2 AGINDU MULTZOA

2.1 MAKINA-LENGOAIA ETA MIHIZTADURA LENGOAIA

Konputagailu batek interpreta dezakeen lengoaia bakarra beraren makina-lengoaia da. Makina-lengoaiako aginduak konputagailuaren arkitektura-ezaugarrien mende daude, eta zerokoen eta batekoen segiden bidez adierazten dira (kodeketa bitarra). Lehenengo konputagailuetarako idatzi ziren programak, makina-lengoaiaz idatzi ziren.

Hala ere, oso astuna da zerokoen eta batekoen segiden bidez programak idaztea; akats asko egiten dira, eta oso zaila da programa zuzenak sortzea. Arazo horiek gainditzeko asmoz, ahalmen semantiko handiagoko lengoaia sinbolikoak sortu ziren. Lengoaia horien artean, mihiztadura-lengoaia da makina-lengoaiatik hurbilen dagoena; edukiera semantiko gutxienekoa, alegia. Beste muturrean, goi-mailako lengoaiak ditugu. Lengoaia sinbolikoak erabiliz, programak errazago idazten dira, baina erabiltzen diren aginduak ezin dira zuzenean konputagailuan exekutatu. Izan ere, programa horiek itzuli egin behar dira konputagailuak ulertzen duen lengoaia bakarrera, eta, itzulpen horretarako, programa bereziak erabili behar dira: mihiztatzaileak (mihiztadura-lengoaietarako) eta konpiladoreak (goi-mailako lengoaietarako).

Goi-mailako lengoaiak dira programatzaileek gehien erabiltzen dituztenak (ADA, Cobol, Fortran, C, Lisp, Java, ...). Lengoaia horietan programatzeko ez da konputagailuaren arkitektura ezagutu behar, lengoaien sintaxia makinaren baliabideen guztiz independentea baita. Aipatu dugun moduan, goi-mailako lengoaien aginduen edukiera semantikoa makina-lengoaiako aginduena baino askoz altuagoa da. Hori dela eta, agindu bakar batekin, makina-lengoaiako n agindu ordezkatzen dira ($1{:}n$ korrespondentzia dago). Beraz, makina-lengoaiako aginduekin erkatuta, goi-mailako aginduek lan gehiago egiten dute (eta, ondorioz, haien exekuzio-denbora altuagoa da).

Lehen aipatu bezala, mihiztadura-lengoaia da makina-lengoaiatik hurbilen dagoen lengoaia sinbolikoa. Aginduak mnemoteknikoen bidez idazten dira. Mihiztadura-lengoaiako agindu batek makina-lengoaiako agindu bat ordezkatzen du, hau da, erlazioa 1:1 da. Horregatik, mihiztadura-lengoaia zuzenki erlazionatuta dago konputagailuaren ezaugarriekin. Mihiztadura-lengoaiako aginduek konputagailuaren baliabideak —erregistroak, memoria, edo sarrera/irteerako gailuak— erabiltzen dituzte zuzenean. Beraz, prozesadore bakoitzak mihiztadura-lengoaia desberdina, berea, erabiltzen du.

Azken helburua makina lengoaian idatzitako programa baten egitura ulertzea den arren, zero eta batekoen bidez programatzea lan gogorra denez, maila altuagoan lan egingo dugu: ARM prozesadorearen mihiztadura lengoaia eta C programazio lengoaia erabiliko ditugu programen exekuzioa behe mailan aztertzeko.

2.1.1 CISC eta RISC diseinu-filosofiak

Makina-lengoaia bat osatzen duten agindu-multzoen diseinuaren eboluzioa konputagailuen eboluzio bera izan da. 1955-1975 urteetan, konputagailuen agindu-multzoak, eta, horrekin batera, hardwarea, gero eta konplexuagoak ziren. Konputagailu horiei **CISC** (*Complex Instruction Set Computer*) deritze. CISC motako konputagailuen ezaugarriak, izenak dioen moduan, hauek dira: agindu-multzoa

konplexua da (formatuak, eragiketa-kodeak, eragiketa motak, helbideratze-moduak, ...), agindu asko dago, eta, ondorioz, hardwarea ere konplexua da (esaterako, hardware berezia eragiketa bereziak egiteko).

70eko hamarkadan, hala ere, aldaketa handi bat gertatu zen diseinu-filosofian. Izan ere, aginduen erabilerari buruzko datu estatistiko zehatzak erabili ziren diseinu-erabakiak hartzeko. Bi puntutan oinarritu zen diseinu-filosofiaren aldaketa:

- Prozesadorea eraikitzeko ditugun baliabideak (transistoreak, finean) mugatuak dira. Beraz, erabaki behar da zertarako erabili halako transistoreak eta zertarako ez, helburu bakar batekin: eraginkortasuna, hots, programen exekuzio-abiadura ahalik eta handiena izatea.

- Agindu konplexuak oso gutxitan erabiltzen dira, eta, oro har, oso kasu berezietarako. Problema bat planteatzen du horrek. Izan ere, programatzaileek goi-mailako lengoaietan idazten dituzte haien programak; gero, konpiladore batek itzuliko ditu agindu horiek makina-lengoaiara. Konpiladoreari dagokio, beraz, prozesadorearen baliabideak ustiatzea, ahalik eta modurik eraginkorrenean, hots makina-lengoaiako agindurik egokienak aukeratzea goi-mailako lengoaien egiturak itzultzeko. Baina asko zailtzen da lan automatiko hori konputagailuaren agindu-multzoa, eragiketak eta helbideratze-moduak oso konplexuak direnean. Ondorioz, konpiladoreak ez dira gauza, kasu askotan, halako agindu konplexuak modu egokian erabiltzeko.

Hausnarketa horren ildoan sortu ziren **RISC** (*Reduced Instruction Set Computer*) konputagailuak. Diseinu-filosofia berriaren leloa hau zen: optimizatu eta erabili baliabideak gehien erabiltzen diren ataletan, eta hartu erabakiak erabilerari buruzko estatistiketan oinarrituta; "konplexutasuna ez da errenta". Oso denbora gutxitan, filosofia berria inposatu zen, eta, harrezkero, konputagailu guztiak hala diseinatu dira.

Beraz, RISC motako konputagailu batean, agindu-multzoa txikia da, formatuak sinpleak dira (aginduen eta eragiketa-kodeen luzera finkoa da edo aukera gutxi dago), helbideratze-moduak bakan batzuk eta sinpleak dira (esaterako: indexatua eta erlatiboa), eta abar.

Ildo berean, konputagailu horien hardwarea ere sinpleagoa da. Oro har, RRR motako konputagailuak dira, hots, eragiketa aritmetikoak erregistroetako datuekin egiten dira, eta emaitzak ere erregistroetan gordetzen dira. Hori dela eta, erregistro orokor asko erabiltzen dira. Eskuarki, memoriako bi agindu bakarrik definitzen dira, *load* eta *store*. Helburua garbia da: programen exekuzio-denbora ahalik eta txikiena izatea.

Aginduak sinpleak direla eta, RISC motako agindu baten exekuzio-denbora CISC motako agindu batena baino txikiagoa da. Baina, bestalde, algoritmo jakin bat programatzeko, aginduak sinpleagoak direnez, agindu gehiago erabili beharko da RISC kasuan CISC kasuan baino; hots, programak zertxobait luzeagoak izango dira.

Adibidez, bi memoria-posizioen arteko trukaketa agindu bakar batekin egin daiteke CISC konputagailu batean, baina hiruzpalau agindu erabili behar dira RISC konputagailu batean:

CISC	RISC

`swap A, B`	`ld r1, A`
	`ld r2, B`
	`st B, r1`
	`st A, r2`

Beraz, zenbait eragiketak edo funtziok denbora gutxiago beharko dute, agian, CISC prozesadore batean RISC prozesadore batean baino. Dena den, hori ez da nahikoa eragiketa hori agindu-multzoan sartzeko. Izan ere, honako bi datuak kontuan hartu behar ditugu: zenbat aldiz erabiliko da eragiketa

berezi hori ohiko programetan? eta, zer ondorio jasan beharko dituzte gainerako aginduek agindu berri hori gehitzeagatik?

Lehen aipatu den moduan, gaurko prozesadore gehienak RISC motakoak dira. ARM prozesadorea ere RISC motako prozesadore bat da. Hala ere, egungo prozesadoreen agindu-multzoak ez dira hasierako RISCenak bezain sinpleak. Gero eta transistore gehiago integratzen dira txip batean prozesadoreak eraikitzeko (200 milioitik gora dagoeneko); beraz, baliabideak ugariak dira, eta, pixkanaka-pixkanaka, agindu-multzoak konplexuago bihurtu dira (ez, hala ere, jatorrizko CISC makinakoak bezainbeste). Edonola ere, filosofiari eutsi zaio: gastatu baliabideak gehien erabiliko diren ataletan, eta hartu erabakiak erabilerari buruzko analisi kuantitatiboetan oinarrituta.

2.1.2 ARM Prozesadorea

Adibide moduan, ikasturtean zehar, prozesadore erreal baten arkitektura eta mihiztadura lengoaia erabiliko ditugu, ARM prozesadorearenak hain zuzen.

ARM (Advanced RISC Machine) arkitektura 1985. urtean sortu zuen *Acorn Computer Group*-ek. Komertzializatu zen lehen bertsioak ARM2 izena jaso zuen eta 1986. urtean merkaturatu zen. Hortik aurrera ARM prozesadoreen familia oso bat garatu da.

Guk aztertuko dugun prozesadorea ARM9-a da, Nintendo DS makinak erabiltzen duen prozesadoreetako bat. Prozesadore hau ARMv4T familiakoa da:

- 32 biteko RISC prozesadore bat da.
- Memoriako posizioak byte batetakoak dira (byterako helbideratzea). Memoriako helbideak 32 bitekoak dira $[0..2^{32}-1]$.
- 16 erregistro orokor dauzka 32 bitekoak, `r0`-tik `r15`-era zenbatuak. `r0`-tik `r12`-ra bitarteko erregistroak errregistro orokorrak dira eta gainerakoak helburu bereziko erregistroak. Adibidez, `r15` erregistroak exekutatu behar den hurrengo aginduaren helbidea gordetzen du beti (hau da, PC erregistro bezala erabiltzen da).
- Exekutatzen ari den programaren egoera gordetzen duen erregistro bat dauka (*current program status register* `CPSR`) baldintzak adierazten dituzten 4 bitekin (4 flag: **N**egatibo, **Z**ero, **C**arry eta o**V**erflow) eta prozesadorearen exekuzio egoera adierazten duten 4 eremurekin.
- Datuak 8 bitekoak (byte), 16 bitekoak (half-word) edo 32 bitekoak (word) izan daitezke. Byte segida bat ere izan daitezke.
- Agindu guztiak luzera berekoak dira: 32 bit.
- Zenbaki osoak birako osagarrian adierazten dira.

2.1.3 Programa baten egitura ARM mihiztadura lengoaian

2.1.1 irudian ARM mihiztadura lengoaian idatzitako programa baten egitura ikus daiteke. Programakoak diren aginduak eta aldagaien definizioak bereiztu ditzakegu. Aldagaiak 8 bit (byte), 16 bit (half-word) edo 32 bitekoak (word) izan daitezke. Byte segida bat bezala ere har daitezke.

Oharrak programako edozein puntutan jar daitezke, horretarako @ ikurra erabiltzen da: ohar bat lerro bukaera arte @ ikurraren atzetik datorren testua izango da.

@ oharrak	@ Adibidea
.code 32 .global main, __cpsr_mask	.code 32 .global main, __cpsr_mask
. . . programako aldagaiak . . .	X: .word 15 Y: .word 10 batu: .word 0

main:
.
programa nagusiko
aginduak
.
.
mov pc, lr
	main:
	ldr r2, X
	ldr r3, Y
	add r5,r2,r3
	str r5, batu
	
	mov pc, lr @ programaren bukaera

2.1.1irudia. ARM mihiztadura lengoaian idatzitako programa baten egitura.

2.1.4 ARM mihiztadura lengoaiako aginduak

ARM mihiztadura lengoaiako agindu batzuk aztertuko ditugu atal honetan. Ez ditugu aztertuko mihiztadura lengoaia honek dituen agindu guztiak, bakarrik gure programak idazteko behar ditugunak.

2.1.4.1 Memoria atzitzeko aginduak

Erregistro eta memoriaren artean datu transferentziak egiteko erabiltzen diren aginduek 1, 2, edo 4 byteko blokeak mugitzeko aukera ematen dute.

<eragiketa>{bald}{tamaina} Rx, <helbidea>

- <eragiketa> izan daiteke:
  - LDR erregistro batean memoriako balio bat kargatzeko.
  - STR memorian gordetzeko erregistro baten balioa.
- Rx: erregistro orokor bat da.
- <helbidea>: memoriako helbide bat adierazten du. Helbide hau adierazteko modu desberdinak daude, eta modu hauei, *helbideratze moduak* deitzen zaie. Memoriako helbide bat adierazteko modurik sinpleena helbide horretan gordeta dagoen aldagaiaren izena erabiltzea da.
- {bald} eta {tamaina} aukerazko eremuak dira. ARMko aginduek baldintza bat ezartzeko aukera ematen dute, baina momentuz alde batetara utziko dugu aukera hau programak ulergarriagoak izateko. Tamainak, datuaren tamaina adierazten du. Besterik adierazi ezean, lehenetsitako aukera, 32 biteko datuak erabiltzea da, baina 8 (LDR**B**) eta 16 bitekoak (LDR**H**) ere izan daitezke datuak, ondorengo adibideetan ikusten den bezala.

2.1.1 Adibidea

```
LDR  r0, X        @ r0 erregistroan X aldagaiaren balioa gorde (32 bit)
STR  r3, Batu     @ Batu aldagaian r3 erregistroaren balioa gorde (32 bit)
LDRB r1, DB       @ r1 erregistroan DB aldagaiaren balioa gorde (8 bit)
LDRH r2, DH       @ r2 erregistroan DH aldagaiaren balioa gorde (16 bit)
```

2.1.4.2 Agindu aritmetikoak

Agindu aritmetikoek bi eragigaien arteko eragiketa aritmetikoak egiten dituzte.

`<eragiketa>{bald}{S}{tamaina}  Rh,Ri, <er2>`

Lehenengo eragigaia (Rh, helburu erregistroa) erregistro orokor bat izango da eta bertan eragiketaren emaitza gordeko da. Eragiketa aritmetikoa iturburu erregistroan (Ri) dagoen datuaren eta bigarren eragigaiaren (er2) artean burutzen da.

- <eragiketa> izan daiteke:

  ADD, SUB, MUL batu, kendu, biderkatu

- <er2> Agindu askotan erabiltzen da bigarren eragigaia adierazteko modu hau. Erregistro orokor bat (Rx) edo berehalako datu bat (#konstante) izan daiteke.

  Salbuespen bezala, mul aginduan bigarren eragigaia ezin daiteke berehalako bat izan, bakarrik erregistro bat izan daiteke.

  Gainera, eragigai hau erregistro bat denean (Rx) aginduan bertan adierazten den desplazamendu bat ezar dakioke (ikusi desplazamenduak burutzeko aginduak). Rx erregistroaren edukia nahiz ezkerrera nahiz eskuinera desplazatu daiteke. Desplazatuko den posizio kopurua berehalako batekin edo beste erregistro baten bidez adierazi daiteke.

- {S} aukerazko eremua da. Eremu hau agertzen denean zenbait flag-en balioak aldatzen dira eragiketaren emaitzaren arabera.

  ADDS, SUBS, MULS N eta Z flag-ak aldatzen dira

2.1.2 Adibidea

```
ADD  r0, r1, #4          @ r0 = r1 + 1
SUB  r3, r2, r1          @ r3 = r2 + r1
ADDS r4, r4, r1          @ r4 = r4 + r1 Z eta N aldatzen ditu r4-ren arabera
ADD r5, r3, r2, lsl #5     @ r5 = r3 + r2 * 2^5
ADD r7, r3, r1, lsl r3     @ r7 = r3 + r1 * 2^r3
```

- **Berehalakoak**

  Berehalako eragigai baten bidez balio konstante bat adierazten dugu aginduan. Berehalakoak 12 bitean adierazten dira, ondorengo egiturari jarraituz:

  - IR (4 bit): bere balioa bider 2 egin ondoren lortzen den zenbakiak IN eremua eskuinera zenbat bit errotatu behar den adierazten du.
  - IN (8 bit): balio bat adierazten du, eskuinera errotatuko dena IR*2 posizio.

  Bitak horrela banatzeko ideia 12 bitekin adieraz daitezkeenak baino zenbaki handiagoak adieraztea da. Modu honetara, 32 bit behar dituzten zenbakiak adieraztera irits daiteke 12 bitekin, gainerako 20 bitak aginduko gainerako eremuak kodetzeko behar baitira. Hala ere, garbi izan behar da 12 bitekin 4096 zenbaki desberdin adieraz daitezkeela, eta ez 32 bitekin sor daitezkeen 4 mila miloi konbinazio desberdinak.

  Hala ere, ARM prozesadoreko diseinatzaileek adierazten dute metodo honekin ohikoenak diren balio azpimultzoak adieraz daitezkeela. Ondokoak dira azpimultzo hauen zenbait adibide:

IR	*IN*	*balioa*
0000	00000010	= 0x0000 0002
0100	00111011	= 0x3B00 0000
1001	00111011	= 0x000E C000

Lehenengo adibidean desplazatu (errotatu) gabeko 8 biteko balio bat daukagu. 8 biteko edozein balio adierazi dezakegu 32 bitean (pisu handieneko 24 bitek 0 balioa izango dute).

Bigarren adibidean 8 biteko zenbaki bat (0x3B) 8 bit edo posizio desplazatu (errotatu) da eskuinera. Errotazio hau egitean ezkerretik ateratzen diren pisu handieneko bitak eskuinetik sartzen dira berriro pisu txikieneko bitetan. Eragiketa hauek erabilita zenbaki oso handiak adieraz daitezke nahiz eta 8 bit jarraien balioa bakarrik finkatu daitekeen.

Hirugarren adibidean bigarren adibideko 8 bit berdinak agertzen zaizkigu, oraingoan 18 biteko errotazio batekin.

Orokorrean kodetu daitezkeen zenbaki batzuk daude eta kodetu ezin diren beste batzuk. Adibidez, 512 balioa kodetu daiteke horrela baina ez 511 balioa. 12 bitekin kodetu ezin daitekeen berehalako bat jartzen badugu gure programan konpiladoreak errore mezu bat aterako du. Kasu honetan berehalakoa lortzeko beste moduren bat erabili beharko dugu, adibidez 511 adierazteko ondokoa egin dezakegu:

```
mov r0, #512
sub r0, #1
```

Hala ere badago 32 biteko edozein zenbaki adierazteko aukera bat hurrengo agindua erabiliz:

```
ldr  r0, =-27
```

Kasu honetan, zenbakia 12 bitean adieraz badaiteke konpiladoreak agindu hau mov r0, #-27 aginduagatik ordezkatuko du. Bestelakoan balioa memorian gordeko du, eta ez aginduan bertan, eta beraz, kasu honetan ezin da esan eragigai hori aginduan bertan dagoen berehalako bat denik.

2.1.4.3 Eragiketa logikoak

Bi eragigaien arteko eragiketa logikoak burutzeko aginduak dira.

```
<eragiketa>{bald}{S}{tamaina}      Rh,Ri, <er2>
```

Lehenengo eragigaia (Rh, helburu erregistroa) erregistro orokor bat izango da eta bertan eragiketaren emaitza gordeko da. Eragiketa aritmetikoa iturburu erregistroan (Ri) dagoen datuaren eta bigarren eragigaiaren (er2) artean burutzen da.

- <eragiketa> izan daiteke:

  AND – AND logikoa Rh := Ri AND er2

  ORR – OR logikoa Rh := Ri OR er2

  EOR – OR esklusiboa Rh := Ri EOR er2

- ANDS, ORRS, EORS eragiketek **gainera** N eta Z flag-ak aldatzen dituzte.

2.1.3 Adibidea

```
AND r0, r1, #0x00000004   @ r0 erregistroko bit guztiak 0 balioa hartuko
                          @ dute 2 bitak ezik, zeinek r1 erregistroan duen
                          @ balioa hartuko duen.
EOR r3, r2, #0XFFFFFFFF   @ r3 = r2-ren baterako osagarria
ORR r4, r4, #0x00000001   @ r4-ko 1 bita 0ra jartzen du
```

2.1.4.4 Datu mugimendurako aginduak

Agindu hauek erregistro orokorren arteko datu mugimenduak burutzen dituzte. Lehenengo eragigaian gordetzen da bigarren eragigaiaren balioa.

<eragiketa>{bald}{S}{tamaina} Rh, <er2>

- <eragiketa> izan daiteke:

MOV	Rh := er2
MOVN	Rh := not(er2)

2.1.4 Adibidea

```
MOV  r0, #45      @ r0 erregistrora 45 balioa mugitzen du
MOV  r3, r2       @ r2 erregistroaren balioa r3 erregistrora mugitzen du
MOVN r4, #8       @ 8 balioa bitez bit ezeztatu eta r4 erregistrora mugitu
```

2.1.4.5 Desplazamenduak

Desplazamenduak maiz erabiltzen dira zenbakiak 2ren berredurengatik biderkatu edo zatitzeko. Zenbaki naturalez ari garenean, nahikoa da desplazamenduak burutu eta 0z osatzea bitak desplazatzean sortzen diren hutsuneak. Zenbaki osoen kasuan berriz, birako osagarrian adierazita badaude, ezkerretara desplazatzean 0rokoak sartzen dira eskuinetik eta eskuinera desplazatzean zeinu bita kopiatzen da ezkerreko bit berrietan.

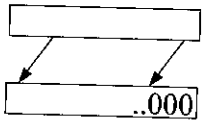
<eragiketa>{bald}{S}{tamaina} Rh,Ri, <er2>

Agindu hauetan , <er2> eragigaiak Ri erregistroaren edukia zenbat bitetan desplazatu behar den adierazten du. Emaitza berriz, Rh erregistroan gordetzen da.

