Miskolci Egyetem . . . és Informatikai Kara

Végh János

Bevezetés a számítógépes rendszerekbe – programozóknak

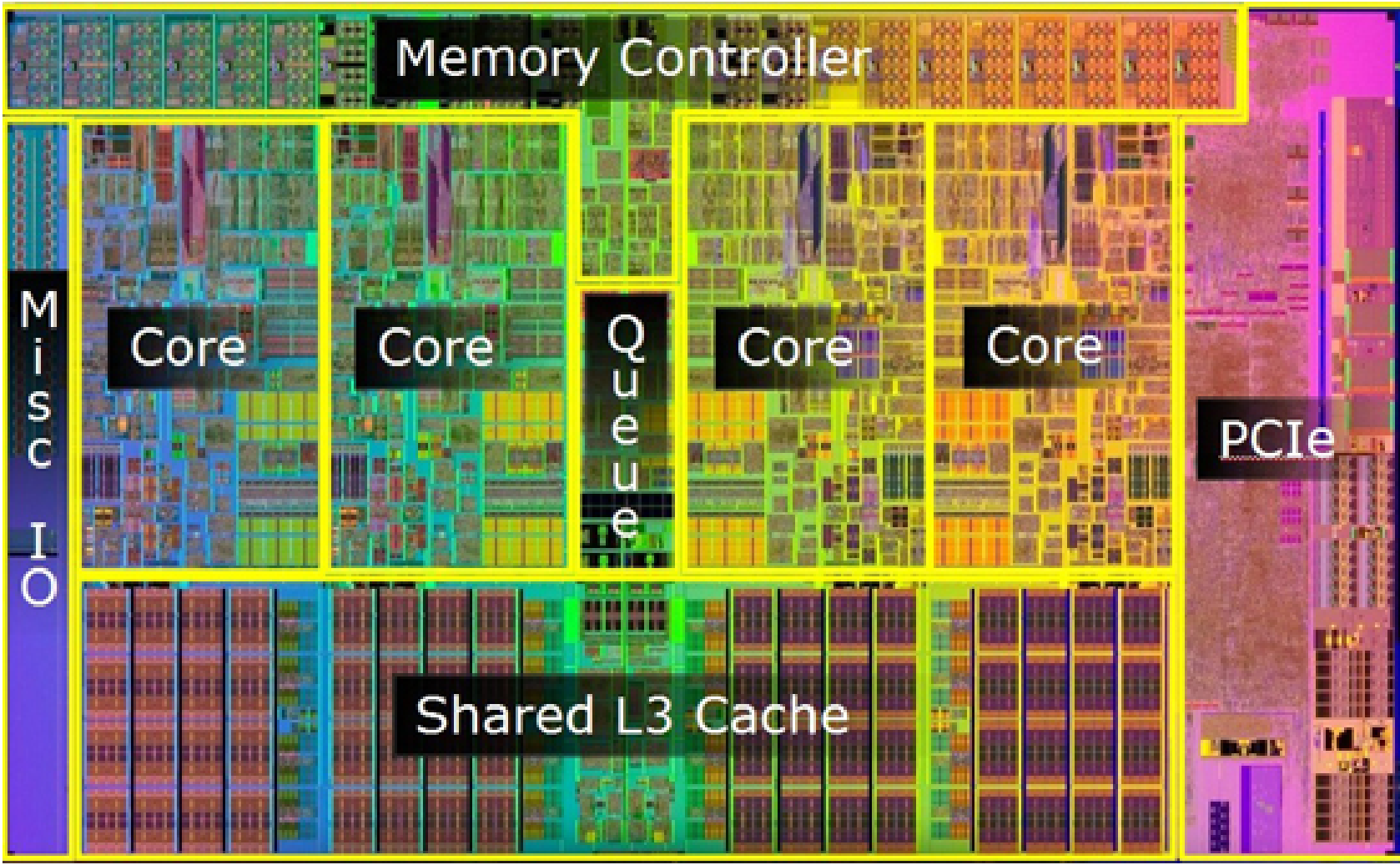
A processzor szerkezete

V0.11@2015.02.16

Ez a segédlet a *Bevezetés a számítógépes rendszerekbe* tárgy tanulásához igyekszik segítséget nyújtani. Nagyrészt az irodalomjegyzékben felsorolt Bryant-O'Hallaron könyv részeinek fordítása, itt-ott kiegészítve és/vagy lerövidítve, néha más jó tankönyvek illeszkedő anyagaival kombinálva. A képzés elején, még a számítógépekkel való ismerkedés fázisában kerül sorra, amikor előbukkannak a különféle addig ismeretlen fogalmak, és megpróbál segíteni eligazodni azok között. Alapvetően a számítógépeket egyfajta rendszerként tekinti és olyan absztrakciókat vezet be, amelyek megkönnyítik a kezdeti megértést.

Ez az anyag még erőteljesen fejlesztés alatt van, akár hibákat, ismétléseket, következetlenségeket is tartalmazhat. Ha ilyet talál, jelezze a fenti címen. Az eredményes tanuláshoz szükség van az irodalomjegyzékben hivatkozott forrásokra, és az órai jegyzetekre, ottani magyarázatokra is.

Tartalomjegyzék


Memory Controller
Misc IO
Core
Core
Queue
Core
Core
PCIe
Shared L3 Cache

FEJEZET

1

A processzor felépítése

Egy modern mikroprocesszor az ember által alkotott legbonyolultabb szerkezetek egyike. Egy körömnyi méretű szilícium lapka tartalmazhat egy teljes nagy teljesítményű processzort, nagy gyorsító memóriát, és a külvilágban levő eszközökhöz kapcsolódáshoz szükséges interfészeket. Számítási teljesítmény szempontjából a mai, egy lapkán megvalósított processzorok mellett eltörpülnek a 20 évvel ezelőtt 10 millió USD-

be került, szoba méretű szuperszámítógépek. Még a mindennapi készülékeinkben (mobil telefonok, digitális személyi asszisztensek, játék konzolok, stb.) található ún. beágyazott processzorok is sokkal nagyobb számítási kapacitásúak, mint amit a korai számítógépek fejlesztői valaha is reméltek.

Eddig a számítógépes rendszereket csak a gépi kódú programozás szintjéig ismertük meg. Megértettük, hogy egy processzornak utasítások sorozatát kell végrehajtani, ahol mindegyik utasítás egy olyan egyszerű műveletet hajt végre, mint pl. két szám összeadása. Egy utasítást bináris formában egy vagy több bájt ábrázol. Az egy bizonyos processzor által támogatott utasítások és azok bájt-szintű kódolása az utasítás készlet architektúra (instruction-set architecture, ISA). A különböző processzor "családok", mint az Intel IA32, IBM/Freescale PowerPC, az ARM processzor család esetében az ISA különböző. Az egyik típusú processzorra fordított program nem fut a másikon. Másrészt viszont egyazon processzor családon belül több különböző modell létezik. Mindegyik gyártó készít folyton növekvő számítási teljesítményű és összetettségű processzorokat, amelyek különböző modelljei ISA szinten kompatibilisek. Bizonyos népszerű családok, pl. az IA32, tagjait több gyártó is előállítja. Emiatt az ISA egy fogalmi réteget jelent a fordítóprogram írók számára, akiknek csak azt kell tudniuk, milyen

utasításokat lehet használni és mi a kódjuk; és csak a processzor tervezőknek kell tudni, hogy hogyan kell azt megépíteni.

Ebben a fejezetben a processzor hardver tervezésére vetünk egy pillantást. Azt tanuljuk meg, milyen módon tudja a hardver egy bizonyos ISA utasításait végrehajtani. Ilyen módon jobban megértjük, hogyan működnek a számítógépek és hogy milyek kihívásokkal szembesülnek a számítógép gyártók. Nagyon fontos szempont, hogy egy modern processzor tényleges működési módja alapvetően eltérhet az ISA által feltételezett számítási modelltől. Az ISA modellből úgy tűnik, hogy csak soros végrehajtás lehetséges, ahol mindegyik utasítást a következő utasítás elkezdése előtt elő kell venni és végrehajtani. Több utasítás különböző részeit egyidejűleg végrehajtva, a processzor lényegesen nagyobb hatékonysággal működik, mintha csak egy utasítást hajtana végre egyszerre. Különleges mechanizmusokra van szükség ahhoz, hogy a processzor ugyanazokat az eredményeket számítsa ki, mint soros végrehajtás esetén. A számítógép tudományban jól ismert, hogy ügyes trükkökkel javíthatjuk a hatékonyságot, míg megőrizzük az egyszerű és absztrakt modell funkcionalitását. Példa ilyenre a WEB böngészők gyorsítótára vagy a kiegyensúlyozott bináris fa és a hash tábla adatszerkezetek.

Valószínűleg nem kerülünk be egy processzor tervező csapatba. Ezekből csak pár tucatnyi létezik. Akkor meg miért tanuljunk processzort tervezni?

- ***Intellektuálisan érdekes és fontos.*** Jó tudni úgy általában, hogyan működnek a dolgok. Különösen érdekes megtanulni annak a rendszernek a belső működését, amelyik a számítógépes szakemberek mindennapjainak részét képezik, és mégis sokuk számára rejtély marad. A processzor tervezésből leszűrhetjük a jó tervezési gyakorlat irányelveit, hogy hogyan készíthetünk egy komplex feladatra egyszerű és szabályos struktúrát.
- ***Megértjük a számítógépes rendszerek működését.*** A processzor-memória interfész processzor oldalának megértése betekintést nyújt a memória rendszerbe és azokba a technikákba, amelyek nagy memóriához gyors elérést nyújtanak.
- ***Sokan terveznek processzort tartalmazó hardvert.*** Mind jellemzőbbé válik, hogy a mindennapi eszközeinkbe is (beágyazott) processzor kerül. Az ilyen beágyazott rendszerek tervezőinek meg kell érteniük, hogyan működnek a processzorok, mivel az ilyen rendszereket az asztali számítógépek szintjénél mélyebb absztrakciós szinten tervezik és programozzák.
- ***Akár processzort is fejleszthetünk.*** Bár csak kevés cég foglalkozik processzor

gyártással, a tervező csoportok létszáma gyorsan növekszik. A processzor tervezés különböző vonatkozásaiban akár 1000 embert is foglalkoztathatnak. Emellett a modern programozható logikai eszközökön akár saját processzort is tervezhetünk.

Azzal kezdjük, hogy definiálunk egy egyszerű utasítás készletet, amit futtatható példaként használunk megvalósítandó processzorunkhoz. Ennek az utasításkészletnek az "Y86" nevet adjuk, mivel az ötlet az IA32 utasításkészletből származik, amit közbeszédben az "x86" néven használunk. Az IA32-höz képest az Y86 utasításkészletben kevesebb adattípus, utasítás és címzési mód van; egyszerűbb a bájt-szintű kódolása is. Ennek ellenére elegendően teljes ahhoz, hogy egész típusú adatokat manipuláló programokat írhassunk rajta. Az Y86 utasítás készletet implementáló terv készítése során a processzor tervezés számos kihívásával találkozunk.

Ennek során egy kevés digitális hardver tervezési ismeretet is tanulunk. Megtanuljuk, hogy egy processzor milyen alapvető építőkövekből áll, azokat hogyan kell összekapcsolni és működtetni. Ennek során a Boole-algebra és a bit-szintű műveletek fogalmaira támaszkodunk. Bevezetünk egy egyszerű HCL "Hardware Control Language" nyelvet, amelyen a hardver rendszerek vezérlési részeit írjuk le. Azoknak is érdemes

ezt átolvasni, akiknek már van valamennyi háttér ismerete logikai tervezésből, hogy megtanulják az itt alkalmazott jelöléseket.

A processzor tervezésének első lépéseként bevezetünk egy funkcionálisan helyes, de nem praktikus, *soros* működésű Y86 processzort. Ez a processzor minden órajel hatására egy teljes Y86 utasítást hajt végre. Az órának elég lassan kell futnia, hogy műveletek sorát lehessen egyetlen órajel alatt végrehajtani. Egy ilyen processzort megvalósíthatunk, de annak teljesítménye jóval alatta lesz annak, ami ennyi hardver felhasználásával elérhető.

A soros tervet alapul véve, átalakítások sorozatán át készítünk egy *futószalagos* processzort is. Ez a processzor az egyes utasítások végrehajtását öt lépésre bontja, amelyek mindegyikét a hardver egy jól elkülöníthető szakasza hajtja végre. Az utasítás végrehajtása a futószalag egyes fázisain halad végig, miközben minden új órajelre egy új utasítás kerül rá a futószalagra. Ennek eredményeként a processzor öt utasítás különböző részeit egyidejűleg hajtja végre. Hogy a processzorunk megőrizze az Y86 ISA soros viselkedését, számos hazárd helyzetet kell kezelnünk, ahol az utasítás helye vagy operandusa függ az éppen még a futószalagon levő másik utasítástól.

Számos segédprogram létezik processzor terveinkkel való kísérletezéshez. Van az

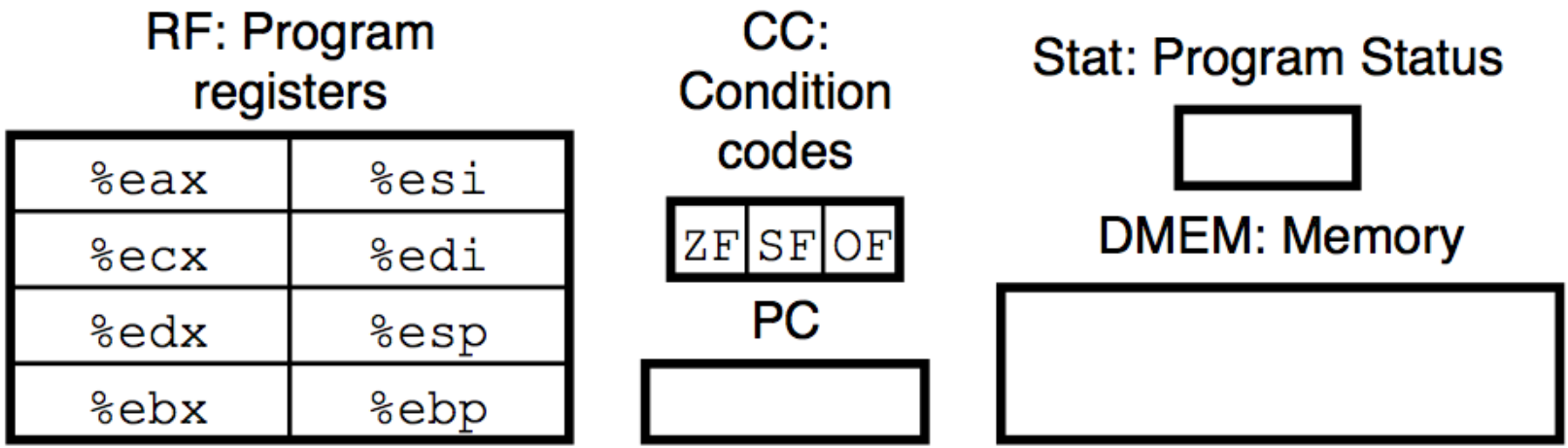
Y86 assembler, az Y86 programot számítógépünkön futtató szimulátor, valamint két soros és egy futószalagos processzor szimulátor. A tervek vezérlő logikáját HCL jelölést használó fájlokkal adjuk meg. Ezeket a fájlokat átszerkesztve és a szimulátort újrafordítva, megváltoztathatjuk és kiterjeszthetjük a szimulátor működését. Számos olyan gyakorlat van, amely új utasítások implementálását tartalmazza és lehetővé teszi, hogy módosítsuk az utasítások végrehajtásának módját. Teszt kódokat is találunk, amelyekkel ellenőrizhetjük módosításaink helyességét. Ezek a gyakorlatok nagyban segítik az anyag megértését és megmutatják azokat a tervezési alternatívákat, amelyekkel a processzor tervezők találkoznak.

1.1. Az Y86 utasítás készlet architektúra

Amikor egy utasításkészletet definiálunk, mint amilyen az Y86 is, meg kell adnunk a különféle állapot elemeket, az utasítás készletet és annak kódolását, a programozási konvenciókat, valamint a kivételes események kezelésének módját.

1.1.1. A programozó által látható állapot

Az Y86 programban az egyes utasítások olvashatják és módosíthatják a processzor állapot (lásd 1.1 ábra) valamely részét. Erre hivatkozunk, mint a programozó által látható állapotra, ahol a "programozó" vagy az a személy, aki az assembly nyelvű programot írja, vagy az a fordítóprogram, amelyik gépi kódot állít elő. A processzor implementációkban látni fogjuk, hogy nem szükséges pontosan az ISA által feltételezett módon ábrázolni és szervezni azt az állapotot, ha biztosítani tudjuk, hogy a gépi kódú programok hozzáférjenek a programozó által látható állapothoz. Az Y86 processzor állapota az IA32 állapotához hasonló. Az Y86-nak nyolc program regisztere van: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, és `%ebp`. Ezek mindegyike egy szót tárol. Az `%esp` regisztert használjuk verem mutatóként a `push`, `pop`, `call` és `return` utasításokban. Ettől eltekintve, a regiszteréknek nincs rögzített jelentése vagy értéke. Van három egybites feltétel kód (condition code): `ZF`, `SF` és `OF`, amelyek a legutóbbi aritmetikai vagy logikai műveletről tárolnak információt. A program számláló (`PC`) tartalmazza az éppen végrehajtás alatt levő utasítás címét.



1.1. ábra. **Az Y86 programozó által látható állapota.**

A memória fogalmilag egy nagy bájt tömb, amelyik programot és adatot is tartalmaz. Az Y86 programok a memória címekre virtuális címeket használva hivatkoznak. Ezeket a címeket hardver és operációs rendszer kombináció fordítja le valódi címekre (más néven fizikai cím), amelyek azt adják meg, hogy az érték ténylegesen hol tárolódik a memóriában. A virtuális memória rendszert úgy képzelhetjük el, hogy az biztosítja az

Y86 programok számára a monolitikus bájt tömb képet.

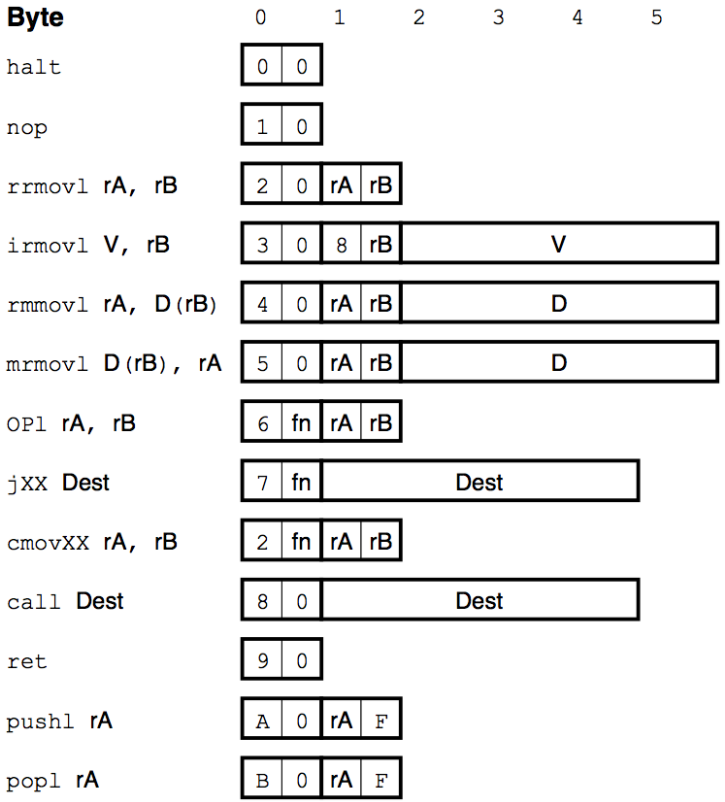
A program állapot utolsó darabja a **Stat** *állapot kód*, amely a program általános állapotát írja le. Ez vagy normális állapotot jelez, vagy valamilyen kivételes esemény bekövetkeztét, mint például hogy egy utasítás érvénytelen memória címről próbált meg olvasni.

1.1.2. Az Y86 utasítások

Az Y86 utasításkészlete (lásd 1.2 ábra) az a tömör összefoglaló, amelynek alapján fogjuk megvalósítani processzorainkat. Ez az utasításkészlet az IA32 utasításkészletének egy alrendszere. Ez csak 4-bájtos egész típusú műveleteket tartalmaz, kevesebb címzési móddal és kisebb utasítás készlettel. Mivel csak 4-bájtos adatokat használunk, ezekre "szó" ("word") kifejezéssel egyértelműen hivatkozhatunk. Az ábrán bal oldalt az utasítások assembly kódja, jobb oldalt a megfelelő bájt kód látható. Az assembly kód formátuma az ATT által használt IA32 formátumra hasonlít.

További részletek az Y86 utasításokról.

- Az IA32 `movl` utasítása négy utasításra oszlik: `irmovl`, `rrmovl`, `mrmovl`, és `rmmovl`, amelyek explicit módon jelzik a forrás és a cél formáját. A forrás lehet közvetlen (immediate, (i)), regiszter (register, r), vagy memória (memory, m), amit az utasítás nevének első betűje jelöl. A cél lehet regiszter (register, r), vagy memória (memory, m), amit az utasítás nevének második betűje jelöl. A négy típus explicit megkülönböztetése sokat segít annak eldöntésében, hogyan implementáljuk azokat.



