most az Assembly utasításokban is ugyan azok a változónevek szerepelnek, mint a C kódban. A későbbi mintakódokban ezek helyét regiszterek veszik át.

Ha több operandusunk is van, akkor azt több elemi utasítás felhasználásával lehet alacsony szinten megvalósítani hasonlóan, mint azt a lenti példakód is mutatja.

```
a = b + c + d;                    ADD t, b, c
                                  ADD a, t, d
```

Ebben a példában három operandussal végzünk összeadást, amit két elemi utasítással lehet megvalósítani. Ez az egyszerű mintakód jó szemléltetés a 2. processzortervezési alapelvre, ami azt mondja, hogy a gyakori feladatok legyenek gyorsak. A RISC architektúrájú processzorok (beleértve az ARM-ot) utasításkészlete csak egyszerű, gyakran használt utasításokat tartalmaz. Ennek köszönhetően az utasítások dekódolására és végrehajtására szolgáló hardver egyszerű és gyors. Ez egyben azt is eredményezi, hogy az összetettebb utasítások (amelyek ritkábban fordulnak elő) több egyszerű utasítással valósíthatóak meg.

Az architektúrával kapcsolatban egy fontos kérdés, hogy hol tárolódnak az operandusok? Erre három lehetőség van: a memóriában, regiszterben vagy ha konstansról van szó, akkor az utasítás gépi kódjában. Az ARMv7 kevés belső regiszterrel rendelkezik, mindössze 16-tal. Mivel egy 32-bites architektúráról van szó, így a regiszterek mérete 32-bit. Tehát a processzor és a fizikai memória 32-bites blokkokban (**szavakban**) tud kommunikálni egymással.

Egy processzor a belső regisztereit éri el a leggyorsabban, sokkal kevesebb idő alatt, mint a fizikai memóriát. De akkor felvetődhet a kérdés, hogy miért csak 16 van belőle. Erre a 3. processzor tervezési alapelv szolgál magyarázatul, ami azt mondja, hogy a kisebb gyorsabb. Mivel kevés regiszter van, ezért a címdekódoló áramkör egyszerű és így a regiszterek címzése ideje rövid. Ezen felül, azzal is számolni kell, hogy ha több regiszter van, akkor több bitre van szükség a regiszterek beazonosításához az utasítások gépi kódjában. Erről később lesz szó részletesebben. A 2. táblázat egy összefoglalást ad a regiszterek felhasználási lehetőségeiről. A regiszterekre az R karakterrel és az azt követő index-el hivatkozunk. Az indexelés 0-tól kezdődik.

A programozási modell megszabja, hogy mely regiszterekben lehet átadni paramétert függvényhíváskor (R0-R3) és melyik regiszterben kell lenni a visszatérési értéknek (R0). Az is szabályozva van, hogy mely regiszterek tartalmát kell elmenteni (R4-R11), ha azt egy függvényben is használni akarjuk adattárolásra. Végül az utolsó három regiszter speciális célú és nem használjuk őket változók tárolására. Ezek közül az utolsó (R15) a jól ismert

programszámláló (**PC**), ami a következő, végrehajtandó utasítás memóriacímét tartalmazza.

Miután megismertük a regisztereket, célszerű pontosítani a korábban használt mintakódokat, ahol az elemi utasításokban is változóneveket használtunk. A változók regiszterekben lesznek tárolva, így az egyik korábbi mintakódunk helyes alakja lent, látható, ahol a változókat tároló regiszterek jelentek meg az utasításban. A regiszterekben tárolt értékekről, a pontosvesszővel kezdődő komment sor ad információt.

```
a = b + c;                    ;R1=a, R0=b, R2=c

                              ADD R1, R0, R2
```

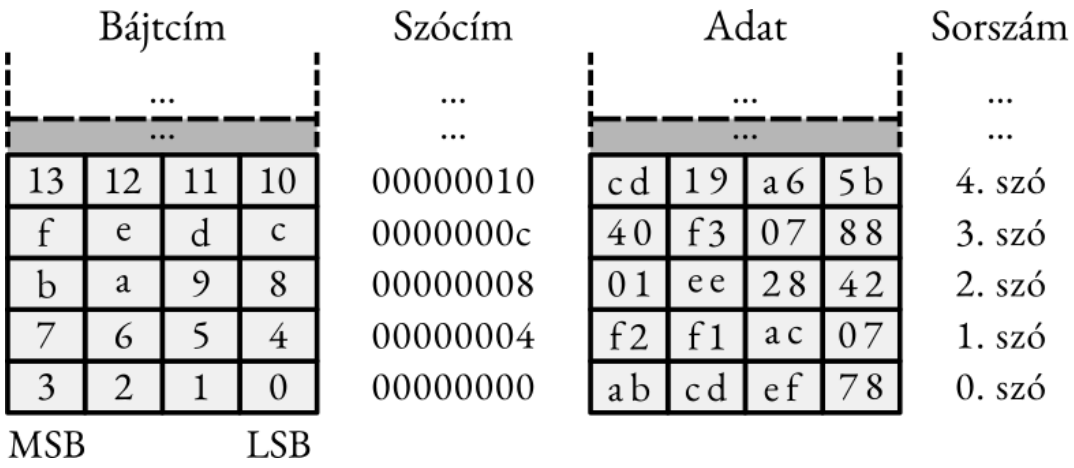
Regiszter	**Felhasználás**
R0	Argumentum, ideiglenes változó, visszatérési érték
R1-R3	Argumentumok, ideiglenes változók
R4-R11	Tárolt változók
R12	Ideiglenes változó
R13	Verem mutató
R14	Link regiszter
R15	Program számláló (PC)

2. táblázat. Regiszterek és azok funkciója.