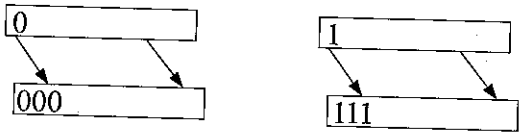
- <eragiketa> izan daiteke:

LSL	Rh := Ri << er2
ASR	Rh := Ri >> er2
LSR	Rh := Ri >> er2

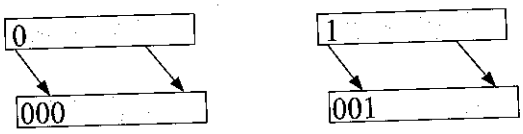
LSL – Desplazamendu logikoa ezkerrera (*Logical Shift Left*). 2ren berredurengatik biderkatzearen parekoa da (<< eragilea C programazio lengoaian).



ASR – Desplazamendu aritmetikoa eskuinera (*Arithmetic Shift Right*). 2ren berredurengatik zatitzearen parekoa da (>> eragilea C programazio lengoaian).



LSR – Desplazamendu logikoa eskuinera (*Logic Shift Right*). Pisu handieneko bitetan sartzen den balioa desplazamendua egitean beti 0 da.



2.1.5 Adibidea

```
LSL r1, r0, #1          @ r1=r0*2
ASR r2, r2, #2          @ r2/4
LSL r4, r4, #4          @ r4=r4*16
```

2.1.4.6 Konparaketak

CMP konparaketa aginduak kenketa bat egiten du Rn eta <er2> eragigaien artean eta ez du emaitzarik gordetzen. Kenketa horren helburua CPSR erregistroko flag-ak aldatzea da.

```
<eragiketa>{bald}      Rn, <er2>
```

2.1.6 Adibidea

```
CMP  r0, #45        @ Ez du r0 aldatzen, N eta Z flag-ak aldatzen ditu
CMP  r3, r2         @ Ez ditu r3 eta r2 aldatzen, N eta Z flag-ak aldatzen
                    @ ditu (kenketaren emaitzaren arabera)
```

2.1.4.7 Jauzi aginduak

Programa baten agindu-fluxua sekuentziala da eskuarki; hots, *i* aginduaren ondoren, *i*+1 agindua exekutatzen da. Aginduen exekuzio-sekuentzia hautsi nahi bada, agindu bereziak erabili behar dira: **jauzi-aginduak**. Jauzi agindu baten exekuzioak PCaren balioa aldatzen du aginduen exekuzio sekuentzian aldaketa bat behartuz. Goi-mailako lengoaietan ohikoak diren **kontrol-egiturak** (if, for, while, ...) jauzi-aginduen bidez gauzatzen dira behe-mailako lengoaietan.

Jauziak **baldintza gabekoak** edo **baldintzapekoak** izan daitezke. Baldintza gabeko jauzia beti betetzen da. Baldintzapeko jauzia, aldiz, bakarrik gauzatzen da baldintza jakin bat betetzen bada, eta hori jakiteko baldintza erregistroko flagak aztertzen dira. Flag hauek aurretik egindako konparaketa baten ondorioz alda daitezke, nahiz eta beste eragiketa mota batzuk ere balio hauek alda ditzaketen. Baldintza betetzen ez bada, jauzia ez da gauzatuko, eta exekuzio sekuentzialarekin jarraituko da. Beraz, jauzi-agindu batek aukera ematen du programaren exekuzio sekuentzialarekin jarraitzeko, edo programaren beste puntu batera jauzi egiteko, han dagoen aginduaren exekuzioarekin jarraitzeko.

Jauzi bat gertatzen bada, exekutatuko den hurrengo agindua ez da jauzi aginduaren ondoren idatzita dagoena. Jauzi-aginduan bertan adierazten da zein litzatekeen exekutatu beharreko hurrengo aginduaren helbidea jauzia beteko balitz; helbide hori etiketa baten bidez identifikatzen da. Etiketek, beraz, jauzien helburuak markatzen dituzte (nora joan behar den), eta karaktere alfanumerikoez osatzen dira (lehenengo karakterea, alfabetikoa, eta, azkena, ":" karakterea). Agindu honen formatu orokorra ondokoa da:

```
B{L}{bald} <etiketa>
```

- {L}: aukera hau erabili edo ez erabiltzearen arabera agindua B (branch) edo BL (branch and link) motakoa izango da. BL motako aginduak itzuleradun jauziak dira

eta azpirrutineri deiak egiteko erabiltzen dira, hurrengo atalean ikusiko dugun bezala. Momentuz bakarrik B motako aginduak erabiliko ditugu.

- {bald}: baldintzapeko jauzietan baldintza adierazteko eremua.
- <etiqueta>: jauzia nora burutuko den adierazteko.

Baldintza gabeko jauzia:

Horrelako agindu bat exekutatzen den guztietan, exekutatuko den hurrengo agindua etiketak identifikatzen duena da.

Eragiketa kodea	Eragigaiak	
B	etiketa	PC :=PC + despl

Baldintzapeko jauziak:

Jauzi agindu hauetan jauzia gauzatzeko bete behar den baldintza zein den adierazten da. Baldintza betetzen den jakiteko flagak aztertu behar dira. Baldintza betetzen bada, jauzia gertatzen da. Hau da, PC-aren balioa aldatzen da exekutatzen den hurrengo agindua etiketak adierazitakoa izan dadin. Baldintza betetzen ez bada PCa ez da aldatzen eta sekuentzian hurrengo dagoen agindua exekutatuko da.

Eragiketa kodea	Eragigaiak	Baldintza
Beq	etiketa	Berdin. flag: Z = 1
Bne	etiketa	Desberdin. flag: Z = 0
Blt	etiketa	Txikiagoa. flag: N = 1
Ble	etiketa	Txikiagoa edo berdina. flag: N = 1 edo Z =1
Bgt	etiketa	Handiagoa. flag: N = 0 eta Z =0
Bge	etiketa	Handiagoa edo berdina. flag: N = 0

2.1.4.8 Programen bukaera

Programa nagusia bukatzeko ondoko agindua erabiliko dugu:

```
mov pc, lr
```

Agindu hau exekutatzean programaren exekuzioa bukatuko da sistemara jauzi bat eginez. Agindu hau ez bada jartzen exekuzio errore bat gertatuko da.

2.1.5 Bektoreen tratamendua ARM mihiztadura lengoaian.

Orain arte datuei erreferentzia egiteko 3 modu (3 helbideratze modu) ikusi ditugu: berehalakoa (eragigaia aginduan bertan adierazitakoa da, eta aginduan bertan dagoenez ez da ez memoria ez erregistro multzoa atzitu behar eragigaia lortzeko), absolutua (datua aginduko eragigaiak adierazten duen memoria helbidean dago, eta beraz, memoria irakurri behar da datua lortzeko) eta erregistro bidezko zuzenekoa (datua aginduan eragigai bezala adierazi den erregistroan dago eta beraz, erregistro hori irakurri beharko da datua lortzeko).

2.1.6 Adibidea

```
ldr r2, X     @ erregistro bidezko zuzenekoa(r2) eta absolutua(X)
sub r5,r2,r3 @ 3 erregistro bidezko zuzenekoak
mov r4, #5    @ erregistro bidezko zuzenekoa(r4) eta berehalakoa(#5)
```

Bektore batetako elementuak nola atzitu ikasi behar dugu orain. Goi mailako lengoaia batean bektore batetako `i` osagaia atzitzeko, bektorearen izena eta atzitzea nahi dugun osagaiaren indizea, `i`, adierazten dira. Adibidez, C programazioa lengoaian B bektoreko `i` osagaia atzitzea nahi badugu, `B[i]` idatziko dugu. Hau da, bektorearen hasierako helbidea eta indizea adierazi behar dira. Mihiztadura lengoaian ere bi gauza horiek adierazi beharko dira: bektorearen hasierako helbidea eta indizearen balioa.

Gogoratu, n izanda bektore baten osagai kopurua, indizeak 0-tik `n-1`-era doazela. Datuaren tamaina ere garrantzitsua da. Zenbaki osoen bektore bat badaukagu, `.word` (32 bit) motako datuekin, bektorearen osagai bakoitzak memoriako 4 posizio okupatuko ditu (helbideratze unitatea bytea da) eta beraz, indizea bider 4 (bider datu tamaina orokorrean) egin beharko da atzitzea nahi den bektoreko elementuaren memoriako helbidea lortzeko. Orokorrean, bektore batetako i osagaiaren helbidea lortzeko (@B[i]) bektorearen hasierako helbideari (@B) desplazamendu bat batu beharko zaio i indizea kontuan hartuta:

```
@B[i] = @B + i*l
```

non `l` bektoreko osagai batek okupatzen duen memoriako posizio kopurua den.

ARM mihiztadura lengoaian bektoreak atzitzeko **oinarri-erregistro + indize erregistro ([rb+rx])** bezala ezagutzen den helbideratze modua erabiliko dugu. Helbideratze modu hau memoria atzitzeko dauden aginduetan erabil daiteke: `ldr` eta `str`. Atzitu behar den bektoreko osagaiaren memoriako helbidea kalkulatzeko `rb` (oinarri-erregistroa, bektorearen hasierako helbidea gorde beharko da hemen) erregistroaren edukiari `rx` (indize-erregistroa, atzitzea nahi den osagaiaren indizea kontuan hartuz bere hasierako helbidearekiko desplazamendua gorde beharko da honetan, i*l) erregistroaren edukia batzen zaio

Helbideratze modu hau erabiltzeko, erregistro batean aldagai baten memoriako helbidea gordetzen duen agindu bat behar dugu. Bi agindu desberdin dauzkagu horretarako:

`ldr r1,=B` edo **`adr r1,B`**

Agindu hauek ez dute memoria atzitzen. Ikusi hurrengo aginduen arteko desberdintasuna:

```
ldr  r1,B      @ r1-en B aldagaiaren edukia edo balioa gordetzen du
ldr  r1,=B     @ r1-en B aldagaiaren helbidea gordetzen du
```

2.1.7 Adibidea

3 osagaitako bektore bat daukagu memoriako 100 helbidetik aurrera gordetzen dena azpiko irudian ikusten den bezala. Bektoreko osagaiak `.word` motakoak dira eta beraz, 4 bytekoak. Irudian ikusten den bezala bektoreko osagaien helbideak 100, 104 eta 108 dira helbideratze unitatea bytea baita eta bektoreko osagaiak 4 bytekoak.

Bektoreko osagaiak atzitzeko oinarri-erregistro + indize-erregistro helbideratze modua erabiliz, lehenengo bektorearen hasierako helbidea lortu behar dugu eta hori oinarri-erregistro lana egingo duen erregistro batean gorde.

```
adr r1,B      @r1 = 100
```

Beste erregistro batek indize erregistro lanak egingo ditu, eta bertan atzitzea nahi dugun bektorearen osagaiari dagokion desplazamendua, bytetan, gorde beharko dugu, bektorearen osagai horri dagokion memoriako helbidea kalkulatzeko erabiliko dena.

```
@ B[0] = @B + 0 * 4 = 100 + 0 = 100 (desplazamendua 0)
@ B[1] = @B + 1 * 4 = 100 + 1 * 4 = 104 (desplazamendua 4)
@ B[2] = @B + 2 * 4 = 100 + 2 * 4 = 108 (desplazamendua 8)
```

Beraz, ondoren ikusten den bezala bektoreko 3 osagaiak atzitu ditzakegu

```
B:      .word 3, 5, 8

        adr r1, B     @r1 oinarri-erregistroa da
        mov r2, #0    @r2 indize erregistroa da
        ldr r3, [r1,r2]      r3=M[r1+r2] r3=M[100] r3=3
        add r2, r2, #4       r2=4
        ldr r4, [r1,r2]      r4=M[r1+r2] r4=M[100 + 4] r4=5
        add r2, r2, #4       r2=8
        ldr r5, [r1,r2]      r5=M[r1+r2] r5=M[100 + 8] r5=8
```

	Memoria				(@)
B[0]	00	00	00	3	100
B[1]	00	00	00	5	104
B[2]	00	00	00	8	108

2.1.8 Adibidea

Bektore batetako ondoz ondoko bi elementuren batura kalkulatzen duen programa idatzi nahi da:

```
batura = bek[i] + bek[i+1]
```

ARM mihiztadura lengoaian:

```
.code 32
.global main, __cpsr_mask

bek: .word -1,4,6,-9,78,12,-34,0,-1,612
i: .word 4
batura: .word 0

main:

adr r1, bek              @ r1= bektorearen hasierako helbidea
ldr r2, i                @ r2=indizea
lsl r4,r2,#2             @ desplazamendua ezkerrera = *4
ldr r3,[r1,r4]           @ r3 = bek[i]
add r2,r2, #1
lsl r4,r2,#2             @ desplazamendua ezkerrera = *4
ldr r5,[r1,r4]           @ r5 = bek[i+1]
add r3, r3,r5
str r3, batura

mov pc, lr
```

2.1.7 eta 2.1.8 adibideetan ikus dezakegu bektore bat nola definitu ARM mihiztadura lengoaian. Aldagai arrunt bat bezala definitzen da, hainbat balio emanda balio bakar bat eman beharrean. Balio horietako bakoitza bektoreko osagai baten balioa izango da, eta jarritako balio kopuruak bektorearen osagai kopurua ezartzen du.

2.1.6 Goi mailako kontrol egituren inplementazioa mihiztadura lengoaian

Aurreko atalean adierazi den bezala, goi-mailako lengoaietan ohikoak diren **kontrol-egiturak** (`if`, `for`, `while`, ...) jauzi-aginduen bidez gauzatzen dira behe-mailako lengoaietan. Hurrengo ataletan, kontrol-egitura horiek behe-mailako lengoaietan nola gauzatzen diren erakusten da. Adibide guztietan kodea lehenengo C programazio lengoaian ematen da eta ondoren ARM mihiztadura lengoaian.

2.1.6.1 if...then...else egitura

`if-else` egitura C lengoaian ondokoa da.

```
if (baldintza) {if-eko gorputza }
    else {else-ko gorputza }
```

Giltzak beharrezkoak dira bakarrik agindu batek baino gehiagok osatzen duenean gorputza. `else` adarra hautazkoa da.

2.1.9 Adibidea

`handi` aldagaian `a` eta `b` aldagaien artean handiena gorde nahi da. Horretarako bi zenbakien artean konparaketa bat egin beharko da, eta konparaketa horren emaitzaren arabera `handi` aldagaian `a` edo `b` gordeko da.

C programazio lengoaian:

```
main()        /* bi zenbakiren arteko handiena */
{
     int a=15, b=16, handi;
      if (a > b) handi = a;
            else handi = b;      // handi = (a > b) ? a : b;
}
```

ARM mihiztadura lengoaian:

```
        .code 32
        .global main, __cpsr_mask

A:      .word 15
B:      .word 16
handi:  .word 0

main:
        ldr  r1, A
        ldr  r2, B
        cmp  r1, r2
        ble  else
        str  r1, handi
        b  fin
else:   str  r2, handi
fin:
        mov pc, lr
```

2.1.6.2 for egitura

C lengoaian `for` egitura horrela adierazten da:

```
for (hasieratzea;baldintza;inkrementua)
    {
    ...
    begiztaren gorputza;
    ...
     }
```

2.1.10 Adibidea

Bektore baten osagaien batura kalkulatzen duen programa idatziko dugu.

C programazio lengoaian:

```
main()                    /* bektore baten osagaien batura*/
 {int batu=0;
  int Bek[]={3,2,3,-5,4,-3,6,5,31,-6];
       for (int i=0;i<10;i++)
             batu=batu+Bek[i];
 }
```

`for` egitura mihiztadura lengoaian idazteko, aldagaiaren hasieratzea (`i=0`), begiztaren baldintza (`i<10`) eta iterazio bakoitzean burutu behar den inkrementua (`i++`) hartu behar ditugu kontuan. `for` egituraren baldintza begiztan jarraitzeko baldintza bat da. Mihiztadura lengoaian kontrako baldintza erabiliko dugu begiztatik ateratzeko.

ARM mihiztadura lengoaian:

```
        .code 32
        .global main, __cpsr_mask
Bek:    .word 3, 2, 3, -5, 4, -3, 6, 5, 31, -6
batu:   .word 0

main:
        mov r0, #0              @ batu=0
        adr r1, Bek
        mov r2, #0              @ hasieraketa (i=0)
FOR:    cmp r2, #10
        beq buk                 @ for-eko baldintza, aurkakoa irten
        lsl r4,r2,#2            @ ezkerrera despl., *4
        ldr r3,[r1,r4]          @ r3-n kargatu i elementua
        add r0, r0,r3
        add r2, r2, #1          @ i++
        b   FOR
buk:    str r0, batu

        mov pc, lr
```

2.1.6.3 While egitura

C lengoaian `while` egitura horrela idazten da:

```
while (baldintza)
    {
    ...
    begiztaren gorputza;
    ...
     }
```

2.1.11 Adibidea

Bi zenbaki positiboren zatitzaile komunetako handiena kalkulatzen duen programa idatziko dugu.

C programazio lengoaian:

```
main()                  /* zatitzaile komunetako handiena */
 {int X=12, Y=18, zkh;
       while (X!=Y)
         if (X>Y) X- =Y;
              else Y- = X;
       zkh=Y;
 }
```

Aurreko kasuan bezala `while`-eko baldintza begiztan jarraitzeko baldintza bat da. Mihiztadura lengoaian baldintza horren aurkakoa erabiliko da begiztatik ateratzeko.

ARM mihiztadura lengoaian:

```
          .code 32
          .global main, __cpsr_mask
x:        .word 12
y:        .word 18
zkh:      .word 0

main:
          ldr r1, x
          ldr r2, y
while:    cmp r1, r2                  @ whileko baldintza, aurkakoa irteteko
          beq buk                     @ if-eko baldintza
          blt txikia
          sub r1, r1, r2
          b while
txikia:   sub r2, r2, r1
          b while
buk:      str r1, zkh

          mov pc, lr
```

2.1.6.4 do...while (repeat) egitura

Kontrol-egitura hau `while` egituraren antzekoa da baina kasu honetan baldintza begiztaren bukaeran aztertzen da eta ez hasieran. C programazio lengoaian horrela idazten da:

```
do {
      ...
   begiztaren gorputza;
   ...
   } while (baldintza);
```

2.1.12 Adibidea

`zenb` aldagai baten kodeketan 1 balioa hartzen duen bit kopurua kontatu eta `batekoak` aldagaian gorde behar da.

C programazio lengoaian:

```
main()        /* 1 balioa duten bitak kontatu*/
     {int batekoak=0, bitak=32;
      int ema, zenb=40;
            do {
                  ema=zenb & 1;
                  if (ema!=0) batekoak++;
                  zenb>>=1;
                  bitak- -;
            } while (bitak>0);
     }
```

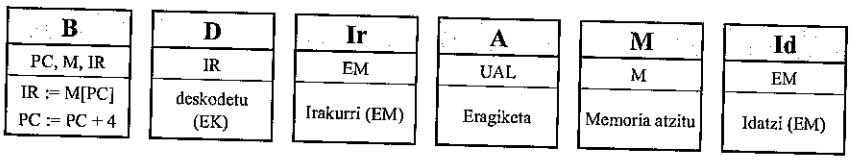
ARM mihiztadura lengoaian:

```
.code 32
.global main, __cpsr_mask

zenb:        .word -27
batekoak:    .word 0

main:
        ldr r1, zenb
        mov r2, #0
        mov r3,#32
repeat: and r4, r1 ,#0x80000000
        cmp r4, #0
        beq zero                          @ ez da beharrezkoa ands erabiliz
        add r2,r2,#1
zero:   lsl r1,r1,#1
        subs r3, r3, #1
        bne repeat
        str r2, batekoak

        mov pc, lr
```

repeat egiturak izan dezakeen abantaila bat *for* edo *while* egiturekin alderatuz, baldintza bukaeran aztertzeak aginduren bat aurrezteko aukera ematen duela da. Hala ere, kasu honetan begiztako gorputza beti gutxienez behin exekutatuko da eta horren ondorioz egitura ezin daiteke edozein problema ebazteko erabili.

2.1.7 Aginduen exekuzio-faseak

Programa bat agindu-segida bat besterik ez da, eta agindu bakoitzaren exekuzioa urrats edo faseetan banatu ohi da. ARM konputagailuan, aginduen exekuzioa sei urratsetan banatu da:



1 Aginduaren bilaketa, B

Programa bat osatzen duten aginduak memorian daude, eskuarki ondoz ondoko posiziotan. Beraz, agindu bat exekutatu ahal izateko, aurretik, haren helbidea zehaztu, eta dagokion memoria-posiziotik irakurri behar da (ingelesez, *fetch* deritzo eragiketa horri).

Hori egin ahal izateko, helburu bereziko erregistroetako bat erabiltzen da, **programaren kontagailua** (ingelesez, *Program Counter*, **PC**). Uneoro, programaren kontagailuak adierazten

du exekutatu behar den aginduaren helbidea. Ondorioz, agindu bat eskuratzeko, PCaren edukia helbide-busean jarri behar da, memoriako helbide horren (memoria-posizio horren) edukia irakurtzeko. Memorian eskuratu den informazioa, agindu bat, datu-busetik bideratzen da prozesadorerantz, eta CPUko beste erregistro berezi batean gordeko da: **agindu-erregistroan**, hain zuzen ere (ingelesez, *Instruction Register*, **IR**).

Agindua irakurrita, programaren kontagailua egokitu behar da, exekutatu behar den hurrengo agindua "erakuts" dezan. Beraz, urrats honen amaieran, PCaren edukia eguneratzen da

2 **Deskodeketa, D**

Aginduan dagoen informazioa kodetuta dago. Agindu bat irakurri ondoren, kontrol-unitateak deskodetu (interpretatu) behar du, zer egin behar den jakiteko (batuketa, kenketa, ...). Horretarako, aginduaren eremu jakin bat, **eragiketa-kodea (EK)**, aztertuko da, eta, horren arabera, beste osagaiek behar dituzten kontrol-seinaleak sortuko dira (erregistroak kargatu, UALeko eragiketa, memorian irakurri edo idatzi, etab.).