1.2. ábra. **Az Y86 utasításkészlete.**

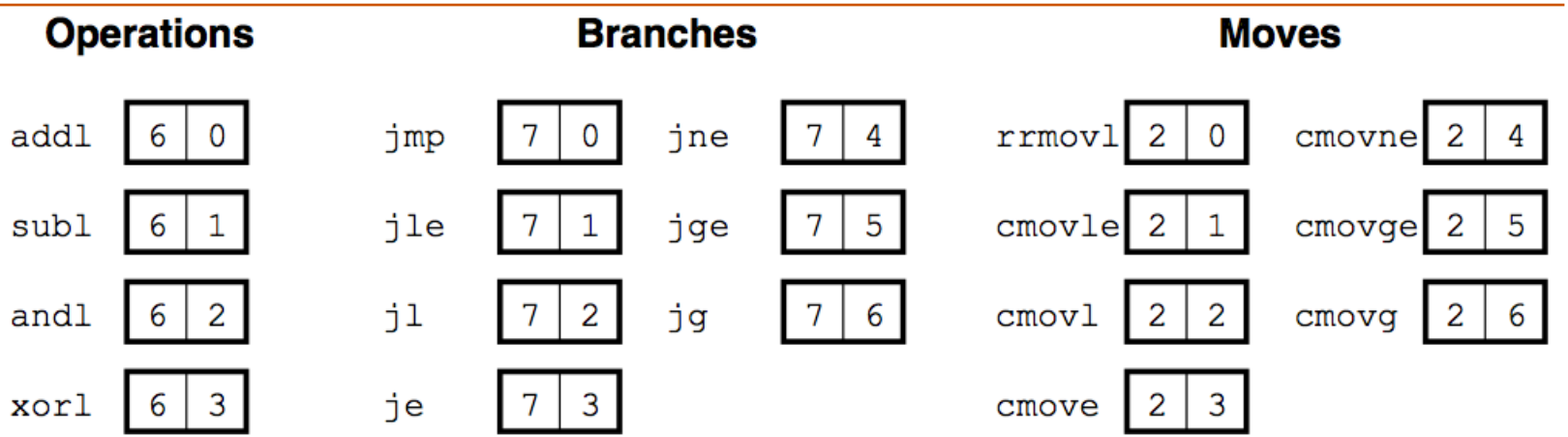
A memória hivatkozást tartalmazó két utasítás formája egyszerűen egy alapból (base) és egy eltolásból (displacement) áll. Nem támogatjuk második index regiszter vagy a cím kiszámításakor regiszter érték skálázás használatát. Mint az IA32 esetében is, itt sem lehet adatot közvetlenül egyik memória helyről egy másikba átvinni, továbbá közvetlen adatot a memóriába írni.

- Négy egész típusú műveletünk van (ezeket `OPl` jelzi): `addl`, `subl`, `andl` és `xorl`. Ezek csak regiszter adatokkal dolgoznak, míg az IA32 memória adatokkal való műveleteket is lehetővé tesz. Ezek az utasítások beállítják a `ZF`, `SF` és `OF` (zero, sign, and overflow) feltétel jelzőbiteket.
- A hét ugró utasítás (ezeket `jXX` jelzi): `jmp`, `jle`, `jl`, `je`, `jne`, `jge` és `jg`. Az elágazások a típusnak és a feltétel kódoknak megfelelően hajtódnak végre. Az elágazási feltételek megegyeznek az IA32 hasonló feltételeivel.
- Van hat feltételes adatmásoló utasítás (ezeket `cmovXX` jelzi): `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, és `cmovg`. Ezeknek ugyanolyan a formátuma, mint a regiszter-regiszter típusú `rrmovl` adatmozgató utasításnak, de a cél regiszter csak akkor frissül, ha a feltétel kódok kielégítik a megkövetelt kényszereket.
- A `call` utasítás a visszatérési címet a verembe írja és a cél-címre ugrik. A `ret`

utasítás tér vissza az ilyen hívásból.

- The `pushl` és `popl` utasítások implementálják a verembe írás és abból olvasás műveleteket, éppúgy, mint az IA32 esetén.
- A `halt` utasítás megállítja az utasítás végrehajtást. Az IA32-nek is van egy hasonló, `hlt` nevű utasítása. Az IA32 alkalmazói programok nem használhatják ezt az utasítást, mivel az az egész rendszer működésének felfüggesztését okozza. Az Y86 esetén a `halt` utasítás a processzor leállását okozza, és az állapot kódot HLT-ra állítja.

1.1.3. Az utasítások kódolása



1.3. ábra. **Az Y86 utasítás készletének funkció kódjai.** A kód egy bizonyos egész műveletet, elágazási feltételt vagy adatátviteli feltételt ad meg. Ezeket az utasításokat `OPl`, `jXX`, és `cmovXX` mutatja, lásd 1.2 ábra.

Az Y86 utasításai (lásd 1.3 ábra) 1 és 6 bájt közötti hosszúságúak, attól függően, milyen mezőket használ az utasítás. Minden utasítás első bájtja az utasítás típusát

azonosítja. Ez a bájt két 4-bites részre oszlik: a nagyobb helyértékű (kód) részre és a kisebb helyértékű (funkció) részre. A kód értékek (lásd 1.2 ábra) tartománya **0** és **0xB** közé esik. A funkció értékek csak akkor számítanak, amikor a hasonló utasítások osztoznak egy közös kódon. Ilyen esetek az egész típusú értékekkel végzett műveletek, a feltételes adatmásolás és az elágazás, lásd 1.3 ábra. Figyeljük meg, hogy `rrmovl` utasításkódja ugyanaz, mint a feltételes adatmásolásnak. Tekinthetjük "feltétel nélküli adatmásolásnak" is, éppúgy mint a `jmp` utasítást feltétel nélküli elágazásnak; mindkettő funkció kódja **0**.

A nyolc program regiszter mindegyikének van egy egyedi azonosítója (ID), ami egy 0 és 7 közötti érték, lásd 1.1 táblázat. Az Y86 regisztereinek számozása megegyezik az IA32 számozásával. Ezek a regiszterek a CPUn belül a regiszter tömbben találhatók, egy kis méretű RAM memóriaként, ahol a regiszter azonosítók szolgálnak címként. A **0xF** azonosítót arra használjuk, az utasítás kódolásakor és az azt megvalósító hardverben, hogy nem kell regisztert elérnünk.

Néhány utasítás csak 1 bájt hosszú, de amelyeknek operandusa is van, azok kódolásához több bájt szükséges. Először is, szükség lehet olyan regiszter kijelölő bájtra, amelyik meghatároz egy vagy két regisztert. Ezeket a regiszter mezőket jelöli **rA** és **rB**, lásd 1.2

1.1. táblázat. **Az Y86 program regiszter azonosítói.** A nyolc program regiszter mindegyikének van egy szám azonosítója (ID), amelynek értéke 0 és 7 közötti szám.

Azonosító	Regiszter név
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp

ábra. Mint azt az utasítások assembly nyelvű változata mutatja, ezek megadhatják, melyik regisztert használjuk forrás és cél regiszterként, vagy – az utasítás típusától függően – címszámításkor bázis regiszterként. Azok az utasítások, amelyeknek nincs regiszter operandusa (mint pl. az elágazások és a `call` utasítás), nincs regiszter kijelölő bájtja sem. Amely utasításokban csak egy regiszter operandusra van szüksége (`irmovl`, `pushl` és `popl`), a második regisztert kijelölő bitek **0xF** értékűek. Ez a megállapodás hasznosnak bizonyul majd a processzor megvalósítása során.

Néhány utasításnak egy további 4-bájtos konstans szóra is szüksége van. Ez a szó szolgálhat az `irmovl` számára közvetlen adatként, eltolási értékként a `rmmovl` és `mrmovl` számára cím megadásakor, és cél-címként elágazások és `call` utasítások esetén. Megjegyezzük, hogy a két utóbbi esetben a címek abszolút címek, eltérően az IA32 programszámlálóhoz viszonyított (PC-relative) relatív címeitől. A processzorok PC-relatív címzést használnak, mivel az elágazások esetén tömörebb kódolást tesz lehetővé, továbbá lehetővé teszi, hogy a kódot a memória egyik részéből a másikba átmásoljuk, anélkül, hogy a cél-címeket meg kellene változtatni. Mivel azonban célunk az egyszerűség, abszolút címzést fogunk használni. Az IA32-höz hasonlóan, az egész értékekben ún. "little-endian" kódolási sorrendet használunk. Amikor az utasításokat

visszafordított (disassembled) formában látjuk, ezek a bájtok fordított sorrendben jelennek meg.

Példaként tekintsük a `rmmovl %esp,0x12345(%edx)` utasítás hexadecimális bájt kódjának előállítását, lásd 1.2 ábra. Az `rmmovl` utasítás első bájtja **40**. Azt is látjuk, hogy forrás regiszterként az `%esp` regisztert kell az **rA**, és alap regiszterként `%edx` értékét kell az **rB** mezőbe kódolni. A regiszter számokat (lásd 1.1 táblázat) használva, a regiszter kijelölő bájt értékéül **42** adódik. Végül, az eltolást 4 bájtos konstans szóként kell kódolnunk. Először a **0x12345** értéket kitöltjük bevezető nullákkal, hogy a szám kitöltse a négy bájtot, azaz a **00 01 23 45** értéket használjuk. Ezt fordított sorrendben a **45 23 01 00** bájt sorozatként írjuk le. Ezeket összetéve, megkapjuk a **404245230100** utasítás kódot.

Bármely utasítás készlet fontos tulajdonsága, hogy a bájt sorozatoknak egyedinek kell lenniük. Egy önkényesen megadott bájt sorozat vagy egy bizonyos utasításnak felel meg, vagy nem használható bájt sorozatként. Ez a tulajdonság az Y86 esetén fennáll, mivel minden egyes utasítás első bájtjában a kód és a funkció egyedi kombinációt tartalmaz, és ennek a bájtnak a megadásával meg tudjuk határozni a további bájtok hosszát és jelentését. Ez a tulajdonság biztosítja, hogy a processzor egy objekt kódú

porgramot úgy tud végrehajtani, hogy semmi kétség nem merül fel a kód jelentésére vonatkozóan. Még ha a kód a program más bájtjai közé van beágyazva, gyorsan meg tudjuk határozni az utasítás sorozatot, ha a sorozat első bájtjával indulunk. Másrészt viszont, ha nem ismerjük a kódsorozat kezdő bájtjának helyét, nem tudjuk megbízhatóan megmondani, hogyan osszuk fel a kód sorozatot utasításokra. Ez komoly problémát jelent a visszafordítók (disassembler) és más hasonló segédprogramok számára, amikor megpróbálnak gépi kódot kivonni az objekt kód bájt sorozataiból.

1.1.4. Az Y86 kivételek

Az Y86 programozói felülete (lásd 1.1 ábra) tartalmaz egy **Stat** állapot kódot is, ami a végrehajtódó program általános állapotát írja le. Ennek lehetséges értékeit mutatja a 1.2 táblázat. Az **1** kód (**AOK**) azt jelzi, hogy a program rendben végrehajtódik, a többi kód pedig azt jelzi, hogy valamilyen típusú kivétel történt. A **2** kód (**HLT**) azt jelzi, hogy a processzor `halt` utasítást hajtott végre. A **3** kód (**ADR**) azt jelzi, hogy a processzor érvénytelen memóriacímről próbált meg olvasni vagy oda írni, akár utasítás elővétel, akár adat olvasás/írás során. A legnagyobb címet korlátozzuk (a pontos határ implementáció függő), és minden ennél nagyobb cím használata **ADR** kivételt okoz. A **4** kód (**INS**) azt jelzi, hogy érvénytelen utasítás kódot próbált meg végrehajtani a processzor.

Az Y86 esetén egyszerűen meg kell állítanunk a processzort, ha a felsorolt kivételek valamelyike előfordul. Teljesebb kivitel esetén a processzor tipikusan meghívna egy kivétel kezelőt, ami egy kifejezetten ilyen típusú kivétel kezelésére szolgáló eljárás. A kivétel kezelők különbözőképpen konfigurálhatók, hogy különböző hatással bírjanak: például abortálják a programot vagy meghívnak egy, a felhasználó által definiált jelzés

1.2. táblázat. **Y86 állapot kódok.** A mi esetünkben, a processzor minden, az AOK kódtól eltérő kód esetén megáll.

Érték	Név	Jelentés
1	AOK	Normál működés
2	HLT	**halt** utasítást találtunk
3	ADR	Érvénytelen címet

kezelőt.

1.1.5. Y86 programok

A példaként használt összegző programot C nyelven készítettük el, lásd 1.1 programlista. Az IA32 kódot a **gcc** fordító állította elő. Az Y86 kód ezzel lényegében megegyezik, kivéve, hogy néha az Y86-nak két utasításra van szüksége, hogy elvégezze azt, amit egyetlen IA32 utasítással elvégezhetünk. Ha a programot tömb indexeléssel készítettük volna, az Y86 kóddá alakítás még nehezebb lenne, mivel az Y86 nem rendelkezik skálázott címzési módokkal. A kód követ sok, az IA32 esetén használt programozási konvenciót, beleértve a verem és a keret mutatók használatát. Az egyszerűség kedvéért nem követi az I32 konvenciót, hogy bizonyos regisztereket a hívott szubrutinnak kell elmenteni. Az csak egy programozási konvenció, amit vagy elfogadunk vagy elvetünk.

Programlista 1.1: Összeadó program C nyelven

```
int Sum(int *Start, int Count)
{
  int sum = 0;
  while (Count) {
    sum += *Start;
    Start++;
    Count--;
  }
  return sum;
}
```

Programlista 1.2: **Az összeadó program (lásd 1.1 ábra) Y86 és IA32 változatú assembly programjának összehasonlítása.** A **Sum** függvény összegzi egy egész tömb elemeit. Az Y86 kód főként abban tér el az IA32 kódtól, hogy több utasításra is szükség lehet annak elvégzéséhez, amit egyetlen IA32 utasítással elvégezhetünk.

```
;IA32 code
;int Sum(int *Start, int Count)
Sum:
  pushl  %ebp
  movl  %esp,%ebp
  movl  8(%ebp),%ecx  ;ecx=Start
  movl  12(%ebp),%edx  ;edx=Count
  xorl  %eax,%eax  ;sum  =  0
  testl  %edx,%edx
  je  .L34
.L35:
  addl  (%ecx),%eax;add *Start

  addl  $4,%ecx  ;Start++

  decl  %edx  ;Count--

  jnz  .L35  ;Stop  when  0
.L34:
  movl  %ebp,%esp
  popl  %ebp
  ret
```

```
;Y86 code
;int Sum(int *Start, int Count)
Sum:
  pushl  %ebp
  rrmovl  %esp,%ebp
  mrmovl  8(%ebp),%ecx  ;ecx=Start
  mrmovl  12(%ebp),%edx ;edx=Count
  xorl  %eax,%eax  ;sum = 0
  andl  %edx,%edx  ;Set  c. codes
  je  End
Loop:
  mrmovl  (%ecx),%esi  ;get *Start
  addl  %esi,%eax  ;add to sum
  irmovl  $4,%ebx
  addl  %ebx,%ecx  ;Start++
  irmovl  $-1,%ebx
  addl  %ebx,%edx  ;Count--
  jne  Loop  ;Stop  when  0
End:
  rrmovl  %ebp,%esp
  popl  %ebp
  ret
```

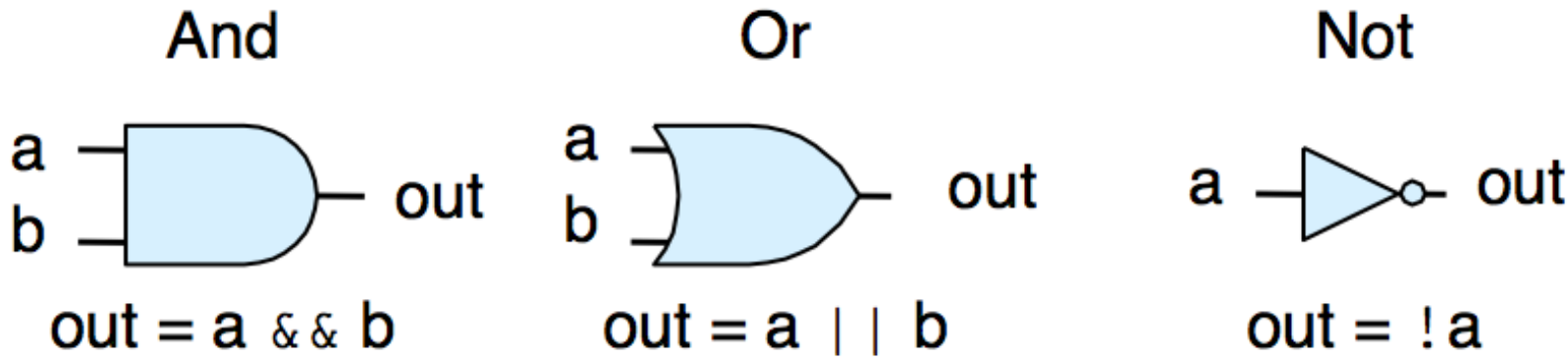
Programlista 1.3: **Minta program Y86 assembly nyelven.**A **Sum** függvényt hívja egy négy elemű tömb elemeinek összegét kiszámolni.

```
#  Execution  begins  at  address  0
    .pos  0
init:  irmovl  Stack,  %esp  #  Set  up  stack  pointer
       irmovl  Stack,  %ebp  #  Set  up  base  pointer
       call  Main  #  Execute  main  program
       halt  #  Terminate  program

#  Array  of  4  elements
       .align  4
array:  .long  0xd
       .long  0xc0
       .long  0xb00
       .long  0xa000

Main:  pushl  %ebp
       rrmovl  %esp,%ebp
       irmovl  $4,%eax
       pushl  %eax  #  Push  4
       irmovl  array,%edx
       pushl  %edx  #  Push  array
       call  Sum  #  Sum(array,  4)
       rrmovl  %ebp,%esp
       popl  %ebp
       ret

       #  int  Sum(int  *Start,  int  Count)
Sum:   pushl  %ebp
       rrmovl  %esp,%ebp
       mrmovl  8(%ebp),%ecx  #  ecx  =  Start
       mrmovl  12(%ebp),%edx  #  edx  =  Count
       xorl  %eax,%eax  #  sum  =  0
       andl  %edx,%edx  #  Set  condition  codes
```

1.1.6. Az Y86 utasítások részletei

1.2. Logikai tervezés és a HCL hardver tervező nyelv

1.2.1. Logikai kapuk



1.4. ábra. **Logikai kapu típusok.** Mindegyik kapu valamelyik Boole-függvénynek megfelelően változtatja a kimenetének az értékét, a bemenet értékeinek megfelelően.

A logikai kapuk a digitális áramkörök alapvető számítási elemei, amelyek a bemeneteik állapota alapján egy Boole függvénynek megfelelő kimeneti állapotot állítanak elő. Az **And**, **Or** és **Not** Boole-függvények standard szimbólumait a 1.4 ábra mutatja. Az ábra alján, a kapuk alatt láthatók a megfelelő HCL kifejezések: **&&** az **And**, || az

Or és **!** a **Not** műveletre. Ezeket a jeleket használjuk a C nyelv bit-szintű **&**, | és ~ bit-szintű operátorai helyett, mivel a logikai kapuk egy-bites mennyiségeken működnek, nem pedig egész szavakon. Bár az ábra csak két-bemenetű **And** és **Or** kapukat ábrázol, azokat n-műveletes kapuként is használják, ahol $n > 2$. A HCL nyelven ezeket is bináris operátorokkal, így egy három bemenetű **And** kaputt, annak **a**, **b** és **c** bemeneteivel, az

```
a && b && c
```

kifejezéssel írunk le.

A logikai kapuk mindig aktívak. Ha valamelyik kapu bemenete megváltozik, rövid időn belül a kimenete is megfelelően megváltozik.