Az utasítások második operandusa azonban nem csak regiszter lehet, hanem egy konstans érték is. Az elemi utasításokban a konstansokat # jelöli és ezt követi a konstans értéke, amit több számrendszerben is megadhatunk. Példaként vegyük a lenti kódrészletet, ahol a második operandus konstans és az értéke 16-os számrendszerben van megadva. A konstansnak az a sajátossága, hogy magában a gépi kódban van elhelyezve, ezért nem kell a memóriából áttölteni valamelyik regiszterbe, hanem azonnal elérhető.

```
a = b + 8;                    ADD R0, R1, #0x10
```

Egy programokban rendszerint sokkal több változó van, mint a rendelkezésre álló regiszterek száma. Az ARM egy **betölt-visszatölt** alapú architektúra, ami annyit jelent, hogy az operandusokat először be kell tölteni a processzor regisztereibe, majd az eredményt vissza kell tölteni a memóriába. Tehát a változókat (operandusokat) a fizikai memóriában (az egyszerűség kedvéért csak memóriaként hivatkozok rá a későbbiekben) kell tárolni és mielőtt a processzor valamilyen műveletet hajtana végre vele, először azt be kell olvasni valamelyik regiszterbe. A memória nagy kapacitású, de lassú. Könnyű elképzelni, hogy ha sokszor kell a memóriából adatot olvasni vagy visszaírni, az számottevően lassítja a program futását. Az optimális az lenne, ha a gyakran használt változók tovább maradnának a regiszterben ezzel csökkentve a memóriaolvasó utasításokat. Ennek a feladatnak a megoldását tipikusan a fordítóprogramra bízzuk.
Az ARM bájtonként címezhető memóriát hasznát (7. ábra). Mivel egy 32-bites architektúráról van szó, így itt az adatszóhossz is 32-bit, azaz 4 bájt.



7. ábra. A fizikai memória felépítésének szemléltetése.

Mivel a memória bájtonként címezhető, ezért arra is van lehetőségünk, hogy csak egy adott bájtját olvassuk vagy írjuk felül egy adatszónak. Az architektúra ehhez dedikált utasításokat is biztosít. Mivel minden bájtnak saját memóriacíme van és az adatszavak 4 bájt szélesek, ebből az is következik, hogy a szomszédos adatszavak közötti memória címtávolság is 4. Vegyük példaként a 2. adatszót (az indexelés 0-tól kezdődik), aminek a címe 0x08 vagy a 8. adatszórt, aminek a címe 0x20.
Az általunk vizsgált architektúrában, a memória adatszavainak olvasására az `LDR` (load register) utasítás áll a rendelkezésünkre. Ennek szemléltetésére vegyük a lenti példát, ahol a memória 0x08-as címéről olvasunk be egy adatszót az R3-as regiszterbe:

```
MOV R2, #0
LDR R3, [R2, #0x08]
```

Az első utasítás egy MOV, ami adatot mozgat egy regiszterbe. Jelen példában a konstans 0-át írjuk bele az R2-es regiszterbe. Ezt követi az LDR utasítás, aminek három argumentuma van. Az első az a regiszter, amibe a memória adatszavát fogjuk letárolni. Az azt követő további két argumentum a memória címének meghatározására szolgál. Ezek közül az első (R2) egy úgynevezett bázis regiszter, a második (0x08) pedig egy eltolás érték. Ezek alapján a tényleges memóriacímet az összegük adja (R2 + 0x08). Bármelyik regiszter használható bázisregiszterként, míg az eltolás értéke lehet konstans vagy egy másik regiszter tartalma. A fenti példában egy „alacsony" memóriacímről olvastunk be információt az R3-as regiszterbe, amit egy konstans értékkel is meg lehetett adni. Ezért nem volt szükség a bázisregiszter tartalmának felhasználására és ez az oka, hogy nulláztuk a bázisregisztert. Itt is igaz, hogy a konstans érték az LDR utasítás gépi kódjában tárolódik, de ez limitet is szab a konstans nagyságára. Ha egy nagy értékű memóriacímről akarunk információt beolvasni, akkor az a cím már nem fér bele egy konstansba és ekkor már a bázis regiszter tartalmára is támaszkodni kell.
Az LDR-hez hasonlóan működik az STR (store register) utasítás is, ami egy regiszter tartalmát tölti be az utasításban megadott memóriacímre. Nézzünk erre is egy példát:

```
MOV R2, #0
STR R1, [R2, #0x34]
```

A példában az STR utasítás az R1-es regiszter tartalmát tölti be a memória 12. adatszavába, aminek a memóriacíme 52 (0x34). Figyeld meg, hogy a címszámítás mechanizmusa itt is ugyan úgy működik, mint az imént látott LDR utasításnál. Az ARM memória író és olvasó utasításai nem hivatkoznak közvetlenül a főmemória címeire, mivel egy 32 bites cím nem fér bele az utasítás 32-bites gépi kódjába. Ehelyett **indirekt címzést** használnak, amely során egy regiszterben tárolt értéket (esetleges eltolással) használnak memóriacímként.
Az imént láttuk a két legfontosabb adatmozgató utasítást. Ezeken kívül számos egyéb utasítást áll a programozó rendelkezésére. Az utasítás funkcionalitása alapján három kategóriába lehet sorolni őket:

1. Adatfeldolgozó
2. Adatmozgató (vagy memória)

3. Elágaztató

Ezek közül az adatfeldolgozó utasítások csoportjába tartozik a legtöbb utasítás beleértve az aritmetikai utasításokat, a logikai utasításokat és a shift utasításokat (ami magába foglalja a forgatást is). A 3. táblázatba összegyűjtöttem a fontosabb adatfeldolgozó utasításokat, amiket érdemes megjegyezni, mert később szükség lesz rájuk. Ezeknek az utasításoknak az első argumentuma kötelezően regiszter, míg a második (ha van) lehet regiszter és konstans is.

A logikai utasítások közül az `AND` és `ORR` a jól ismert logikai ÉS illetve VAGY műveleteket végzi el az operandusok bitjei között és ez kerül a célregiszterbe (első argumentum). Az `EOR` utasítás XOR műveletet hajt végre, amit már szintén tanultál digitális technikán. Az `MVN` (move and not) is egy jól ismert műveletet fog végrehajtani és ez az invertálás. Végül a `BIC` utasítás maradt a logikai utasítások kategóriájából, ami a háttérben logikai és műveletet hajt végre, annyi eltéréssel az `AND` utasításhoz képest, hogy a második operandust invertálja, mielőtt végrehajtja a logikai ÉS műveletet az operandusok között.

Kategória	**Utasítás**
Logikai utasítások	`AND Ra, Rb, c`
	`ORR Ra, Rb, c`
	`EOR Ra, Rb, c`
	`BIC Ra, Rb, c`
	`MVN Ra, Rb`
Shift utasítások	`LSL Ra, Rb, c`
	`LSR Ra, Rb, c`
	`ASR Ra, Rb, c`
	`ROR Ra, Rb, c`
Aritmetikai utasítások	`ADD Ra, Rb, c`
	`SUB Ra, Rb, c`
	`MUL Ra, Rb, c`

3. táblázat. Adatfeldolgozó utasítások fiktív argumentumokkal.