3 **Eragigaien irakurketa, Ir**

Gorago aipatu den bezala, datu jakinen gainean exekutatzen dira aginduak; aginduak berak adieraziko du non dauden datuak. Oro har, hiru aukera ditugu: aginduan bertan daude, erregistro-multzoan, edo memoria nagusian. Aginduarekin batera irakurri badira, orduan ez da ezer egin behar datuak lortzeko, IR erregistroan kargatu baitira aginduaren bilaketa fasean. Erregistro-multzoan badaude, EMa irakurri behar da, une honetan, eragigaiak lortzeko. Azkenik, eragigaia memorian badago, prozesua konplexuagoa da; memoria irakurri baino lehen, datuaren helbidea kalkulatu behar da. Hainbat aukera badago ere, oro har, erregistro baten edukia erabiltzen da datuaren helbidea kalkulatzeko. Hori dela eta, kasu horretan ere EMa irakurri behar da.

Laburbilduz, beharrezkoa denean, urrats honetan erregistro-multzoko erregistroak irakurtzen dira, behar diren eragigaiak hor daudelako edo memoriako helbide bat kalkulatu behar delako.

4 **Eragiketa aritmetiko/logikoa, A**

Urrats honetan egiten da aginduak adierazten duen eragiketa; horretarako, prozesadoreko unitate aritmetiko/logikoa erabiltzen da. Bestalde, zenbait kasutan, batuketa bat egin behar da eragigaien memoria-helbidea lortzeko; batuketa hori ere, urrats honetan egiten da.

5 **Memoriako atzipena, M**

Memoria erabili behar bada, datu bat irakurtzeko edo emaitzaren bat idazteko, urrats honetan erabiltzen da, aurreko urratsetan lortutako helbidea erabiliz. Bi atzipen mota daude: irakurketa, datu bat eskuratzeko, eta idazketa, memorian datu bat gordetzeko.

6 **Emaitzen idazketa, Id**

Urrats hau erabiltzen da aurreko faseetan lortutako emaitza erregistro multzoko erregistro batean gordetzeko; aginduan bertan adierazten den erregistroan, hain zuzen ere.

Agindu orokor baten exekuzioan bereiz daitezkeen urratsak edo faseak dira aurrekoak. Hala ere, oso eginkizun desberdineko aginduak exekutatzen ditu konputagailu batek; hori dela eta, agindu guztiak ez dira aurreko fase guztietatik igarotzen. Era berean, fase bakoitzeko exekuzio-denbora desberdina da; esaterako, askoz denbora gehiago behar da memoria irakurtzeko edo idazteko, agindua deskodetzeko edo erregistro-multzoa irakurtzeko baino. Ondorioz, aginduen exekuzio-denbora desberdina izango da, haien konplexutasunaren arabera.

2.1.13 Adibidea

Adibide moduan, ondoko agindu sekuentziaren exekuzioa aztertuko dugu.

```
.code 32
.global main, __cpsr_mask

a1:     .word -27
a2:     .word 0

main:
mov r1,#3          {3 balioa r1 erregistrora mugitu)
ldr r2, a1         {memoriako 0 helbideko edukia r2-n gorde)
add r4, r2, r1     {r2 eta r1-en edukia batu eta emaitza r4-n gorde)
str r4, a2         {r4-ren edukia memoriako 4 helbidean gorde}
mov pc, lr
```

Programa hau, goi mailako lengoaian idatzitako ondoko aginduaren parekoa da `a1=a2 + 3`. Suposatu aldagaiak memoriako 0 helbidetik aurrera gordetan daudela.

- movi r1, #3

B	D	A	Id
PC, M, IR	IR	UAL	EM
IR := M[PC] PC := PC + 4	deskodetu (EK)	mugitu (#3)	r1

- `ld r2, @0  (ldr r1, a1)`

B	D	M	Id
PC, M, IR	IR	M (*read*)	EM
IR := M[PC] PC := PC + 4	deskodetu (EK)	M[@0]	r2

- `add r3, r1, r2`

B	D	Ir	A	Id
PC, M, IR	IR	EM	UAL	EM
IR := M[PC] PC := PC + 4	deskodetu (EK)	r1, r2	batu	r3

- `st r3, @1  (str r3, a2)`

B	D	Ir	M
PC, M, IR	IR	EM	M (*write*)
IR := M[PC] PC := PC + 4	deskodetu (EK)	r3	M[@1]

2.1.8 Little-endian eta big-endian

Konputagailu gehienen helbideratze unitatea bytea da baina byte baino handiagoak diren datuak erabiltzen dituzte. Analiza dezagun adibide hau: badaukagu byteka helbideratzen den memoria bat (hots, memoria-posizioak byte batekoak dira), eta 4 byteko datu bat. Jakina, datu horrek 4 posizio okupatuko ditu memorian; baina, memorian gorde behar denean, zein ordenatan gordeko dira 4 byte horiek? Hau da, zein byte gordeko da helbide baxueneko posizioan eta zein altuenekoan? Izan ere, bi

aukera ditugu datua gordetzeko: atzetik aurrera edo aurretik atzera. Bi ordena horiek ***little-endian*** eta ***big-endian*** deitzen dira.

Lehendabizikoak, *little-endian*, pisu gutxieneko bytea kokatzen du memoriako helbide baxuenean, eta hortik aurrera gainerakoak. *Big-endian* ordenak, aldiz, datuaren pisu handieneko bytea kokatzen du helbide baxuenean.

Adibidez: datua = 00 03 A1 0F (4 byte, hamaseitarrez)

Memoriako helbideak	*i*	*i+1*	*i+2*	*i+3*
Little-endian	0F	A1	03	00
Big-endian	00	03	A1	0F

Hori bai, ordena bera erabiltzen da, beti, byte baten bitak adierazteko: pisu handieneko bita ezkerrean, eta pisu txikieneko bita eskuinean.

Byteen ordena memorian ez da arazoa konputagailu jakin batean; bai, aldiz, datu horiek konputagailu batetik bestera pasatu beharko balira, zuzenean; bi konputagailuetan byteen ordena desberdina bada, kontuz ibili beharko da datuen jatorrizko esanahia mantentzeko.

Hainbat mikroprozesadorek bi formatuekin lan egin dezakete (ARM, PowerPC, DEC Alpha, PA-RISC, MIPS) eta batzutan ondorengo izenekin ikus ditzakegu: *mixed-endian* edo *middle-endian*.

2.1.9 Programen mihiztatzea, estekatzea eta karga

Programa bat idatzi ondoren, hainbat prozesu bete behar dira programa hori exekutatu ahal izateko. Batetik, mihiztadura-lengoaiaz idatzita dagoen programa makina-lengoaiara itzuli behar da (konputagailuak ulertzen duen lengoaia bakarra). Gero, programa osatzen duten modulu[1] guztiak estekatu (elkartu, *to link*) behar dira; azkenik, programa exekutatzeko, memoria nagusian kargatu behar da. Atal honetan, prozesu horiek aztertuko ditugu.

2.1.9.1 Programa mihiztatzailea

Konputagailuek makina-lengoaian dauden programak bakarrik exekutatzen dituzte. Beraz, mihiztadura-lengoaiaz zein goi-mailako lengoaiaz idatzita dauden programak makina-lengoaiara itzuli behar dira. Mihiztadura-lengoaiaz idatzita dauden programak itzultzeko, **mihiztatzaile** izeneko programa berezia erabili behar da (goi-mailako lengoaiak itzultzeko, konpiladoreak erabiltzen diren moduan). Mihiztadura-lengoaia bakoitzari mihiztatzaile desberdina dagokio.

Programa mihiztatzaileak, beraz, makina-kodera itzultzen ditu mihiztadura-lengoaiaz idatzitako agindu eta datuak. Itzulpen-prozesuan zehar, aldagaien eta aginduen helbideak kalkulatu behar dira, aldagaien hasierako balioak esleitu, etiketek adierazten dituzten helbideak lortu, eta abar. Oro har, itzulpena bi fase edo pasalditan egiten da, hau da, bi aldiz analizatzen da programa. Prozesua ulertzeko, azter dezagun nola egiten den modulu edo fitxategi bakarreko programa baten mihiztatze-prozesua. 2.1.2 irudian, prozesuaren eskema orokorra ageri da.

[1] Programa bat hainbat modulutan (fitxategitan) bana daiteke. Modulu guztiak itzuli behar dira makina-lengoaiara, kontuan harturik erreferentzia egingo diotela elkarri.

2.1.9.2 Lehenengo pasaldia

Lehenengo fasean, programaren aginduak irakurri ahala, banan-banan, sintaxia egiaztatzen da, eta moduluari (fitxategiari) dagokion **sinbolo-taula** sortzen. Taula horretan gordetzen dira: programan agertzen diren sinboloak (aldagaien izenak, etiketak, ...), dagozkien helbide erlatiboak (programaren hasierarekiko), eta aldagaien hasiera-balioak. Bi atal bereizten dira sinbolo-taulan: datuei dagokiena (aldagaiak), eta etiketei dagokiena. Horrez gain, moduluak behar duen memoria kopurua ere erabakitzen da lehen pasaldian

Oro har, mihiztatzaileak ezin izango ditu etiketa guztiak itzuli lehenengo pasaldian (ezin izango die helbide bat esleitu); Hain zuzen ere, bere definizioa baino lehen ageri bada etiketa bat (adibidez, jauzi bat non jauzi behar den lekua bere etiketarekin beherago dagoen) ez dakigu zein den adierazten duen helbidea. Horregatik, sinbolo-taulan idazten dira sinbolo horiek, besterik gabe, eta bigarren pasaldirako uzten da haien ebazpena.

2.1.9.3 Bigarren pasaldia

Lehenengo pasaldia egin ondoren, mihiztatzailea prest dago **objektu-kodea** sortzeko, hots, jatorrizko programa makina-lengoaiaz, bitarrez, emateko. Izan ere, kode hori lehenengo fasean ere sor daiteke agindu askotarako; ez, ordea, guztietarako.

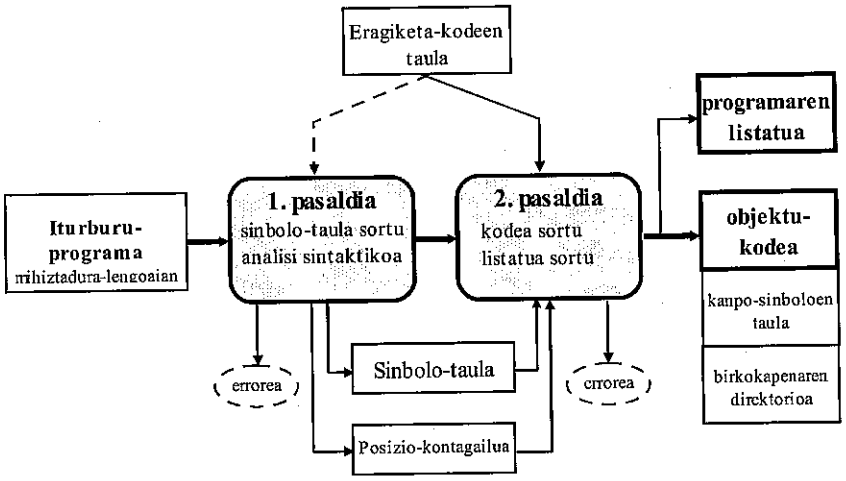
Beraz, mihiztaketaren bigarren fasean, aginduak berrirakurtzen dira eta dagokien makina-kodea sortzen da, 1ekoen eta 0koen segidak, hain zuen ere (eragiketa-kodeak, eragigaiak, helbideak, ...). Eragiketa-kodea itzultzeko, aurredefinitutako taula bat erabiltzen da; taula horretan daude mihiztadura-lengoaiako mnemoteknikoei (`add`, `load`, ...) dagozkien kode bitarrak. Gero, eragigaiak adierazi behar dira, bitarrez. Hiru aukera besterik ez dago: berehalako datu bat (arazorik ez bitarrez adierazteko); erregistro bat (haren kode bitarra eman behar da); edota memoria-helbide bat (helbidea absolutua bada, nahikoa da bitarrez ematea; aldagai baten izena bada, lehen fasean sorturiko sinbolo-taulan egongo da haren helbidea). Hala, agindu bakoitzari dagokion makina-kodea, kode bitarra sortu da.

Tartean, kode-taulan ez dagoen agindu bat edo sinbolo-taulan ez dagoen sinbolo bat detektatzen bada, erroretzat jo, abisu bat eman, eta prozesua bertan behera utziko da.

Bigarren pasaldian sortzen den objektu-moduluak honako informazio hau jasoko du: (a) moduluaren izena eta luzera; (b) **sarrera-puntua** edo programaren lehenengo aginduaren helbidea; (c) **kanpo-erreferentziak**, hau da, beste objektu-modulutan definituta dauden aldagaien edo errutinen erreferentziak; (d) programaren datuak, datu-blokearen 0 helbidetik aurrera, eta (e) makina-lengoaiako aginduak, agindu-blokearen 0 helbidetik aurrera. Hau da, programa exekutatzeko behar den informazio guztia.

Gainera, objektu-programa **birkokagarria** da, hau da, programaren aginduei ez zaizkie memoria-posizio finkoak esleitu, erlatiboak baizik. Izan ere, programa memorian kargatzen denean erabakiko dira helbide fisiko absolutuak. Gauza bera gertatzen da helbide bat adierazten duten eragigaiekin (helbideratze absolutua zein indexatua erabiltzen dutenak).

Lehenengo pasaldiak sinbolo-taula eta behin-behineko listatu bat sortzen ditu, non agindu batzuk dagoeneko itzuli baitira. Gainerako aginduak bigarren pasaldian itzultzen dira, aurreko faseko sinbolo-taula erabiliz, eta, hala, behin-betiko listatua sortuko du. Listatu horretan, datuen zein aginduen helbideak erlatiboak dira, dagozkien blokeekiko.



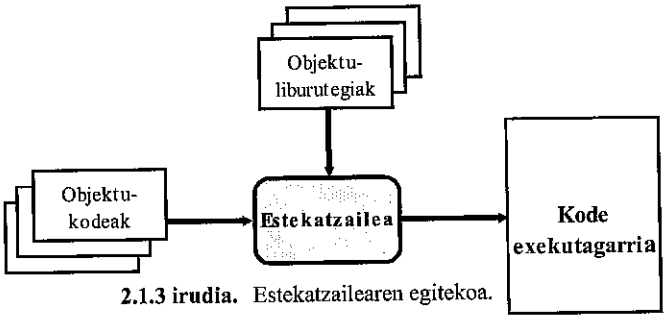
2.7.1 **2.1.2 irudia.** Mihiztatze-prozesua bi pasalditan.

2.1.8.4 Programa estekatzailea *(linker)*

Oro har, hainbat errutinatan edo azpiprogramatan banatzen dira programak. Fitxategi desberdinetan, hots, iturburu-modulu desberdinetan egon ohi dira halako errutinak, eta banaka mihiztatzen dira; ondorioz, fitxategi desberdinetan izango dira dagozkien objektu-moduluak. Beraz, ohikoa da, modulu baten barruan, beste modulu edo fitxategietan dauden errutinak erreferentziatzea, eta, halaber, errutina horiek erabiltzea haien moduluetan definituta ez dauden aldagai globalak. **Kanpo-erreferentzien** arazoa deritzo horri.

Programatzaileen azpirrutinez gain, ohikoa da, eta lagungarria, aurredefinitutako errutinak (matematikakoak, gailuak kontrolatzekoak, ...) erabiltzea programak osatzeko. Errutina komun horiek liburutegietan biltzen dira, programatzaileek erabil ditzaten behin eta berriz idatzi behar izan gabe. Programatzaileen lana errazteaz gain, programazio-erroreak gutxitzen ditu liburutegiak erabiltzeak, eta kode optimizatuagoa sortzen du. **Objektu-liburutegiak** deritze liburutegi horiei. Aurreko errutinak bezala, errutina horiek ere ez daude definituta erabiltzailearen fitxategiaren barruan.

Estekatzailearen egitekoa, hain zuzen ere, hainbat objektu-modulutatik abiatuta modulu exekutagarri bakarra sortzea da. Hori lortzeko, bi lan bete behar ditu. Batetik, modulu bakoitzeko erreferentziei dagozkien helbide erlatibo berriak sortu, modulu bakarraren hasiera-helbidea kontuan harturik; eta, bestetik, estekatu behar dituen moduluen kanpo-erreferentziak ebatzi. 2.1.3 irudian ageri da estekatzailearen egitekoa.



2.1.3 irudia. Estekatzailearen egitekoa.

2.1.8.5 Programa kargatzailea

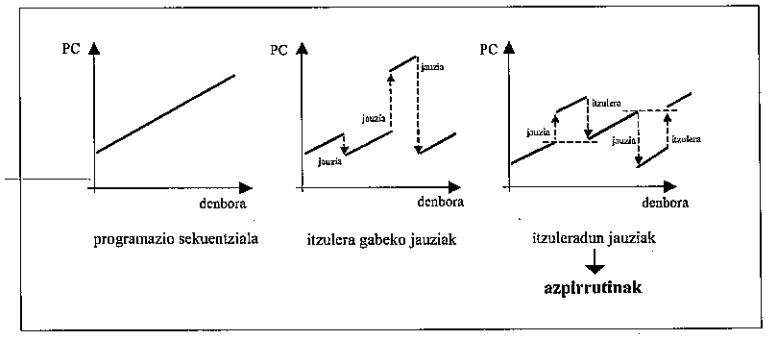
Azkenik, programa bat mihiztatu eta estekatu ondoren, memorian kargatu eta abiarazi behar da. **Kargatzaileak** betetzen du ataza hori. Kanpo-memoria batean (disko batean, esaterako) dagoen modulu exekutagarria hartu, eta memoria nagusian kopiatuko du; moduluaren hasiera-helbidea kargatuko du PC erregistroan, eta, azkenik, helbideratze absolutua edo indexatua erabiltzen den kasuetan, benetako helbide fisikoak kalkulatuko ditu.

Azkenik, programa abiaraziko du, lehenengo agindua —sarrera-puntukoa— exekutatuz.

2.2 AZPIRRUTINAK

2.2.1 Sarrera

Aurreko kapituluetan azaldu den moduan, programa baten aginduak ondoz ondoko helbideetan gordetzen dira memorian. Hori dela eta, programak sekuentzialki exekutatzen dira, hots, agindu bat exekutatu ondoren memoriako hurrengo agindua exekutatuko da. Aginduen exekuzio-ordena edo fluxua aldatu ahal izateko, PCaren sekuentzia eteten duten aginduak erabili behar dira: jauzi-aginduak, hain zuzen ere. 2.2.1 irudian ageri den moduan, bi motako jauziak daude: itzulera gabekoa eta itzuleraduna. Orain arte ikusi ditugun jauziak —`b`, `beq`, ...— itzulera gabekoak dira, programa ez baita itzuliko jauzia egin den puntura. Aldiz, jauzi itzuleradunak exekutatzen direnean —`bl`, `bleq`, `call` ...—, hainbat agindu exekutatu ondoren, programak berreskuratuko du aginduen jatorrizko fluxua (gorde baita jauziaren ondorengo aginduaren helbidea). Kapitulu honen helburua jauzi itzuleradunak lantzea da, hots, azpirrutinak aztertzea.



2.2.1 irudia. Itzulera gabeko jauziak eta itzuleradunak.

Goi-mailako programazio-lengoaiek funtzioak eta prozedurak erabiltzen dituzten modu berean, mihiztadura-lengoaiek azpirrutinak (azpiprogramak) erabiltzen dituzte. Lan jakin bat burutzen duen makina-lengoaiako agindu-multzoa da azpirrutina bat, eta programa nagusiaren edo azpirrutina baten edozein puntutatik aktiba, dei, edo exekutarazi daiteke, baita azpirrutinatik bertatik ere. Zentzu horretan, bi **aktibazio mota** daude: kanpo-aktibazioa —programa nagusitik edo beste azpirrutina batetik aktibatzen denean— eta barne-aktibazioa —azpirrutinatik bertatik aktibatzen denean—.

Azpirrutinak erabiltzeak baditu abantailak eta desabantailak. Abantailen artean, hauek dira aipagarrienak:

- Programak toki gutxiago behar du memorian. Izan ere, azpirrutinari dagokion kodea behin bakarrik ageri da programan (memorian), nahiz eta programa deitzailearen puntu askotatik aktiba daitekeen; azpirrutinen kodea berrerabili egiten da.
- Programaren kodeak egitura argiagoa du. Ondorioz, programa zuzenak lortu arte ezinbestekoak diren zuzenketak, aldaketak eta antzekoak egitea errazagoa da. Horrez gain, programa-liburutegiak erabiltzeko aukera dago.

Baina azpirrutinak erabiltzeak baditu desabantailak ere:

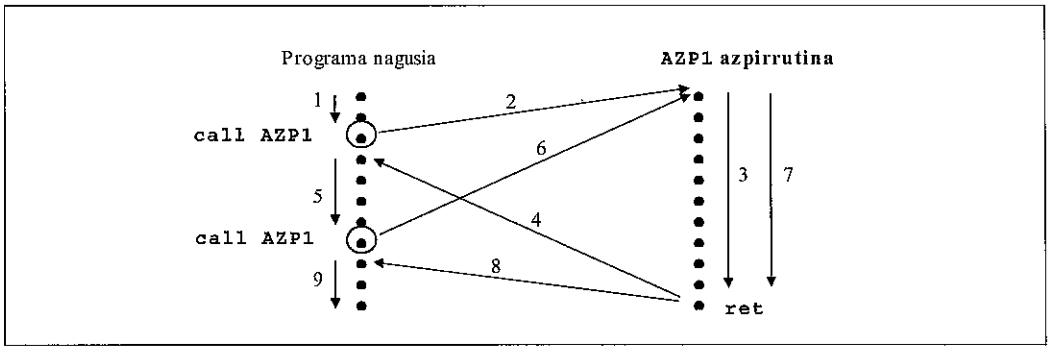
- Programen exekuzio-denbora hazten da, agindu gehiago exekutatu behar baitira: azpirrutinak deitzeko, programa deitzailera itzultzeko, eta deitzen duen programaren eta azpirrutinaren arteko informazio-trukea egiteko (parametroak pasatu eta emaitza itzuli).