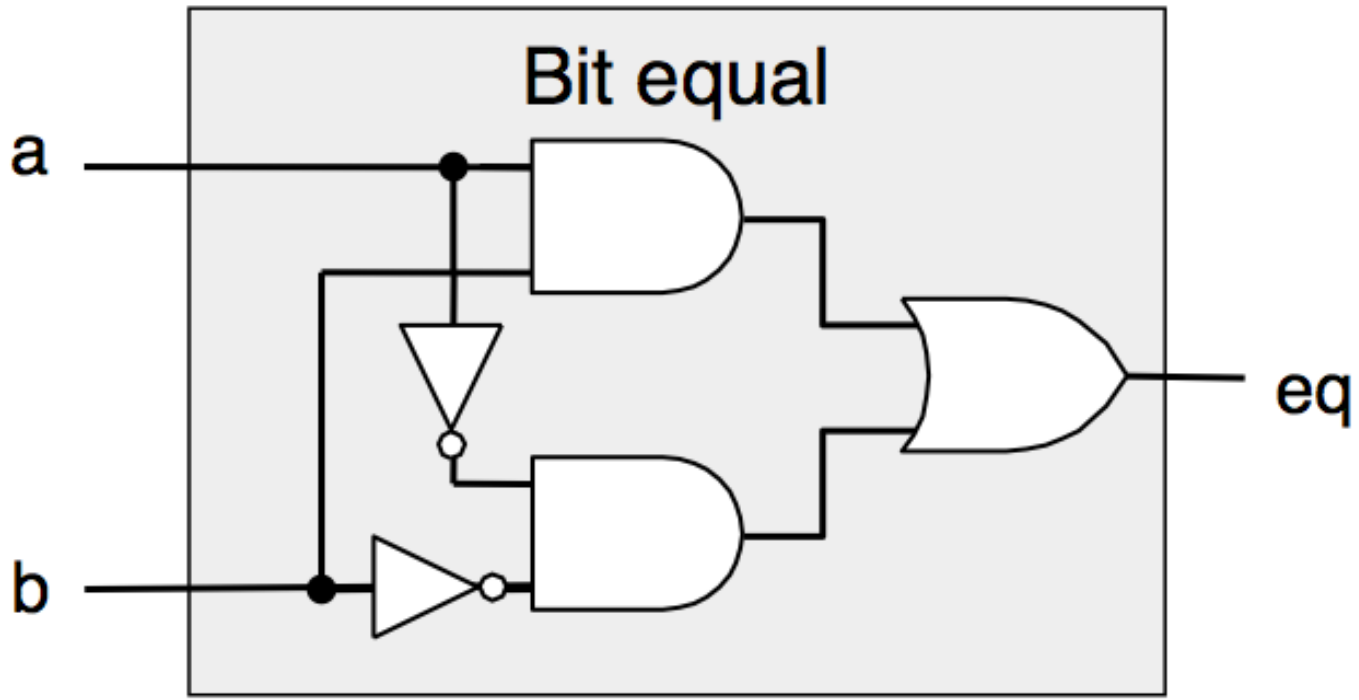
1.2.2. Kombinációs áramkörök és HCL logikai kifejezések

Több logikai kapu hálózatba kapcsolásával olyan, számításra alkalmas blokkokat készíthetünk, amelyeket kombinációs áramköröknek (combinational circuits) nevezünk. Ezekre a hálózatokra két fontos korlátozás vonakozik:

- Két vagy több logikai kapu kimenete nem kapcsolható össze. Egyébként a vezetéket egyidejűleg két különböző irányba is megpróbálhatnánk meghajtani, ami vagy érvénytelen feszültséget eredményez, vagy áramköri meghibásodást okozhat.
- Az áramkör nem lehet ciklikus. Azaz, nem lehet olyan útvonal a hálózatban, amelyik kapuk sorozatán át hurkot képez. Egy ilyen hurok a hálózat által számított érték bizonytalanságát okozhatja.

A 1.5 ábra egy jól használható egyszerű kombinációs áramkört mutat. Az áramkörnek két bemenete van, **a** és **b**. Egy **eq** kimenetet generál, úgy, hogy az **1** értékű lesz, ha **a** és **b** egyaránt **1** értékű (ezt a felső **And** kapu detektálja), vagy mindkettő **0** értékű (ezt az alsó **And** kapu detektálja). Ezt a függvényt a HCL nyelven a

```
bool eq = (a && b) || (!a && !b);
```



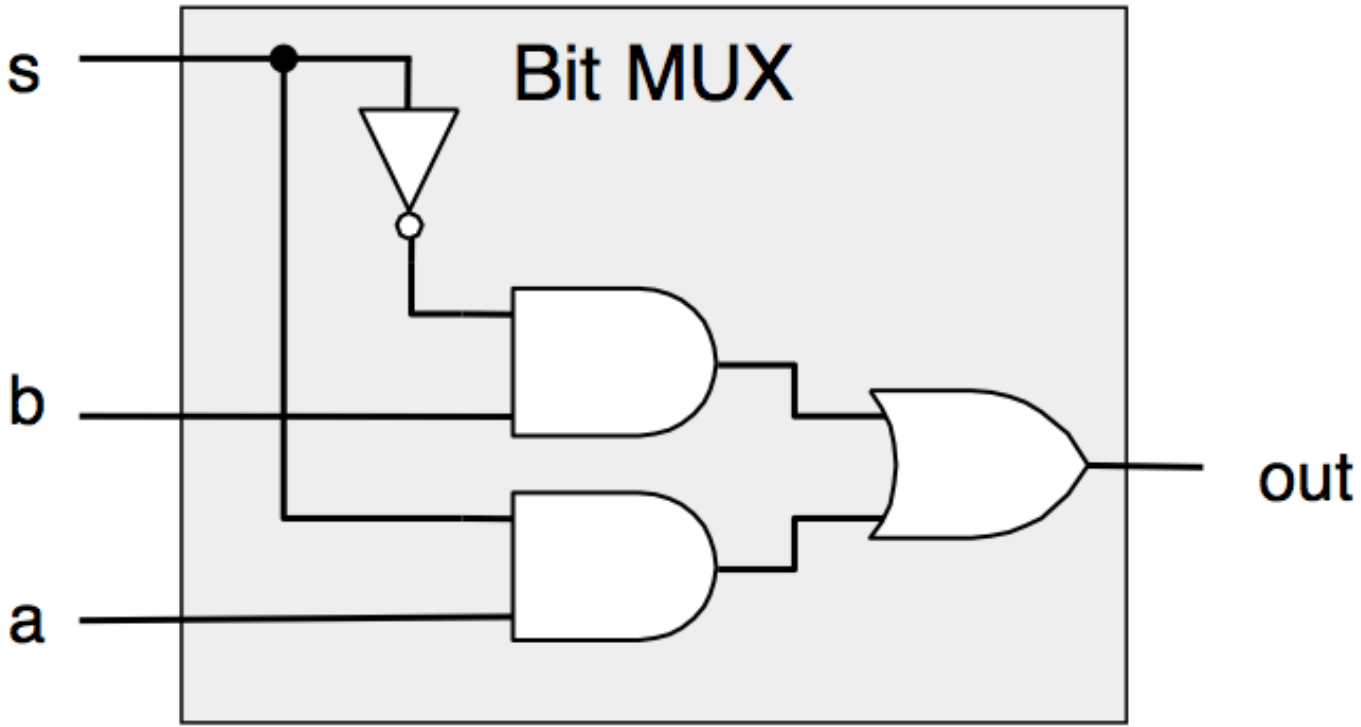
1.5. ábra. **Kombinációs áramkör bit egyenlőség vizsgálatára.** A kimenet **1** értékű lesz, ha mindkét bemenet **0** vagy mindkettő **1**.

kifejezéssel írjuk le.

Ez a kód egyszerűen definiálja (a **bool** típusú adatként megjelölt) **eq** bit-szintű jelet, az **a** és **b** bemenetek függvényeként. Mint a példa mutatja, a HCL nyelv C-szerű szintaxisú, ahol '=' a jel nevét egy kifejezéshez rendeli. A C-től eltér azonban, hogy ezt nem tekintjük valamiféle számítás elvégzése eredményének és az eredmény valamiféle memóriahelyre írásának. Ez egyszerűen csak olyan módszer, amellyel egy nevet rendelhetünk egy kifejezéshez.

A 1.6 ábra egy másik egyszerű, de jól használható kombinációs áramkört mutat, amit multiplexer (vagy röviden MUX) néven ismerünk. Egy multiplexer különböző bemenő jelek közül választ ki egyet, egy bemeneti vezérlő jel értékétől függően. Ebben az egybites multiplexerben az adatjelek az **a** és **b** bemeneti bitek, a vezérlő jel pedig az **s** bemeneti bit. A kimenet értéke egyenlő lesz **a**-val, amikor **s** értéke **1**, és egyenlő lesz **b**-vel, amikor **s** értéke **0**. Ebben az áramkörben a két **And** kapu határozza meg, hogy átengedik-e saját bemenő adatukat az **Or** kapunak.

A felső **And** kapu akkor engedi át a **b** jelzést (mivel annak a másik bemenetén **!s** van), amikor **s** értéke **0**, az alsó **And** kapu pedig akkor, amikor **s** értéke 1. A kimenő jelet leíró kifejezés, ami ugyanazokat a műveleteket használja, amiket a kombinációs áramkör:



1.6. ábra. **Egybites multiplexer áramkör.** A kimenet értéke megegyezik az **a** bemenet értékével, amikor az **s** vezérlő jel **1** és megegyezik a **b** bemenet értékével, amikor **s** értéke **0**.

```
bool out = (s && a) || (!s && b);
```

HCL nyelvű kifejezéseink világosan rámutatnak arra a párhuzamra, amely a kombinációs logikai áramkörök és a C nyelvű logikai kifejezések között van. Mindkettő logikai kifejezéseket használ arra, hogy kiszámítsa a kimeneti értéket a bemenetek függvényében. Azonban van néhány olyan különbség a számítás eme kétféle kifejezése között, amelyre érdemes felfigyelnünk:

- Mivel a kombinációs áramkör logikai kapuk sorából áll, jellemző tulajdonsága, hogy a kimenetek folytonosan követik a bemenetek változását. Ha valamelyik bemenet megváltozik, akkor bizonyos késleltetés után, a kimenetek is megfelelően megváltoznak. Ezzel szemben egy C kifejezés csak egyszer számítódik ki, amikor az a program végrehajtása során sorra kerül.
- A C logikai kifejezésekben tetszőleges egész érték megengedett, és a **0** értéket tekintjük **false** értéknek és minden egyebet **true** értéknek. Ezzel szemben a logikai kapuk kizárólag **0** és **1** logikai értékekkel működnek.
- A C logikai kifejezéseknek van olyan tulajdonságuk, hogy részben is kiértékelhetők. Ha egy **And** vagy **Or** művelet eredménye az első argumentum kiértékeléséből

már meghatározható, akkor a másodikat már ki sem értékeljük. Például, a

```
(a && !a) && func(b,c)
```

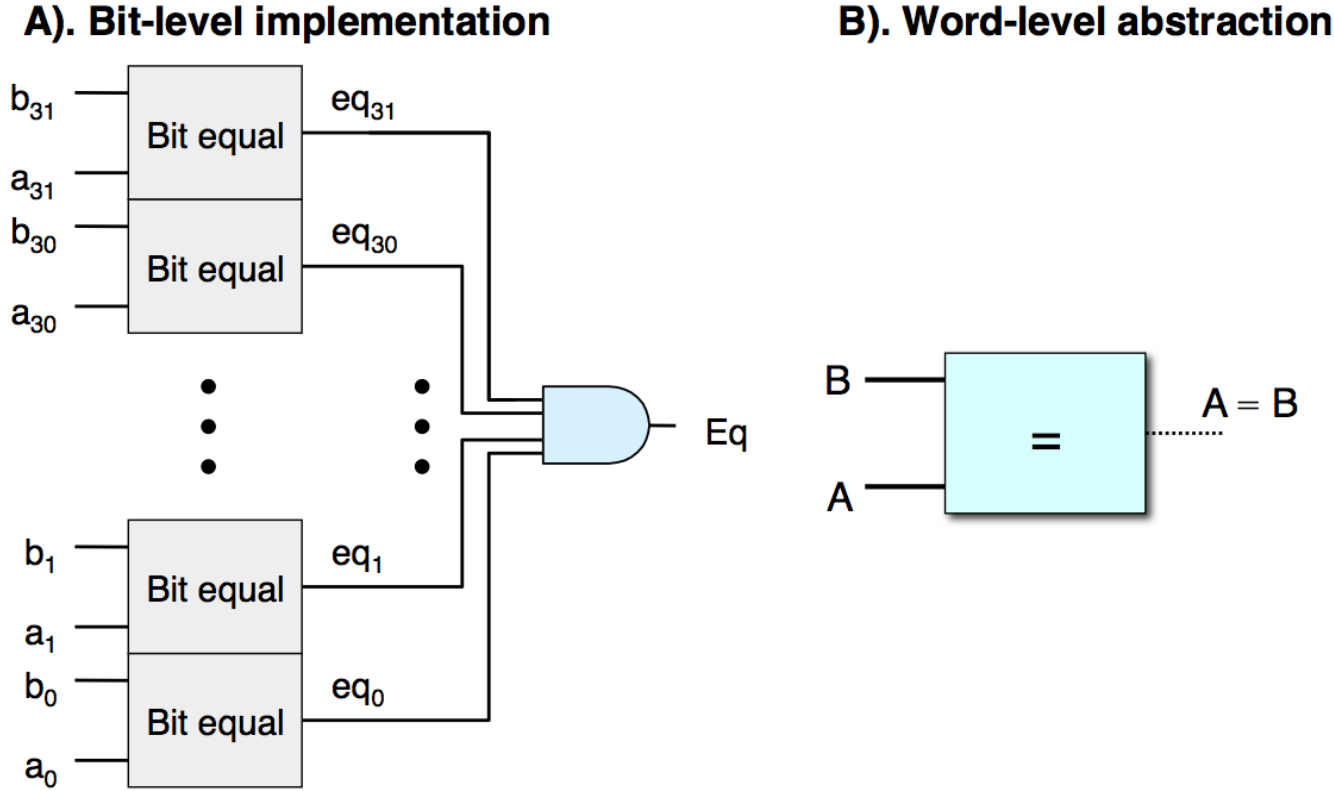
kifejezés esetén a `func` függvényt meg sem hívjuk, mivel a `(a && !a)` kifejezés értéke biztosan `0`. Ezzel szemben a kombinációs logikákra nem vonatkozik valamiféle rész-kiértékelési szabály. A kapuk egyszerűen csak válaszolnak a változó bemeneteikre.

1.2.3. Szó-szintű kombinációs áramkörök és HCL logikai kifejezések

Logikai kapukból nagy hálózatokat összeállítva, olyan kombinációs áramköröket hozhatunk létre, amelyek sokkal összetettebb függvényeket számítanak ki. Tipikusan olyan áramköröket fogunk tervezni, amelyek adat szavakkal működnek. Ezek olyan, bit-szintű jelekből álló csoportok, amelyek egy egész számot vagy valami vezérlő jelet ábrázolnak. Például, leendő processzorunk különféle szavakat tartalmaz, amelyeknek hossza 4 és 32 bit közé esik, és amelyek egész számokat, címeket, utasítás kódokat és regiszter azonosítókat ábrázolnak.

A szó-szintű számításokat végző kombinációs áramkörök úgy épülnek fel, hogy logikai kapukat használnak a kimenő szó egyes bitjeinek kiszámítására és a bemenő szavak egyes bitjein alapulnak. Például, a 1.7 ábra olyan áramkört mutat, amelyik azt vizsgálja, hogy az **A** és **B** 32-bites szavak egyenlők-e. Azaz a kimenet akkor és csak akkor lesz **1** értékű, ha **A** minden egyes bitje megegyezik **B** megfelelő bitjével. Ezt az áramkört úgy valósíthatjuk meg, hogy 32 példányt használunk a 1.5 ábrán bemutatott bit-egyenlőség vizsgáló áramkörből és azok kimenetét egy **And** kapu bemeneteivé kombináljuk.

A HCL nyelven egy szó-szintű jelet deklarálunk **int**-ként, a szó méret megadása



1.7. ábra. **Szavak egyenlőségét vizsgáló áramkör.** A kimenet értéke **1**, amikor az **A** szó minden egyes bitje megegyezik a **B** szó megfelelő bitjével. A szó-szintű egyenlőség a HCL egyik művelete.

nélkül. Ezt csak az egyszerűség érdekében tesszük. Egy valódi hardver leíró nyelvben minden egyes szót úgy deklarálhatunk, hogy a bitek számát is megadjuk. A HCL lehetővé teszi, hogy szavak egyenlőségét vizsgáljuk, így a 1.7 ábrán bemutatott áramkör funkcionalitását a

```
bool Eq = (A == B);
```

egyenlőséggel fejezhetjük ki, ahol az **A** és **B** argumentumok egész típusúak. Vegyük észre, hogy ugyanazokat a szintaxis konvenciókat használjuk, mint a C nyelvben: '=' jelöli az értékadást, míg '==' az egyenlőség operátor.

Amint a 1.7 ábra jobb oldala mutatja, a szó-szintű áramköröket közepes vastagságú vonallal rajzoljuk, hogy a szó több vezetékét ábrázoljuk, az eredményül kapott logikai jelet pedig szaggatott vonallal.

A 1.8 ábra egy szó-szintű multiplexer áramkört mutat. Ez az áramkör egy 32-bites **Out** szót generál, amelyik megegyezik a két bemenő szó, **A** és **B**, valamelyikének értékével, az **s** vezérlő bit értékétől függően. Az áramkör 32 azonos al-áramkörből áll, amelyek mindegyike a 1.6 ábrán bemutatotthoz hasonló szerkezetű. Ez azonban nem egyszerűen a bit-szintű multiplexer 32 másolata: a szó-szintű változat csökkenti az inverterek számát úgy, hogy csak egyszer állítja elő a **!s** jelet és minden bit pozícióban

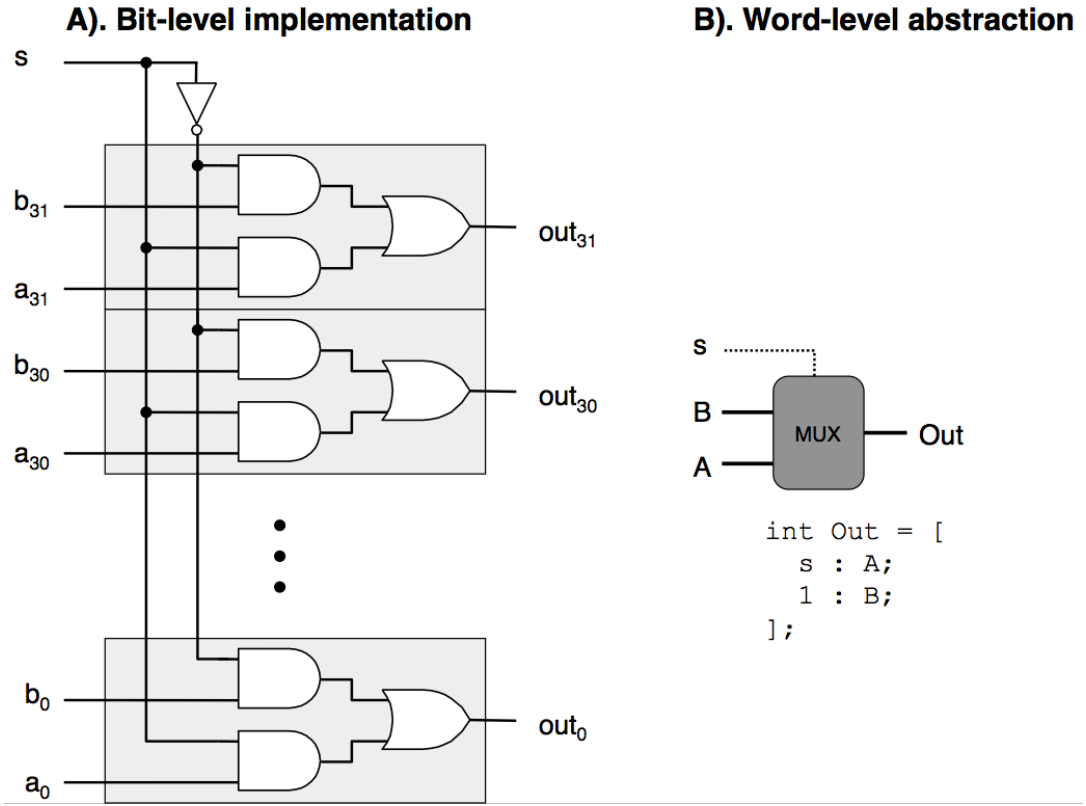
azt használja.

A processzor tervezés során sok formában fogunk multiplexert használni. Ez teszi lehetővé, hogy valamilyen vezérlő feltételtől függően kiválasszunk egy szót több forrás lehetőség közül. A multiplexelő függvényeket a HCL nyelvben **case** kifejezések írják le. Egy **case** kifejezés általános formája:

[
$select_1$: $expr_1$
$select_2$: $expr_2$
:
$select_k$: $expr_k$
]

A kifejezés esetek sorát tartalmazza, ahol az egyes esetek tartalmaznak egy $select_i$ kifejezés választót, ami azt adja meg, melyik esetet kell választani, valamint egy $expr_i$ kifejezést, ami az eredményt adja meg.