A shift utasításkategóriába 4 utasítás tartozik. Ezek közül az első kettő a logikai balra shiftelés (`LSR`) és a logikai jobbra shiftelés (`LSR`). A jobbra shiftelésnek elérhető az aritmetikai párja is, amit az osztás művelet helyettesítésére is használhatunk, amennyiben

az osztó kettő hatványa. Végül egy forgatás művelet is elérhető (ROR), ami egy regiszter bináris tartalmát jobbra forgatja az utasításban megadott bitmennyiséggel. Amennyiben a shift mennyisége konstanssal van megadva az utasításban, az legfeljebb 5-bites lehet. De ha belegondolunk, akkor nincs is szükség 5-biten túlnyúló értékre. Ha egy regisztert 32 bittel forgatsz el jobbra, akkor visszakapod az eredeti értéket, ha 33-mal forgatod jobbra, akkor az ekvivalens azzal, mintha csak 1-bittel forgattad volna balra. Mivel 5-biten a legnagyobb tárolható érték 31, így ezzel az összes lehetséges forgatási lehetőséget elő lehet állítani.

Az aritmetikai utasítások közül csak hármat említettem meg a 3. táblázatban. Az ADD és SUB utasításokról már korábban volt szó, így most csak a MUL utasítással foglalkozunk. A MUL szorzást végez két 32 bites operandussal és az eredményt 32-biten tárolja. Ennek csak az a szépséghibája, hogy két 32-bite szám szorzata akár 64-bites is lehet. Ennek alapján, 64 bitre van szükség az eredmény veszteség mentes (megbízható) letárolásához. Mivel a MUL utasítás csak 32 biten tárolja az eredményt, így csak azokban az esetekben lehet használni, amikor biztosak vagyunk abban, hogy a szorzás eredménye 32-bitbe belefér. Ha a szorzás eredménye nem fér bele 32 bitbe, akkor az UMUL (unsigned) vagy az SMUL (signed) utasításokat lehet használni. Mindkét utasítás 64 biten tárolja az eredményt, ami azzal is jár, hogy nem három, hanem négy argumentuma lesz az utasításnak (UMULL R1, R2, R3, R4). Ezek közül az első két regiszterbe fog az eredmény tárolódni, míg az utolsó kettő lesz a két operandus.

A programkódok nem mindig akarjuk szekvenciálisan (egymást követően sorról sorra) végrehajtani. Például a jól ismert programozási sémáknál, mint az elágaztatások vagy a ciklusok csak akkor fogunk végrehajtani egy adott kódblokkot, ha egy feltétel igaz. Az általunk vizsgált ARM architektúra feltételes kapcsolókat használ annak eldöntésére, hogy egy utasítást végre kell hajtani vagy sem. A feltételes kapcsolók összefoglalása a 4. táblázatban tekinthető meg.

Ezek a feltételes kapcsolók a processzor **aritmetikai-logikai egysége (ALU)** által generált **jelzőbitek (N, Z C, V)** alapján értékelődnek ki. A negatív (N) bit akkor vesz fel 1-es értéket, ha az eredmény negatív, kettes komplemens ábrázolást használva. A nulla (Z) bit akkor lesz 1-es, ha az eredmény minden bitje nulla. Az átvitel (C) bit akkor lesz beállítva (1), ha a művelet során carry érték generálódik. Végül, a túlcsordulás (V) bit akkor lesz beállítva, ha egy aritmetikai művelet túlcsordulást eredményez. Ez tipikusan akkor fordul

elő, ha azonos előjelű operandusokkal végzett műveletet követően az eredmény előjele az operandusokétól eltérő.

Kód	Kapcsoló	Megnevezés	Feltétel
0000	EQ	Egyenlő	Z
0001	NE	Nem egyenlő	$\bar{Z}$
0010	CS / HS	Carry 1 / Előjel nélküli nagyobb	C
0011	CC / LO	Carry 0 / Előjel nélküli kisebb	$\bar{C}$
0100	MI	Negatív	N
0101	PL	Pozitív	$\bar{N}$
0110	VS	Túlcsordulás	V
0111	VC	Nincs túlcsordulás	$\bar{V}$
1000	HI	Előjel nélküli nagyobb mint	$\bar{Z}C$
1001	LS	Előjel nélküli kisebb egyenlő	$Z\ OR\ \bar{C}$
1010	GE	Előjeles nagyobb egyenlő	$\overline{N \oplus V}$
1011	LT	Előjeles kissebb mint	$N \oplus V$
1100	GT	Előjeles nagyobb mint	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Előjeles kisebb egyenlő	$Z\ OR\ (N \oplus V)$
1110	AL	Nincs feltétel	-

4. táblázat. Feltételes kapcsolók.

A korábban bemutatott 16 regiszteren túl, a programozási modell másik fontos alapregisztere az **aktuális programállapot-regiszter (CPSR)**. A CPSR legfelső négy bitjében van letárolva az imént említett ALU jelzőbitek aktuális állapota. Ezeket a biteket kétféleképpen lehet frissíteni. Az egyik lehetőség, hogy az adatfeldolgozó utasításokat kibővítjük az S kapcsolóval, ami „kényszeríti” az ALU-t a jelzőbitek frissítésére. Például az `ADD R1, R2, R3` utasítás elvégzi az összeadást (R1 = R2 + R3), de a jelzőbiteket nem frissíti az összeadás eredménye alapján. Viszont, ha az `ADD` utasítást kibővítjük az `S`

kapcsolóval (`ADDS R1, R2, R3`), akkor ez az utasítás már a jelzőbiteket is frissíti azon felül, hogy ugyan úgy elvégzi az összeadást. Minden adatfeldolgozó utasítás frissíti az N és Z jelzőbiteket az eredménytől függően. A shift műveletek az N és Z mellett a C jelzőbitet is, míg az aritmetika utasítások a V bitet is frissítik.