- Prozesadorearen konplexutasuna areagotzen da, hardware berezia gehitu behar baita azpirrutinen kudeaketa eraginkorra izan dadin. Izan ere, programen exekuzioak analizaturik, egiaztatu da azpirrutinen kontrolarekin erlazionatutako kodea (deiak eta itzulerak), batez bestean, aginduen %3-%10 tartean dagoela. Hala izanik, azpirrutinen erabilera eraginkorra izateko, garrantzitsua izango da hardwareak eskain dezakeen edozein laguntza.

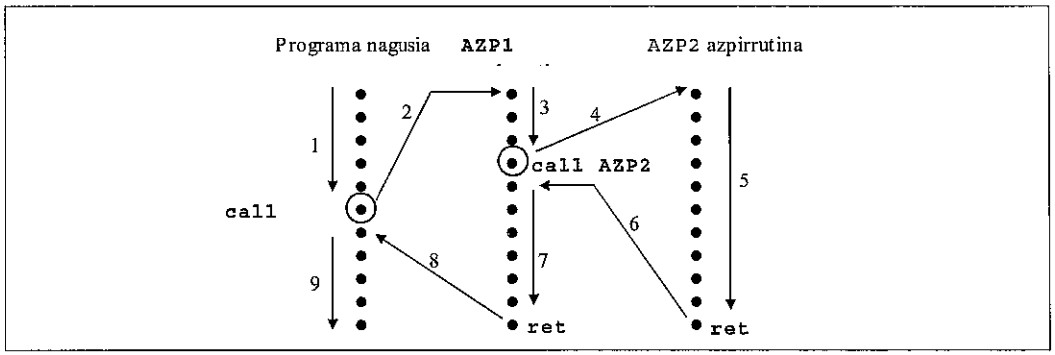
2.2.2 Azpirrutinen sailkapena

Azpirrutinen exekuzioaren analisia egin baino lehen, ikus dezagun azpirrutinen sailkapen nagusia:

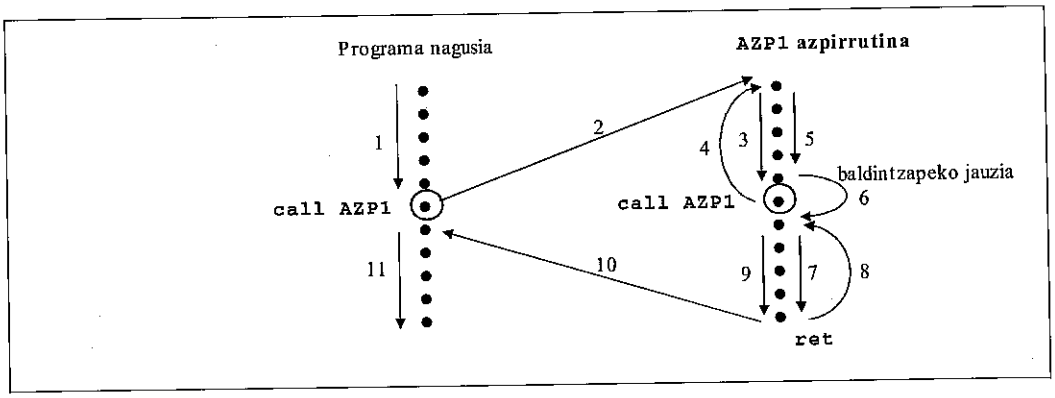
- **Maila bakarrekoak**: azpirrutinaren barnean ez da beste azpirrutinarik aktibatzen. 2.2.2 irudian ageri da adibide bat: azpirrutina bat bi aldiz deitzen da programa nagusitik. Irudian ageri diren zenbakiek exekutatze-ordena denboran zehar adierazten dute.
- **Maila anitzekoak**: azpirrutinaren barnean beste azpirrutina batzuk aktibatzen dira (2.2.3 irudia).
- **Errekurtsiboak**: azpirrutinak bere buruari dei egiten dio (2.2.4 irudia) zuzenean, edo, agian, beste azpirrutina baten bidez (zeharka).



2.2.2 irudia. Maila bakarreko azpirrutina.



2.2.3 irudia. Maila anitzeko azpirrutina.



2.2.4 irudia. Azpirrutina errekurtsiboa.

Bestalde, **erabileraren** arabera, beste sailkapen hau egin daiteke:

- **Ez-berrabiagarriak**: ezin dira berriz aktibatu aurreko aktibazioaren exekuzioa bukatu artean.
- **Berrabiagarriak**: bigarren mota honetan, onartzen da azpirrutina berriz aktibatzea, nahiz eta aurreko aktibazioa oraindik bukatu gabe egon. Halako azpirrutinak oso erabilgarriak dira, esaterako, sistema eragileetarako zein konpiladoreetarako konputagailu multiprogramatuetan (aldi berean programa bat baino gehiago exekutatzea onartzen duten sistemak).

  Azpirrutina berrabiagarriek baldintza batzuk bete behar dituzte: (a) azpirrutinak ezin du bere kodea aldatu exekuzioaren ondorioz (ohiko ezaugarria egungo edozein programatan); eta (b) azpirrutinaren datuetarako behar den memoria-eremuak desberdina izan behar du exekuzio batetik bestera.

2.2.3 Azpirrutinen exekuzioaren analisia

Azpirrutina bat exekutatzeko, pauso hauek jarraitu behar dira: (a) azpirrutinak erabiliko dituen parametroak edo datuak dagokien tokian idatzi; (b) azpirrutinara jauzi, bere kodea exekutatzeko; eta, azkenik, (c) deia egin duen programara itzuli, haren exekuzioarekin jarraitzeko. Azpirrutinaren exekuzioarekin lotutako informazio guztia (parametroak itzulera-helbideak, ...) **aktibazio-bloke** izeneko datu-egituran pilatzen da.

Azpirrutinak exekutatzeko jarraitu beharreko pauso guztiak aztertuko ditugu, zehatz-mehatz, hurrengo ataletan, kasu sinpleenetik konplexuenera.

2.2.3.1 Deia eta itzulera

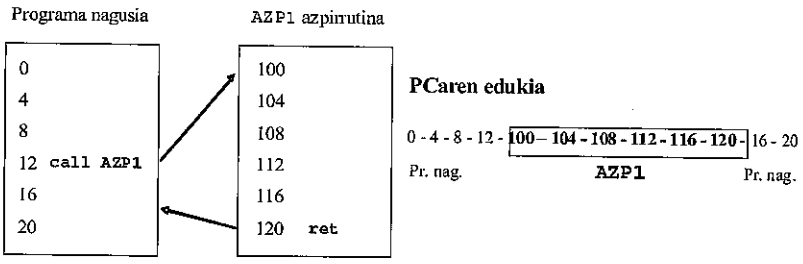
Deia egiteko, hots, azpirrutinaren kodera jauzi egiteko, agindu berezi bat erabiltzen da. Agindu horrek izen desberdina hartzen du mihiztadura lengoaia desberdinetan. Momentuz, modu orokorrean, agindu hori `call azpirrutina_izena` dela suposatuko dugu.

Agindu horren eginkizuna bikoitza da. Batetik, agindu-sekuentzia eteten du, jauzi bat eginez. Horretarako, ohiko `PC := PC + 4` egin beharrean, `PC := PC + despl_azpi` egiten du, non `despl_azpi` desplazamendua `call` aginduan adierazi den azpirrutinara dagoen distantzia baita. PCak adieraziko duen helbide berri horretan hasten da azpirrutinaren kodea. PCa aldatzea da, hain zuzen ere, edozein jauzi-aginduk egiten duena. Baina, bestetik, eta hau da `call` aginduaren

berezitasuna, itzulera-helbidea (@itzulera, ARMaren kasuan `PC_deia + 4`) gordetzen du, azpirrutinaren exekuzioa amaitzean programa nagusira itzuli ahal izateko.

Programa nagusira itzultzeko, beste agindu bat erabili ohi da; **`ret`** da agindu hori. Aginduaren eginkizuna sinplea da: PC erregistroan kargatu behar du `call` aginduak gorde duen itzulera-helbidea (`PC := @itzulera`). ARM prozesadorearen mihiztadura-lengoaian ez dago `ret` motako agindurik, eta beraz, ikusiko dugun moduan, itzulera helbidea zuzenean PC erregistroan kargatu behar da errutina bat bukatzen denean.

Bi agindu horien bidez, 2.2.5 irudian ageri den moduan, exekuzioaren fluxu sekuentziala (PCak hartzen dituen balioak) etetea lortzen da. Adibidean, agindu bakoitzak lau memoria-posizio betetzen ditu.



2.2.5 irudia. Azpirrutina baten exekuzioa: exekutatzen diren aginduen helbideen segida.

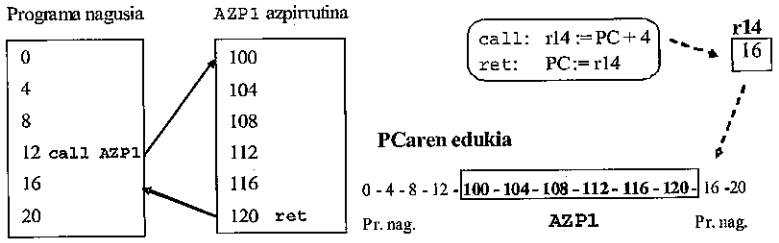
2.2.5 irudiko adibidean, azpirrutina bukatzen denean, PCaren edukia 120tik 16ra aldatu behar da, deia egin duen programara itzultzeko. Itzulera-helbide hori `call` aginduak gorde du, eta horretarako toki jakin bat behar da. Non gorde daiteke itzulera-helbidea? Aukera bat baino gehiago dago:

a. ***Memoria-posizio edo erregistro finko bat azpirrutina guztientzat.***

Kasu honetan itzulera-helbidea beti toki berean gordetzen da, dela erregistro jakin batean dela memoria-posizio jakin batean. Hortik hartuko da itzulera-helbidea, azpirrutina bukatzean, programa nagusira itzultzeko. Erregistro bat erabiltzen denean, prozesua azkarragoa da memoria erabiltzen denean baino.

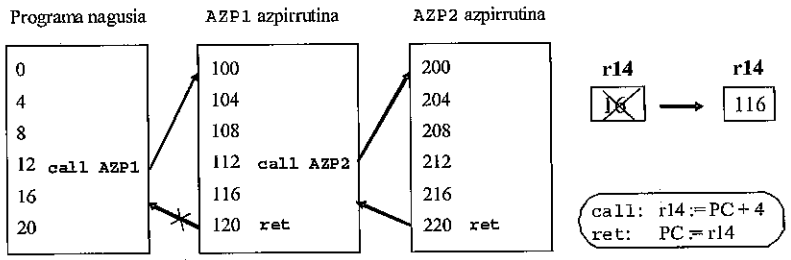
Adibidez, IBM 360-ak r14 erregistroa erabiltzen zuen itzulera-helbidea gordetzeko (2.2.6 irudia). Aukera bera erabiltzen da gaur egun prozesadore askotan (MIPS, PowerPC, ...), halakoek agindu berezi bat erabiltzen baitute, Jump-and-Link (JAL) izenekoa, azpirrutina sinpleak exekutarazteko. Esaterako, MIPS prozesadore-familiak r31 erregistroa erabiltzen du itzulera-helbidea gordetzeko, Jump-and-Link agindua exekutatzen denean. ARM prozesadoreak berriz, Branch and Link (BL) izeneko agindua erabiltzen du beti azpirrutinak exekutatzeko eta itzulera helbidea `r14` (*link register* edo `lr` bezala ezagutzen da erregistro hau) erregistroan gordetzen du.

Erregistro bat erabiltzea itzulera-helbidea gordetzeko ez da soluzio orokorra, ez baitu balio azpirrutina mota guztietarako; hala ere, exekuzioa azkarra denez, oso erabilia da arazorik sortzen ez duten kasuetan.



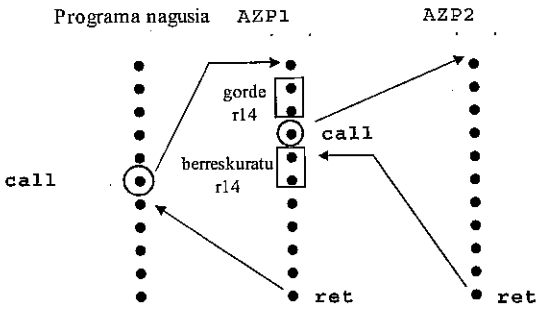
2.2.6 irudia. Itzulera-helbidearen kudeaketa IBM 360-an.

Itzulera-helbidea erregistro batean gordetzeak badu arazo nabarmena maila anitzeko azpirrutinak erabili behar direnean. Azpirrutina bukatu baino lehen beste bat aktibatzen bada, aurrekoaren itzulera-helbidea galduko da, itzulera-helbide berria erregistro berean (adibidean, r14-an) idatziko baita, eta aurrekoa galdu! 2.2.7 irudian ageri da arazo hori: `AZP1` azpirrutina exekutatzera jauzi egin denean gorde den itzulera-helbidea, 16, galdu egin da `AZP2` deitu denean, une horretan bigarren itzulera-helbide bat idatzi baita, 116. Hori dela eta, ezin izango da `AZP1` azpirrutinatik programa nagusira itzuli.



2.2.7 irudia. Maila anitzeko azpirrutinentzat (a) soluzioak duen arazoa.

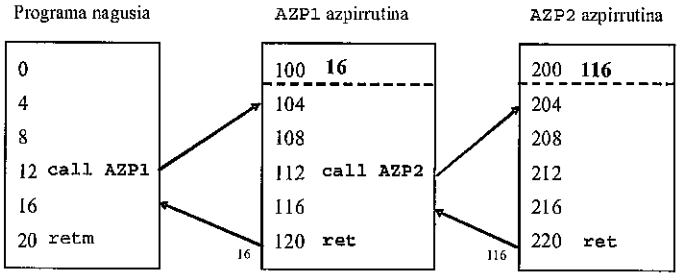
Maila anitzeko azpirrutinak exekutatu ahal izateko, aurreko `call` aginduarekin gorde den itzulera-helbidea lekuren batean (memorian edo beste erregistro batean) salbatu beharko da hurrengo deia egin aurretik; itzuli ondoren, lehenengo itzulera-helbidea berreskuratu beharko da, gorde den tokitik. Arazoa hala konpontzen denean, "software irtenbidea" erabili dela esaten da. IBM 360-an adibidez, 2.2.8 irudian ageri den moduan egiten zen.



2.2.8 irudia. Maila anitzeko azpirrutinen kudeaketa IBM 360 makinan.

b. ***Memoria-posizio (edo erregistro) desberdin bat azpirrutina bakoitzarentzat.***

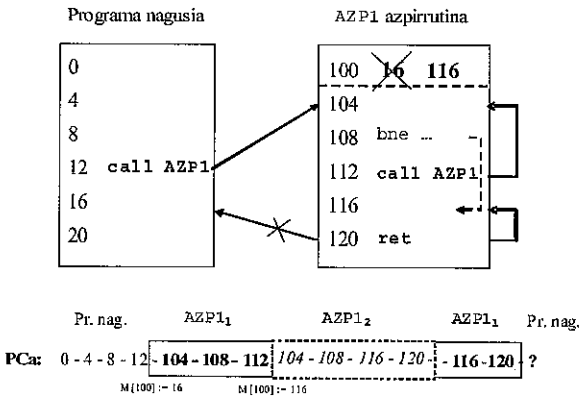
Azpirrutina bakoitzak memoria-posizio edo erregistro finko bat erabiltzen du dagokion itzulera-helbidea gordetzeko. HP 2100-ean adibidez, azpirrutinaren lehenengo memoria-posizioan (kodearen aurrean) gordetzen zen itzulera-helbidea, eta hortik eskuratzen zuen geroago `ret` aginduak. 2.2.9 irudian, estrategia hori erabili da maila anitzeko azpirrutina bat (`AZP1`) exekutatzeko.



2.2.9 irudia. Maila anitzeko azpirrutinen kudeaketa, itzulera-helbidea azpirrutina bakoitzaren hasieran gordetzen denean.

Gauza bera egin daiteke erregistroak erabiliz. PDP-11 konputagailuan, esaterako, azpirrutinaren deian bertan adierazten zen itzulera-helbidea gordetzeko erabili behar zen erregistroa.

Ikusi dugun moduan, metodo honek maila bakarreko eta maila anitzeko azpirrutinak onartzen ditu, baina, hala ere, ezin da erabili azpirrutina errekurtsiboak antolatzeko (azpirrutina errekurtsiboak bere burua deitzen du). Izan ere, dei errekurtsibo bakoitzaren ostean, aurreko deiaren itzulera-helbidea (adibidean, 16) galdu egiten da, memoriako helbide berean idazten baita beti (2.2.10 irudia). Errekurtsibitatea erabili nahi bada, beste teknikaren bat erabili beharko da.

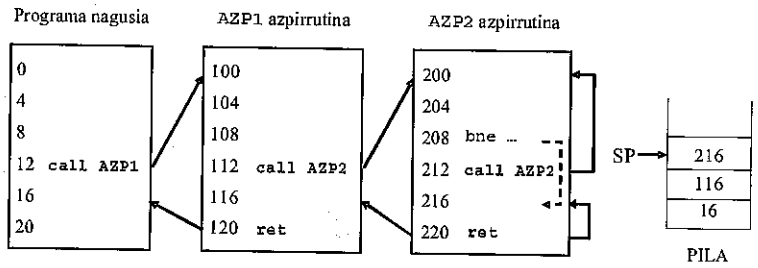


2.2.10 irudia. (b) teknika erabiliz, errekurtsibitatea kudeatzeko arazoak.

c. ***Memoriako pila.***

Pila izeneko datu-egitura, datu-biltegi berezi bat besterik ez da. Eskuarki, memoriako zati bat. Datuak pilan uzten dira eta pilatik hartzen dira, baina biltegi hori modu berezian kudeatzen da, LIFO *(Last In First Out)* moduan: gorde den azkeneko datua da aurrena ateratzen. Memoriako zati horretan, pilan, idazteko edo irakurtzeko, erakusle berezi bat erabiltzen da, SPa (pilaren erakuslea, *Stack Pointer*), eta agindu bereziak ere bai: `push` eta `pop`.

Azpirrutinen itzulera-helbidea gordetzeko hirugarren aukera pila da, hain zuzen ere. Hala egiten denean, azpirrutinak habira daitezke, bata bestearen gainean metatuko baitira itzulera-helbideak pilan, batere informaziorik galdu gabe. SP erakusleak erakutsiko du non gorde helbide horiek. Beraz, maila bakarreko azpirrutinak, maila anitzekoak, eta errekurtsiboak exekuta daitezke arazorik gabe (2.2.11 irudia).



2.2.11 irudia. Itzulera-helbidea pilan gordetzen da.

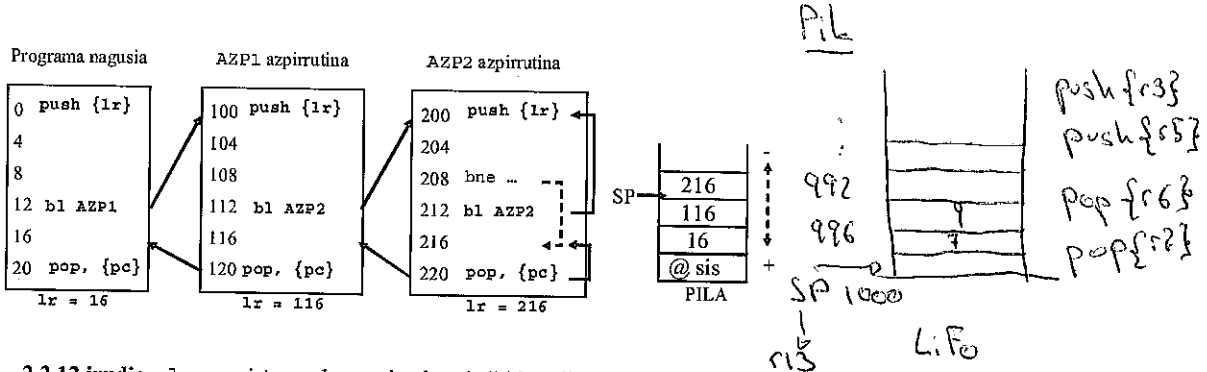
Ia prozesadore guztiek egiten duten moduan, ARM prozesadoreak ere pila erabiltzen du itzulera-helbideak gordetzeko. Hala ere, lehen aipatu den bezala, ARMko mihiztadura lehengoaian `bl` agindua erabiltzen da (itzuleradun jauzi agindua) azpirrutinak deitzeko eta `bl` aginduak itzulera helbidea `lr` erregistroan gordetzen du.

- `bl`, azpirrutina bat deitzeko agindua:

```
bl azpirrutina_izena   ≡   lr = PC_deia + 4
                           {itzulera-helbidea lr erregistroan gordetzen da}
                           PC := @ azpirrutina
                           {azpirrutinaren helbidea PCan kargatzen da}
```

Maila anitzeko eta azpirrutina errekurtsiboen arazoa konpontzeko, azpirrutina baten exekuzioaren hasieran `lr` erregistroa pilan gordetzen da, eta azpirrutinaren exekuzioaren bukaeran bere balioa berreskuratzen da pilatik. `ret` bezalako agindurik ez dagoenez, azpirrutina baten exekuzioa bukatzeko itzulera helbidea berreskuratu eta `PC` erregistroan gorde behar da.