A C nyelv **switch** utasításától eltérően, nem követeljük meg, hogy a választó kifejezések kölcsönösen kizárók legyenek. Logikailag, a választó kifejezések sorban értékelődnek ki, és az első olyan esetet választjuk, amelyiknek eredménye **1**. Például, a 1.8 ábrán szereplő multiplexert leíró HCL kifejezés



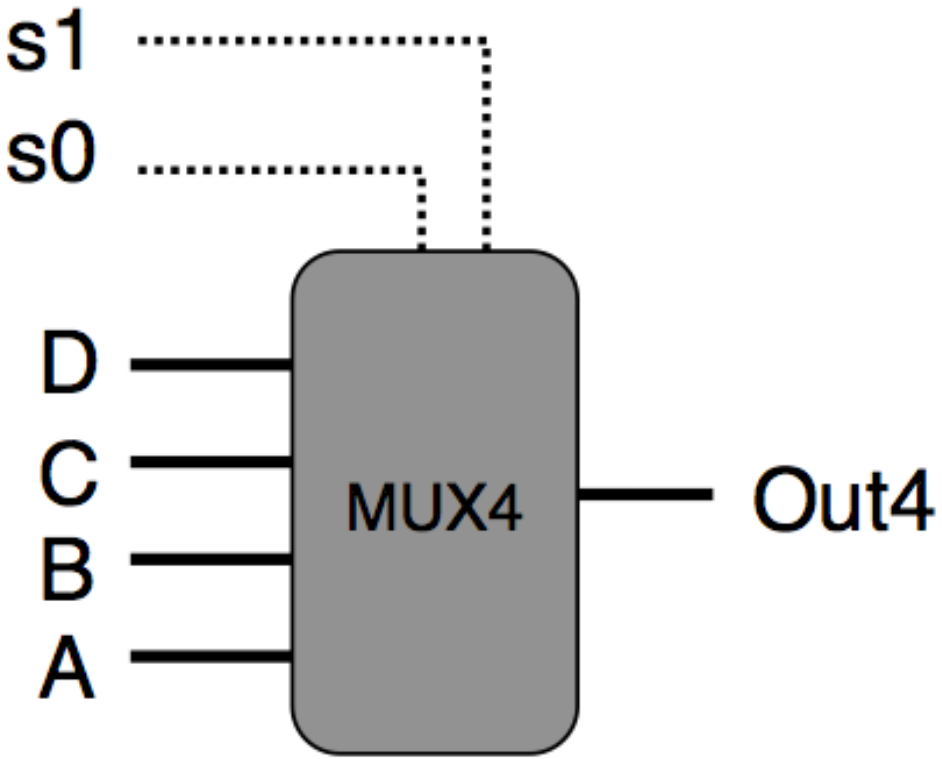
1.8. ábra. **Szó-szintű multiplexelő áramkör.** A kimenet értéke megegyezik az **A** bemenő szó értékével, amikor az **s** vezérlő jel **1** értékű, és a **B** értékével egyébként. A HCL nyelvben a multiplexereket **case** kifejezések írják le.

```
int out = [
        s: A;
        1: B;
];
```

Ebben a kódban a második választó kifejezés egyszerűen **1**, ami azt jelzi, hogy ezt az esetet kell választanunk, ha egyik korábbi sem volt jó. Ez az a módszer, amivel a HCL nyelven alapértelmezett (default) értéket lehet megadni. Csaknem minden **case** kifejezés ilyen módon végződik.

A nem-kizáró lehetőségek megengedése a HCL kódot olvashatóbbá teszi. Egy tényleges hardver multiplexernek kölcsönösen kizáró jelekkel kell vezérelni, melyik bemenő szót kell átadni a kimenetre; a 1.8 ábrán ilyen az **s** és **!s**. Hogy egy HCL **case** kifejezést hardverré fordítson, a logikai szintézis programnak analizálnia kellene a kiválasztó kifejezéseket, és feloldani a lehetséges konfliktusokat, hogy csak az első megfelelő esetet választhassuk.

A választó kifejezések tetszőleges logikai kifejezések lehetnek, és tetszőleges számú eset fordulhat elő. Ez lehetővé teszi, hogy a **case** kifejezések olyan eseteket is leírjanak, ahol sok választási lehetőség van, komplex kiválasztási feltétellel. Példaként tekintsük a 1.9 ábrán látható 4-utas multiplexert. Ez az áramkör az **A**, **B**, **C** és **D** bemeneti szavak



1.9. ábra. **Négy-utas multiplexer.** A **s1** és **s0** jelen különböző kombinációi határozzák meg, melyik adat kerül át a kimenetre.

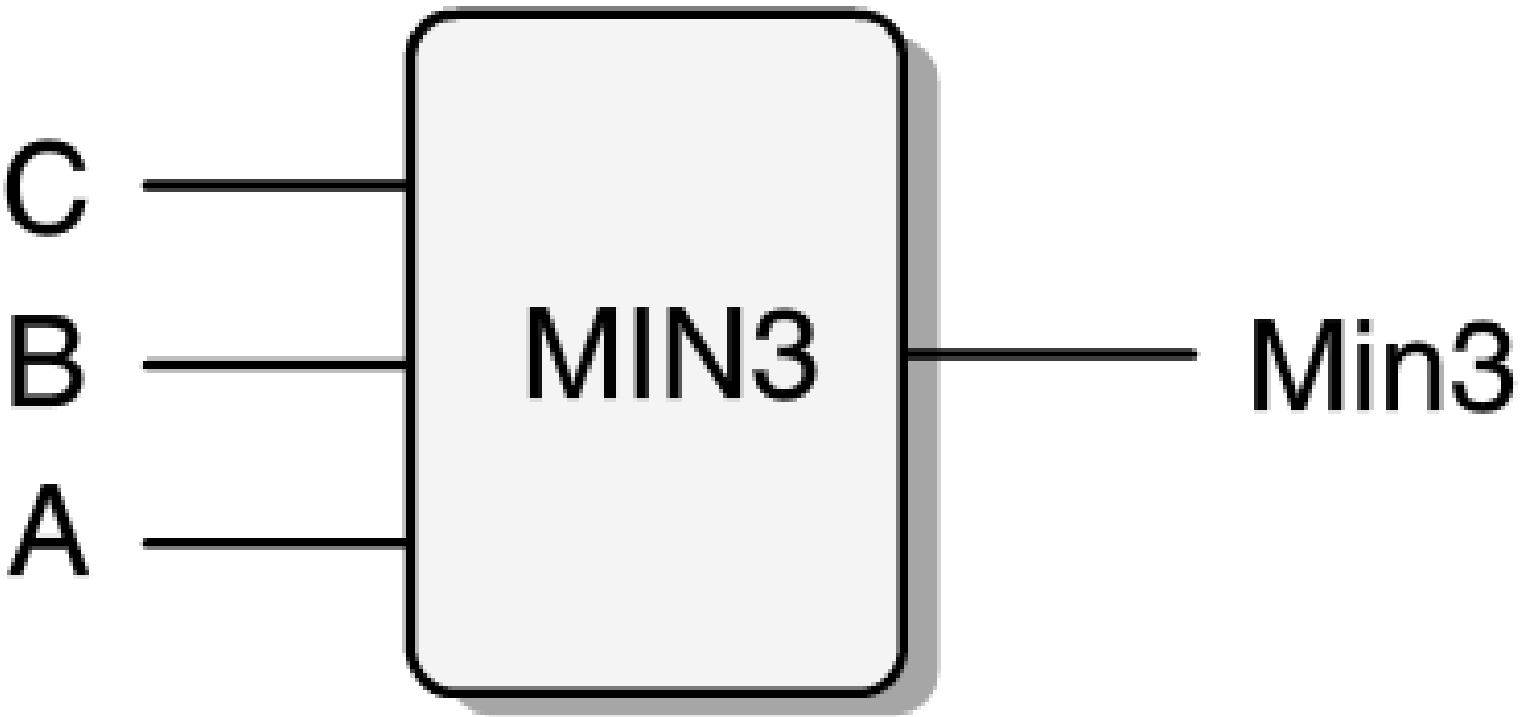
közül választ az **s0** és **s1** vezérlőjelek alapján, azokat egy két-bites bináris értékként kezelve.

A HCL nyelven ezt olyan logikai kifejezéssel írhatjuk le, amelyik a vezérlő bitek kombinációját használja:

```
int Out4 = [
        !s1 && !s0  : A; # 00
        !s1         : B; # 01
        !s0         : C; # 10
        1           : D; # 11
];
```

A jobb oldali megjegyzések (a # jellel kezdődő szöveg a sor végéig megjegyzés) mutatja, melyik **s1** és **s0** kombináció hatására választódik ki az illető eset. Vegyük észre, hogy néha a kiválasztó kifejezés egyszerűsíthető, mivel mindig az első megfelelő esetet választjuk. Például, a második kifejezés **!s1**, a teljesebb **!s1 && s0** helyett, mivel az egyetlen további lehetőség, hogy **s1** értéke **0**, azt pedig az első választó kifejezésként használtuk. Hasonló módon, a harmadik kifejezés **!s0** lehet, a negyedik pedig egyszerűen **1**.

Utolsó példaként, tegyük fel, hogy olyan áramkört akarunk tervezni, amelyik megtalálja a legkisebb értéket az **A**, **B** és **C** szavak közül, lásd 1.10 ábra. Az ennek megfelelő

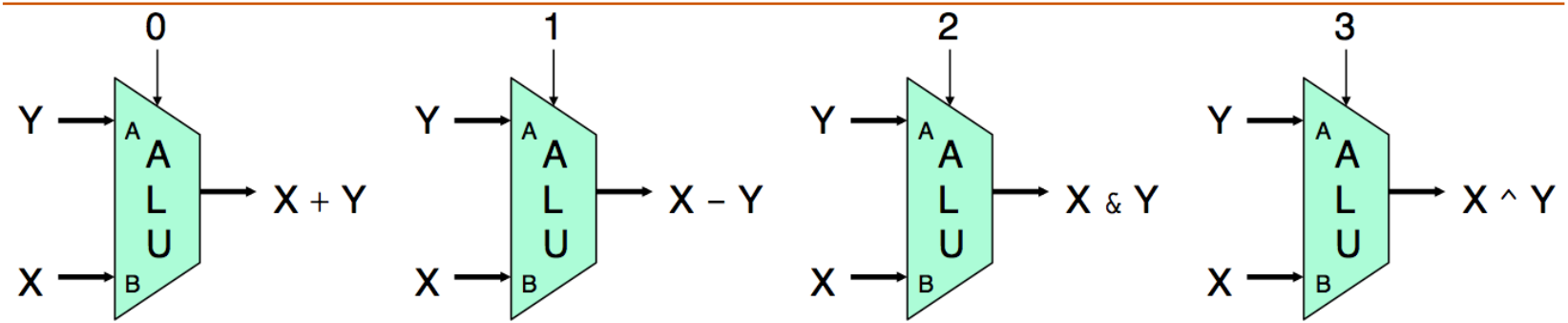


1.10. ábra. **A legkisebb érték megtalálása.**

HCL kifejezés:

```
int Min3 = [
        A <= B && A <=C  : A;
        B <= A && B <=C  : B;
        1                : C;
];
```

Kombinációs logikai áramkörökkel nagyon sokféle műveletet végezhetünk szó-szintű adatokon. Ennek a részletes tárgyalása meghaladja a tananyag kereteit. Az egyik fontos kombinációs áramkör, amit aritmetikai és logikai egységként (arithmetic/logic unit, ALU) ismerünk, a 1.11 ábrán látható. Az áramkörnek három bemenete van: az **A** és **B** címkéjű adat bemenet, valamint egy vezérlő bemenet. Az utóbbi értékétől függően az áramkör különböző aritmetikai vagy logikai műveletet végez az adat bemeneteken kapott értékekkel. Vegyük észre, hogy az ALU-nál feltüntetett négy művelet megfelel az Y86 utasításkészlete által támogatott négy egész típusú műveletnek, és a vezérlő kód értéke is megegyezik ezen utasítások funkció kódjával, lásd 1.3 ábra. Vegyük észre az operandusok elrendezését kivonáshoz: az **A** bemenet értékét vonjuk ki a **B** bemenet értékéből. Ezt az elrendezést a `subl` utasítás argumentumainak sorrendje alapján választottuk.



1.11. ábra. **Aritmetikai és logikai egység (ALU).** A funkció választó bemenet értékétől függően, az áramkör a négy különböző aritmetikai és logikai művelet egyikét végzi el.

1.2.4. Halmazban tagság

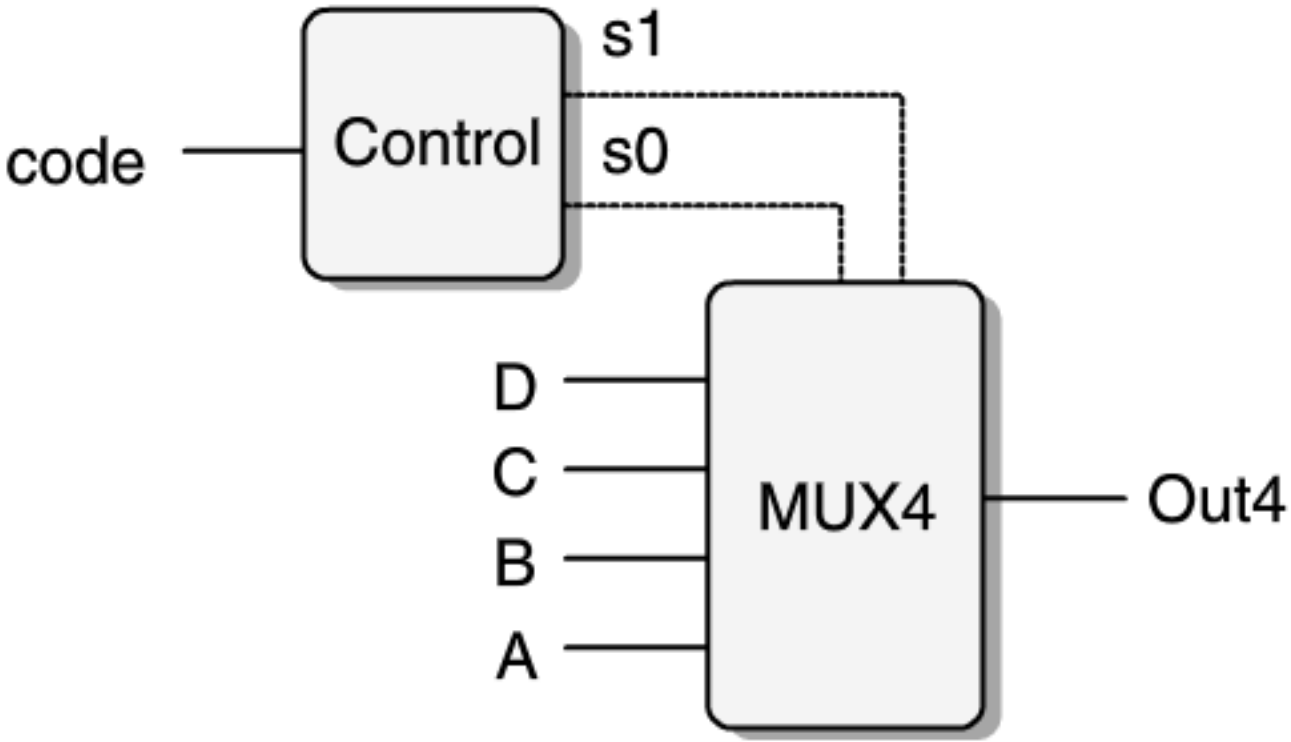
Processzor tervezés során sok olyan példával találkozunk, amikor egy jelet több jellel kell összehasonlítani, azaz például, hogy az éppen végrehajtott utasítás benne van-e az utasításkódok valamely csoportjában. Egyszerű példaként tegyük fel, hogy a 1.12 ábrán látható négyutas multiplexer számára akarjuk előállítani az **s1** és **s0** jeleket egy két bites kód alsó és felső bitjének megfelelő kiválasztásával.

Ebben az áramkörben a 2-bites jelkód vezérelné a négy adat szó (**A**, **B**, **C** és **D**) közötti választást. Az **s1** és **s0** jelek előállítását a lehetséges kód értékek egyenlőségének vizsgálata alapján állíthatjuk elő:

```
bool s1 = code == 2 || code == 3;
bool s0 = code == 1 || code == 3;
```

Egy tömörebb jelöléssel így írhatjuk le, hogy **s1** értéke **1** amikor **code** benne van a **{2,3}** halmazban, és **s0** értéke **1** amikor **code** benne van az **{1,3}** halmazban:

```
bool s1 = code in { 2, 3 };
bool s0 = code in { 1, 3 };
```



1.12. ábra. **Halmaz tagság meghatározás.**

A "halmaz tagja" vizsgálat általános formája

$iexpr$ in $iexpr_1, iexpr_2, \ldots, iexpr_k$

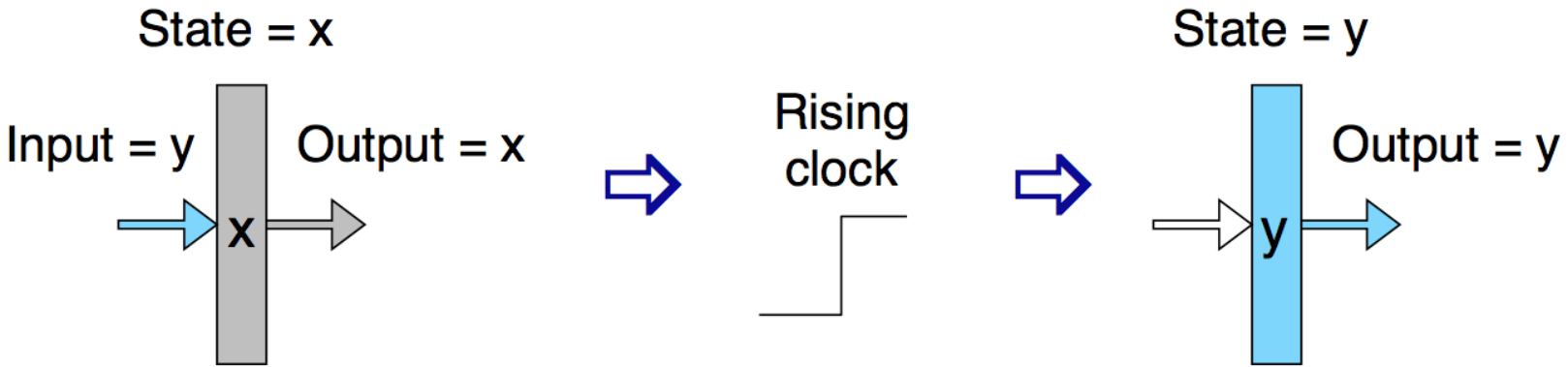
ahol az $iexpr$ vizsgált kifejezés megegyezik az $iexpr_1, iexpr_2, \ldots, iexpr_k$ jelöltek valamelyikével. (mindegyik kifejezés egész típusú).

1.2.5. Memória és órajelek

A kombinációs regiszterek, természetükből kifolyólag, nem tárolnak információt. Helyette, egyszerűen reagálnak a bemenetükre adott jelekre, és a bemenetek függvényében kimenő jeleket generálnak. Hogy szekvenciális áramköröket hozzunk létre, azaz olyan rendszereket, amelyeknek állapota van és azon az állapoton műveleteket tudnak végezni, olyan eszközöket kell bevezetni, amelyek képesek bitként ábrázolt információt tárolni. Ezeket a tárolóeszközöket egy órajellel (ami olyan periodikus jel, ami azt határozza meg, hogy mikor kell ezekbe az eszközökbe új értéket beírni) vezéreljük. Kétfajta ilyen memória eszközt vizsgálunk:

- (Órajel vezérelt) regiszter, ami egyes biteket vagy szavakat tárol. Az órajel vezérli a regiszter bemenetén levő érték betöltését.
- (Véletlen hozzáférésű) memória, ami több, címmel azonosított szót tárol; írni és olvasni is lehet. A véletlen hozzáférésű memóriákhoz tartozik pl. (1) a processzor virtuális memória rendszere, ahol hardver és operációs rendszer kombinációja kelti azt a látszatot a processzor számára, hogy egy nagy címtér bármely elemét el tudja érni; (2) a regiszter tömb, ahol a regiszter azonosítók szolgálnak címként.

Egy IA32 vagy Y86 processzorban a regiszter tömbben nyolc program regiszter található



1.13. ábra. **Regiszter művelet.** A regiszter kimenetei mindaddig őrzik a korábbi állapotot, amíg meg nem érkezik az órajel felfutó éle. Ekkor a regiszter átveszi a bemeneteken levő értéket és ettől kezdve ez lesz az új regiszter állapot.