Az ALU jelzőbitjeinek frissítésére egy másik lehetőség a `CMP` (compare) utasítás használata, aminek két operandusa van és nincs célregisztere. Ahogy a neve is jelzi, az utasítás két értéket hasonlít össze, ami gyakorlatilag egy kivonást jelent és ennek eredménye alapján állítódnak be a jelzőbitek. Hogy teljes legyen a kép a feltételes kapcsolók működéséről, vegyünk egy példát, ahol tudjuk, hogy az R3-ban tárolt érték 18, míg az R4-ben tárolt érték 20:

```
CMP   R3, R4
SUBEQ R1, R2, R0
ORRMI R2, R5, R6
```

A fenti példában, mivel tudjuk R3 és R4 tartalmát, így meg tudjuk határozni, hogy mi lesz a `CMP` utasításnál kapott eredmény (18 – 20 = -2) és ennek alapján a jelzőbitek értéke (N=1, Z=0, C=0, O=0). Ha tudjuk a jelzőbitek értékeit, azt is el tudjuk dönteni, hogy a két feltételes utasítás végrehajtódig vagy sem. A `SUBEQ` utasítás kapcsolója az `EQ`, ami akkor teljesül, ha Z = 1. Ebben a példában ez nem teljesül, ezért ez az utasítás nem lesz végrehajtva. A következő utasításnál a feltételes kapcsoló az `MI`, ami N = 1 esetén teljesül, ezért a példában szereplő `ORRMI` végre lesz hajtva.

Most már ismered az alapvető adatmozgató és adatfeldolgozó utasításokat. Az utasítások harmadik csoportja az elágaztató utasításoké, ahol mindösszesen csak két utasítást kell megismerned. Ahogy a csoport elnevezése is jelzi, ezek az utasítások lehetővé teszik a sorrenden kívüli utasításvégrehajtást. Ez egyszerűen azt jelenti, hogy a programkód adott pontjáról, a vezérlés át tud ugrani egy másik pontra és onnan folytatódik az utasításvégrehajtás. A két utasítás, amivel foglalkozni fogunk az a `B` és a `BL`. A `B` egy elágaztató vagy más szóval ugró utasítás, ami lehet feltételes, amennyiben feltételes kapcsolót társítunk hozzá. A `BL` is egy elágaztató utasítás, amit majd függvényhívásra fogunk használni. Először a `B` utasítással foglalkozunk, ami működési elvben megegyezik a `goto` utasításával a C programozási nyelvből. Nézzünk egy ide vágó példát:

```
MOV R0, #12
```

```
B    TARGET
AND R1, R2, #3

TARGET
SUB R1, R2, R3
```

A fenti példában a `B` utasítás argumentuma egy **címke**, ami valójában egy memóriacímet jelent. Azt a memóriacímet, ahová át kell helyezni az utasítás végrehajtást. A példában egy feltétel nélküli ugrás történik a `B TARGET` utasítás végrehajtása után és ezt követően a következő utasítás, a `SUB R1, R2, R3` lesz. Tehát az ugró utasítást követő `AND` utasítás soha nem lesz végrehajtva. Nézzük egy másik példát is, ahol az ugrás feltételhez kötött:

```
MOV R0, #4
ADD R1, R0, R0
CMP R0, R1
BEQ TARGET
ORR R1, R1, #1

TARGET
ADD R1, R1, 78
```

Ebben a példában az `EQ` kapcsoló miatt, az ugrás akkor fog végrehajtódni, ha az R0 és az R1-es regiszterekben ugyan az az érték van. Ekkor ugyanis a `CMD` által végzett kivonás eredménye 0, így Z=1.

Felmerülhet a kérdés, hogy mire lehet a gyakorlatban használni a feltételes ugrásokat? Valójában ezek szolgáltatják az alapját, a magas szintű programozási nyelvek elágaztatásainak, ahol el kell dönteni, hogy végrehajtódik egy adott kódblokk vagy sem. A most következő példák szemléltetni fogják, hogy hogyan lehet Assembly utasítások szintjén megvalósítani a magasszintű programozási nyelvekben megszokott programozási szerkezeteket. Kezdjük egy egyszerű `if` feltétellel, ahol nincs `else` ág:

```
if (i == j)                ;R0=f, R1=g, R2=h, R3=i, R4=j
    f = g + h;                  CMP R3, R4
f = f - i;                      BNE L1
```

```
                ADD R0, R1, R2
            L1
                SUB R0, R0, R2
```

A C kódban, a feltétel azt vizsgálja, hogy az *i* változó értéke megegyezik-e a *j* változó értékével. Ha megegyezik, akkor a feltétel igaz és az `if` blokkhoz tartozó programkód lefut. Ezzel szemben az Assembly utasításoknál azt vizsgáljuk, hogy az *i* értéke eltér-e a *j* értékétől! Figyeld meg, hogy a jobb oldali kódnál először a `CMP` utasítás kivonja *j* értékét *i*-ből (R3 és R4 regiszterek), majd ezt követi egy ugró utasítás, amelynél az `NE` feltételes kapcsolót használjuk. Ez a feltétel akkor teljesül, ha az előző kivonásnál a két operandus nem egyenlő, mert így az eredmény nem nulla, tehát a Z=0. Összegezve azt mondhatjuk, hogy a C kódnál azt vizsgáljuk, hogy mikor kell végrehajtani a kódblokkot, míg az Assembly programkódban, ennek a fordítottját. Azt, hogy a kódblokkot mikor kell átugrani. Az előző C programkódot másként is át lehet fordítani elemi utasításokra. Erre mutat példát a következő kódrészlet:

```
if (i == j)                ;R0=f, R1=g, R2=h, R3=i, R4=j
    f = g + h;                  CMP   R3, R4
f = f - i;                      ADDEQ R0, R1, R2
                                SUB   R0, R0, R2
```

A fenti példában kevesebb utasításra volt szükség, mint az előző esetben, ezen felül, nincs benne ugró utasítás, ami tipikusan több órajelciklust igényel a végrehajtáshoz, mint más adatmozgató utasítások. Itt azt használtuk ki, hogy az `if` blokkja csak egyetlen utasítást foglal magába, így elég csak ennek az egy utasításnak a végrehajtását feltételhez kötni, amit könnyen meg tudunk tenni a megfelelő feltételes kapcsoló használatával. Ez a két példakód arra is rávilágít, hogy ugyan azt a magas szintem megírt programkódok több alternatív módon is meg lehet valósítani elemi utasításokkal. Ezt alapul véve, a fordítóprogramokat is lehet konfigurálni, hogy végrehajtási sebességre legyen optimalizálva a lefordított program. Most bővítsük ki az előző C programkódot egy `else` ággal:

```
if (i == j)                    CMP R3, R4
   f = g + h;                  BNE L1
else                           ADD R0, R1, R2
   f = f - i;                  B   L2
                              L1
                               SUB R0, R0, R2
                              L2
```

Az átfordított alakban a feltétel vizsgálata ugyan úgy történik, mint a korábbi példakódnál. Viszont, most egy feltétel nélküli ugrás zárja az `if` blokkot (`B L2`), ami arra kell, hogy amennyiben az `if` feltétele teljesül, akkor az `else` ág kódblokkja már ne legyen végrehajtva és azt ugorjuk át. Ha viszont a feltétel nem teljesül, akkor a feltételes ugrásnak az `else` blokk elejére kell ugrani és azt kell végrehajtani.