Ikusi 2.2.12 irudian programa nagusiaren hasieran ere `lr` erregistroa pilaratzen dela. Programa nagusia bukatzeko eta sistemari kontrola itzultzeko PCan sistemako itzulera helbidea gorde behar da. Itzulera helbide hori `lr` erregistroan dago, eta programa nagusiak azpirrutina bat deitzen badu, dei horretan `lr` erregistroaren balioa galduko da. Beraz programa nagusian ere `lr` erregistroa pilaratzen da eta bukaeran pilatik berreskuratzen da itzulera helbidea.



2.2.12 irudia. `lr` erregistroan dagoen itzulera-helbidea pilan gordetzen da ARM makinetan.

2.2.12 irudian ikusten den bezala `push` agindua erabiltzen da balioak pilan sartzeko eta `pop` agindua pilatik balioak atera eta berreskuratzeko. ARM makinetan pila helbide txikietarantz hasten da (kontrako aukera ere erabiltzen da beste makina batzutan). `push {lr}` aginduak, lehendabizi `SP`a dekrementatuko du (`SP = SP-4`) eta ondoren `SP`ak adierazitako memoriako helbidean `lr` erregistroaren edukia gordeko du. `pop {pc}` aginduak berriz, pilaren tontorrean dagoena irakurri (`SP`ak adierazitako helbidetik), `PC` erregistroan kargatu, eta ondoren `SP` erregistroa inkrementatuko du (`SP = SP+4`) pilaren tontorreko balioa kendu asmoz.

Bai `push` eta bai `pop` aginduek aldiko erregistro baten balioa baino gehiago idatzi/iraurri ditzakete pilan, ondorengo deskribapenetan ikusten den bezala:

- `push`:

```
push {r1,r2...rn}      ≡     sp = sp - 4;
                             M[sp]= rn;
                             .....
                             sp = sp - 4;
                             M[sp] = r1;
```

- `pop`:

```
pop {r1,r2...rn}       ≡     r1 = M[sp];
                             sp = sp + 4;
                             .....
                             rn = M[sp];
                             sp = sp + 4;
```

ARM mihiztadura lengoaian, `push` eta `pop` eragiketak beste bi agindurekin ere burutu daitezke. ARMak, load/store motako agindu bakarraren bidez, memorian irakurketa edo idazketa bat baino gehiago egiteko aginduak eskaintzen ditu, pilarekin lan egiteko erabilgarriak izan daitezkeenak. Agindu hauek `ldm` eta `stm` izena dute eta ondoan ikusi dezakegu beraien formatua. Agindu hauek `Rn` erregistroak adierazitako memoria helbidetik aurrera egiten dituzte memoriako irakurketak (`ldm`) edo idazketak (`stm`).

`<eragiketa>{bald}{modua} Rn{!}, <erregistro-zerrenda>`

- `<eragiketa>` izan daiteke `LDM` edo `STM`.
- `{modua}` eremuak, `Rn` erregistroa nola eguneratzen den kontrolatzen du eta aukera desberdinak eskaintzen ditu memoriako edozein eremu edo pila atzitzearen arabera:

a.- memoriako edozein eremu atzitzen dugunean:

`<eragiketa>`**`ia`** ondoren inkrementatu

`<eragiketa>`**`ib`** aurretik inkrementatu

`<eragiketa>`**`da`** ondoren dekrementatu

`<eragiketa>`**`db`** aurretik dekrementatu

b.- pilarako bereziak:

`<eragiketa>`**`fd`** ondoren inkrementatu

`<eragiketa>`**`ed`** aurretik inkrementatu

`<eragiketa>`**`fa`** ondoren dekrementatu

`<eragiketa>`**`ea`** aurretik dekrementatu

Kasu honetan **a** letrak adierazten du pila helbide handietarantz hasten dela, **d** letrak helbide txikietarantz hasten dela, **f** (full) letrak SP erakusleak pilan sartu den azken datuaren helbidea duela eta **e** (empty) letrak SP erakusleak pilaren tontorrean dagoen datuaren hurrengo helbidea duela (hutsik dagoen lehen posizioaren helbidea).

- {!} aukera erabiltzen bada berriz, azken helbidea Rn erregistroan idazten da. Ikusi ondoko adibideak:

```
ldmia sp!, {r1, r2}    ≡     ldmfd sp!, {r1, r2}
                       r1 = M[sp];
                       r2 = M[sp+4];
                       sp = sp + 8; (! erabiltzeagatik)
```

Agindu hauek eta pop {r1,r2} guztiz baliokideak dira.

```
stmdb sp!, {r1, r2}    ≡     stmfd sp!, {r1, r2}
                       M[sp-4]= r1;
                       M[sp-8] = r2;
                       sp = sp - 8; (! erabiltzeagatik)
```

Agindu hauek eta push {r1,r2} guztiz baliokideak dira.

Modua adierazten ez bada aginduaren izenean (ldm eta stm aginduak), lehenetsitako aukera **ia** modua da.

ARM prozesadorean r13 erregistroa erabiltzen da SP erakuslea gordetzeko.

2.2.3.2 Azpirrutinetarako parametroak eta azpirrutinen emaitzak

Programa nagusitik azpirrutina bat aktibatu aurretik, azpirrutina horrek behar dituen datuak edo parametroak prestatu behar dira; modu berean, azpirrutina bukatzen denean, emaitzak jaso behar dira programa nagusian. Informazio-truke hori bi modutan egin daiteke:

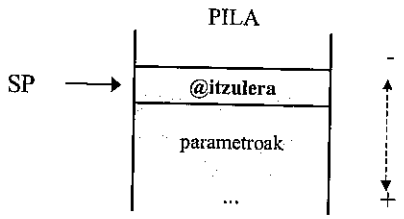
- ***balio*** bidez: aldagaiaren kopia bat egiten da, eskuarki pilan, eta azpirrutinak kopia horrekin lan egiten du; beraz, jatorrizko aldagaia ez da aldatzen.
- ***erreferentzia*** bidez: aldagaiaren helbidea adierazten zaio azpirrutinari; ez da kopiarik egiten, eta, beraz, azpirrutinak alda dezake jatorrizko datua.

ADA programazio-lengoaiaz, adibidez, *in* moduan pasatutako parametroak balio bidez pasatzen dira, eta *in out* edo *out* moduan pasatutakoak, aldiz, erreferentzia bidez. Bektore, matrize eta antzeko datu-egiturak erreferentzia bidez pasatu ohi dira, ez baita batere eraginkorra haien kopia bat egitea pilan (datu asko izan ohi dira). C programazio-lengoaian berriz, parametro bat erreferentzi bidez pasatzeko zuzenean parametro edo aldagaiaren helbidea pasa beharko dugu.

Deia egiten duen programaren eta azpirrutinaren arteko informazio-trukea (parametroak eta emaitzak) biek atzi dezaketen toki jakin batean egin behar da. Aurreko atalean aztertu diren hiru aukerak berak ditugu: (a) memoria-posizio edo erregistro jakinak eta beti berdinak azpirrutina guztientzat; (b) memoria-posizio edo erregistro jakinak azpirrutina bakoitzarentzat; eta (c) pila.

Aipatu berri ditugun arrazoiak direla eta, hirugarren aukera da ohikoena, maila anitzeko azpirrutinak eta errekurtsibitatea erabiltzeko aukera ematen baitu. Azpirrutina batekin lotutako informazio guztia, beraz, pilan metatuko da, eta aktibazio-bloke izeneko datu-egitura osatuko da. Azpirrutina baten exekuzioari buruz orain arte dakiguna kontuan hartuta, aktibazio-blokea 2.2.13 irudian ageri den moduan osatuko da: azpirrutinak behar dituen parametroak kargatu dira pilan, eta, gainean, itzulera-helbidea. SP erakusleak beti pilaren tontorra erakusten du.

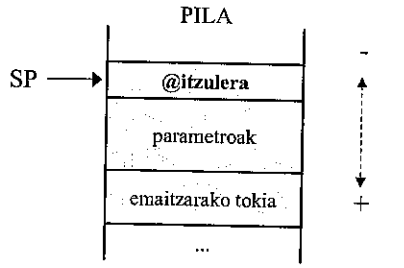
Pilan dauden elementuak irakurtzeko, pilaren erakuslea, SP, erabili behar da, helbideratze erlatiboa erabiliz (ikusi 2.2.1 adibidea).



2.2.13 irudia. Aktibazio-blokea: parametroak eta itzulera-helbidea.

Azpirrutina exekutatutakoan, pilan kargatu diren parametroak "ezabatu"[2] egin behar dira, tokia berreskuratzeko. Lan hori programa deitzaileak berak egin dezake, baina ez da aukera bakarra, itzulera-aginduak berak ere egunera baitezake SP erakuslea. i80x86an, adibidez, `ret n` aginduaren bidez egiten da hori (`n` = parametro kopurua). ARM konputagailuan, lehenengo aukera erabiltzen da, eta, horretarako, `add sp, sp, #parametro_kopurua x 4` agindua erabili behar da (berriz ere suposatuz parametroak 4 byte edo hitz batetakoak direla).

Azpirrutinek emaitzen bat itzultzen duten kasuetan (goi-mailako programazio-lengoaietako funtzioek egin ohi duten moduan), deia egiten duen programak, parametroak pilaratu aurretik, tokia erreserbatu beharko du pilan azpirrutinaren emaitzarako (2.2.14 irudia). Horretarako, `sub sp, sp, #4` agindua erabiliko da (suposatuz emaitza hitz bat eta beraz 4 bytetakoa dela).



2.2.14 irudia. Aktibazio-blokea: funtzioaren emaitza gordetzeko tokia, parametroak, eta itzulera-helbidea.

Azpirrutinaren exekuzioa bukatutakoan, deia egin duen programak parametroak ezabatuko ditu, eta azpirrutinak itzulitako emaitza eskuratu ahal izango du pilatik. Pilatik hitz bat irakurtzeko (eta han desagerrarazteko), `pop` agindua erabiltzen da.

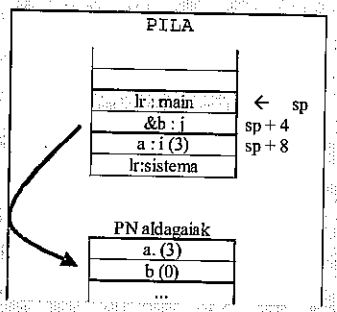
2.2.1 Adibidea

Adibide honetan programa nagusitik dei bat egiten da AZP azpirrutinara bi parametro pasata, bat balio bidez eta bestea erreferentzi bidez.

```
void AZP (int i, int *j)
{
    (*j)= i*i;
}
void main ()
{
    int a=3;
    int b=0;

    AZP(a, &b);
}
```

[2] Pilaren edukia ez da, berez, ezabatzen. Edukia desagerrarazteko, pilaren tontorra markatzen duen erakuslea, SPa, handiagotzen da, besterik ez.

```
.code 32
.global main, __cpsr_mask

main:

        push {lr}   @itzulera helbidea pilan gorde
        ldr r1, a
        adr r0, b   @b-ren helbidea r0-n gorde
        push {r0,r1} @parametroak pilan sartu, r1,r0 ordenan
        bl AZP
        add sp, sp, #8 @parametroak ezabatu pilatik
        pop {pc} @itzulera helbide PCan kargatu

   a: .word 3
   b: .word 0

AZP:
        push {lr}
        ldr r3, [sp, #8] @r3 = i (3)
        ldr r4, [sp, #8] @r4 = i (3)
        mul r5,r3,r4
        ldr r6, [sp, #4] @r6 = @b = j
        str r5, [r6]
        pop {pc}
```



Programa nagusitik bi parametro pasatzen dira AZP azpirrutinara, a balio bidez eta b erreferentzi bidez. b erreferentzi bidez pasatzeko bere helbidea lortu behar dugu eta horretarako erabiltzen da `adr r0, b` agindua. Bi parametro horiek azpirrutinaren exekuzioa bukatu ondoren pilatik kendu behar dira eta horretarako erabiltzen da `add sp, sp, #8` agindua, pilatik bi hitz ezabatzen dituena.

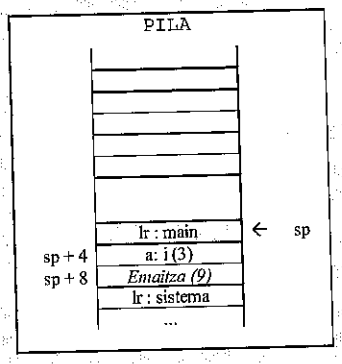
Azpirrutinaren exekuzioan parametroak pilatik irakurtzen dira SParekiko helbideratze erlatiboa erabiliz (adibidez `ldr r3, [sp, #8]` aginduan SPak adierazitako memoriako helbideari 8 batuko zaio azken memoria helbidea kalkulatzeko). Pila helbide txikiagoetarantz hasten denez, SParen azpitik dauden parametroak atzitzeko desplazamendu positiboak erabiltzen dira. Erreferentzi bidez pasatako parametroaren kasuan, kontuan hartu behar da pilan daukaguna parametro horren helbidea dela (@b) eta beraz, pilatik helbide hori irakurri behar dela ondoren helbide horretan, eta beraz, jatorrizko b aldagaian idazteko.

2.2.2 Adibidea

Adibide honetan AZP azpirrutina funtzio bat da eta beraz bere emaitza pilan utziko du.

```
int AZP (int i)
{
    return (i*i);
}
void main ()
{
    int a=3;
    int ema;

    ema=AZP(a);
}
```

```
.code 32
.global main, __cpsr_mask

main:
        push {lr}
        ldr r0, a
        sub sp, sp, #4 @emaitzarako tokia gorde pilan
        push {r0} @parametroa pasa
        bl AZP
        add sp, sp, #4 @parametroa kendu pilatik
        pop {r0} @emaitza jaso pilatik
        str r0, ema
        pop {pc}
   a: .word 3
   ema: .word 0

AZP:
        push {lr}
        ldr r3, [sp, #4]
        ldr r4, [sp, #4]
        mul r5,r3,r4
        str r5, [sp, #8] @emaitza utzi pilan, emaitzarako tokian
        pop {pc}
```



Programa nagusiak pilan parametroak sartu aurretik emaitzarako tokia uzten du SPa dekrementatuz (`sub sp, sp, #4`). Azpirrutinak kalkulatzen duen emaitza hor utziko du, eta programa nagusira bueltatzean honek `pop` agindu baten bidez jasoko du emaitza pilatik (parametroak pilatik kendu ondoren).

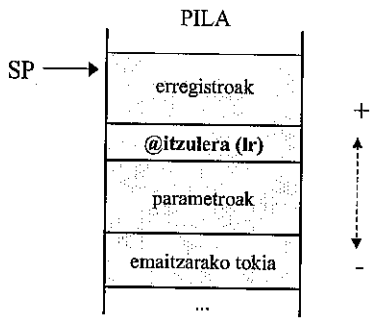
2.2.3.3 Deia egiten duen programaren egoera gordetzea eta berreskuratzea

2.2.1 eta 2.2.2 adibideetan, AZP azpirrutinak ez ditu erabiltzen deia egin duen programak erabilitako erregistroak (`lr` erregistroa ezik). Hori dela eta, ez dago arazorik programa nagusiari berriro ekiten zaionean, berak erabiltzen dituen erregistroak ez baitira aldatu. Hala ere, hori ez da ohikoa, azpirrutinek erregistro-multzoko edozein erregistro erabiltzeko aukera izan behar baitute. Azpirrutinen exekuzioak **gardena** izan behar du programa nagusiarentzat; hots, programa nagusiari berriz ekiten zaionean, haren egoera ere (erregistroen edukia) berreskuratu behar da. Bi aukera daude gardentasuna lortzeko:

1. Azpirrutinaren ardura da programa nagusiko egoera ez aldatzea. Azpirrutinaren exekuzioaren hasieran, bertan erabiliko diren erregistro guztien edukia pilan gordetzen da; modu berean, amaitzean, erregistroen jatorrizko edukia pilatik berreskuratzen da. Hala, azpirrutinak edozein erregistro erabil eta alda dezake, jatorrizko balioak berreskuratuko baitira bukaeran.
2. Deia egiten duen programaren ardura da haren egoera gordetzea. Errutinara jauzi egin aurretik, erabiltzen dituen erregistroak gordetzen ditu pilan, eta exekuzioaren kontrola berreskuratzen duenean, azpirrutina amaitu bezain laster, erregistroen edukia berreskuratuko du.

Hala lehenengo kasuan nola bigarrenean, litekeena da behar diren erregistroak baino gehiago gordetzea pilan: lehenengo kasuan, programa nagusian erabiltzen ez diren erregistroak; eta, bigarrenean, berriz, azpirrutinan aldatuko ez direnak.

Gure programetan, lehenengo aukera erabiliko dugu, eta, beraz, aktibazio-blokea 2.2.15 irudian ageri den moduan geratuko da.

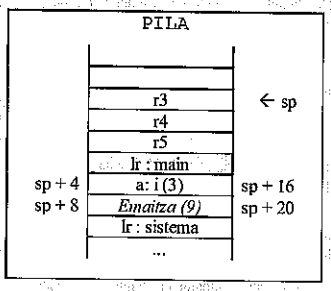


2.2.15 irudia. Aktibazio-blokea: emaitzarako tokia, parametroak, itzulera-helbidea eta deia egiten duen programaren egoera gordetzeko tokia.

2.2.3 Adibidea

Aurreko adibide bera da hau, baina, kasu honetan, azpirrutinaren exekuzioa "gardena" da programa deitzailearentzat.

```
.code 32
.global main, __cpsr_mask

main:

        push {lr}
        ldr r0, a
        sub sp, sp, #4
        push {r0}
        bl AZP
        add sp, sp, #4
        pop {r0}
        str r0, ema
        pop {pc}

   a: .word 0x00000003
 ema: .word 0x00000000

AZP:
        push {r3,r4,r5,lr}
        ldr r3, [sp, #16]
        ldr r4, [sp, #16]
        mul r5,r3,r4
        str r5, [sp, #20]
        pop {r3,r4,r5,pc}
```



Azpirrutinaren exekuzioaren hasieran, azpirrutinak erabiliko dituen erregistroen balioak gordetzen dira (r3, r4, eta r5}, eta bukaeran, berreskuratzen dira. Hori dela eta, edozein erregistro erabili ahal izango du azpirrutinak. Ikusi azpirrutinaren kodean parametroak eta emaitza atzitzeko SParekiko desplazamenduak aldatzen direla 2.2.2 adibidearekiko.

2.2.3.4 Aktibazio-blokearen kudeaketa

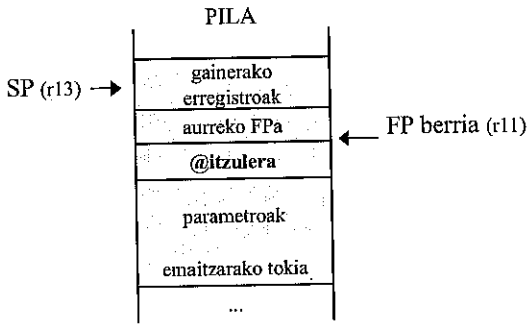
2.2.3 adibidean ageri den moduan, hiru eremu bereizi ditugu aktibazio-blokean: beheko aldean, azpirrutinaren parametroak; ondoren, itzulera-helbidea; azkenik, gainean, salbatu diren erregistroak. Parametro bat eskuratzeko erabili behar den helbidea, beraz, SParekiko desplazamendu bat eginez lortuko da. Gorde den erregistro kopuruaren mende dago desplazamendu hori, eta horrek zailtzen du azpirrutinen eta, oro har, programen garapen-prozesua. Izan ere, programazio-prozesuan zehar,

aldaketa bat dela eta, erregistro gehiago gorde behar bada, desplazamenduak aldatu beharko dira parametroak atzitzen dituzten agindu guztietan.

Aktibazio-blokea atzitu ahal izateko SP erakuslea erabili gabe, prozesadore askok aktibazio-blokerako **erakusle berezi** bat erabiltzen dute: **FPa** (*Frame Pointer*, aktibazio-blokearen erakuslea). Erregistro horrek aktibazio-blokeko posizio finko bat erakusten du azpirrutinaren exekuzio osoan zehar. FP erregistroa azpirrutinaren hasieran ezartzen da, hardware bidez edo software bidez, eta, itzulera arte, edukiari eutsiko dio. Hori dela eta, aktibazio-blokean egin behar diren atzipenak, oraingoz parametroak eskuratzeko edo emaitza uzteko, FParekiko helbideratze erlatiboa erabiliz egingo dira. Hala, posizio bakoitza atzitzeko behar den desplazamendua beti bera izango da azpirrutinaren exekuzioaren zehar.

Zenbait prozesadoretan, FP erregistroa makinaren erregistro zehatz bat da (BP erregistroa, adibidez, i80x86 prozesadoreetan). ARM prozesadorearen kasuan, **r11** erregistroa erabiliko dugu FParena egiteko. Helburu orokorreko erregistro bat erabiliko dugunez, FP (r11) erregistroaren edukia ere pilan gorde behar da, azpirrutinaren aktibazio-blokearen parte gisa, gainerako erregistroekin egiten den modu berean (makinaren egoera salbatzeko).

2.2.16 irudian ageri da nola geratuko den aktibazio-blokea FPa gehituta.



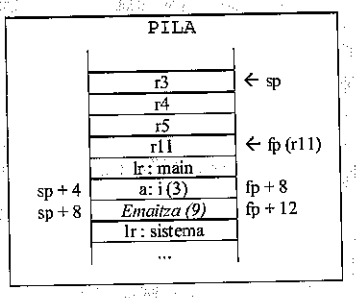
2.2.16 irudia. Aktibazio-blokea, aurreko FPa gordetzen denean.

2.5.4 Adibidea

Aurreko adibide bera da hau, baina, kasu honetan, FP-a (r11) erabiltzen da PILAko parametroak atzitzeko.