Mint láthatjuk, a "regiszter" szó két különböző dolgot jelent, amikor hardverről vagy gépi kódú programozásról beszélünk. Hardverként, egy regiszter kimeneteivel és bemeneteivel közvetlenül kapcsolódik az áramkör többi részéhez. A gépi kódú

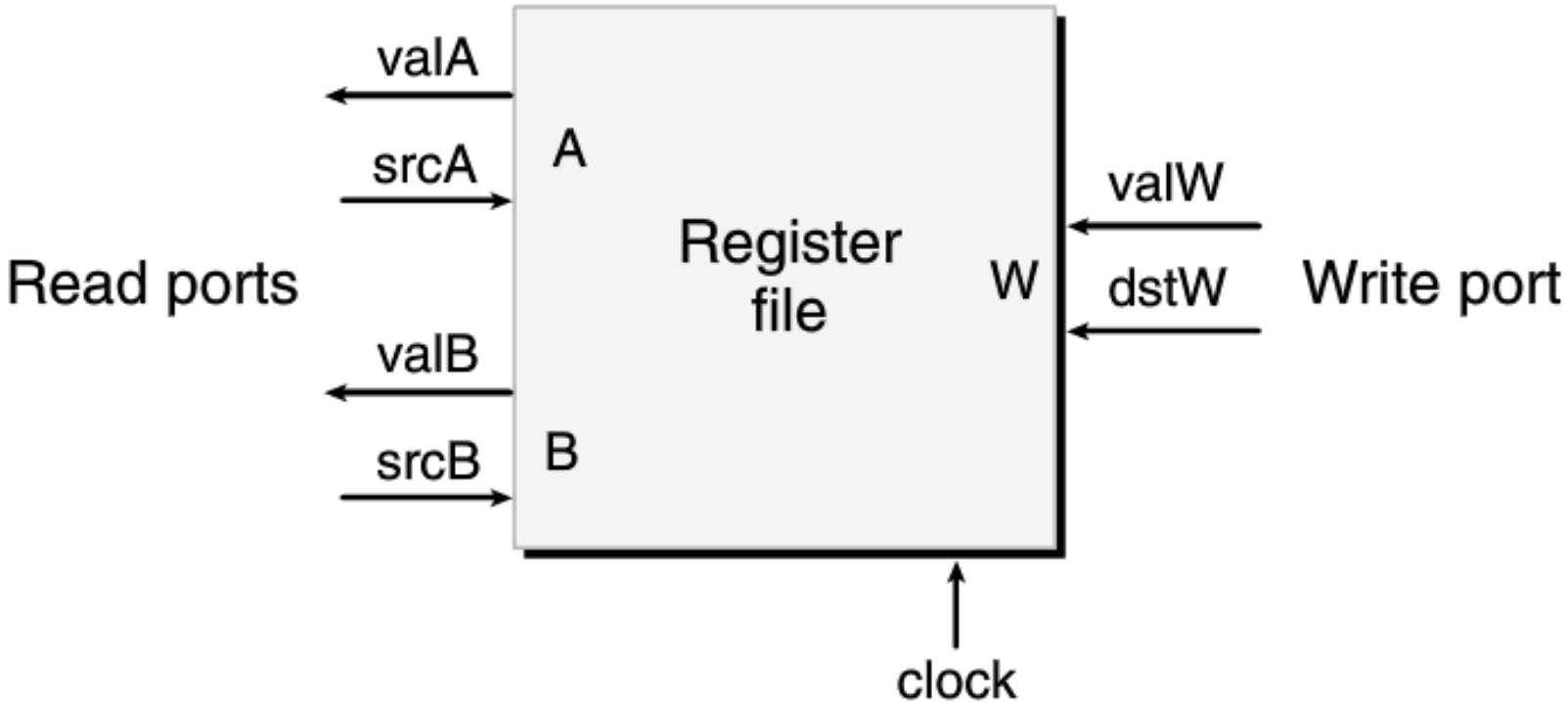
programozás esetén a regiszterek címezhető szavak CPU-n belüli kis gyűjteményét jelentik, ahol a címek a regiszter azonosítók. Ezeket a szavakat általában a regiszter tömbben tároljuk, bár látni fogjuk, hogy a hardver néha közvetlenül ad át egy szót az egyik és egy másik utasítás között, hogy kikerülje azt a késleltetést, amit az okozna, ha először írni, majd olvasni kellene a regiszter tömböt. Amikor feltétlenül szükséges megkülönböztetni, a két osztályt "hardver regiszter" és "program regiszter" névvel illetjük.

A 1.13 ábra részletesebben mutatja, hogyan működik egy regiszter. Az idő nagy részében a regiszter rögzített állapotban van (az ábrán ezt **x** mutatja), és pillanatnyi állapotának megfelelő kimeneti jelet generál. A jelek a regisztert megelőző kombinációs logikán át terjednek, egy új értéket hoznak létre a regiszter bemenetén (az ábrán ezt **y** mutatja), a regiszter kimenete viszont változatlan marad, amíg az órajel alacsony értékű. Amint az órajel felfut, a bemenő jelek betöltődnek a regiszterbe, mint annak következő (**y**) állapota, és ez lesz a regiszter új állapota, amíg a következő órajel meg nem érkezik. A regiszterek egyfajta sorompóként szolgálnak a kombinációs áramkörök között az áramkör különböző részein. Az érték a regiszter bemenetéről annak kimenetére minden órajel alatt csak egyszer, a felfutó élre kerül át. A mi

Y86 processzoraink órajel-vezérelt regisztereket fognak használni a program számláló (program counter, PC), a feltétel kódok (condition codes , CC) és a program állapot (program status, Stat) tárolására.

Ennek a regiszter tömbnek két olvasható portja van, **A** és **B**, valamint egy írható **W** portja. Egy több portos véletlen hozzáférésű memória lehetővé teszi több írási és olvasási művelet egyidejű végrehajtását. Az ábrán bemutatott regiszter tömb két programregiszterét olvashatjuk és a harmadikat frissíthetjük, egyidejűleg. Mindegyik portnak van cím bemenete, ami jelzi, hogy melyik program regisztert kell kiválasztani, valamint egy adat kimenete vagy bemenete, ami a program regiszter értékéhez tartozik. A címek a regiszter azonosítók (lásd 1.1 táblázat). A két olvasható regiszter cím bemenete **srcA** és **srcB** (ami a“source A” és “source B” rövidítése). Az írható port cím bemenete **dstW** (ami a “destination W” rövidítése), valamint egy **valW** (ami a “value W” rövidítése).

A regiszter tömb nem kombinációs áramkör, mivel van belső tárolója. Ebben az implementációban azonban az adatot úgy olvashatjuk a regiszterből, mintha az egy kombinációs logikai blokk lenne, amelynek a címek a bemenetei és az adatok a kimenetei. Amikor **srcA** vagy **srcB** megkapja valamelyik regiszter azonosítóját, valamennyi

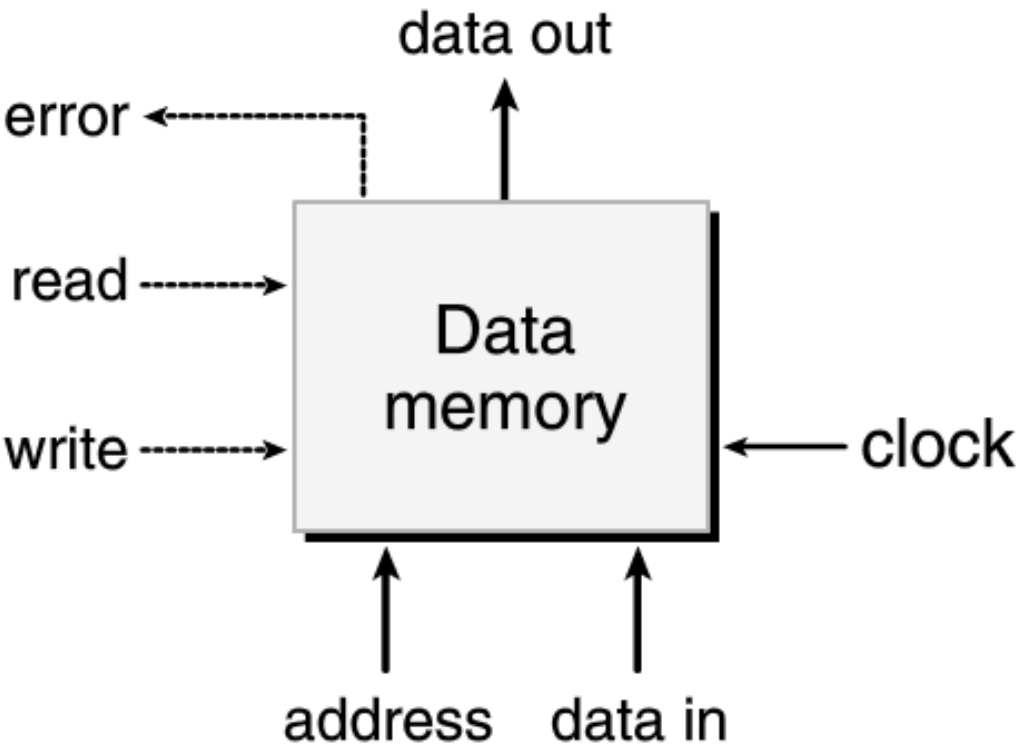


1.14. ábra. **A regiszter tömb működése.**

késés után a megfelelő program regiszterben tárolt érték megjelenik a **valA** vagy **valB** kimeneteken. Például, az **srcA** bemenetre a **3** értéket adva, az %ebx regiszter értéke olvasható ki, és ez jelenik meg a **valA** kimeneten.

A regiszter tömbbe írás úgy vezérlődik, hogy az órajel hatására egy érték beíródik az órajel vezérelt regiszterbe. Amikor egy órajel felfutó éle megérkezik, a **valW** bemeneten levő érték beíródik abba a program regiszterbe, amelyiknek az azonosítója a **dstW** bemeneten található. Amikor a **dstW** értéke a **0xF** különleges értékre van állítva, semmilyen program regiszterbe nem történik írás. Mivel a regiszter tömböt írni és olvasni egyaránt tudjuk, természetes a kérdés: "Mi történik, ha egyidejűleg próbáljuk meg írni és olvasni ugyanazt a regisztert?". Azonnal meg is adhatjuk a választ: amikor egy regisztert úgy frissítünk, hogy ugyanaz a regiszter azonosító van az az olvasható porton, megfigyelhetjük a régi értékről az újra való átmenetet. Amikor majd a regisztert processzor tervünkben használjuk, ezt a tulajdonságot figyelembe kell venni.

Ennek a memóriának egyetlen cím bemenete van, egy adat bemenete íráshoz, és egy adat kimenete olvasáshoz. A regiszter fájlhoz hasonlóan, a memóriából olvasás a kombinációs logikához hasonlóan működik: ha egy címet teszünk az **address** bemenetre és a **write** vonalra **0**-t írunk, akkor – valamekkora késleltetés után – az ezen a címen



1.15. ábra. **RAM memória működése.**

tárolt adat megjelenik a **data out** kimeneten. Az **error** jel **1** értékű lesz, ha a cím az értéktartományon kívül van, különben **0**. A memóriába írást a **clock** órajel vezérli: beállítjuk a kívánt címet az **address**, az adatot a **data in** vonalra, és a **write** vonalat pedig **1** értékűre. Amikor ezután megérkezik az órajel, a memóriában a megadott helyen a tartalom frissül, feltéve, hogy a cím érvényes. A **read** művelethez hasonlóan, az **error** jel **1** értékű lesz, ha a cím érvénytelen. Ezt a jelzést egy kombinációs logika állítja elő, mivel a megkívánt határ vizsgálat csupán a bemenő cím függvénye, nem tartalmaz valamiféle belső állapotot.

Processzorunkban csak olvasható memória (read-only memory, ROM) is található, utasítások beolvasására. A legtöbb valódi rendszerben ezek a memóriák egyetlen, két-portos memória rendszerbe illeszkednek: egyiken olvassuk az utasításokat, a másikon írjuk és olvassuk az adatokat.

1.3. Az Y86 soros megvalósítása

Most már az összes komponenssel rendelkezünk, ami az Y86 processzor megvalósításához szükséges. Első lépésként írjunk le egy SEQ-nak nevezett (szekvenciális) processzort. A SEQ minden órajel hatására elvégzi azokat a lépéseket, amelyek egy teljes utasítás végrehajtásához szükségesek. Azonban ehhez meglehetősen hosszú ciklusidő szükséges, és így az elérhető órajel sebesség elfogadhatatlanul kicsi lenne. A SEQ fejlesztésével az a célunk, hogy megtegyük az első lépést végső célunk, egy hatékony, futószalagos processzor megvalósításának irányába.

1.3.1. A végrehajtás szakaszokra bontása

Általánosságban, egy utasítás végrehajtása számos műveletből áll. Ezeket úgy próbáljuk meg bizonyos szakaszokra bontani, hogy valamennyi utasítás végrehajtása ugyanazt a mintát kövesse, bár az egyes utasítások hatása nagyon is különböző. Az egyes lépéseknél a végrehajtás részletei függenek attól, hogy milyen utasítást hajtunk végre. Az említett szerkezet létrehozásával lehetővé tesszük, hogy olyan processzort tervezzünk, amelyik a lehető legjobban kihasználja a rendelkezésre álló hardvert. Az egyes szakaszok és az azokban végrehajtott műveletek informális leírása:

- ***Utasítás elővétel, Fetch***
  Ebben az állapotban a processzor beolvassa az utasítás bájtjait a memóriából, a program számláló (program counter, PC) értékét használva memória címként. Az utasításból kiválasztja a két 4-bites részt, amelyekre **icode** (instruction code) és **ifun** (intruction function) névvel hivatkozunk. Esetlegesen egy regiszter kijelölő bájtot is elővesz, amely egy vagy akár két regiszter operandust (**rA** és **rB**) is kijelöl. Az is lehet, hogy egy 4-bájtos **valC** konstans értéket is elővesz. Kiszámítja azt a **valP** értéket, amelyik a sorban következő utasítás címe lesz (azaz, **valP** értéke a **PC**

értéke, plusz az elővett utasítás hossza).

- ***Dekódolás, Decode***
  Ebben a szakaszban a processzor beolvassa a két operandust a regiszter tömbből, ami megadja **valA** és **valB** értékét. Jellemzően az **rA** és **rB** utasítás mezőkben megjelölt regisztereket olvassa be, de bizonyos utasítások esetén az `%esp` regisztert is.
- ***Végrehajtás, Execute***
  Ebben a szakaszban az aritmetikai és logikai egység (ALU) vagy elvégzi az utasítás (az **ifun** értéke) által meghatározott műveletet, kiszámolja a memória hivatkozás tényleges címét, vagy pedig csökkenti/növeli a veremmutatót. Az eredményül kapott értékre **valE** néven hivatkozunk. Esetlegesen a feltétel jelzőbiteket is beállítja. Ugró utasítás (`jump`) esetén kiszámítja a feltétel kódokat és (az **ifun** által megadott) elágazási feltételeket, hogy megállapítsa, kell-e elágaztatni a programot.
- ***Memória, Memory***
  Ebben a szakaszban adatot írhat a memóriába, vagy olvashat onnét. A beolvasott értékre **valM** néven hivatkozunk.
- ***Visszaírás, Write back***

A visszaírási állapotban a két eredményt visszaírja a regiszter tömbbe.

- ***PC frissítés, PC update***

  A programszámláló (**PC**) a következő utasításra áll.

A processzor folyamatosan ebben az utasítás végrehajtási hurokban mozog. Az általunk megvalósítani kívánt egyszerű processzor akkor áll meg, amikor kivétel fordul elő: végrehajt egy `halt` utasítást vagy érvénytelen utasítást talál, esetleg érvénytelen címről próbál adatot olvasni vagy oda beírni. Egy teljesebb megvalósítás esetén a processzor kivétel-kezelő módba kerülne, és elkezdené megvalósítani a kivétel típusa által meghatározott speciális kivétel kezelő kódot.

Mint azt az eddigi leírásból láttuk, meghökkentő mennyiségű művelet szükséges egyetlen utasítás végrehajtásához is. Nem csak az utasítás által meghatározott műveletet kell végrehajtani, hanem címeket kiszámítani, frissíteni a verem mutatót, és meghatározni a következő utasítás címét. Szerencsére, a végrehajtás általános menete minden utasítás esetén hasonló. Nagyon egyszerű és egyforma struktúra használata nagyon fontos, amikor hardvert tervezünk, mivel minimalizálni akarjuk a hardver teljes mennyiségét, hiszen végső soron azt az integrált áramköri lapka kétdimenziós felületére kell leképeznünk. A bonyolultság csökkentésének egyik útja, hogy

a különböző utasítások a lehető legnagyobb mennyiségű hardvert közösen használják. Például, processzoraink mindegyike egyetlen ALUt tartalmaz, amit az utasítás típusától függően különböző módokon használ. Hardver blokkokat két példányban használni sokkal költségesebb, mint szoftverben több kópiával rendelkezni. Hasonlóképpen, sokkal nehezebb hardverben foglalkozni speciális esetekkel és furcsaságokkal, mint szoftverben.

Feladatunk, hogy a különböző utasítások végrehajtásához szükséges számításokat ebbe a keretbe beillesszük. Ennek bemutatására a 1.4 programlistán bemutatott kódot fogjuk használni. A 1.3-1.6 táblázatok azt írják le, hogyan haladnak át a különböző Y86 utasítások az egyes szakaszokon. Érdemes ezeket a táblázatokat figyelmesen tanulmányozni. Ezek formája lehetővé teszi, hogy közvetlenül hardverre képezzük le az utasításokat. A táblázatok egyes sorai valamely jelzés vagy tárolt állapot értékadását írják le (a ← értékadó operátorral). Ezeket úgy olvassuk, mintha azokat felülről lefelé haladva értékeltük volna. Később, amikor a számításokat hardverre képezzük le, látni fogjuk, hogy ezeket a számításokat nem kell szigorú sorrendben elvégezni.

A 1.3 táblázat azokat a feldolgozási lépéseket mutatja, amelyeket az `rrmovl` (register-register move) és `irmovl` (immediate-register move) `OPl` típusú utasítások (egész

Programlista 1.4: **Egy Y86 minta-utasítás sorozat.** Ezzel követjük nyomon az utasítás végrehajtását a különböző szakaszokban.

```
0x000:  30f209000000  |      irmovl  $9,  %edx
0x006:  30f315000000  |      irmovl  $21,  %ebx
0x00c:  6123          |      subl  %edx,  %ebx #  subtract
0x00e:  30f480000000  |      irmovl  $128,%esp #  Problem  11
0x014:  404364000000  |      rmmovl  %esp,100(%ebx)  #  store
0x01a:  a02f          |      pushl  %edx       #  push
0x01c:  b00f          |      popl  %eax        #  Problem  12
0x01e:  7328000000    |      je  done          #  Not  taken
0x023:  8029000000    |      call  proc        #  Problem  16
0x028:                |  done:
0x028:  00            |       halt
0x029:                |  proc:
0x029:  90            |       ret              #  Return
```

típusú és logikai műveletek) esetében kell elvégeznünk. A 1.2 ábrából azt látjuk, hogy jól választottuk meg az utasítások kódolását: a négy egész típusú műveletben (`addl`, `subl`, `andl` és `xorl`) az **icode** értéke ugyanaz. Ezek mindegyikét ugyanazzal a lépés sorozattal kezelhetjük, kivéve, hogy az ALU-t az **ifun**-ba kódolt egyedi utasítás műveletnek megfelelően kell beállítanunk.

Egy egész típusú műveletet megvalósító utasítás végrehajtása a fenti általános mintát követi. Az utasítás elővételi (**fetch**) szakaszban nincs szükségünk egy konstans szóra,

ezért **valP** értéke **PC + 2** lesz. A dekódolási (**decode**) szakaszban beolvassuk mindkét operandust. Az **execute** fázisban ezeket, az **ifun** funkció választóval együtt, az ALU rendelkezésére bocsátjuk, úgy hogy az utasítás eredménye **valE** lesz. Ezt a számítást mutatja a **valB OP valA** kifejezés, ahol **OP** az **ifun** által kijelölt műveletet jelenti. Jegyezzük meg a két argumentum sorrendjét – ezt következetesen használja az Y86 (és az IA32). Például, a `subl %eax,%edx` utasítás az **R[%edx] - R[%eax]** értéket számolja ki. Ezekkel az utasításokkal semmi nem történik a **memory** szakaszban, majd **valE** íródik be az **rB** regiszterbe a **write-back** szakaszban, és az utasítás befejeződéseként a **PC** felveszi a **valP** értéket. Egy `rrmovl` utasítás végrehajtása nagyon hasonlít egy aritmetikai műveletéhez. Azonban, nem kell elővennünk a második regiszter operandust. Helyette, a második ALU bemenetet nullára állítjuk és azt hozzáadjuk az elsőhöz, aminek eredménye **valE = valA**, majd ezt beírjuk a regiszter tömbbe. Hasonló a végrehajtás `irmovl` utasítás esetén is, kivéve, hogy a **valC** konstans értéket adjuk az első ALU bemenetre. Ezen túlmenően, a program számlálót 6-tal kell megnövelnünk `irmovl` esetén, a hosszabb utasítás formátum miatt. Ezen utasítások egyike sem változtatja meg a feltétel kódokat.

A 1.4 táblázatban láthatjuk a memóriát író és olvasó `rmmovl` és `mrmovl` utasítások

1.3. táblázat. **Az Y86 szekvenciális megvalósításában az `OPl`, `rrmovl`, és `irmovl` utasítások során végzett számítások.** Ezek az utasítások kiszámítanak egy értéket és az eredményt egy regiszterben tárolják. Jelölések: **icode : ifun** jelzi az utasításkód bájt, **rA : rB** a regiszter kijelölő bájt két komponensét. Az **M1[x]** jelölés 1 bájt elérését jelöli a memória **x** helyén, **M4[x]** pedig 4 bájtét.

Stage	**OP1 rA, rB**	**rrmovl rA, rB**	**irmovl V, rB**
Fetch	icode : ifun $\leftarrow M_1$[PC]	icode : ifun $\leftarrow M_1$[PC]	icode : ifun $\leftarrow M_1$[PC]
	rA:rB $\leftarrow M_1$[PC+1]	rA:rB $\leftarrow M_1$[PC+1]	rA:rB $\leftarrow M_1$[PC+1]
			valC $\leftarrow M_4$[PC+2]
	valP $\leftarrow$ PC+2	valP $\leftarrow$ PC+2	valP $\leftarrow$ PC+6
Decode	valA $\leftarrow$ R[rA]	valA $\leftarrow$ R[rA]	
	valB $\leftarrow$ R[rB]		
Execute	valE $\leftarrow$ valB OP valA	valE $\leftarrow$ 0 + valA	valE $\leftarrow$ 0 + valC
	Set CC		
Memory			
Write back	R[rB] $\leftarrow$ ValE	R[rB] $\leftarrow$ ValE	R[rB] $\leftarrow$ ValE
PC update	PC $\leftarrow$ ValP	PC $\leftarrow$ ValP	PC $\leftarrow$ ValP

végrehajtását. Az előbbihez hasonló alap folyamot látunk, csak az ALUt is használjuk, hogy **valC**-t hozzáadjuk **valB]**-hez, aminek eredményeként kapjuk a tényleges címet (az eltolási érték és az alap regiszter érték összegeként) a memória művelethez. A **Memory** szakaszban vagy a **valA** regiszter értéket írjuk a memóriába, vagy a memóriából beolvassuk a **valM** értéket.