Most már láttál példákat arra vonatkozóan, hogy hogyan lehet az elágaztatásokat elemi utasításokkal megvalósítani. Ez után jöjjenek a ciklusok. Először kezdjük egy `while` ciklussal:

```
int pow = 1;                    MOV R0, #1
int x   = 0;                    MOV R1, #0
while (pow != 128){            WHILE
   pow = pow * 2;               CMP R0, #128
   x = x + 1;                   BEQ DONE
}                               LSL R0, R0, #1
                                ADD R1, R1, #1
                                B   WHILE
                               DONE
```

A fenti példakódban két változónk van, amit az R0 és R1 regiszterekben tárolunk. Figyeld meg itt is, hogy míg a C kódban a `while` ciklus addig fog lefutni, amíg a pow változó nem

lesz egyenlő 128-cal, addig az elemi utasítások szintjén azt vizsgáljunk, hogy ez mikor lesz egyenlő 128-cal (ismét az `EQ` kapcsolót használjuk), mert ekkor kell befejezni a ciklust a feltételes ugrással. Különben maradunk a ciklusban és lefut a ciklus magja. Figyeld meg, hogy a kettővel való szorzást egy logikai balra shifteléssel váltottunk ki, majd növeltük *x* értékét eggyel. Végül a feltétel nélküli ugrás zárja a `while` blokkot, ami visszaugrik a feltételvizsgálathoz és az egész folyamat kezdődik az elejétől.

A másik ciklusszervezési lehetőség a `for` ciklus használata. Ez szintakszisban eltér a `while` ciklushoz képest, mivel itt a `for` fejrészében nem csak egy feltételt adunk meg, hanem bevezetünk egy ciklusváltozót is és azt is megadhatjuk, hogy a ciklusváltozó hogyan legyen módosítva ciklusonként. Ennek ellenére az elemi szintű megvalósítása a `for` ciklusnak logikailag meg fog egyezni a `while` ciklusnál látottal. Vizsgáljuk meg ezt is egy példán keresztül:

```
int i;                              ;R0 = i, R1 = sum
int sum = 0;                         MOV R0, #1
                                     MOV R1, #0
for (i=1; i!=10; i=i+1)             FOR
   sum = sum + i;                    CMP R0, #10
                                     BEQ DONE
                                     ADD R1, R1, R0
                                     ADD R0, R0, #1
                                     B   FOR
                                    DONE
```

Itt ismét két változónk van, amit mindkét kódrészletben az elején inicializálunk. A C kódban a ciklusváltozónk az *i*, amit ciklusonként eggyel növelünk. A ciklus addig teljesül, amíg az *i* nem egyenlő 10-zel. Hasonlóan, mint azt már korábban is láttad, ismét ennek a fordítottját vizsgáljuk az elemi utasítások szintjén, azaz, ha az *i* értéke (R0) egyenlő 10-zel, akkor lesz a ciklus megtörve a `BEQ` utasításnak köszönhetően. A ciklus blokkját itt is egy

feltétel nélküli ugrás zárja, hasonlóan, mint az előző példakódnál, ami visszaviszi a vezérlést a `for` kezdetére.

Az elágazást és a ciklusokat követően most nézzük meg, hogy a tömbök kezelése hogyan oldható meg elemi utasítások felhasználásával. A tömbökről tudjuk, hogy a C programozási nyelvben nagy mennyiségű, azonos típusú adatok tárolására szolgálnak. Az itt tárolt elemek az indexeik alapján érhetők el. Ahhoz, hogy az indexek alapján egyértelműen meg lehessen határozni egy tömbelem memóriacímét, tudnunk kell a tömb kezdőcímét is. A tömb címe megegyezik a 0. elemének a címével. Mivel az elemek adattípusa kötött, így a tömb címétől kezdődően ki tudjuk számolni minden egyes elemének a memóriacímét. Ennek szemléltetéséhez ismét nézzünk egy példát, ahol van egy öt elemű `int` tömbünk és valahol a programkódban a tömb 0. és 1. elemét megváltoztatjuk:

```
int array[5];                       ;R0 = array base address
…                                   MOV R0, #0x60000000
array[0] = array[0] * 8;            LDR R1, [R0]
array[1] = array[1] * 8;            LSL R1, R1, 3
                                    STR R1, [R0]
                                    LDR R1, [R0, #4]
                                    LSL R1, R1, 3
                                    STR R1, [R0, #4]
```

A jobboldali programkód elején definiálni kell, hogy hol lesz a tömb kezdőcíme a memóriában (R0). Ahhoz, hogy a tömb 0. elemét felülírjuk a korábbi érték nyolcszorosával, először be kell tölteni a korábbi értéket egy regiszterbe. Ne felejtsd el, hogy a 0. elem címe megegyezik a tömb címével, ezért az első `LDR` utasításban nincs szükség eltolásra. A 8-cal (2^3) való szorzást itt is logikai balra shifteléssel oldottuk meg. Az új érték az R1 regiszterben van, szóval ennek a tartalmát kell visszatölteni a tömb 0. elemének memóriacímére az `STR` utasítással. Nem meglepő, hogy ugyan azt a címet használtuk az `LDR` és az `STR` utasításoknál is, mivel ugyan oda kell visszaírni az új értéket, ahonnan a korábbit kiolvastuk. Ugyan ezt a folyamatot kell elvégezni az 1. tömbelem módosítása során is,

annyi eltéréssel, hogy most a memóriacím meghatározásánál eltolás értéket is alkalmazunk, ami 4. Ez az eltolásérték onnan ered, hogy az `int` típus 4 bájton tárolódik, tehát az egymást követő tömbelemek közötti címtávolság is 4 lesz.