Azpirrutinaren hasieran, bere balioa aldatu aurretik, r11 erregistroaren balioa pilan sartzen da lr erregistroarekin batera. Ondoren, SParen balioa kopiatzen da r11 erregistroan. Une horretatik aurrera r11 erregistroarekiko helbideratze erlatiboa erabiltzen da pilako elementuak atzitzeko r11-ren balioa ez baita gehiago aldatuko azpirrutinaren exekuzioan zehar.

```
.code 32
.global main, __cpsr_mask

main:
        push {lr}
        ldr r0, a
        sub sp, sp, #4
        push {r0}
        bl AZP
        add sp, sp, #4
        pop {r0}
        str r0, ema
        pop {pc}
   a: .word 3
 ema: .word 0
```



```
AZP:
        push {r11,lr}
        mov r11, sp
        push {r3,r4,r5}
        ldr r3, [r11, #8]
        ldr r4, [r11, #8]
        mul r5,r3,r4
        str r5, [r11, #12]
        pop {r3,r4,r5,r11,pc}
```

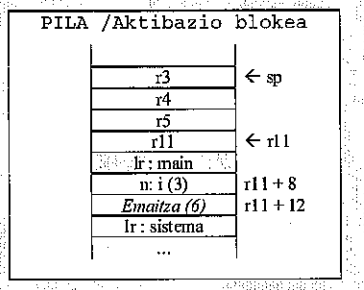
2.2.5 Adibidea

Faktorial programa: *faktoriala iteratiboki kalkulatzen da, FAKT funtzioaren bidez, zeinak, emaitza gisa, zenbakiaren faktoriala itzultzen duen. Goi-mailako programa C-z hau da:*

(C)

```
int FAKT (int i)
{
   int b, k;
   k = 1;
   b = 1;
   while (k<i)
   {
      k++;
      b = b * k;
   }
   return (b);
}

void main ()
{
   int n=3;
   int emai;
   emai = FAKT(n);
   printf ("%d", emai);
}
```

```
.code 32
.global main, __cpsr_mask

main:
        push {lr}
        sub sp, sp, #4
        ldr r0, n
        push {r0}
        bl FAKT
        add sp, sp, #4
        pop {r0}
        str r0, ema
        pop {lr}

n: .word 3
ema: .word 0

FAKT:
        push {r11,lr}
        mov r11,sp
        push {r3,r4,r5}
        mov r3, #1
        mov r4, #1
        ldr r5, [r11, #8]
while:  cmp r4, r5
        bge buk
        add r4,r4,#1
        mul r3,r4,r3
        b while
buk:    str r3, [r11, #12]
        pop {r3,r4,r5,r11,pc}
```



2.2.6 Adibidea

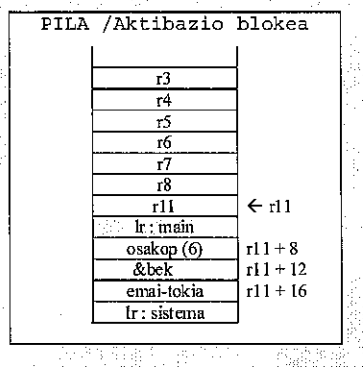
Bektoreak parametro bezala: *funtzioak parametro bezala jasotzen duen bektorean dagoen zenbaki negatibo kopurua itzultzen du.*

(C)

```
int NEGATIBO (int * bek; int
osakop)
{
 int i=0;
 int kont=0;
   for (i=0; i< osakop; i++)
      if (bek[i]< 0) kont++;
   return (kont);
 }
```

```
void main ()
{
   int b={3, -3,-4,5,12,-5};
   int emai;
   emai = NEGATIBO(b,6);
   printf ("%d", emai);
}
```

```
.code 32
.global main, __cpsr_mask

main:

        push {lr}
        sub sp, sp, #4
        adr r1, b
        mov r0, #6
        push {r0,r1}
        bl NEGATIBO
        add sp, sp, #8
        pop {r0}
        str r0, emai
        pop {lr}

b: .word 3, -3,-4,5,12,-5;
emai: .word 0;


NEGATIBO:
        push {r11,lr}
        mov r11, sp
        push {r3-r8}
        mov r5, #0
        mov r3, #0
        ldr r4, [r11,#8]
for:    cmp r4, r3
        beq bukfor
        ldr r6, [r11, #12]
        ldr r7, [r6, r3, lsl#2]
        add r3, r3, #1
        blt nega
        b for
nega:   add r5, r5, #1
        b for
bukfor: str r5, [r11, #16]
        pop {r3-r8,r11,pc}
```



2.3 AGINDU-FORMATUA: HELBIDERATZE MODUAK

2.3.1 Sarrera

Dagoeneko aipatu den moduan, konputagailuak 1ekoen eta 0koen segidak —behe-mailako aginduak —exekutatzen ditu. Agindu horiek konputagailu horren makina-lengoaia osatzen dute, edo, mnemoteknikoen bidez adierazita, mihiztadura-lengoaia. Konputagailu bakoitzak berezko makina-lengoaia erabiltzen du, eta lengoaia hori erabat erlazionaturik dago prozesadoreko hardwarearekin.

Makina-lengoaien ezaugarri nagusiak hauek dira:

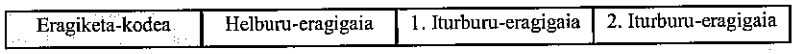
- agindu-multzoa, hau da, exekuta daitezkeen aginduak
- aginduen formatua, hots, nola egituratzen den informazioa
- eragigai kopurua
- helbideratze-moduak
- datuen formatua

Makina-lengoaiako agindu batek eragiketa bat eta bere eragigaiak adierazten ditu. Beraz, aginduetan hiru atal bereizten dira: (a) **eragiketa-kodea**, egin nahi den eragiketa adierazteko; (b) **iturburu-eragigaiak**, eragiketa egiteko behar diren datuak nola eskura daitezkeen adierazteko; eta (c) **helburu-eragigaia**, eragiketaren emaitza non gorde behar den adierazteko. Informazio hori guztia, hots, agindua bera, bit-segida baten bidez adierazten da, eta memorian gordetzen da.

Kapitulu honetan, aginduak kodetzeko erak azalduko dira, eta, horrez gainera, aginduen kodeketak konputagailuaren eraginkortasunean izan dezakeen eragina. Orain arte bezala, adibide gisa ARM9 makina eta bere makina- eta mihiztadura-lengoaiak hartuko ditugu.

2.3.2 Agindu formatua

Esan bezala, aginduaren informazioa bit-segida batean gordetzen da; hor adierazten da, hainbat **eremutan** banatuta, zer eragiketa bete behar den, zer eragigai erabili behar diren, eta emaitza non gorde behar den. Informazio hori guztia egituratzeko erabiltzen den formatuari **agindu-formatua** deritzo. Adibide gisa, lau eremuko agindu baten formatua ageri da 2.3.1 irudian.



2.3.1 irudia. Lau eremuko agindu-formatu baten adibidea.

Hurrengo ataletan, agindu-formatuen osagaiak eta haien kodeketa aztertuko ditugu.

2.3.2.1 Aginduaren luzera

Agindu-multzo bat diseinatu behar denean, parametro asko erabaki behar da, eta haien artean aginduen luzera, hots, aginduak adierazteko erabili behar den bit kopurua. Erabaki hori hartzeko, irizpide asko erabil daitezke; erabakiaren arabera, aginduen bilaketa fasea errazagoa edo konplexuagoa izango da. Hauek dira aukera nagusiak:

a. **Luzera finkoko** formatua: agindu guztiak tamaina edo bit kopuru berekoak dira.

   Luzera finkoko formatuaren abantaila nagusia aginduen bilaketa fasean ageri da, memoriatik irakurri behar den bit kopurua beti bera baita, edozein agindutarako. Baina, beste alde batetik, memoria-edukiera ez da modu egokian aprobetxatzen; izan ere, hainbat agindutan ez dira eremu guztiak erabili behar, eta, agindu guztiak tamaina berekoak direnez, eremu horiei dagozkien bitak alferrik okupatuko dira memorian.

   Gure ereduko makinak 32 biteko agindu luzera finkoa du.

b. **Luzera aldakorreko** formatua: aginduak tamaina edo bit kopuru desberdinekoak dira.

   Tamaina aldakorra denez, aginduan bertan adierazi beharko da haren luzera, eragiketa-kodean edo eremu jakin batean. Eragiketa-kodean adierazten bada, aginduaren luzera ez da ezagutuko deskodeketa egin arte. Luzera aldakorreko formatuaren abantailak eta eragozpenak luzera finkokoaren kontrakoak dira. Batetik, aginduaren bilaketa konplexuagoa da (ez baitakigu zenbat bit irakurri behar diren agindu osoa eskuratzeko), baina, bestetik, hobeto aprobetxatuko da memoria, agindu bakoitzak behar duen espazioa bakarrik okupatuko baitu (hala ere, aginduek okupatzen duten memoria-espazioa ez da, gaur egun, oso parametro kritikoa).

2.3.2.2 Eragiketa-kodearen formatua

Eragiketa-kodeak adierazten du exekutatu behar den eragiketa, eta, jakina, agindu guztiak kodetu ahal izateko adina bit behar du. Aginduaren formatuarekin gertatzen den bezala, finkoa nahiz aldakorra izan daiteke eragiketa-kodearen luzera. Egiten den hautuaren arabera, errazagoa edo zailagoa izango da aginduen deskodeketa.

a) Luzera finkoko eragiketa-kodea

Eragiketa-kodeak bit kopuru finkoa du, agindu guztietan bera. Kodetu behar den agindu kopuruak zedarrituko du luzera (l):

$$l = \lceil \log_2 (\textit{agindu kopurua}) \rceil$$

Eragiketa-kodea luzera finkokoa denean, errazagoa da aginduen deskodeketa, aldez aurretik ezagutzen baita zenbat eta zein bitek osatzen duten eragiketa-kodea.

Hala ere, kodetu behar den agindu kopurua handia bada, bit asko erabili beharko da eragiketa-kodea adierazteko; ondorioz, luzeagoak izango dira aginduak, edo, bestela, aginduen luzera mugatu nahi bada, bit gutxi geratuko dira eragigaiak adierazteko. Adibide honetan ageri da arazo hori.

2.3.1 Adibidea

Konputagailu baten agindu-multzoa diseinatu behar da. Agindu-multzoak 76 agindu ditu, honela banatuta:

- *hiru eragigaiko 15 agindu*
- *bi eragigaiko 14 agindu*
- *eragigai bakarreko 31 agindu*
- *eragigairik gabeko 16 agindu*

Eragigai bakoitza 4 bitetan kodetu behar da, eta aginduek luzera finkokoak izan behar dute, 16 bitekoak. Egin daiteke kodeketa hori luzera finkoko eragiketa-kodeak erabiliz?

Eragiketa-kodea luzera finkokoa izanik, 76 agindu kodetu ahal izateko, 7 bit behar dira ($\log_2 76 = 6{,}25 \rightarrow 7$). Ondorioz, eragigairik gabeko aginduak 7 bitekoak izango dira; eragigai bakarrekoak, $7 + 4 = 11$ bitekoak; bi eragigaikoak, $7 + 2 \times 4 = 15$ bitekoak; eta, azkenik, hiru eragigaikoak, $7 + 3 \times 4 = 19$ bitekoak.

Aginduek 16 bitekoak izan behar dutenez, lehenengo hiru formatuak arazorik gabe kodetu daitezke, 16 bit baino gutxiago behar dituztelako (bit batzuk erabili gabe geratuko dira); azkenak, berriz, bit gehiago beharko luke. Ondorioz, ezinezkoa da 16 biteko formatu finko batekin agindu horiek guztiak kodetzea.

Beraz, aurreko aginduak 16 bitetan kodetu behar badira, beste kodetze-estrategiak eta formatuak erabili beharko dira.

b) Luzera aldakorreko eragiketa-kodea

Aginduen eragiketa-kodea luzera aldakorrekoa izan daiteke. Izan ere, agindu guztiek ez dute eragigai kopuru bera erabiltzen; beraz, eragigai gutxi duten aginduek eragiketa-kode luzeagoak eduki ditzakete, eta eragigai asko dutenek, berriz, laburragoak. Estrategia hori erabiliz, txikiagoa da, batez beste, aginduen luzera, edo, luzerari eutsi nahi bazaio, eragigai gehiago adieraz daitezke.

Hala ere, eragiketa-kodeak luzera desberdinekoak direnez, konplexuagoa izango da aginduen deskodeketa. Izan ere, ez dakigu zenbat bit deskodetu behar diren; beraz, zatika egin beharko da deskodeketa. Hasteko, bit kopuru jakin bat analizatu beharko da: eragiketa-kode bat osatzen duen kopuru minimoa, hain zuzen ere. Bit horiek adieraziko dute nahikoa den ala bit gehiago deskodetu behar diren; bigarren kasuan, jarraitu beharko da prozesuarekin, eragiketa-kode osoa deskodetu arte.

2.3.2.3 Eragigaien formatua. Helbideratze-moduak.

Eragiketa-kodea adierazteko erak aztertu ondoren, eta aginduen kodeketarekin jarraituz, eragigaien formatua analizatu behar dugu. Agindu batean erabili behar diren eragigaiak modu askotan adieraz daitezke, haien helbideratze-moduen arabera. Beraz, aginduen formatua definitzeko, eragigaien helbideratze-moduak aztertu behar dira.

Helbideratze-moduek adierazten digute nola lortu eragigaiak eragiketa baterako eta non gorde eragiketaren emaitza (memorian, erregistro batean, ...).

Oro har, helbideratze-modu asko erabiltzen dira eragigaiak adierazteko. Izan ere, hainbat datu-mota (konstanteak, aldagai eskalarrak, bektoreak, matrizeak, pilak, ilarak, erregistroak, ...) prozesatu behar dira, eta eraginkorra da helbideratze-modu egokiak izatea datu-egitura horiekin modu erosoan lan egiteko (esaterako, indizeak erabiltzea bektoreak atzitzeko).

Helbideratze-moduek eragin zuzena dute bi arlotan: eragigaia adierazteko behar den bit kopuruan, eta eragigaia eskuratzeko behar den denboran:

- datua erregistro batean egonez gero, bit gutxi beharko dira erregistro hori adierazteko (erregistro kopurua txikia da); aldiz, memorian egonez gero, bit gehiago beharko dira haren helbidea emateko.
- datua eskuratzeko behar den denbora ere desberdina da: denbora gutxiago behar da prozesadoreko erregistro batean dagoen datua atzitzeko, memorian dagoena irakurtzeko baino. Oso garrantzitsua da ezaugarri hori, jakina, programak azkar exekutatzea baita konputagailuen helburuetako bat.

Bestalde, konputagailu baten helbideratze-moduak diseinatu behar direnean, honako helburu hauek ere kontuan hartu behar dira:

- Memoriako eremu edo zati bat definituta, eremu horren edozein posizio atzitzeko aukera izan behar da.
- Modu eroso eta eraginkorrean atzitu behar dira goi-mailako lengoaietan erabiltzen diren datu-egiturak (bektoreak, matrizeak, ...).

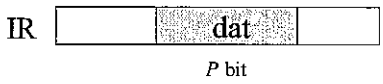
Hurrengo ataletan, helbideratze-modu erabilienak azalduko ditugu.

2.3.2.3.1 Berehalakoa *(immediate addressing)*

Lehenengo helbideratze-modu hau sinpleena da: eragigaia konstante bat da, eta aginduan bertan dago, hots, IR agindu-erregistroan. 2.3.2 adibidean, helbideratze-modu hori duen agindu bat ageri da eta 2.3.1 taulan agindu hori ARM makina batean nola kodetzen den ikusten da.

Eragigaiaren eremuaren luzerak edo bit kopuruak mugatzen du modu horretan adieraz daitezkeen datuen magnitudea. Oro har, birako osagarria erabiltzen da datuak adierazteko; ondorioz, eragigaiaren eremua p bitekoa izanik, datuen tartea $[-2^{p-1}, 2^{p-1}-1]$ da.

Abantaila garbi bat du helbideratze-modu honek: erabili behar den datuaren atzipena azkarra da oso. Izan ere, datua hortxe dugu, ez da ezer egin behar datua eskuratzeko. Helbideratze-modu honekin, ostera, datu konstanteak besterik ezin dira erabili (aldagaien hasiera-balioak adibidez). Gainera, konstante horien tamaina mugatua da, eta gerta daiteke konstante handi bat adierazi behar izatea eta lekurik ez izatea IR erregistroan.



2.3.2 irudia. Berehalako helbideratze-modua

ARM mihiztadura-lengoaian, berehalako datuak # karakterearen atzetik idazten dira, hamartarrez (zuzenean zenbakia jarrita), edo hamaseitarrez (aurretik 0x jarrita) adieraz daitezke. Berehalakoak 12 bitean adierazten dira, ikusi dugun bezala ondorengo egiturari jarraituz:

- IR (4 bit): bere balioa bider 2 egin ondoren lortzen den zenbakiak IN eremua eskuinera zenbat bit errotatu behar den adierazten du.
- IN (8 bit): balio bat adierazten du, eskuinera errotatuko dena IR*2 posizio.

Bitak horrela banatzeko ideia 12 bitekin adieraz daitezkeenak baino zenbaki handiagoak adieraztea da. Modu honetara, 32 bit behar dituzten zenbakiak adieraztera irits daiteke 12 bitekin. Hala ere, garbi izan behar da 12 bitekin 4096 zenbaki desberdin adieraz daitezkeela, eta ez 32 bitekin sor daitezkeen 4 mila miloi konbinazio desberdinak. Modu honetara adieraz daitezkeen berehalakoak ondokoak dira: $[0...255]*2^{2n}$.

2.3.2 Adibidea

Ondorengo ARM aginduak hamartarrez idatzitako berehalako datu bat erabiltzen du. 2.3.1 taulan ikus daiteke nola gauzatzen den horren kodeketa ARM makinan.

```
add r2,r1,#12
```

Baldintza	00	I	Eragiketa	S	Rn	Rd	2. Eragigaia
1110	00	1	0100	0	0001	0010	0000 | 00001100
Beti			ADD		R1	R2	#12

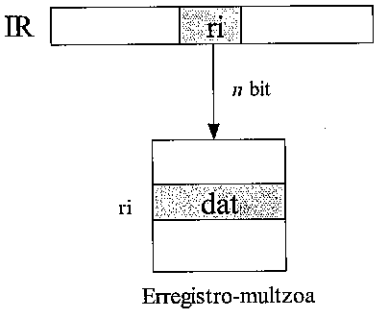
2.3.1 taula. Berehalako helbideratze-moduaren kodeketa ARM makinan.

2.3.2.3.2 Erregistro bidezko zuzenekoa *(register addressing)*

Erabili behar den datua erregistro-multzoan dago, erregistro jakin batean, eta erregistroaren zenbakia edo helbidea adierazten da aginduan, IR erregistroan, eragigaiari dagokion eremuan. Beraz, erregistro bat irakurri behar da datua eskuratzeko[3] (2.3.3 irudia):

datua = EM [erregistro-zenbakia]

Helbideratze-modu hau, jakina, erregistro-multzoan gorde diren datuak atzitzeko erabiltzen da.



2.3.3 irudia. Erregistro bidezko zuzeneko helbideratze-modua

Eragigaia adierazten duen agindu-eremuaren luzerak, *n*-k, nahikoa behar du izan erregistro-multzoko edozein erregistro adierazteko; beraz, honako hau bete behar da:

$$n = \lceil \log_2 (\textit{erregistro kopurua}) \rceil$$

Helbideratze-modu honek zenbait abantaila ditu. Batetik, erregistroen atzipena azkarra da (hori dela eta, erregistroetan gordetzen dira maiz erabili behar diren datuak); eta, bestetik, erregistro bat adierazteko behar den bit kopurua, *n*, txikia da, ohi denez, erregistro kopurua ez baita oso handia. Hala ere, datuak memoriatik erregistroetara kargatzeko eta erregistroetatik memoriara eramateko aginduak gehitu beharko dizkiogu programari, eta, beraz, programak luzeagoak izango dira.

ARM konputagailuek 16 erregistro orokor dituzte, eta, beraz, 4 bit behar dira erregistro zenbaki bat adierazteko.

2.3.3 Adibidea

Ondorengo ARM aginduak 4 bitez adierazitako hiru erregistro atzitzen ditu:

```
add r3,r2,r4
```

2.3.2 taulan ikusten da nola kodetzen diren erregistro bidezko zuzeneko helbideratzeak ARM makinan.

Baldintza	00	I	Eragiketa	S	R_{il}	R_h	2. Eragigaia
1110	00	0	0100	0	0010	0011	0000 0000 | 0100
Beti			ADD		R2	R3	R4

2.3.2 taula. Erregistro bidezko zuzeneko helbideratze-moduaren kodeketa ARM makinan..

2.3.2.3.3 Absolutua *(direct or absolute addressing)*

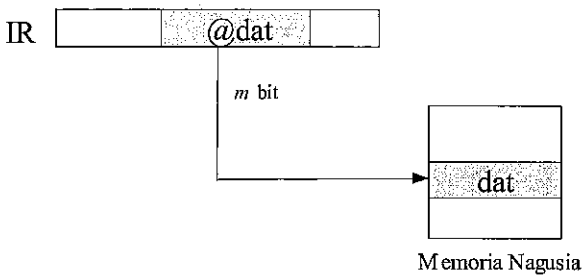
Kasu honetan, memorian dago erabili behar den datua, eta haren helbidea adierazten da aginduan (2.3.4 irudia). Datua eskuratzeko, beraz, memoria irakurri behar da, besterik gabe:

datua = M[helbidea]

[3] Orain arte ri erabili badugu ere, atal honetan EM[ri] erabiliko dugu ri erregistroaren edukia adierazteko, erregistro-multzoan atzipen bat egin behar dela azpimarratzeko.

Helbidea adierazten duen agindu-eremuaren luzerak, m-k, memoriako edozein posizio identifikatzeko gai izan behar du. Bere tamaina, bitetan, honela kalkula daiteke:

$$m = \lceil \log_2 (\textit{memoriako posizio kopurua}) \rceil$$



2.3.4 irudia. Helbideratze-modu absolutua

Memoria-posizioak, oro har, asko dira; beraz, edozein posizio helbideratu ahal izateko, eremu horrek handia izan behar du, eta agian ez da toki nahikorik izango IR erregistroan. Adibidez, 4 GB-ko memoria helbideratzeko, 32 bit behar dira. Zer egin, esaterako, aginduak, guztira, 32 bitekoak badira?