A 1.5 táblázat tartalmazza a `pushl` és `popl` utasítások végrehajtásához szükséges lépéseket. Ezek az utasítások az Y86 legnehezebben megvalósítható utasításai közé tartoznak, mivel tartalmaznak memória elérést és verem mutató csökkentést vagy növelést is. Bár a két utasítás végrehajtásának menete hasonló, vannak jelentős különbségek is. A `pushl` utasítás az eddig megismert utasításokhoz hasonlóan kezdődik, de a **Decode** szakaszban az `%esp`-t használjuk a második regiszter operandus azonosítójaként, aminek eredményeként **valB** a veremmutató értékét veszi fel. Az **Execute** szakaszban az ALU-t használjuk arra, hogy a verem mutatót 4-gyel csökkentsük. Ezt a csökkentett értéket használjuk a memória íráshoz címként és a **Write back** szakaszban visszaírjuk értékét `%esp`-be. Azzal, hogy az írási művelet címeként **valE**-t használjuk, alkalmazkodunk az Y86 (és az IA32) szokásos módszeréhez, hogy a `pushl` utasításnak írás előtt kell csökkentenie a verem mutatót, még akkor is, ha a verem mutató tényleges

1.4. táblázat. **Az Y86 processzor soros implementációjában az `rmmovl`, és `mrmovl` utasításainak kiszámítása.** Ezek az utasítások írják és olvassák a memóriát.

Stage	**rmmovl rA, D(rB)**	**mrmovl D(rB),rA**
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]
	valC ← M_4[PC+2]	valC ← M_4[PC+2]
	valP ← PC+6	valP ← PC+6
Decode	valA ← R[rA]	
	valB ← R[rB]	valB ← R[rB]
Execute	valE ← valB + valC	valE ← calA+ valC
Memory	M_4[valE] ← valA	valM ← M_4[valE]
Write back		
		R[rA] ← valM
PC update	PC ← ValP	PC ← ValP

frissítése nem történik meg a memória művelet befejeződéséig.

1.5. táblázat. **Az Y86 processzor soros implementációjában az `pushl` , és `popl` utasításainak kiszámítása. Ezek az utasítások kezelik a verem memóriát.**

Stage	**pushl rA**	**popl rA**
Fetch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]
	valP ←PC+2	valP ←PC+2
Decode	valA ← R[rA]	valA ← R[%esp]
	valB ← R[%esp]	valB ← R[%esp]
Execute	valE ← valB + (-4)	valE ← calA+ (-4)]
Memory	M_4[valE] ← valA	valM ← M_4[valA]
Write back	R[%esp] ← valE	R[%esp] ← valE
		R[rA] ← valM
PC update	PC ← valP	PC ← valP

A `popl` utasítás végrehajtása nagyon hasonlít a `pushl` végrehajtására, kivéve, hogy

a **decode** szakaszban a veremmutató két másolatát olvassuk be. Ez nyilvánvalóan redundáns, de látni fogjuk, hogy a verem mutatót **valA** és **valB** értékként is használva az ezután következő utasítás végrehajtást a többi utasításéhoz hasonlóbbá teszi, a terv egyformaságát jelentősen javítva. Az ALU-t használjuk arra, hogy a verem mutatót az **Execute** szakaszban a **4** értékkel megnöveljük, de a memória műveletben a növelés előtti értéket használjuk címként. A **write-back** szakaszban frissítjük mind a verem mutató regisztert a megnövelt értékkel, mind az **rA** regisztert a memóriából beolvasott értékkel. A növelés nélküli verem mutatót használva memória címként megmaradunk annál az Y86 (és IA32) hagyománynál, hogy a `popl` utasításnak előbb kell olvasni a memóriát és csak azután növelni a verem mutatót.

A 1.6 táblázat mutatja három vezérlés átadó utasításunk, a `jump` utasítások, `call` és `ret` végrehajtását. Azt látjuk, hogy ezeket az utasításokat is az eddig megismert általános feldolgozási módszerrel kezelhetjük.

Mint az egész típusú műveletek esetén is, valamennyi ugró utasítást egységes módon kezelhetünk, mivel azok csak abban különböznek, hogy kell vagy nem kell elágazni. Egy ugró utasítás ugyanúgy megy át a **Fetch** és **Decode** szakaszokon, mint az előző utasítások, kivéve, hogy nincs szüksége regiszter kijelölő bájtra. Az **Execute** szakaszban

megvizsgáljuk a feltétel kódokat és az ugrási feltételt annak meghatározására, hogy kell-e elágaztatni; ennek eredménye egy 1-bites **Cnd** jel. A **PC update** szakaszban megvizsgáljuk ezt a jelzőbitet, és a PC-t vagy a **valC** értékre (az ugrás célpontja) állítjuk, ha a jelzőbit **1** értékű, vagy **valP** értékre (a következő utasítás címe) ha a jelzőbit **0** értékű. A **x ? a : b** jelölés hasonló a C feltételes kifejezéséhez: annak eredménye **a** ha **x** nem-nulla, és **b** amikor **x** nulla.

A `call` és `ret` utasítások hasonlóságot mutatnak a `pushl` és `popl` utasításokkal, kivéve, hogy most program számláló értéket teszünk a verembe vagy veszünk ki onnét. A `call` utasítással a **valP** értéket, a `call` utasítást követő utasítás címét írjuk a verembe. A **PC update** szakaszban a PC értékét **valC**-re állítjuk, ami a hívás cél címe. A `ret` utasítással a veremből kivett **valM** értéket írjuk a PC-be a **PC update** szakaszban.

Ezzel létrehoztunk egy egységes szerkezetet, amely valamennyi Y86 utasítást kezeli. Bár ezek az utasítások nagyon különböző karakterűek, végrehajtásukat egységesen hat szakaszra tudjuk osztani. Most már tervezhetünk olyan hardvert, amely megvalósítja ezeket a fázisokat és azokat egymással összekapcsolja.

1.6. táblázat. **Az Y86 soros megvalósításában a `jXX`, `call`, és `ret` utasítások során végzett számítások.** Ezek az utasítások vezérlés átadással járnak.

Stage	**jXX Dest**	**call Dest**	**ret**
Fetch	icode:ifun ← M_1[PC]	icode:ifun← M_1[PC]	icode:ifun ← M_1[PC]
	valC ← M_4[PC+1]	valC ← M_1[PC+1]	
	valP ← PC + 5	valP ← PC + 5	valP ← PC + 1
Decode			valA ← R[%esp]
		valB ← R[%esp]	valB ← R[%esp]
Execute		valE ← valB+ (-4)	valE ← valB + 4
	Cnd ← Cond(CC,ifun)		
Memory		M_4[valE] ← valP	valM ← M_4[valA]
Write back		R[%esp] ← ValE	R[%esp] ← ValE
PC update	PC ← Cnd ? valC : valP	PC ← valC	PC ← ValM

Példa: A `subl` utasítás végrehajtása

Példa gyanánt kövessük végig a 1.4 minta-program 3. sorában levő `subl` utasítás végrehajtását. Azt láthatjuk, hogy az előző két utasítás az `%edx` és `%ebx` regisztereket 9 illetve 21 értékre állítja. Azt is látjuk, hogy az utasítás a **0x00c** címen található és két bájtból áll, amelyek értéke **0x61** és **0x23**. Az utasítás végrehajtása az egyes szakaszokban az alábbi táblázatnak megfelelően zajlik. A táblázatban bal oldalt látjuk az `OPl` utasítás generikus szabályait (lásd 1.3 táblázat), az erre az esetre vonatkozó számításokat pedig a jobb oldalon.

Stage	Generic **OP1 rA, rB**	Specific **rrmovl%edx, %ebx**
Fetch	icode : ifun ← M_1[PC]	icode : ifun ← M_1[0x00C]=6:1
	rA:rB ← M_1[PC+1]	rA:rB ← M_1[0c00d]=2:3
	valP ← PC+2	valP ← 0x00c+2=0x00e
Decode	valA ← R[rA]	valA ← R[%edx]=9
	valB ← R[rB]	valB ← R[%ebx]=21
Execute	valE ← valB OP valA	valE ← 21-9=12

Példa: A `rmmovl` utasítás végrehajtása

Kövessük most végig a 1.4 minta-program 5. sorában található `rmmovl` utasítás végrehajtását. Láthatjuk, hogy az előző utasítás a **128** kezdőértéket adta az `%esp` regiszternek, az `%ebx` regiszterben pedig még az a **12** érték található, amelyet a 3. sorban a `subl` utasítás számolt ki. Azt is látjuk, hogy az utasítás a **0x014** címen található és 6 bájt hosszú. Az első két bájt értéke **0x40** és **0x43**, az utolsó négy pedig a **0x00000064** (decimal 100) érték, fordított bájt sorrendben. A számítási szakaszok a következők:

Stage	Generic **rmmovl rA, D(rB)**	Specific **rrmovl%esp, 100(%ebx)**
Fetch	icode : ifun ← M_1[PC]	icode : ifun ← M_1[0x014]=4:0
	rA:rB ← M_1[PC+1]	rA:rB ← M_1[0x015]=4:3
	valC ← M_4[PC+2]	valC ← M_4[0x016]=100
	valP ← PC+6	valP ← 0x014+6=0x01a
Decode	valA ← R[rA]	valA ← R[%esp]=128
	valB ← R[rB]	valB ← R[%ebx]=12
Execute	valE ← valB + valC	valE ← 12+100=112

Példa: A `pushl` utasítás végrehajtása

Kövessük most végig a 1.4 minta-program 6. sorában található `pushl` utasítás végrehajtását. Most az `%edx` regiszterben a **9**, az `%esp` regiszterben a **128** érték található. Azt is láthatjuk, hogy az utasítás a **0x01a** címen található és 2 bájtból áll, amelynek értékei **0xa0** és **0x2f**. A végrehajtási szakaszok:

Stage	Generic **pushl rA**	Specific **pushl%edx**
Fetch	icode : ifun $\leftarrow M_1$[PC]	icode : ifun $\leftarrow M_1$[0x01a]=a:0
	rA:rB $\leftarrow M_1$[PC+1]	rA:rB $\leftarrow M_1$[0x01b]=2:8
	valP $\leftarrow$ PC+3	valP $\leftarrow$ 0x01a+2=0x01c
Decode	valA $\leftarrow$ R[rA]	valA $\leftarrow$ R[%edx]=9
	valB $\leftarrow$ R[%esp]	valB $\leftarrow$ R[%esp]=128
Execute	valE $\leftarrow$ valB + (-4)	valE $\leftarrow$ 128+(-4)=124
Memory	M_4[valE] $\leftarrow$ valA	M_4[124] $\leftarrow$ 9

Példa: A `je` utasítás végrehajtásának nyomon követése

Kövessük most végig a 1.4 minta-program 8. sorában található `je` utasítás végrehajtását. Valamennyi feltétel kódot nulla értékűre állította a `subl` utasítás (3. sor), ezért biztosan nem lesz elágazás. Az utasításkód a **0x01e** címen van és 5 bájtból áll. Az első bájt értéke **0x73**, a következő négy pedig az ugrás célpontjának (**0x00000028**) fordított bájtsorrendben megadott címe. A végrehajtási szakaszok:

Stage	Generic **jXX Dest**	Specific **je 0x028**
Fetch	icode : ifun $\leftarrow M_1$[PC]	icode : ifun $\leftarrow M_1$[0x01e]=7:3
	valC $\leftarrow M_4$[PC+1]	valC $\leftarrow M_4$[0x01f] = 0x028
	valP $\leftarrow$ PC+5	valP $\leftarrow$ 0x01e+5=0x023
Decode		
Execute		
	Cnd $\leftarrow$ Cond(CC,ifun)	Cnd $\leftarrow$ (Cond(0,0,0),3)
Memory		

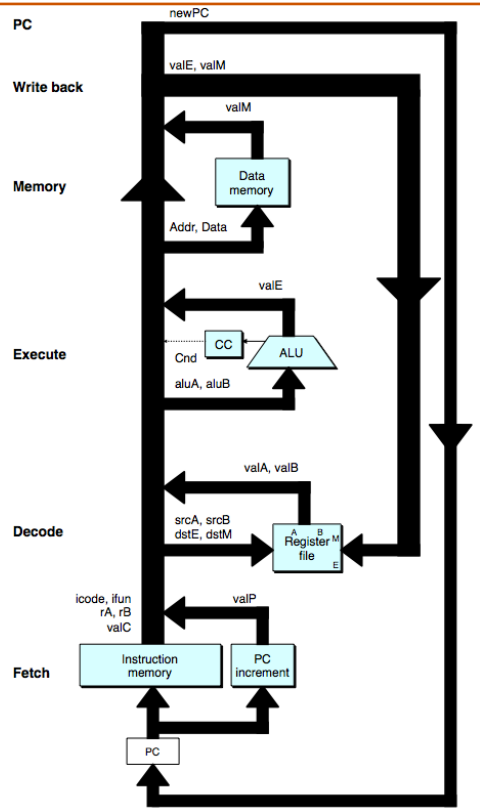
Példa: A `ret` utasítás végrehajtásának nyomon követése

Kövessük most végig a 1.4 minta-program 13. sorában található `ret` utasítás végrehajtását. Az utasítás címe **0x029** címen található és 1 bájtból áll, amelynek értéke **0x90**. Az előző `call` utasítás az `%esp` regisztert a **124** értékre állította és a **0x028** visszatérési címet beírta a **124** memória címre. A végrehajtási szakaszok:

Stage	Generic **ret**	Specific **ret**
Fetch	icode : ifun $\leftarrow M_1$[PC]	icode : ifun $\leftarrow M_1$[0x029]=9:0
	valP $\leftarrow$ PC+1	valP $\leftarrow$ 0x029+1=0x02a
Decode	valA $\leftarrow$ R[%esp]	valA $\leftarrow$ R[%esp]=124
	valB $\leftarrow$ R[%esp]	valB $\leftarrow$ R[%esp]=124
Execute	valE $\leftarrow$ valB + 4	valE $\leftarrow$ 124+4=128
Memory	valM $\leftarrow M_4$[valA]	valM $\leftarrow M_4$[124] = 0x28

1.3.2. A SEQ hardver szerkezete

Az Y86 utasítások megvalósításához szükséges számításokat hat szakaszból álló végrehajtási sorozatba tudtuk besorolni: **fetch**, **decode**, **execute**, **memory**, **write back**, és **PC update**. A 1.17 ábra mutatja egy olyan hardver szerkezet absztrakt képét, amely el tudja végezni ezeket a számításokat. A program számlálót egy regiszterben tároljuk, anelyet az ábra bal alsó sarkában "PC" jelöl. Az információ a (vastag fekete vonallal jelölt) kábeleken terjed, kezdetben felfelé, majd jobbra. A feldolgozás a különböző szakaszokhoz rendelt hardver egységekben történik. A visszacsatolási útvonalak az ábra jobb oldalán lefelé haladnak és regiszter tömb elemei és a program számláló frissítésére szolgáló értékeket tartalmazzák. Mit tárgyaltuk, a SEQ adat feldolgozása a hardver egységekben egyetlen órajel alatt zajlik. A diagram nem mutatja a kisebb kombinációs logikai blokkokat, a különböző hardver egységeket vezérlő logikát és a megfelelő értékeket az egységekhez szállító útvonalakat. Ezeket a részleteket később adjuk hozzá. Az a módszer, amellyel a processzort alulról felfelé haladó a folyamatként ábrázoljuk, nem szokásos. Ennek okát majd a futószalagos processzor tervezésének tárgyalásakor magyarázzuk meg.



1.16. ábra. **ASEQ, a soros megvalósítás absztrakt képe.** Az utasítás végrehajtása során az információ feldolgozás az óramutató járásának megfelelően történik.

A feldolgozás különböző szakaszaihoz tartozó hardver egységek:

- ***Fetch*** A program számláló regisztert használva címként, az utasítás memória beolvassa az utasítás bájtjait. A PC növekmény számító kiszámítja a megnövelt program számláló (**valP**) értékét.
- ***Decode*** A regiszter tömbnek két portja van (**A** és **B**), amelyeken keresztül a **valA** és **valB** regiszter értékeket egyidejűleg ki tudja olvasni.
- ***Execute*** Ez a szakasz az aritmetikai és logikai egységet (ALU) az utasítás típusától függően különböző célokra használhatja. Egész műveletek esetén elvégzi a megadott műveletet, más utasítások esetén összeadóként funkcionál, kiszámítja a növelt vagy csökkentett veremmutató értéket, a tényleges címet vagy egyszerűen (nulla hozzáadásával) átadja a bemeneteit a kimeneteire.
  A **CC** feltétel kód regiszter három feltétel-kód bitet tartalmaz. A feltételt kódok új értékét az ALU számítja ki. Amikor ugró utasítást kell végrehajtani, az elágazást vezérlő **Cnd** jel a feltétel kódok és az ugrás típusa alapján számítódik ki.
- ***Memory*** Az adat memória írja vagy olvassa a memória egy szavát, amikor memória utasítást hajt végre. Az adat és az utasítás memória ugyanazokat a fizikai memóriahelyeket használja, csak más céllal.

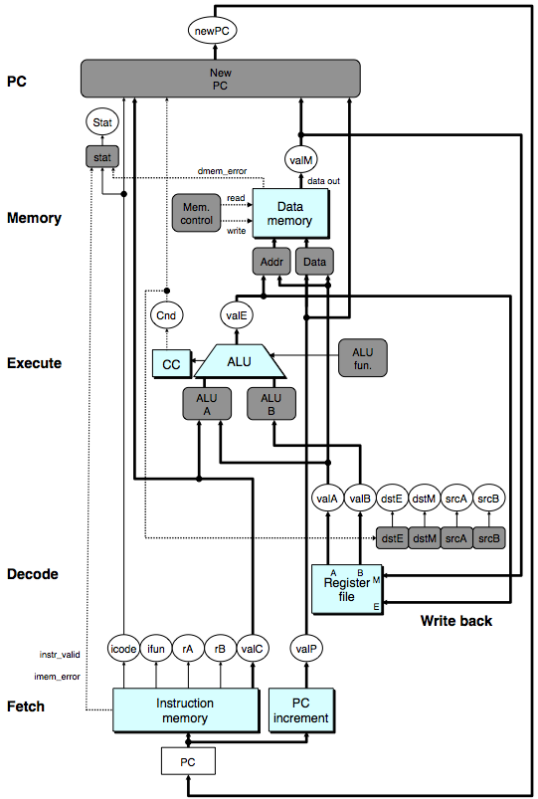
- ***Write back*** A regiszter tömbnek két portja van. Az **E** portot használjuk az ALU által számított értékek beírására, az **M** portot pedig a memóriából beolvasott értékek beírására.

A 1.17 ábra mutatja be a SEQ részletesebb ábrázolását, ami a megvalósításhoz szükséges.(Itt még mindig nem látni az összes részletet, azokat az egyes szakaszok megvalósításánál találjuk.) Itt és a többi hardver diagram esetén is, a következő rajzi jelöléseket használjuk:

- A hardver egységeket kis kék dobozok mutatják. Ilyenek a memóriák, az ALU, stb. Mind a négy processzor implementációban ugyanazt az alap egység készletet fogjuk használni. Az egységeket "fekete doboz" gyanánt kezeljük és nem megyünk bele azok tervezési részleteibe.
- A logikai vezérlő blokkokat lekerekített szürke négyszögekként rajzoltuk. Ezek a blokkok szolgálnak a jelforrás készletek közötti választásra vagy valamely logikai függvény kiszámítására. Ezeket teljes egészében megvizsgáljuk, sőt, HCL leírást is fejlesztünk rájuk.
- A vezeték neveket kerekített fehér dobozok jelzik. Ezek csak címkék a vezetéken, nem önálló hardver elemek.

- A szó-szélességű adat kapcsolatokat közepes szélességű vonalak jelzik. Ezek valójában 32-bites vezetékek, párhuzamosan kapcsolva, amelyek egy szó adatot visznek át az egyik hardverről a másikra.
- A bájt és az ennél kisebb szélességű adat kapcsolatokat vékony vonal jelöli. Ezen vonalak mindegyike egy négy vagy nyolc vezetéket tartalmazó köteget jelöl, az átviendő adat típusától függően.
- Az egybites kapcsolatokat pontozott vonal jelöli. Ezek a lapkán levő egységek és blokkok közötti vezérlő értékeket ábrázolják.