Most nézzünk egy kicsit összetettebb mintakódot, ahol egy tömb elemeit egy `for` ciklusban frissítjük. A lenti kód egy 200 elemű tömb minden elemén végigmegy és 10-zel növeli. Az elemi utasítások szintjén, a programkód elején ismét meghatározásra kerül a tömb címe és a `for` ciklus felépítése is logikailag teljesen megegyezik azzal, amit korábban láttunk. A ciklus elejét egy címke jelzi, azt követi a feltétel vizsgálat és a ciklus végén egy feltétel nélküli ugrás visz vissza a ciklus kezdetéhez. A ciklus törzsében ismét ki kell olvasni a tömb elemeit egyesével, majd a 10-zel megnövelt értéket vissza kell tölteni. Ebben a mintakódban a legérdekesebb az, hogy hogyan számoljuk ki a tömbelemek címét. Ehhez az R2-es regisztert fogjuk felhasználni és ebben tároljuk le a címszámításhoz szükséges eltolás értéket. Az aktuális eltolás értéket az *i* * 4 szorzat adja, amit ismét logika shifteléssel valósítottunk meg.

```
int scores[200];
int i;

for (i=0; i < 200; i = i + 1)
  scores[i] = scores[i] + 10;
```

```
;R0 = array base address, R1 = i
   MOV R0, #0x14000000
   MOV R1, #0
LOOP
   CMP R1, #200
   BGE L3
   LSL R2, R1, #2
   LDR R3, [R0, R2]
   ADD R3, R3, #10
   STR R3, [R0, R2]
   ADD R1, R1, #1
   B LOOP
L3
```

10.1 Függvényhívás

Korábban említettem, hogy két ugró utasítást foguk használni ebben a jegyzetben. Ezek közül az egyik a `B` utasítás, amivel már foglalkoztunk, míg a másik a `BL`, ami függvényhívást tesz lehetővé. A `BL` utasításnak is egy argumentuma van, egy címke, ami a függvény kezdetét jelöli. Azonban függvényhíváskor egy egyszerű elágazás nem elegendő, mert nem tudnánk, hová térjünk vissza. A megfelelő visszatéréshez mentenünk kell a függvényhívást követő utasítás címét (PC+4) a függvény meghívásakor, és amikor az befejeződött, a mentett memóriacímet kell betölteni a PC-be. A mentés helye a **Link (LR) regiszter**, ami az egyik speciális célú regiszter a 16 közül. A mentett memóriacím PC-be töltése a `MOV PC, LR` utasítással könnyen megoldható. Ezért ezt az utasítást arra fogjuk használni, hogy visszatérjünk a függvényből a hívóhoz.

Függvényhíváskor van néhány további szabály, amit be kell tartani. Az egyik, a függvény lehetséges paramétereinek az elhelyezését köti meg. A paramétereknek az R0-R3 regiszterekben kell lenni. Továbbá, ha van visszatérési érték, akkor azt az R0 regiszterbe kell tárolni. Ha a függvényen belül felhasználjuk az R4-R11 regisztereket, akkor azok korábbi tartalmát meg kell őrizni és mielőtt a függvény visszatérne a hívóhoz, az eredeti értékeiket vissza kell tölteni. Ez igaz lesz az LR regiszterre is. Most nézzünk egy példát arra vonatkozóan, amikor a fenti megkötések nem teljesülnek:

```
int main() {
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);
    ...
}
int func(int f, int g, int h, int i){
    int result;
    result = (f + g) - (h + i);
    return result;
```

```
;R4 = y
MAIN
  ...
  MOV R0, #2
  MOV R1, #3
  MOV R2, #4
  MOV R3, #5
  BL FUNC
  MOV R4, R0
  ...
```

```
}                                   FUNC
                                        ADD R8, R0, R1
                                        ADD R9, R2, R3
                                        SUB R4, R8, R9
                                        MOV R0, R4
                                        MOV PC, LR
```

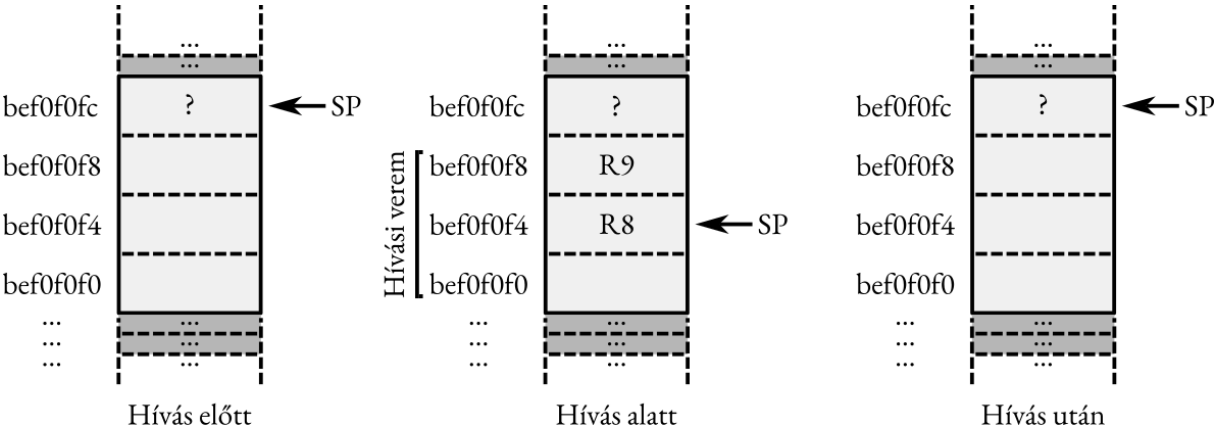
A fenti C programkódban van egy main függvényünk, amiből egy func nevű függvényt hívunk meg. A hívott függvény négy paramétert kap, amelyekkel egyszerű aritmetikai műveleteket végez. Az elemi utasítások szintjén, a függvényhívás előtt inicializáljuk a paramétereket. A paraméterek az R0, R1, R2, és R3 regiszterekbe kerültek bele. Ezzel a résszel nincs probléma, mert ezek a regiszterek dedikáltan használhatóak paraméter átadásra. A `BL FUNC` utasítás után a végrehajtás a FUNC címkét követő utasítással folytatódik. A függvényen belül (a FUNC címkét követő kódsorok) az R8, és R9 regisztereket használtuk fel az f+g és a h+i összegek ideiglenes tárolására. Majd ezt követően, az R4-be tároltuk le a végeredményt. Mivel van egy olyan megkötés, ami azt mondja, hogy a visszatérési értéknek az R0-ban kell lenni, ezért a MOV utasítással a végeredményt R4-ből áthelyeztük az R0-ban.