Helbide bat adierazteko bit kopuru txikiagoa erabili ahal izateko, hainbat **segmentutan** zatitzen da helbideratze-espazioa konputagailu batzuetan (adibidez, 80x86 familian eta haren ondorengoetan). Halakoetan, memoriako helbideak bi zatitan ematen dira; batetik, memoria-segmentu jakin baten hasiera- edo oinarri-helbidea, eta, bestetik, desplazamendua, hots, oinarri-helbidetik datura dagoen posizio kopurua edo "distantzia". Izan ere, desplazamendua baino ez da adierazten IR erregistroan — bit gutxi, alegia—, eta memoria-segmentuen hasiera-helbideak erregistro berezietan adierazten dira.

Programa baten datuen eta aginduen memoria-helbideak ez dira beti berak. Izan ere, programa bat exekutatu behar denean, lehendabizi memorian kargatu behar da, eta une horretan erabakitzen dira datuen eta aginduen posizioak, memorian libre dagoen espazioaren arabera. Horregatik esaten da programak "birkokagarriak" direla. Beraz, programa batean adierazitako memoria-helbide absolutuak egokituko dira, exekuzio bakoitzean, programa kargatzen den memoria-eremura.

Helbide bat adierazteko, aurredefinitutako aldagai baten izena ematea da aukera bakarra ARM makinetan. Hala ere, makina lengoaian dagoen aginduan, ezin da memoriako helbide bat gorde: aginduak 32 bitekoak dira eta memoriako helbideak ere 32 bitekoak dira. Hau dela eta, aginduan ez da memoriako helbide oso bat gordetzen 2.3.3 taulan ikus daitekeen bezala, gordetzen dena desplazamendu bat da. Desplazamendu hau PC erregistroaren edukiari batu beharko zaio atzitu behar den memoriako helbidea kalkulatzeko. Beraz, helbideratze absolutu bat baino gehiago PCarekiko helbideratze erlatiboa dela esan daiteke (oinarri-erregistroa + desplazamendua).

2.3.4 Adibidea

ARM mihiztadura-lengoaiako agindu honetan helbideratze absolutua erabiltzen da:

```
ldr r2, B  {aldagaiaren izenak helbidea adierazten du}
```

baldintza	00	I	P	U	B	W	L	R_n	R_h	Desplazamendua
1110	01	0	1	0	0	0	1	1111	0010	000001101100
Beti		*1	*2	*3	*4	*5	*6	PC	R2	#108

*1= Immediate=berehalakoa --> 0 izanik desplazamendutzat konstantea hartuko du
*2= Pre/Post indexing --> 1ekoa izanik pre-indexing beraz, kalkulua memoriara jo aurretik eginen da
*3= Up/Down --> 0koa izanik kenketa eginen da
*4= Byte/Word --> 0koa izanik hitza jasoko da
*5= Write back --> 0koa izanik ez du helbide berria gordeko oinarri-erregistroan
*6= Load/Store --> 1koa izanik, load eginen da Emaitza --> r2=Mem[PC - 128]

2.3.3 taula. Helbideratze absolutuaren kodeketa ARM makinan.

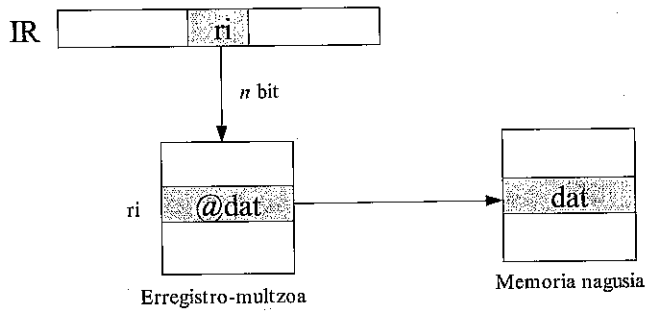
2.3.2.3.4 Erregistro bidezko zeharkakoa *(register indirect addressing)*

Memorian dago erabili behar den datua, eta erregistro batean gorde da haren helbidea. Izan ere, datuaren helbidea duen erregistroa adierazten da aginduan, IR erregistroan. Beraz, datua eskuratzeko, erregistro bat irakurri behar da lehenik, eta, gero, lortutako helbidearekin memoriara jo (ikus 2.3.5 irudia).

datua = M[EM[erreg-zenb]]

Eragigaia adierazteko eremuan, beraz, *n* bit behar dira, erregistro bat adierazteko:

$$n = \lceil \log_2 (\textit{erregistro kopurua}) \rceil$$



2.3.5 irudia. Erregistro bidezko zeharkako helbideratze-modua.

Helbideratze-modu honetan, beraz, memoriako helbide bat adierazteko bit gutxi erabiltzen dira, erregistro bat adierazteko bit kopurua bakarrik, baina, ordainean, eragigaiak bi pausotan atzitu behar dira. Helbideratze-modu hau oso aproposa da bektoreekin lan egiteko; bektorearen hasierako helbidea erregistro batean gordetzen da, eta, gero, bektorearen osagaien helbideak eskuratzeko, erregistroaren edukia gehitu besterik ez da egin behar. Hori dela eta, oso erabilia da datu egituratuak —bektoreak, matrizeak, pilak, ilarak, ...— modu eraginkorrean atzitzeko.

Arestian aipatu den moduan, 4 bit nahikoak dira ARM konputagailuan erregistro bat adierazteko. Helbideak gordetzen dituzten erregistroak makoen artean adierazten dira.

2.3.5 Adibidea

ARM mihiztadura-lengoaiako agindu hauetan, erregistro bidezko zeharkako helbideratzea erabiltzen da:

```
ldr r1, [r2]   ikus 2.3.4 taula.
str r3, [r1]   ikus 2.3.5 taula.
```

Baldintza	00	I	P	U	B	W	L	R_n	R_h	Desplazamendua
1110	01	0	1	1	0	0	1	0010	0001	000000000000
Beti		*1	*2	*3	*4	*5	*6	R2	R1	0

*1= Immediate=berehalakoa --> 0 izanik desplazamendutzat konstantea hartuko du
*2= Pre/Post indexing --> 1ekoa izanik pre-indexing beraz, kalkulua memoriara jo aurretik eginen da
*3= Up/Down --> 1koa izanik batuketa eginen da, kontuan izanik desplazamendua 0 dela ez da batuketarik izango
*4= Byte/Word --> 0koa izanik hitza jasoko da
*5= Write back --> 0koa izanik ez du helbide berria gordeko oinarri-erregistroan
*6= Load/Store --> 1koa izanik, load eginen da
Emaitza --> r1=Mem[r2]

2.3.4 taula. Erregistro bidezko zeharkako helbideratzearen kodeketa ARM makinan load agindu batentzat.

Baldintza	00	I	P	U	B	W	L	Rn	Ri	Desplazamendua
1110	01	0	1	1	0	0	0	0001	0011	000000000000
Beti		*1	*2	*3	*4	*5	*6	R1	R3	0

*1= Immediate=berehalakoa --> 0 izanik desplazamendutzat konstantea hartuko du
*2= Pre/Post indexing --> 1ekoa izanik pre-indexing beraz, kalkulua memoriara jo aurretik eginen da
*3= Up/Down --> 1koa izanik batuketa eginen da, kontuan izanik desplazamendua 0 dela ez da batuketarik izango
*4= Byte/Word --> 0koa izanik hitza jasoko da
*5= Write back --> 0koa izanik ez du helbide berria gordeko oinarri-erregistroan
*6= Load/Store --> 0koa izanik, store eginen da
Emaitza --> Mem[r1]=r3

2.3.5.taula. Erregistro bidezko zeharkako helbideratzearen kodeketa ARM makinan store agindu batentzat.

2.3.2.3.5 Oinarri-erregistroa gehi desplazamendua: erlatiboa *(base or displacement addressing)*

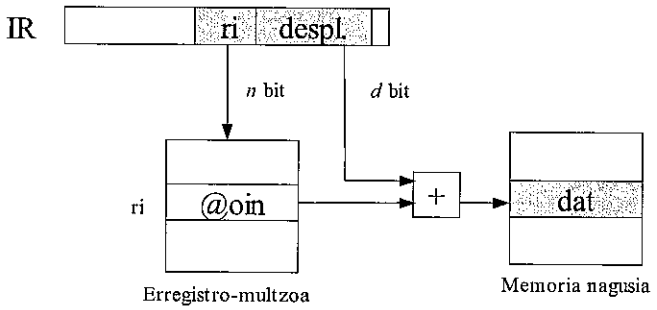
Helbideratze indexatuan gertatzen den moduan, honetan ere, bi zatitan banatzen da datuaren helbidea: batetik, erregistro baten identifikadorea, eta, bestetik, desplazamendua. Datuaren helbidea lortzeko, lehenik, erregistroa irakurri behar da, eta, gero, irakurri den helbideari desplazamendua gehitu (2.3.7 irudia):

datua = M[EM[erreg-zenb] + desplazamendua]

Erabiltzen den erregistroari oinarri-erregistroa deritzo, datu-egitura baten erreferentzia- edo oinarri-helbidea gordetzen baitu. Helbide horretatik aurrera (edo atzera) desplazatuko gara, IR erregistroan bertan adierazten den desplazamenduaren bidez. Desplazamendua adierazteko erabiltzen den bit kopuruak, *d*-k, desplazamendu horien distantzia mugatuko du; izan ere, hasiera-puntu jakin batetik memoriako edozein posizio atzitu ahal izateko, desplazamenduaren bit kopuruak memoriako helbideena izan beharko luke (oro har, hala ere, bit kopuru txikiagoa erabiltzen da).

Beraz, $n + d$ bitekoa da eragigaia adierazteko eremua: n bit erregistro bat identifikatzeko, eta d bit $[-2^{d-1}, 2^{d-1}-1]$ tarteko desplazamendua adierazteko (adierazpidea birako osagarria izanik):

$$n = \lceil \log_2 (\textit{erregistro kopurua}) \rceil$$



2.3.6 irudia. Helbideratze-modu erlatiboa.

ARM konputagailuetan, desplazamendua 12 bitekoa da, hots, [–2048, +2047] tartean egon daiteke. Beraz, 16 bitekoa da helbideratze erlatiboa: 4 bit oinarri-erregistrorako, eta 12 desplazamendurako. Dagoeneko aipatu den moduan, erregistro bidezko zeharkako helbideratzea erlatiboaren kasu berezi bat baino ez da; desplazamendua zero denekoa, alegia. Hau da:

```
ldr r1, [r5]    ≡    ldr r1, [r5, #0]
```

2.3.6 Adibidea

ARM mihiztadura-lengoaiako agindu honetan, helbideratze erlatiboa erabiltzen da:

`ldr r1, [R2, #16]` *ikus 2.3.6 taula*

Baldintza	00	I	P	U	B	W	L	R_n	R_h	Desplazamendua
1110	01	0	1	1	0	0	1	0010	0001	000000010000
Beti		*1	*2	*3	*4	*5	*6	R2	R1	#16

*1= Immediate=berehalakoa --> 0 izanik desplazamendutzat konstantea hartuko du
*2= Pre/Post indexing --> 1ekoa izanik pre-indexing beraz, kalkulua memoriara jo aurretik eginen da
*3= Up/Down --> 1koa izanik batuketa eginen da
*4= Byte/Word --> 0koa izanik hitza jasoko da
*5= Write back --> 0jkoa izanik ez du helbide berria gordeko oinarri-erregistroan
*6= Load/Store --> 0koa izanik, store eginen da Emaitza --> r1=Mem[r2+16]

2.3.6 taula. Helbideratze erlatiboaren kodeketa ARM makinan

Helbideratze-modu erlatiboa eta indexatua funtzionalki oso antzekoak dira: datuaren helbidea kalkulatzeko, erregistro baten edukiari konstante bat gehitu behar zaio. Desberdintasuna interpretazioan eta erabileran datza. Indexatuan, oinarri-helbidea konstante bat da, eta gehitzen den desplazamendua, indizea, aldakorra da, erregistro baten edukia baita (eta alda daiteke exekuzioan zehar). Erlatiboan, aldiz, oinarri-helbidea aldakorra da, erregistro baten edukia baita, eta desplazamendua konstantea da. Hori guztia dela eta, oinarri-helbideak edota desplazamenduak adierazteko bit kopuruak ez datoz beti bat (adibidez, oinarri-helbidea ARM makinetan ezin da adierazi). Gainera, zenbait konputagailutan (esaterako, Motorola familian) erregistro zehatzak erabili behar dira helbideak edo datuak gordetzeko.

2.3.7 Adibidea

Ondoko programa idatziko dugu mihiztadura lengoaian helbideratze erlatiboa erabilita:

for (i=0; i< 9; ++)
B[i] = A[i] + A[i+1];

```
        .code 32
        .global main, __cpsr_mask

        B: .word 0,0,0,0,0,0,0,0,0
        A: .word 5, 4, 3,7,8,6,7,5,1,2

        main:
           mov r0, #0          @ r0= batura
           adr r1, B           @ r1= Bek-en helbidea
           adr r2, A
FOR:    cmp r0, #9
        beq buk
        ldr r3,[r2]
        ldr r4, [r2, #4]
        add r5, r3, r4
        add r2, r2, #4
        str r5, [r1]
        add r1, r1, #4
        add r0, r0,#1
        b   FOR
buk:    mov pc, lr
```

2.3.2.3.6 Memoria bidezko zeharkakoa *(memory indirect addressing)*

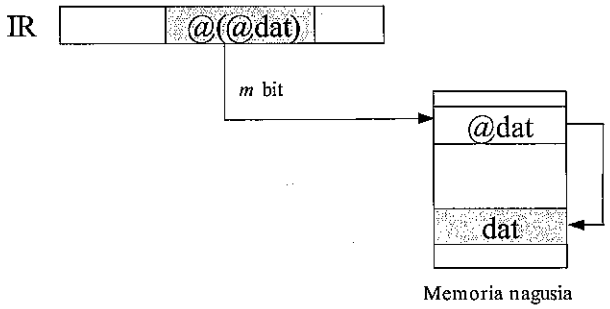
Aurreko kasuan bezala, memorian dago eskuratu behar den datua, baina haren helbidea ez dago, ez IR erregistroan ezta erregistro orokor batean ere. Hain zuzen ere, datuaren helbidea ere memorian gorde da, eta IR erregistroak helbide hori gordetzen duen memoria-posizioaren helbidea zehazten du. Ondorioz, 2.3.7 irudian ageri den bezala, memoriako bi atzipen egin behar dira datua eskuratzeko: bata, datuaren helbidea irakurtzeko, eta, bestea, berriz, datua bera lortzeko.

$$\text{datua} = \text{M[M[helb]]}$$

Eragigai-eremuaren luzerari dagokionez, helbideratze-modu absoluturako behar den bit kopuru bera behar da, bi kasuetan memoriako helbide bat gorde behar baita:

$$m = \lceil \log_2 (\textit{memoriako posizio kopurua}) \rceil$$

Helbideratze-modu hau ez da askorik erabiltzen, memoria bi aldiz atzitu behar delako, eta, oro har, atzipen-denbora handia delako.



2.3.7 irudia. Memoria bidezko zeharkako helbideratze-modua.

Oro har, zeharkako helbideratzeak, erregistro bidezkoak zein memoria bidezkoak, hainbat zeharkatze-maila erabil ditzake. Halakoetan, maila adina memoriako atzipen egin beharko da, eta azkenekoan aurkituko da datua. Erregistro bidezko zeharkakoan, datua eskuratzeko egin behar den memoriako atzipen kopurua adierazten du zeharkatze-mailak; memoria bidezko zeharkakoan, aldiz, atzipen bat gehiago egin behar da.

ARM makinak ez du helbideratze modu hau.

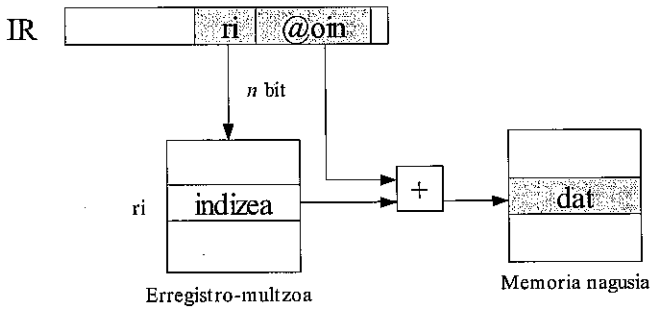
2.3.2.3.7 Indexatua *(indexed addressing)*

Kasu honetan ere, memorian dago atzitu nahi den datua. Haren helbidea, hala ere, bi partetan adierazten da: erregistro bat (indize-erregistroa) eta memoriako helbide bat (oinarri-helbidea). Datuaren helbidea lortzeko, oinarri-helbidea eta indize-erregistroaren edukia batu behar dira. Beraz,

$$\text{datua} = \text{M[helbidea} + \text{EM[erreg-zenb]]}$$

2.3.8 irudian ageri da helbideratze-modu honen eskema grafikoa.

Laburbilduz, datu bat eskuratzeko, lehenik, erregistro bat irakurri eta batuketa bat egin behar da datuaren helbidea kalkulatzeko, eta, gero, memoria atzitu.



2.3.8 irudia. Helbideratze-modu indexatua.

Bi parametro adierazi behar direnez, IR erregistroko eragigai-eremuaren luzera aurreko helbideratze-moduena baino handiagoa da. Oraingoan, luzera $m + n$ bitekoa da, m bit memoriako helbide bat adierazteko eta n bit erregistro bat adierazteko:

$$m = \lceil \log_2 (\textit{memoriako posizio kopurua}) \rceil$$

$$n = \lceil \log_2 (\textit{erregistro kopurua}) \rceil$$

Helbideratze-modu absolutua azaltzean aipatu dugun bezala, m oso handia bada (64 bit, esaterako), segmentutan bana daiteke memoria.

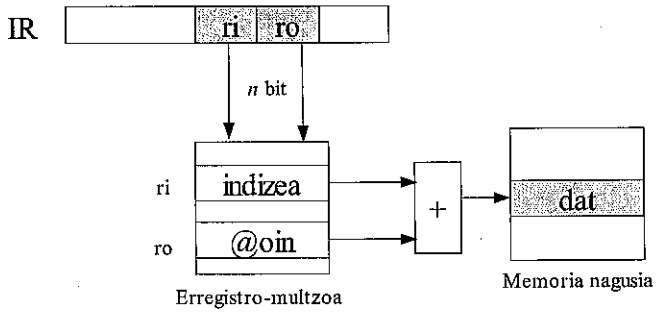
Helbideratze indexatua maiz erabiltzen da bektore baten osagaiak atzitzeko, edota goi-mailako lengoaiako datu egituratuekin lan egiteko.

ARM makinak ez du helbideratze modu hau.

2.3.2.3.8 Oinarri-erregistroa gehi indize-erregistroa

Helbideratze-modu honetan, aurreko bietan bezala, datua memorian dago, eta, eragigaiaren eremuan, bi erregistro adierazten dira: oinarri-helbidea gordetzen duen erregistroa (oinarri-erregistroa) eta indize-erregistroa (ikus 2.3.9 irudia). Datuaren helbidea lortzeko, bi erregistro horien edukia, oinarri-helbidea eta indizea, batu behar dira. Beraz,

datua = M[EM[oinarri-erregistroa] + EM[indize-erregistroa]]



2.3.9 irudia. "Oinarri-erregistroa gehi indize-erregistroa" helbideratze-modua.

Laburbilduz, eragigai bat eskuratzeko, lehenik, haren helbidea kalkulatu behar da, eta, horretarako, bi erregistro irakurri eta haien edukiak batu behar dira; gero, memoria atzitu beharko da. Adi: zenbait makinatan, erregistroak banan-banan irakurri behar dira: kasu horietan, atzipen mota hau motelagoa da, denbora gehiago beharko baita bi erregistroak irakurtzeko.

Eragigaiari dagokion eremuan bi erregistro adierazi behar direnez, haren luzera $n + n = 2n$ bitekoa izango da, non n erregistro bat kodetzeko behar den bit kopurua den:

$$n = \lceil \log_2 (\textit{erregistro kopurua}) \rceil$$

Aurrekoak bezala, helbideratze-modu hau egokia da bektoreen osagaiak atzitzeko, edo goi-mailako lengoaiako datu egituratuekin lan egiteko.

Helbideratze modu honek, bi erregistro erabiltzen dituenez, ARM makina-lengoaian 8 bitez kodetuko da, erregistroko launa.

2.3.8 Adibidea

ARM mihiztadura-lengoaiako agindu honetan, oinarri erregistro gehi indize erregistro helbideratze modua erabiltzen da:

```
ldr r2, [R3, R4]
```
ikus 2.3.6 taula

Baldintza	00	I	P	U	B	W	L	R_n	R_h	2. Eragigaia
1110	01	1	1	1	0	0	1	0011	0010	00000000 | 0100
Beti		*1	*2	*3	*4	*5	*6	R3	R2	R4

*1= Immediate=berehalakoa --> 1 izanik desplazamendutzat erregistroaren edukia hartuko du
*2= Pre/Post indexing --> 1ekoa izanik pre-indexing beraz, kalkulua memoriara jo aurretik eginen da
*3= Up/Down --> 1koa izanik batuketa eginen da
*4= Byte/Word --> 0koa izanik hitza jasoko da
*5= Write back --> 0jkoa izanik ez du helbide berria gordeko oinarri-erregistroan
*6= Load/Store --> 0koa izanik, store eginen da
Emaitza --> r1=Mem[r2+16]

2.3.6 taula. ARM makinan oinarri-erregistroa gehi indize-erregistroa kodetzeko modua

2.3.2.3.9 Pre/post-indexazioa eta berridazketa

Kasu honetan ez gara helbideratze-moduez ari, atzitu behar den memoriako helbidea kalkulatzeko batuketak behar dituzten helbideratze-moduek (erlatiboa eta o.e+i.e) erabil ditzaketen estrategiez baizik. Memoria helbideratu behar denean, ohikoa da erregistro bat eta beste datu baten arteko batuketa egin behar izatea. Kalkulu hori memoria helbideratu aurretik (pre-indexazioa) edo ondoren (post-indexazioa) egin daiteke.