Az eddig bemutatott (1.3-1.6 számú) táblázatokban az egyes sorok egy bizonyos érték (pl. **valP**) kiszámítását mutatták, vagy bizonyos hardver egység (pl. memória) aktiválását. Ezek a számítások és aktiválások vannak feltüntetve a 1.7 táblázat második oszlopában. Az eddig leírt jeleken túlmenően, ez a lista négy regiszter azonosító jelet tartalmaz: **srcA**, a **valA** forrása; **srcB** a **valB** forrása; **dstE**, ahová **valE** beíródik; **dstM**, ahová **valM** beíródik. A két jobb oldali oszlop pedig, illusztrációként, azt mutatja, milyen konkrét számításokat kell elvégezni az `OPl` és `mrmovl` utasítások esetén. Hogy ezeket a számításokat hardverre leképezzük, olyan vezérlő logikát akarunk megvalósítani, amelyik átviszi az adatokat a különböző hardver egységek között és ezeket az egységeket olyan



1.17. ábra. **Az Y86 SEQ (szekvenciális) megvalósítása.** Néhány vezérlő jelet, valamint regiszter és vezérlő szavak közötti kapcsolatokat nem mutatja az ábra.

módon működteti, hogy azok a különböző utasítás típusok esetén az ott megadott műveleteket végezzék. Ez a célja a 1.17 ábrán szürke kerekített négyszögként mutatott logikai blokkoknak. Vegyük sorra az egyes végrehajtási szakaszokat és tervezzük meg ezeknek a blokkoknak a részleteit.

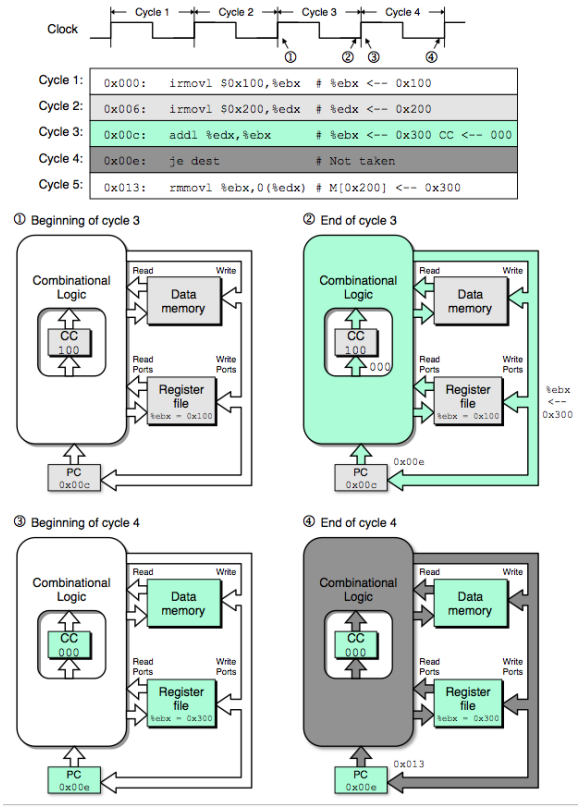
1.7. táblázat. **A számítási lépések azonosítása a soros megvalósításban.** A második oszlop mutatja a SEQ egyes fázisaiban kiszámított értéket vagy elvégzett műveletet. Az `OPl` és `mrmovl` műveletei példaként szerepelnek.

Stage	Computation	`OPl` **rA, rB**	**mrmovl D(rB), rA**
Fetch	icode, ifun	icode : ifun ← M_1[PC]	icode : ifun ← M_1[PC]
	rA, rB	rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]
	valC		valC ← M_4[PC+2]
	valP	valP ← PC+2	valP ← PC+6
Decode	valA, srcA	valA ← R[rA]	
	valB, srcB	valB ← R[rB]	valB ← R[rB]
Execute	valE	valE ← valB OP valA	valE ← valB + valC
	Cond codes	Set CC	
Memory	read/write		valM ← $M_4[valE]$
Write back	E port, dstE	R[rB] ← ValE	
	M port, dstM		R[rA] ← ValM
PC update	PC	PC ← ValP	PC ← ValP

1.3.3. A SEQ időzítés

Amikor a 1.3-1.6 táblázatokat bemutattuk, azt mondtuk, hogy ezeket úgy kell olvasni, mintha valamiféle programozási jelöléseket tartalmaznának, ahol az értékadások felülről lefelé haladva történnek meg. A 1.17 ábrán bemutatott hardver szerkezet viszont ettől teljesen eltérő módon működik: egy órajel váltja ki azt a folyamot, amelyik a kombinációs logikán át egy egész utasítást hajt végre. Lássuk tehát, hogy a hardver hogyan képes a megismert táblázatokban leírt viselkedést megvalósítani.

SEQ implementációnk egy kombinációs logikából és kétfajta memória eszközből áll: órajel vezérelt regiszterekből (a program számláló és feltétel kód regiszter) valamint véletlen hozzáférésű memóriákból (a regiszter tömb, az utasítás memória és az adat memória). A kombinációs logika nem igényel valamiféle sorrendiséget vagy vezérlést: az értékek egyszerűen áthaladnak a logikai kapuk hálózatán, amikor a bemenetek megváltoznak. Mint már említettük, azt tételezzük fel, hogy egy véletlen hozzáférésű memóriából az olvasás egy kombinációs logikához hasonló módon működik, ahol a kimenő szó egy cím bemenet alapján generálódik. Ez ésszerű feltevés kisebb memóriák (mint egy regiszter tömb) esetén, és ezt speciális órajel generáló áramkörökkel imi-



1.18. ábra. **A SEQ két végrehajtási ciklusának nyomon követése.** Mindegyik ciklus azzal kezdődik, hogy az állapot elemeket (program számláló, feltétel kód regiszter, regiszter tömb, és adat memória) az előző utasításnak megfelelően betöltjük. A jelek a kombinációs logikán áthaladva hozzák létre az új állapot elemeket.

tálni tudjuk nagyobb áramkörök esetén is. Mivel utasítás memóriánkat csak utasítás beolvasásra használjuk, ezt az egységet úgy kezelhetjük, mintha az kombinációs logika lenne. Emiatt négy olyan hardver egységünk marad, amelyek explicit sorrendi vezérlést igényelnek: a program számláló, a feltétel kód regiszter, az adat memória és a regiszter fájl. Ezeket egyetlen órajel kezeli, amelyik kiváltja az új értékek regiszterekbe töltődését és értékek beírását a véletlen hozzáférésű memóriákba. A program számlálóba minden órajellel egy új utasítás kerül. A feltétel kód regiszter csak akkor frissül, amikor egy egész típusú műveletet hajtunk végre. Az adat memóriát csak akkor írjuk, amikor `rmmovl`, `pushl` vagy `call` utasítást hajtunk végre. A regiszter tömb két író bemenete lehetővé teszi, hogy minden órajel hatására egyidejűleg két regisztert frissítsünk, de használhatjuk a **0xF** speciális regiszter azonosítót annak jelzésére, hogy erre a portra nem akarunk írni.

Mindössze a regiszterek és memóriák órajel vezérlése szükséges ahhoz, hogy processzorunk aktivitásainak sorrendjét vezérelni tudjuk. Hardverünk ugyanazt a hatást éri el, mint amit a 1.3-1.6 táblázatokban bemutatott értékadások sorban való végrehajtásával, bár ezek az állapotfrissítések egyidőben történnek, és csak akkor, amikor az órajel felfut, a következő ciklust elindítandó. Ez az egyenértékűség az

Y86 utasításkészlet természetéből következik, és azért, mert úgy szerveztük meg a számításokat, hogy a tervünk figyelembe veszi a következő elvet: *A processzornak soha nem kell visszaolvasnia egy utasítás által frissített állapotot, hogy be tudja fejezni az adott utasítás végrehajtását.* Ez az elv nagy szerepet játszik implementációnk sikerében. Illusztrációként tételezzük fel, hogy a `pushl` utasítást úgy implementáltuk, hogy először csökkentjük %`esp` értékét néggyel, majd ezután használjuk %`esp` frissített értékét az író művelet címeként. Ez a megközelítés sértené az előbbi elvet: megkövetelné, hogy olvassuk be a regiszter tömbből a frissített értéket, hogy el tudjuk végezni a memória műveletet. Ehelyett implementációnk a verem mutató csökkentett értékét mint a **valE** jelet állítja elő és ezt a jelet (lásd 1.5 táblázat) használja mind adatként a regiszter írásához, mind címként a memória írásához. Ennek eredményeként egyszerre tudja a regisztert és a memóriát beírni, amikor az órajel felfut a következő ciklus elkezdésekor.

Az elv másik illusztrációjaként: láthatjuk, hogy bizonyos műveletek (az egész típusú műveletek) beállítják a feltétel kódokat, és bizonyos utasítások (az ugró utasítások) pedig olvassák ezeket a kódokat, de nincs olyan utasítás, aminek olvasni és írni is kellene a feltétel kódokat. Bár a feltétel kódokat addig nem állítjuk be, amíg a következő ciklust indító órajel emelkedni nem kezd, azok mégis frissülnek, mielőtt bármely másik

Programlista 1.5: **Egy Y86 minta-utasítás sorozat.** Ezzel követjük nyomon az utasítás végrehajtását a különböző szakaszokban.

```
1  0x000:  irmovl  $0x100,%ebx    #  %ebx  <--  0x100
2  0x006:  irmovl  $0x200,%edx    #  %edx  <--  0x200
3  0x00c:  addl  %edx,%ebx        #  %ebx  <--  0x300  CC  <--  000
4  0x00e:  je  dest               #  Not  taken
5  0x013:  rmmovl  %ebx,0(%edx)   #  M[0x200]  <--  0x300
6  0x019:  dest:  halt
```

utasítás megpróbálná azokat olvasni. A 1.18 ábra mutatja, hogyan hajtaná végre a SEQ hardver a 1.5 programlista 3. és 4. sorában található utasítás sorozatot. Az 1...4 diagramok mutatják a négy állapot elemet, továbbá a kombinációs logikát és az állapot elemek közötti kapcsolatokat. Az ábra úgy mutatja a kombinációs logikát, hogy az benne van a feltétel kód regiszterben, mivel a kombinációs logika (mint az ALU) bemenő jelet állít elő a feltétel kód regiszter számára, más részei (mint az elágazás számító és a PC választó logika) pedig bemenetként használja a feltétel kód regisztert. Az ábrán a regiszter tömb és az adat memória különálló kapcsolatokkal rendelkezik írásra és olvasásra, mivel az olvasási műveletek úgy haladnak át ezeken az egységeken, mintha azok kombinációs logika lennének, míg az írási műveleteket az órajel vezérli.

A 1.18 ábra színkódolással mutatja, hogyan viszonyulnak az áramköri jelek az éppen végrehajtás alatt levő utasításokhoz. Feltételezzük, hogy a végrehajtás a **100** értékű állapotban (**ZF**, **SF**, **OF** sorrendben) kezdődik el. A 3. órajel elején (1. pont), az állapot elemek azt az állapotot tartalmazzák, amelyet a 1.5 programlista 2. sorában szereplő `irmovl` utasítás végrehajtása utáni frissítés okozott; az ábrán világos szürkével. A kombinációs logikát az ábra fehérrel mutatja, ami azt jelzi, hogy még nem volt ideje reagálni a megváltozott állapotra. Az órajel ciklus azzal kezdődik, hogy a **0x00c** cím betöltődik a program számlálóba. Ennek hatására az `addl` utasítás (a programlista 3. sora), kékkel jelölve, beolvasódik és végrehajtódik. Az értékek átfolynak a kombinációs logikán, beleértve az olvasást a véletlen hozzáférésű memóriából. A ciklus végére (2. pont) a kombinációs logika előállította a feltétel kódok új értékét (**000**), az `%ebx` program regiszter frissítését, és a program számláló új értékét (**0x00e**). Ezen a ponton a kombinációs logika az `addl` utasításnak megfelelően frissült (kékkel jelölve), de az állapot még a második `irmovl` utasításnak megfelelő állapotot (világos szürkével) őrzi.

Amint az órajel a 4. ciklus kezdetén (3. pont) megemelkedik, megtörténik a program számláló, a regiszter tömb, és a feltétel kódok frissítése, így ezeket már kékkel mutatjuk, de a kombinációs logika még nem reagált ezekre a változásokra, így azt fehér ábrázolja.

Ebben a ciklusban a `je` utasítás (a programlista 4. utasítása), itt sötét szürkével jelölve, beolvasódik és végrehajtódik. Mivel a **ZF** feltétel kód értéke **0**, nem történik elágazás. A ciklus végére (4. pont), a program számláló **0x013** új értéke áll elő. A kombinációs logika frissült a `je` utasításnak megfelelően (sötétszürke színű), de az állapot még azt az értéket őrzi, amit az `addl` utasítás állított be (kék színű), amíg a következő ciklus el nem kezdődik.

Amint ez a példa is mutatja, órajelet használni az állapot elemek frissítésére, kombinálva az értékek kombinációs logikán való áthaladásával, elegendő ellenőrzést biztosít SEQ implementációnk utasításai által végzett számítások ellenőrzésére. Az órajel minden alacsonyból magasba történő átmeneténél a processzor új utasítás végrehajtását kezdi meg.

1.3.4. A SEQ szakaszainak implementálása

Ebben a szakaszban szemügyre vesszük a SEQ megvalósításához szükséges blokkok HCL leírásait. A teljes HCL működés leírás az arch:hcl helyen található. Néhány blokkot itt bemutatunk, másokat gyakorló feladatként valósítunk meg. Az ajánluk, hogy ezeket a gyakorló feladatokat úgy dolgozzuk ki, hogy ezzel lemérhetjük, mennyire sikerült megérteni, hogy hogyan viszonyulnak ezek a blokkok a különböző utasítások számítási igényeinek megvalósításához. A SEQ leírásának része a HCL műveletek argumentumaiként használt különböző egész és logikai jelek definíciója, amit itt nem tárgyalunk. Ide tartoznak a különböző hardver jelek nevei, az utasítás kódok konstans értéke, a funkció kódok, a regiszter nevek, ALU műveletek és állapot kódok. Csak azokat mutatjuk be, amelyeket a vezérlő logika explicit módon használ. A használt konstansokat a1.8 táblázat dokumentálja. Megállapodás szerint a konstans értékekre nagybetűs neveket használunk.

A 1.3-1.6 számú táblázatokban bemutatott utasításokon felül bemutatjuk a `nop` és `halt` utasítások végrehajtását is. A `nop` utasítás egyszerűen átfolyik az állapotokon, különösebb feldolgozás nélkül, kivéve, hogy a PC értékét eggyel megnöveli. `halt` utasítás

a processzor állapotát HLT értékűre állítja, aminek hatására megáll a működés.

Utasítás elővétel szakasz

A **fetch** szakasz tartalmazza az utasítás memória hardver egységet, lásd 1.19 ábra. Ez az egység egyszerre 6 bájtot olvas be a memóriából, a PC-t használva az első bájt (byte 0) címeként. Ezt az első bájtot utasítás bájtként értelmezi és (a **Split** egység által) két négybites részre vágja. A **icode** és **ifun** jelű blokkok ezután kiszámítják az utasítás és funkció kódokat a memóriából kiolvasott érték alapján, vagy pedig, ha az utasítás érvénytelen (amit az **imem_error** jel mutat), egy **nop** utasításnak megfelelően. Az **icode** értéke alapján három 1-bites jelet számíthatunk ki (szaggatott vonallal jelölve):

- **instr_valid**: Megfelel ez a bájt egy tényleges Y86 utasításnak? Ezt a jelet illegális utasítás detektálására használjuk.
- **need_regids**: Tartalmaz az utasítás regiszter kijelölő bájtot?
- **need_valC**: Van az utasításban konstans szó?

Az **instr_valid** és **imem_error** jeleket (amik akkor keletkeznek, amikor az utasítás címe a határokon kívül esik) használjuk a **memory** szakaszban a státus kód előállítására.

Példaként **need_regids** (ami egyszerűen azt adja meg, hogy **icode** értéke egyike azon utasításoknak, amelyeknek van regiszter kijelölő bájtja) HCL definíciója:

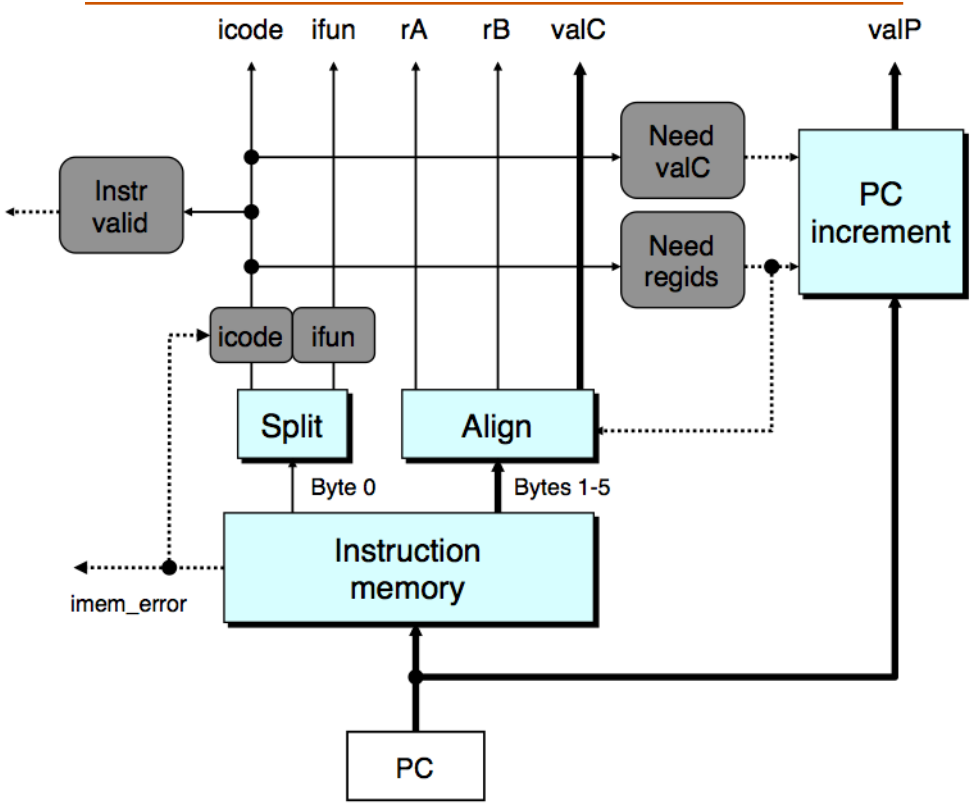
```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                   IIRMOVL, IRMMOVL, IMRMOVL };
```

Amint a 1.19 ábra mutatja, a memóriából beolvasottak közül fennmaradó 5 bájtban van kódolva a regiszter kijelölő bájt és a konstans szó. Ezeket a bájtokat az **Align** unit dolgozza fel regiszter mezőkké és konstans szóvá. Amikor a kiszámított **need_regids** értéke **1**, akkor az első bájtot az egység a **rA** és **rB** regiszter kijelölőkre vágja szét, különben pedig ez a két mező a **0xF** (**RNONE**) értéket kapja annak jelzésére, hogy az utasítás nem jelölt ki regisztereket. Idézzük fel (lásd 1.2 ábra), hogy ha bármely utasításnak csak egy regiszter operandusa van, akkor a regiszter kijelölő bájt másik mezője **0xF** (**RNONE**) értéket kap. Ezért úgy vehetjük, hogy az **rA** és **rB** jelek vagy ez elérni kívánt regisztert adják meg, vagy azt jelzik, hogy nem akarnak regiszterhez hozzáférni. Az **Align** jelű egység a **valC** állandó értéket is előállítja. Ehhez vagy az 1...4 vagy a 2...5 bájtokat használja, **need_regids** értékétől függően.

A **PC increment** hardver egység állítja elő a **valP** értéket, a PC aktuális értéke, továbbá a **need_regids** és a **need_valC** jelek alapján. Amikor a PC értéke **p**, **need_regids** értéke **r** és **need_valC** értéke **i**, a programszámláló növelő egység a **p + 1 + r + 4i** értéket állítja elő.

1.8. táblázat. **A HCL leírásban használt konstans értékek.** Ezek az értékek ábrázolják az utasítások, funkció kódok, regiszter azonosítók, ALU műveletek, és állapot kódok értékét.