A fenti mintaprogrammal több probléma is van. Az egyik kézenfekvő az, hogy a SUB R4, R8, R9 utasításban az eredményt már azonnal lehetett volna az R0 regiszterbe is tárolni az R4 helyett. Ezzel megspórolunk egy felesleges utasítást és az R4-es regisztert sem kell használni. A másik probléma, hogy az R8 és R9 regisztereket felhasználtuk a függvényben, de nem őriztük meg a tartalmukat. A kérdés az, hogy hol lehet ezeknek a regisztereknek eltárolni az értékét? Ebben lesz segítségünkre a **verem**. A verem, a fizikai memóriában helyezkedi el, annak egy dedikált része. A verem egy **LIFO (last in first out)** adatszerkezet, ami annyit jelent, hogy a verembe utoljára letárolt adatszót tudjuk először kivenni. A verem legfelső elemének a memóriacímét az **SP (stack pointer)** regiszter tárolja. Az SP a 3. speciális célú regiszter a belső 16 regiszterből. Ha egy új adatszót akarunk tárolni a veremben, annak először helyet kell csinálni. Mivel az ARM architektúránál a verem mérete lefelé növekszik a fizikai memóriában, így a „helycsinálás” a gyakorlatban az jelenti, hogy az SP értékét csökkentjük. Ha már nincs tovább szükség egy veremben tárolt értékre, akkor annak a helyét fel kell szabadítani, nehogy kifussunk a

rendelkezésre álló memóriából. Most nézzük meg, hogy hogyan kellett vonal az előző mintakódban az R8 és R9-es regiszterek értékét tárolni:

```
FUNC
  SUB SP, SP, #8
  STR R9, [SP, #4]
  STR R8, [SP]
  ADD R8, R0, R1
  ADD R9, R2, R3
  SUB R0, R8, R9
  LDR R8, [SP]
  LDR R9, [SP, #4]
  ADD SP, SP, #8
  MOV PC, LR
```

Figyeld meg, hogy a fenti példában az első utasítás az SP értékét 8-cal csökkenti. Ezzel azt értük el, hogy az SP egy 8 bájttal lejjebb lévő címre mutat (8. ábra). Ez a 8 bájt két adatszóméretet ad ki, ahová az R8 és R9 regisztereket tároljuk el. A regiszterek tárolását az `STR` utasítással oldottuk meg. Ezt követi a visszatérési érték kiszámítása, amit most már azonnal az R0 regiszterbe tárolunk. Mielőtt visszatérnénk a hívóhoz a `MOV PC, LR` utasítással, vissza kell helyezni a veremből R8 és R9 korábbi értékeit. Ezt az `LDR` utasítással tettük meg, ahol a címbeállítás az SP tartalmához viszonyítva történt.



8. ábra. A verem tartalma a fenti mintakód futtatása előtt, közben és után.

A magasszintű programozási nyelvekben arra is van lehetőségünk, hogy egy függvényből egy másik függvényt hívjunk. Az előző példában egyetlen **levélfüggvényünk** volt, ami

egy olyan függvényt jelent, amely nem hív meg további függvényt. Most nézzünk egy olyan példát, ahol két függvényünk van és az egyik ciklikusan hívja a másikat:

```
int func1(int a, int b) {
     int i, x;
     x = (a + b)*(a – b);
     for (i=0; i<a; i++)
          x = x + f2(b+i);
     return x;
}
int func2(int p) {
     int r;
     r = p + 5;
     return r + p;
}
```

```
; R0=a, R1=b, R4=i, R5=x
FUNC1
  PUSH {R4, R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0}
  ADD   R0, R1, R4
  BL    FUNC2
  ADD   R5, R5, R0
  POP {R0}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP {R4, R5, LR}
  MOV   PC, LR

FUNC2
  PUSH {R4}
```

```
ADD   R4, R0, 5
ADD   R0, R4, R0
POP {R4}
MOV   PC, LR
```

Először vessünk egy pillantást a C kódra, ahol feltételezzük, hogy valahol van egy main függvény és onnan hívjuk meg a func1 függvényt. Ebben egy `for` cikluson belül kerül meghívásra a func2 függvény, aminek egy paramétere van. Ami igazán érdekes, az a jobb oldali kódrészlet, ahol két eddig még nem használt utasítás is megjelenik. Ezek a POP és a PUSH, amik más processzor architektúráknál is gyakran elérhetőek. Segítségükkel a regisztereket kompakt módon, a regiszterszámuk sorrendjében lehet menteni és visszaállítani. A func1 kódrészlet egy PUSH utasítással eltárolja a verembe az R4, R5 és LR regisztereket. Vedd észre, hogy most az LR regisztert azért kell tárolni, mert a func2 függvényhívás felülírja az LR tartalmát. Ha ezt nem mentjük el, akkor problémában lennénk, amikor a func1 függvényből kell visszatérni a főprogramba, mert nem ismernénk a visszatérési címet. A func2 meghívása előtt van még egy PUSH utasítás, ami az R0-t tölti a verembe. Erre azért van szükség, mert az R0-R3 regisztereket a függvényen belül szabadon lehet használni anélkül, hogy megtartanánk a korábbi értékét. Ez azt jelenti, hogy ha a hívó oldalon ezekben a regiszterekben hasznos információ van, akkor a hívó oldalon kell tárolni az értéküket és miután a függvény visszatért a hívóhoz, vissza kell tölteni a korábban letárolt értéket. A fenti példakódban a func2 R0-ban tárolja a visszatérési értéket, de R0 hasznos információt tartalmaz a func1 számára is, ezért már itt menteni kell, mielőtt a func2-t meghívódna. Miután a func2 függvény visszatér, egy POP utasítás írja vissza R0 korábbi értékét, ami a C kódban az *a* változónak felel meg. A func2-ben is van egy PUSH és egy POP utasítás, amivel az R4 regiszter korábbi értékét őrizzük meg, mivel az R4-et a func2 függvény ideiglenes tárolásra használta. Végül a func1 függvényben, a visszatérés előtt a POP utasítással visszaállítjuk a korábban tárolt R4, R5 és LR regisztereket és ezzel azt is biztosítjuk, hogy a megfelelő utasításhoz térjen vissza a func1 a főprogramban.