- **Pre-indexazioa** erabiltzen denean bi datuak kortxete artean idazten dira eta emaitza berridaztea nahi izanez gero, bukaeran harridura ikurra jarri behar da.

```
ldr r2, [R3, R4]      @ r2=Mem(r3+r4)
ldr r2, [R3, R4]!     @r2=Mem(r3+r4) eta r3=r3+r4
```

- **Post-indexazioa** erabiltzen denean beti egiten da berridazketa, baina kalkulua memoria atzitu ondoren egiten da.

```
ldr r2, [R3], R4      @r2=Mem(r3) eta r3=r3+r4
ldr r2, [R3], #4      @r2=Mem(r3) eta r3=r3+#4
```

2.3.9 Adibidea

2.3.7 Adibideko programa bera idatziko dugu berridazketarako aukera erabilita.

```
        .code 32
        .global main, __cpsr_mask
        B: .word 0,0,0,0,0,0,0,0,0
        A: .word 5, 4, 3,7,8,6,7,5,1,2
        main:
           mov r0, #0
           adr r1, B
           adr r2, A
FOR:       cmp r0, #9
           beq buk
           ldr r3,[r2]
           ldr r4, [r2, #4]!
           add r5, r3, r4
           str r5, [r1], #4
           add r0, r0,#1
           b   FOR
buk:       mov pc, lr
```

2.3.10 Adibidea

ARM makina batean 0x000000F0 helbidetik hasita memoriako posizioen edukia ondokoa da:

@000000F0	00	00	00	06
	00	00	00	08
	80	90	90	90
	FF	34	65	80
	88	32	77	35

Era berean, erregistro batzuen edukia hau da:

r1 =0x000000F0 *r2= 4* *r4 = 0x00348536*

Exekutatu behar dira bost agindu hauek:

1. `ldr r5, [r1,r2]!`
2. `str r5, [r1, #4]`
3. `mov r6, #0x000000FC`
4. `ldr r7, [r6], #4`
5. `str r4, [r6]`

Adierazi nola aldatzen den memoriaren eta erregistroen edukia agindu horiek exekutatzen direnean.

Agindu bakoitzaren exekuzioan, aldaketa hauek gertatuko dira:

1. `ldr r5, [r1,r2]!`

Agindu honetan memoriatik hitz bat irakurriko da. Horretarako, lehenik memoriako helbidea kalkulatu behar da. Oinarri-erregistro + indize-erregistro helbideratze-moduan memoriako helbidea lortzeko bi erregistroen edukien batura kalkulatu behar da.

Memoriako helbidea hau izango da: **`r1+r2= 0x000000F0 + 4= 0x000000F4`**

Memoriako helbide horretatik irakurketa bat egin eta datua r5 erregistroan idatziko da:

`r5:=M[0x000000F4]=0x00000008`

Azkenik, ! ikurrak adierazten duen berridazketa dela eta, r1 erregistroaren edukia eguneratuko da r1+r2 baturaren emaitzarekin, beraz: `r1=0x000000F4`

2. str r5, [r1, #4]

Agindu honetan memorian hitz bat idatziko da. Horretarako, lehenik memoriako helbidea kalkulatu behar da. Helbideratze modu honetan, erlatiboa, r1 erregistroari #4 desplazamendua batzen zaio memoriako helbidea kalkulatzeko, beraz:

Memoriako helbidea hau izango da: `r1 + 4 = 0x000000F4 + 4 = 0x000000F8`. Memoriako helbide horretan r5 erregistroaren edukia idatziko da:

`M[0x000000F8]:=0x00000008`

@000000F0	00	00	00	06
	00	00	00	08
	00	**00**	**00**	**08**
	FF	34	65	80
	88	32	77	35

3. mov r6, 0x000000FC

`r6:= 0x000000FC`

4. ldr r7, [r6], #4

Agindu honetan memoria irakurriko da r6 erregistroak adierazitako helbidetik (erregistro bidezko zeharkako helbideratzea). Memoria irakurri ondoren r6 helbide-erregistroari 4-ko inkrementu bat egingo zaio eta r6-ren edukia eguneratuko da.

`r7 := M[r6] = M[0x000000FC] = 0xFF346580`

`r6 := r6 + 4 = 0x000000FC +4 = 0x00000100`

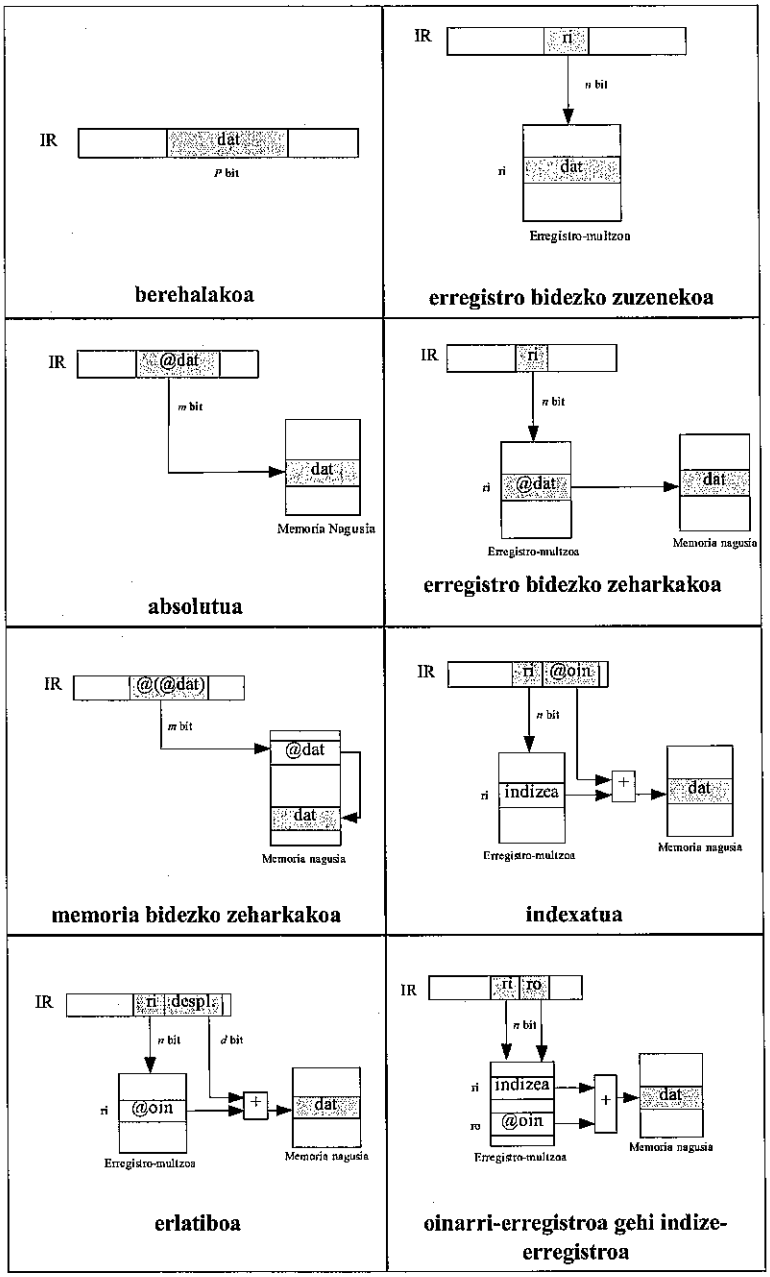
5. str r4, [r6]

M[r6]:= r4

M[0x00000100]:= 0x00348536

@000000F0	00	00	00	06
	00	00	00	08
	00	00	000	08
	FF	34	65	80
	00	**34**	**85**	**36**

Atal honekin bukatzeko, 2.3.10 irudian bildu dira aztertu ditugun helbideratze-modu guztiak.



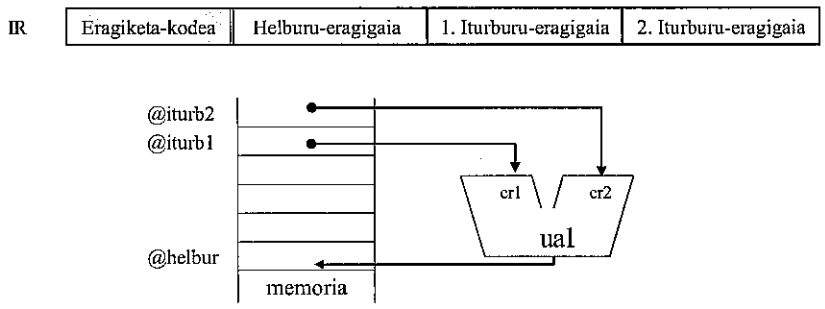
2.3.10 irudia. *Helbideratze-moduen laburpen grafikoa.*

2.4 KONPUTAGAILUEN SAILKAPENA AGINDU FORMATUAREN ARABERA

Ezaugarri asko eta oso desberdinak erabil daitezke konputagailuak sailkatzeko. Agindu aritmetikoetan ageri diren eragigai esplizituen kopurua da horietako bat. Hala, lau konputagailu mota bereizten dira: hiru eragigaiko aginduak dituztenak, bi eragigaikoak, eragigai bakarrekoak, eta eragigai espliziturik erabiltzen ez dituztenak. Sailkapen hori ez da, hala ere, erabat zehatza, konputagailu komertzial gehienek agindu-formatu horietatik bat baino gehiago erabiltzen dutelako. Hurrengo ataletan, agindu-formatu bakoitzaren ezaugarriak azalduko ditugu.

2.4.1 Hiru eragigai esplizituko konputagailuak

Aginduek hiru eragigai erabiltzen dituzte eragiketa bat egiteko. Eragigai batek —lehenengoak esaterako[4]—, emaitza non gorde behar den adieraziko du, eta, beste biak eragiketarako datuak izango dira. 2.4.1 irudian, adibide bat ageri da.



2.4.1 irudia. Hiru eragigaiko aginduen formatua eta eragiketa baten exekuzioa (helbideratze-modu absolutua).

Eragigaien helbideratze-moduen arabera, konputagailu desberdinak izango ditugu. Hiru eragigaiak memorian baldin badaude, konputagailuari **MMM motakoa** deritzo (memoria, memoria, memoria); aldiz, eragigaiak erregistroetan daudenean, **RRR motakoa** deritzo konputagailuari (erregistro, erregistro, erregistro). Ageri denez, helbideratze-modu desberdinak erabili behar dira konputagailu horietan, batean memoria-posizioak eta bestean erregistroak helbideratu behar baitira. Tarteko ereduak ere egon daitezke; bi eragigai erregistroetan eta bat memorian, esaterako.

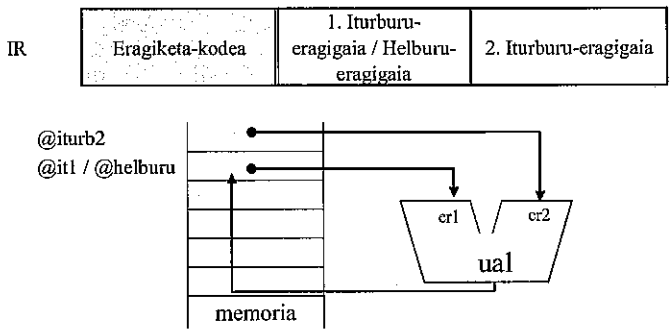
Exekuzio-denborari begira, MMM motako aginduen exekuzioa RRR motakoena baino motelagoa da, memoria atzitu behar delako (eta agian, behin baino gehiagotan, helbideratze-moduaren arabera). Bestalde, MMM motako konputagailuen aginduak luzeagoak dira RRR motakoenak baino, memoria-helbideak adierazteko erregistroak adierazteko baino bit gehiago behar baitira.

2.4.2 Bi eragigai esplizituko konputagailuak

Bi eragigai besterik ez dira adierazten aginduetan. Eragigai batek (lehenengoa zein bigarrena izan daiteke, konputagailuaren arabera) iturburu-eragigai bat eta helburu-eragigaia adierazten ditu aldi

[4] Helburu-eragigaia edozein posiziotan egon daiteke; ohikoa da lehenengoa izatea, baina azkena ere izan daiteke.

berean. Eragigaiak adierazteko, edozein helbideratze-modu erabil daiteke. 2.4.2 irudian ageri da adibide bat.



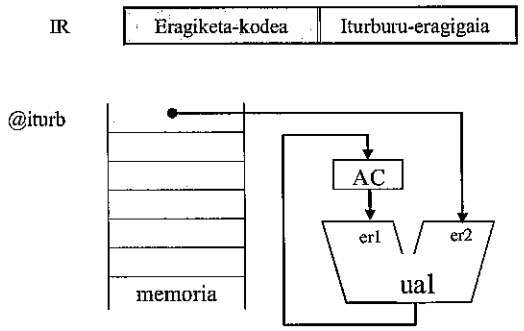
2.4.2 irudia. Bi eragigaiko aginduen formatua eta eragiketa baten exekuzioa (helbideratze-modu absolutua).

Aurreko kasuan bezala, erregistroetan (RR konputagailuak) edo memorian (MM konputagailuak) egon daitezke bi eragigaiak (edo bat memorian eta bestea erregistro batean).

2.4.3 Eragigai esplizitu bakarreko konputagailuak

Eragigai esplizitu bakarreko aginduak exekutatzen dituzte halako konputagailuek. Aginduan bertan adierazten den iturburu-eragigaiaz gain, bigarren eragigai bat erabili ohi da. Inplizitua da eragigai hori, eta beti bera: erregistro berezi bat; **metagailua** (AC, *accumulator*), hain zuzen ere. Metagailua, oro har, unitate aritmetiko/logikoaren sarrera bati lotuta dago. Beraz, eragiketa bat egin nahi denean, lehenengo eragigaia metagailutik hartuko da (hor kargatu dugu aurretik), eta, bigarrena, berriz, agindutik. Azkenik, emaitza metagailuan gordeko da. Adibidez, honako hau egingo du `add X` aginduak: `AC := AC + X`

Konputagailu hauei **metagailuzko makinak** deritze. 2.4.3 irudian ageri da adibide bat.



2.4.3 irudia. Eragigai bakarreko aginduen formatua eta eragiketa baten exekuzioa (helbideratze-modu absolutua).

Mota horretako aginduak azkarrak eta laburrak dira. Erregistroekin gertatzen den modu berean, agindu bat behar da metagailua kargatzeko eta beste bat haren edukia memoriara eramateko; esaterako:

- `load X` `AC := M[X]`

  X aldagaiaren balioa kargatzen da metagailuan.

- `store Y` `M[Y] := AC`

  Metagailuaren edukia memoriara eramaten da.

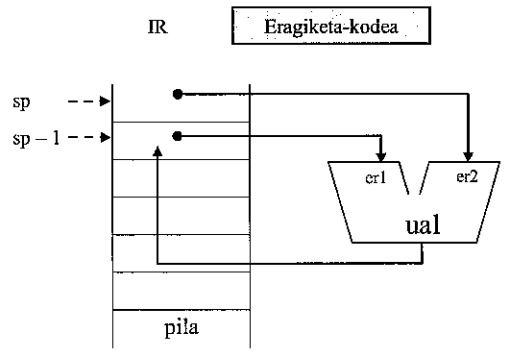
Konputagailu horietan, gerta daiteke programak luzexeagoak izatea, eta, beharbada, exekuzio-denbora ere.

2.4.4 Eragigai esplizitu gabeko konputagailuak

Makina horietan, agindu aritmetikoek ez dute eragigai espliziturik: informazio guztia eragiketa-kodean kodetu da. Agindu horien formatua, beraz, oso sinplea da, 2.4.4 irudikoa bezalakoa.

Kalkuluak egiteko, konputagailuak **pila** izeneko datu-egitura erabiltzen du, eta, horregatik, **pilazko makina** deritzo. Datu-biltegi berezi bat besterik ez da pila, eskuarki memoriako zati bat. Datuak pilan uzten dira eta pilatik hartzen dira, baina biltegi hori modu berezian kudeatzen da, LIFO *(Last In First Out)* moduan: gorde den azkeneko datua da aurrena ateratzen.

2.6.5 irudian ikusten den moduan, pilako tontorreko (azkeneko) datuekin lan egiten da: hortik hartzen dira eragigaiak, eta hor utziko da emaitza. Pilan gorde diren azkeneko bi datuak, hau da, tontorrekoak, dira agindu horien eragigai inplizituak. Eragiketa bat egiten denean, bi datu horiek desagertzen dira pilatik, eta, haien tokian —tontorrean, alegia—, eragiketaren emaitza gordetzen da. Erregistro berezi bat erabiltzen da pilaren tontorra non dagoen adierazteko: SPa, **pilaren erakuslea** (*Stack Pointer*). Pilaren egoera aldatzen den bakoitzean, datu bat sartu edo atera delako, SParen balioa ere aldatu behar da, pilaren tontor berria erakuts dezan.



2.4.4 irudia. Eragigairik gabeko aginduen formatua eta eragiketa baten exekuzioa.

Bi agindu berezi erabiltzen dira datuak pilan sartzeko, edota pilatik eskuratzeko:

- `push X`

  ```
  SP := SP + 1
  M[SP] := M[X]
  ```

  Agindu horrek `X` aldagaia pilaren tontorrean kargatzen du. Horretarako, aurretik, pilaren erakuslea eguneratzen du (kasu honetan, 1eko bat gehituz), kargatu behar den daturako tokia egiteko pilan. Gero, `X` aldagaiaren edukia pilan sartzen du, SP erakusleak adierazitako posizio berrian; tontorrean, hain zuzen ere.

  Ageri denez, SP erregistro bidezko zeharkako helbideratze-modua erabiltzen da pila atzitzeko.

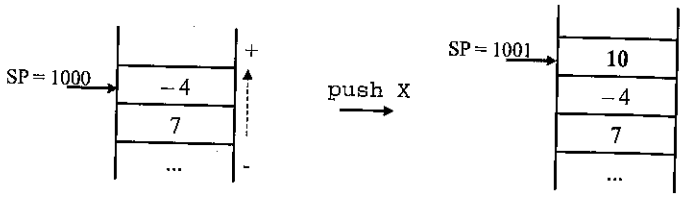
- `pop Y`

  ```
  M[Y] := M[SP]
  SP := SP - 1
  ```

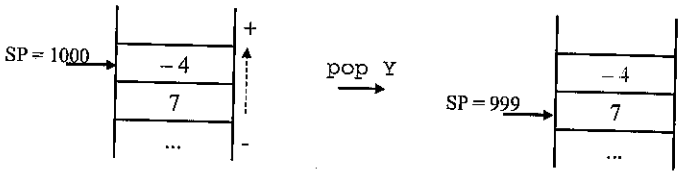
  Pilaren tontorreko datua `Y` aldagaian gordetzen du. Gero, SParen balioa eguneratzen du, tontor berria erakusteko, pilatik datu bat kendu da eta.

Aurreko definizioetan, helbide jakin batetik aurrera hazten da pila, eta, horregatik, datu bat pilan sartu baino lehen, SP erregistroaren edukia gehitu behar da. Kontrakoa ere guztiz zilegi da, hots, helbide batetik atzera joatea pila osatzeko.

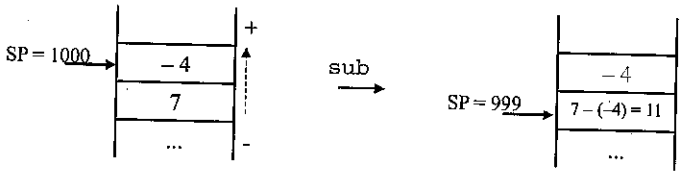
2.4.5 irudian ageri da `push`, `pop`, eta `sub` aginduen eragina pilaren gainean.



(a) pilaren egoera `push X` agindua exekutatu ondoren (X-ren hasierako balioa 10 da).



(b) pilaren egoera `pop Y` agindua exekutatu ondoren (bukaeran, Y aldagaiaren balioa –4 izango da).



(c) pilaren egoera `sub` agindua exekutatu ondoren; bi eragigaiak pilako tontorretik hartu dira, eta emaitza pilan utzi da.

2.4.5 irudia. Pilaren erabileraren adibide batzuk.

Pilazko konputagailuen aginduak, oro har, oso laburrak dira, ez baitira eragigaiak adierazi behar; gainera, oso azkar exekutatzen dira, eragigaien posizioak aurretik zehaztuta baitaude. Konputagailu mota horietarako programetan, **idazkera poloniar alderantzizkoa** erabili ohi da formulak adierazteko. Idazkera horretan, eragilea eragigaien atzetik idazten da (eskuarki erabiltzen dugun notazioan, berriz, eragilea eragigaien artean idazten da). 2.4.1 taulan, idazkera poloniar alderantzizkoaren adibide batzuk ageri dira. Bi abantaila ditu idazkera horrek: batetik, ez da parentesirik erabili behar; eta, bestetik, ez dira eragigaien arteko lehentasunak definitu behar: ezkerrean dagoen eragilea izango da aurrena exekutatzen.

Ohiko idazkera	Idazkera poloniar alderantzizkoa
$A+B$	$AB+$
$(A+B)/C$	$AB+C/$
$A+B/C$	$ABC/+$
$(B+C)/((B-C)\times D)$	$BC+BC-D\times/$
$A/(B+C/(D-A))$	$ABCDA-/+/$
$(A\times B\times C+A/B)/(A+B+C)$	$AB\times C\times AB/+AB+C+/$

2.4.1 taula. Eragiketa batzuk ohiko idazkeran eta poloniar alderantzizkoan idatzita. A, B, C, ... aldagaiak dira.