Name	Value (Hex)	Meaning
INOP	0	Code for nop instruction
IHALT	1	Code for halt instruction
IRRMOVL	2	Code for rrmovl instruction
IIRMOVL	3	Code for irmovl instruction
IRMMOVL	4	Code for rmmovl instruction
IMRMOVL	5	Code for mrmovl instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for call instruction
IRET	9	Code for ret instruction
IPUSHL	A	Code for pushl instruction
IPOPL	B	Code for popl instruction
FNONE	0	Default function code
RESP	4	Register ID for %esp
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for halt

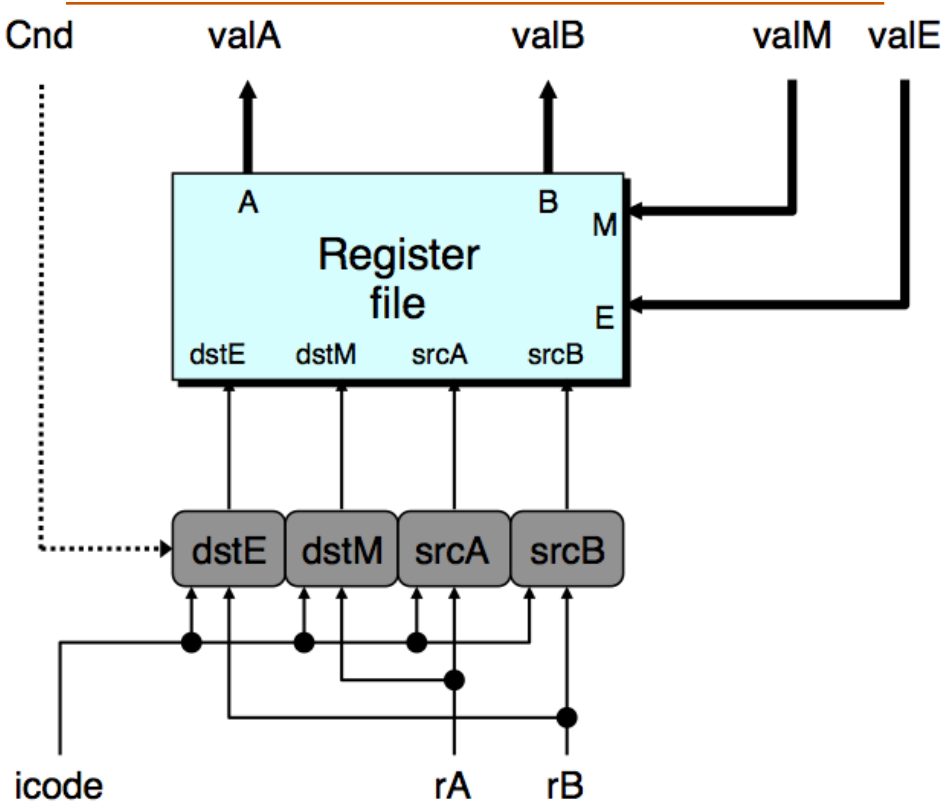


1.19. ábra. **SEQ utasítás elővétel (Fetch).** Hat bájt olvasódik be az utasítás memóriából a PC értékét használva kezdőcímként. Ezekből a bájtokból különféle utasítás mezőket készítünk. A **PC increment** blokk számítja ki **valP** értékét.

Dekódolási és visszaírási szakasz

A 1.20 ábra mutatja be részletesen azt a logikát, amelyik a dekódolási és a visszaírási szakaszt valósítja meg. Ez a két szakasz azért kerül itt össze, mert mindkettőnek el kell érnie a regiszter tömböt. A regiszter tömbnek négy portja van. Támogatja két egyidejű olvasás (az **A** és a **B** porton) valamint két írás (az **E** és az **M** porton) elvégzését. Minden portnak van cím és adatvonala, ahol az adatvonalra a regiszter azonosító kerül, az adat vonal pedig 32 vezetékből áll, amelyik vagy kimenő szóként (olvasás esetén) vagy bemenő szóként (írás esetén) használ a regiszter tömb. A két olvasó port cím bemenetei **srcA** és **srcB**, míg az író portok cím bemenetei **dstE** és **dstM**. A bemenetek bármelyiként a **0xF** (**RNONE**) speciális érték azt jelzi, hogy nem kell regisztert elérni. Az ábra alján a négy blokk négy különböző regiszter azonosítót állít elő a regiszter tömb számára, az **icode** utasítás kód alapján; az **rA** és **rB** regiszter kijelölők, valamint esetlegesen a **Cnd** feltétel jel a végrehajtási szakaszban számítódnak ki.

Az **srcA** azt jelöli ki, hogy melyik regisztert kell beolvasni a **valA** előállításához. A kívánt érték az utasítás típusától függ, amint az a 1.3-1.6 számú táblázatok első soraiban látjuk. A felsorolt tagokat egyetlen kifejezésbe kombinálva **srcA** kiszámítására



1.20. ábra. **A SEQ dekódoló és viszaíró szakasza.** Az utasítás mezők dekódolásával áll elő négy cím (kettő íráshoz, kettő olvasáshoz), amelyik a regiszter tömb regisztereit azonosítja. A regiszterekből beolvasott értékek lesznek a **valA** és **valB** jelek. A **valE** and **valM** write-back értékek szolgálnak adatként az írási műveletekben.

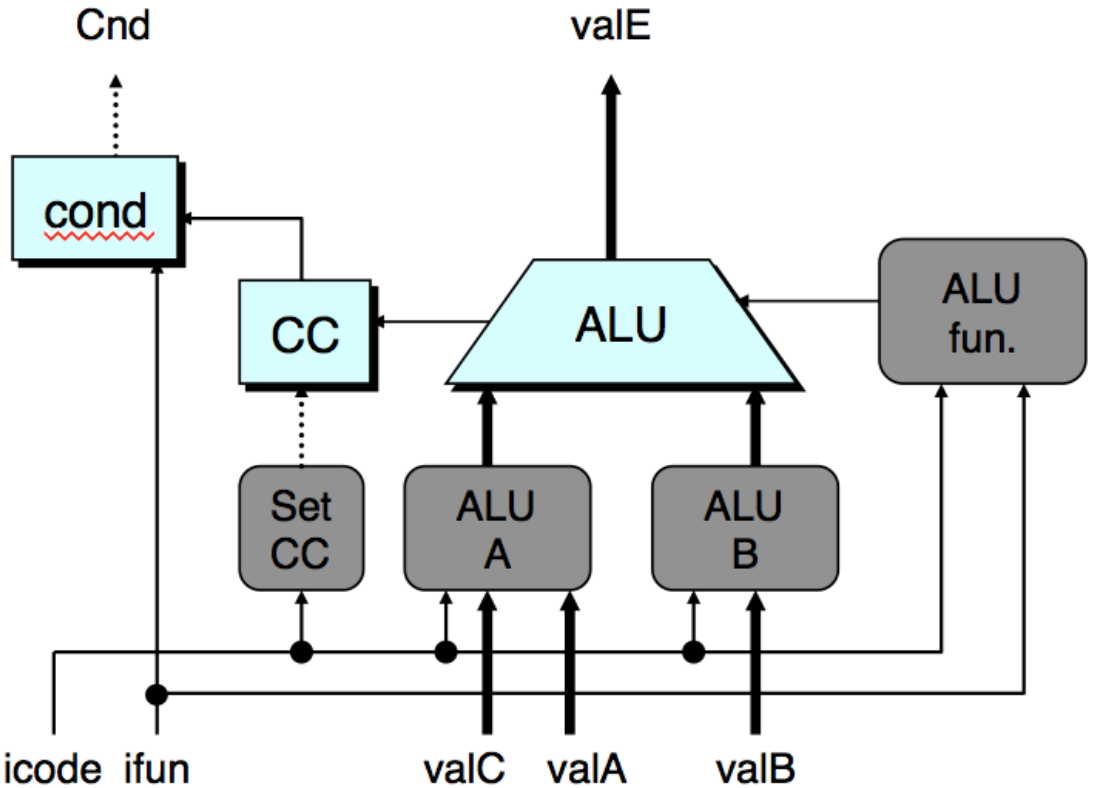
a következő HCL leírást kapjuk (idézzük fel, hogy **RESP** az %esp regiszter azonosítója):

```
# Code from SEQ
int srcA = [
icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
icode in { IPOPL, IRET } : RESP;
1 : RNONE; # Don't need register
];
```

A **dstE** regiszter jelöli az **E** port számára a cél regisztert, ahol a kiszámított **valE** értéket tároljuk. A 1.3-1.6 számú táblázatokon ezt láthatjuk a visszaírás szakasz első lépéseként. Ha pillanatnyilag elhanyagoljuk a feltételes adatmozgató utasításokat, valamennyi különböző utasítás esetén az alábbi HCL kifejezéssel írhatjuk le **dstE** értékét:

```
# VIGYÁZAT: A feltételes adatmozgatás itt hibás
int dstE = [
icode in { IRRMOVL } : rB;
icode in { IIRMOVL, IOPL} : rB;
icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
1 : RNONE; # Nem kell semmilyen regisztert írni
];
```

Az **execute** szakasz tárgyalása során újból tárgyaljuk ezt a jelet, valamint hogy hogyan kell implementálni a feltételes adatmozgatást.



1.21. ábra. **A SEQ végrehajtási szakasza.** Az ALU vagy egész típusú műveletet végez, vagy összeadóként működik. A feltétel jelzőbitek az ALU értékének megfelelően állnak be. A feltétel kódokat vizsgáljuk meg, hogy történjen-e elágazás.

A végrehajtási szakasz

A végrehajtási szakasz az aritmetikai és logikai egységet (ALU) tartalmazza. Ez az egység végzi az **add**, **subtract**, **and** és **Exclusive-Or** műveleteket az **aluA** és **aluB** bemeneteken az **ALU fun** bemenet értékétől függően. Az adatokat és a vezérlő jeleket három vezérlő blokk állítja elő, lásd 1.21 ábra. Az ALU kimenő jeléből lesz a **valE** érték.

A 1.3-1.6 számú táblázatokon az egyes utasítások ALU számításait látjuk az **execute** szakasz első lépéseként. Az operandusok **aluB** után álló **aluA** sorrendben vannak feltüntetve, annak biztosítására, hogy a `subl` utasítás a **valA** értéket vonja ki a **valB** értékből. Azt látjuk, hogy az **aluA** értéke lehet **valA**, **valC**, **-4** vagy **+4**, az utasítás típusától függően. Ezért az **aluA** értéket előállító vezérlő blokk viselkedését a következő módon írhatjuk le:

```
int aluA = [
icode in { IRRMOVL, IOPL } : valA;
icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
icode in { ICALL, IPUSHL } : -4;
icode in { IRET, IPOPL } : 4;
# Más utasítások nem használják az ALU-t
];
```

Ha megnézzük az ALU által az **execute** szakaszban végzett műveleteket, azt látjuk, hogy azok legtöbbször összeadások. Az `OPl` utasítások esetében azonban azt akarjuk, hogy

az utasítás **ifun** mezőjében kódolt műveletet hajtsa végre. Az ALU esetében ezért a HCL leírás:

```
int alufun   = [
             icode == IOPL : ifun;
             1 : ALUADD;
             ];
```

Az **execute** szakasz a feltétel kódokat is tartalmazza. Az ALU minden működési lépés során három olyan jelzést állít elő, amelyeken a feltétel kódok (zero, sign és overflow) értékén alapulnak. Mi azonban csak akkor akarjuk a feltétel jelzőbiteket beállítani, amikor `OPl` utasítást hajtunk végre. Ennek érdekében előállítjuk a **set_cc** jelet, hogy kell-e a feltétel jelzőbiteket beállítani:

```
bool set_cc = icode in  IOPL ;
```

A **cond** jelű hardver egység a feltétel jelzőbitek és a funkció kód kombinációját használja annak meghatározására, hogy feltételes elágazás vagy adatátvitel kerül-e sorra (lásd 1.3 ábra). Előállít egy **Cnd** jelet, amelyiket használ a **dstE** kiszámításához feltételes adatátvitel esetén, valamint a következő PC kiszámításához feltételes elágazás esetén. Egyéb utasítások esetén ugyan a **Cnd** jel **0** vagy **1** lehet, az utasítás funkció kódjától és a feltétel kódok értékétől függően, de azt a vezérlő logika elhanyagolja. Ennek részleteit most nem tárgyaljuk.

A memória szakasz

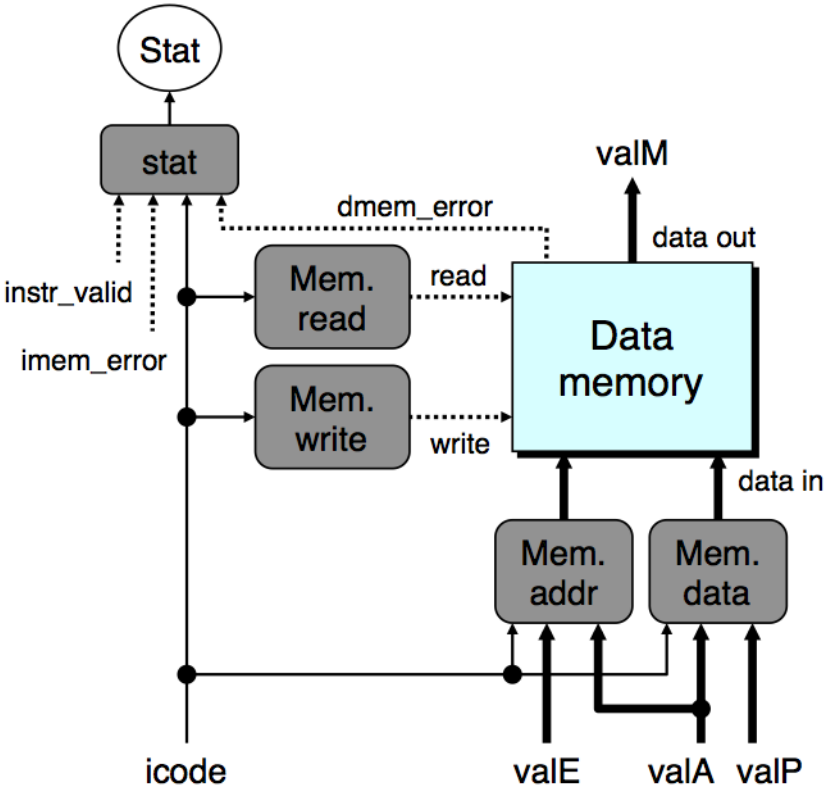
A memória szakasz feladata program adatok olvasása a memóriából vagy beírása a memóriába. Két vezérlő blokk állítja elő a memória címet és a memória adatot (írás esetén), lásd 1.22 ábra. Két további blokk állítja elő a vezérlő jeleket, amelyek azt jelzik, hogy írás vagy olvasás műveletet kell végrehajtani. Amikor olvasás műveletet kell végezni, a **data memory** egység állítja elő a **valM** értéket. A kívánt memória művelet az egyes utasítás típusokra a 1.3-1.6 számú táblázatok "memory" szakaszában láthatók. Figyeljük meg, hogy a memória olvasások és írások címe mindig **valE** vagy **valA**. Ezt a blokkot a következő HCL kifejezéssel írhatjuk le:

```
int    mem_addr = [
       icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
       icode in { IPOPL, IRET } : valA;
       # Más utasítások nem használnak memória címet
       ];
```

A **mem_read** vezérlő jelet csak azoknál az utasításoknál akarjuk beállítani, amelyek adatot olvasnak a memóriából, amit a következő HCL kód fejez ki:

```
bool mem_read = icode in  IMRMOVL, IPOPL, IRET ;
```

A **memory** szakasz utolsó feladata az utasítás végrehajtásának eredményeként előálló **Stat** állapot kód kiszámítása, a **fetch** szakaszban előállított **icode**, **imem_error** és **instr_valid** értékekből, valamint a **data memory** által előállított **dmem_error** jelből.



1.22. ábra. **A SEQ memória szakasza.** A **data memory** egység memória értékeket írhat és olvashat. A memóriából beolvasott értékből lesz a **valM** jel.

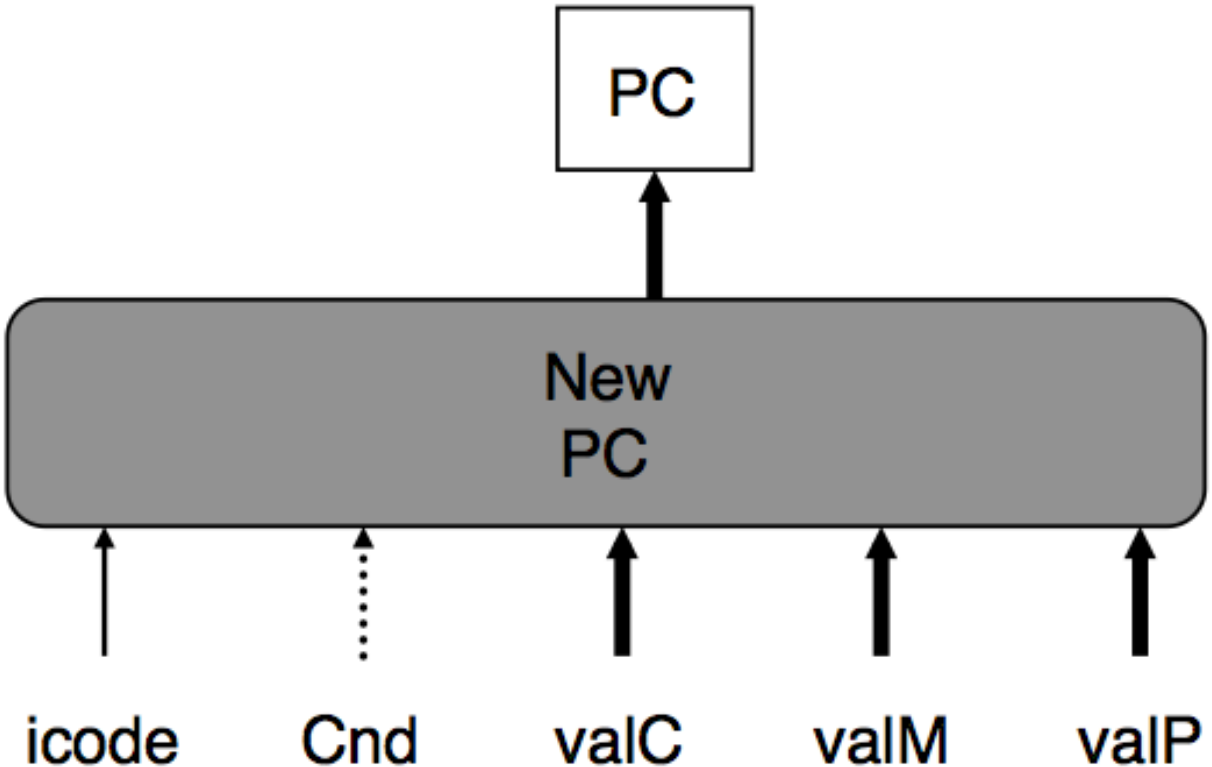
PC frissítés szakasz

A SEQ az utolsó szakaszban új értéket állít elő a program számláló számára, lásd 1.23 ábra. Amint a 1.3-1.6 számú táblázatokon az utolsó lépések mutatják, a PC új értéke a **valC**, **valM**, és **valP** jelek valamelyike lesz, az utasítás típusától és attól függően, hogy volt-e elágazás. HCL nyelven ez a következő módon írható le:

```
int    new_pc = [
       # Call. Use instruction constant
       icode == ICALL : valC;
       # Taken branch. Use instruction constant
       icode == IJXX && Cnd : valC;
       # Completion of RET instruction. Use value from stack
       icode == IRET : valM;
       # Default: Use incremented PC
       1 : valP;
       ];
```

A SEQ áttekintése

Végig mentünk az Y86 processor teljes tervén. Azt láttuk, hogy ha a különböző utasítások egyes szakaszaiban szükséges lépéseket egységes folyammá szervezzük, akkor az egész processzort kis számú hardver egységből állíthatjuk össze és egyetlen órajellel vezérelhetjük a szükséges számítások megfelelő sorrendben való végrehajtását. A vezérlő logikának ezen egységek közötti jeleket kell a megfelelő útvonalon



1.23. ábra. **SEQ PC frissítés szakasz.** A PC következő értéke a **valC**, **valM**, és **valP** jelek valamelyike lesz, az utasítás kódtól és az elágazás jelzőbittől függően.

továbbítani, továbbá előállítani az utasítás típusától és az elágazási feltételektől függő vezérlőjeleket.

A SEQ egyetlen baja, hogy túlságosan lassú. Az órajelnek elég lassúnak kell lennie ahhoz, hogy a jelek egyetlen órajel alatt át tudjanak haladni valamennyi szakaszon. Példaként tekintsük egy `ret` utasítás végrehajtását. Frissített programszámlálóval indulva az órajel ciklus elején, az utasítást be kell olvasni az utasítás memóriából, a verem mutatót a regiszter tömbből, az ALU-nak csökkenteni kell a veremmutató értékét, a visszatérési címet be kell olvasni a memóriából, hogy meg tudjuk határozni a program számláló következő értékét. Mindezt pedig be kell fejezni az órajel ciklus végéig.

Az ilyen stílusú megvalósítás nagyon rosszul használja ki hardver egységeinket, mivel minden egység csak a teljes órajel ciklus töredékében aktív. Látni fogjuk, hogy sokkal jobb hatékonyságot tudunk elérni a futószalag elv alkalmazásával.

1.4. A futószalag elv általában

1.5. Y86 megvalósítás futószalagon

1.6. Összefoglalás

Tárgymutató

Táblázatok jegyzéke

Ábrák jegyzéke

Programlisták

Bibliography

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2014. ISBN: 978-1-292-02584-1.

[2] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information Technology Approach*. Fourth. John Wiley & Sons, Inc., 2010. ISBN: 978-0-470-40028-9.

[3] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information Technology Approach*. http://www.wiley.com/go/global/englander. 2010.

[4] Neil Matthew and Richard Stones. *Beginning Linux Programming*. http://longfiles.com/fzi3sbsh0lhu/LinuxProgr4th147627.pdf.html. Wrox Press Ltd, 2008. ISBN: 978-0-470-14762-7.

[5] Clive "Max" Maxfield. *DIY Calculator*. http://diycalculator.com/. 2003.

[6] Clive "Max" Maxfield. *How Computers Do Math*. John Wiley & Sons, Inc., 2005. ISBN: 0471732788.

[7] Clive "Max" Maxfield and Alvin Brown. *The Official DIY Calculator Data Book*. John Wiley & Sons, Inc., 2005. ISBN: 0471732788.

[8] Stanley J. Warford. *Computer Systems*. Jones and Bartlett, 2010. ISBN: 0-7637-7144-9.