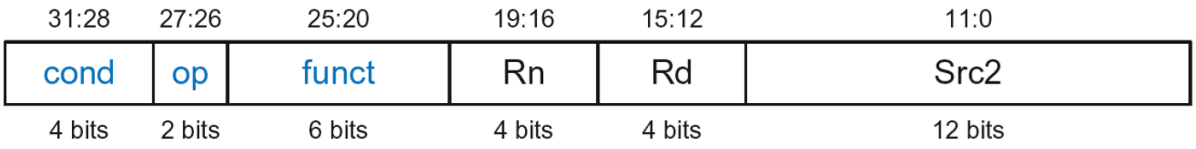
A gyakorlatban még egy olyan probléma is felmerülhet, hogy nem négy, hanem annál több argumentumot akarunk átadni egy függvénynek. Ez is megoldható. Ilyenkor az első négy

argumentumot, a már korábban ismertetett szabály alapján, az R0-R3 regiszterekbe kell helyezni. A további argumentumokat a verem tetejére. Ezt a függvény is eléri és innen ki tudja olvasni.

11. ARMv7 utasítás kódolás

Már tudod, hogy az Assembly utasításokat, az Assembler lefordítja gépi kódra, mivel a processzor csak a bináris formában lévő gépi-kódokat képes megérteni. Az utasítások kódolása azt jelenti, hogy a gépi-kódban hogyan írjuk le a processzor számára, hogy pontosan milyen műveletet kell elvégeznie. A korábban bemutatott processzortervezési alapelvek közül az első (a szabályszerűség egyszerűsíti a tervezést), az utasítások kódolásánál is megjeleni. Az ARMv7 egy 32-bites architektúra, ahol minden utasítás 32-biten van kódolva. Ez a 32 bit több részmezőre bomlik szét, ahol a mezők a többitől eltérő információ tartalommal rendelkeznek az elvégzendő utasításról. A 32-bites érték mezőkre történő felbontását, hasonló módon kell elképzelni, mint ahogyan egy csomag felépítését a számítógépes hálózatoknál. Természetesen, a legjobb itt is az lenne, ha minden utasításnál a 32-bit pontosan ugyan azokra a mezőkre bomlana szét, ugyan akkora mérettel és elhelyezkedéssel. Sajnos ezt, az utasítások célja és eltérő igényei miatt nem lehet megvalósítani. Az utasítások kódolása többnyire utasítás típusonként változik a RISC processzoroknál, ami tervezéskor rugalmasságot biztosít. Itt jelenik meg az utolsó tervezési alapelv, ami szerint a jó tervezés kompromisszumokat igényel. Habár a különböző utasításformátumok bonyolítják a dekódolást, de lehetővé teszik az egységes 32 bites utasításméretet.

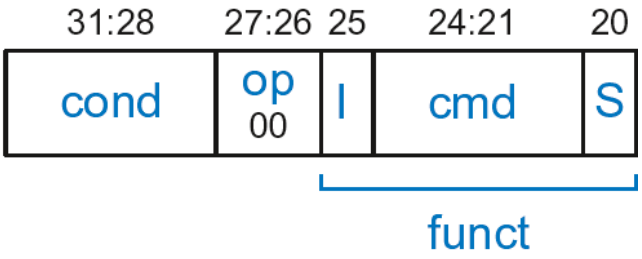
A jegyzetben tanult processzorarchitektúránál három utasításcsoportról volt szó: adatfeldolgozó utasítások, memória utasítások és elágaztató utasítások. Először nézzük meg a 9. ábrán, hogy az adatfeldolgozó utasításoknál milyen almezőkre bomlik fel az utasítás kódolásához használt 32-bit.



9. ábra. Az adatfeldolgozó utasítások 32-bites gépi-kódjának mezői.

Balról jobbra féle haladva, az első a feltétel mező (cond), amely az utasításhoz társított feltételes kapcsoló kódját tartalmazza, amennyiben az utasítás végrehajtása feltételhez kötött. A következő a műveleti kód (op) mező, aminek az értéke 00 lesz az adatfeldolgozó

utasításokban. A harmadik a funkció mező (func), ami további három almezőre bomlik szét, ahogy az a 10. ábrán is látható.



10. ábra. A funkció mező részei.

Az első almező egy bites (I) és azt jelzi, hogy az utasítás második operandusa konstans (1) vagy regiszter (0). A cmd mező az utasítás azonosítója, ami az jelöli, hogy milyen adatfeldolgozó utasítást kell elvégeznie a processzornak. Végül az S almező az utasításhoz társítható S kapcsolónak a használatát jelöli. Emlékezz vissza, hogy ezzel a kapcsolóval jelezzük, ha az ALU-nak frissítenie kell a jelzőbiteket az utasítás végrehajtását követően. Ha az utasításban szerepel az S kapcsoló, akkor ennek az almezőnek az értéke 1, különben 0.

A soron következő mező a 4-bites Rn, ami a célregiszter indexét tartalmazza. A mező mérete azért 4-bites, mivel összesen 16 regiszter van, így ezek bináris indexei 4-bitet igényelnek. Az Rn mezőt az Rd követi, ami az első operandus indexe. Az első operandusnak regiszternek kell lenni, így hasonlóan az Rd-hez ez a mező is egy regiszternek az indexét tartalmazza. Végül az Src2 mező zárja a sort. Az eddigi információk alapján könnyen ki lehet találni azt, hogy ez a mező a második operandusról hordoz információt. Azon felül, hogy a második operandus lehet konstans és regiszter, az ARM még azt is megengedi, hogy a regiszter tartalmának felhasználása előtt, még valamilyen logikai utasítást végezzünk el rajta. Mivel több lehetőség is van a második operandus megadására, ezért az Src2 mező több eltérő almezőre bomlik fel az operandus típusától függően.

•••

Irodalomjegyzék

[1]. D. A. Patterson, J. L. Henessy, Computer Organization and Design. The hardware/software interface, ARM Edition. Morgan Kaufmann, 2017.

[2]. S. L. Harris, D. M. Harris, Digital design and computer architecture, ARM edition. Morgan Kaufmann, 2016.

[3]. M. Wolf Computers as components. Principles of embedded computing system design, 4th ed., Morgan Kaufmann, 2017.